

MASTER

Acceleration of a geodesic fiber-tracking algorithm for diffusion tensor imaging using CUDA

van Aart, E.

Award date:
2010

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Acceleration of a Geodesic Fiber-Tracking
Algorithm for Diffusion Tensor Imaging
using CUDA

Evert van Aart

June 2010

Graduation Supervisor

Dr. Andrei Jalba

Eindhoven University of Technology
Department of Mathematics and Computer Science
Visualization group

Committee Members

Dr. Andrei Jalba

Dr. Anna Vilanova

Eindhoven University of Technology
Department of Biomedical Engineering
Biomedical Image Analysis group

Dr. Bart Mesman

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems group

Contents

1	Introduction	3
1.1	Problem Description	3
1.2	Document Structure	4
2	Related Work	5
2.1	Diffusion Tensor Imaging	5
2.1.1	Diffusion in White Matter	5
2.1.2	Diffusion Weighted Imaging	6
2.1.3	Second-Order Tensor Model	8
2.1.4	Anisotropy Measures	9
2.1.5	Diffusion Tensor Model Limitations	10
2.2	Fiber-Tracking in DTI Data	11
2.2.1	Streamline Methods	11
2.2.2	Front-Propagation Methods	13
2.3	GPU Acceleration of Fiber-Tracking Methods	14
2.4	Conclusion	14
3	Geodesic Fiber-Tracking	15
3.1	Introduction to Geodesics	15
3.2	Ordinary Differential Equations	16
3.3	Computing Geodesic Fibers	18
3.3.1	Data-Preparation Stage	18
3.3.2	Tracking Stage	20
3.4	Fiber Filtering	21
3.4.1	Region-to-Region Connectivity	21
3.4.2	Two-Point Ray Tracing Method	22
3.5	Summary	24
4	CUDA Implementation	25
4.1	Introduction to CUDA	25
4.1.1	Similar Technologies	26
4.1.2	Architecture	27
4.1.3	Design Considerations	30
4.2	Implementation Overview	31
4.2.1	Specifications	32
4.2.2	Structure	33
4.3	Pre-Processing Kernel	34
4.4	Derivatives Kernel	36

4.5	Tracking Kernel	37
4.5.1	Tile-Based Approach	37
4.5.2	Kernel Structure	39
4.5.3	Optional Stopping Criteria and Connectivity Measure	39
4.6	Introducing Textures	40
4.6.1	Non-Coalesced Memory Accesses	41
4.6.2	Texture Cache	41
4.6.3	Texture-Filtering Interpolation	43
4.6.4	Texture vectors	44
4.6.5	Boundary conditions	44
4.6.6	Addressing	45
4.6.7	Writing to textures	45
4.6.8	Conclusion	46
4.7	Summary	47
5	Results	49
5.1	Algorithm	50
5.2	Data-Preparation Stage	52
5.2.1	Pre-Processing Kernel	52
5.2.2	Derivatives Kernel	54
5.2.3	Memory Setup	54
5.2.4	Intra-Device Copying	55
5.2.5	Total Data-Preparation Time	57
5.3	Tracking Stage	57
5.3.1	Configuration	57
5.3.2	Texture-Filtering versus In-Kernel Interpolation	59
5.3.3	Texture Memory versus Global Memory	60
5.3.4	Limiting Factor	61
5.4	GPU versus CPU	64
5.5	Conclusions	65
6	Conclusion	67
6.1	Future Work	68
	Bibliography	69

Chapter 1

Introduction

1.1 Problem Description

Diffusion-Weighted Imaging (DWI) is a recent, non-invasive MRI technique that allows the user to measure the diffusion of water molecules in a given direction. Within fibrous tissue, water is known to experience greater diffusion parallel to the direction of the fibers than perpendicular to it. Therefore, DWI data can be used to deduce and analyze the structure of fibrous tissue, such as the white matter of the brain, and muscular tissue in the heart. Since DWI is the only technique that allows in-vivo imaging of fibrous structures like white matter, it has a number of unique applications, such as mapping the connectivity of the white matter, and diagnosis and monitoring of neurological diseases. Diffusion Tensor Imaging (DTI) describes the diffusion measured with DWI in different directions as a second-order tensor. In this project we focus in DTI. Although other models could be applied to DWI, DTI is still the most common model used in practice.

