Institut für Informatik
Technische Universität München

Acceleration of Medical Imaging Algorithms Using Programmable Graphics Hardware

Thomas Schiwietz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Nassir Navab, Ph.D.
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Rüdiger Westermann
	2. Univ.-Prof. Dr. Bernhard Preim, Otto-von-Guericke-Universität Magdeburg

Die Dissertation wurde am 30.01.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.05.2008 angenommen.

To my family and friends

Abstract

This thesis presents acceleration techniques for medical imaging algorithms. The rapid development of medical scanning devices produces huge amounts of raw data. On the one hand, high-resolution images can be computed from the raw data and, thus, providing the physicians better basis for diagnosis. On the other hand, the amount of raw data leads to longer processing times. About three years ago, graphics processing units (GPUs) have become programmable and can be used for other tasks than graphics. GPUs are very suitable for medical imaging algorithms – they are fast, relatively cheap, the instruction set is powerful enough for many of algorithms, and with standard APIs easy to program. In this work, medical imaging filtering, reconstruction, segmentation, and registration algorithms are accelerated using GPUs. Averagely a 10-times speedup is achieved compared to optimized SSE3 CPU implementations. Some implementations run 100 times faster than their CPU counterparts. The results are already used successfully in products by Siemens Medical Solutions.

Zusammenfassung

Diese Arbeit behandelt die Beschleunigung von Algorithmen aus der medizinischen Bildverarbeitung. Durch die schnelle Entwicklung bei medizinischen, bildgebenden Aufnahmegeräten wächst auch die Rohdatenmenge. Einerseits lassen sich dadurch immer schärfere Bilder rekonstruieren, die den Ärzten genauere Diagnosen ermöglichen, andererseits steigt mit der Datenmenge auch die Verarbeitungszeit rapide an. Seit ca. 3 Jahren lassen sich spezielle Grafikprozessoren relativ frei programmieren und damit zweckentfremden. Grafikprozessoren bieten sich sehr gut für medizinische Bildverarbeitung an - sie sind schnell, das Instruktionsset reicht aus um eine Vielzahl von Algorithmen von der Grafikhardware ausführen zu lassen und durch mehrere Programmierschnittstellen (APIs) relativ leicht zu programmieren. In dieser Arbeit wird die komplette medizinische Bildverarbeitungs-Pipeline bestehend aus Filterung, Rekonstruktion, Segmentierung, und Registrierung durch Grafikhardwareunterstützung beschleunigt. Im Durchschnitt wird eine Beschleunigung gegenüber einer hochoptimierten CPU-Implementierung von Faktor 10 erreicht. Manche Implementierung laufen sogar 100 mal schneller als ihre CPU Gegenstücke. Die Ergebnisse werden bereits von Siemens Medical Solutions in der Praxis erfolgreich eingesetzt.

Acknowledgements

Prof. Westermann asked me to join his group when I finished my Diploma thesis in 2003 under the supervision of Dr. Jens Krüger about GPU-accelerated fluid simulation. I never thought of doing a PhD so my answer was something like ”Let me think about it.”. After thinking about it for a while Prof. Westermann offered me an interesting position sponsored by Siemens Corporate Research (SCR). Two years working in Princeton, NJ at SCR and another 2 years at the Technische Universität München, Germany. Finally, I accepted the offer. Overall it was an eye-opening experience to learn from both worlds, the corporate research and the academic research. Therefore, my biggest thanks go to my advisors at both institutions: Prof. Dr. Westermann (TUM) and Gianluca Paladini (SCR). Both periods where very interesting, exciting and very different from each other all together. This thesis contains many images from Siemens data sets. I thank Siemens for letting me using them. During the last four years I had the privilege to work with many people from both groups.

- Prof. Westermann's group at TUM: Dr. Jens Krüger, Dr. Joachim Georgii, Jens Schneider
- Gianluca Paladinin's group at SCR: Dr. Shmuel Aharon, Dr. Leo Grady, Dr. Ti-chiun Chang, Dr. Peter Speier, Dr. Supratik Bose, Christoph Vetter, Thomas Möller

Thank you very much for your help and inspiration! Further, I'd like to thank all people who proof-read this text, especially Dr. Martin Kraus and Christian Dick. I certainly didn't regret doing the PhD after all.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	Application Programming Interface
Cg	CC for Graphics
CPU	Central Processing Unit
CT	Computed tomography
CTM	Close-To-Metal (ATI)
CUDA	Compute Unified Device Architecture (NVIDIA)
DFT	Discrete Fourier Transform
FBP	Filtered Backprojection
FFT	Fast Fourier Transform
FOV	Field of View
GLSL	OpenGL Shading Language
GPGPU	General-Purpose GPU (programming)
GPU	Graphics Processing Unit
HLSL	High Level Shader Language
MR	Magnetic Resonance
MRI	Magnetic Resonance Imaging
PET	Positron Emission Tomography
PIV	Particle Image Velocimetry
RAM	Random Access Memory
RGBA	Red/Green/Blue/Alpha
SIMD	Single Instruction, Multiple Data
SPECT	Single Photon Emission Computed Tomography
SSE	Streaming SIMD Extensions

Chapter 1

Introduction

The field of medical imaging is advancing rapidly as both hardware and software are undergoing steady improvements on all fronts. Devices such as CT (computed tomography) and MR (magnetic resonance) scanners are being improved by reducing the amount of radiation, reducing the time required for scanning a patient, or improving the scanner's sampling resolution to visualize finer details of the human body. However, the growing amount of raw data provided by today's scanners often exceeds the amount of RAM (random access memory) in desktop or even server computers; and with more data, the processing time increases, too. Thus, great challenges are to deal with high memory consumption and algorithm acceleration. Also, new medical imaging algorithms try to provide better interpretations of the acquired data in the areas of image filtering, reconstruction, segmentation, registration, and visualization. With the help of improved scanning devices, the challenge is to develop algorithms and efficient implementations to compute optimal medical images in terms of sharpness, signal-to-noise ratio, and highest resolution in a very short amount of processing time providing the physicians with a better basis for their diagnosis and, thus, faster and more effective ways to cure patients.

This thesis addresses the processing time issue of modern medical imaging software by providing a fast tool set of GPU-accelerated algorithms. GPUs (graphics processing units) have become powerful but cheap general-purpose processors capable of executing almost any algorithm. This tool set provides efficient implementations of solvers for systems of linear equations and a large variety of algorithms in all areas of medical imaging. Furthermore, a novel medical image registration algorithm is presented along with acceleration strategies. The results are already used in Siemens medical imaging software products.

1.1 Programmable Graphics Hardware

In recent years, GPUs have become very popular as co-processors since they are *cheap* and *fast* at the same time. The price of a high-end consumer graphics card is about $500, which is similar to the price of high-end CPUs (central processing units). The difference between the two types of processors lies in the architecture and the peak performance. While CPUs are optimized for serial programs, GPUs are optimized for parallel programs. Since many medical imaging algorithms are parallel algorithms, GPUs have clear performance advantages over CPUs in this area.

Two decades ago, expensive dedicated graphics hardware was developed mainly for the movie industries. During the 1980s, SGI was one of the most famous manufacturers of special graphics hardware. In the mid-nineties, consumer graphics hardware was introduced with a feature set designed for 3D games. Since the gaming market had very high growth rates at that time, consumer graphics hardware quickly spread and became a standard component in the PC world. Nowadays, the feature set of graphics hardware is much more general and it has surpassed the old dedicated hardware in many aspects such as price and performance. Since a couple years, the term *GPU* is used to describe the graphics chip to account for its programmable structure in contrast to older generations providing only a set of fixed functions. The programmability is particularly important if GPUs should be used to execute non-graphics algorithms. Modern GPUs are capable of executing a large variety of algorithms outside the graphics area. GPUs have proven to be fast co-processors as shown in successful implementations of algorithms in numerous areas. For example, they have been used as co-processor for sorting [KW05], data bases [GLW+05], fluid dynamics [KW03, Har03], or deformable bodies [GW05b] just to name a few.

In this thesis, we investigate the usage of GPUs in the area of medical imaging. Using modern scanning hardware today's medical images need large amounts of memory on both disc and RAM to be stored. Both the image resolution and the dynamic range increases gradually over the years and scanner generation. And with larger images the processing time of medical imaging algorithms increases, too. Furthermore, the image data is processed in floating-point precision for best quality which drives the memory requirements even further. Since GPUs have a significant higher peak performance than CPUs and provide 32-bit floating-point precision, they are perfect co-processors for the acceleration of *parallel* medical imaging algorithms. Serial algorithms cannot be adapted to run on the GPU easily without modification.

We present a collection of sophisticated GPU-accelerated medical imaging algo-

rithms. We propose to accelerate as many consecutive algorithms as possible in order to minimize the amount of data transfer time to and from main memory. In particular, we discuss GPU acceleration techniques for solvers of systems of linear equations, many GPU-accelerated algorithms from the area of medical image filtering, reconstruction, segmentation, and registration. Furthermore, we present a novel interactive physics-based registration algorithm.

Before we discuss algorithms related to medical imaging, we give an introduction to consumer graphics hardware in Chapter 2. We start with a short history on the impressive performance improvement in the last couple of years. Then, we briefly discuss the most important components of modern GPUs followed by the programming model and the most important APIs (application programmer's interface). An introduction how to use the GPU as a general-purpose co-processor is discussed in detail after that. This is usually referred to as GPGPU (general-purpose GPU) programming. Finally, all commonly used data structures and operators are presented since this is the foundation for each medical imaging algorithm discussed in the rest of this thesis.

Since many medical imaging algorithms rely on the solution of systems of linear equations, we investigate selected solvers in Chapter 3. Starting with common matrix types and structures, data structures are presented to store them compactly. Novel GPU-accelerated implementation strategies of the Jacobi method, the conjugate gradient method, and the multigrid method are discussed in detail.

1.2 The Medical Imaging Pipeline

The focus of this thesis is on GPU-accelerated algorithms for medical imaging. Therefore, we assume that raw data from a scanning device is (at least partially) available in the software system. The transformation of the raw data provided by scanners to meaningful images is called the *medical imaging pipeline* as depicted in Figure 1.1. The pipeline includes steps such as reconstruction, filtering, segmentation, registration, and visualization. That is, the raw data input of a scanning device is transformed and processed to compute a filtered, reconstructed, segmented, and/or registered high-resolution image, which is as meaningful as possible to physicians.

1. *Raw Data*: Raw data is acquired from a scanning device such as a CT, MR, PET, SPECT, or ultrasound scanner. Some types of scanners do not acquire raw data in the image domain. Depending on the modality raw data is acquired in projection domain or in the k-space domain, for example. The process of transforming the

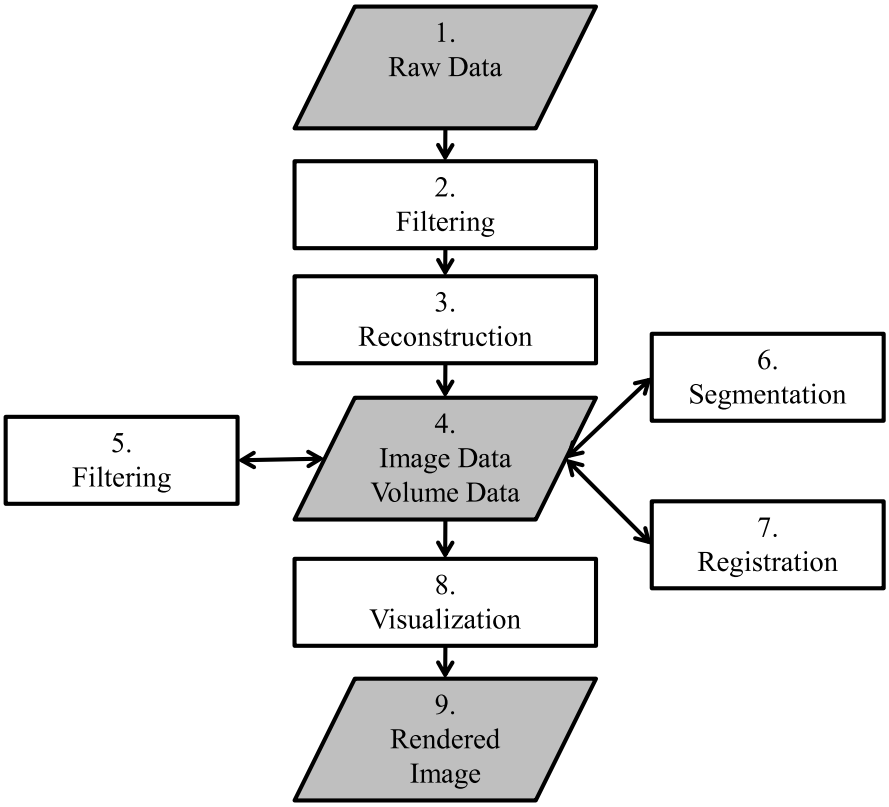


Figure 1.1: *The medical imaging pipeline. Grey boxes represent data while white ones processes. Note that processes 5.–7. alter the image data 4.*

data from the acquisition domain to the image domain is called reconstruction (see step 3).

2. *Filtering*: Typically, raw data needs to be filtered before it can be used to reconstruct an image. The reconstruction theory of many image modalities requires the raw data to be high-pass filtered. Therefore, we present an efficient GPU-accelerated implementation of the fast Fourier transform (FFT). Moreover, the image or volume raw data might contain image artifacts (i.e., errors) arising from the scanning procedure (e.g., calibration errors), from patient motion, metal inlays, or other sources. Some of these artifacts can be removed algorithmically. We mainly focus on the calibration errors. Three non-linear filter are discussed to reduce noise, ring artifacts and cupping artifacts with GPU implementations. If these artifacts are removed before the reconstruction process, the resulting image quality is improved significantly. We describe all the details in Chapter 4.

3. *Reconstruction*: The reconstruction process transforms raw data from the acqui-

sition domain to the image domain. Also, reconstruction sometimes refers to computing a higher-dimensional image from a set of lower-dimensional images. The reconstruction process does not necessarily need a complete set of raw data since incremental updates from the scanner can be processed immediately without waiting for the data set to complete. If the reconstruction process is fast enough to reconstruct a partial image before the next partial raw data set arrives, the data acquisition and the reconstruction process can run simultaneously. However, the memory requirements are huge since the projection data and the reconstruction volume must be kept in memory. We present memory swapping strategies for the GPU-accelerated CT reconstruction process as both the raw data and the reconstructed volume might be too big to fit into memory. Besides CT reconstruction we discuss MR reconstruction from radial measurement lines in detail. Two reconstruction algorithms called *filtered backprojection* and *gridding* with GPU implementation are presented. Both the CT and the MR reconstruction algorithms and GPU acceleration can be found in Chapter 5.

4. *Image Data / Volume Data*: The reconstructed 2D image or 3D volume defines one intensity value at each Cartesian grid coordinate. Typical volume sizes depend on the scanning modality. For example, a CT volume can have 2048^3 voxels. Also, the dynamic range varies very much depending on the scanning modality. A 12-bit dynamic range $[0-4095]$ is very common for CT detectors. Note that other modalities than CT have other interpretations of the intensity values. For high-quality medical image processing the volume data must be stored in floating-point precision to allow algorithms to perform precisely.

5. *Filtering*: In Chapter 4 we also describe filters that are applicable to image data. Obviously, some of the raw data filters can also be applied to image data e.g., the FFT or noise filtering. We discuss the *curvature smoothing* algorithm and its GPU implementation. Curvature smoothing belongs to the class of edge-preserving noise reduction filtering algorithms. Filtering not only improves the visual quality for humans but is also used as preprocess for other algorithms such as segmentation and registration. A smoothed image helps segmentation algorithms to find boundaries more easily. In a similar fashion this is true for registration algorithms, too.

6. *Segmentation*: Usually, a medical image contains more information than a viewer is interested in. Segmentation classifies tissue and material types within a volume. The term segmentation also denotes each classified part of a volume. Segmented

areas can easily be hidden or removed from the volume allowing an unobstructed view. For example, a physician might be interested in the vascular structure of a human body only. Using segmentation the vascular structure is classified and extracted from the rest of the image data. This way, it can be viewed independently. Among many other criteria, segmentation algorithms can be divided into binary and non-binary segmentation algorithms. The former define for each pixel in the image a binary value that specifies whether to include the voxel into the segmentation (or not). Contrarily, non-binary segmentation algorithms define a probability of belonging to the segmentation for each voxel. We discuss a sophisticated general-purpose multi-modal non-binary segmentation algorithm in Chapter 6 including its GPU acceleration.

7. *Registration*: Image registration matches pairs of images. That is, two images taken under different circumstances are matched such that the tissues and shapes overlap each other. Applications are, for instance, comparisons in time, between different image modalities, or pre- and post-operative comparisons. The amount of different approaches is huge and we briefly mention the most important ones in Chapter 7. Furthermore, we present a novel interactive physics-based registration algorithm allowing realistic deformations. Real-world heterogeneous tissue stiffness distributions are taken into account while the simulation runs interactively.

8. *Visualization*: The visualization process displays 2D and 3D images on screen. Nowadays, large volumes, deformable visualization, or importance-driven visualization are active research topics. Since this thesis is about image processing techniques we leave the discussion to other papers [KSW06, BHW+07].

9. *Rendered Image*: An image generated by the visualization process from the image data. The visualization process can also be used as intermediary results for human guided filtering, segmentation, or registration. The final image is usually given to a physician for diagnosis.

Arbitrary complex medical imaging systems can be created by building a processing pipeline built of medical imaging algorithms. For example, the pipeline steps 4–7 may be executed multiple times during one pipeline run. For example, an image is filtered to please the segmentation algorithm and afterwards the segmented image is registered to another image.

1.3 Sponsor-Driven Work

This thesis was sponsored by *Siemens Corporate Research*; thus, only algorithms relevant to Siemens medical products were subject to research. All selected algorithm are not only of theoretical interest but also of practical relevance in products. The condition every algorithm had to fulfill in this thesis is its product-ready stability and reliability. Most of the presented acceleration strategies are already used in Siemens medical products. Therefore, the research conducted for this thesis is not fundamental research but applied research with a strong focus on applicability to real-world data sets instead of phantom data sets.

Siemens is holding the copyrights to all data sets used in this thesis. The data sets were used to create the images shown in this thesis.

1.4 Contribution of this Thesis

This work shows the effectiveness of GPUs in the area of medical imaging. In fact, GPUs can be used efficiently throughout the medical imaging pipeline. Very often, several steps of the pipeline can be executed on the GPU without any transfer to main memory. This provides the fastest performance since additional data transfer significantly slows down the processing. Typically, at least a four-times speed-up compared to a hand-optimized SSE2 CPU implementation is achieved for each algorithm. In some cases we achieve considerably higher speed-ups; e.g., MR reconstruction. One of the most important building blocks is our linear algebra library with implementations of GPU-accelerated conjugate gradient and multigrid solvers. Since several algorithms rely on the solution of a system of linear equations, we benefit in many areas. Finally, our novel registration algorithm is capable of registering two images using physical models. A tissue stiffness distribution on a fine granular level is taken into account allowing only realistic deformations. At the same time our algorithm is interactive and provides immediate feedback to the user.

This thesis is supported by the following peer reviewed publications:

1. *Interactive Model-based Registration*, Thomas Schiwietz, Joachim Georgii, Rüdiger Westermann; in Proceedings of Vision, Modeling and Visualization 2007 [SGW07].

2. *Freeform Image*, Thomas Schiwietz, Joachim Georgii, Rüdiger Westermann; in Proceedings of Pacific Graphics 2007 [TS07].

3. *GPU-accelerated MR Image Reconstruction From Radial Measurement Lines*, Thomas Schiwietz, Ti-chin Chang, Peter Speier, Rüdiger Westermann; ISMRM 2007: Workshop on Non-Cartesian MRI [SCSW07].

4. *A Fast And High-Quality Cone Beam Reconstruction Pipeline Using The GPU*, Thomas Schiwietz, Supratik Bose, Johnathan Maltz, Rüdiger Westermann; in Proceedings of SPIE Medical Imaging 2007 [SBMW07].

5. *MR Image Reconstruction Using The GPU*, Thomas Schiwietz, Ti-chin Chang, Peter Speier, Rüdiger Westermann; in Proceedings of SPIE Medical Imaging 2006 [SCSW06].

6. *Random Walks For Interactive Organ Segmentation In Two And Three Dimensions: Implementation And Validation*, Leo Grady, Thomas Schiwietz, Shmuel Aharon, Rüdiger Westermann; in Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI 2005) [GSAW05b].

7. *Random Walks For Interactive Alpha-Matting*, Leo Grady, Thomas Schiwietz, Shmuel Aharon, Rüdiger Westermann; in Proceedings of VIIP 2005 [GSAW05a].

8. *Numerical Simulations On PC Graphics Hardware*, Jens Krüger, Thomas Schiwietz, Peter Kipfer and Rüdiger Westermann; in Proceedings of EuroPVM/MPI 2004, Special Session Parsim [KSKW04].

9. *GPU-PIV*, Thomas Schiwietz, Rüdiger Westermann; in Proceedings of Vision, Modeling and Visualization 2004 [SW04].

Chapter 2

Programmable Graphics Hardware

The medical imaging pipeline described in Section 1.2 is computationally very expensive since it consists of time and space consuming algorithms. A lot of research has been conducted in the area of GPGPU (general-purpose graphics processing unit) programming over the last couple of years showing its efficiency [OLG+05, KW03]. The GPUs of consumer graphics hardware (programmable graphics hardware) have evolved from triangle rasterizers for 3D graphics to full-fledged programmable processors. Although the instruction set is not as powerful as the one of a CPU, it is sufficient for many types of algorithms. Consumer graphics hardware has become cheap and fast at the same time. Roughly every six months a new generation of GPUs is released outperforming the previous generation considerably. In this section, we give an overview about the history and evolution of consumer graphics hardware, as well as the pipeline data structures, shaders, and APIs. Furthermore, a comprehensive introduction to GPGPU programming is given since all algorithms presented in this thesis are non-visualization ones.

2.1 Evolution of Programmable Graphics Hardware

For a detailed history of programmable graphics hardware we refer the reader to [Krü06]. Instead, we focus on the performance development. A commonly used comparison to the evolution of CPUs is based on Moore's law: Many people say that Moore's law is tripled in graphics hardware. Moore's law is an empirical observation that the number of transistor doubles within 24 months. Sometimes the time frame is quoted as 18 months. However, Moore's law is often misquoted as doubling the *performance* in 18 months. While enormous improvements are made with each generation of hardware, the number of transistors is usually no accurate indicator of the performance. How-

NVIDIA	Year	Memory (MB)	Fill rate (MT/s)
Riva TNT	1998	16	180
Riva TNT 2	1999	32	250
GeForce 256	1999	32	480
GeForce 2	2000	64	1800
GeForce 3	2001	128	1920
GeForce 4	2002	128	2200
GeForce FX	2003	256	3200
GeForce 6800	2004	256	5600
GeForce 7900	2005	512	10320
GeForce 8800 GTX	2006	768	36800

Table 2.1: *NVIDIA chips sorted by release year. The maximum amount of memory in MB and the fill rate in million textured pixels per second is shown. This table is taken from [Wikb].*

ATI	Year	Memory (MB)	Fill rate (MT/s)
Rage 128	1999	32	206
Radeon 7000	2000	64	550
Radeon 8500	2001	128	2200
Radeon 9700	2002	128	2600
Radeon 9800 XT	2003	256	3296
Radeon X800 XT PE	2004	256	8000
Radeon 1900 XTX	2005	512	10400
Radeon HD 2900 XT	2007	512	11888

Table 2.2: *ATI chips sorted by release year. The maximum amount of memory in MB and the fillrate in million textured pixels per second is shown. This table is taken from [Wika].*

ever, huge performance improvements from generation to generation can be observed doubtlessly. Table 2.1 shows the most important GPU chips by NVIDIA sorted by release year. The following columns show the maximum amount of memory and the fill rate measured in million textured pixels per second. Figure 2.1 illustrates the fill rate as a graph. Accordingly, Table 2.2 and Figure 2.2 show the same information for ATI GPUs. All numbers are taken from the websites [Wikb] and [Wika].

2.2 The Shader Model 4.0 Standard

The Shader Model 4.0 standard is the latest incarnation of the Shader Model standards defining all capabilities a chip of graphics hardware has to fulfill in order to call itself compliant to the standard. It is mainly developed by Microsoft and closely coupled to the lower operating system layers of Microsoft Windows Vista. A good overview about the Shader Model 4.0 is described in [Bly06]. We refer the reader to the literature for

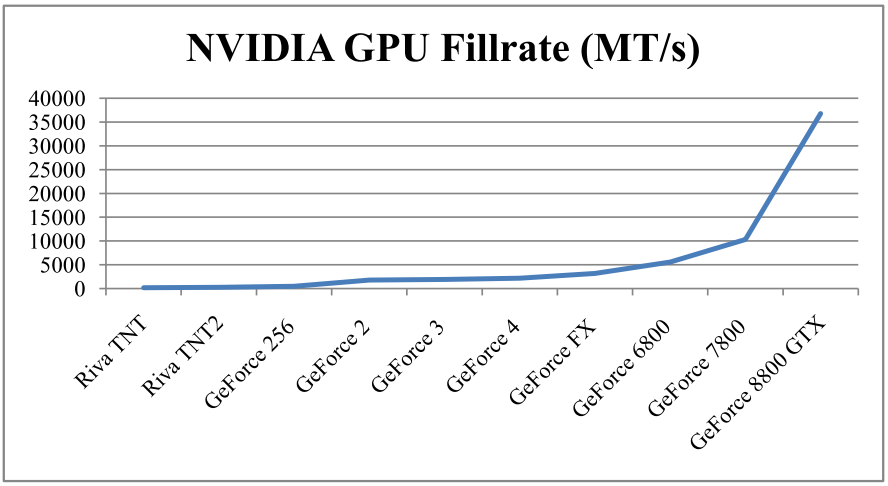


Figure 2.1: *Illustration of the NVIDIA GPU fill rate in million textured pixels per second.*

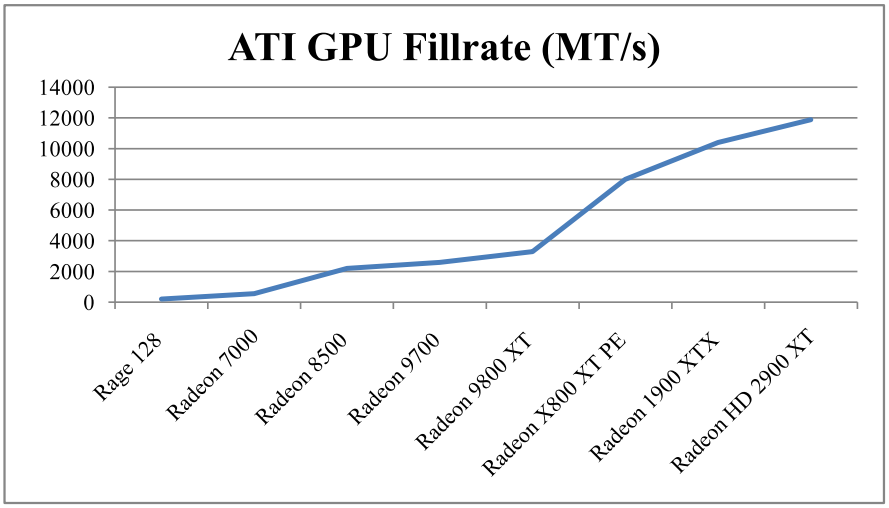


Figure 2.2: *Illustration of the ATI GPU fill rate in million textured pixels per second.*

older (and newer) standards [Krü06]. Basically, the Shader Model standard defines the data structures, the execution processes (the pipeline), and all states and buffers. We explain in the following sections the data structures and the pipeline in detail. Figure 2.3 depicts a graphics board consisting of a GPU - the processor - and onboard memory. The onboard memory is often called GPU memory or texture memory. We call it GPU memory in the following. Next, we describe the data structures available in GPU memory and the execution pipeline, i.e. the shader programs executed by the GPU.

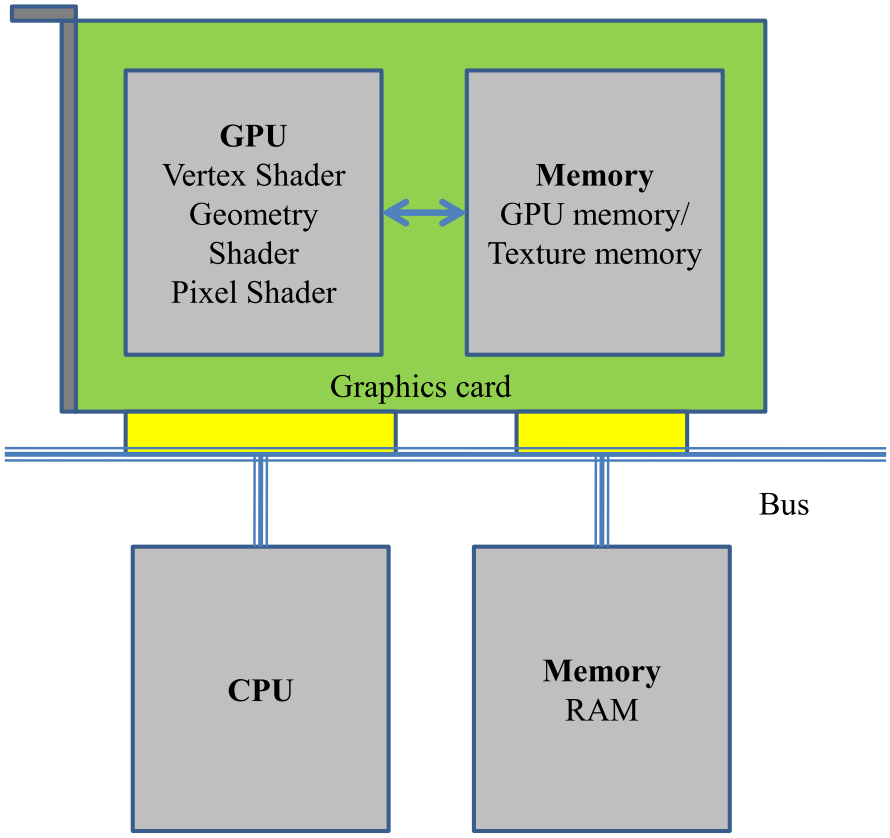


Figure 2.3: *A graphics card has a GPU and onboard memory. It is connected to the PCI bus.*

2.2.1 Data Structures

All GPU data structures are stored in graphics memory. Usually, the graphics memory is onboard with the GPU for fast access, but especially in laptop computers it is often shared with the system memory. There are a limited number of different data types that can be used by the graphics hardware. The most important ones are:

1. *Vertex data*: Most importantly, vertex data defines vectors in 4D space. Additional attributes can be specified such as texture coordinates, normals or colors. In general, vertex attributes are arbitrarily defined by the programmer.

2. *Index data*: An index is a pointer to a vertex where one vertex can be indexed several times. Very often, vertices are shared among multiple primitives. A primitive is a point, a line, or a triangle. With the help of an index buffer, multiple copies of one and the same vertex can be avoided thus reducing the processing time in the vertex shader due to vertex caching.

3. *Texture data*: Textures are 1D, 2D, and 3D arrays of data. Each array element is a tuple of up to four components. Each component is of type char, short, integer or float. Sometimes the tuple is called RGBA as they were invented to store the *red*, *green*, *blue* and *alpha* component of a color. The array elements are often called *texels* (texture elements) or *voxels* (volume elements) in the case of 3D textures. The fastest read and write access provide 2D textures.

Depending on the creation flags, the GPU data structures are CPU readable, CPU writeable, GPU readable, or GPU writeable. All access properties can never be active at the same time. In Shader Model 4.0, a GPU writeable texture is never CPU readable and writeable. If GPU memory needs to be transferred (copied) to main memory a GPU memory internal copy to a CPU readable format must be created first. The access properties are used to optimize performance, especially minimizing the bus transfer times.

2.2.2 Pipeline

The pipeline is a set of stages connected in fixed way as depicted in Figure 2.4. It is not possible to pick a subset of the pipeline but the whole pipeline is executed from the beginning to the end. In the following, the individual steps 1 to 7 are discussed briefly. For an in-depth discussion, we refer the reader to [Bly06].

1. *Input Assembler*: The input assembler stage consists of four setting groups defining vertices and attributes. A vertex is a point in space with additional attributes that are interpolated across the circumscribed primitive. Vertices are described by the *input-layout state*:

   (a) *Vertex layout*: Typical components of a layout are position, texture coordinates, surface normals, and color. The Shader Model 4.0 allows a large number of arbitrary user-defined attributes to be specified. The vertex layout defines the data structure for the components, i.e. the byte position within the structure, the length of the component in bytes, and the data type.

   (b) *Vertex buffer*: The vertex buffer is an array of vertices where each array element has the same vertex layout.

   (c) *Index buffer*: An index points to a vertex in a vertex buffer. A set of indices is called an index buffer. Index buffers allow reusing shared vertices among a sequence of primitives. Consider a box consisting of eight vertices. Each vertex is shared by three triangles. With the help of index buffers, each vertex has to be specified only once. Also, the GPU memory has a vertex cache

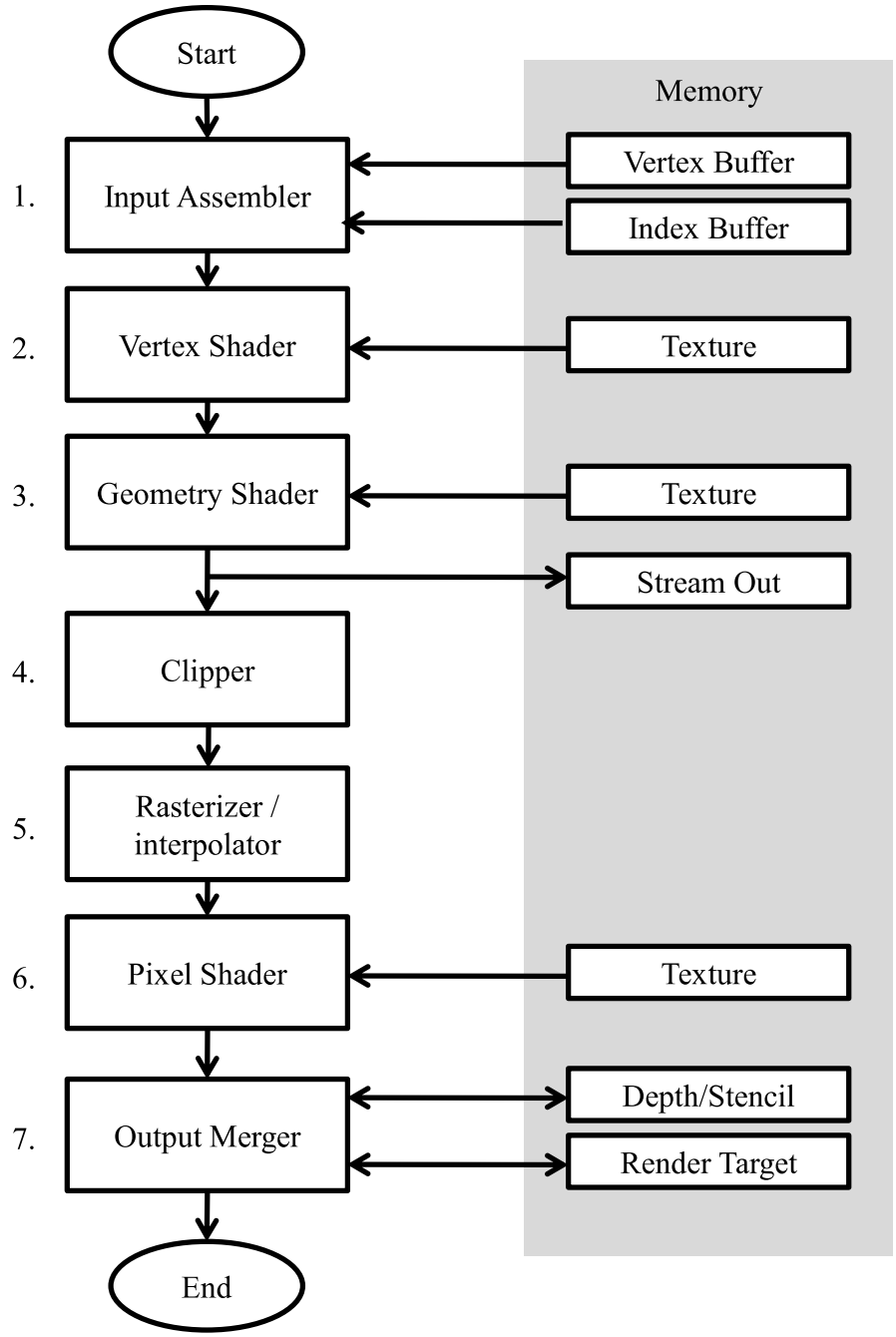


Figure 2.4: *The Shader Model 4.0 graphics pipeline.*

holding previously transformed (processed) vertices. This allows fetching transformed vertices from cache instead of executing the vertex shader several times.

(d) *Primitive topology*: The vertex topology defines how the stream of vertices

and indices are interpreted in terms of the primitive type. Supported primitive types are points, lines, and triangles. Further, the buffers are either interpreted as lists or as strips. While lists describe each primitive separately, strips reuse the last one or two previously indexed vertices in order to save processing time by reusing transformed (cached) vertices.

2. *Vertex Shader*: The (user-defined) vertex shader program is executed for each vertex entering the pipeline. This shader was the first programmable part of the pipeline originally intended to transform coordinate systems from one space to another. Nowadays, the vertex shader is able to perform arbitrary operations on the vertex data structure. Further, 16 textures can be bound and sampled in this stage. For example, geometry information can be stored in the textures allowing the vertex shader program to look up positions.

3. *Geometry Shader*: The (user-defined) geometry shader stage was introduced in the Shader Model 4.0 and is the latest addition to the standard. It operates on the primitive level and has access to the vertex data structures of all primitive vertices. It can output up to 1024 floats in the form of vertices thus every primitive can be outputted several times or not at all. Each copy may be altered, i.e., transformed differently. Furthermore, the primitive type can be changed in this stage, too. For example, a wireframe render can be implemented easily by sending the triangle primitives to the pipeline and the geometry shader outputs lines instead of triangles. A line strip is assembled in the geometry shader from the vertex data. Like the vertex shader, the geometry shader has random access to texture data. Furthermore, the geometry shader is able to *stream out* its results to buffers in GPU memory. This way, subsequent passes are able to use this data as input.

4. *Clipper*: With all vertex positions in projection space, the clipper divides the triangles at the intersection to the frustrum of the semi-unit cube. Clipping leaves the visible part of a primitive and discards the invisible, i.e., the parts lying outside the semi-cube. Once all triangles are clipped, the perspective division is performed by diving the x, y, z-components by the homogeneous component w. Now, with the visible part of the primitives determined, the primitives are rasterized to the target buffer, which can be the screen (frame buffer) or an arbitrary buffer in graphics memory.

5. *Rasterizer / Interpolator*: After clipping and projective division, all vertices have a 2D position on the target raster and a depth value. The rasterizer interpolates

the edges of the primitive from top to bottom and then fills pixels between the edges scan-line by scan-line. The interpolation unit interpolates all attributes in the vertex layout according to the programmer chosen method from a set of interpolation schemes. All interpolated values of the vertex layout are available at each rasterized pixel. The interpolated values are accessible by the subsequent stage, the pixel shader. Note that the rasterizer discards pixels failing the depth test (z-test). A z-buffer stores depth values for all pixel and updates values as soon as a pixel passes the depth test by replacing it with the pixel's depth value. Therefore, the pixel's depth value is also interpolated across the primitive. The z-comparison function among other states can be set in the *depth-stencil state*. Additional states such as cull mode can be set in the *rasterizer state*.

6. *Pixel Shader*: A (user-defined) pixel shader program written by a programmer is executed for each rasterized pixel. It has access to all interpolated vertex attributes. Moreover, it has random read access to all kinds of buffers. Originally, image data was looked-up in the textures. Today, any information can be stored in textures since the introduction of the float-type in GPU memory. The pixel shader determines the result value written to the corresponding pixel in the target buffer. Furthermore, it can also alter its depth value.

7. *Output Merger*: The final stage in the pipeline is responsible for merging multiple output buffers. Sometimes, pixels must not be replaced by the pixels in the target buffer. For transparency effects the new pixels and the old pixels are combined by weighted blending. This stage is not yet programmable, but it is also a candidate for future Shader Models. Today, several predefined states from the *blend state* structure can be chosen.