The process of using the measured diffusion to reconstruct the underlying of fiber structures is called fiber-tracking. Many different fiber-tracking algorithms have been developed since the introduction of Diffusion Tensor Imaging. This paper focuses on a novel approach, in which fibers are constructed by finding geodesics on a Riemannian manifold defined by the DTI data. This technique has several advantages over existing ones, such as its relatively low sensitivity to measurement noise (in particular when compared to the computationally simple but highly noise-sensitive streamlines technique), and its ability to identify multiple fiber connections between two points, which makes it suitable for analysis of complex structures.

One of the largest downsides of this algorithm is that it is computationally expensive, and therefore not yet suitable for potential real-time applications. The goal of the project presented in this report was to overcome this problem by implementing the existing algorithm on the highly parallel architecture of a Graphical Processing Unit (GPU), using the CUDA programming language. Since the nature of the algorithm under discussion allows for meaningful parallelization, the running time can be reduced by a factor of up to 60, compared to an implementation on a single-core CPU. The report describes the structure of the implementation in CUDA, as well as the relevant design considerations.

The primary contribution of the work presented herein is the acceleration of the geodesic fiber-tracking algorithm. This novel algorithm has been shown to compare favorably to existing fiber-tracking algorithms, but is hampered by its high computational complexity. Using CUDA, we were able to significantly speed up this algorithm, thus making it for suitable

for practical applications. Additionally, this report explores the design considerations relevant to CUDA-aided acceleration of 3D streamline tracking algorithms, in the sense that the structure of our implementation may be used as a guideline for implementations of similar algorithms. Finally, we have integrated the implementation in an full-featured research tool, allowing users to use the implemented algorithm for practical purposes.

1.2 Document Structure

This report discusses the acceleration of a fiber-tracking algorithm on the GPU. Chapter 2 describes the context for our research by discussing related works in the fields of Diffusion Tensor Imaging, fiber-tracking algorithms, and GPU acceleration of these algorithms. It provides an introduction to the relevant concepts for this thesis. Chapter 3 describes the geodesic fiber-tracking algorithm, which is the main focus of our research. The acceleration of this algorithm using the GPU is discussed in detail in Chapter 4, which also includes a general introduction on the CUDA language. Chapter 5 describes the performance of this GPU implementation in terms of speed and accuracy, followed by a conclusion in Chapter 6.

Chapter 2

Related Work

The research presented in this report is concerned with the acceleration of a Diffusion Tensor Imaging-based fiber-tracking algorithm by means of a Graphics Processing Unit. This chapter discusses the related work and theoretical background of the three main aspects of our subject: Diffusion Tensor Imaging, fiber-tracking algorithms, and acceleration of such algorithms using the GPU. As such, it provides context and motivation for our own research, and serves as an introduction for readers unfamiliar with one or more aspects of the research presented in this thesis.

2.1 Diffusion Tensor Imaging

Diffusion Tensor Imaging (DTI) is a Magnetic Resonance Imaging (MRI) technique that allows non-invasive measurement of the diffusion of water molecules within human or animal tissue. From this data, information about the structure of biological tissue can be extracted, using for example fiber-tracking algorithms (see Section 2.2). This section introduces the concepts behind DTI, and discusses its advantages and disadvantages when compared to different imaging techniques.

2.1.1 Diffusion in White Matter

Diffusion Tensor Imaging works on the knowledge that the diffusion of water molecules within biological tissue is influenced by the macroscopic structure of the tissue. Let us consider a uniform volume of water. The theory of Brownian motion dictates that molecules within this volume will diffuse randomly in all directions. Specifically, if we take a single molecule within the volume, which is located at position $\mathbf{x} = \mathbf{x}_0$ at time $t = t_0$, then the volume describing all possible positions of this molecule at $t = t_1$ (i.e. the *probability distribution* for \mathbf{x}_1 will be a sphere centered on \mathbf{x}_0). This type of diffusion is called *isotropic*. If, however, the molecule starts close to a barrier that slows or completely blocks diffusion, the probability of diffusion through this barrier will be low, and the probability distribution volume for \mathbf{x}_1 will no longer be spherical. We call this diffusion process *anisotropic*. The difference between isotropic and anisotropic diffusion is illustrated in Figure 2.1.

The fact that diffusion is anisotropic inside certain volumes can be used to identify the structure of fibrous biological tissues. In a volume containing parallel fibers, diffusion will be large along the direction of the fibers, and small in all other directions. Diffusion Tensor Imaging lets us measure this diffusion of water in multiple directions, which allows for in-

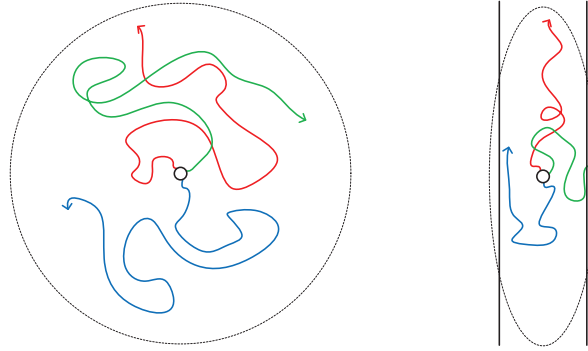


Figure 2.1: Isotropic (left) and anisotropic (right) diffusion in 2D. The colored arrows represent possible molecule trajectories from the starting point between in the interval from t_0 to t_1 . In the first figure, the dashed circle indicates that the diffusion is equally large in all directions; assuming finite viscosity, the location of the molecule at t_1 will be inside this circle. In the presence of barriers, diffusion is restricted in certain directions, and the area describing the possible locations at t_1 becomes an ellipse.

vivo identification of the structure of the tissue. Within the human body, DTI is commonly used in two areas: muscular tissue, and white matter. Both types of tissue are fibrous, meaning that DTI can be used to infer information about its structure from the DTI data. Diffusion Tensor Imaging can for example be used to detect anomalies in the structure of the heart [47], allowing us to monitor the regeneration of muscle tissue after infarction.

The main focus of this report is the application of DTI in the brain white matter. White matter consists of a large number of myelinated neurons (generally referred to in this report as the *fibers* of the white matter), which serve to transport electrical signals between different areas of the gray matter, where the actual 'processing' of these signals is performed. The white matter can be considered as a giant, complex network, with a total length of between 60,000 and 200,000 kilometers, depending largely on the person's age [33]. DTI allows us to map the connectivity of this network, thus giving us greater insight into the functional structure of the brain, especially when combined with the Functional Magnetic Resonance Imaging (fMRI) technique [10, 14, 25]. The DTI data has also been used during the planning stages of neurosurgery [12], and in the diagnosis and treatment of certain diseases, such as Alzheimer's disease [53], multiple sclerosis [16], and strokes [22]. Since the tissue of the white matter is macroscopically homogeneous, other imaging techniques, such as T2-weighted MRI, are unable to detect the structure of the underlying fibers, making DTI uniquely suitable for in-vivo inspection of white matter. This is illustrated in Figure 2.2.

2.1.2 Diffusion Weighted Imaging

Diffusion Weighted Imaging uses the Magnetic Resonance Imaging technique (MRI) to obtain information about the diffusion of water in biological tissue in a given direction. This is accomplished by measuring the spin echoes of water molecules in the presence of pulsed gradient fields, which define the direction in which the diffusion is measured. The sequence used for this purpose is the so called Stejskal-Tanner sequence [58], which is shown in Figure 2.3. In this sequence, a gradient field is applied for a short time (pulsed) before and after

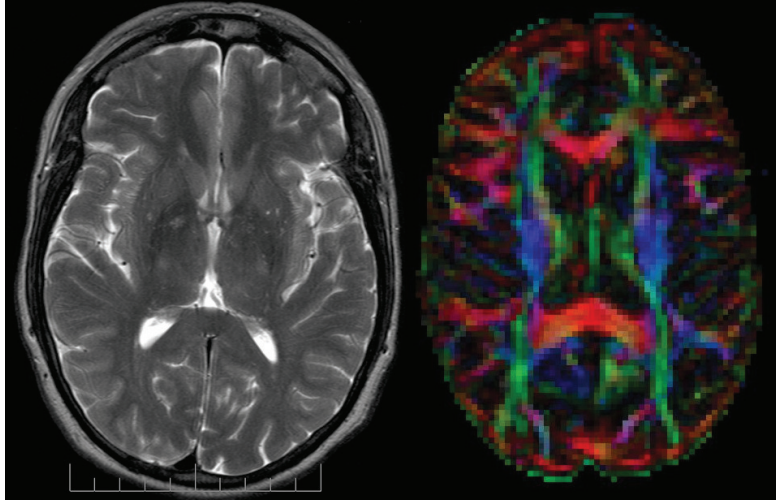


Figure 2.2: Left: T2-weighted image of the brain (Source: Wikipedia). The white matter is pictured as a largely uniform gray area with very little contrast, making it impossible to detect the individual fiber bundles. Right: DTI image of the brain. The voxels (volumetric pixels) have been colored according to the main direction of diffusion, with each of the three axes corresponding to one of the three main colors (red, green and blue). The brightness of a voxel depends on the level of anisotropy at that point. Distinct fiber bundles can be identified in the DTI Image.