2.2.3 Shaders

Figure 2.5 illustrates the three programmable stages - the shaders - in the Shader Model 4.0 pipeline with a simple example of a triangle rendered orthogonally to the screen. The vertex shader unit executes a vertex program for each of the three vertices v_0, v_1, v_2. It can transform the position of vertices to another space or change attributes such as texture coordinates. Next, the geometry shader receives all vertices belonging to the primitive pre-processed by the vertex shader. It can transform the vertices again, but more importantly, it may output multiple primitives from one set of vertices, for example shifted in space. It may also change the primitive type, for example from triangle to line. Finally, the pixel shader is executed for each pixel circumscribed by

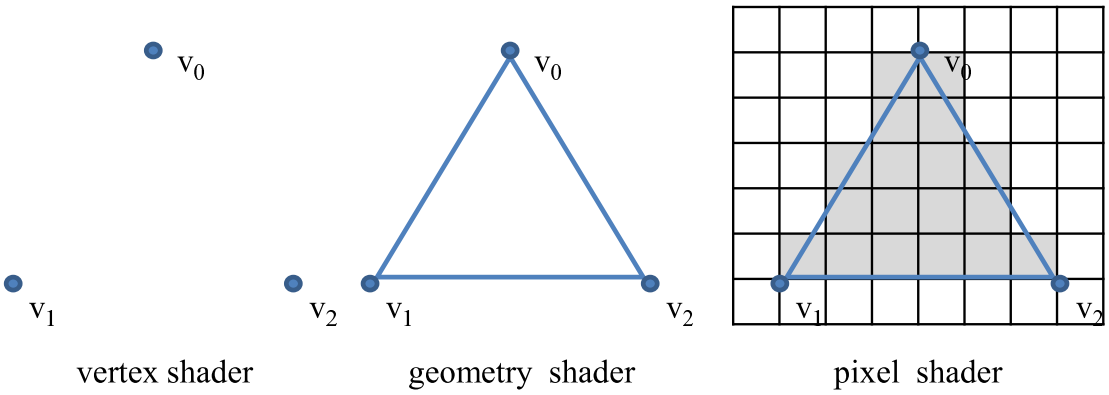


Figure 2.5: *The three programmable stages vertex shader, geometry shader, and pixel shader in the Shader Model 4.0 pipeline. The vertex shader operates on the vertices, the geometry shader on the primitive, and the pixel shader on each rasterized pixel.*

the primitive (indicated as light gray boxed in the right image Figure 2.5) in the target buffer.

Today, the graphics hardware is often an unified shader architecture. This means that a shader unit in hardware is capable of executing vertex, geometry, and pixel shader programs although the input and output structures differ. Incidently, there is a lot of common logic among the three shaders justifying the unified architecture. All shaders have the following common command sets:

- *Arithmetic instructions*: Apart from the primitive arithmetic instruction such as addition, multiplication, etc., graphics hardware has special commands for dot products, cross products and more. Since many 4×4 matrix-vector multiplications are required in computer graphics, a hardware accelerated command is advantageous. A matrix-vector multiplication can be expressed as 4 dot products. All commands are SIMD (single instruction multiple data) instructions performing a single instruction on up to four data components at the same time.

- *Data access instructions*: Data access instructions allow reading values from textures in GPU memory. A sampler state defines the kind of interpolation when fetching a texel, which can be nearest neighbor sampling or linear interpolation (in 2D bilinear, in 3D trilinear). Also, a border mode is specified that determines the sampling behavior outside the texture. Outside fetches can be redirected to the border or to mirror, for example. Finally, the mip-map state defines how to sample different texture resolutions.

- *Control flow instructions*: Control flow instructions allow branching and loops.

Statements such as *if-then-else*, *for*-loops, and *while*-loops can be used in every shader. Branching used to be inefficient, but has become much faster on newer hardware. Still, there are cases requiring *for*-loops to be unrolled. The operating system vendors want to make sure that no shader program can put the graphics card into an endless loop hanging the whole system. That's why Microsoft resets the driver by deleting and restarting it as soon as one GPU call takes more than 2 second. Fortunately, the timeout can be changed.

A large number of temporary registers is available for intermediate results for every program. Each shader is processed independently and potentially in parallel. This is also true for every rasterized pixel. However, no processing order can be assumed by the programmer. Therefore, algorithms dependent on the processing order cannot be executed by GPUs without modification.

2.3 General-Purpose GPU Programming (GPGPU)

Since the year 2003 the field of general purpose GPU programming (GPGPU) exists. The Shader Model 2.0 was implemented in graphics hardware allowing 32-bit floating point textures and random read access to textures. Non-graphics, therefore, general-purpose applications can be implemented since. A GPU accelerated algorithm might result in a multitude of passes each altering different textures. Once the GPU processing is done, the results are read-back to main memory. Thus, the GPUs can be used as co-processors for many algorithms. We briefly review the use of the GPU as a co-processor for arbitrary algorithms; for additional information we refer to the website [GPG].

An algorithm always needs data structures for storage. Therefore, the first question is how to store data in GPU memory. Second, the algorithmic logic must be executed using GPU commands manipulating the data structures in GPU memory. We explain this in the next section in a more detailed way. Finally, the results are read back to main memory for further processing or storage on disc. A nice by-product of GPU processing is that intermediate results can be displayed at virtually no extra cost as all data is already present in GPU memory. This can be used for progressive visualizations and be useful for better understandings of an algorithm.

As already stated earlier, read access to textures is available in all shader stages. Unfortunately, GPU write access is not available in a random access manner. We described the details in the upcoming sections. First, we discuss 2D data structures and 3D data structure with GPU read and write access. Finally, we review the reduction operator allowing to compute sum, find minimums and maximums in textures.

2.3.1 2D Data Structures and GPU write access

As described previously, there are GPU data structures for 1D, 2D, and 3D arrays called *textures*. These textures are used in GPGPU programming to store any data required by the algorithm. Since textures are nothing else but 1D, 2D, or 3D arrays, every data structure of an algorithm must be mappable to an array. Thus, mapping algorithms that use arrays only is a straightforward task. On the other hand, algorithms with non-array data structures have to be adapted to use arrays. As there is no way to store pointers on the GPU, linked lists, trees, graphs and so forth cannot be implemented directly. Sometimes, the pointers can be circumscribed using table indices and therefore represented as textures, but it often requires a redesign of the algorithm. The textures can be filled by values from main memory for the initial setup. Subsequently, the textures must be updated by the GPU in order to execute an algorithm using the GPU. As described above, the pipeline is executed for everything. So it must be used to update textures, too.

To set values in a texture by the GPU, some stages of the GPU pipeline are programmed specially to achieve the texture update. First, the texture to be updated is set as the render target. The render-to-texture functionality was originally intended to implement multiple cameras within a rendered scene.

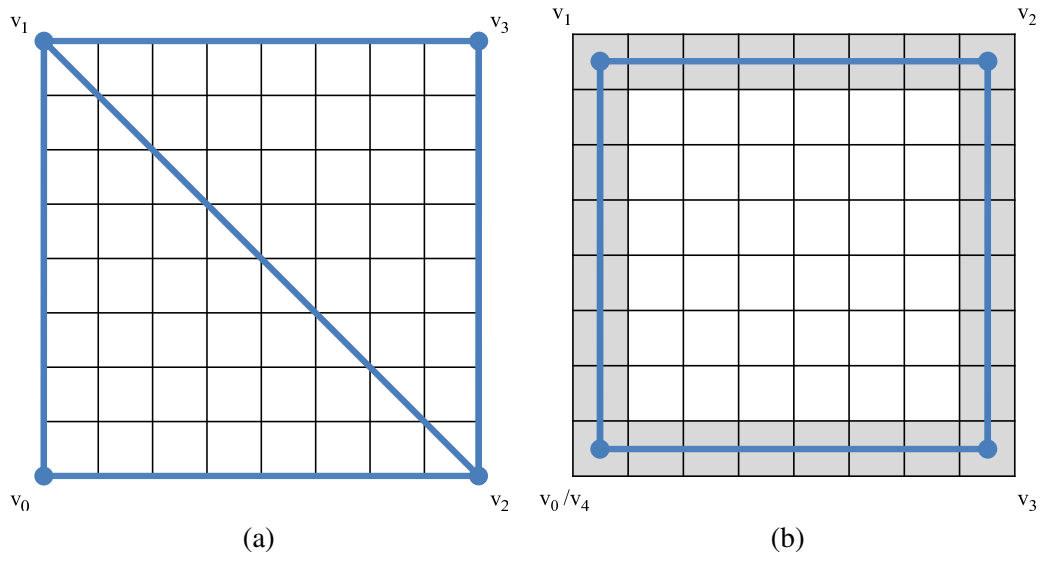


Figure 2.6: *The texture update is illustrated in two examples. A full update using a quadrilateral in image (a), and a partial (border) update using four lines on image (b).*

Using render-to-texture the output of the pipeline can be directed to a texture instead

of the display. Now, the programmer must cover the area of the texture to write to by primitives. For example, if the entire texture needs to be updated, two triangles defined by four vertices (v_0, v_1, v_2, v_3) are rendered orthogonally onto the texture. Orthogonal refers to straight from the top without projective distortion. Figure 2.6 illustrates two examples for updating a different part of a texture. On the left-hand side, a full texture update is shown. Using a triangle strip, two triangles (v_0, v_1, v_2) and (v_2, v_1, v_3) are defined by the vertices (v_0, v_1, v_2, v_3). When the two triangles are rasterized, all texel in the texture are covered and overwritten by the pixel shader result. Another example of a texture update is illustrated in the image on the right-hand side. Especially in numerical algorithms, boundary conditions have to be considered separately from the rest of the domain. In order to update the boundary cells only, a line strip is defined by five vertices located in the center of the corner cells. Note that the order of the vertices is now a round trip starting and ending in one vertex, in contrast to the zigzag-order of the previous example. The line strip results in the following line segments: $(v_0, v_1), (v_1, v_2), (v_2, v_3), (v_4, v_5)$. The pixels closest to the ideal line are rasterized.

If the update geometry cannot be described simply by defining some triangles or lines, multiple geometries can be rasterized in a row or packed into one pass. Another option for arbitrary sparse updates is to use the z-buffer. The z-buffer was created to take care of the correct depth order in rendered 3D scenes. Basically, it stores a normalized depth value for each pixel in the render target. The z-buffer can also be used as a mask in image processing. By setting the z-buffer to 0 where an update is allowed and 1 where the update is not allowed, a z-less test will discard all blocked pixels if the z-plane of the rendered primitive is between 0 and 1. Apart from the z-buffer, another option to build per-pixel masks is to use the stencil buffer. Since the idea remains the same, we will not go into detail here. For each rasterized pixel that survived the z-test, the pixel shader program is executed. By rendering two triangles covering the entire domain those pixels masked by the z-buffer are discarded.

As mentioned above, a pixel shader program is executed for each rasterized pixel. It may perform arbitrary arithmetical operations, access interpolated vertex attribute data, access other textures at random positions for reading, and use control flow instructions. The length of a pixel shader program is not restricted anymore in the Shader Model 4.0 standard. Further, it can also discard pixels during the execution, for example in a branch of a control flow block. So far, we have discussed how to write values to one texture. The Shader Model 4.0 allows writing to 8 textures at the same time with unique output values each. This is useful whenever more data needs to be stored per pixel than four values (RGBA). The GPU memory can be filled up with any number

of textures. Therefore, with the help of multiple textures and multiple passes (pipeline runs), complex algorithms can be built on the GPU to achieve the desired algorithmic goal.

2.3.2 3D Data Structures and GPU write access

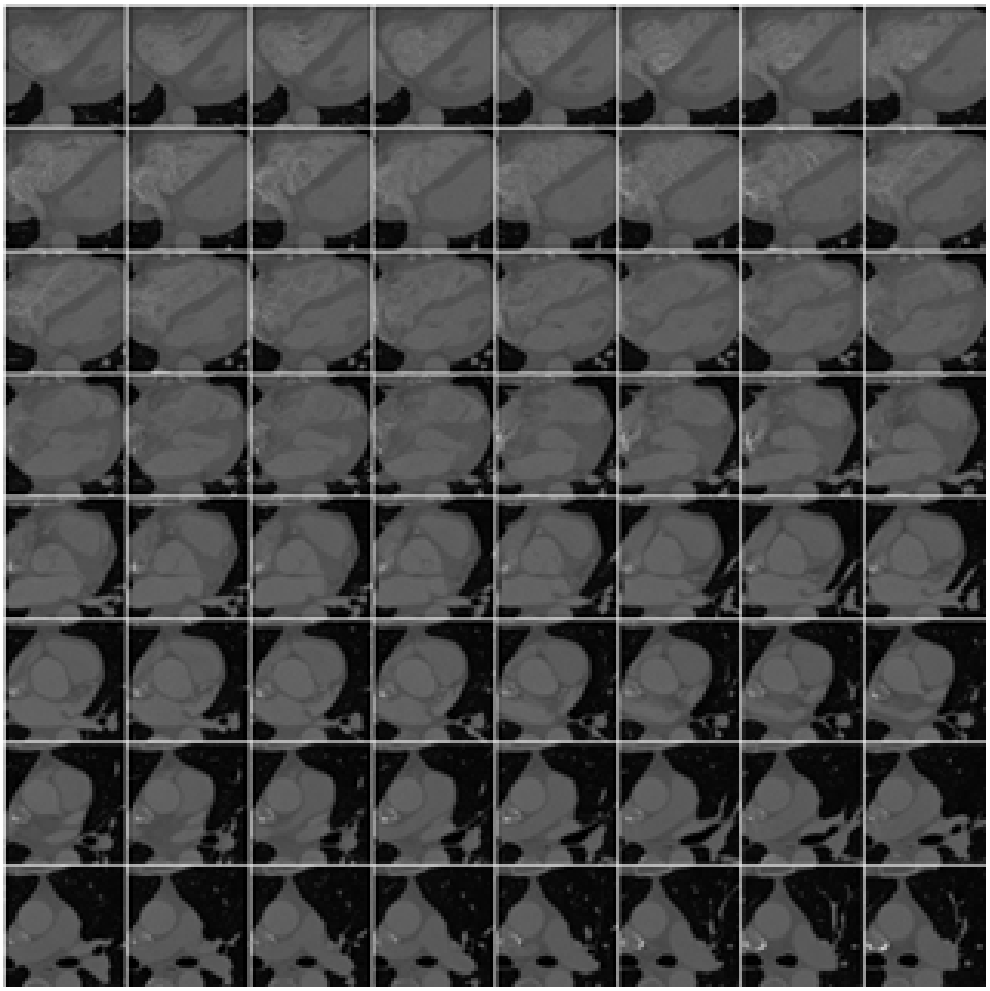


Figure 2.7: *A flat texture of a* $64 \times 64 \times 64$ *cardiac data set. Eight slices are put in one row side-by-side.*

Especially in medical imaging, 3D data structures are required very often since most modern scanner types acquire raw data to provide information of the 3D space. Until recently, a *flat texture* (sometimes called *texture atlas*) had to be used in order to represent GPU-writeable structures. A flat texture is a 2D texture putting 3D slices of a volume next to each other. Figure 2.7 illustrates a flat texture of a 64^3 cardiac data

set. Flat textures can be updated in a single render pass using two triangles covering the entire texture. Also, a single slice can be updated by defining two triangles covering one single slice of the flat texture. Obviously, flat textures have strict size limitations as there is a maximum limit for the size of 2D textures. Currently, the limit in the Shader Model 4.0 graphics hardware is 8192 pixels in every dimension. Table 2.3 shows possible flat texture sizes for various 3D volume sizes. A volume of 512^2 voxels is not representable in a flat texture anymore since the maximum 2D texture size is surpassed.

Volume size	32^3	64^3	128^3	256^3
Flat texture size	256×128	512×512	2048×1024	4096×4096

Table 2.3: *Volume sizes and suggested corresponding flat texture sizes.*

With the latest generation of graphics hardware rendering to 3D textures is possible. Conceptionally, a 3D texture is a stack of 2D textures (slices). That's why the write access is restricted to slices perpendicular to the z-axis (w-axis in texture space). Two triangles have to be defined covering one slice and a geometry shader program specifies the slice to write to. Naturally, all slices of the volume can be updated in a single rendering call by batching all triangles into one vertex buffer and one index buffer.

Volume size	32^3	64^3	128^3	256^3
Simple: 3D texture	ca. 80000	23700	4338	555
Simple: Flat texture	ca. 85000	34360	4560	582
Complex: 3D texture	ca. 60000	11400	2053	254
Complex: Flat texture	ca. 70000	21305	2840	crash

Table 2.4: *Update performance of 3D textures versus flat textures measured in updates (frames) per second. In the* simple *scenario the pixel shader writes a constant in every voxel of the volume. In the* complex *scenario the pixel shader write the result from a dependent texture fetch to all voxels, i.e. the result from one texture fetch is used to address another one. All timings were taken using a GeForce 8800 GTX graphics card.*

We have compared the update speed of various volumes using the flat texture representation and the 3D representation. Table 2.4 shows the results of our comparison measuring a full volume update (all voxels) of different volume sizes using a 3D texture and a flat texture. We have conducted two tests: in the simple test the pixel shader returns just a constant, in the more complex example a dependent texture is computed. Considering the more complicated voxel addressing in flat textures, the performance drawback of 3D textures is surely acceptable, especially for larger volumes. As the simple test shows, the pure write performance is almost identical in both cases. The

performance difference in the complex test can be explained by the texture read instructions. The flat texture test fetches from a 2D texture whereas the 3D texture test from a 3D texture. The difference to the simple test indicates that the performance bottleneck is the 3D texture fetch.

2.3.3 Reductions

Many algorithms require functions that operate on a set of values (array). Typical examples are computing the sum or finding the minimum or maximum value in an array. Another example in linear algebra is the vector dot product. Two vectors are multiplied component-wise followed by computing the sum over all components. Those kind of operator are called *reduction* operators on the GPU. In this section, we discuss value operators and position operators with GPU implementation for single-valued and multi-valued results.

2.3.3.1 Value Operators

Some of the most common operators on N elements $e(i)$ with $i = 0, , N-1$ are the following:

$$\text{sum} = \sum_{i=0}^{N-1} e(i) \tag{2.1}$$

$$\text{avg} = \frac{1}{N}\text{sum}(e) \tag{2.2}$$

$$\text{min} = \min(e) \tag{2.3}$$

$$\text{max} = \max(e) \tag{2.4}$$

Suppose the number of elements is small $N = 4$ or $N = 16$, and the values are stored in a (1-component) texture. A custom shader can be written that fetches all N elements and computes the desired operator on the GPU. We will discuss the GPU implementation for larger N below.

2.3.3.2 Position Operators

Special versions of the min and max operators are often required if not only the minimum or maximum value is of interest, but also the position of them in the array.

$$\min_{pos} = \{i \in [0; N-1] | e(i) = \min(e)\} \quad (2.5)$$

$$\max_{pos} = \{i \in [0; N-1] | e(i) = \max(e)\} \quad (2.6)$$

Again, suppose the number of elements is small $N = 4$ or $N = 16$ and the values are stored in a 1-component texture. A new texture with the same size but 4 components is created and the contents of the original texture copied to the first component. The three remaining channels are used to store the position of each value in texture coordinates. A shader program computes the position operator by performing the min and max operators as described in the previous section but only on the first component of this 4-component texture. The last three components are simply copied together with the min/max in N elements. This way, the position is coupled with the result.

2.3.3.3 Single Reduction

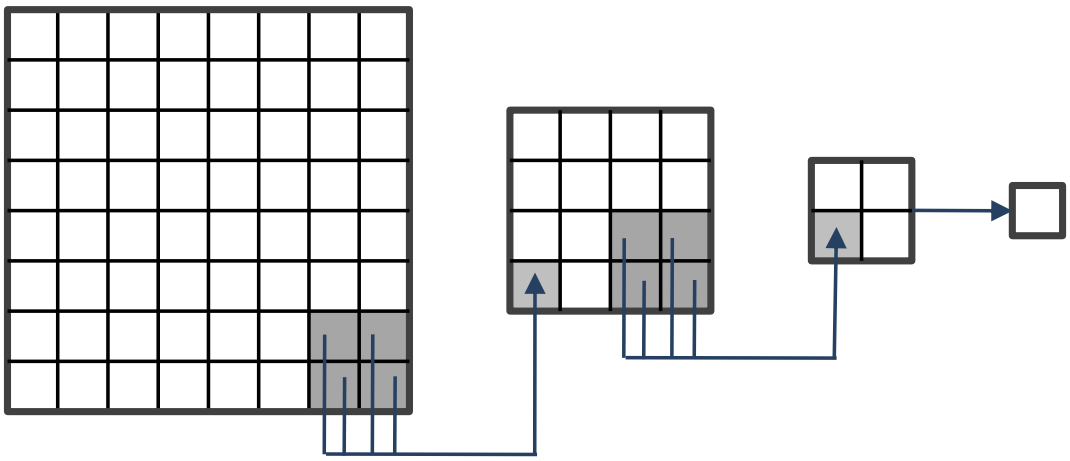


Figure 2.8: *A reduction operation combines a set of elements (here four) to a new element. The process is repeated until there is only one element left.*

The principle how to compute these operators using GPUs is based on the idea of separation. That is, a data set can be split into subsets and the operator is executed on each subset. Next, the results are fed into the operator again yielding another smaller result subset. This process is repeated until there is only one result remaining.

There is a hardware feature called mip-mapping that performs the averaging operator on a texture by generating a sequence of textures each half the size of the previous one in all dimensions. Figure 2.8 illustrates this hierarchy. In the 2D case, four com-

ponents are averaged and written to the next smaller texture. This process is repeated until only a 1×1 texture remains where the overall average is stored. The average operator can be expressed by mip-mapping. Both 2D and 3D textures with all relevant data types are supported by mip-mapping.

The other operators can be implemented using the same idea implemented manually with shader programs providing considerably more flexibility. A hierarchy of textures half the size of the previous one is generated and filled by the reduction operator. The difference is that the operator is now a shader program allowing arbitrary reduction operators to be implemented. Furthermore, the reduction hierarchy can be adapted as well. For example, the hierarchy might consist of textures a quarter the size of the previous one, if 4×4 texture elements are combined in the reduction operator. The number of elements that are combined in the operator is called the kernel size. On older hardware 2×2 is optimal, nowadays longer shader programs are preferable and, therefore, larger kernel sizes must be taken into consideration. Furthermore, non-power of two dimensions can be supported by special shaders for each level.

As illustrated in Figure 2.8 for 2D textures, a set of components is fetched, the desired operator is executed on the set of components, and the result is written to a smaller texture. This process is repeated until only one texture element remains which is the result of the reduction. The same principle can be used for 3D textures as well. In this case, a 3D kernel must be used.

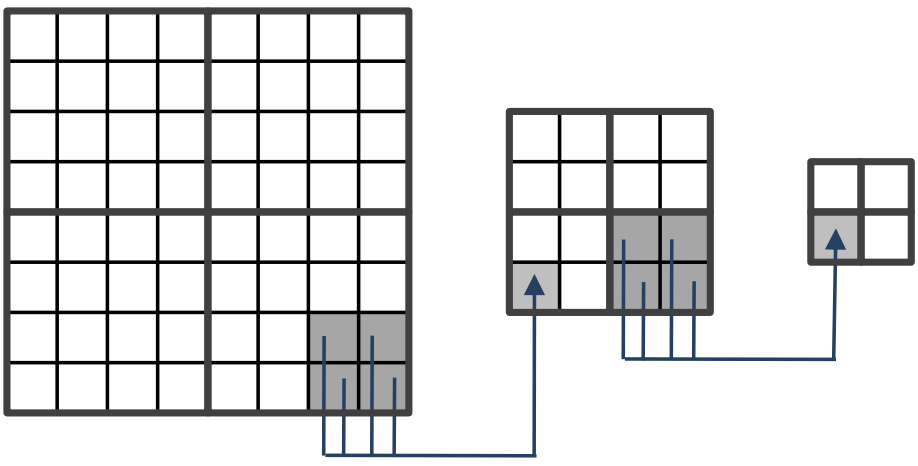


Figure 2.9: *Multiple reduction problems are arranged into one texture. The reduction stops as soon as all original problems have solved.*

2.3.3.4 Multi Reduction

We call a set of similar single reduction problem multi reduction. Instead of computing k equal single reduction operations sequentially, a multi reduction computes all single reductions at the same time. As shown in Figure 2.9, $k = 4$ reduction problems are arranged in a 2×2 grid. The number of reduction steps depends on the size of the single reduction. This way, more work load is pushed to GPU minimizing the amount of CPU-GPU communication. Especially, if there are many single reductions, for example 16×16 single reductions each of size 32×32, the combination of all single reduction into one multi reduction provides huge speedups.

2.4 Graphics programming APIs

Currently, there are two major graphics programming APIs available: DirectX and OpenGL. Both APIs provide similar functionality for graphics hardware programming. Recently, pure GPGPU programming languages have surfaced: CTM and CUDA. Most of our implementations have been written using DirectX 9 and DirectX 10. There a numerous shading languages to program the shading units as each API has its own. Only a few shading languages are available for multiple APIs and operating systems, e.g. Cg by NVIDIA. All shading languages have a close relationship to the C language in common.

2.4.1 OpenGL

Invented by SGI in 1992, OpenGL is the oldest graphics hardware API still alive today. It is available under many operating systems and, thus, portable. OpenGL allows the integration of vendor-specific extensions. Once an extension is accepted by a broader range of developers and vendors, the extension is eventually accepted into the standard ARB extensions (architecture review board). OpenGL grows by extensions rarely replacing existing functionality. This philosophy makes OpenGL backward compatible to the first day. Every OpenGL program compiles and runs without changes as extensions are rarely removed. On the other hand, outdated functionality and extensions bloat the standard producing a lot of legacy code. Further, newer extensions often provide similar functionality but execute much faster. A novice programmer might be overwhelmed with choosing the right extension. OpenGL has its own shading language called GLSL (OpenGL Shading Language). The famous *Red Book* [WDS99] describes the OpenGL API and the *Orange Book* [Ros06] the shading language.

2.4.2 DirectX 9 / DirectX 10

The DirectX library was created by Microsoft in the mid-nineties. Contrarily to OpenGL, the update philosophy of DirectX is fundamentally different. With every major revision, the API is rebuilt from scratch. This has both advantages and disadvantages, too. First, a program written with an older revision cannot use any new features from newer revisions. It has to be rewritten entirely. On the other hand, the API stays very clean as no legacy code has to be maintained. Thus, the programmer is forced to use the latest technology without exceptions. This results in the fastest possible code.

DirectX 9 introduced the HLSL language (high-level shading language) [Gra03]. HLSL was derived from Cg (C for graphics) developed previously by Microsoft and NVIDIA. The feature sets of GLSL and HLSL are equal but there are some differences in the language. HLSL allows defining techniques in order to structure sets of vertex/pixel shader programs into a pass. DirectX 9 allows both fixed function programming and shader programming.

DirectX 10 removes the fixed function pipeline in favor of HLSL [Bly06]. It introduces the geometry shader as new pipeline stage. Moreover, the resource management was redesigned entirely. The different memory pools were discarded, and the concept of a most general buffer was introduced. With appropriate views a buffer can be treated as a texture, a vertex buffer, an index buffer, and so forth. DirectX is implementing the Shader Model 4.0 standard.

2.4.3 CTM / CUDA

As one might think, employing GPUs for non-graphics applications is rather complicated. In-depth understanding of the data structures and the pipeline is required for optimal performance. Additionally, many of the stages provide a lot of functionality that is not needed for GPGPU applications. Therefore, both NVIDIA and ATI have developed APIs to use the GPUs for non-graphics, GPGPU applications. ATI's *CTM* (Close-to-Metal) provides a very low-level access to the graphics hardware while NVIDIA's approach *CUDA* (Compute Unified Device Architecture) is a more abstract high-level API. Also, NVIDIA provides a CUDA library for linear algebra operators and it is relatively easy to program although for optimal performance in depth understanding of the underlying architecture is also required. CUDA supports scattered writing which is a highly desirable feature for GPGPU applications. Unfortunately, the performance of the scattered write is far from optimal. Since both APIs are very young, it is difficult to estimate the acceptance and usage in the community, although CUDA seems to become

more and more popular.

Chapter 3

Solving Systems of Linear Equations Using the GPU

In this section, we discuss systems of linear equations and solution algorithms with GPU implementation strategies. System of linear equations arise from the discretization of partial differential equations in areas such as physics, engineering, or computer science. In this thesis, the random walker algorithm (see Section 6.3), the physically-based deformation method (see Section 7.2), and the optical flow algorithm (see Section 7.3.3) are all based on systems of linear equations. Also, CT and MR reconstruction (see Chapter 5) are further examples though we use different approaches to compute them.

This chapter is organized as follows. First, we give an introduction to the topic and discuss basic solution methods with related work. Since more efficient solvers depend on the structure and properties of the matrix we review those next. Then, we discuss a selection of efficient solution algorithms, namely the Jacobi, Gauss-Seidel, conjugate gradient, and multigrid method. Compact data structures to store matrices and implementations using the CPU and the GPU are presented in the following. Finally, we conclude with our result.

3.1 Introduction

A system of linear equations is a collection of mutually dependent linear equations that need to be satisfied simultaneously. It can be written in matrix-vector form

$$Ax = b, \tag{3.1}$$

where $A \in \mathcal{R}^{n \times n}$ is a matrix, $x \in \mathcal{R}^n$ is the solution vector, and $b \in \mathcal{R}^n$ is a right hand side vector where n is the number of equations. Generally, the system is not necessarily squared but all the applications we discuss in this thesis have squared matrices. Furthermore, n is a power of two most of the time. The matrix A and the right hand side vector b are given, so that Equation 3.1 must be solved for x.

There exist several basic solution methods with an algorithmic complexity of $\mathcal{O}(n^3)$. The Gauss Elimination [Weia] transforms the matrix to upper triangular form with a new right hand side b' by exploiting the linearity of the system. Backwards substitution yields the solution vector x. Another way to compute x is achieved by multiplying Equation 3.1 with the inverse matrix A^{-1} from the left or right on both sides. In order to determine the inverse of matrix A^{-1}, the Gauss-Jordan Elimination [Weib] can be utilized. This method also has computational complexity $\mathcal{O}(n^3)$. Without going into further details, LU-decomposition [Weic] is yet another approach to solve such systems. The matrix A is decomposed into $A = LU$, where L is a lower triangular matrix, and U is an upper triangular matrix. The system is solved using an intermediate solution to the L-system yielding the right hand side for the U system to solve for x. Here, the decomposition of the matrix is the most computational intensive part.

Fortunately, there are more efficient solution methods if the matrix is sparse and satisfies additional properties. Before we go into the details we review related work.

3.1.1 Related Work

The literature on solving systems of linear equations is vast. Good introductions to many solution algorithms can be found in [PTVF92]. We focus on papers discussing GPU-accelerated implementations of standard algorithms.

As the Jacobi method is simple and maps to the GPU in a straightforward way it has been implemented by [Har03] and many others.

Although there exist several parallel implementations of the Gauss-Seidel method such as [KRF94], none can be ported efficiently to the GPU. We discuss the method and the problems inherent with it in detail.

One of the first publications on the solution of sparse systems of linear equations were [KW03] and [BFGS03] using the conjugate gradient method. A detailed introduction to the conjugate gradient method can be found in [She94]. We briefly review the approach and discuss alternative storage methods for the matrix.

The Multigrid method has been discussed in [Bra77, Hac85, BHM00]. In the GPU community it appeared in [BFGS03]. Further work on this approach was conducted in [GST07] showing mixed precision methods.

3.1.2 Matrix Structures

In this section, we discuss different kinds of matrix structures. By structure we refer to the number and location of zero entries in the matrix. In many applications matrices are *sparse* matrices, i.e. many matrix entries are zero. The ratio of non-zero entries to the total number of entries is called the *fill-ratio* $f = m/n^2$, where m is the number of non-zero entries and n^2 is the total number of matrix entries.

Typically, three different kinds of matrix structures are distinguished:

- The *full matrix* has non-zero entries almost everywhere in the matrix. Matrices with a fill-ratio greater than 80% can also be considered as full matrices. Full matrices are commonly stored in a 2D array which allows read and write operations in $\mathcal{O}(1)$. On the down side, a system of linear equations with a full matrix cannot be solved more efficiently than $\mathcal{O}(n^3)$, so iterative methods will not solve the system significantly faster. Fortunately, many real-world problems can be approximated using sparse matrices.

- The *random-sparse matrix* (or *scattered-sparse matrix*) has a low fill-ratio ($< 20\%$) with no particular structure. The non-zero entries are scattered at random positions in the matrix. Commonly, a row-compressed data structure is used to store the matrix. A pair of position and value is stored in a linked list for each non-zero entry. Obviously, this kind of data structure complicates all matrix-vector operators significantly. Many paper discuss efficient data structures and algorithms for random-sparse matrices [Geo07].

- The *band matrix* is another special case of the sparse matrix. All non-zero entries are placed on diagonals of the matrix. We use the term *band* and *diagonal* coequal in the following. In medical imaging algorithms, the system of linear equation often describes a relationship between neighboring pixels of an image, for example the left/right/upper/lower neighborhood. For example, the random walker algorithm (see Section 6.3) and the optical flow algorithm (see Section 7.3.3) result in band matrices. Band matrices are stored space efficiently using 1D arrays for each diagonal. Again, the matrix-matrix and matrix-vector operators need to be adapted to work properly on the 1D array data structure. We discuss the details including a GPU implementation in Section 3.3.

Since most of the problems we deal with result in band matrices we focus on this kind of structure in the rest of this chapter regarding theory and implementation.

3.1.3 Matrix Properties

Apart from the matrix structure, many different matrix properties can be distinguished. Most solution algorithms have several preconditions a matrix has to fulfill in order to work properly. The following list is not complete but sufficient for the algorithms we discuss:

- A matrix A is *symmetric*, if $A^T = A$.
- A matrix A is *diagonally dominant*, if $|a_{ii}| > \sum_{i \neq j} |a_{ij}|, \forall i, j$. The absolute sum all off-diagonals must be smaller than the absolute main diagonal.
- A matrix A is *positive definite*, if $\forall x \in \mathcal{R}^n, x^T Ax > 0$.

3.2 Algorithms

Today, there are many well-known algorithms to solve systems of linear equations with sparse matrices in complexity less than $\mathcal{O}(n^3)$. Most methods rely on an iterative scheme or on a hierarchical approach. In this section, we review and discuss solvers for systems of linear equations, namely the Jacobi method, the Gauss-Seidel method, the conjugate gradient method, and the multigrid method. The selection of the solvers is by no means complete, but it contains the ones we investigated and optimized our implementations most.

Most solution algorithm with a runtime less than $\mathcal{O}(n^3)$ only work efficiently if the matrix is sparse since the matrix-vector product is assumed to run in $\mathcal{O}(n)$. Further, depending on the solution method, additional requirements to the matrix have to be fulfilled. Matrices need to be *positive definite*, *symmetric*, or *diagonally dominant*, for example. Most importantly, the matrices must have full rank otherwise no unique or no solution can found at all.

3.2.1 The Jacobi Method

The Jacobi method is probably one of the most intuitive and simplest method to solve a system of linear equations. Linear equation i is solved for x_i and computed using the other (old) values in the vector $x_j, j \neq i$. The algorithm computes each component of the solution vector independently and then iterates the whole process until convergence. The Jacobi method converges if the matrix is diagonally dominant.

The matrix A is decomposed into

$$A = L + D + U, \tag{3.2}$$

where D is the main diagonal of A, L is the lower triangular matrix, and R is the upper triangular matrix. With Equation 3.2, Equation 3.1 evaluates to

$$Lx + Dx + Ux = b. \tag{3.3}$$

Solving Equation 3.3 for x yields

$$x = D^{-1}(b - (L + U)x). \tag{3.4}$$

The inverse diagonal matrix D^{-1} can easily be determined if each entry is non-zero. By adding an iteration counter k the iterative algorithm is formulated to

$$x^{(k+1)} = D^{-1}(b - (L + U)x^{(k)}). \tag{3.5}$$

The iteration is repeated until the system converges. Two vectors are required to store $x^{(k)}$ and $x^{(k+1)}$ since all values from the last iteration are needed to compute all new entries. This also means that the processing order of the components is not important.

3.2.2 The Gauss-Seidel Method

The Gauss-Seidel method is very similar to the Jacobi method. It also requires the matrix to be diagonally dominant and relies on the same matrix decomposition. By keeping the lower triangular matrix on the left hand side Equation 3.3 is reformulated to

$$(D + L)x^{(k+1)} = b - Ux^{(k)}. \tag{3.6}$$

Exploiting the lower triangular matrix this equation is rewritten using entry indices

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}), i = 1, ..., n. \tag{3.7}$$

The difference to the Jacobi method is that values before entry k have already been updated while values behind entry k are still left untouched. A CPU implementation needs only one vector to store the solution vector because it can overwrite entry k if the entries are processed sequentially from 1 to n.

Unfortunately, the Gauss-Seidel method does not map to GPU structures efficiently.

The GPU is a parallel processor and no guaranteed processing order can be assumed at any time. Furthermore, reading and writing from and to the same texture is not allowed due to the unpredictable processing order. So, there is no direct way to implement the Gauss-Seidel method on the GPU. There are many papers showing parallelization strategies for the Gauss-Seidel method [KRF94], but non of them can be ported to the GPU easily.

3.2.3 The Conjugate Gradient Method

In contrast to the Jacobi or Gauss-Seidel method, the conjugate gradient method requires the matrix to be symmetric and positive definite. Also, each entry of the matrix must be real-numbered. The conjugate gradient method minimizes

$$E(x) = \frac{1}{2}x^T Ax - b^T x \tag{3.8}$$

as the gradient of Equation 3.8 is

$$E'(x) = Ax - b, \tag{3.9}$$

and, therefore, the minimum is a solution to $Ax = b$.

Starting from an arbitrary point $x \in \mathcal{R}^n$ the minimal point defined by Equation 3.8 in $\mathcal{R}^n$ is found by sliding down the n-dimensional surface to the bottom most point. The minimum can be reached by a series of steps along the steepest gradient. From the starting point, the steepest gradient is selected as direction. The direction is traveled until the derivative of the direction becomes zero. From that point the steepest gradient is selected (which might be a gradient that has been selected before), and the iteration starts over. The algorithm stops as soon as the residual becomes small enough.

In order to optimize the steepest gradient algorithm, the traveling direction is chosen so that a direction is never selected twice. That is, for each traveling direction, the optimal distance is traveled. The explanation of this method requires the definition of *conjugate* vectors. Two vectors a and b are *conjugate* to a matrix A if

$$a^T Ab = 0. \tag{3.10}$$

Instead of choosing the steepest gradient as traveling direction, the traveling directions are chosen conjugate to each other. This allows traveling the exact distance along each direction and, therefore, the same direction has never to considered twice. Initially, the residual can be used as the traveling direction.

Before we formulate the algorithm, we briefly describe preconditioners as we use them in all our applications. Preconditioners are used to lower the number of iterations the conjugate gradient loop takes to converge. An approximate inverse matrix M^{-1} of the matrix A is computed in order to solve $M^{-1}Ax = M^{-1}b$. In all our implementations we use the simple *Jacobi preconditioner* that divides all solution components by its corresponding diagonal entry of the matrix. It is implemented on the GPU very easily in contrast to (non-parallel) algorithms such as the incomplete Cholesky factorization.

The preconditioned conjugate gradient algorithm is shown in pseudo code in Algorithm 1. A maximum number of iterations i_{max} and a convergence limit ϵ is specified as termination criteria. Generally, the conjugate gradient algorithm has an algorithmic complexity of $\mathcal{O}(n^2)$.

Algorithm 1 The preconditioned conjugate gradient algorithm

$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$z \Leftarrow M^{-1}r$
$p \Leftarrow z$
$\rho \Leftarrow r^T z$
while $i < i_{max}$ and $\rho_{new} > \epsilon$ **do**
  $q \Leftarrow Ap$
  $\alpha = \frac{\rho}{p^T q}$
  $x \Leftarrow x + \alpha p$
  $r \Leftarrow r - \alpha q$
  $z \Leftarrow M^{-1}r$
  $\rho_{new} \Leftarrow r^T z$
  $\beta = \frac{\rho_{new}}{\rho}$
  $p \Leftarrow z + \beta p$
  $\rho \Leftarrow \rho_{new}$
  $i \Leftarrow i + 1$
end while

3.2.4 The Multigrid Method

Another approach for solving systems of linear equations is provided by the multigrid method [Bra77, Hac85, BHM00]. The idea is to compute a hierarchy of systems describing the original problem reduced in the number of equations in each level. A V-cycle starting from the finest level going to the coarsest level and up to finest level again is the iterative solution function.

The hierarchy approach only makes sense, if a vector can be transferred to a coarser representation without losing information. This condition is true, if the vector does

not contain high frequency components. Therefore, a multigrid approach has to eliminate all frequencies not representable by the next coarser level in order to restrict it without sacrificing information. The high frequency smoothing can be achieved by employing the Gauss-Seidel method which dampens high frequencies before the low frequencies. Depending on the matrix structure, the Jacobi method also can be used for this task. Typically, only a few iterations are performed to reduce the high frequencies even if some are remaining. This is later corrected by multiple V-cycles (see below). The counter direction (inverse reduction) from the coarse representation to the fine representation is simply achieved by interpolation. This is also sometimes called *prolongation*.

Now, we discuss the construction of the matrix hierarchy. The hierarchy has l levels, where l denotes the finest level, i.e. the original system of linear equations. A matrix A_i, where $i \in [1; l]$ denotes the level, is reduced to the next coarser level A_{i-1} using the Galerkin property

$$A_{i-1} = R_i \cdot A_i \cdot R_i^T, \tag{3.11}$$

where $R_i \in \mathcal{R}^{n_i \times n_{i-1}}$ is a restriction matrix. The size of the restriction matrix is determined by the size of the finer level n_i and the coarser level n_{i-1}. While R_i is a restriction matrix, R_i^T functions as an interpolation (prolongation) matrix where values are distributed to nodes on the finer grid. Not only matrices but also vectors can be transferred to coarser or finer levels by multiplication with restriction matrices.

Basically, *geometrically* derived restriction matrices average a neighborhood of the finer matrix such as the following filtering kernels for 2D images

$$\frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{or} \quad \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}. \tag{3.12}$$

In contrast, an *algebraic* multigrid approach computes the restriction matrices from the Eigenvalues of the system matrices. Since we use the Galerkin property to compute the matrix hierarchy, which is an algebraic approach to compute the matrices, our approach is a mixture of the geometric and the algebraic approach.

Algorithm 2 shows the pseudo code for one V-cycle. Note, that all A_c can be precomputed if more than V-cycle is calculated. Typically, the number of V-cycles is rather small in order to compute a converged solution. Generally, the multigrid algorithm has an algorithmic complexity of $\mathcal{O}(n)$. In this recursive representation the multigrid first computes the residual $r = b - Ax$ which is reduced to the next coarser system. Using

Algorithm 2 The multigrid method V-cycle (recursive implementation, l denotes the level).

procedure Solve(A, x, b, l)
  if $l > 0$ **then**
    Relaxation of x
    $r \Leftarrow b - Ax$
    $r_c \Leftarrow R_l \cdot r$
    $e_c \Leftarrow 0$
    $A_c \Leftarrow R_l \cdot A \cdot R_l^T$
    Solve$(A_c, e_c, r_c, l-1)$
    $e_f \Leftarrow R_l^T \cdot e_c$
    $x \Leftarrow x + e_f$
    Relaxation of x
  else
    $x \Leftarrow A^{-1}b$
  end if

the restricted residual the error equation $Ae = r$ is solved. Finally, e is transferred to the finer level and the solution x is corrected accordingly. In the following, we describe the relationship between the original equation and the error equation.

The solution of the error equation helps computing the solution to original problem. Suppose an approximate solution x' has been found for the system

$$Ax' = b, \tag{3.13}$$

then the residual is

$$r' = b - Ax'. \tag{3.14}$$

The correct solution x is derived from the approximate solution x' corrected by e

$$x = x' + e. \tag{3.15}$$

Exploiting the linearity of the system yields

$$A(x' + e) = b. \tag{3.16}$$

In order to compute the correct solution x using the approximate solution x' Equation 3.16 is solved for e

$$\begin{aligned} Ax' + Ae &= b & (3.17)\\ Ae &= b - Ax'. & (3.18) \end{aligned}$$

The right hand side is equal to the residual (Equation 3.14) so that

$$Ae = r'. \tag{3.19}$$

Therefore, solving the system $Ae = r'$ allows correcting the approximate solution x' using Equation 3.15.

3.3 Implementation

In this section, we discuss the CPU and GPU implementation of the solvers described above. All solution algorithms have in common that they are based on data structures to represent matrices and vectors as well as a set of operators such as:

1. Vector addition, subtraction, dot product

2. Matrix-vector and matrix-matrix multiplication

As already stated earlier, we focus on band matrices regarding data structures and operators. In the following, we discuss first vector then matrix data structures and operators.

3.3.1 Vector Data Structures and Operators

In a CPU program the data structure for a vector is a simple 1D array. In a GPU program we store the vectors in 2D or 3D textures (see Section 2.3.1 for more details) as those provide the fastest write performance. Most of our applications are based on regular 2D or 3D grids, so it is straightforward to represent vectors using 2D or 3D textures. Regarding 3D data structures, as shown in Section 2.3.2, a 3D texture provides almost the same write performance as 2D flat textures without having to compute texture coordinate transformations.

Regarding the operators, component-wise vector additions, subtractions, and multiplications in 2D textures are easily implemented by rendering two triangles covering the entire texture (see Section 2.3.1) and using a pixel shader program to operate on the components of the vectors. Likewise, a stack of double triangles compiled to one vertex buffer is used to achieve an update of 3D textures in a single rendering pass. The vector dot product is composed of two separable operators: a component-wise vector multiplication followed by a sum-reduction (see Section 2.3.3.3).

3.3.2 Matrices

The matrix operators are dependent on the matrix structure and representation. First we discuss two ways to store matrices, then the operators including GPU implementation are presented.

3.3.2.1 Storage

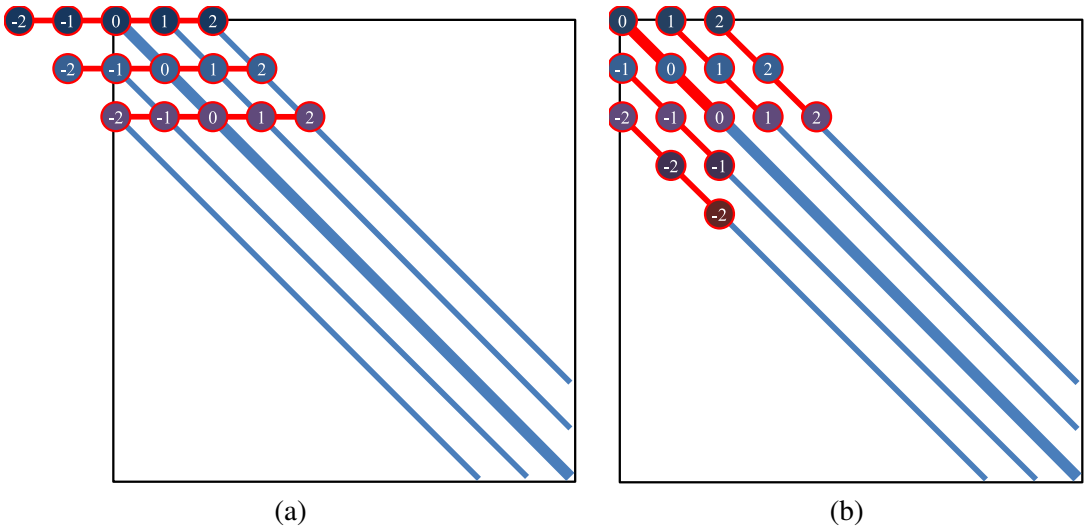


Figure 3.1: *A band matrix with 5 bands can be stored in two ways. One option is to store the band entries line-by-line (a). Another option is to store each band along the band from top left to bottom right (b). The red lines indicate the storage scheme. The colored and numbered circles indicate matrix entries. Equal colors indicates matrix entries belonging to the same line, equal numbers indicate matrix entries belonging to the same diagonal.*

Figure 3.1 illustrates two different ways to store a band matrix. The diagonals can either be stored *line-by-line* as illustrated in the left image or *band-by-band* as shown in the right image. We explain the difference in the following.

The *line-wise* storage of a matrix stores all matrix coefficients of one row in one data set. Note that the band positions are constant in each row so that these position need to be stored only once. Generally, this storage scheme is less flexible if the number of diagonals is not known in advance. We found the line-by-line storage of matrices very well suited for systems operating on a 2D or 3D regular grid since all diagonal coefficients can be retrieved directly from the pixel position. We exploited this storage scheme in our random walker implementation (see Section 6.3). Furthermore, the random walker algorithm is based on a Laplace matrix which allows computing the

main diagonal from the off-diagonals. Therefore, in 2D, storing the four off-diagonal coefficients of the 5 diagonals is sufficient for every row of the matrix (or pixel). Four coefficients are efficiently stored in an RGBA texture. In 3D, the additional diagonals are stored in a second texture.

The *band-wise* storage is probably a more intuitive storage scheme. Each band is stored in an 1D data structure from the top-left to the bottom-right together with an offset describing the signed distance to the main diagonal. We define the main diagonal offset as 0. Bands to the right have increasing positive numbers and bands to the left decreasing negative numbers. Figure 3.1 shows the band numbering in both images. Further, we compute the 2D band starting position (x, y) from the offset where the main diagonal has the position $(0, 0)$, the x-axis increases to the right and the y-axis increases to the bottom.

There are several ways to manage a set of bands in a band-wise stored matrix $A \in \mathcal{R}^{n \times n}$. If the matrix is full it has $2n$ bands. The set can be stored (sorted) in an $\mathcal{O}(n)$ array in order to quickly add or remove bands in $\mathcal{O}(1)$ though much space might be wasted if the matrix has only a few bands. Another option is a linked list which is more memory efficient but adding and deleting bands is $\mathcal{O}(b)$ where b is the number of bands. Addition and deletion of occur during matrix multiplications as described in Section 3.3.2.2. The most elegant solution is to manage the bands in a hash table. This way, even very large matrices with only a few bands do not waste too much memory and band addition and deletion is performed in $\mathcal{O}(1)$.

Compared to the line-wise storage, this structure is much more flexible in terms of band addition and deletion. Especially, if the number of bands is arbitrary the band-wise storage is superior to the line-wise storage.

3.3.2.2 Operators

The matrix-vector product

$$u = Av \tag{3.20}$$

is discussed now assuming the matrix is stored band-wise. In order to exploit the band structure zero entries must be ignored because computing the dot products of all matrix rows and v is no efficient algorithm. Instead, the matrix-vector multiplication is rearranged as a set of band-vector multiplications where each band is processed only once and the matrix entries outside the bands are never accessed. The loop over all

components for all bands is shown in Equation 3.21

$$u[k] = u[k] + b_p[i] \cdot v[j], \tag{3.21}$$

where p denotes band index in the set of bands b, and i, j, k are indices to individual components of each vector. Obviously, depending on the band offset the indices i, j, k must be shifted in order to process the correct components. Fortunately, this is simple using the band starting position (x_p, y_p) of the band p. The j index is shifted by x_p and k by y_p while i remains unshifted. Further improvements are achieved by determining the minimal index ranges for i, j, k in order to eliminate accesses outside the vectors. With this algorithm a matrix-vector multiplication is computed in $\mathcal{O}(|b| \cdot n)$ since all bands $|b|$ are multiplied by n components of the vector v

$$\begin{pmatrix} 0 & A & 0 & 0 \\ 0 & 0 & B & 0 \\ 0 & 0 & 0 & C \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} Ab \\ Bc \\ Cd \\ 0 \end{pmatrix}. \tag{3.22}$$

Equation 3.22 illustrates the band matrix-vector multiplication in a simple example. The matrix consists of 1 band with the entries (A, B, C) at offset 1 and starting position $(1, 0)$. The loop iterates over the three elements of the band applying the shifts 0 and 1 to the vector v and u.

Analogously, a band matrix-matrix multiplication

$$C = A \cdot B \tag{3.23}$$

is computed by multiplying all bands from matrix A with all bands from matrix B. Each band-band multiplication result is a band of the matrix C. Let p, q, r be band indices in the sets of bands in the matrices A, B, C where r results from the multiplication of the bands p, q. The offset r_o (signed distance to the main diagonal) of the resulting band r is computed from the offsets p_o, q_o of the bands p, q with

$$r_o = p_o + q_o. \tag{3.24}$$

Following the compact matrix-vector multiplication from above, we apply the same algorithm to the matrix-matrix multiplication with the inner loop

$$r[k] = r[k] + p[i] \cdot q[j], \tag{3.25}$$

where $[\cdot]$ denotes a component of each band indexed by i, j, k. The indices i, j must be shifted by i_s, j_s to account for the location of the bands. With the band starting positions denoted as $(x_p, y_p), (x_q, y_q), (x_r, y_r)$, the shifts are computed by

$$i_s = y_r - y_p \tag{3.26}$$

$$j_s = x_r - x_q. \tag{3.27}$$

The band matrix-matrix multiplication is computed in $\mathcal{O}(|b_A||b_B| \cdot n)$ where $|b_A|$ and $|b_B|$ are the number of bands in the matrices A, B respectively, and n is the size of the matrix in one dimension.

3.3.2.3 GPU Implementation

In this section, we show how to transfer the data structure presented in Sections 3.3.1 and 3.3.2 to the GPU. Furthermore, we present a novel technique allowing to compute matrix-vector operators with an arbitrary number of bands using the GPU in a single rendering pass.

Similarly to the GPU storage of vectors, we store the bands of a matrix also in 2D or 3D textures. With RGBA channels four bands can be stored in each texture.

One way to implement the matrix-vector operator on the GPU is to write a shader that multiplies one specific band b_p and the vector v at a time accumulating to the result vector u. This shader is executed for each band in the matrix. Nowadays, it is much more efficient to push as much work as possible to the GPU instead of executing a short shader multiple times.

A huge performance gain is achieved by writing a shader that multiplies all bands with the vector *in a single pass*. Unfortunately, writing a custom shader for each matrix structure used by an application is tedious work since a new shader must be written as soon as the number of position of the bands changes.

Especially in the multigrid method a multitude of matrices arises from the hierarchy and it would be very tedious work to write shaders for each matrix-vector multiplication taking the number and location of all bands into account. Instead, we propose a novel technique to automate this process. We propose a system that generates a custom shader for each band matrix so that the GPU matrix-vector operator is created automatically as soon as the matrix construction has been finished.

Basically, we create a string with the proper matrix-vector multiplication code. Using small string building blocks a whole shader is assembled from the structure of the matrix. The shader takes all bands and all offsets and shifts into account. Therefore,

all band textures containing 4 bands each are fetched as well as the vector texture at the respective positions. Since all shifts are stored as 1D offset, the 2D texture coordinate is linearized to 1D, then shifted by the offset, and finally transformed to 2D again. The clamp to border addressing mode sets all texture fetches outside the texture to 0. Finally, all products are summed and written to the target vector texture.

The shader generator is not limited to matrix-vector multiplications. We have also created generators for the Jacobi method and the calculation of the residual. Of course, the conjugate gradient method can be composed with the matrix-vector operator as well, but we see greater benefit for the multigrid method.

Generally, restriction matrices as they appear in the multigrid method cannot be stored using the band matrix structure without wasting much space due to their scattered sparse structure. Using a line-wise representation matrix-vector products can be implemented efficiently anyway. Computing the Galerkin property (see Equation 3.11) is much more difficult. Even an efficient CPU implementation is a challenging task. Instead of computing a full product taking every matrix entry into account, the sparsity of both the restriction matrix and the middle matrix must be exploited, that is, jumping from non-zero entry to non-zero entry. Furthermore, the positions of the non-zero entries often mismatch when computing the dot products so that a good deal of the component multiplications can be spared entirely. But this is only possible, if the matrix structures are analyzed thoroughly in an automated processing step. The output is a stream of matrix indices to multiply during the product. And this is what makes a GPU implementation difficult since varying streams length are not trivially unpackable using a GPU. Therefore, we decided to compute the Galerkin property on the CPU following [Geo07] and leave the GPU implementation to future work.

3.4 Results

Our CPU and GPU matrix operators are very general in the type of band matrices. We have implemented a library containing CPU implementations of the Jacobi, Gauss-Seidel, conjugate gradient, and the multigrid method. On the GPU we have implemented the Jacobi, the conjugate gradient, and the multigrid method. Although it depends on the kind of system we are looking at, the GPU conjugate gradient and the CPU multigrid are among the fastest. We present timings in the section with the applications of this tool box.

Chapter 4

Medical Image Filtering

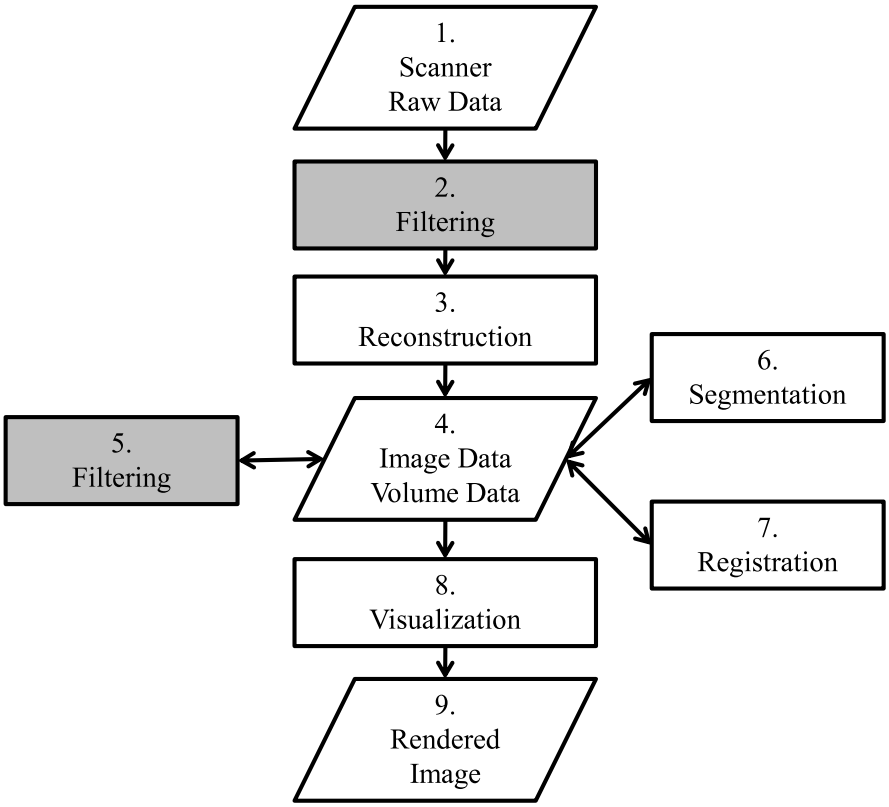


Figure 4.1: *Both raw data and image data are filtered for reasons such as noise removal.*

Image filtering is a very important technique primarily used to reduce noise, but also to sharpen an image, to enhance the edges of an image, or generally to increase or decrease certain structures in an image. In the medical imaging pipeline filtering is used on raw data as well as on image data (see steps 2 and 5 in Figure 4.1). Filtering

is required as a part of most reconstruction algorithms. Similarly, filtering is often used on images and volumes as a pre-process to segmentation and registration algorithms. Many of these algorithms produce better results if the input image only contains a low level of noise. Furthermore, some segmentation and registration algorithms work on the edges of images extracted by edge filters. Commonly, there are two classes of filtering algorithms used in medical imaging: linear and non-linear ones. We discuss selected algorithms for both types of filters regarding theory and GPU implementation. First, we describe linear filtering in spatial and frequency domain with an efficient GPU-accelerated implementation of the fast Fourier transform (FFT). After that, non-linear filters for noise reduction and filters for ring and cupping artifact removal in the CT reconstruction process are presented.

4.1 Linear Filtering

Linear image filtering is very popular because it is effective, easy to implement, and provides satisfying results for a number of applications. Basically, a filter kernel k is convolved with an input image p_i resulting in an output image p_o, where $N \times N$ is the number of pixels in the images and $M \times M$ is the filter kernel size where M is and odd number. The larger the filter kernel size the greater is the global effect of the filter. Typical filter types are softening, sharpening, or edge detection filters. However, the filter kernels are principally independent of the image, that is, linear filters are not controlled by the image data.

There are two ways to apply linear filtering to an image: the spatial domain and the frequency domain approach. We discuss both approaches in the following.

4.1.1 Spatial Domain Filtering

The spatial domain convolution is expressed by

$$p_o(x, y) = \sum_{j=-s}^{s} \sum_{i=-s}^{s} p_i(x + i, y + j) \cdot k(i, j), \tag{4.1}$$

where $s = \lfloor M/2 \rfloor$ specifying the support of the filter. Since the computational complexity of the spatial domain convolution is $\mathcal{O}(N^2 \cdot M^2)$, the filter kernel sizes are usually kept rather small. Table 4.1 shows common 3×3 filter kernels.

Using the GPU for spatial domain image filtering was one of first applications in the field of GPGPU programming. An image is uploaded as a texture to GPU memory.

$$\frac{1}{6}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

sharpen blur edge detect

Table 4.1: *Three different 3x3 filter kernels are presented: sharpen, blur, and edge detect. The filter kernels describe the weighting of an area of samples.*

For best efficiency, the kernel coefficients should be uploaded to the constant registers of the GPU. Alternatively, a kernel texture could be created. To store the filtered image another texture with the same size as the image is needed to perform the filter operation. Two triangles covering the target image are rendered as described in Section 2.3.1. For each pixel in the output image a pixel shader program fetches all samples from the input image covered by the filter kernel.

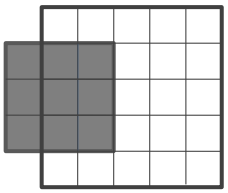


Figure 4.2: *A 3x3 filter kernel (blue) on a 5x5 image.*

Pixels outside the image might be fetched as depicted in Figure 4.2. In that case, pixels from the nearest border (*clamp*) or a predefined default value 0 is used (*border color*). The amount of texture fetches can be very high, especially when convolving a 3D kernel on a volume $\mathcal{O}(N^3M^3)$. Some types of filters are separable such as the Gauss kernel filter. Those kind of filters are applied in each dimension separately minimizing the complexity for d dimensions to $\mathcal{O}(N^dM)$.

4.1.2 Frequency Domain Filtering

An alternative to the spatial domain convolution is the frequency domain convolution

$$p_o = F^{-1}(F(p_i) \cdot F(k)), \tag{4.2}$$

where F denotes the Fourier transform, and F^{-1} the inverse Fourier transform, respectively. Here, the filtering process is reduced to a component-wise multiplication in the frequency domain if both the input image p_i and the filter kernel k have been transformed to frequency domain previously. This approach is efficiently computed in $\mathcal{O}(N^2 \log N)$ for 2D images if the fast Fourier transform (FFT) is used. In the follow-

ing, we review the FFT and an efficient GPU implementation.

4.1.2.1 Fast Fourier Transform

In this section, we review the derivation of the FFT from the discrete Fourier transform (DFT) as well as an efficient GPU implementation. A performance comparison of our implementation with the FFTW library [FJ98] is presented afterwards.

The discrete Fourier Transform $F(n)$ of a 1D discrete signal $f(k),\ k = 0, \cdots, N-1$ with N samples is expressed as a superposition of complex sinusoids

$$F(n) = \sum_{k=0}^{N-1} f(k) e^{\frac{-i2\pi kn}{N}}, \tag{4.3}$$

where $2\pi n/N,\ n = 0, \cdots, N-1$ are the sampled frequencies in the Fourier plane.

Direct computation of the DFT has the computational complexity $\mathcal{O}(N^2)$. The FFT reduces the complexity of the DFT to $\mathcal{O}(N \log N)$. As described by Brigham [Bri88] Equation 4.3 can be written as a matrix-vector product

$$\begin{pmatrix} F(0) \\ F(1) \\ F(2) \\ \vdots \\ F(N-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2N-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2N-2} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(N-1) \end{pmatrix} \tag{4.4}$$

where

$$\omega^k = e^{\frac{i2\pi k}{N}} = \cos\left(\frac{2\pi k}{N}\right) + i \sin\left(\frac{2\pi k}{N}\right); \tag{4.5}$$

and the $N \times N$ matrix is referred to as the DFT matrix.

Due to periodicity $\omega^{nk} = \omega^{n[k]_N}$, where $[k]_N$ denotes $k \mod N$. Consecutively using this identity and matrix shuffling (column-wise permutation) that groups the even and odd columns, the DFT matrix in Equation 4.4 can be split into a chain of $\log N$ sparse matrices [Bri88] referred to as the FFT matrices if N is a power of two. Each row of any FFT matrix contains exactly two non-zero entries with one entry being always 1 which does not need to be stored explicitly. We illustrate this factorization for a four-element input signal.

$$\begin{pmatrix} F(0) \\ F(2) \\ F(1) \\ F(3) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & \omega^2 & 0 & 0 \\ 0 & 0 & 1 & \omega^1 \\ 0 & 0 & 1 & \omega^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & \omega^2 & 0 \\ 0 & 1 & 0 & \omega^2 \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \end{pmatrix} \tag{4.6}$$

As one can see in Equation 4.6, the output signal $F(n)$ has to be rearranged to yield the Fourier coefficients in the right order. The 2D FFT can be computed by consecutive 1D FFTs first along the rows and then along the columns or vice versa. Referring to Equation 4.6, for a 2D signal of size N^2, the FFT can be computed in $\mathcal{O}(N^2 \log N)$.

4.1.2.2 GPU Implementation

A GPU-accelerated implementation of the FFT was pioneered by Moreland et al. [MA03]. Concerning the performance this implementation turned out to be slower than the FFTW library. Schiwietz et al. [SW04] and Jansen et al. [JvRLHK04] presented different implementations with a performance being comparable to that of the FFTW.

Taking into account the particular structure of the FFT matrices, we employ a special representation for the FFT matrices on the GPU. Every FFT matrix is represented as an 1D RGBA texture containing in the i-th texel the value of w^k and the column index of w^k and 1 in the i-th row of the matrix. As an example, Table 4.2 shows the texture content for the first matrix on the right hand side of Equation 4.6. The real and imaginary components of w^k are stored in the R and G channels and the two column indices in the B and A channels of the four component texture.

$\begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} \omega_r^2 \\ \omega_i^2 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} \omega_r^1 \\ \omega_i^1 \\ 3 \\ 2 \end{pmatrix}$	$\begin{pmatrix} \omega_r^3 \\ \omega_i^3 \\ 3 \\ 2 \end{pmatrix}$

Table 4.2: *Four-component texture layout for the precomputed FFT table for the first matrix on the right hand side of Equation 4.6. Here, ω_r^k and ω_i^k denote, respectively, the real and imaginary parts of ω^k defined in Equation 4.5.*

The 1D signal to be transformed is stored in one line of a 2D texture with the real part in the R channel and the imaginary part in the G channel. In this way, a stack of 1D signals can be transformed efficiently. To perform the matrix-vector multiplication at a particular FFT stage, two triangles covering the entire texture is rendered. The two

triangles, along with the 2D texture containing the signal and the 1D texture containing the FFT table, are used as input parameters to the shader program (see Figure 4.3).

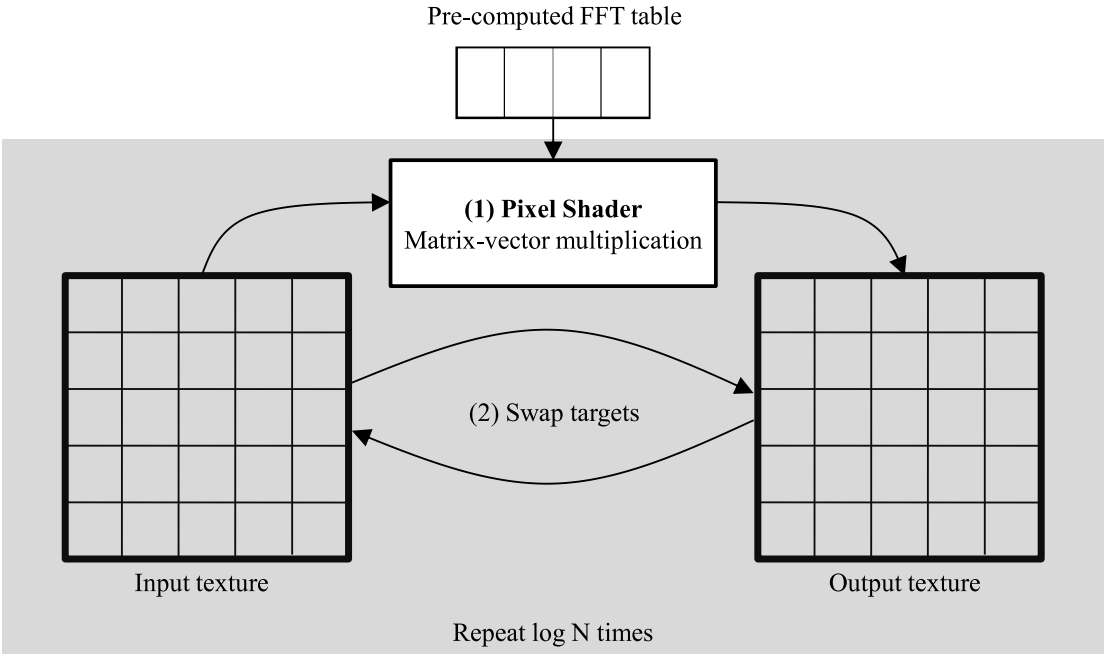


Figure 4.3: *The FFT shader samples the FFT matrix entries of a specific stage. Then, it samples the two vector components indexed by the table and performs the dot product of matrix row and vector. The roles of the input and the output texture are swapped in each stage.*

To avoid switching to a different FFT table in every FFT stage, all these textures are combined into one single 2D texture. The respective row to be accessed in each stage is specified in a constant parameter to the shader program. In every pass, we swap the roles of the input texture and the output texture as illustrated in Figure 4.3. After $\log N$ passes, the row-wise FFTs have been computed, and the results have to be reordered. We precompute the swapping positions and store them in another texture. This texture stores an index to the untangled position of each vector component. A pixel shader program queries the reorder texture to get the correct position of the vector component. Finally, a 2D transformation can be achieved by a consecutive column-wise 1D transformation.

The FFT performance can be doubled if a second data set of the same size is available, since both sets can be transformed in parallel by storing the second set in the BA channels and by using vector instructions on the GPU. This effectively halves the number of texture fetches and arithmetic instructions.

4.1.2.3 Performance

To verify the effectiveness of the proposed FFT implementation, we investigated its performance for different image sizes. In particular, we compare the performance of the GPU-FFT with that of the FFTW library [FJ98] which is an efficient CPU implementation of the FFT leveraging various acceleration strategies like SSE parallelization, cache optimization, and precomputed FFT tables. To conduct a fair comparison, the FFTW MEASURE setting was enabled and the code was run in 32-bit floating point precision. In all our experiments, the time it takes to perform the FFT of a discrete complex 2D signal was measured. Table 4.3 essentially shows the GPU-FFT to be able to process even high resolution images at interactive rates. The performance doubles instantly as the two complex signals are stored in one RGBA texture. We measured the performance using an ATI Radeon 9700 GPU. With these two transformations conducted in parallel, our implementation is clearly superior to that of the FFTW library [FJ98].

	256^2	512^2	1024^2
FFTW	2	15	66
GPU-FFT	2	8	38

Table 4.3: *FFT performance in milliseconds. Note that the GPU implementation transforms two 2D signals at the same time.*

4.2 Non-linear Filtering

In contrast to linear filters, non-linear filters cannot be expressed by a convolution with a filter kernel. Typically, non-linear filters are iterative, data-dependent algorithms; for example, image gradients, non-linear image sampling positions, or additional lookup tables are used to drive the algorithms. In the following, we describe three non-linear filter algorithms and the implementation on the GPU. The presented filters reduce noise artifacts, ring artifacts and low frequency artifacts and are all used in the image reconstruction pipeline (see Chapter 5).

The first algorithm is a non-linear noise reduction algorithm based on the local curvature of an image described in Section 4.2.1. Then, a filter algorithm to extract ring artifacts from an image is presented in Section 4.2.2. Usually, this filter is applied to reconstructed slices of a volumetric data set as ring artifacts can occur due to calibration errors of the scanner gantry. Finally, a cupping artifact removal filter is described in Section 4.2.3 to remove a non-uniform distribution of the lowest frequency band in an

image. We present efficient GPU implementations for all algorithms.

4.2.1 Curvature Smoothing

In order to improve the image quality noise has to be removed from images. This is true for both raw data and image data. By applying a noise removal filter to raw data an image of better quality can be reconstructed. One could apply a linear low-pass filter to the image but, unfortunately, this type of filter is not suitable because it not only reduces the noise but smoothes out the edges, too.

A non-linear curvature-based filter for more sophisticated noise reduction was published in [NK95]. Iteratively, an image is smoothed depending on the local curvature and gradient magnitude of the image. In order to preserve the edges, areas of the image with a high gradient are not smoothed while areas with low gradient are smoothed. In the following, we describe the details of the algorithm and an efficient GPU implementation.

4.2.1.1 Algorithm

A continuous scalar field ϕ is smoothed iteratively by the function

$$\phi^{i+1} = \phi^i + \beta C|\nabla\phi| \tag{4.7}$$

where i is the iteration index, β is a free parameter and C is the mean curvature

$$C = \frac{1}{2}(\kappa_1 + \kappa_2) = \frac{|\nabla\phi|^2 \text{trace}(H(\phi)) - \nabla\phi^T H(\phi)\nabla\phi}{2|\nabla\phi|^3}, \tag{4.8}$$

$H(\phi)$ denotes the Hessian matrix of ϕ. Equation 4.8 is derived from the fundamental forms [Far02, Ebe01]. In 2D, Equation 4.8 evaluates to Equation 4.9 and to Equation 4.10 in 3D.

$$C = \frac{\frac{\partial^2\phi}{\partial x^2}(\frac{\partial\phi}{\partial y})^2 + \frac{\partial^2\phi}{\partial y^2}(\frac{\partial\phi}{\partial x})^2 - 2\frac{\partial\phi}{\partial x}\frac{\partial\phi}{\partial y}\frac{\partial^2\phi}{\partial x\partial y}}{2|\nabla\phi|^3} \tag{4.9}$$

$$C = \frac{(\frac{\partial\phi}{\partial x})^2(\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2})+(\frac{\partial\phi}{\partial y})^2(\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial z^2})+(\frac{\partial\phi}{\partial z})^2(\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2})}{2|\nabla\phi|^3} - \frac{2\cdot(\frac{\partial\phi}{\partial x}\frac{\partial\phi}{\partial y}\frac{\partial^2\phi}{\partial x\partial y}+\frac{\partial\phi}{\partial x}\frac{\partial\phi}{\partial z}\frac{\partial^2\phi}{\partial x\partial z}+\frac{\partial\phi}{\partial y}\frac{\partial\phi}{\partial z}\frac{\partial^2\phi}{\partial y\partial z})}{2|\nabla\phi|^3} \tag{4.10}$$

We discretize Equations 4.9 and Equations 4.10 using central differences

$$\begin{aligned}
\frac{\partial \phi}{\partial x} &= \frac{p_{i+1,j} - p_{i-1,j}}{2}, \\
\frac{\partial \phi}{\partial y} &= \frac{p_{i,j+1} - p_{i,j-1}}{2}, \\
\frac{\partial^2 \phi}{\partial x^2} &= p_{i+1,j} - 2p_{i,j} + p_{i-1,j}, \\
\frac{\partial^2 \phi}{\partial y^2} &= p_{i,j+1} - 2p_{i,j} + p_{i,j-1}, \\
\frac{\partial^2 \phi}{\partial x \partial y} &= \frac{(p_{i-1,j-1} + p_{i+1,j+1}) - (p_{i-1,j+1} + p_{i+1,j-1})}{4}
\end{aligned} \tag{4.11}$$

where $p_{i,j}$ denotes the pixel intensity at the raster location (i, j). The curvature C is weighted by the gradient magnitude $|\nabla\phi|$ for normalization purposes. In order to prevent a division by zero in Equation 4.9, the algorithm returns 0 if the gradient magnitude $|\nabla\phi|$ gets small. Since derivatives cannot be computed accurately in the boundary region, we use an out-flow boundary condition: the curvature of the direct neighbor perpendicular to the border tangent is replicated. After that, the image pixels $p_{i,j}$ are updated by the weighted curvature $\beta C|\nabla\phi|$. The free parameter β must be set to a value in the range of $[0; 0.5]$ for 2D images, otherwise oscillations destroy the image. 3D volumes require β to be between $[0; 0.25]$. Usually, four to six iterations are sufficient for satisfying results reducing the noise while preserving the dominant structures of the data set.

4.2.1.2 GPU Implementation

We implement the algorithm using the GPU as follows: Two textures are required to store the input image ϕ^i and the image of the next iteration ϕ^{i+1}. Using the discretization presented in Equation 4.11 only direct neighbors have to be accessed. First, the $C|\nabla\phi|$ is computed which is then used to updated the next image ϕ^{i+1} with respect to β.

The boundary conditions can be handled in two ways (see left image of Figure 4.4). One possibility is to split the image into two triangles (one quadrilateral) covering the interior of the image without boundary and a set of lines and points covering the boundary pixels and the corners. Since the boundary pixel need to copy the curvature of the nearest interior pixel, texture coordinates are pre-computed for the lines and points to address the closest interior pixel. The four lines can be combined into one pass, so

can the four points. In total, four rendering calls are required: 1) the inner area (two triangles), 2) boundary sides (four lines), 3) boundary corners (four points), 4) iteration update (one quadrilateral).

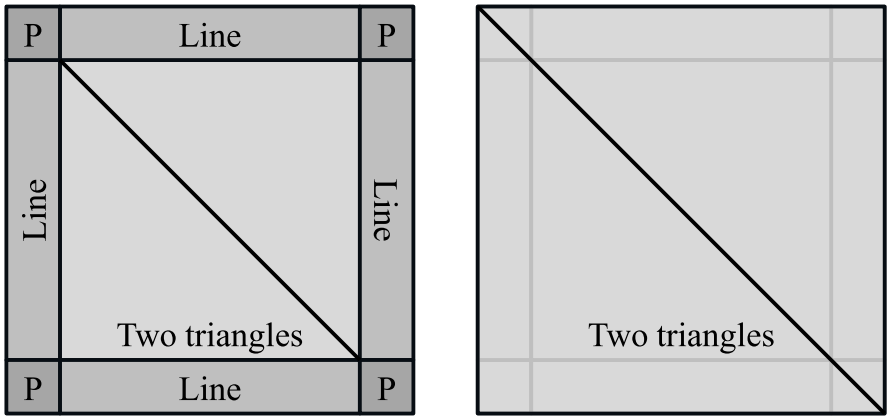


Figure 4.4: *Two ways to handle boundary conditions are illustrated. On the left, the non-boundaries parts are computed first by rendering two triangles. Next, four lines and 4 points (P) are rendered. Alternatively, the boundary condition can also be evaluated in the pixel shader by rendering two triangles covering the entire area (right image).*

Another option to handle the boundary is to take care of it in the pixel shader that computes the curvature (see right image of Figure 4.4). The boundary condition requires the boundary pixels to copy the value from their closest inner neighbor. This is achieved by shifting boundary pixels to the next interior pixel. Since every pixel is addressed by its 2D texture coordinates, the indices of boundary pixels are altered accordingly. Texture coordinates in the range of $[0; N-1]$ are clamped to $[1; N-2]$ in order to treat boundary pixels as inner pixels, which is equal to copying the closest inner pixels to the boundary pixels. A couple of extra instructions are needed for the boundary check and adjustment for every pixel but it turns out that this method is almost two times faster than the method with separate geometries. Furthermore, the latter method is by far simpler to implement, especially in 3D.

4.2.1.3 Large Volumes

The GPU implementation described previously works only correct if all necessary data fits into GPU memory all at once. The algorithm requires $2 \cdot \text{voxels} \cdot \text{sizeof}(\text{float})$ bytes of memory. Currently, graphics cards have typically about 640 MB of memory and, therefore, only volumes up to 256^3 voxels can be processed in one piece. To support larger volumes, a bricking strategy is required that computes the algorithm in parts of

the data set sequentially. A large data set is divided into a set of bricks. The brick size depends on the size of the data set and the algorithm. Two bricking algorithm can be used:

1. *Swap per iteration*: The first brick is uploaded to GPU memory, one iteration of curvature smoothing is computed and the brick is downloaded again. The process continues for all bricks and for all iterations. This strategy produces high bus traffic but on the up side only one pixel overlap to neighboring bricks is required.

2. *Swap per brick*: In order to reduce the bus traffic, it is preferable to compute all iterations of curvature smoothing on one brick at once before it is swapped out. Because of the derivatives the amount of overlapping pixels to neighboring bricks must be the same as the number of curvature smoothing iterations.

The second bricking method is preferable over the first one because of the significantly lower bandwidth requirement, although the number of bricks increases slightly in the second method.

4.2.1.4 Performance

Our GPU implementation shows a significant speed-up over an optimized CPU implementation. We use an Intel Core 2 6600 2.4 GHz CPU, equipped with 2 GB RAM and an NVIDIA GeForce 8800 GTX graphics card for the timings. Table 4.4 shows the results for 2D images, Table 4.5 for 3D images.