the refocusing 180 degree pulse of the standard spin echo sequence. The first gradient pulse spatially 'tags' the protons by dephasing them. The magnitude of the phase depends on the magnitude of the magnetic field, and thus on the proton's location within the gradient field. The second gradient pulse, which is equal in magnitude and duration but of opposite sign, aims to rephase the protons. This rephasing will be imperfect for protons that have moved along the direction of the gradient in the time between the two pulses, and this will result in a signal attenuation (see Figure 2.4).

In order to compute the diffusion coefficient, we first compute the reference S_0 , which is the measured signal without gradient pulses. The output signal of the Stejskal-Tanner imaging sequence shown in Figure 2.3 is then given by the following equation,

$$S = S_0 \exp \left[-\gamma^2 \delta^2 \left(\Delta - \frac{\delta}{3} \right) |\mathbf{g}|^2 D \right]. \quad (2.1)$$

Here, δ and Δ are the duration of the gradient pulse and the time between gradient pulses, respectively (see Figure 2.4), γ the gyromagnetic ratio of the proton, and $|\mathbf{g}|$ is the magnitude of the gradient field. The scalar value D represents the diffusion coefficient of the molecules along the direction of the gradient, and is commonly called the *Apparent Diffusion Coefficient* (ADC). Since γ , δ , Δ and $|\mathbf{g}|$ are known, the ADC can be expressed as the natural logarithm of the ratio between S and S_0 , multiplied by a constant factor. The ADC can potentially reveal information about the structure of the scanned tissue, but is limited in its applicability by the fact that it quantifies the diffusion in a single direction.

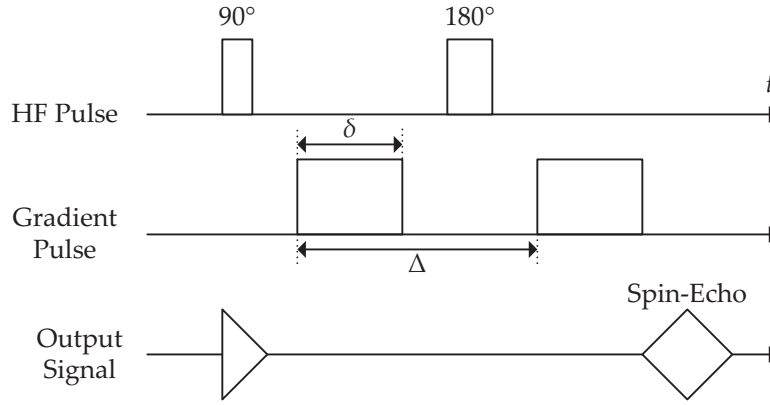


Figure 2.3: Stejskal-Tanner sequence used in MRI to measure in-vivo diffusion of water molecules. In addition to the 90° and 180° high-frequency pulses of the standard spin-echo sequence, the Stejskal-Tanner uses two gradient pulses of equal magnitude and duration but of opposite sign.

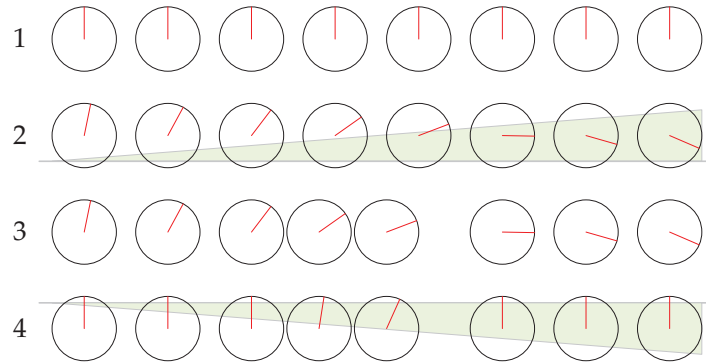


Figure 2.4: Illustration of the effect of the gradient pulses in the Stejskal-Tanner sequence. **1:** All protons are in phase. **2:** The first gradient pulse is applied. The protons are dephased; the magnitude of the phase depends on their spatial location. **3:** Some protons move due to the effects of diffusion. **4:** The second gradient pulse is applied. Stationary protons are rephased; rephasing is imperfect for protons that have moved. The remaining phase difference will result in an attenuated signal compared to the standard spin-echo sequence. Illustration based on Wandell *et al.* [62].