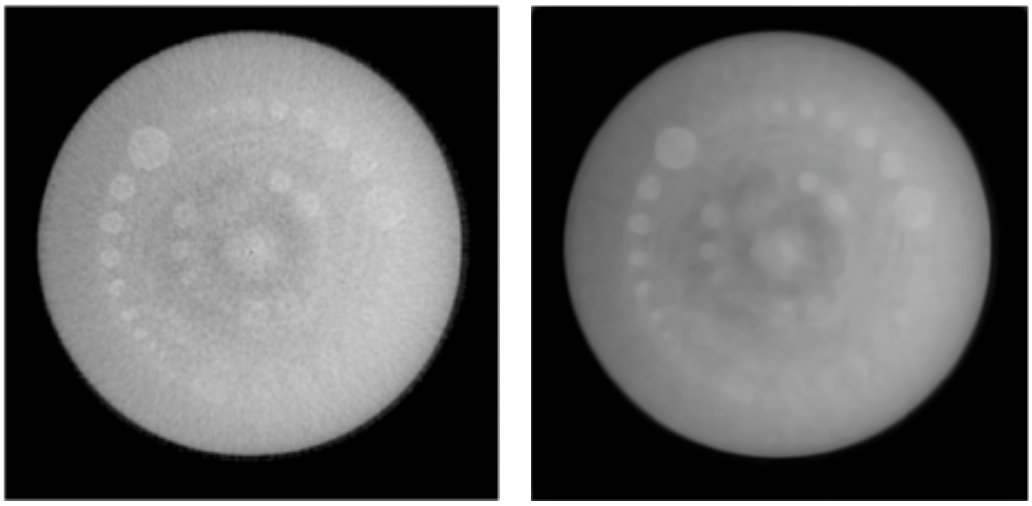
2D image	256^2	512^2	1024^2
CPU	14.74	92.42	213.55
GPU	0.12	0.45	1.80

Table 4.4: *Timings in milliseconds for the curvature smoothing algorithm applied to different image sizes. 10 iterations were measured.*

3D volume	64^3	128^3	256^3
CPU	1514.14	11127.81	89218.74
GPU	0.2	0.247	0.473

Table 4.5: *Timings in milliseconds for the curvature smoothing algorithm applied to different volumes sizes. 10 iterations were measured.*

Figure 4.5 shows an example of curvature smoothing on phantom data. Bright circles can be seen on the disc with a considerable amount of noise. After 10 iterations of curvature smoothing the noise is gone and at the same time the bright circle remain



(a) (b)

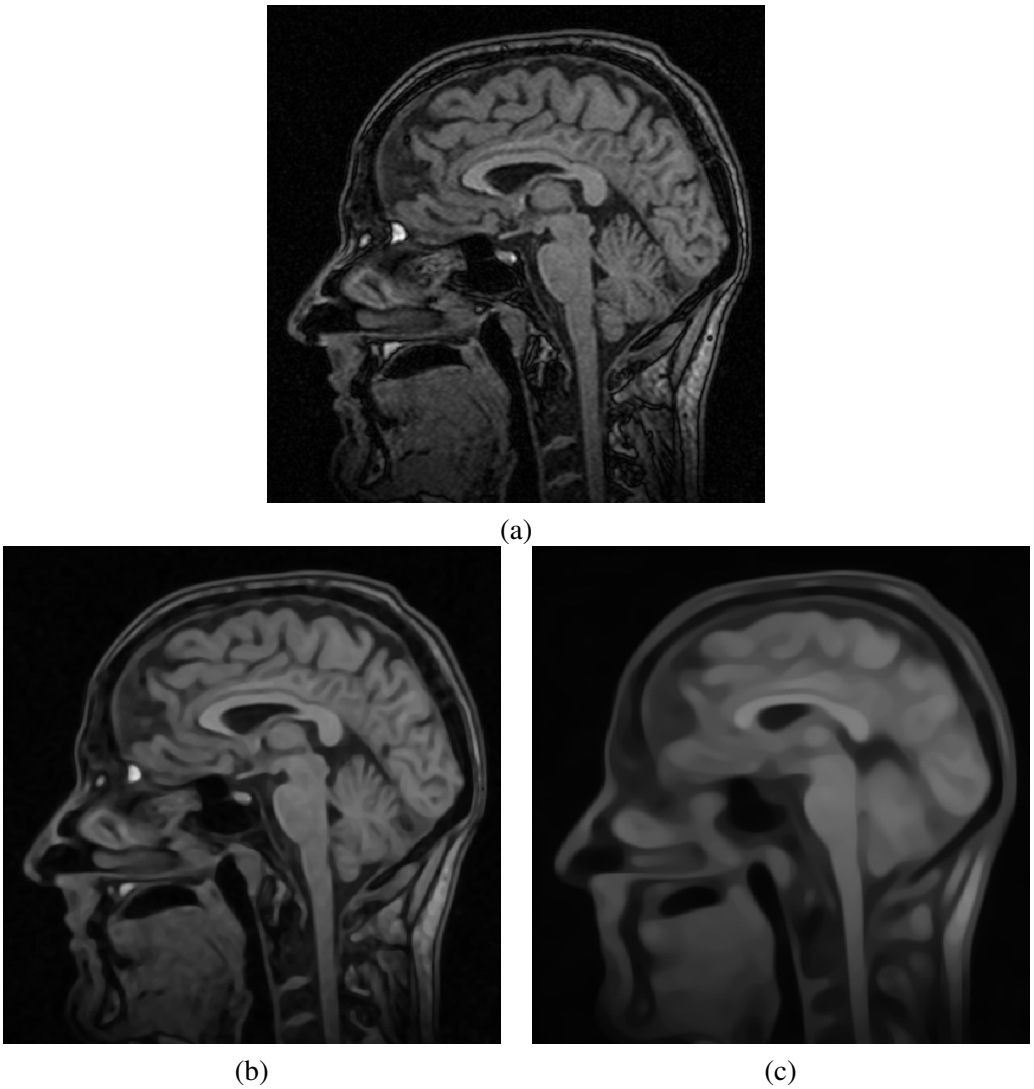
Figure 4.5: *(a) A noisy input image of a phantom. (b) The same image after 10 iterations of curvature smoothing. Obviously, the noise is gone and the edges of the bright circles are still sharp.*

sharp. Figure 4.6 demonstrates the effect of curvature smoothing on a real data set. Image (a) is the original unfiltered image. 10 iterations of filtering were applied to image (b) and 100 iterations to image (c). As one can see, 10 iterations is usually enough to reduce the noise while keeping the edges and destroying not too much image information. Many registration and segmentation algorithms compute considerably better results on pre-filtered images using the curvature smoothing algorithm.

4.2.2 Ring Artifact Removal

Ring artifacts appear in reconstructed volumes because of detector calibration errors. Since the detector calibration is never completely accurate, a post-processing filter helps to reduce the ring artifacts that typically arise. The algorithm is a 2D filter that is applied to all slices of the volume separately as the ring artifacts appear equally in each slice and not volumetrically. Basically, the algorithm extracts the ring artifacts and removes them from the original image. More specifically, the algorithm consists of the following components (see Figure 4.7).

1. *Dynamic Range Restriction*: The dynamic range of the image is clamped to a user defined range. For example, the input image is clamped to 11-bit $[0; 2047]$ to avoid the introduction of artifacts from objects with high contrast.



(a)

(b) (c)

Figure 4.6: *An MR image of a human head. Image (a) shows the original image, image (b) is filtered with 10 iterations of curvature smoothing and image (c) with 100 iterations. Note that after 100 iterations the brain stem and the corpus callosum have still very sharp edges.*

2. *Median Filtering*: Each pixel is filtered by the median of five samples along the radial line to the image center. Figure 4.8(a) depicts the radial line intersecting the image center. The distance d between the sample points is the ring width depending on the gantry geometry. With a median of five samples the sample position are located at the distances $[-2d; -d; 0; d; 2d]$ along the radial line. Finally, the median is subtracted from the original image in order to extract the ring artifacts.

3. *Circular Smoothing*: The extracted ring artifact image is refined by a circular

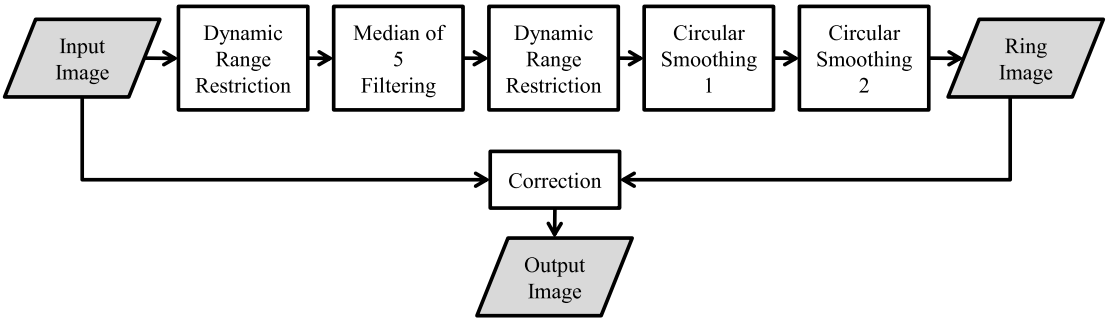


Figure 4.7: *The flow diagram of the ring correction filter. The input image is processed using a variety of filters in order to extract the ring image. Then, the input image is corrected by subtraction of the ring image.*

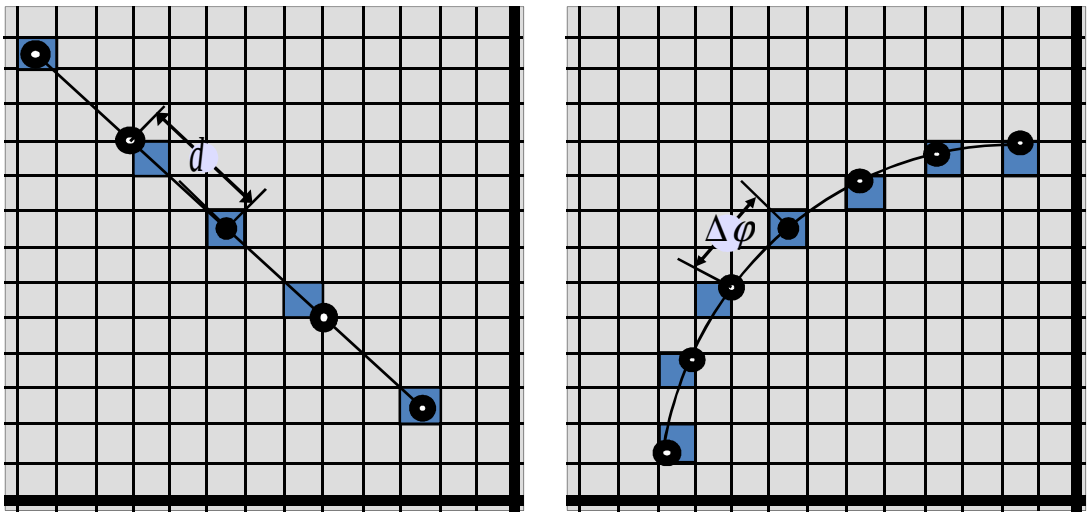
smoothing filter in order to reduce the noise. The center of the circles is fixed to the image center and all circles (with arbitrary radius) are smoothed. The filter averages values along the circular neighborhood. We use a constant angular distance $\Delta\varphi$ between the sample points. In our implementation we use six samples in both directions on the circle resulting in an average of 13 samples. Figure 4.8(b) depicts the geometry of the circular sampling locations.

4. *Correction*: The extracted ring artifacts are subtracted from the original image.

The dynamic range is restricted a second time after the median filtering with another user defined range. Also, the circular smoothing is computed twice with different angles between the samples.

The GPU implementation precomputes the sample locations for both the median and the circular filter pixel-wise. That is, we use Cartesian coordinates instead of polar coordinates in the precomputed tables. Furthermore, we store the linear addresses of the sample locations in the texture components to save memory. The linear address l at location (x, y) is computed with $l(x, y) = (x \cdot w) + y$ where w is the image width. The inverse functions $l^{-1}(a)$ are defined as $l_x^{-1}(a) = a \mod w$ and $l_y^{-1}(a) = \lfloor a/w \rfloor$.

The median filter requires four additional samples along the radial line for each pixel. A four component texture is sufficient to store the locations. A pixel shader program delinearizes the four positions, samples the values and computes the median by bubble sorting the values and returning the middle element subtracted from the pixel value in the original image. Similarly, the circular smoothing filter precomputes the sample locations in textures. Using the same addressing scheme, three textures with four components each are allocated to store the 12 sample locations. A pixel shader program samples the 13 values and returns the average. In a final pass, the extracted



(a) (b)

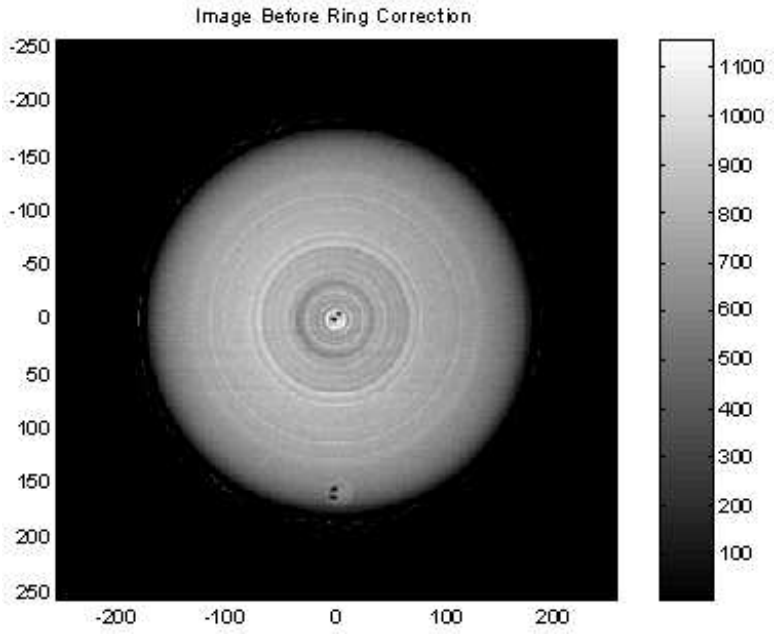
Figure 4.8: *Image (a) shows the sampling pattern for the median filter. The median filter samples four additional values in the distances* $[-2d; -d; d; 2d]$ *along the radial line connecting the current pixel to the center of the image in the lower right corner. Image (b) shows the sampling pattern for the circular smoothing: it smoothes along the circular neighborhood. The circle mid-point is always located in the center of the image while the radius is the distance to the pixel to be smoothed. The samples are taken in equidistant angular steps* $\Delta\varphi$.

ring artifacts are subtracted from the original image. The result of the ring correction algorithm can be seen in Figure 4.9.

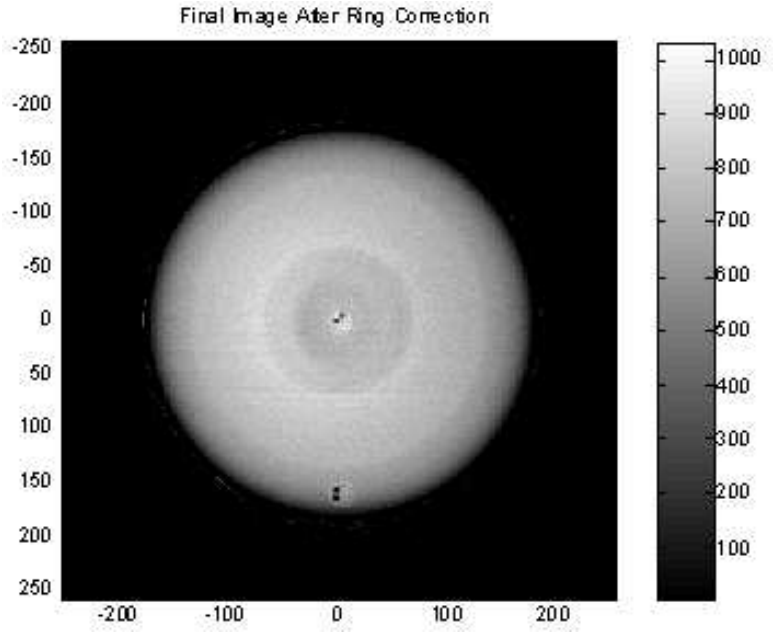
We tested the performance of the ring artifact removal using a data set of 256 slices with 512×512 pixels. Both a CPU and GPU implementation were compared in terms of processing time. Our GPU implementation turns out to be about three times faster than a multi-threaded CPU implementation using SSE instructions. Note that the GPU timings include bus transfer times in both directions. In absolute numbers the CPU implementation took 6.8 seconds (Xeon 3.0 GHz, 2 GB RAM) while the GPU version took 2.5 seconds to process all 256 slices (NVIDIA Quadro FX 4500).

4.2.3 Cupping Artifact Removal

Cupping artifacts occur where a significant proportion of the lower energy photons in the beam spectrum have been scattered and absorbed. Typically, these artifacts are visible as low frequency errors resulting in a non-uniform intensity distribution inside the image. In order to correct the cupping artifacts the low frequency errors must be identified and corrected. It can be observed that the cupping artifacts appear in a range



(a)



(b)

Figure 4.9: *Image (a) suffers from sever ring artifacts. The result of ring correction is shown in image (b).*

of 0.4 to 4 cycles per image dimension. This filter band needs to be extracted and removed from the original image. The cupping artifacts are extracted using Equation

4.12.

$$f_c(x,y) = F^{-1}[F[f(x,y)](1 + kW(u,v))] \tag{4.12}$$

where $f_c(x,y)$ denotes the cupping artifacts, $f(x,y)$ is the two-dimensional input image, $F(x,y)$ is the Fourier transform of the image $f(x,y)$, $W(u,v)$ is a low-pass filter to extract the cupping artifacts and k is a filter gain scalar. We use the Butterworth filter for $W(u,v)$

$$W_n(\omega) = \frac{1}{\sqrt{1 + (\frac{\omega}{\omega_h})^{2n}}} - \frac{1}{\sqrt{1 + (\frac{\omega}{\omega_l})^{2n}}}, \tag{4.13}$$

where $\omega = \sqrt{u^2 + v^2}$ denotes the angular frequency in radians per second, ω_l is the low cut-off frequency, ω_h the high cut-off frequency, and n the order of the filter. In particular, the following parameter values are used:

- Low cut-off frequency ω_l: 2 cycles per meter
- High cut-off frequency ω_h: 3 cycles per meter
- Filter order n: 2
- Filter gain k: 2.25

The filter response consists of the low-frequency cupping artifacts residing in the image. Now, the original input image is corrected using the extracted cupping image. Equation 4.15 describes the correction

$$f'(x,y) = f(x,y) + k \cdot (f_c(x,y) - \beta) \tag{4.14}$$

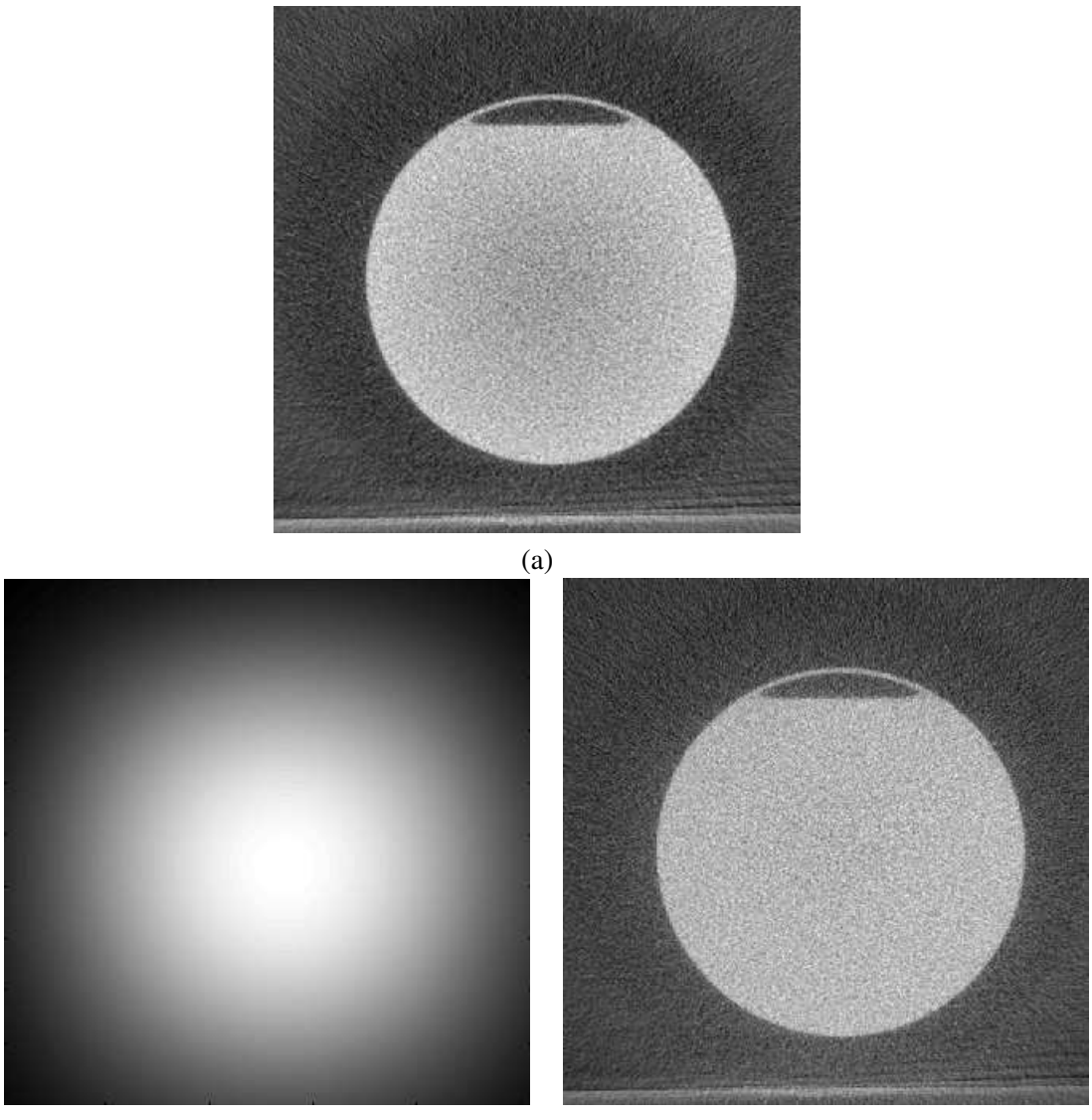
$$\beta = \text{avg}(f_c) + \frac{1}{2}\max(f_c), \tag{4.15}$$

where $f'(x,y)$ is the corrected image, avg and $\max$ compute the average and maximum value in all values of the argument, and the weighting factor k is the same as in Equation 4.13.

This algorithm has the following properties:

- Translation invariant: low sensitivity to the location of the object center.
- Low sensitivity to the object size.

- Little effect when cupping artifacts are not present.
- Very fast as only one forward/backward 2D FFT has to be computed.



(a) (b) (c)

Figure 4.10: *(a) An input image. Typical cupping artifacts can be seen in the center of the image. The contrast falls off towards the center of the image. (b) The filter response after convolving the input image with the Butterworth filter. (c) The image after cupping artifact correction. Now, the slice has a uniform intensity distribution.*

In our GPU implementation, we first calculate the Butterworth filter kernel and store the filter coefficients in a 2D texture. Then, we convolve the filter kernel with the input

image using our GPU-based implementation of the FFT (see Section 4.1.2.1). The input image is transformed to frequency domain, then a component-wise multiplication with the kernel is computed, and the result is transformed back to the spatial domain. In order to compute β for the shift, the maximum and average value in the filter response have to be determined. Therefore, we use two reduce operations on the filter response texture: a reduce maximum and a reduce average (see Section 2.3.3). With the average and maximum value β can be computed. Now, the lowest frequency component (DC) of the filter response is corrected by subtracting β from each pixel weighted by the user-defined filter gain k. The texture update is easily achieved by rendering two triangles covering the whole texture and using an appropriate pixel shader program. Finally, the original image is added resulting in an image without cupping artifacts.

Again, we used a reconstructed volume of 256 slices with 512×512 pixels for testing purposes. As it turns out, our GPU implementation is about twice as fast as our CPU implementation. It takes about 10 seconds to process all 256 slices on the GPU (NVIDIA Geforce 8800 GTX) compared to 4.8 seconds on the CPU (Xeon 3.0 GHz, 2 GB RAM).

Chapter 5

Medical Image Reconstruction

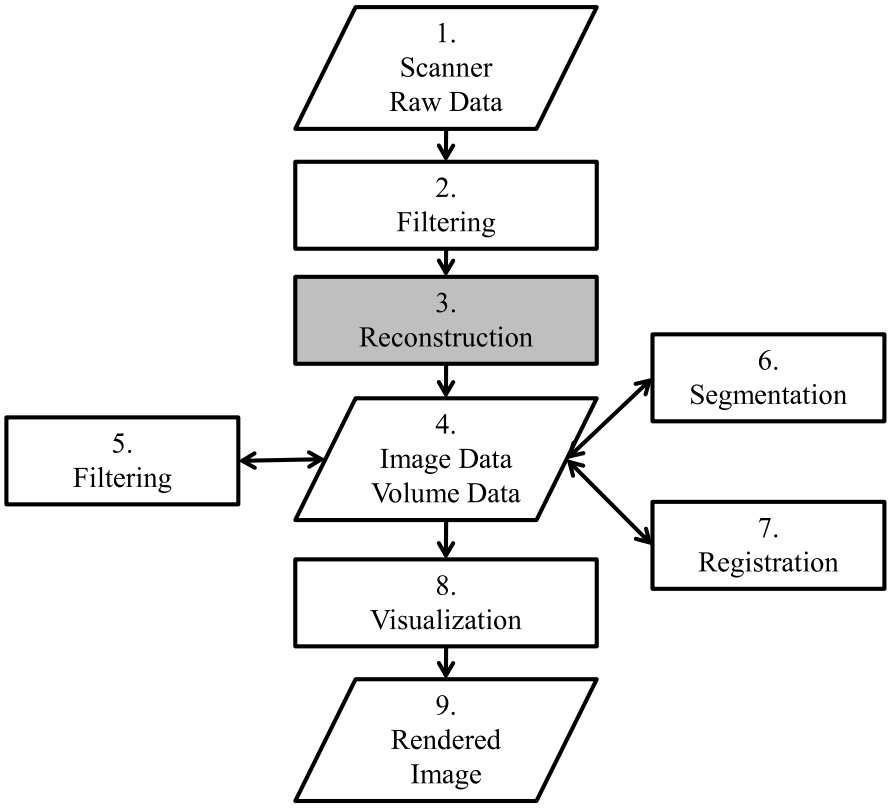


Figure 5.1: *Some scanning modalities require the acquired data to be transformed to the image domain. This process is called reconstruction.*

Usually, the raw data acquired from a scanner is not yet a meaningful image to humans. A process called *reconstruction* transforms the raw data into a visually comprehensive image. While a single projection image of a cone beam scanner can be

viewed independently since 2D X-ray images are acquired, a reconstructed 3D volume provides more spatial tissue type information. The raw data from magnetic resonance (MR) scanners cannot be viewed without reconstruction at all, since it is acquired in *k*-space, a frequency domain. Other modalities such as PET or SPECT require also reconstruction algorithms. In this chapter, we focus on the reconstruction process of CT and MR scanners with GPU acceleration.

5.1 CT Cone Beam Reconstruction

A whole new era of medical imaging was introduced with the invention of cone beam CT scanners in 1972. In the following decades, improvements were achieved in the areas of reduction of radiation, scanning time, and image quality (details and sharpness). Today, the amount of scanned data per scanning session is up to several gigabytes, as the number of projection images (currently around 1000), and the projection image size (currently up to 4096×4096) has grown tremendously.

Nowadays, C-arm and spiral CT scanners are commonly used. In contrast to C-arm scanners spiral scanners move along the patient axis. The reconstruction algorithm for spiral scannersf is different to C-arm scanners since the acquired data may overlap along the patient axis. In the following, we discuss the theory of CT image acquisition and reconstruction from C-arm scanners. First, the physical properties (X-rays) and geometrical arrangement (source-detector trajectory) are presented along with reconstruction theory (inverse Radon transform). Then, our GPU implementation is presented in detail together with strategies for efficient memory management. Finally, results and timings are given.

5.1.1 Theory

In this section, properties of X-rays radiated from a source, attenuated by an object, and recorded by a detector are explained with the purpose of generating an image of the inside of the object. From a set of 2D images a 3D volume is computed with the help of the inverse Radon transform.

5.1.1.1 Radiation and Attenuation

X-ray images are created by the radiation of an X-ray source towards the object of interest. The source emits energy I_0 and it is attenuated by the object while the remaining energy I_d is measured by a detector.

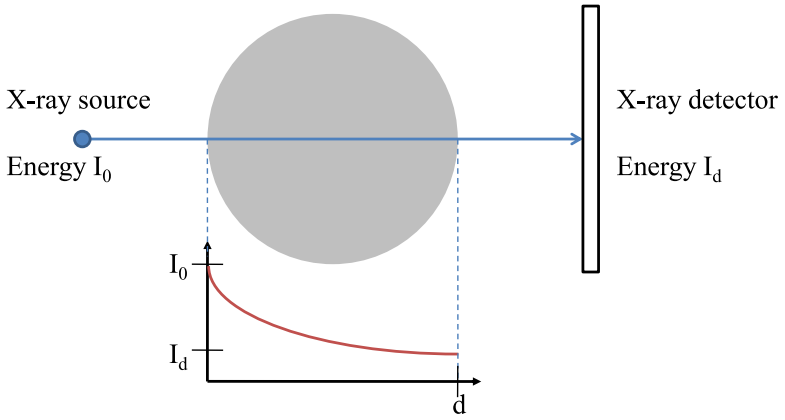


Figure 5.2: *An X-ray with energy I_0 emitted from a source is attenuated by a homogeneous object (gray circle) of radius d. A detector measures the remaining energy I_d.*

Figure 5.2 illustrates the energy attenuation of an X-ray by an object of homogenous material (tissue). The attenuation depends on the type of material or tissue of the object. For example, hard tissue types, e.g. bones, absorb most of the incoming energy while soft tissue types let the X-ray pass through without much attenuation. The energy attenuation follows

$$I_d = I_0 \cdot e^{-\mu d}, \tag{5.1}$$

where d is the traveling distance of the ray within the object and μ is the attenuation coefficient. By computing the attenuation coefficient μ from the source energy I_0 and the detector energy I_d, conclusions about the material/tissue type can be drawn since μ is an indicator of tissue density. The material/tissue type cannot be determined uniquely from the density since most of the human organs fall into the same range of attenuation coefficients. But bones can be identified accurately since nothing else is as dense as bones in human bodies besides implants. The attenuation coefficient can be computed by solving Equation 5.1 for μ.

$$\mu = \frac{1}{d} \ln(\frac{I_0}{I_d}) \tag{5.2}$$

Since a human body does not consist of a single homogeneous tissue type, but of many different types such as bones, organs, etc Equation 5.1 is not sufficient to compute the variety of tissue types along one ray.

Figure 5.3 illustrates an object with two different tissue types where the black circle indicates very soft tissue that does not attenuate the X-ray energy very much. Equation

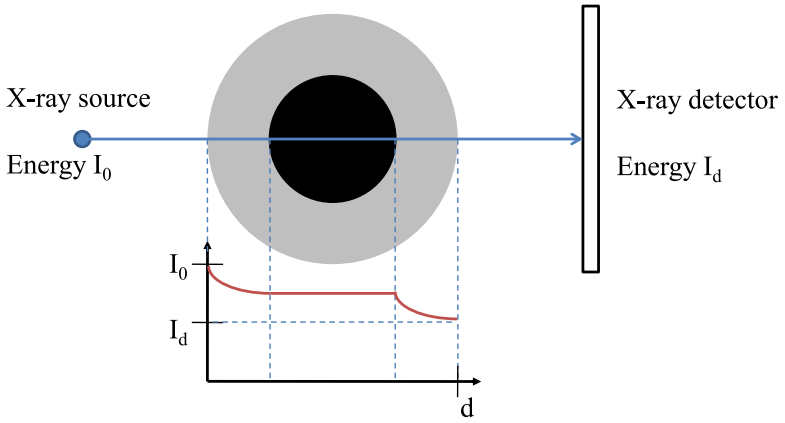


Figure 5.3: *An X-ray is attenuated by an object consisting of inhomogeneous materials (tissues).*

5.1 extends to multiple tissue types as follows

$$I_d = I_0 \cdot e^{-\mu_1 d_1 - \mu_2 d_2 - \mu_3 d_3 \ldots} = I_0 \cdot e^{-\sum_i \mu_i d_i} \tag{5.3}$$

where d_i is the traveling distance of the ray through the tissue type with attenuation coefficient μ_i. The individual attenuation coefficients cannot be computed from a single ray. In order to determine all individual coefficients, multiple images from different view points are necessary. This will be covered in the sections below.

Extending the method depicted in Figure 5.3 to 2D, images are captured where the superposition of all energy attenuation is visualized. 3D volumes can be computed using the inverse Radon transform. Before we discuss the inverse Radon transform, the forward Radon transform is explained in the next section.

5.1.1.2 The Radon Transform

The Radon transform is a projection of an n-dimensional function f to an $(n-1)$-dimensional representation using line integrals. Let a line rotated by angle ϕ be defined by its parametric representation $ax + by + c = 0$ where $a = -sin(\phi), b = cos(\phi)$ and c is a signed distance to the origin. Figure 5.4 illustrates a set of parallel lines sharing the same angle ϕ. Given a function $f(x, y)$ illustrated by the gray circle in the center of the figure, the Radon transformation R_f is defined as

$$R_f(\phi, c) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y)\delta(x\cos\phi + y\sin\phi - c)dxdy \tag{5.4}$$

where $\delta(x)$ is a delta function with the property

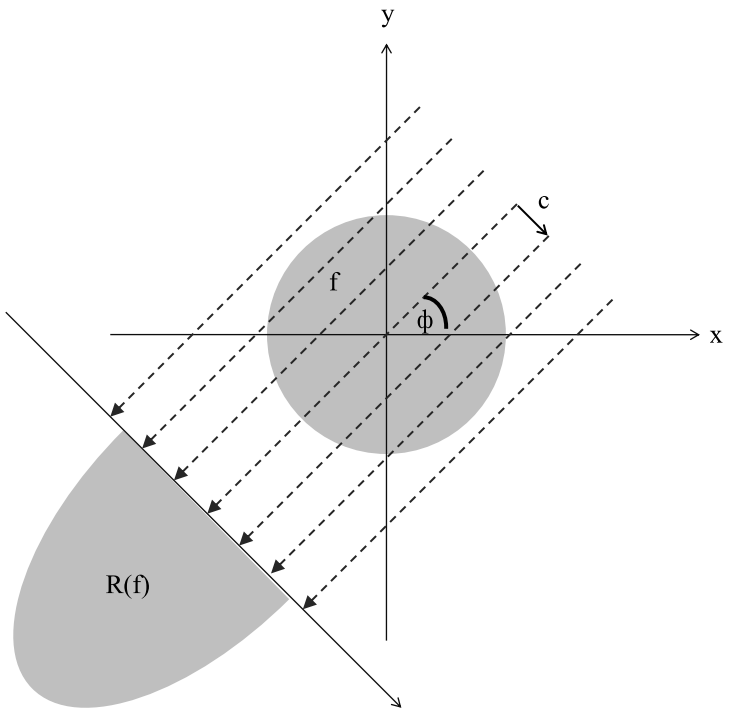


Figure 5.4: *The Radon transformation: A function f (gray circle) is Radon transformed to R_f. The lines are spaced at the distance c and rotated about ϕ degrees.*

$$\int_{-\infty}^{\infty} f(x)\delta(x-a)dx = f(a). \tag{5.5}$$

The delta function opens the gate whenever the parametric line equation is 0 so that the function value f is integrated. The Radon transform is the integral along each line defined by (ϕ, c). The result is illustrated by the gray half ellipsoid in the lower left corner of Figure 5.4. Since an X-ray source is usually a point-based source radiating rays that are not parallel but share the same starting point, the Radon transform can be adapted to this trajectory.

The Radon transform can be applied to higher dimensional data sets as well. Starting from a 3D volumetric data set, the Radon transform integrates lines perpendicular to a plane. The result is a 2D image which is also sometimes called a *projection image*. In the field of visualization a volume renderer is a non-parallel Radon transformation. Starting from an eye point, a view ray is generated through each pixel in the image plane and traced through the volume data set. Intensities are integrated along the way following the Radon transform.

5.1.1.3 The Inverse Radon Transform

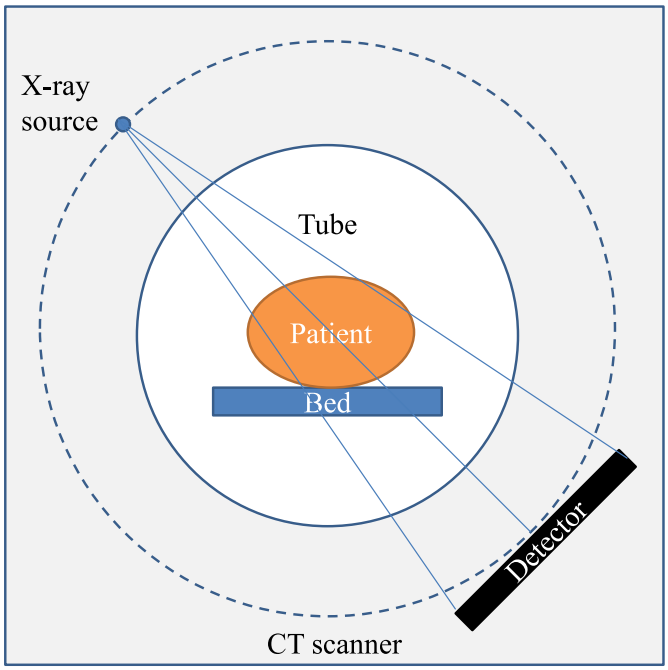


Figure 5.5: *A source/detector pair rotates around the patient acquiring a set of 2D projection images. Later, a 3D volume can be reconstructed from the 2D images.*

Basically, computed tomography is the exact opposite of the Radon transform. After the acquisition of Radon projection images, the original object is reconstructed from them. Technically, the X-ray source and the X-ray detector are mounted on a ring circling the patient (see Figure 5.5). The sequence of projection images is captured by taking an image followed by rotating the pair a few degrees repeatedly (see Figure 5.6).

Using this set of 2D images, a 3D data set can be computed using the inverse Radon transform such that at each 3D point in space the energy attenuation coefficient μ and therefore material/tissue indicators are determined.

The most important algorithms for the inversion of the Radon transform are *algebraic reconstruction* [MYC95] and *filtered backprojection* (FBP) [SBMW07]. The latter method smears the line integrals of each projection image back into volume. Mathematically, the filtered backprojection is described as

$$f(x,y) = \int_0^{2\pi} (R_f * h)(\phi, y \cdot \cos\phi - x \cdot \sin\phi) d\phi. \tag{5.6}$$

Before the projections R_f are smeared back into the volume, a high-pass filter must be applied to the projection data. The filter is expressed as a convolution of R_f and

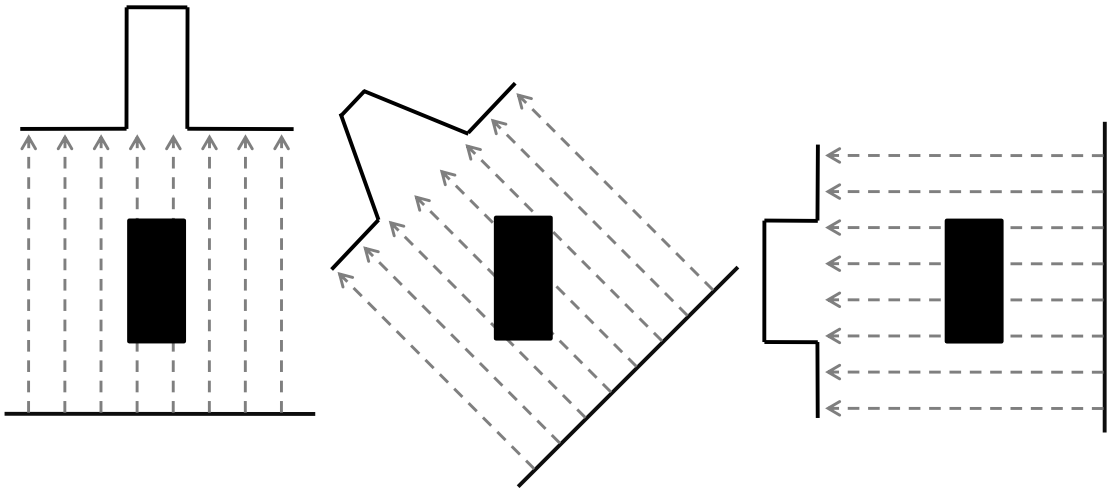


Figure 5.6: *The Radon transform of the black box is shown under three different projection angles. The Radon transform is shown as the black lines.*

h in Equation 5.6. Figure 5.7 illustrates several different high-pass filter functions h in frequency domain, namely the *Ram-Lak*, *Shepp-Logan*, *Cosine*, and the *Hamming* filter. The ideal ramp filter (Ram-Lak) leaves the high-frequency region without major changes. Since noise is also located in the mid to high-frequency region, the reconstructed data looks noisy, too. Therefore, other functions decreasing the highest frequencies in the data produce better results.

Note, that Equation 5.6 is only valid for parallel geometries. Cone beam trajectories can be reconstructed using a perspective projection matrix to transform world-space coordinates to the cone beam coordinate system. The projection matrix depends on the view point, on the viewing angles, and field of view in relation to source/detector gantry.

5.1.2 GPU Implementation

In this section, we describe the data structures and shader of our GPU implementation. Since the projection images and reconstructed volumes might get very large, a memory management strategy is needed. We discuss two approaches with comparisons of memory consumption and bus traffic.

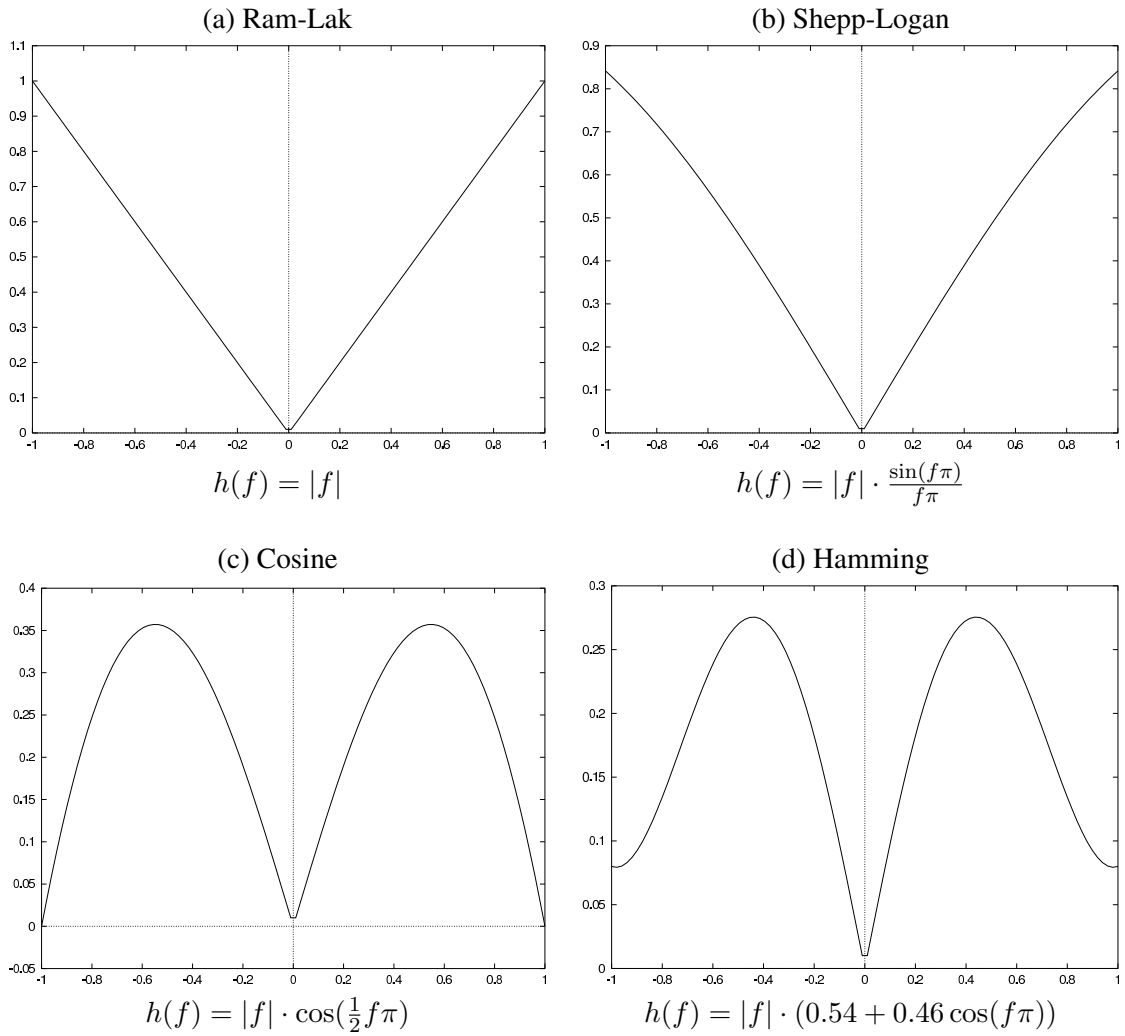


Figure 5.7: *The plots of four high-pass filters in frequency domain.*

5.1.2.1 Data Structures and Shaders

Principally, the volume and one projection image must be represented in GPU memory as textures. It is possible to update a 3D texture in a single rendering pass using Shader Model 4.0 hardware. On older hardware a volume had to be represented slice-by-slice using a stack of 2D textures. In order to reconstruct volumes with maximum quality, each voxel must be float-typed. As for the projection images we use 2D textures to represent them in GPU memory.

We use our GPU-accelerated FFT implementation (Section 4.1.2.2) to filter the projection images. First, we transform each projection image to the frequency domain, then we apply the filter kernel h by component-wise multiplication, and finally the re-

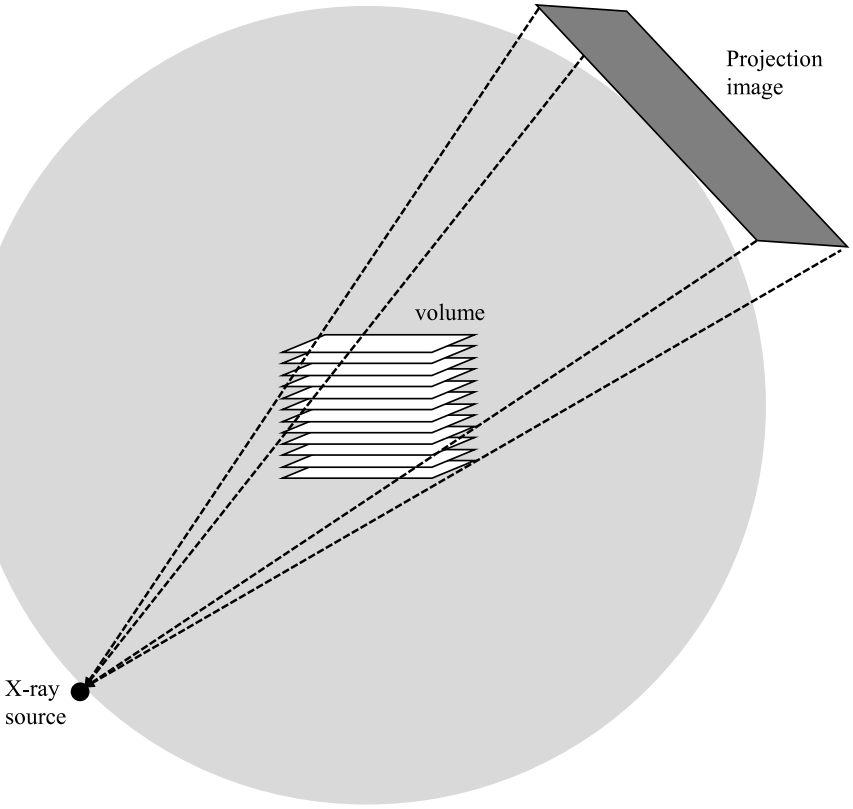


Figure 5.8: *The backprojection of a projection images to the volume is illustrated.*

sult is transformed back to the spatial domain. Two projection images can be packed into an RGBA texture for parallel transformation since complex numbers are generated.

Our backprojection implementation works voxel-driven. That is, we determine for each voxel in the volume the corresponding pixels in the projection images. As shown in Figure 5.8 each projection image affects the entire volume. If the volume is represented as a 2D texture stack, two triangles covering one slice of the volume are rendered. Using a 3D texture, the double triangles of all slices can be put into one vertex buffer allowing the GPU to backproject a projection image to the entire volume at once. Each vertex of the triangles is given a world-space position (x_w, y_w, z_w) that is transformed to projection space using the projection matrix P, where $P \in \mathcal{R}^{4\times3}$. The transformation is performed in the vertex shader.

$$\begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = P \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \tag{5.7}$$

The projection space coordinates (x_p, y_p, z_p) are interpolated across the triangles and are accessible at the voxel level.

The rasterization units of the GPU fill the interior of the circumscribed triangles with voxels and for each voxel a pixel shader program is executed. It fetches the interpolated projection space coordinates (x_p, y_p, z_p) and perform the perspective division yielding the 2D projection image coordinates (u, v).

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x_p/z_p \\ y_p/z_p \end{pmatrix} \tag{5.8}$$

Next, the 2D coordinate (u, v) is checked against the (user-defined) margins of a valid area in the projection image. If (u, v) is outside the valid area, the voxel does not receive a contribution from the pixel at (u, v) in the projection image. Otherwise, the sample is accumulated to the previously projected values of this voxel. It is weighted by the inverse squared distance $1/p_z^2$ to the origin of the projection space system to account for the radiation intensity fall-off.

We count the number of rays passing from the valid area of all projection images through all voxels. Only voxels with a user-defined percentage of valid rays hits are considered as reconstructed correctly while voxels failing this condition are set to zero in post-processing.

We execute the backprojection for each projection image onto the volume. That is, we upload only one projection image at a time and backproject it to all slices of the volume. So, the slices of the volume and one projection image resides in GPU memory concurrently. In the following, we discuss memory management strategies if the GPU memory is not large enough to fit the volume and the projection data in one piece.

5.1.2.2 Memory Management

The above described GPU implementation works only if the output volume and one projection image (all float-typed) fit into GPU memory completely. To reconstruct larger volumes, a memory management strategy is necessary that divides the volumes into chunks. In this section, we discuss two memory management strategies and derive reconstruction pipeline algorithms from them.

At the time when we wrote our implementation, graphics cards had typically 512 MB of RAM. One can easily see that this is not enough memory to reconstruct a volume with 512^3 voxels in floating-point precision. A volume like this requires 512 MB. This does not fit into GPU memory since the graphics card driver uses some megabytes for

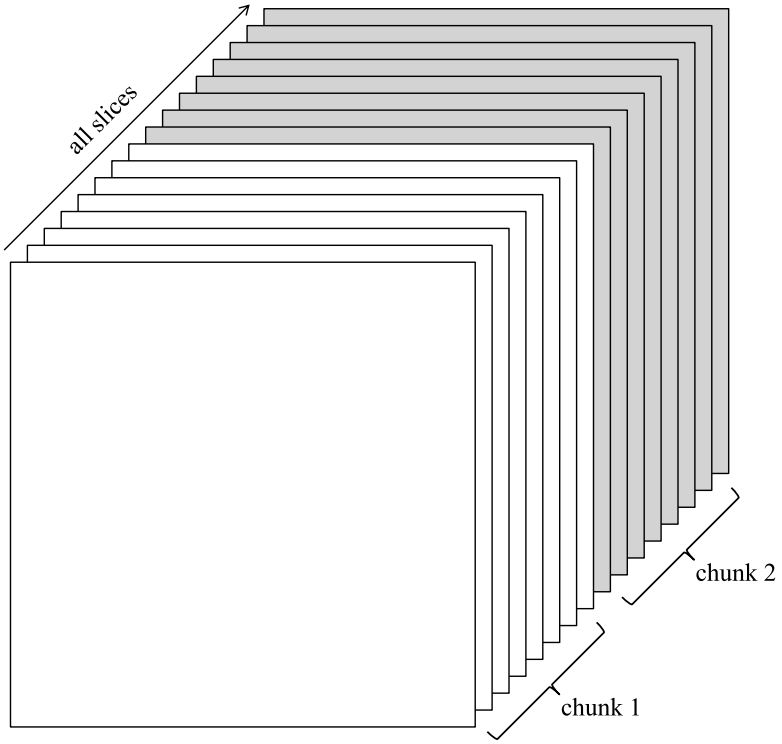


Figure 5.9: *The output volume is divided into a number of chunks. A chunk is a stack of slices representing a part of the entire volume.*

internal buffers and, moreover, at least one projection image need to be stored in GPU memory as well. Therefore, a memory management strategy is required that swaps data between main memory and GPU memory. In general, both projection images and slices of the output volume can be swapped. Since the output volume does not fit into memory in one piece, we divide the output volume into *chunks* as depicted in Figure 5.9. A chunk is a stack of slices representing a part of the entire volume. The slices of all chunks resemble the entire output volume. We describe two swapping strategies for projection images and chunks in the following, but hybrid forms are also possible. In particular, we are interested in the amount of bus traffic between main memory and GPU memory. This is a bottleneck and must be minimized for optimal performance. To quantify the bus traffic we denote the number of projection images as p_n and the size of one image in bytes as p_s. Similarly, we denote the number of chunks as c_n and the size of one chunk in bytes as c_s. As usual, we call copying data from main memory to GPU memory upload and vice versa download. Two swapping strategies:

- *Swapping chunks*: The projection image is the central pipeline element that stays in GPU memory until it is not needed anymore ($p_n p_s$ bytes upload, 0 bytes down-

load). Each projection image is first pre-processed and then backprojected to the volume. Since the volume does not fit into GPU memory in one piece, all chunks have to be swapped in and out for each projection image ($p_n \times c_n c_s$ bytes upload and download).

- *Swapping projection images*: First, all projection images are pre-processed ($p_n p_s$ bytes upload) and then stored in main memory ($p_n p_s$ bytes download). Next, the chunks are processed sequentially. For each chunk, all projection image are backprojected. This means that all projection images have to be uploaded for each chunk ($c_n \times p_n p_s$ bytes upload). Once a chunk is reconstructed completely, post-processing filters can be applied directly before it is downloaded ($c_n c_s$ downloads).

Tables 5.1 and 5.2 show the bus transfer for the two memory swapping strategies. It depends on the size of the projection image and the size of the output volume which strategy produces the smaller bus traffic. A typical scenario for us looks like this: We have $p_n = 360$ projection images with 512×512 float-valued pixels, i.e., $p_s = 1$ MB and the desired output volume with 512^3 floating-point voxels is divided into $c_n = 3$ chunks. Each chunk takes about $c_s = 170$ MB. Swapping chunks causes about 360 GB traffic, while swapping projection images causes only 2.25 GB of traffic. Therefore, our implementation is based on the second approach.

Swapping chunks	bytes upload	bytes download
Projection images	$p_n p_s$	0
Chunks	$c_n c_s \times p_n$	$c_n c_s \times p_n$

Table 5.1: *Bus transfer for swapping chunks strategy where p_n is the number of projection images and c_m is the number of chunks. The size in bytes is represented by p_s and c_s respectively.*

Swapping projection images	bytes upload	bytes download
Projection images	$p_n p_s + c_n \times p_n p_s$	$p_n p_s$
Chunks	0	$c_n c_s$

Table 5.2: *Bus transfer for swapping projection images strategy where p_n is the number of projection images and c_m is the number of chunks. The size in bytes is represented by p_s and c_s respectively.*

As shown in Algorithm 3, all projection images are preprocessed using the GPU. Sequentially, the projection images are uploaded to GPU memory, the curvature smoothing filter (see Section 4.2.1) and the high-pass are applied and the results are downloaded and saved in main memory. Then, the output volume is reconstructed chunk-

Algorithm 3 Cone beam reconstruction pipeline using swapping of projection images.

procedure ReconstructionPipeline(p, c)
  for all projection images p^i **do**
    Upload p^i to GPU memory
    Curvature smooth p^i
    High-pass filter p^i
    Download p^i to main memory
  end for
  for all chunks c^j **do**
    for all projection images p^i **do**
      Upload p^i to GPU memory
      Backproject p^i to c^j
    end for
    Ring artifact removal in c^j
    Cupping artifact removal in c^j
    Download c^j
  end for

by-chunk. All projection images are uploaded sequentially to GPU memory and back-projected to all slices of the chunk. Afterwards, the ring artifact removal (see Section 4.2.2) and the cupping artifact removal (see Section 4.2.3) are applied to the chunk in post-processing. Finally, the chunk is downloaded to main memory. This procedure is repeated for all chunks.

5.1.3 Results

All our measurements were done using Windows XP on a Xeon Dual Core Processor with 3.2 GHz processor and 2 GB RAM. The graphics card is an NVIDIA Quadro FX 4500 with 512 MB RAM. Table 5.3 shows timings on the CPU and on the GPU of three different data sets. We have reconstructed the data sets with varying output volume resolutions. Conclusively, our GPU implementation is approximately four times faster than our multi-threaded SSE2 optimized CPU implementation.

The resulting image quality is virtually the same for both CPU-based and GPU-based implementations with an average pixel error of 10^{-5}. The GPU is very suitable not only for backprojection, but for the entire reconstruction pipeline with very high accuracy. With increasingly growing amounts of data, it is possible that the GPU-based cone beam reconstruction pipeline will be indispensable in the future.

	CPU time (seconds)	GPU time (seconds)
input: 360 images (1024×1024) output: $512 \times 512 \times 256$ voxel	178	53
input: 200 images (1024×1024) output: $256 \times 256 \times 256$ voxel	35	18
input: 200 images (1024×1024) output: $512 \times 256 \times 256$ voxel	123	40

Table 5.3: *We have measured the runtime of Algorithm 3 without thel pre-processing stages but with backprojection and post-processing stages. This table shows timings for three different data sets using our CPU and GPU implementations. The data sets vary in the size and number of projection images as well as the output volume size.*

5.2 Magnetic Resonance Reconstruction

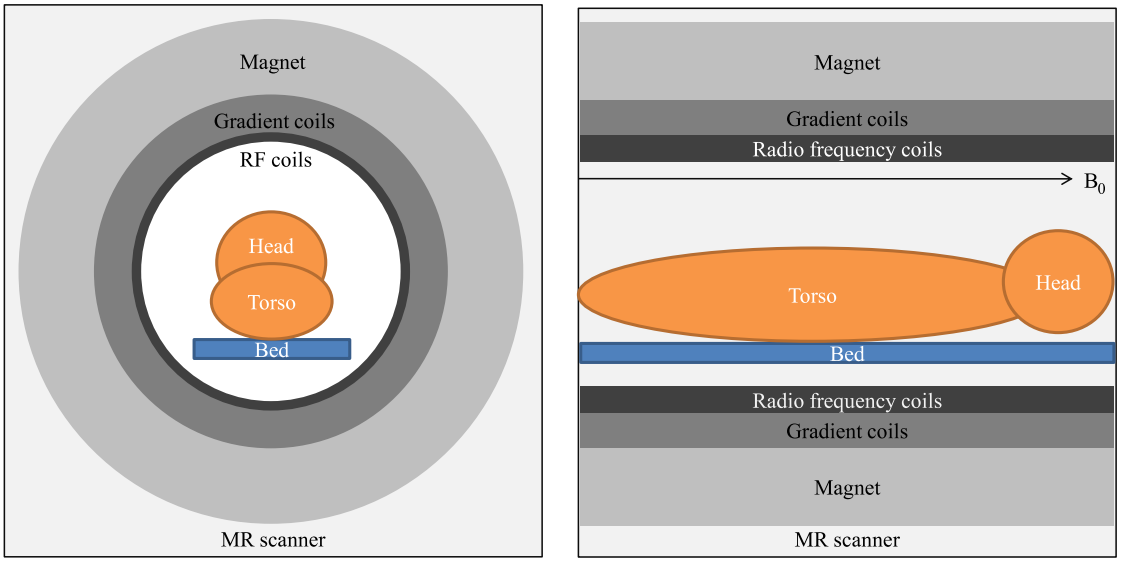
In this section, we present the reconstruction of 2D images from MR measurement data. After a brief introduction to the physics of magnetic resonance scanners we discuss scanning trajectories. Then, we present two reconstruction algorithms, namely the gridding and the filtered backprojection method including GPU acceleration.

5.2.1 Physics of Magnetic Resonance Imaging

Magnetic resonance scanners are one of the youngest scanning modalities available today. MR scanners have many advantages over CT scanners as they capture soft tissue accurately and provide virtually no danger to the human health. The latter makes MR scanners applicable for interventional operations. On the other hand, the image resolution is not as high as in CT images.

Basically, the scanning process works by applying a strong magnetic field to the object in the scanner as nuclei align themselves to it. Applying radio waves in the strong magnetic field raises the nuclei of the object to a higher energy level. By turning off the radio signal the nuclei drop to their original energy level emitting radiation of a specific frequency that identifies the type of nuclei. The magnetic resonance imaging process was improved by using a single high-energy pulse instead of searching through the frequencies looking for responses. Now, multiple nuclei respond to the pulse at different frequencies. Using the Fourier transform, the recorded frequency information is transformed to the spatial domain as image.

While the spin of a nucleus is a quantum mechanical property some of the related phenomena can be understood in classical physics. In such a model, every nucleus spins around a rotational axis. If an external magnetic field is applied to a nucleus, the



(a) front view (b) side view

Figure 5.10: *Image (a) shows a front view and image (b) shows a side view of an MR scanner.*

rotational axis spins around the direction of the magnetic field - this is called precession in classical mechanics. As soon as the magnetic field is turned off, the nucleus returns to its original state while energy is emitted. By applying a second magnetic field perpendicular to the first one, the nucleus goes into permanent precession, if the second magnetic field's frequency coincides with the resonance frequency of the nucleus. As soon as the second magnet field is turned off, the nucleus drops to its original configuration and emits energy. The emitted energy gives rise to a magnetic field that can be transformed to electrical energy by placing a coil near the nucleus. In order to scan a single arbitrarily aligned slice of a body, magnetic gradients are applied to the perpendicular magnetic field to restrict it to a plane. Then, the nuclei outside the plane will not go into precession and, thus, not emit energy. The measured induction is a frequency and phase signal, which can be transformed to an image using the inverse Fourier transform. Figure 5.10 illustrates an MR scanner schematically.

5.2.2 Scanning Trajectories

Fast MR image reconstruction has become more and more important for real-time applications such as interventional imaging. An imaging speed of at least 5 frames per second is necessary in order to provide immediate feedback to the physicians in operation situations. This motivates faster image acquisition and reconstruction.

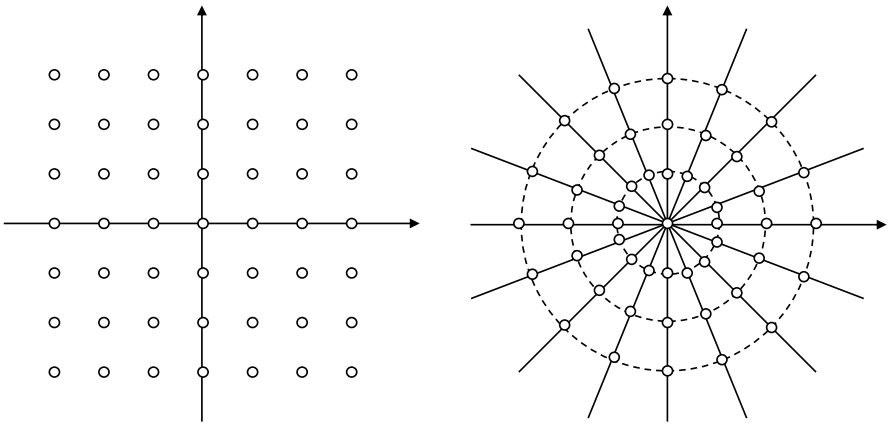


Figure 5.11: *Cartesian and radial sampling trajectories in* k*-space.*

In MR imaging (MRI), the complex-valued raw data measured from the scanner correspond to the Fourier coefficients of the target image; and the Fourier space is referred to as *k*-space. The switching patterns of the magnetic field gradients applied during the measurement determine the sampling trajectories in *k*-space [BKZ04]. Conventional MRI adopts Cartesian trajectories as illustrated on the left of Figure 5.11. When the entire *k*-space is sampled in this fashion, the image is reconstructed by a straightforward application of the FFT to the raw data. Thilaka et al. [TD05] have implemented this using the GPU. This scan pattern is relatively slow in the phase encoding direction [LL00].

Fortunately, higher acquisition speed can be achieved by using a non-Cartesian scan in *k*-space such as radial, spiral or Lissajou trajectories. We focus on radial trajectories (see Figure 5.11 on the right) which offer the following advantages over the Cartesian scan. With their high sampling density near the center of *k*-space (i.e. low frequency components), the reconstructed images achieve relatively high signal-to-noise ratio (SNR) and contrast. They are also less sensitive to motion and flow; and this prevents from producing unacceptable artifacts. Finally, they allow the scan of tissues or objects with very short measurement time in the T2 modality.

Each measurement line passing through the center of the *k*-space consists of n complex-valued measurement samples. Further, each lines is characterized by the angle ϕ_k the line was acquired. Arranging all 1D measurement lines row-by-row in a 2D structure $s(\phi_k, x_i)$ is called the *sinogram* where k is an index to the set of angles ϕ and i is an index addressing individual samples along each line. Note that the measurement lines are acquired in the polar coordinate system and the sample points do not necessarily correspond to discrete Cartesian points. An example of a typical sinogram

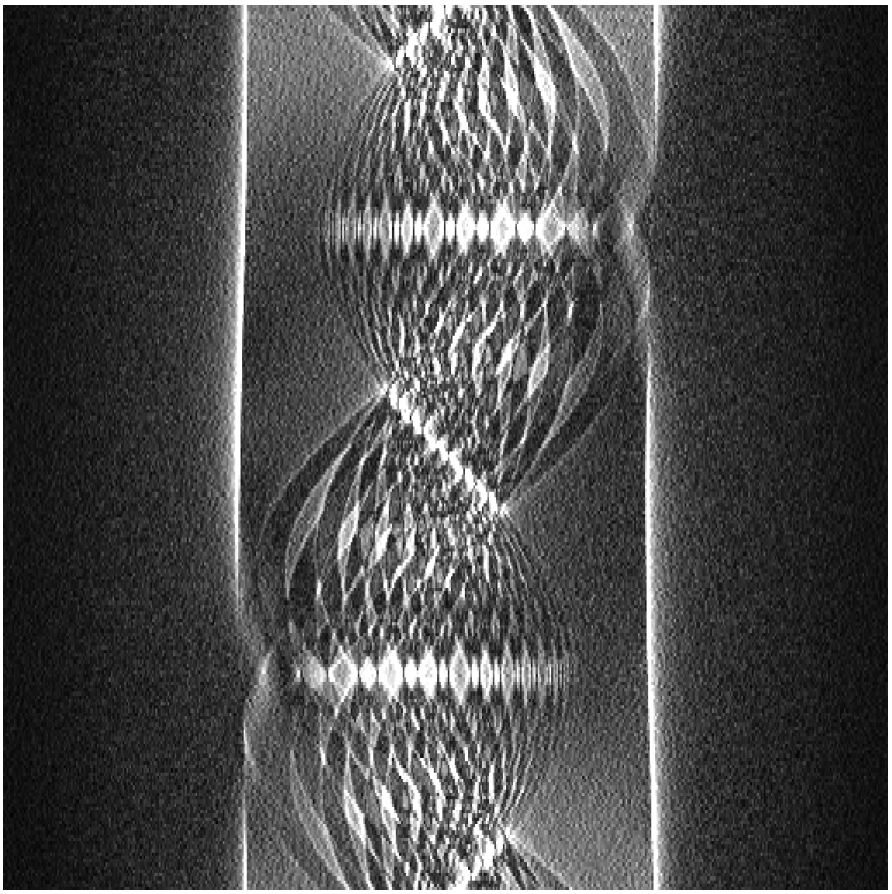


Figure 5.12: *The sinogram of a phantom data set. The magnitude of each complex-valued measurement sample is visualized as gray value.*

can be seen in Figure 5.12. The sine waves gave the sinogram its name. Note that the magnitude of each measurement sample is depicted as gray value.

In the following, we present two reconstruction algorithms for MR images from radial measurement lines: the gridding and the filtered backprojection algorithm.

5.2.3 The Gridding Algorithm

A popular technique to reconstruct images from non-Cartesian trajectories in k-space is the so-called gridding method [O'S85, JMNM91]. This is the method of choice, among various possible MR image reconstruction methods such as iterative approach [SNF03] and pseudoinverse calculation [dWBM$^+$00] if one considers high computation efficiency with reasonable reconstructed image quality. The basic idea of gridding is to resample the raw measurement data on the Cartesian grid. Then, the fast

Algorithm 4 The gridding algorithm for MR measurement data from radial trajectories.

procedure Gridding(m)
  for all measurement lines m_i **do**
    Density compensation of m_i
    Interpolation of m_i (in polar *k*-space) to the Cartesian *k*-space
  end for
  Inverse FFT of the Cartesian *k*-space
  Deapodization of transformed *k*-space

Fourier transform (FFT) is used to reconstruct the target image. The density compensation is necessary to account for the non-uniform sampling. Of particular interest, Dale et al. [DWD01] proposed a fast implementation which exploits table look-up operations. This approach is the foundation on which our implementation is based. Given a reasonable window size for interpolation (usually 3×3 or 5×5), this algorithm provides acceptable image quality with high computational efficiency. However, on currently available MR image reconstruction hardware, this algorithm is still a performance bottleneck for real-time imaging applications.

Gridding algorithms [O'S85, JMNM91] are computationally efficient for reconstructing images from MR measurement data that have arbitrary nonuniform trajectories. The algorithm for radial measurement lines acquired in polar space is sketched in Algorithm 4. We clarify the notation in the following:

- *Density compensation*: To account for nonuniform sampling in *k*-space, each sample point is weighted by a compensation factor. There are several algorithms that aim at approximating the optimal density compensation functions [PM99, SN00].

- *Interpolation*: Each sample point contributes, according to a certain interpolation window such as the Kaiser-Bessel window [JMNM91] to the neighboring Cartesian coordinates. In this step, one can choose to oversample the original *k*-space data and make the grid denser than the original. This oversampling can reduce aliasing significantly and allows for the use of smaller interpolation kernels.

- *Deapodization*: This is a postprocessing operation to compensate for the non-ideal roll-off (non-square) of the frequency responses of the interpolation kernels. This step is sometimes referred to as *roll-off correction*.

We elaborate the interpolation using a Kaiser-Bessel function in a more detailed way. As described by Dale et al. [DWD01], the table-based Kaiser-Bessel window

gridding algorithm distributes each measurement sample on a radial line to a small area (3×3 or 5×5) around the nearest cell on the Cartesian grid. Conceptually, we map the radial measurement line with a thickness of 3 (or 5) pixels to the Cartesian grid. This is illustrated by the skewed light gray box on the right hand side of Figure 5.13. Then, for each covered Cartesian grid cell (k_x, k_y), we compute the contribution $c_i(k_x, k_y)$ from the three nearest measurement samples $m_i(k_x, k_y)$ located at $\mathcal{N}_i(k_x, k_y)$, where $i = 0, 1, 2$.

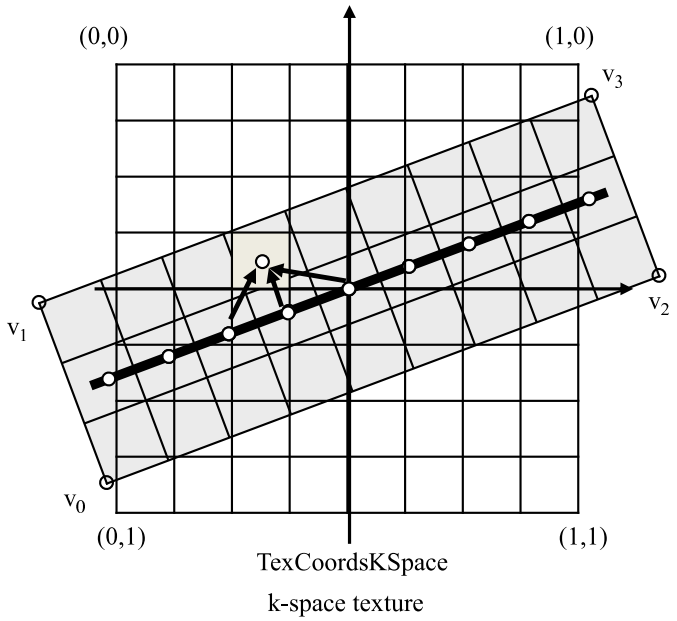


Figure 5.13: *A $N \times 3$ pixels quadrilateral defined by the vertices $v_0 ... v_3$ is rotated by the vertex shader according to the measurement angle. Two coordinate systems are defined: one addressing* k*-space (*TexCoordsKSpace*), the other addressing the measurement data (*TexCoordsMeasurement*). The pixel shader performs a grid-driven interpolation from the three nearest measurement samples with weights computed according to the density compensation function and the distance.*

- The density compensation factor $\rho(D_{\mathcal{N}_i}(k_x, k_y))$, which is inversely proportional to the distance $D_{\mathcal{N}_i}(k_x, k_y)$ between $\mathcal{N}_i(k_x, k_y)$ and the *k*-space origin, is multiplied with $m_i(k_x, k_y)$.
- Then, we use the distance $d_{m_i}(k_x, k_y)$ between $\mathcal{N}_i(k_x, k_y)$ and (k_x, k_y) to look up the precomputed Kaiser-Bessel table and obtain the $w(d_{m_i}(k_x, k_y))$ weighting coefficient so that we have

$$c_i(k_x, k_y) = w(d_{m_i}(k_x, k_y)) \cdot \rho(D_{\mathcal{N}_i}(k_x, k_y)) \cdot m_i(k_x, k_y). \tag{5.9}$$

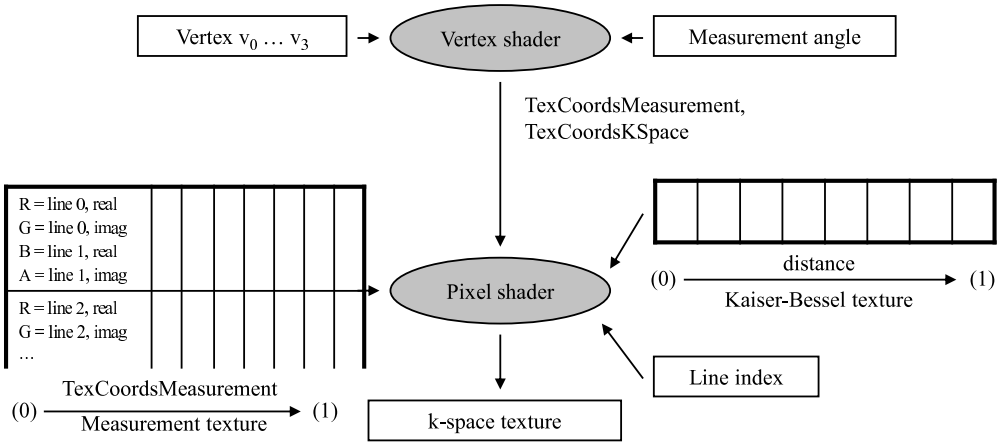


Figure 5.14: *An overview of our GPU gridding implementation showing shaders and texture tables.*

Contrarily, our GPU implementation gathers surrounding measurement samples for each Cartesian grid cell since distribution operations were introduced only very recently on the latest generation of GPUs with inefficient performance. Figure 5.14 shows an overview about the shaders and textures. We store the sinogram $s(\phi, x)$ in a 2D texture with four channels called *measurement texture*. Since each measurement sample is complex-valued, two samples can be stored in each RGBA element. Even-numbered measurement lines are stored in the RG channels and odd ones are stored in the BG channels. The gridded results in the Cartesian *k*-space is represented in the *k-space texture*. Again, two channels are required to store this complex-valued signal. The remaining two channels can be used to represent another set of the gridded results. This is useful when there are several measurement channels as two sets of data can be reconstructed in parallel.

We store the Kaiser-Bessel lookup table in a 1D texture (*Kaiser-Bessel texture*) as it is computationally inefficient to evaluate the Kaiser-Bessel function on the fly. This table is precomputed at program initialization. We provide the measurement and *k*-space texture as input data to the shader program. Further, a *line index* parameter is passed to the shader program in order to index a specific measurement line in the sinogram.

Two triangles covering $N \times 3$ pixels are rotated by a vertex shader program according to the measurement angle in order to rasterize the correct Cartesian grid cells in *k*-space. Two coordinate system are required in the pixel shader. One coordinate system addresses the Cartesian grid of *k*-space, called the *TexCoordsKSpace*. The other coordinate system addresses the radial measurement line, called the *TexCoordsMeasurement*.

Both coordinate systems are fixed at each vertex as attributes and interpolated in each Cartesian cell bilinearly.

The gridding is performed in the pixel shader program. The pixel shader gets the interpolated coordinates TexCoordsMeasurement and TexCoordsKSpace. Using TexCoordsMeasurement, we can compute $D_{\mathcal{N}_i}(k_x, k_y)$ and thus $\rho(D_{\mathcal{N}_i}(k_x, k_y))$ for each Cartesian grid cell. Next, $\mathcal{N}_i(k_x, k_y)$ is evaluated and $d_{m_i}(k_x, k_y)$ is calculated. The distance $d_{m_i}(k_x, k_y)$ is used to index the Kaiser-Bessel texture to retrieve $w(d_{m_i}(k_x, k_y))$. Finally, the measurement sample $m_i(k_x, k_y)$ indexed by TexCoordsMeasurement is weighted to obtained $c_i(k_x, k_y)$ as defined in Equation 5.9. This procedure is repeated for $i = 0, 1$, and 2. Using the coordinate system TexCoordsKSpace, these $c_i(k_x, k_y)$ are accumulated into the *k*-space texture from different triangles that encompass (k_x, k_y). The overall contributions that a Cartesian grid cell (k_x, k_y) receives are

$$C(k_x, k_y) = \sum_{v \in \mathcal{R}(k_x, k_y)} \sum_{i=0}^{2} c_i^v(k_x, k_y), \tag{5.10}$$

where $\mathcal{R}(k_x, k_y)$ denotes the collections of radials lines that contain neighboring measurement samples of (k_x, k_y); and $c_i^v(k_x, k_y)$ is the i_{th} contribution from the v_{th} radial line.

After all measurement lines have been gridded, we use our FFT implementation to transform the gridded samples to obtain an image. Finally, the deapodization is applied to the image. We precompute the deapodization coefficients on the CPU and store them in a texture. Then, a pixel-wise multiplication of the image and the deapodization texture is performed on the GPU to render the final image. Before we present results, we discuss the filtered backprojection algorithm as alternative.

5.2.4 The Filtered Backprojection Algorithm

Another method to reconstruct MR data from radial measurement lines is the filtered backprojection algorithm. The difference to the already discussed algorithm in Section 5.1 is that the rays are arranged in a parallel beam instead of a cone beam. According to the central slice theorem, a line passing through the *k*-space origin can be inversely Fourier transformed to obtain projection data in the image domain where the projection angle is perpendicular to the *k*-space line. Therefore, the filtered backprojection algorithm can also be used to reconstruct MR images from radial measurement lines. The algorithm is shown in Algorithm 5.

We discuss the algorithm together with our GPU implementation next. We use

Algorithm 5 The filtered backprojection algorithm for MR measurement data from radial trajectories.

procedure FBP(m)
  for all measurement lines m_i **do**
    High-pass filter m_i
    Transform m_i to spatial domain m_i'
    Backproject m_i' parallel and accumulate to image
  end for

the same texture as in the gridding algorithm to store sinogram $s(\phi, x)$. One additional texture is required to store the accumulation of all backprojection. As we explain below, this texture needs at least two channels. We call this texture the *image texture*. Figure 5.15 illustrates the shaders with all inputs.

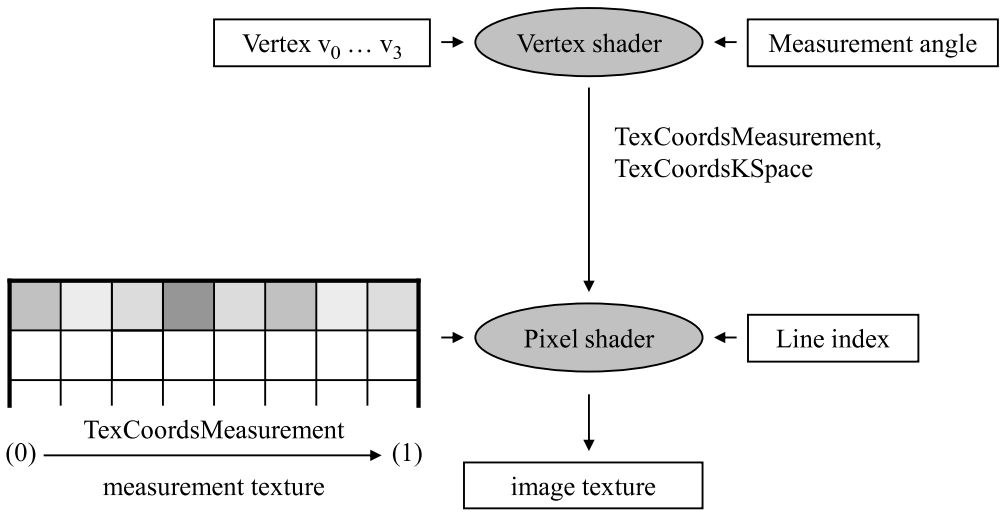


Figure 5.15: *An overview of our GPU backprojection implementation.*

As shown in Algorithm 5, each measurement line is high-pass filtered, transformed to the spatial domain, and backprojected onto the image. We implement the high-pass filter from Section 5.1.1.3 using a component-wise multiplication of each measurement sample and the filter kernel coefficient. Therefore, we store the filter kernel in a 1D texture. Since all measurement lines are stored in a single 2D texture (sinogram) and the measurement data is already in frequency domain (k-space), a single rendering call is sufficient to filter all raw data. Note that the high-pass filter step corresponds to the density compensation in the gridding algorithm.

Next, the filtered measurement data is transformed to the spatial domain using our GPU-FFT implementation described in Section 4.1.2.2. Again, since all measurement

lines are stored in a single 2D texture, the 1D transformation of all lines is done with one rendering call per FFT stage. Note that resulting spatial domain data is still complex-valued.

Finally, each measurement line acquired under an angle ϕ_i is backprojected onto the image domain. That is, each line is smeared across the image under the acquisition angle. In this step, interpolation is of advantage in order to improve the reconstructed image quality, for example by linear interpolation of the two closest samples.