2.1.3 Second-Order Tensor Model

By measuring diffusion in a number of directions using different gradients and combining the resulting data, we can model the diffusion of water molecules at a specified location using a second-order tensor. Specifically, let \mathbf{J} be the diffusive flux, and let ∇C be the concentration gradient. The *diffusion tensor* \mathbf{D} is the 3×3 symmetric tensor that satisfies $\mathbf{J} = -\mathbf{D}\nabla C$, or:

$$\begin{bmatrix} J_x \\ J_y \\ J_z \end{bmatrix} = \begin{bmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{xy} & D_{yy} & D_{yz} \\ D_{xz} & D_{yz} & D_{zz} \end{bmatrix} \begin{bmatrix} \frac{\delta C}{\delta x} \\ \frac{\delta C}{\delta y} \\ \frac{\delta C}{\delta z} \end{bmatrix} \quad (2.2)$$

In 1994, Basser *et al.* [4] introduced a mathematical model for the estimation of the individual elements of this diffusion tensor using the echo intensities measured by the MR device. Since then, a number of different tensor estimation methods have been proposed, each with distinct advantages and disadvantages. This estimation of second-order tensors using diffusion data obtained through Diffusion Weighted Imaging is referred to as *Diffusion Tensor Imaging* (DTI), and can be obtained by performing Diffusion Weighted Imaging using a number of different gradient directions. The methods described in this report are assumed to have pre-estimated DTI tensors available to them as input. The estimation methods used to obtain these tensors and their accuracy are beyond the scope of this research. For an overview of a number of estimation methods and their individual advantages and disadvantages, please refer to Koay *et al.* [24].

A common approach to visualizing the individual DTI tensors is to use their *eigenvectors* and *eigenvalues* to create an *ellipsoid*. Here, principal directions of the ellipsoid are defined by the three eigenvectors, and the radius of the ellipsoid in each principle direction is determined by the corresponding eigenvalue. For eigenvectors \mathbf{e}_1 , \mathbf{e}_2 and \mathbf{e}_3 , we have eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_3$. We define the *main eigenvector* as the eigenvector corresponding to the largest eigenvalue λ_1 . In a number of existing fiber-tracking algorithms (see Section 2.2), the main eigenvector is interpreted as the most likely orientation of the underlying fiber bundles. Figure 2.5 shows a diffusion tensor and its corresponding eigenvectors and eigenvalues.

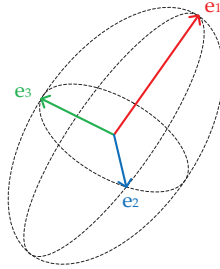


Figure 2.5: 3D ellipsoid described by the three eigenvectors of a DTI tensor. The eigenvectors \mathbf{e}_1 , \mathbf{e}_2 and \mathbf{e}_3 are shown in red, blue and green, respectively. In this case, $\lambda_1 \gg \lambda_2 \approx \lambda_3$, meaning that the tensor is strongly oriented along the main eigenvector. The shape and size of the ellipsoid intuitively translate to the measured diffusion.

2.1.4 Anisotropy Measures

A DTI tensor can be classified based on the relative sizes of its eigenvectors (and, therefore, on the shape of the tensor's ellipsoid). Specifically, we distinguish between (nearly) *spherical* ellipsoids ($\lambda_1 \approx \lambda_2 \approx \lambda_3$), *oblate* ellipsoids ($\lambda_1 \approx \lambda_2 \gg \lambda_3$), and *prolate* ellipsoids ($\lambda_1 \gg \lambda_2 \approx \lambda_3$). The shape of the ellipsoid corresponds to the anisotropy of the diffusion in a voxel: spherical ellipsoids represent voxels with isotropic diffusion, while prolate

ellipsoids indicate highly anisotropic diffusion (i.e. a large diffusion coefficient in one direction, and smaller coefficients in all other directions). Oblate ellipsoids represent voxels with *planar* diffusion, meaning that the diffusion coefficient is relatively large in more than one direction. This can for example occur if a single voxel contains multiple fibers with different orientations (see Section 2.1.5).