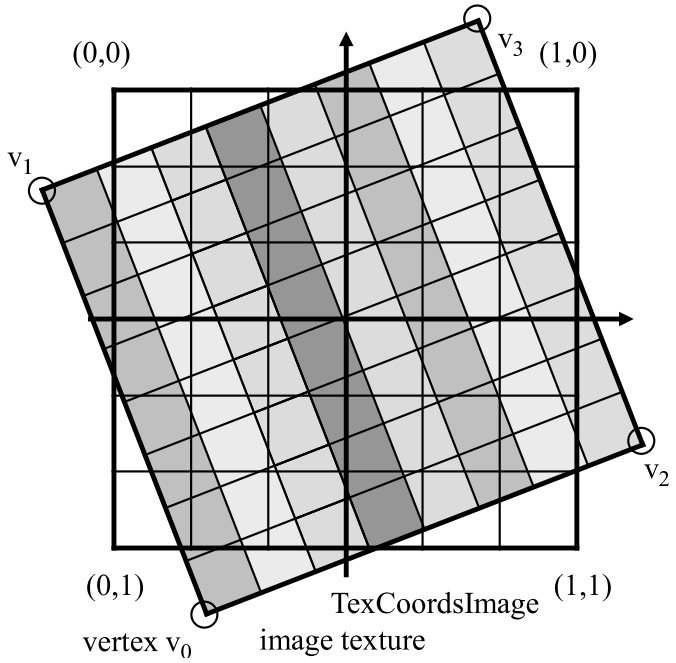


Figure 5.16: *A quadrilateral covering the image domain is rotated by a vertex shader program according to the measurement angle. Two coordinate systems are defined: one addressing* k*-space (*TexCoordsKSpace*), the other addressing the measurement data (*TexCoordsMeasurement*). The pixel shader samples the measurement data at the interpolated TexCoordsMeasurement position and writes it to the interpolated TexCoordsKSpace position. The measurement line is repeated over the entire quadrilateral.*

Figure 5.16 illustrates the backprojection implementation on the GPU. Four vertices $v_0 ... v_3$ setup as two triangles covering $N \times N$ pixels. The two triangles are rotated according to the measurement angle by a vertex shader program. Similar to the gridding implementation, two coordinate systems are required. The coordinate system *TexCoordsImage* addresses the image domain (*image texture*) for the accumulation of the previous backprojections. The other coordinate system *TexCoordsMeasurement* addresses *k*-space samples in the measurement texture. The backprojection is executed in a pixel shader program. The sinogram $s(\phi_k, x_i)$ corresponds to our measurement

texture where ϕ_k is the *line index* and x_i is the *TexCoordsMeasurement*. One measurement line is readdressed inside the rotated quadrilateral to achieve the backprojection (smearing).

Similar to the implementation of the gridding algorithm, the backprojection results have to be accumulated. Using TexCoordsImage, the previous backprojected and accumulated values can be accessed and accumulated. Nowadays, modern GPUs support the blending of floating-point valued textures so that the accumulation can be achieved using the blending states.

We like to point out that the backprojection uses complex values. Both the real and the imaginary part are backprojected separately. The final image is obtained by computing

$$p(x, y) = \sqrt{r(x, y)^2 + i(x, y)^2}, \tag{5.11}$$

where r and i are the real and imaginary values. To conclude this approach we consider it easy to implement and very GPU friendly. No look-up tables or dependent textures fetches are required.

5.2.5 Results

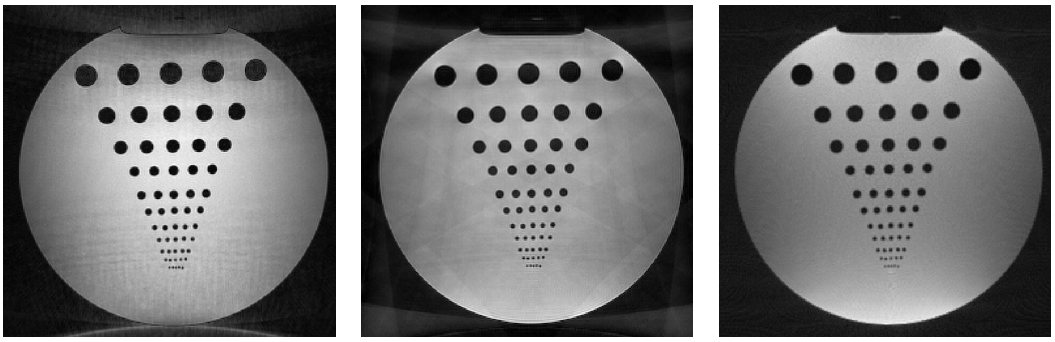
All our experiments were run under Windows XP on a Pentium 4 3.0 GHz processor equipped with an ATI Radeon X1800 XT graphics card. We measure the reconstruction performance for two sets of MR measurement data, which were obtained from a Siemens Magnetom Avanto 1.5T scanner using a trueFISP pulse sequence with a radial trajectory in *k*-space. For the phantom image shown in Figure 5.17, there are a total of 504 radial lines with 512 samples each. During the MR scanning, three coils/channels were used; and the scanning parameters are TR = 4.8 ms, TE = 2.4 ms, flip angle = $60°$, and FOV = 206 mm with a resolution of 256 pixels. For the head image depicted in Figure 5.18 there are a total of 248 radial lines again with 512 samples each. In this data set, four channels were used; and the scanning parameters are TR = 4.46 ms, TE = 2.23 ms, flip angle = $50°$, and FOV = 250 mm with a resolution of 256 pixels.

	Backprojection		Gridding	
	CPU	GPU	CPU	GPU
Phantom	16700	130	730	200
Head	8400	60	600	170

Table 5.4: *MR reconstruction time in milliseconds on the CPU and GPU using backprojection and gridding.*

Table 5.2.5 shows the performance of our various implementations. We have implemented the backprojection and gridding algorithms on both the CPU and GPU for speed comparisons. The speed up from the CPU to the GPU is about 3.5 times for the gridding algorithm. For the filtered backprojection, we observe a speed up of about 100 times. Figure 5.17 shows the reconstructed images for the scanned phantom. The results obtained from our filtered backprojection implementation are identical on the CPU and on the GPU. For the gridding implementations, the results show slight differences but comparable image quality.

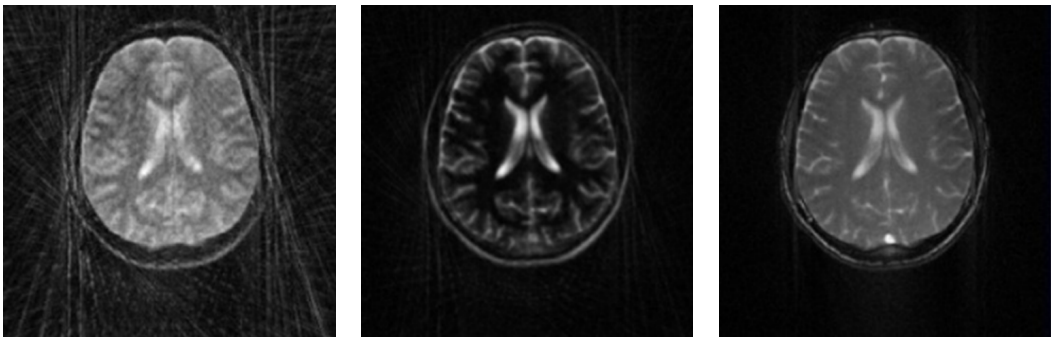
For the GPU implementations, we measure the bus transfer time between main memory and GPU memory in both directions. Basically, the measurement data is uploaded to GPU memory and one reconstructed image is downloaded. The upload time for the phantom image is 6.2 milliseconds for 504 measurement lines with 512 samples each in three channels. For the head image the upload time is 4.4 milliseconds for 248 measurement lines with 512 samples each in four channels. The download time for a reconstructed 256×256 image is 0.4 milliseconds.



(a) (b) (c)

Figure 5.17: *The reconstructed images by using (a) gridding algorithm on the CPU, (b) gridding algorithm on the GPU, and (c) filtered backprojection which yields identical results on the CPU and GPU. The measurement data were obtained from 3 MR coils/channels. For each channel, there are 511 measurement lines with each containing 512 complex samples. We show the reconstruction speeds in Table 5.2.5.*

Conclusively, our GPU backprojection is the best implementation in these experiments. It computes correct images (identical to CPU implementations) at high performance. In order to conduct a fair comparison the GPU gridding implementation must be removed from the experiments since the image quality differs due to the grid-driven interpolation on the CPU as compared to the grid-distribution approach on the CPU. But even comparing the fastest CPU implementation (CPU gridding) with the fastest



(a) (b) (c)

Figure 5.18: *This first image was reconstructed from 31 measurement lines of 512 (complex) samples in 3 channels to a* 256×256 *image in 8 milliseconds. The subsequent images are reconstructed from additional rotated interleaves. All images were reconstructed using the backprojection algorithm. Our CPU and GPU implementations yield identical results.*

GPU implementation (backprojection) yields a 10 times speedup roughly.

Chapter 6

Medical Image Segmentation

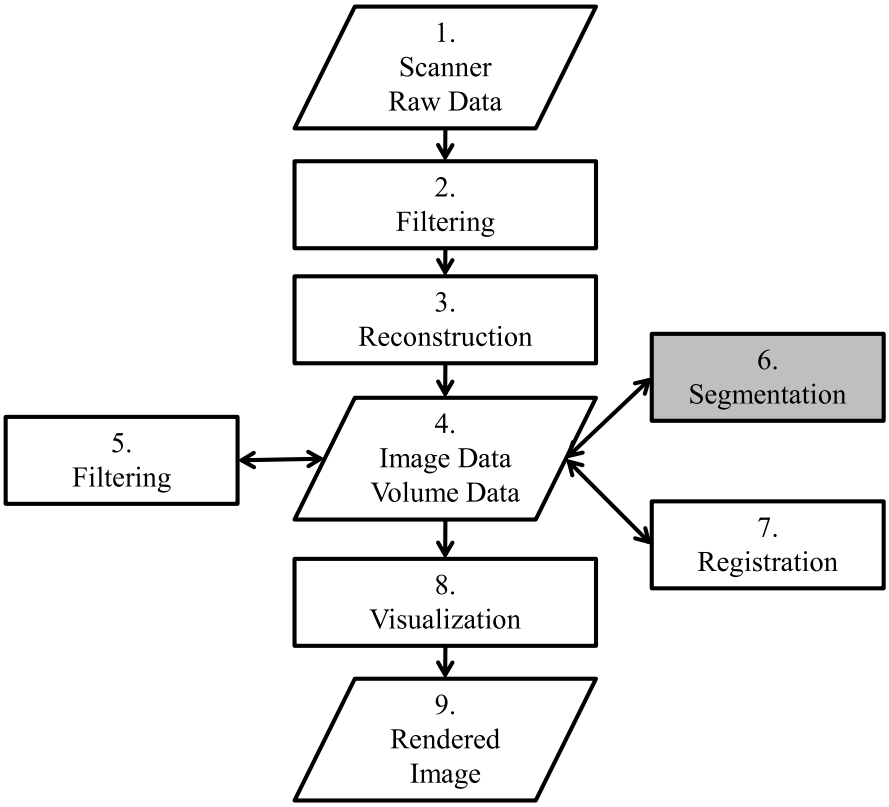


Figure 6.1: *Medical image segmentation isolates objects by (automatically) finding the boundaries and removing the background.*

Medical image segmentation separates objects in an image by finding the boundaries of the objects and extracting them from the image. Especially in 3D it is difficult to see organs inside the body since they might be occluded by tissue in front of them.

Segmentation helps getting a better view on an organ by extracting it from the surroundings. The extraction is, of course, only virtual in the medical image. Other applications besides visualization include finding the boundary of a tumor, measuring volumes of segmented objects, or guiding computer-controlled surgery robots.

6.1 Introduction

It is tedious work to segment organs manually thus many segmentation algorithms have been developed in the last decade. A very simple kind of automatic segmentation is achieved by employing a transfer function. Transfer functions map values, e.g. Hounsfield units, to other values. For example, parts of a data set are set transparent. A simple transfer function is a ramp function which sets values below a user-defined bottom threshold to 0 and above a top threshold to 1. In between the values are mapped linearly. Obviously, this allows only a rough segmentation of the data set into air, fat, water and bones. Since most organs contain both fat and water the transfer function is not capable of providing arbitrary organ segmentations.

Two types of user interaction are distinguished in the field of image segmentation. While *automatic segmentation* algorithms find the boundaries of objects in an image without user guidance, *semi-automatic* algorithms rely on a few labeled pixels provided by the user. That is, the user selects some pixels belonging to the object and, depending on the algorithm, some pixels not belonging to the object separately. User-selected pixels are referred to as *labeled pixels* or simply *labels*. Fully automatic algorithms are often specialized in finding one specific type of object whereas semi-automatic algorithms can be used to classifying everything in an image. Semi-automatic algorithms can be made fully-automatic if the labels are computed by an algorithm instead of a human.

Segmentation algorithms can also be classified by the number of objects (or regions) an algorithm is able to separate at the same time. Instead of separating only an object (foreground) and the surroundings (background), *multi-segmentation* allows separating an image into a multitude of n regions. For example, multiple organs are segmented from the rest of a human body in a single run of an algorithm.

Furthermore, segmentation algorithms differ in the kind of output. While *binary segmentation* algorithms classify each pixel into one of the n regions uniquely, *non-binary segmentation* algorithms assign a probability to each pixel of belonging to each of the n regions. Non-binary segmentation is considered to be more powerful than binary segmentation since there exist many ambiguous regions in human bodies.

6.2 Classes of Segmentation Algorithms

There is a very active research community working on novel segmentation algorithms or on improvements to existing ones. Today, a large variety of different classes of segmentation algorithm have been developed. The following list describes three classes briefly.

- *Watershed Segmentation*: A topological surface is build from the gradients of an image. If the image were flooded from the minima of the topological map an image segmentation is achieved by looking at the basins limited by the topological surface [CCGA07, NWvdB00, dBP07].
- *Level Set Segmentation*: Level Set methods are based on active contours minimizing a cost function. The cost function is chosen depending on the type of the desired segmentation and additional smoothing constraints [Set99]. Lefohn et al. [LCW03] have implemented a GPU-accelerated algorithm for level sets. A review on statistical level set methods can be found in [CRD07].
- *Graph-based Segmentation*: The image is represented by a regular graph where each pixel (voxel) value is living on a node in the regular grid and the edges represent the gradients. Examples for segmentation algorithms based on graphs are *normalized cuts* [SM97], *graph cuts* [YM01], and the *random walker* [LG04].

Often, further knowledge is integrated into the algorithms such as histogram equalizations, shape priors, flow-based constraints and many more. The desirable goals of a prefect segmentation algorithm are characterized by the following (incomplete) properties.

- No limitation to specific types of organs.
- No limitation to specific image acquisition modalities (CT, MR, Ultrasound, ...).
- Non-binary segmentation (probability).
- Manageable amount of parameters.
- Minimal amount of user guidance and labeled points.
- Robust to noise.
- Easy to implement.

- Interactive real-time performance.

Unfortunately, it is almost impossible to create an algorithm that satisfies all of the goals above. Nowadays, people focus on a subset of these goals and try to build special purpose algorithms. The random walker algorithm [LG04] is an exception since it satisfies many of the above mentioned properties. It is a non-binary segmentation algorithm that works on many image modalities even with weak boundaries. It is easy to implement runs very fast on the GPU. Also, Siemens uses it in many applications. In the upcoming sections, we review the algorithm and discuss its evaluation [GSAW05b]. Further, we present an efficient GPU implementation with timings.

6.3 The Random Walker Algorithm

We briefly review the random walker algorithm. A more detailed explanation can be found in [LG04]. The random walker algorithm allows segmenting an arbitrary number K of regions simultaneously. We distinguish the regions by unique colors. The algorithm is semi-automatic requiring the user to label at least one pixel of the image/volume in each of the K colors.

The algorithm is based on a regular 4-connected grid in 2D and a 6-connected grid in 3D. Each pixel of the image is represented by a node in the grid. While the edges are symmetric, we specify an edge weight depending on the pixel gradients. We chose the edge weight to be large if the gradient is small and vice versa. This way, neighboring pixels with similar intensities have a stronger relationship expressed by the edge weights. We discuss the edge weight computation below in detail.

A random walker standing at an arbitrary vertex in the previously defined grid has four or six choices (in the non-boundary region) to move to another vertex. The edge weighs represent probabilities to chose a direction. Therefore, the random walker moves easily between pixels with similar intensities while it avoids traveling to pixels with a large intensity difference.

As stated earlier, at least K pixels have been labeled by the user (or automatically using an additional algorithm) with K different colors. Suppose a random walk is started from every unlabeled pixel. Every random walker starts moving according to the probability distribution and eventually reaches a labeled pixel. Beginning from the starting point of each random walker, the probability to reach a labeled pixel is computed by multiplying the edge weights along the path. Therefore, K probabilities (for each color) are determined by the random walks for each (unlabeled) pixel.

Finally, an image segmentation is obtained from the random walker paths by looking at the K probabilities of each pixel. We assign to each pixel the color with the highest probability.

In the following, we discuss the theory of the random walker algorithm, the GPU implementation, and a validation of the algorithm with timings.

6.3.1 Theory

First, we discuss the computation of the edge weights. As already mentioned, the edge weights depend on the image gradient since the random walkers are supposed to stay in homogeneous regions and to avoid stepping over sharp boundaries in the image. A Gaussian is used to compute the edge weights as follows

$$w_{ij} = \exp\left(-\beta(g_i - g_j)^2\right), \tag{6.1}$$

where g_i indicates the image intensity at pixel i. The value of β represents the only free parameter in this algorithm and was set to $\beta = 1000$ in all 2D experiments and $\beta = 4000$ for all 3D experiments.

Fortunately, the random walks do not have to be simulated but the exact same result is computable by solving a system of linear equations. More precisely, the random walks are equivalent to the discrete Dirichlet problem [S.45, PL84, RD89] where the labeled pixels represent boundary conditions. A system of equations is solved for each of the K regions where the pixels of the boundary condition are fixed to 1 if the label color corresponds to the current system, otherwise they are fixed to 0.

The random walker problem is expressed by the homogeneous system

$$Lx = 0, \tag{6.2}$$

where L is the Laplacian matrix and x the solution vector. The Laplacian matrix L is defined by its components $L_{v_i v_j}$ as follows

$$L_{v_i v_j} = \begin{cases} \sum w_{ik} & \text{if } v_k \text{ and } v_i \text{ are adjacent nodes and } i = j, \\ -w_{ij} & \text{if } v_i \text{ and } v_j \text{ are adjacent nodes,} \\ 0 & \text{otherwise.} \end{cases} \tag{6.3}$$

The vertices are separated into two sets V_L (labeled vertices) and V_U (unlabeled vertices). Since vertices from V_L are fixed and require no solution, the matrix L is

rearranged

$$L = \begin{bmatrix} L_L & B \\ B^T & L_U \end{bmatrix}. \tag{6.4}$$

Further, the solution vector is split into two sets x_L and x_U analogously.

$$\begin{pmatrix} L_L & B \\ B^T & L_U \end{pmatrix} \begin{pmatrix} x_L \\ x_U \end{pmatrix} = 0. \tag{6.5}$$

Since x_L is fixed Equation 6.5 needs to be solved for x_U only

$$B^T x_L + L_U x_U = 0, \tag{6.6}$$

thus

$$L_U x_U = -B^T x_L. \tag{6.7}$$

The inhomogeneous system described in Equation 6.7 is solved for all labels c with $1 < c \leq K$. Note that x_L must be adapted to account for each label color. If the system of color c is solved, all pixel x_L with label c must be set to 1 whereas all other x_L set to 0. Further, the probability vector for each pixel always sums to 1. That is, once $K - 1$ systems are solved the remaining one is determined from the others.

Equation 6.7 is a sparse, symmetric, positive-definite system of linear equations with $|V_U|$ number of equations and 5 diagonals in 2D and 7 in 3D. In the following section, we discuss the GPU implementation of the random walker algorithm.

6.3.2 GPU Implementation

We implement the random walker system of linear equations using the GPU solvers presented in Chapter 3. Specifically, we compare a GPU-accelerated conjugate gradient and a multigrid solver in the following. First, we describe the vector storage, then the multigrid and conjugate gradient implementation with a discussions of advantages and disadvantages. Timings are shown in the next section.

Since GPU textures store up to four channels (RGBA) per pixel we exploit this data structure to compute four label colors at the same time. That is, we solve four systems of linear equations simultaneously. If more than four label colors are desired, more textures can be used. The four channel storage applies to all vectors such as the solution vector x. Also, all intermediate vectors required by the solution algorithm are packed the same way. Our GPU implementation computes all linear algebra operators on four channels in parallel.

We implemented both a 2D and a 3D version of the random walker systems. In a 2D implementation 2D textures are used to store all vectors since it is straightforward to correlate vector components to pixel positions. In 3D the same mapping is achieved by using 3D textures. As described in Section 2.3.2, 3D textures have a slightly slower writing performance but time consuming coordinate system transformations are saved in contrast to flat textures. However, when we implemented the 3D version no GPU supported for writing to 3D textures was available (Geforce 6 series and Radeon X800 series). This is why our timings shown in the next section have been done using a flat texture layout in 3D.

6.3.2.1 Multigrid

In this section we describe our multigrid implementation of the random walker algorithm as introduced in Section 3.2.4. According to Equation 6.7 labeled pixels are removed from the system and only unlabeled vertices (pixels) are solved for. There are two options to account for this.

- The fixed equations are actually removed from the hierarchy leaving a system with an arbitrary number of equations. But if the number of equations is not a power of two, care must be taken for the restriction matrices as they have to be adapted, too. Also, the GPU multigrid implementation gets slower since GPUs handle power of two texture more efficiently. Another drawback of this approach is that the hierarchy must be recomputed every time the labeled pixels change. This prevents the user to change labels while the solver is running in contrast to our conjugate gradient solver.
- The fixed equations are not removed but the fixed values are reset after every operator. Using this approach the hierarchy can be pre-computed for one image and does not change due to the change of pixel labels.

We found the latter method more promising, but overall, we are not satisfied with the multigrid approach. The main drawback is the difficulty to handle arbitrary label placement efficiently.

6.3.2.2 Conjugate Gradient

In contrast, the perconditioned conjugate gradient method works very well with the GPU. The mapping of the vectors and matrices to GPU structures is simple, arbitrary label placements are handled efficiently and the performance is real-time.

Since the Laplacian matrix L is a diagonal band matrix and has only a few diagonals we store the matrix line-by-line (see Section 3.3.2.1). That is, the matrix texture has the same size as the image or any of the vectors. In 2D we store the four off-diagonal entries in an RGBA 2D texture representing the edge weights to the four direct neighboring vertices. The main diagonal is computed online by the negative sum over all off-diagonal elements. Since the matrix is symmetric our representation contains redundancies but we avoid additional indirect texture fetches this way. In 3D 6 off-diagonals have to be stored. This time we exploit the symmetry and store only three weights in an RGBA 3D texture or 2D flat texture respectively. The remaining off-diagonals are retrieved from additional texture fetches to neighboring vertices.

In contrast to the multigrid method, handling labeled vertices is simple. The equations do not have to be rearranged but the labeled vertices (pixels) are simply fixed and masked as unalterable using the z-buffer. Therefore, we initialize the z-buffer by setting 1 were updates are allowed and 0 were updates are forbidden. The z-buffer is updated every time the labels are changed. This method is very efficient if the number of labeled pixels is large; the hardware accelerated early-z test discards pixels failing the z-test before entering the rest of the pipeline.

As mentioned before four systems (colors) are processed simultaneously using RGBA textures. Once three systems have converged the solution to the remaining system is calculated trivially since the probability in each pixel sums to 1. This provides typically a huge speedup since one color is always used for the background and the rest for the foreground labels. The systems of the foreground labels usually converge relatively fast since the foreground objects are small compared to the size of the image. Thus, three systems converge fast and the remaining system is computed directly from the other systems.

Another advantage of using the GPU to solve the systems of linear equations is that it is possible to visualize all intermediate vectors of any solution algorithm in real-time. This kind of *progressive visualization* is especially useful with the random walker since it allows the user to watch the probabilities converging to the final solution. Further, we integrated fully interactive label editing enabling the user to add and remove labels of any color while the conjugate gradient solver is running. The right hand side vector is updated with the new label placement instantaneously. This turned out to be a very useful feature since the user can watch the progress of the random walks and interactively steer the solver by drawing additional labels if the random walker runs into an undesired direction.

Finally, we discuss the memory consumption of the conjugate gradient random

walker implementation. At the time when we implemented the random walker a typical graphics card had 256 MB of RAM. At the time of writing this thesis the average amount of memory is about 640 MB but 1,5 GB boards are also available. The amount of required memory is computed with $(n^d \cdot K \cdot \text{sizeof(float)} \cdot 7)$, where d is the number of dimensions and n is the number of pixels in all dimensions. Totally, there are 7 texture sets required to store the matrix, all conjugate gradient vectors plus intermediate vectors. A 128^3 voxel volume can be segmented with 256 MB of RAM. About 2 GB is sufficient to segment 256^3 volumes.

In the following section we validate properties of the random walker algorithm and present timings and examples.

6.3.3 Validation

The theoretical properties of the random walker algorithm can be found in [LG04]. It has been shown that the random walker

- Is robust to noise.
- Finds reasonable boundaries in low-contrast and homogeneous regions.
- Computes good results with many imaging modalities.

In this section, we discuss properties of the algorithm [GSAW05b]. A good algorithm computes good segmentations with just a few labeled pixels. At the same time, the label placement should not affect the result too much. Specifically, we want to find answers to the following questions:

1. The influence of label placement to the segmentation result.
2. The influence of the amount of labeled pixels to the segmentation result.
3. The average time to label pixels by the user and runtime of the algorithm using the GPU.

6.3.3.1 Label Placement Accuracy

A good segmentation algorithm returns similar results if the labeled pixels remain within the segmented object. We tested the robustness of the random walker algorithm with respect to label placement using the following experiment. The left column of Figure 6.2 shows the data sets. From top to bottom the imaging modalities are CT, MR, PET, Ultrasound, and CT in 3D. First, we created a reference segmentation on

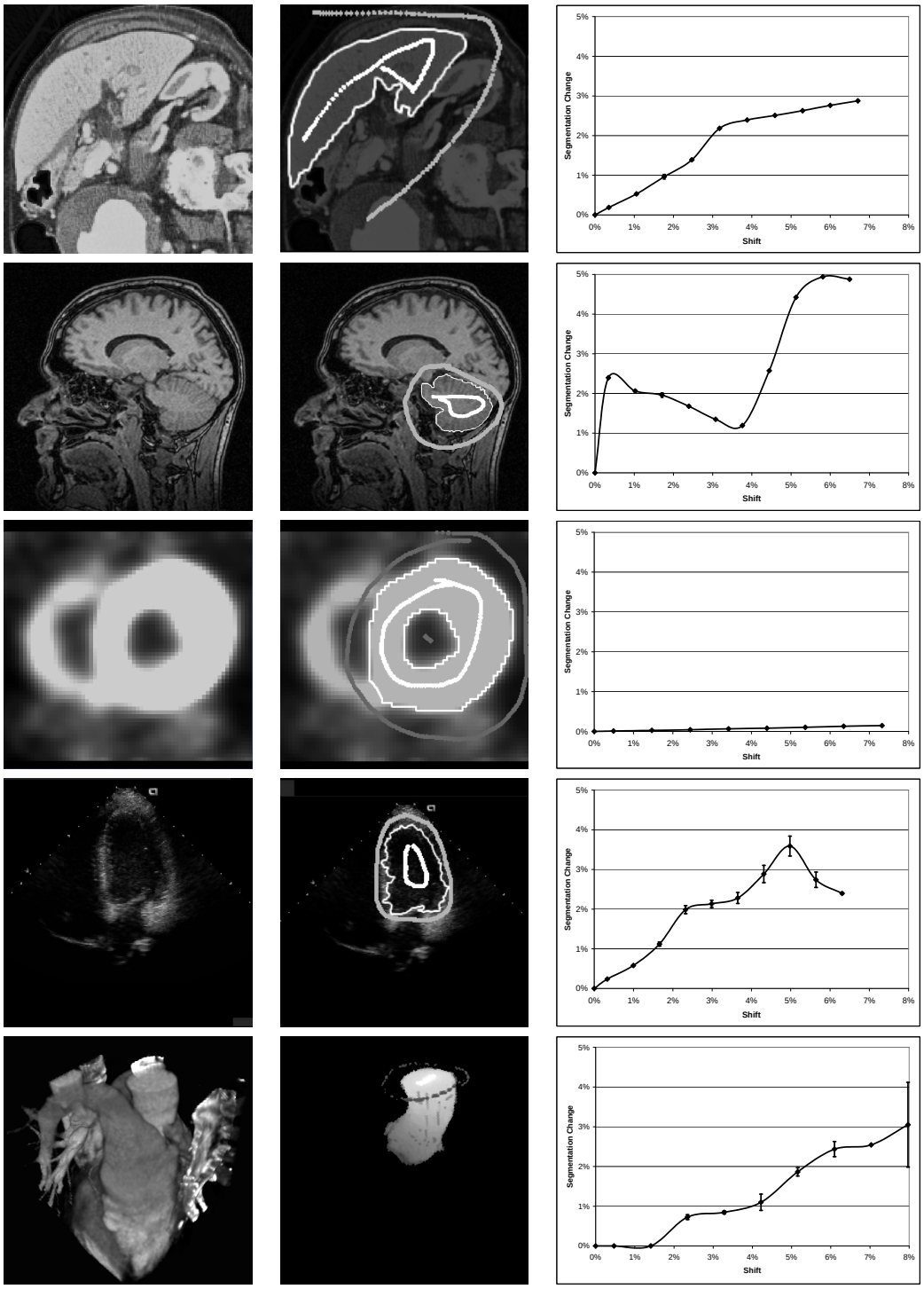


Figure 6.2: *The left column shows the original data sets, the middle column shows the reference labels and segmentation, and the right column shows the experimental results.*

all data sets as shown in the middle column. Then, we shifted all foreground seeds (shown in white) into one random direction with one random magnitude, but did not allow shifting foreground labels to the background region. Next, the random walker algorithm computed a new segmentation using the shifted seeds and determined the number of pixels where the segmentation result differs to the reference segmentation. We computed the ratio of flipped segmentation pixels to the total number of foreground pixels in the reference segmentation. This experiments was repeated 1000 times. The right column of Figure 6.2 shows the results. The x-axis of the diagrams indicates the ratio of shift magnitude to the image resolution. The y-axis represents the ratio of pixels with flipped segmentation assignment to the number of foreground pixels in the reference segmentation. The diagrams show the mean as plotted lines and the error bars illustrated the one standard errors of the means.

The diagrams show that a larger shift magnitude result also in a larger change in the segmentation. But as one can see in the scales of the diagrams the segmentation result never changes more than 5%. The exact numbers also depend on the original input images. It is obvious that the random walker is more likely to stay in regions surrounded by sharp boundaries such as the PET image. In contrast, low-contrast boundaries in images are more sensible to the location of the labeled pixels as can be seen in the MR image. However, since the maximum segmentation change is always below 5% we conclude the algorithm to be very insensitive to label placement.

6.3.3.2 Amount of Labels

Concerning how many labeled pixels are actually necessary to get a good segmentation result we conducted the following experiment. We reused the reference segmentation created for the last experiment and labeled each pixel of the foreground object as foreground. This corresponds to a purely manual segmentation. All remaining pixels were assigned the background label. By employing the morphological erosion operator, the amount of foreground and background labels were removed successively and we computed a new segmentation after each erosion. We compared each new segmentation to our reference segmentation. Figure 6.3 shows the results from our experiments. The x-axis show the ratio of removed labels to a fully labeled image and the y-axis shows the ratio of the number of pixels with switched labels to the total number of pixels with foreground labels.

Our results indicate that the random walker algorithm is very robust to the number of labeled pixels. Note that the segmentation changes dramatically only if about 70% of the labels in the reference segmentation have been removed.

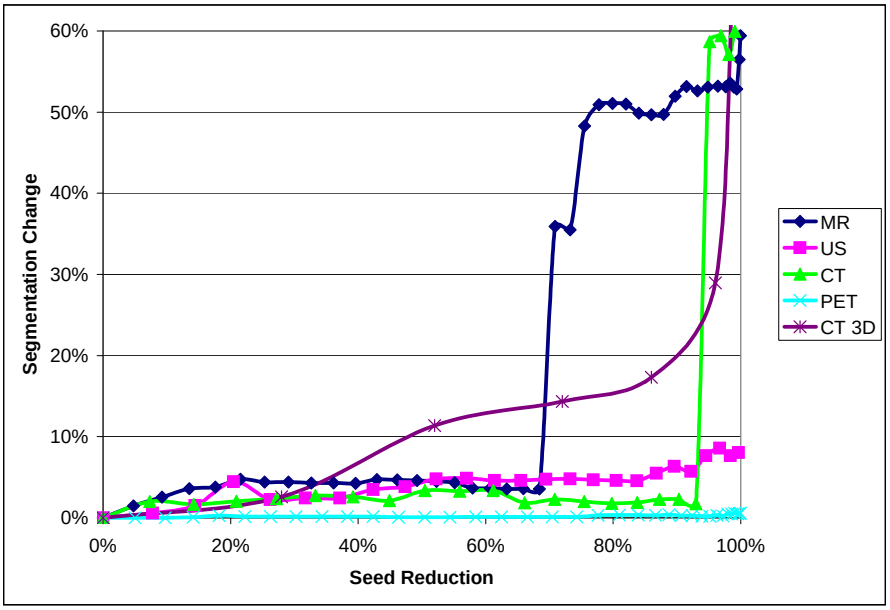


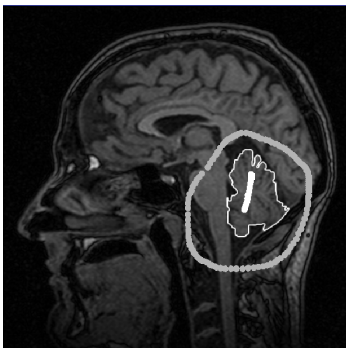
Figure 6.3: *We used the same set of medical images as in Figure 6.2. Here, we labeled all pixels of the foreground reference segmentation with foreground labels and vice versa. Using the morphological erosion operator one shell after another was removed and the random walker algorithm computed a new segmentation using the reduced labeled pixels. We repeated this process until one of the two label groups was removed. The x-axis shows the ratio of remaining foreground labels to reference foreground labels. As in Figure 6.2, the y-axis shows the amount of pixels that have switched the label assigned by the random walker algorithm.*

6.3.3.3 Timings

In our final experiment we have measured the time a user needs to label pixels and the time the random walker algorithm takes to compute the segmentation result. Figure 6.4 shows the timings divided into three groups. The user time, the time our CPU implementation took, and the time our GPU implementation took. Again, we conducted this experiment using a multitude of images from different imaging modalities. The user performing the pixel labeling had little experience with our tool. Typically, the user time varies between 2 and 5 seconds while the CPU computed between 3 to 83 seconds. The best computer performance was achieved by the graphics card with timings between 0.3 and 1.5 seconds. Segmenting a 3D volume (not shown) takes about 5 seconds label placement by the user, 35 seconds on the CPU and 1 second on the GPU. The timings were measured on a Pentium 4 2.8 GHz with an ATI Radeon X800 XT graphics card.

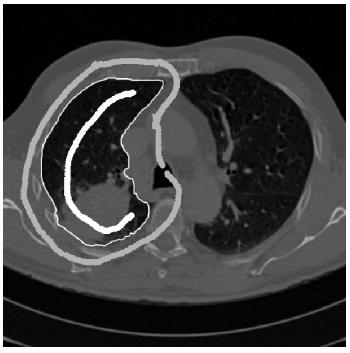
Brain tumor
User: 2s
CPU: 10s
GPU: 0.7s

Brain
User: 2s
CPU: 6s
GPU: 1.5s



Cerebellum
User: 2s
CPU: 83s
GPU: 0.3s

Lung tumor
User: 3s
CPU: 3s
GPU: 0.9s



Lung with tumor
User: 5s
CPU: 23s
GPU: 1.3s

Left atrium
User: 4s
CPU: 3s
GPU: 0.8s

Figure 6.4: *We have measured both the time a user needs to label pixels and the time the computer takes to compute the segmentation. By user time we refer to the label placement for the random walker algorithm. It also includes correction made by the user, if the result was not satisfying. We compare the timings of our random walker CPU and our GPU implementation based on the user-provided labels. The total time is either user + CPU or user + GPU.*

Chapter 7

Medical Image Registration

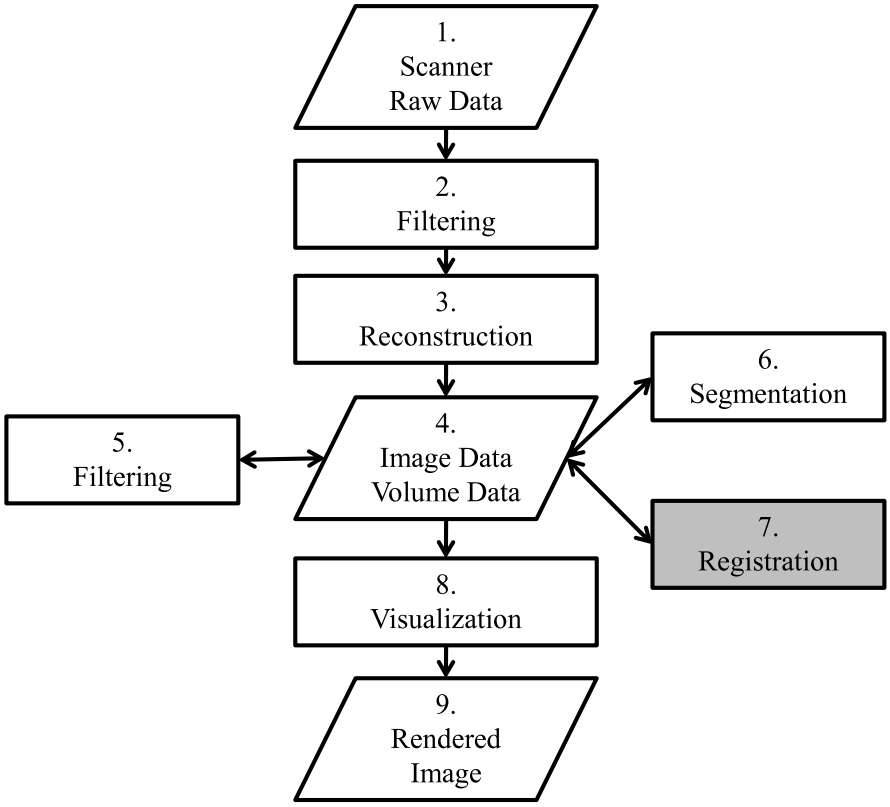


Figure 7.1: *Medical image registration aligns a set of images into a common coordinate system.*

Physicians often take several images of a patient using different devices such as ultrasound or MR scanners in order to visualize a multitude of aspects of an organ. One scanning device is good at visualizing soft tissue, others provide better images of brain or muscle activity. The images are not only different in terms of image modality, or

camera position, but are acquired at different times, too. Although patients try to hold still, the inner organs will certainly move due to heart beat, blood flow, etc. The task of medical image registration is to (automatically) align two images by transforming one image so that it matches the other one as closely as possible. Alignment is essentially image warping according to a displacement field generated by an image registration algorithm. Registered images allow the physicians to study a multitude of aspects using a fused visualization.

The challenge for the computer scientists is to build registration algorithm that generate the displacement field. The number of approaches is vast. An overview can be found in [MV98, HHH01, HBHH01]. Indirect approaches such as *feature*-based registration methods do not register the image data, but extract geometric features such as curves or surfaces that are aligned using a set of corresponding points in both images. In contrast, *image*-based registration uses the image information directly to register images. Here, image information does not exclude frequency domain or Eigenvalue decomposition information. The algorithm we present in this chapter is image-based.

Further, image-based registration algorithms rely on the similarity of the two images. Similarity measures between two images can only be computed correctly, if both images are present in the same modality domain including transfer function settings. That is, before a similarity measure can be computed the images must be transformed into the same image domain. Our algorithm does not deal with this problem assuming the images are already comparable.

Statistical similarity measures include the sum of squared differences, cross correlation (see Section 7.3.1), or mutual information just to name a few. These statistical measures per se do not exploit spatial information, thus assuming the image consists of uncorrelated random values. However, there are numerous techniques to integrate spatial information into statistical similarity measures such as distance weighting similarity. Also very intuitive is the division of the image into disjoint (or regularly overlapping) windows in which the measure is computed independently.

From the statistical similarity measure, a single displacement vector or vector field is generated. This determines the two most common types of spatial image warping: rigid and non-rigid transformation. Rigid transformation applies the same transformation to each pixel of an image. A rigid transformation is an arbitrary multi-combination of translation, rotation, and scaling. It is obviously limited in its applications, however, usually a lot faster to compute.

A typical rigid registration algorithm yielding a rigid transformation is *principle component analysis* registration. The covariance matrix of each image is computed by

weighting each pixel position with its intensity value. The mean translation vectors describe the relation between the image centers. An Eigenvector analysis of the covariance matrix yields the relative rotation and scaling of both images. Using a rigid transformation composed of the translation, rotation, and scaling incorporated from both images matches the images best from the perspective of the entire images.

By far more flexible are non-rigid transformation methods. Here, a (potentially) unique transformation is applied to each pixel. This allows accurate deformations within the image, for example heart pumping. An important feature most non-rigid registration algorithms rely on is the *regularizer*, also sometimes called *smoothing constraint*. The regularizer enforces neighboring vectors to point into a similar direction and have a similar length. This way, the displacement field is prevented from pointing to uncorrelated directions. The regularization can be enforces explicitly [AWS00, RSW99, SKP05] or incorporated as an additional constraint by design.

The algorithm we present in this chapter is a non-rigid algorithm, that further incorporates real-world physical deformation properties such as material stiffness and a realistic force dynamic range. Since the literature on medical image registration is too vast, we limit ourselves to publications in the area of non-rigid physically-based algorithms (see Section 7.1). Our algorithm is composed of several exchangeable components that we discuss first:

- The theory of elasticity along with implementation strategies and applications is presented in Section 7.2.
- Various displacement estimation algorithms are presented in Section 7.3 including cross-correlation and optical flow.

Finally, we present our novel registration algorithm in Section 7.4 composed of the previously described algorithms.

7.1 Related Work

The field of medical image registration algorithms is huge. Comprehensive surveys can be found in [MV98, ZF03, Mod04]. As already stated before, we focus this overview about methods related to our method, namely, physically-based non-rigid registration.

Variational approach have gained much attention in the last couple of years. Nonlinear registration problems are expressed as variational partial differential equations approximating the problem [CDR02]. Also, much effort has been put into advanced regularizers that adapt the amount of smoothing locally instead of globally. Since it

is often unrealistic to smooth over tissue boundaries linear elastic and viscous fluid models have shown great potential to overcome this problem [CRM96, WS01, WS98, BWF+05]. Curvature-based constraints for non-linear registration problem have been discussed in [FM03, Hen06]. Only a few approaches have tried to simulate heterogeneous tissue stiffness realistically [DDLD04, KFF05, JHHK06].

Concerning the discretization methods, central difference approaches are used widely. To the best of our knowledge, finite elements approaches have rarely been used due to the performance impact [HRS+99]. However, the more accurate models can be created and computed using finite elements especially due to the simple assignment of tissue stiffness. Using efficient multigrid solvers [GW05a] the performance bottleneck can be eliminated allowing an efficient and accurate non-rigid registration algorithm.

7.2 Physically-correct Deformation

A crucial part of our registration algorithm is the physically-correct deformation. We begin this section with a brief description of the elasticity theory (Section 7.2.1) followed by the discretization using finite elements (Section 7.2.2). In Section 7.2.3 we show several ways to assign a unique stiffness to each finite element. Finally, implementation details are presented in Section 7.2.4.

7.2.1 Elasticity Theory

This section gives a brief overview about the theory of elasticity. A more in depth description can be found here [GW05a, Geo07]. The theory of elasticity describes the deformation of an object in equilibrium. It can be formulated as follows [Bat02]

$$\Pi = \frac{1}{2}\int_\Omega \epsilon^T \sigma \,\mathrm{d}x - \int_\Omega g^T u \,\mathrm{d}x - \int_{\partial\Omega} f^T u \,\mathrm{d}x = \min \tag{7.1}$$

where $\Omega \in \mathcal{R}^n$ is the reference configuration in n dimensions. A displacement function $u : \Omega \to \mathcal{R}^n$ describes the displacement of every point of the reference configuration $\{x + u(x) | x \in \Omega\}$. The potential energy Π is composed of three terms: the elastic energy inside the body (first term), *volume forces* (second term) and *surface forces* (third term). The correct displacement u is found by minimizing Equation 7.1. Considering the first term, ϵ and σ depend on the *strain tensor* $\mathcal{E}$ and *stress tensor* Σ. The components of the strain tensor $\mathcal{E}$ are computed by

$$\mathcal{E}_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) + \frac{1}{2}\sum_{k=1}\frac{\partial u_k}{\partial x_i}\frac{\partial u_k}{\partial x_j}, \tag{7.2}$$

which describes the ratio of stretching along each side of an infinitesimal cube in all dimensions. The stress tensor Σ describes the forces arising from the displacement acting on cut planes through the body. It is coupled to the strain tensor via Hook's law as follows

$$\Sigma = \lambda\left(\sum_{i=1}\mathcal{E}_{ii}\right)\cdot I_{3,3} + 2\mu\mathcal{E}, \tag{7.3}$$

where λ and μ are Lamé coefficients derived from traversal contraction and longitudinal stretching.

For performance reasons the strain tensor is simplified (Cauchy strain tensor) by removing the quadratic terms

$$\mathcal{E}_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right). \tag{7.4}$$

Equation 7.4 is not rotationally invariant yielding artificial forces. A good trade-off between performance and accuracy is the *corotated strain* which is based on the idea of computing rotations of the material separately. For more details about it see [GW05a].

7.2.2 Discretization Using Finite Elements

Finite elements are simple geometric bodies with an easy to define interpolation function. This way, arbitrary points in the interior of the element can be interpolated with given values at the vertices of the elements. The interpolation function can be chosen linear or higher-order depending on the desired behavior. The deviation of the interpolation function for a variety of finite elements is described here [Geo07, Bat02]. We use a regular grid of finite elements to approximate the stiffness distribution of an medical image. Ideally, one pixel of the image is covered by two triangles of the simulation grid for maximum accuracy. In reality, we use a coarser grid for the sake of real-time performance. Usually, a grid of 128×128 double triangles is sufficient for accurate results. Although we haven't implemented our registration algorithm in 3D yet, the regular grid extends naturally using tetrahedra.

For the linear Cauchy strain, a stiffness matrix K_e is computed for each finite ele-

ment Ω due to the relationship

$$\frac{\partial}{\partial u_e}\frac{1}{2}\int_\Omega \epsilon^T \sigma \, \mathrm{d}x = K_e u_e, \tag{7.5}$$

where u_e is the local displacement vector of the finite element vertices. The finite element stiffness matrix K_e is derived from the shape matrix of each finite element [GW05a, Geo07]. All finite element stiffness matrices K_e are assembled into one system K sharing all common vertices among the finite elements. Finally, the system

$$Ku = f \tag{7.6}$$

with the external forces f on the right hand side can be solved. The forces f contain vertex, surface, and volume forces simultaneously acting on each vertex of the finite element grid. So far, Equation 7.6 describes the static equilibrium. The approach is extended to dynamic behavior following the Lagrangian equation of motion as follows

$$M\ddot{u} + C\dot{u} + Ku = f, \tag{7.7}$$

where M is the mass matrix, C is the damping matrix, and K is the stiffness matrix. Again, the matrices are composed of assembled per-element matrices. The element damping matrix C_e is computed by $C_e = \alpha M_e$, where α is a damping constant. The element mass matrix M_e is computed by integration over the finite element with its shape function.

The deviation of the systems using corotated strain, non-linear strain, higher-order finite elements as well as different kinds of time integration and boundary condition handling can be found in the literature [GW05a, Geo07, Bat02].

7.2.3 Stiffness Assignment

Depending on the modality the image was acquired, the intensity values of the image might or might not be a good indicator of tissue stiffness. CT images have the closest relationship between intensity and stiffness, but ambiguities remain. Other modalities have no intensity/stiffness relationship at all. In order to assign stiffness values to each of the finite elements we have implemented various methods.

1. *Manual assignment*: Given an appropriate user interface the user assigns stiffness manually element after element using a painting interface. This is very tedious work for the user since all elements must be assigned a stiffness individually.

Using stroke and brush tools the process is accelerated, but still requires a lot of time.

2. *Semi-automatic assignment*: The manual assignment is assisted by a segmentation algorithm that extracts objects in the image. The user assigns stiffness values per object instead of per finite element. We have integrated our Random Walker implementation from Chapter 6 into our tool chain for fast stiffness assignment. Since the Random Walker works on many medical image modalities we observe a significant speedup as compared to manual assignment.

3. *Automatic assignment*: As stated earlier, if the image modality shows relationship between intensity and stiffness, the stiffness assignment can be done automatically (or at least as an initial guess). That is, the intensity values are mapped to stiffness values using a transfer function. The transfer function might be a ramp allowing to emphasize parts of the intensity distribution while neglecting others. Of course, other functions are possible too.

We use a combination of the aforementioned methods. Starting with the automatic assignment using a transfer function, we refine the assignment using our segmentation and paint tools.

7.2.4 Implementation and Timings

In order to solve the static system using Cauchy strain efficiently, a multigrid solver is employed (see Section 3.2.4). Since the regular grid results in a regular band matrix in both 2D and 3D, our band matrix multigrid approach is used to solve the system (see Section 3.3).

Timings of our CPU solver implementation can be seen in Table 7.1. The CPU is a Intel Core 2 Duo 6600. A resolution of 128×128 double triangular finite elements provides real-time performance with about 30 updates a second. This is sufficient for instantaneous updates from the system. Even 256×256 provides acceptable interactive real-time performance.

grid size	128^2	256^2	512^2
time	28.8	116	478

Table 7.1: *The timings in milliseconds of physically-correct image deformation using different resolutions of finite element grid.*

Theoretically, our GPU implementation could be used to solve the system, too, but the floating precision is not sufficient for stable results. The floating point precision will certainly improvement in the upcoming years. All large vendors have already announced double precision GPUs.

7.2.5 Image Deformation Application

So far, we have a physically-correct deformation system that allows applying forces to the vertices of the grid. The external forces can be applied via user input, i.e. mouse clicks create a force field scaled by a Gauss kernel. Such a system is useful as a standalone application for manual image deformation and registration. Suppose a stiffness distribution has been created and copied to the finite element grid. Our application allows the user to drag and drop objects in a medical image. It guarantees deformation that is physically correct (no collision detection, though, so far).

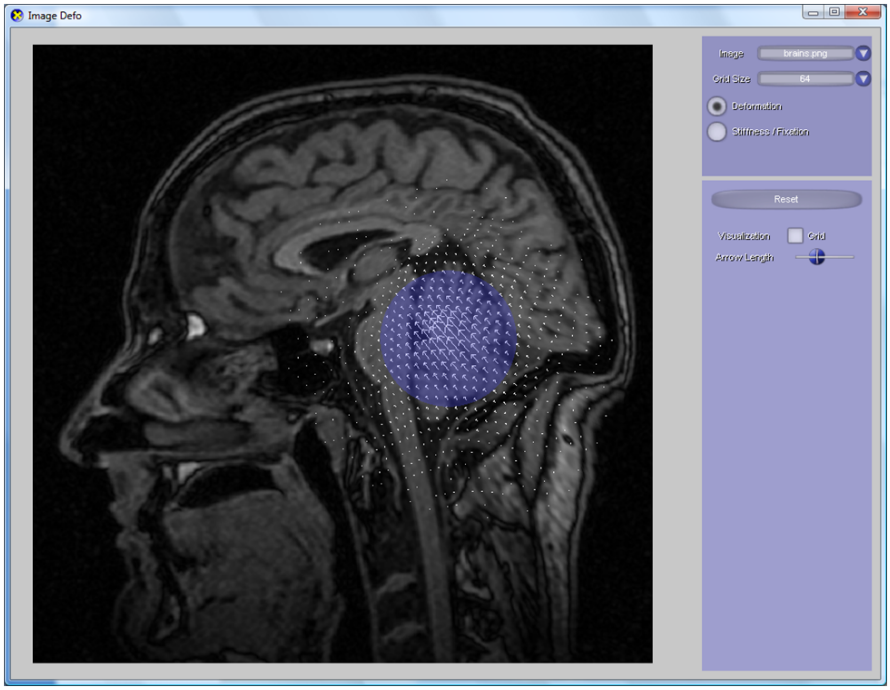


Figure 7.2: *Manual registration using mouse drags.*

Figure 7.2 shows the user interface in action. The blue circle indicates the Gauss kernel attached to the mouse cursor. By clicking the left mouse button, the user can drag the selected area to a new position. In this example, the cerebellum was assigned hard stiffness while everything else was assigned soft stiffness for demonstration purposes. This way, the cerebellum is stiffly moved to a new position while the background

deforms. The white arrows visualize the force field showing the Gauss distribution. In the following section, we discuss displacement estimation algorithm in order to automatically find a force field that matches one image to another one.

7.3 Displacement Estimation

Displacement estimation algorithms allow computing a vector field based on two images or volumes. This vector field describes a deformation in order to warp the first image to the second one. There are many well-known methods available today. We briefly describe the theory and the GPU implementation of the most common ones, normalized cross-correlation, image gradient and optical flow. All these algorithms are based on local or global intensity distributions over the medical images.

Let us fix our notation for the displacement estimation algorithms. We define all notations in 2D, but they can be extended to 3D analogously. The two images are denoted as $I(x, y) \in \mathcal{R}^{n \times n}$ and $J(x, y) \in \mathcal{R}^{n \times n}$, where image I is displaced so that it matches image J. The resulting displacement field $u(x, y) \in \mathcal{R}^{l \times l}$, where l is the resolution of the displacement field. For many algorithms, l can be chosen to balance performance and accuracy. Often l is chosen smaller than the image size n, sometimes $l > n$ if sub pixel displacement is taken into account.

As we discuss later in Section 7.4 in a more detailed way, the resulting displacement fields from displacement estimation algorithms are not physically correct deformations. All algorithm are only based on pixel intensities with no unique physical correspondence. Further, the smoothing constraints do not preserve natural boundaries.

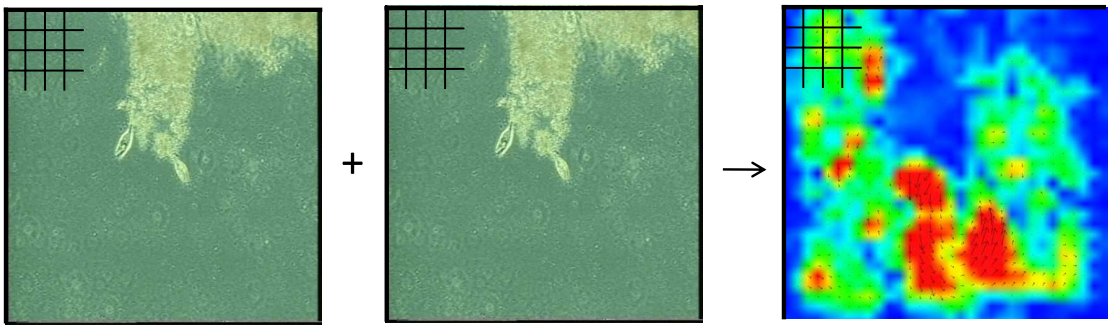


Figure 7.3: *A displacement field is reconstructed from two images I and J.*

In our registration algorithm we use a displacement field as an estimator and correct it physically using our physically-correct deformation method. Thus, we apply the predicted displacement field as external forces to the simulation grid. The elas-

ticity simulation returns – given a good stiffness distribution – a physically plausible deformation. This way, we automate the process of manually finding external forces.

7.3.1 Normalized Cross-Correlation

Normalized cross-correlation divides both images into *interrogation* windows of the same size (see Figure 7.3). All interrogation windows either do not overlap or overlap regularly. We denote the size of an interrogation window with m (in all dimensions). Typically, an interrogation window size is 32×32. The number of interrogation windows is denoted $k = n/m$. Note, that the same division pattern is used for both images.

Regardless of the algorithm, preprocessing of the interrogation window data in both images helps improving the displacement results. It is advantageous to subtract the mean values μ_I and μ_J from both windows in order to shift the intensity values to the common basis 0, where $\mu_I = \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} I(x+i, y+j)$ and μ_J analogously. If the result of the subtraction gets smaller than 0, we clamp it to 0. Implementation-wise, each image is stored in a texture. Using the avg-multi-reduction operator (see Section 2.3.3.2 and 2.3.3.4 for more details), the average value in all $k \times k$ interrogation windows is determined and stored in a $k \times k$ texture. In a subsequent rendering pass, the average value texture is subtracted from the original image pixel-wise using nearest-neighbor sampling and clamping to 0.

One application of the normalized cross-correlation is *particle image velocimetry*. This method was developed for the reconstruction of real-world vector fields, such as air, using particles released into the vector field. Using a high-speed camera and a laser light, the particles are photographed at fixed time intervals. The recorded sequence of two images is then used to reconstruct the original vector field from the particle positions in the images. Normalized cross-correlation is one algorithm for the vector field reconstruction. The method can easily be extended to record a set of consecutive images in order to reconstruct time-varying vector fields.

In the following, we discuss the spatial domain cross-correlation and the frequency domain cross-correlation.

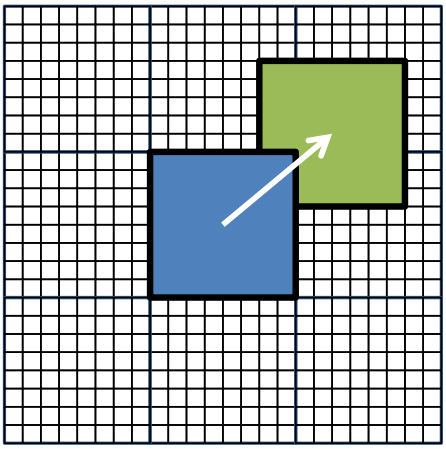


Figure 7.4: *The blue interrogation window from image I is cross-correlated to a green interrogation window of image J regarding a displacement vector (u, v) in white.*

7.3.1.1 Spatial Domain Cross-Correlation

Given a 2D displacement vector (u, v), the cross-correlation between image I and J can be computed as

$$C_{spatial}(u, v) = \sum_{i=-m/2}^{m/2} \sum_{j=-m/2}^{m/2} I(i, j) \times J(u + i, v + j), \tag{7.8}$$

if m is an even number.

In order to find the displacement vector for each interrogation window, the correlation for each displacement vector (u, v) living inside the *search* window s is computed. In Figure 7.4, the search window size is $s = 2m$, since the search window size is measured from the center of the interrogation windows. The search window size s can be chosen arbitrarily, but larger search windows account for a more global solution than small ones. Assumptions about the maximum displacement help optimizing the search window size, since the larger the search window the slower the computation. Finally, the displacement vector (u, v) with the largest cross-correlation in the search window is selected as the resulting displacement vector for each interrogation window.

The likelihood of false maxima can be reduced by normalizing Equation 7.8 with division by

$$\sqrt{\sum_{i=-m/2}^{m/2} \sum_{j=-m/2}^{m/2} (I(i, j) - \mu_I)} - \sqrt{\sum_{i=-m/2}^{m/2} \sum_{j=-m/2}^{m/2} (J(i + u, j + v) - \mu_J)}, \tag{7.9}$$

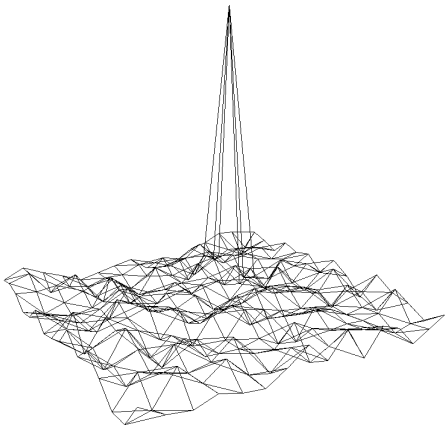


Figure 7.5: *A plot of the cross-correlation in a search window of images I and J.*

where μ_I and μ_J are the mean values of the current interrogation window of the images I and J.

The spatial domain cross-correlation has complexity $\mathcal{O}(k^d \cdot m^d \cdot s^d)$, where d is the number of dimensions, here $d = 2$. Assuming that the search window size s is of similar size as the interrogation window size m, the complexity of the spatial domain cross-correlation is $\mathcal{O}(k^d \cdot m^{d+d})$. In 2D, the dominant term is m, so the complexity is $\mathcal{O}(m^4)$.

A GPU implementation allocates a 2D texture with the size $(sk) \times (sk)$ to store all cross-correlations in the search window s for all interrogation windows k. A pixel shader program computes the current interrogation window index and search direction from the texture coordinates using modulo operations. Now, Equation 7.8 and 7.9 are calculated and the resulting cross-correlation written to the texture. Remember, the equations contain loops over 2D areas, which results in many texture fetches. Finally, a $\max_{pos}$ multi-reduction operator is used to find the position of the maximum correlation in each search window. See Sections 2.3.3.2 and 2.3.3.4 for more details on the reduction operator.

7.3.1.2 Frequency Domain Cross-Correlation

In order to reduce the amount of computation time, the cross-correlation can also be computed in frequency domain. Using the Wiener-Khinchin Theorem, the cross-correlation can be formulated in frequency domain as

$$C_{frequency}(w) = \max(\mathrm{F}^{-1}(\mathrm{F}(I_w) \cdot \mathrm{F}^*(J_w))), \tag{7.10}$$

where F denotes the Fourier transform, F^{-1} the inverse Fourier transform, F^* the complex conjugate of the Fourier transform, I_w and J_w denote an arbitrary pair of interrogation windows, and the $\max$ operator finds the maximum value (peak) in the window $m \times m$. The $\max$ operator is often extended to account for sub-pixel accuracy.

Given an interrogation window in image I, and an interrogation window at the same position in image J, the cross-correlation can be found by transforming both windows to the frequency domain, complex conjugate multiplying both, and transforming the result back to spatial domain, and, finally, searching for the maximum value.

The computational complexity of this algorithm is significantly lower than the spatial domain algorithm with complexity $\mathcal{O}(m^4)$. The frequency domain approach is dominated by the two Fourier transforms. A 2D FFT has the computational complexity $\mathcal{O}(m^2 \log m)$ if applied to a window of size $m \times m$. The complex conjugate multiplication and the search for the maximum are all $\mathcal{O}(m^2)$. So, the total complexity is $\mathcal{O}(m^2 \log m)$.

It is obvious that $C_{spatial} \neq C_{frequency}$. Overall the quality of the frequency domain approach does not match the spatial domain counterpart. The frequency domain approach relies on the periodicity assumption of the Fourier transform which is not fulfilled. Nevertheless, the frequency domain approach is much faster, and the results still convincing.

Now, we discuss the GPU implementation of this approach. We assume the interrogation window average has already been removed from each window using the method described in the beginning of Section 7.3.1. First, the two gray-scale images I and J are copied into one 4-component texture. This enables accessing both images at one position with one texture fetch. As Figure 7.6 depicts, the odd-components are left empty for the later storage of the imaginary parts of the complex Fourier transformation values.

Next, all interrogation windows of both images must be transformed to the frequency domain. It is very inefficient to transform each window separately, instead, we use the same idea as in multi reductions (see Section 2.3.3.4) by combining all single problems into a larger domain and perform all single steps for all windows at the same time. Referring to Section 4.1.2.2 describing the GPU implementation of the Fast Fourier Transform, we modify the FFT tables to account for a repeating window structure. Basically, a Fourier transform of an $m \times m$ interrogation window is required. We build this table and repeat it k times (for each window in a row). As shown in Figure 7.7, a windowed multi-FFT can be computed using $\log m$ matrix-vector products. Also note, that both images are stored in one texture are transformed simultaneously without

extra cost.

The cross-correlation is a complex conjugate multiplication of the previously transformed windowed images. Since two complex values are multiplied to one, two values are repacked in order to exploit the parallelism of the graphics hardware architecture. Using the same function as described above, the inverse windowed multi-FFT is executed in order to transform the frequency domain response back to spatial domain.

Finally, the displacement vector of each block is determined by finding the peak (maximum value) in each window. Once again, we use a $\max_{pos}$-multi-reduction operator (see Sections 2.3.3.2 and 2.3.3.4) to find the position of the maximum in each block.

For further peak accuracy improvements, sub-pixel displacement has to be taken into account. The neighborhood around the previously found peak contributes to the correct solution. We have implemented the most common fitting functions *center of mass* and *Gauss fit*. In 1D, the functions are as follows if v_i is the peak location.

- Center of mass: $\frac{v_{i-1}-v_{i+1}}{v_{i-1}+v_i+v_{i+1}}$

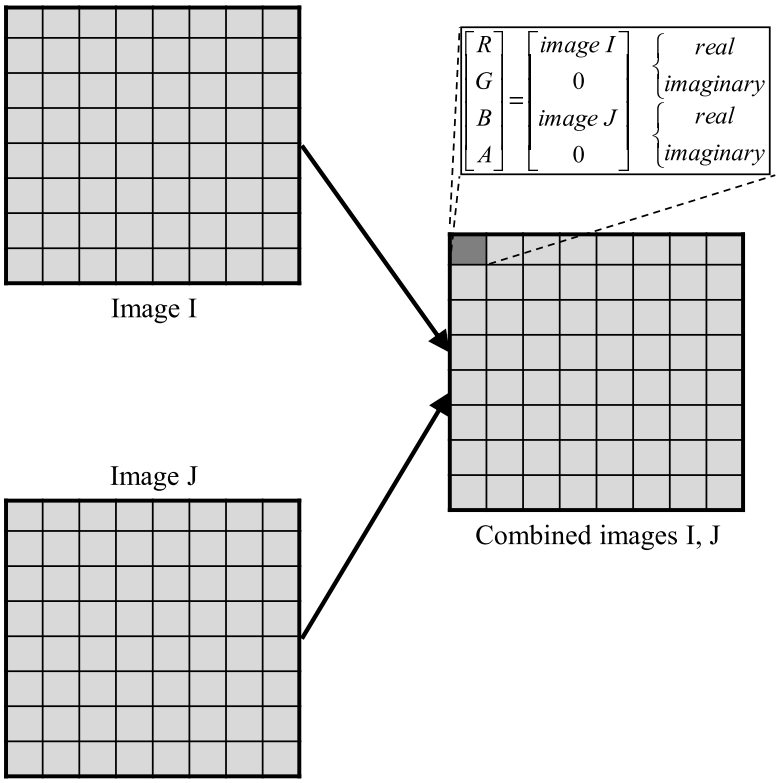


Figure 7.6: *The gray-valued images I and J are combined into one 4-component texture with space for imaginary parts.*

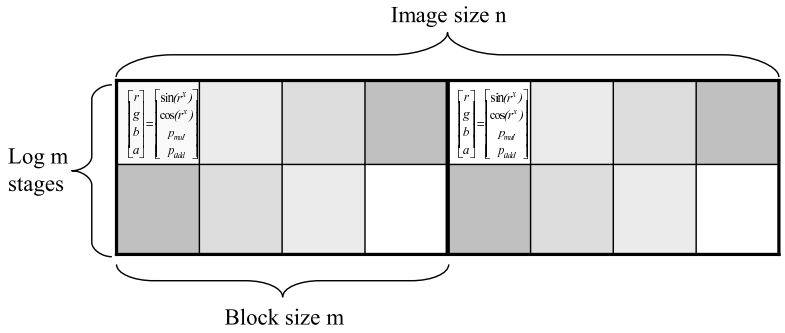


Figure 7.7: *All Fourier transforms of window size m are computed in parallel.*

- Gauss fit: $\frac{\ln v_{i-1} - \ln v_{i+1}}{2 \cdot (v_{i-1} + 2v_i + v_{i+1})}$

Once the displacement vector of all interrogation windows have been found the set of vectors forms a vector field. In the following, we optimize each vector with respect to its neighborhood. Typically, smooth vector fields are desirable.

An outlier vector is a vector that does not share similar orientation or length with regard to its neighbors. A simple method to correct outliers is replace them by the average vector in its neighborhood while the support can be chosen arbitrarily. This process is efficiently implemented using a pixel shader program. More sophisticated algorithms to correct outliers can, of course, also be implemented.

In order to increase the density of the vector field, additional cross-correlation passes with equally shifted interrogation windows can be computed. Now, the interrogation windows no longer are disjoint but overlap and special care must be taken at boundaries of the images. We clamp interrogation windows living partially outside the image domain to the border. Once all shifted vector fields have been computed, they are merged into a higher resolution field. Subsequent smoothing passes help reducing noticeable noise in the vector field.

We have measured the performance of our system on an ATI Radeon 9800 GPU and Pentium 4 3.0 GHz with 1 GB of RAM. About half of the time is consumed by the FFTs (forward and backward). A typical particle image velocimetry image has a resolution of 800×600. Since our implementation is restricted to power of two dimensions, we zero-pad the margins. Table 7.2 shows the performance of our system measured in milliseconds.

Even so the GPU is very outdated our system already provides performance fast enough for real-time video signals.

window/image	256^2	512^2	1024^2
8^2	6.62	20.8	76.9
16^2	7.9	29.4	111.1
32^2	9.5	35.7	142.8

Table 7.2: *GPU frequency domain cross-correlation performance in milliseconds.*

7.3.2 Intensity Gradient

Another very simple approach to find a (sparse) displacement field is the intensity gradient. The vector direction depends only on the spatial gradient of image I whereas the vector length depends on the difference between the two images I and J. The following equation is the intensity gradient

$$f(x, y) = (J(x, y) - I(x, y))\nabla I(x, y). \tag{7.11}$$

Although the intensity gradient is very simple to compute and the GPU implementation straightforward, the results are usually not satisfying. The gradient is discretized using central differences which lead to globally independent vectors. Therefore, the vector field is sparse and random.

7.3.3 Optical Flow

The third displacement estimation we review is the Horn-Schunck optical flow algorithm [HS81] and implementation strategies. In contrast to previous approaches, the Horn-Schunck method computes a dense and smooth vector field by enforcing two constraints. The first constraint forces the vectors to point to pixels with the same intensity as its origin. This constraint can never be solved uniquely since more often than not there are many pixels with the same intensity in its neighborhood. All vectors are uncorrelated and point in arbitrary directions. That is why, there is need for a second constraint, the regularizer or smoothing constraint. The smoothing constraint enforces neighboring vectors to be similar. That is, they point in a similar direction with a similar vector length. The amount of regularization is controlled by a free parameter and, therefore, can be adjusted as needed.

There is also another popular optical flow method called the Lucas-Kanade optical flow, which is in contrast to the Horn-Schunck optical flow a local method. Lucas-Kanade is faster to compute than Horn-Schunck but provides no globally-smoothed vector field. We leave discussions and implementations (also GPU implementations) to

the numerous publications published about this topic [BWF+05, BWKS06].

Since a temporal dependency dt between the two images I and J is required in the optical flow formulation, we rephrase our previously notation for this approach. Both images are addressed using the notation $E(x, y, t)$ that stores (continuous or discrete) intensity values at position (x, y) at time t. Assuming a discrete time difference dt, then let $E(x, y, t) = I(x, y)$ and $E(x, y, t + dt) = J(x, y)$. As any other method, the optical flow is not restricted to two dimensions. It generalizes to an arbitrary number of dimensions. For the sake of simplicity and clearness, we stick to the 2D case.

7.3.3.1 Theory

In the following, we review the two constraints ϵ_1 and ϵ_2 in a formal way and derive the Horn-Schunck optical flow system of linear equations. After that, we discuss various CPU and GPU implementations using different solvers from Chapter 3.

The first constraint states that the intensity of an image at time t at an arbitrary position (x, y) (within the image) is found in image $t + dt$ with a displacement vector (dx, dy)

$$E(x, y, t) = E(x + dx, v + dy, t + dt). \tag{7.12}$$

The spatial displacement vector (u, v) is computed by removing the temporal dependency as follows

$$u := \frac{dx}{dt} \tag{7.13}$$

$$v := \frac{dy}{dt} \tag{7.14}$$

With small enough spatial and temporal differences, the Taylor expansion yields

$$\epsilon_1 = \frac{\mathrm{d}E}{dx}u + \frac{\mathrm{d}E}{dy}v + \frac{\mathrm{d}E}{dt}, \tag{7.15}$$

which states the first constraint ϵ_1.

Horn and Schunck use the following estimations for the partial derivatives of E:

$$\begin{aligned}\frac{\mathrm{d}E}{dx} = \frac{1}{4}(&E(x,y+1,t)-E(x,y,t)+E(x+1,y+1,t)-E(x+1,y,t)+\\&+E(x,y+1,t+1)-E(x,y,t+1)+E(x+1,y+1,t+1)-E(x+1,y,t+1))\end{aligned} \tag{7.16}$$

$$\begin{aligned}\frac{\mathrm{d}E}{dy} = \frac{1}{4}(&E(x+1,y,t)-E(x,y,t)+E(x+1,y+1,t)-E(x,y+1,t)+\\&+(E(x+1,y,t+1)-E(x,y,t+1)+E(x+1,y+1,t+1)-E(x,y+1,t+1)\end{aligned} \tag{7.17}$$

$$\begin{aligned}\frac{\mathrm{d}E}{dt} = \frac{1}{4}(&E(x,y,t+1)-E(x,y,t)+E(x+1,y,t+1)-E(x,y+1,t)+\\&+(E(x,y+1,t+1)-E(x,y+1,t)+E(x+1,y+1,t+1)-E(x+1,y+1,t)\end{aligned} \tag{7.18}$$

The second constraint ϵ_2, the smoothness constraint, minimizes the derivatives of the displacement vector along the axes.

$$\epsilon_2 = \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 \tag{7.19}$$

Now, a vector field is computed that satisfies both constraints. In order to find such a vector field both constraints ϵ_1 and ϵ_2 must be minimized. This can be reformulated in a system of linear equations. We refer to [HS81] for the details about the transformation. Ultimately, the system looks as follows:

$$\begin{pmatrix} (\frac{\mathrm{d}E}{dx})^2 - \alpha\Delta & \frac{\mathrm{d}E}{dx}\frac{\mathrm{d}E}{dy} \\ \frac{\mathrm{d}E}{dx}\frac{\mathrm{d}E}{dy} & (\frac{\mathrm{d}E}{dy})^2 - \alpha\Delta \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -\frac{\mathrm{d}E}{dx}\frac{\mathrm{d}E}{dt} \\ -\frac{\mathrm{d}E}{dy}\frac{\mathrm{d}E}{dt} \end{pmatrix} \tag{7.20}$$

Note, that the free parameter α determines the amount of smoothing. A value of 0 disables the regularizer. Typical values range from 100 to 1000. Note, that the matrix has to be rebuild whenever the images, that is the image derivatives, or the free parameter α changes. The system has 7 diagonals in 2D and 11 in 3D. Figure 7.8 illustrates the matrix for a 2D image pair of 64×64 pixels. We have implemented two solvers for the optical flow system of linear equations: a conjugate gradient solver (see Section 3.2.3) and a multigrid solver (see Section 3.2.4).

7.3.3.2 Implementation

Our GPU-accelerated conjugate gradient solver uses the techniques described in Section 3.3. We do not store the matrix explicitly, but compute the matrix entries on-the-fly every time it is needed. This enables great flexibility if the images or the free parameter α change. We precompute the image derivatives once new images have been selected

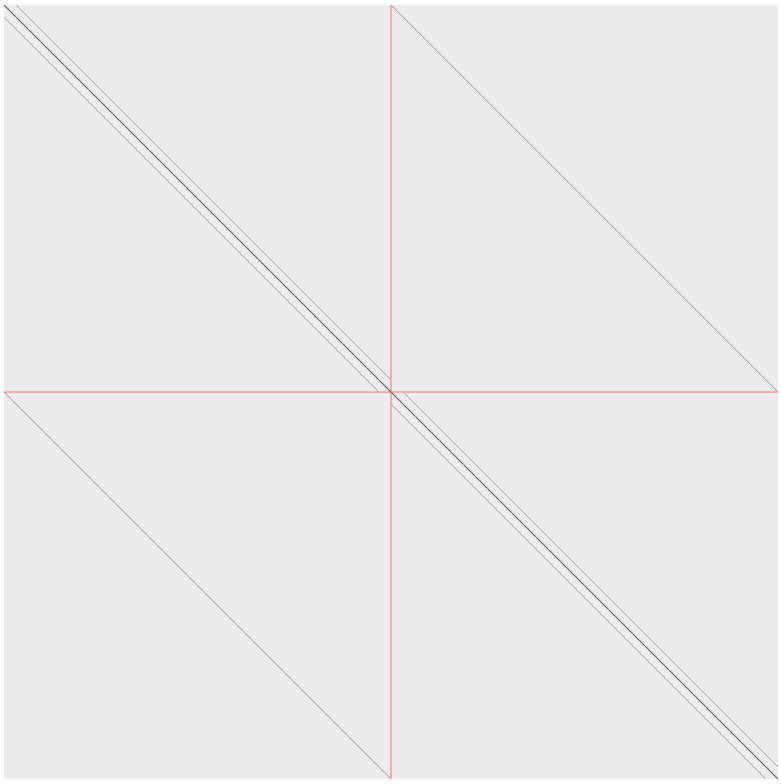


Figure 7.8: *An optical flow system matrix of a 2D image pair with* 64×64 *pixels. The red lines have been added to visualize the block structure of the matrix.*

by the user. This way, computing the matrix entries becomes only a few arithmetical instructions.

We have also used our multigrid solver to compare the timings. The implementation is following the details described in Section 3.2.4. The optical flow system matrix is not diagonally dominant. Therefore, a Jacobi relaxation as presmoothing and postsmoothing operator does not converge, theoretically. However, according to our tests it converges anyways. Table 7.3 shows our timings comparing conjugate gradient and multigrid using the CPU and the GPU.

	64^2	128^2	256^2	512^2
GPU conjugate gradient	35	65	856	-
CPU multigrid	25	102	345	1293

Table 7.3: *The performance of our optical flow implementations in milliseconds. All timings include matrix rebuild and hierarchy update for the multigrid solver.*

Figure 7.9: *Two images of the well-known Hamburg taxi image sequence. The taxi turning right and the car below is moving to the right.*

7.4 A Physically-based Registration Algorithm

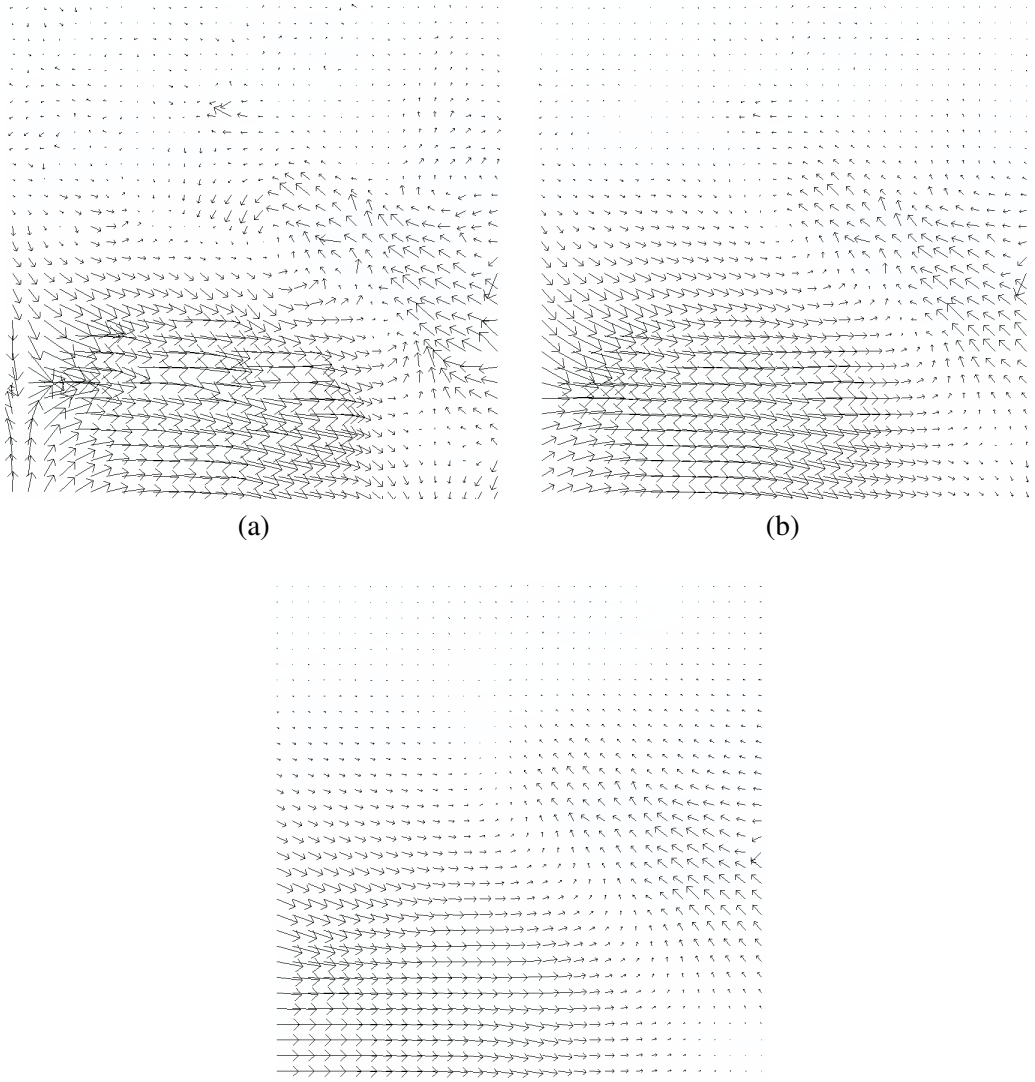
We present a novel interactive physically-based registration method. Figure 7.11 shows the most important steps of it.

We call the image to deform the *template* image, whereas the deformation target is called the *reference* image. Our method is a predictor/corrector approach. First a displacement field is estimated which is subsequently corrected according to physical deformation behavior. We use the algorithms presented previously as basis: A displacement estimation algorithm (Section 7.3) predicts a vector field which is correct by physically correct deformation (Section 7.2).

Before the physical deformation can be used, the template image is discretized into a regular grid of finite elements (triangles). Ideally, each finite element covers only a few pixels. Using stiffness assignment tools (see Section 7.2.3) an individual real-world stiffness value is assigned to each finite element. Next, a rigid alignment of the two images (template and reference) is computed to account for large scale shifts and rotations of the whole images. We employ a principle component analysis for this task.

The iterative main loop consists of the following steps:

1. *Displacement field estimation*: One of the algorithms described in Section 7.3 is used to estimate a vector field. Since all those algorithms are only intensity-based and potentially use a regularizer smearing the boundaries of tissue types, the vector field has to be correct with respect to physical properties.



(a) (b)

(c)

Figure 7.10: *Reconstructed vector fields with resolution* 32×32 *from the taxi sequence. The effect of different weights of the regularizer (smoother) are shown in images (a) through (c). Image (a) was computed with* $\alpha = 10$*, image (b) with* $\alpha = 100$*, and image (c) with* $\alpha = 1000$*.*

2. *Physical correction*: The estimated displacement field is applied as external forces to the simulation grid. For each finite element vertex we bilinearly interpolate the estimated displacement from the four closest optical flow vectors. Note, that forces are accumulated in all iterations, although the previous forces are damped. We compute the average force vector and subtract it from all forces in order to keep the mesh in place. Care must be taken with the scaling of the displacement vectors. The scaling can be a user-defined constant, or computed in each itera-

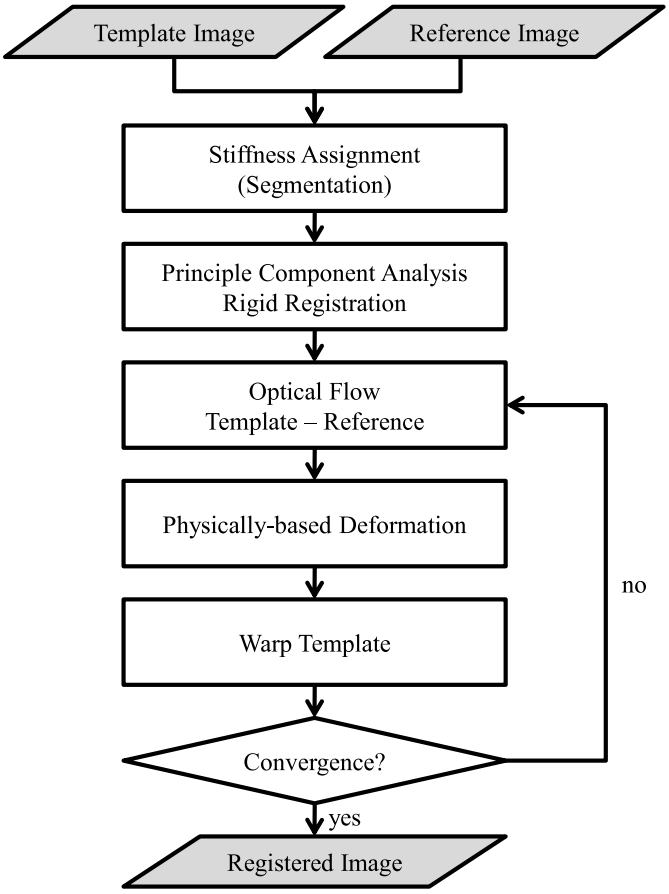


Figure 7.11: *A flow diagram of our registration algorithm.*

tion from average force strengths, for example. Now, the physical deformation algorithm is employed to compute a corresponding displacement field from the forces. The displacement field governs the laws of physics and respects the individual tissue stiffness. In our tests, we use the Cauchy strain tensor since we consider deformations to be small.

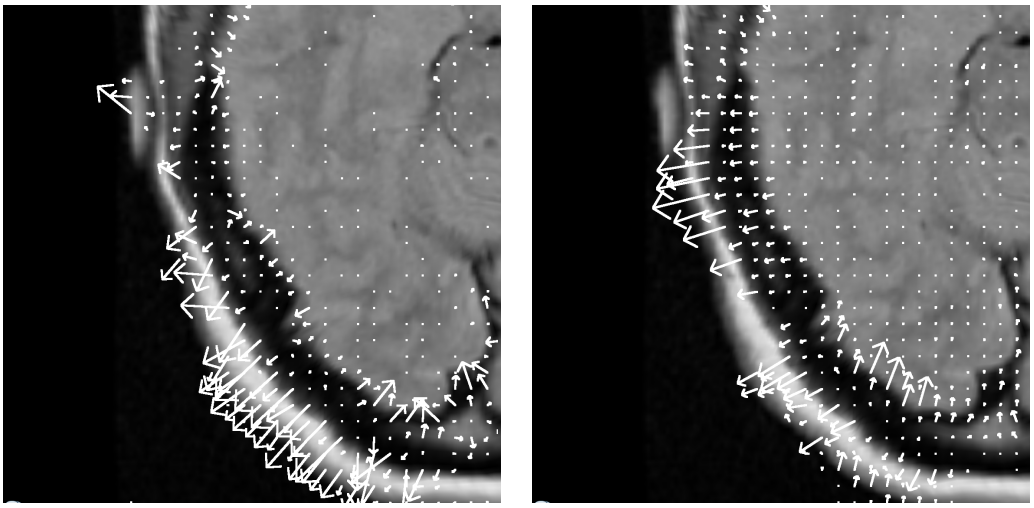
3. *Image warping*: The template image is warped according to the displacement field using the GPU. A shader program is used for this task and the result is written to a new texture. The new texture replaces the last template image in upcoming iterations.

4. *Convergence*: The quality of the warping is measured by an image similarity measure such as the sum of squared differences computed on the GPU. If the quality is not satisfactory yet, the warped template image is fed into the loop again starting from the displacement field estimation.

Once the difference between the images is below a threshold, the images are considered registered and the final deformed result is presented.

7.4.1 Results

We have used a computer with an Intel Core 2 Duo 6600 2.4 GHz equipped with an NVIDIA GeForce 8800 GTX for our experiments. Table 7.4 shows timings of the physical deformation in various resolutions for one iteration. Timings for the displacement estimators can be found throughout Section 7.3. For sake of performance we use a 128^2 grid in all our examples. We did not observe significant accuracy improvements when using finer grids.



(a) intensity gradient (b) optical flow

Figure 7.12: *The image gradient displacement field (a) is compared to the optical flow displacement field (b). The optical flow displacement is smoother than the one of the intensity gradient.*

A big advantage of our method that is requires no preprocessing and all parameters can be changed during runtime. For example, the stiffness distribution or scaling parameters can be changed anytime. Our experiments show that typically 30 - 50 iterations are required to converge averagely. The images 7.13, 7.14 and 7.15 where generated using our algorithm. The images show synthetic and real data images and accurate registration using our algorithm.

Grid size	128^2	256^2	512^2
Physical Deformation	28.8 ms	116 ms	478 ms

Table 7.4: *The performance of our registration algorithm in milliseconds for one iteration.*

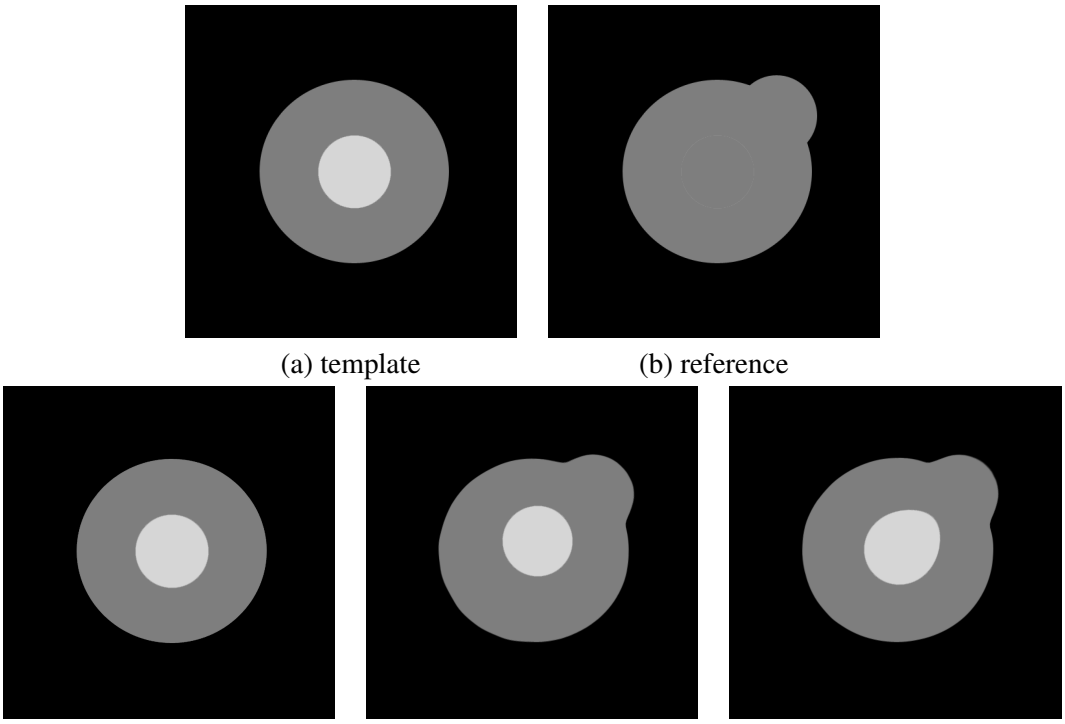
7.4.2 Discussion

We have validated our approach using the registration test framework by Schnabel et al. [STCS$^+$03]. We have create a series of image pairs and deformed them physically correct. Then, we used our algorithm to register the images again. Our algorithm is very accurate (sum of squared differences 10^{-3}) as Figure 7.16 shows. Even large deformations are no problem with our approach.

Our approach is different to Modersitzki [Mod04] is various aspects. Instead of using a finite difference discretization, we use finite elements. A better approximation of the partial derivatives is provided, thus the simulation is more accurate and stable. Furthermore, we use the optical flow algorithm instead of the image gradient as displacement field predictor. Although the optical flow algorithm has a regularizer which might contradict the physical corrector, overall we observe better results. Figure 7.12 compares the predicted displacement field of the intensity gradient and optical flow.

Both, the predictor and the corrector stages are black boxes that can be exchanged with other algorithms. For example, algorithms that take modality specific behaviors into account. The physical deformation algorithm is especially powerful as it can handle all real-world stiffness values from very soft to very hard. Although only Cauchy strain is implemented yet, other types of strains such as corotated / non-linear can be used as well. The system is solved implicitly and is unconditionally stable.

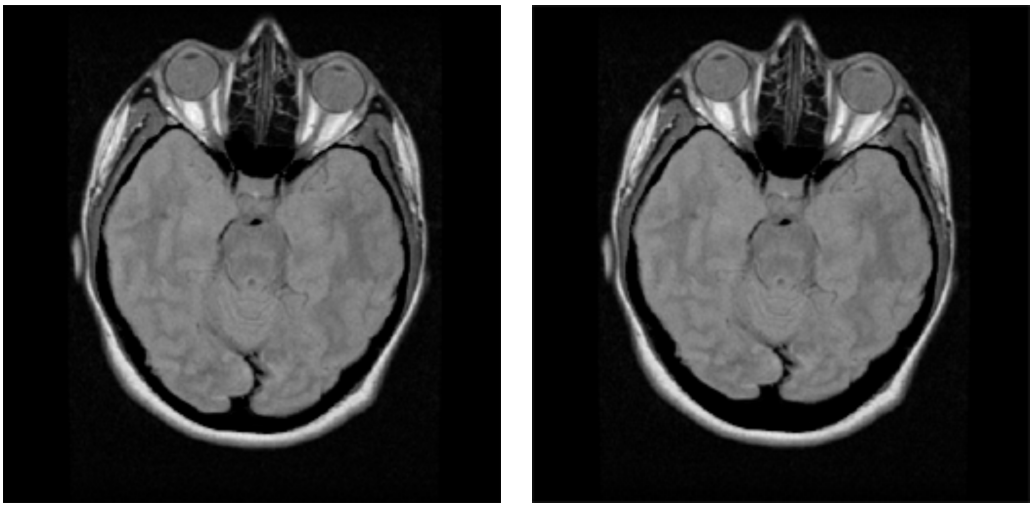
Future work includes the implementation in 3D using hexahedrons. The physical deformation implementation already supports 3D deformations and a 3D optical flow is very straightforward from the 2D version. Mixed boundary conditions are also of great interest. Here, displacements can be fixed on a set of vertices while simulating the rest further. Multi-modal registration problems can be computed once we integrate a mapping between the two modalities so that they can be compared by standard similarity measures.



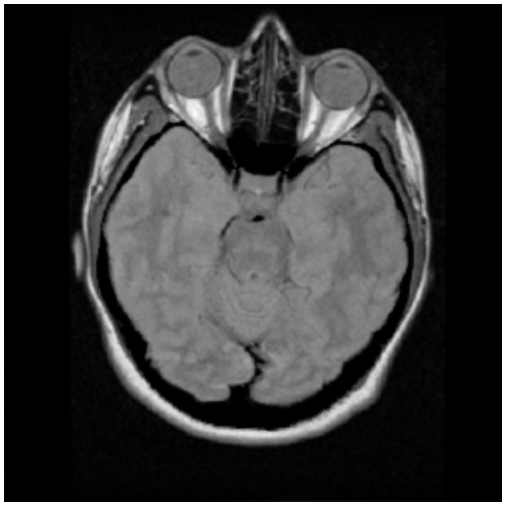
(a) template (b) reference

(c) all hard (d) mixed (e) all soft

Figure 7.13: *Two different tissue types are simulated by this images of a synthetic object. Starting from the template image (a) and reference image (b) three experiments are conducted with different stiffness distributions. First, both virtual tissue types are assigned the same hard stiffness. Image (c) shows the result that nothing moves out of the original shape. In the next experiment hard stiffness was assigned to the inner core, while soft stiffness was assigned in the darker ring. Image (d) shows the result computed by our algorithm showing the inner core moving to the upper right undeformed. Finally, both tissue types are assigned soft stiffness. The result can seen in image (e) where the outer matches the shape of the reference image while the inner core deforms.*

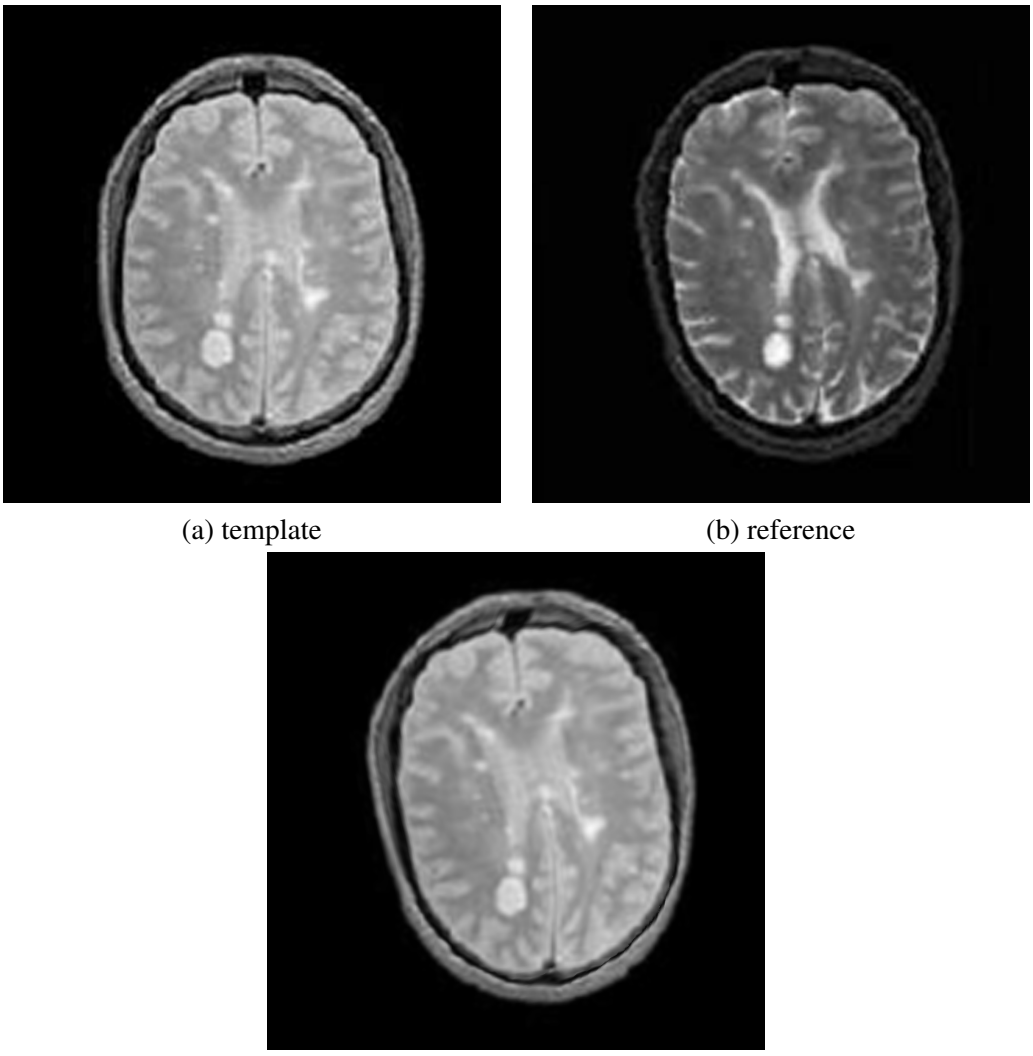


(a) template (b) reference



(c) registered

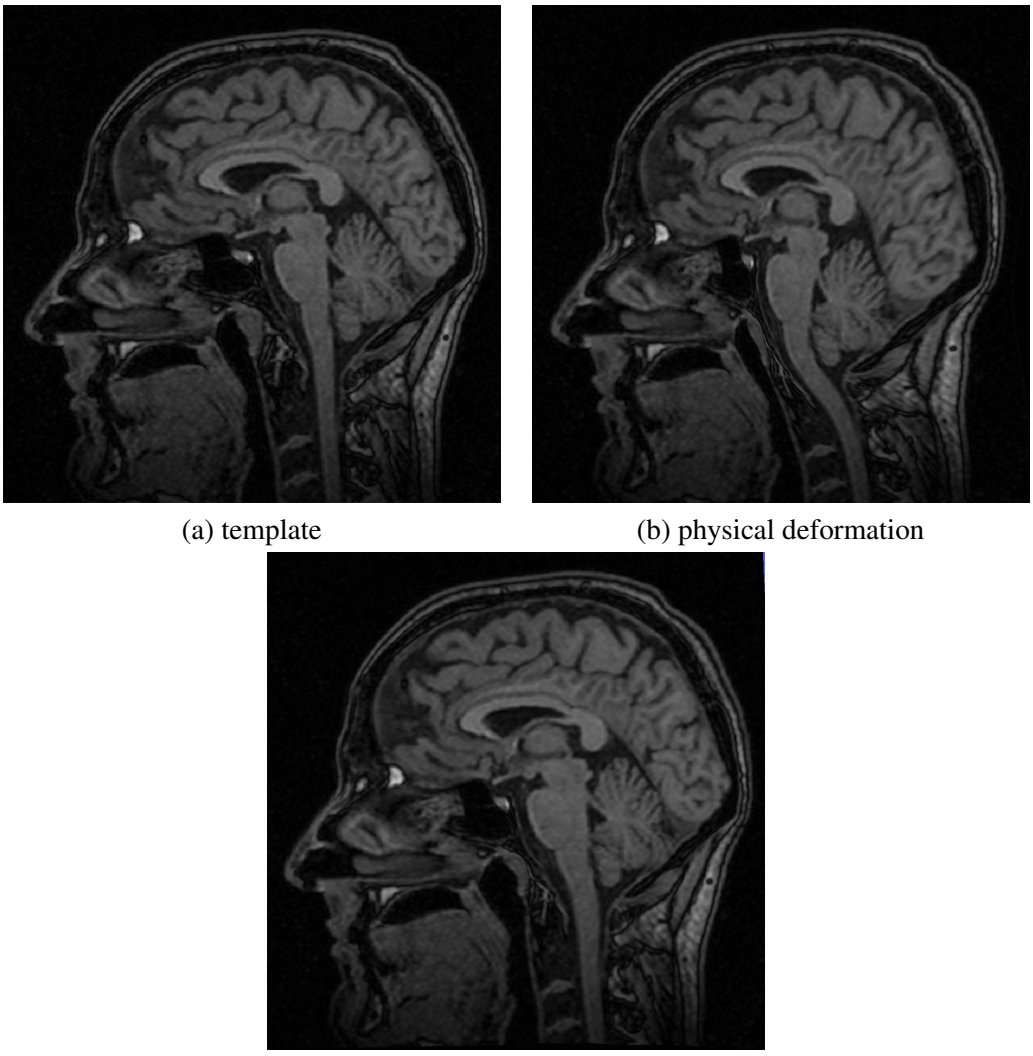
Figure 7.14: *Again, the template image is shown in image (a) and the reference image in (b). Here, the brain was shrunken manually in the reference image. The registration result of our algorithm is shown in image (c). This time, we used a* 256×256 *grid. Further, we specified the following stiffness parameters for various regions of the image: skull* 10^8*, grey matter* 10^6*, area between brain and skull* 10^4*. The results show how the skull remains stiff and the soft tissue deforms. The registration time was about 1 second using 5 iterations of the registration loop and rest sum of squared differences was* 10^2*.*



(a) template (b) reference

(c) registered

Figure 7.15: *There is a significant contrast difference of the template image (a) and the reference image (b). Image (c) shows the result of our algorithm. It ran 0.5 seconds on a* 128×128 *grid.*



(a) template (b) physical deformation

(c) registered

Figure 7.16: *A template image (a) is registered to a reference image (b) that was artificially deformed using a physically correct deformation algorithm. Image (c) shows the result of our registration algorithm.*

Chapter 8

Conclusion

In this section, we summarize the benefits and the contribution of this thesis followed by a section about future work.

8.1 Contribution

The contribution of this thesis is threefold. First, the GPU acceleration of systems of linear equations solvers are investigated (Chapter 3). Second, a large collection of medical imaging algorithms with GPU acceleration are discussed (Chapters 4 through 7). And third, a novel registration algorithm is introduced and evaluated (Chapter 7).

1. *GPU accelerated solvers for systems of linear equations*: Although GPU implementations of solvers for systems of linear equations have been investigated before, we contribute a much more general framework than any of our predecessors. Our fully automated GPU shader generator creates single pass GPU shaders of any kind of matrix-vector operator such as multiplications or Jacobi iterations. Using this generator complex solvers are implemented easily without additional manual work. Most of the previous work deals with the Jacobi method or the conjugate gradient method. Using our operators we present a novel GPU implementation of the multigrid method which shows great potential. Since solvers for systems of linear equations are extremely important for many tasks, we provide an important contribution not only to the applications discussed in this thesis but also for many others.

2. *A large collection of medical imaging algorithms*: We cover the most important areas in the field of medical image processing namely medical image filtering, reconstruction, segmentation, and registration. In all fields except registration

we picked algorithms relevant to Siemens products and describe the theory and GPU implementation in detail. Thus, many GPU implementations of algorithms are actually integrated in products and, therefore, are used by physicians in daily routine. We achieved at least a four times speedup. As some algorithms map to the GPU structures better than others, the performance we get varies, too. We found the biggest speedup in the filtered backprojection algorithm for MR images. Our GPU implementation is about 100 times faster than an optimized CPU backprojection.

3. *A novel physics-based registration algorithm*: Finally, we present a novel registration algorithm combining physical correctness with interactivity. Previous approaches were only able to favor one of the two. Instead, our approach allows simulating heterogeneous tissue stiffness distributions on a fine granular level while performing in real-time. This way, the image deformation obeys the laws of physics and cannot result in arbitrary synthetic deformations.

8.2 Future Work

Generally, all our GPU implementations would greatly benefit from higher floating-point precision. Even the DirectX 10 GPU generation does not comply to the IEEE floating-point standard in every respect although this should have no effect on the precision. However, double precision GPU processing is what many researchers desire. Fortunately, NVIDIA has already announced products capable of higher precision. It is unlikely that double precision will be enabled in mainstream products, though. Probably, double precision will be provided only in high-end general purpose GPU boards such as the NVIDIA's Tesla product line. However, it might eventually become available in mainstream product.

When we started implementing medical imaging algorithms using GPUs, there were only graphics APIs available for programming. If one would start doing a PhD about general-purpose GPU programming today, modern GPGPU languages should be considered extensively. Especially the language CUDA seems to be a raising star in the GPGPU community. Although easy to getting started, it is hard to gain optimal performance from it due to the different types of shared memories among the groups of multi-processors. Anyway, a huge advantage over the graphics APIs is the scattered writing capabilities allowing even more algorithms to be performed by GPUs. Again, it is most likely a matter of time until this feature will also appear in graphics APIs.

Above all technical improvements, the algorithms in the medical imaging pipeline are constantly being improved or replaced by novel algorithms so that novel GPU implementations are required, too. Furthermore, as GPUs become more powerful, the goal of creating a GPU-only processing pipeline will become more realistic. In this work we tried to reduce bus transfer times to the minimum but we did not succeed in keeping the data in GPU memory all the time. Of course, a lot more memory is also required to achieve this goal.

Conclusively, we achieve convincing results that are already used in products replacing older CPU implementations because our GPU implementations run on average an order of magnitude faster. However, there are many ways to further improve and extend our foundation of GPU-acceleration for medical imaging algorithms. We believe that this is just the beginning of a new era in medical imaging.

Bibliography

[AWS00] Luis Alvarez, Joachim Weickert, and Javier Sanchez, *Reliable estimation of dense optical flow fields with large displacements*, International Journal of Computer Vision **39** (2000), no. 1, 41–56.

[Bat02] Klaus-Jürgen Bathe, *Finite element procedures*, Prentice Hall, 2002.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroeder, *Sparse matrix solvers on the GPU: conjugate gradients and multigrid*, ACM Trans. Graph. **22** (2003), no. 3, 917–924.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick, *A multigrid tutorial, second edition*, SIAM, 2000.

[BHW+07] M.S. Burns, M. Haidacher, W. Wein, I. Viola, and E. Groeller, *Feature emphasis and contextual cutaways for multimodal medical visualization*, EuroVis 2007 Proceedings, May 2007.

[BKZ04] Matt A. Bernstein, Kevin F. King, and Xiaohong Joe Zhou, *Handbook of mri pulse sequences*, Elsevier Academic Press, Burlington, MA, USA, 2004.

[Bly06] David Blythe, *The direct3d 10 system*, SIGGRAPH '06: ACM SIGGRAPH 2006 Papers (New York, NY, USA), ACM, 2006, pp. 724–734.

[Bra77] Archi Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of Computation **31** (1977), no. 138, 333–390.

[Bri88] E. Oran Brigham, *The fast fourier transform and its applications*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[BWF+05] Andrés Bruhn, Joachim Weickert, Christian Feddern, Timo Kohlberger, and Christoph Schnörr, *Variational optical flow computation in real-time*, IEEE Transactions on Image Processing **14** (2005), 608–615.

[BWKS06] Andrés Bruhn, Joachim Weickert, Timo Kohlberger, and Christoph Schnörr, *A multigrid platform for real-time motion computation with discontinuity-preserving variational methods*, Int. J. Comput. Vision **70** (2006), no. 3, 257–277.

[CCGA07] Claudia Chevrefils, Farida Cheriet, Guy Grimard, and Carl-Eric Aubin, *Watershed segmentation of intervertebral disk and spinal canal from mri images.*, ICIAR (Mohamed S. Kamel and Aurlio C. Campilho, eds.), Lecture Notes in Computer Science, vol. 4633, Springer, 2007, pp. 1017–1027.

[CDR02] Ulrich Clarenz, Marc Droske, and Martin Rumpf, *Towards fast non-rigid registration*, DFG 1114 (2002).

[CRD07] D. Cremers, M. Rousson, and R. Deriche, *A review of statistical approaches to level set segmentation: integrating color, texture, motion and shape*, International Journal of Computer Vision **72** (2007), no. 2, 195–215.

[CRM96] Gary E. Christensen, Richard D. Rabbitt, and Michael I. Miller, *Deformable templates using large deformation kinematics.*, IEEE Transactions on Image Processing **5** (1996), no. 10, 1435–1447.

[dBP07] Johan de Bock and Wilfried Philips, *Line segment based watershed segmentation.*, MIRAGE (Andr Gagalowicz and Wilfried Philips, eds.), Lecture Notes in Computer Science, vol. 4418, Springer, 2007, pp. 579–586.

[DDLD04] Valerie Duay, Pierre-François D'Haese, Rui Li, and Benoit M. Dawant, *Non-rigid registration algorithm with spatially varying stiffness properties.*, ISBI, 2004, pp. 408–411.

[dWBM+00] Rik Van de Walle, Harrison H. Barrett, Kyle J. Myers, Maria I. Altbach, Bart Desplanques, Arthur F. Gmitro, Jan Cornelis, and Ignace Lemahieu, *Reconstruction of mr images from data acquired on a general nonregular grid by pseudoinverse calculation*, IEEE Transaction on Medical Imaging **19** (2000), no. 12, 1160–1167.

[DWD01] Brian Dale, Michael Wendt, and Jeffrey L. Duerk, *A rapid look-up table method for reconstructing mr images from arbitrary k-space trajectories.*, IEEE Transaction on Medical Imaging **20** (2001), no. 3, 207–217.

[Ebe01] David H. Eberly, *3d game engine design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[Far02] Gerald Farin, *Curves and surfaces for cagd: a practical guide*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[FJ98] Matteo Frigo and Steven G. Johnson, *FFTW: an adaptive software architecture for the FFT*, 1998, URL: `http://www.fftw.org`.

[FM03] Bernd Fischer and Jan Modersitzki, *Curvature based image registration*, Journal of Mathematical Imaging and Vision (JMIV) **18** (2003), no. 1, 81–85.

[Geo07] Joachim Georgii, *Real-time simulation and visualization of deformable objects*, Ph.D. thesis, Technische Universität München, 2007.

[GLW+05] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha, *Fast computation of database operations using graphics processors*, SIGGRAPH '05: ACM SIGGRAPH 2005 Courses (New York, NY, USA), ACM, 2005, p. 206.

[GPG] GPGPU, *General-purpose computation using graphics hardware*, http://www.gpgpu.org.

[Gra03] Kris Gray, *Microsoft directx 9 programmable graphics pipeline*, Microsoft Press, Redmond, WA, USA, 2003.

[GSAW05a] Leo Grady, Thomas Schiwietz, Shmuel Aharon, and Rüdiger Westermann, *Random walks for interactive alpha-matting*, Proceedings of the Fifth IASTED International Conference on Visualization, Imaging and Image Processing (Benidorm, Spain) (J. J. Villanueva, ed.), ACTA Press, Sept. 2005, pp. 423–429.

[GSAW05b] ———, *Random walks for interactive organ segmentation in two and three dimensions: Implementation and validation*, Proceedings of MICCAI 2005 (Palm Springs, CA) (J. Duncan and G. Gerig, eds.), LNCS 3750, no. 2, MICCAI Society, Springer, Oct. 2005, pp. 773–780.

[GST07] Dominik Göddeke, Robert Strzodka, and Stefan Turek, *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*, International Journal of Parallel, Emergent and Distributed Systems **22** (2007), no. 4, 221–256.

[GW05a] Joachim Georgii and Rüdiger Westermann, *Interactive simulation and rendering of heterogeneous deformable bodies*, Vision, Modeling and Visualization 2005, 2005, pp. 381–390.

[GW05b] ———, *Mass-spring systems on the gpu*, Simulation Modelling Practice and Theory **13** (2005), 693–702.

[Hac85] Wolfgang Hackbusch, *Multi-grid methods and applications*, Springer Series in Computational Mathematics, Springer, 1985.

[Har03] Mark Jason Harris, *Real-time cloud simulation and rendering*, Ph.D. thesis, The University of North Carolina at Chapel Hill, 2003, Director-Anselmo Lastra.

[HBHH01] D. Hill, P Batchelor, M. Holden, and D. Hawkes, *Medical image registration*, Phys. Med. Biol. **26** (2001), R1–R45.

[Hen06] Stefan Henn, *A full curvature based algorithm for image registration*, Journal of Mathematical Imaging and Vision (JMIV) **24** (2006), no. 2, 195–208.

[HHH01] J. V. Hajnal, D. L. G. Hill, and D. J. Hawkes, *Medical image registration*, CRC Press, 2001.

[HRS+99] Alexander Hagemann, Karl Rohr, H. Siegfried Stiehl, Uwe Spetzger, and Joachim M. Gilsbach, *A biomechanical model of the human head for elasitc registration of MR-images*, Bildverarbeitung fur die Medizin, 1999, pp. 44–48.

[HS81] Berthold K. P. Horn and Brian G. Schunck, *Determining optical flow*, Artificial Intelligence **17(1-3)** (1981), 185–203.

[JHHK06] Florian Jäger, Jingfeng Han, Joachim Hornegger, and Torsten Kuwert, *A variational approach to spatially dependent non-rigid registration*, Proceedings of the SPIE, Medical Image Processing, vol. 6144, 2006, pp. 860–869.

[JMNM91] John I. Jackson, Craig H. Meyer, Dwight G. Nishimura, and Albert Macovski, *Selection of a convolution function for fourier inversion using gridding*, IEEE Transaction on Medical Imaging **10** (1991), no. 3, 473–478.

[JvRLHK04] Thomas Jansen, Bartosz von Rymon-Lipinski, Nils Hanssen, and Erwin Keeve, *Fourier volume rendering on the gpu using a split-stream-fft*, VMV, 2004, pp. 395–403.

[KFF05] Sven Kabus, Astrid Franz, and Bernd Fischer, *On elastic image registration with varying material parameters*, Bildverarbeitung für die Medizin (BVM), Springer Verlag, 2005, pp. 330–334.

[KRF94] D. P. Koester, S. Ranka, and G. C. Fox, *A parallel gauss-seidel algorithm for sparse power system matrices*, Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing (New York, NY, USA), ACM, 1994, pp. 184–193.

[Krü06] Jens Krüger, *A gpu framework for interactive simulation and rendering of fluid effects*, Ph.D. thesis, Technische Universität München, http://mediatum2.ub.tum.de/node?id=604112, 2006.

[KSKW04] Jens Krüger, Thomas Schiwietz, Peter Kipfer, and Rüdiger Westermann, *Numerical simulations on PC graphics hardware*, ParSim 2004 (Special Session of EuroPVM/MPI 2004), 2004.

[KSW06] Jens Krüger, Jens Schneider, and Rüdiger Westermann, *ClearView: An interactive context preserving hotspot visualization technique*, IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006) **12** (2006), no. 5.

[KW03] Jens Krüger and Rüdiger Westermann, *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Transactions on Graphics (TOG) **22** (2003), no. 3, 908–916.

[KW05] Peter Kipfer and Rüdiger Westermann, *Improved GPU sorting*, GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Matt Pharr, ed.), Addison-Wesley, 2005, pp. 733–746.

[LCW03] Aaron E. Lefohn, Joshua E. Cates, and Ross T. Whitaker, *Interactive, gpu-based level sets for 3d segmentation.*, MICCAI (1) (Randy E. Ellis and Terry M. Peters, eds.), Lecture Notes in Computer Science, vol. 2878, Springer, 2003, pp. 564–572.

[LG04] Grady Leo and Funka-Lea Gareth, *Multi-label image segmentation for medical applications based on graph-theoretic electrical potentials*, Computer Vision and Mathematical Methods in Medical and Biomedical Image Analysis, ECCV 2004 Workshops CVAMIA and MMBIA (Prague, Czech Republic) (Milan Šonka, Ioannis A. Kakadiaris, and Jan Kybic, eds.), Lecture Notes in Computer Science, no. LNCS3117, Springer, May 2004, pp. 230–245.

[LL00] Zhi-Pei Liang and Paul C. Lauterbur, *Principles of magnetic resonance imaging*, IEEE Press, 2000.

[MA03] Kenneth Moreland and Edward Angel, *The FFT on a GPU*, HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (Aire-la-Ville, Switzerland, Switzerland), Eurographics Association, 2003, pp. 112–119.

[Mod04] Jan Modersitzki, *Numerical methods for image registration*, Oxford university press, New York, 2004.

[MV98] J.B.Antoine Maintz and Max A. Viergever, *A survey of medical image registration*, Medical Image Analysis **2** (1998), no. 1, 1–36.

[MYC95] K. Mueller, R. Yagel, and J. F. Cornhill, *Accelerating the anti-aliased algebraic reconstruction technique (ART) by table-based voxel backward projection*, Proceedings of IEEE EBMS 17th annual Conference, EMBC 95, 1995, p. 497.

[NK95] Predrag Neskovic and Benjamin B. Kimia, *Geometric smoothing of 3d surfaces and non-linear diffusion of 3d images*, 1995.

[NWvdB00] H. Nguyen, M. Worring, and R. van den Boomgaard, *Watersnakes: Energy-driven watershed segmentation*, 2000.

[OLG+05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell, *A survey of general-purpose computation on graphics hardware*, Eurographics 2005, State of the Art Reports, August 2005, pp. 21–51.

[O'S85] J.D. O'Sullivan, *Fast sinc function gridding algorithm for fourier inversion in computer tomography*, IEEE Transaction on Medical Imaging **M1-4** (1985), no. 4, 200–207.

[PL84] Doyle Peter and Snell Laurie, *Random walks and electric networks*, Carus mathematical monographs, no. 22, Mathematical Association of America, Washington, D.C., 1984.

[PM99] J. G. Pipe and P. Menon, *Sampling density compensation in mri: rationale and an iterative numerical solution*, Magnetic Resonance in Medicine **41** (1999), no. 1, 179–186.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in c: The art of scientific computing*, Cambridge University Press, New York, NY, USA, 1992.

[RD89] Courant R. and Hilbert D., *Methods of mathematical physics*, vol. 2, John Wiley and Sons, 1989.

[Ros06] Randi J. Rost, *Opengl(r) shading language (2nd edition)*, Addison-Wesley Professional, January 2006.

[RSW99] Esther Radmoser, Otmar Scherzer, and Joachim Weickert, *Scale-space properties of regularization methods*, Scale-Space Theories in Computer Vision, 1999, pp. 211–222.

[S.45] Kakutani S., *Markov processes and the Dirichlet problem*, Proc. Jap. Acad. **21** (1945), 227–233.

[SBMW07] Thomas Schiwietz, Supratik Bose, Johnathan Maltz, and Rüdiger Westermann, *A fast and high-quality cone beam reconstruction pipeline using the gpu*, Proceedings of SPIE Medical Imaging 2006 (San Diego, CA), SPIE, Feb. 2007.

[SCSW06] Thomas Schiwietz, Ti-Chiun Chang, Peter Speier, and Rüdiger Westermann, *Mr image reconstruction using the gpu*, Proceedings of SPIE Medical Imaging 2006 (San Diego, CA), SPIE, Feb. 2006.

[SCSW07] ———, *Gpu-accelerated mr image reconstruction from radial measurement lines*, Proceedings of ISMRM 2007: Workshop on non-Cartesian MRI, 2007.

[Set99] J. A. Sethian, *Level set methods and fast marching methods*, Cambridge University Press, 1999.

[SGW07] Thomas Schiwietz, Joachim Georgii, and Rüdiger Westermann, *Interactive model-based image registration*, VMV, 2007.

[She94] Jonathan R. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, Tech. report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[SKP05] Marius Staring, Stefan Klein, and Josien P.W. Pluim, *Nonrigid registration with adaptive, content-based filtering of the deformation field*, Proceedings of SPIE Medical Imaging: Image Processing, vol. 5747, February 2005, pp. 212 – 221.

[SM97] Jianbo Shi and Jitendra Malik, *Normalized cuts and image segmentation.*, CVPR, 1997, pp. 731–737.

[SN00] Hossein Sedarat and Dwight G. Nishimura, *On the optimality of the gridding reconstruction algorithm*, IEEE Transaction on Medical Imaging **19** (2000), no. 4, 306–317.

[SNF03] Bradley P. Sutton, Douglas C. Noll, and Jeffrey A. Fessler, *Fast, iterative image reconstruction for mri in the presence of field inhomogeneities*, IEEE Transaction on Medical Imaging **22** (2003), no. 2, 178–188.

[STCS+03] J. A. Schnabel, C. Tanner, A. D. Castellano-Smith, A. Degenhard, M. O. Leach, D. R. Hose, D. L. G. Hill, and D. J. Hawkes, *Validation of nonrigid image registration using finite element methods: Application to breast mr images*, IEEE Transactions on Medical Imaging **22** (2003), 238–247.

[SW04] Thomas Schiwietz and Rüdiger Westermann, *GPU-PIV*, VMV, 2004, pp. 151–158.

[TD05] Sumanaweera Thilaka and Lui Donald, *Medical image reconstruction with the FFT*, GPU Gems 2 (Matt Pharr, ed.), Addison Wesley, March 2005, pp. 765–784.

[TS07] R. Westermann T. Schiwietz, J. Georgii, *Freeform image*, Proceedings of Pacific Graphics 2007, 2007.

[WDS99] Mason Woo, Davis, and Mary Beth Sheridan, *Opengl programming guide: The official guide to learning opengl, version 1.2*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Weia] Eric W. Weisstein, *Gaussian elimination*, http://mathworld.wolfram.com/GaussianElimination.html.

[Weib] ———, *Gaussian-jordan elimination*, http://mathworld.wolfram.com/Gauss-JordanElimination.html.

[Weic] ———, *Lu decomposition*, http://mathworld.wolfram.com/LUDecomposition.html.

[Wika] Wikipedia, *Comparison of ATI graphics processing units*, http://en.wikipedia.org/wiki/Comparison_of_ATI_Graphics_Processing_Units.

[Wikb] ———, *Comparison of NVIDIA graphics processing units*, http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units.

[WS98] Yongmei Wang and Lawrence H. Staib, *Integrated approaches to non rigid registration in medical images*, Proceedings of WACV'98, 1998, pp. 102–108.

[WS01] Joachim Weickert and Christoph Schnörr, *A theoretical framework for convex regularizers in pde-based computation of image motion*, International Journal of Computer Vision **45** (2001), no. 3, 245–264.

[YM01] Boykov Y. and Jolly M.P., *Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images.*, International Conference on Computer Vision **1** (2001), no. 1, 105–112.

[ZF03] Barbara Zitova and Jan Flusser, *Image registration methods: a survey*, Image and Vision Computing **21** (2003), no. 11, 977–1000.