An *anisotropy measure* allows us to quantify the level of anisotropy based on the eigenvalues of a tensor. Such a measure is usually expressed by scalar value, which will be high for tensors with prolate ellipsoids, and low for tensors with spherical or oblate ellipsoids. Anisotropy measures are for example used by fiber-tracking algorithms, which aim to reconstruct fiber pathways in white matter (see Section 2.2). A common fiber-tracking algorithm uses the main eigenvector of a tensor as the local direction of the reconstructed fibers. For this algorithm, tensors with low anisotropy values represent areas of uncertainty, as the main eigenvector is not robustly defined for such tensors. In this case, computing an anisotropy measure allows the algorithm to detect such areas, and to react appropriately. The algorithm may for example choose not to track fibers through regions of low anisotropy, as doing so would likely result in erroneous fiber trajectories.

A number of different anisotropy measures have been proposed [5, 13]. Throughout the remainder of this paper, we use the Fractional Anisotropy as our main anisotropy measure:

$$FA = \sqrt{\frac{1}{2} \frac{\sqrt{(\lambda_1 - \lambda_2)^2 + (\lambda_1 - \lambda_3)^2 + (\lambda_2 - \lambda_3)^2}}{\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}}} \quad (2.3)$$

Here, λ_1 , λ_2 and λ_3 are the eigenvalues of the tensor in descending order of magnitude.

2.1.5 Diffusion Tensor Model Limitations

One problem with the DTI technique described above stems from the fact that the voxels, which are commonly in the order of millimeters, are far larger than the fibers (axons) within the white matter, which have a diameter in the order of micrometers. When a voxel in the DTI image contains a number of parallel fibers, the resulting tensor will be strongly orientated along the direction of the fiber, and its main eigenvector will be approximately parallel to this direction. However, when a voxel contains multiple fiber bundles with different orientations, or branching fiber bundles, the corresponding tensor will no longer be strongly anisotropic, instead becoming planar or even largely isotropic. This effect, which is commonly referred to as the *partial volume effect* [1, 63], is illustrated in Figure 2.6.

The research in this thesis focuses on fiber-tracking algorithms, which aim to deduce the trajectories of fibers in white matter using the DTI tensors, see Section 2.2. Depending on the algorithm used, the partial volume effect can have great impact on the accuracy of the computed fibers. The consequences of this partial volume effects on the accuracy of DTI fiber-tracking algorithms will be discussed in Section 2.2 and Chapter 3.

A potential solution to the problem introduced by the partial volume effect is to measure the diffusion in more directions than the six needed for the diffusion tensor estimation, and to subsequently model this diffusion using a more complex mathematical model. This approach is generally referred to as High Angular Resolution Diffusion Imaging (HARDI), and while its acquisition times and computational complexity are both far greater than those of DTI, it has the distinct advantage that it preserve information about crossing fibers. As such, it can potentially be used for more accurate fiber tractography methods (see Section 2.2), which would be especially useful in complex areas of the brain. A number of HARDI-based

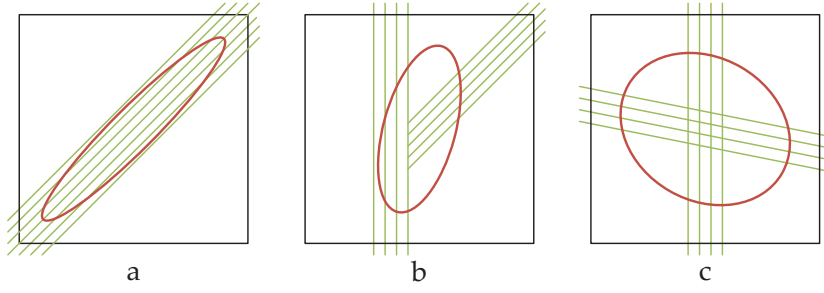


Figure 2.6: A 2D illustration of the partial volume effect. In a situation *a*, the pixel (black square) contains a number of fibers that all run in the same direction (green lines). As a result, the corresponding tensor (red ellipse) is strongly orientated in the direction of the fibers. However, when the pixel contains branching fibers (*b*) or crossing fibers (*c*), the isotropy of the tensor increases, and its main eigenvector no longer run parallel to any of the fiber bundles.

fiber-tracking algorithms have been developed [49, 2], but are considered to be beyond the scope of this research.

2.2 Fiber-Tracking in DTI Data

As explained in the previous section, DTI data can provide valuable information about the orientation of fibers with fibrous biological tissue. The process of reconstructing fibers using DTI data is commonly called *DTI fiber-tracking*. Since the advent of DTI, a number of different DTI fiber-tracking methods have been developed. This section discusses the concepts behind some of these algorithms, as well as their advantages and disadvantages.

2.2.1 Streamline Methods

The most straightforward and most commonly used DTI fiber-tracking algorithm is the *streamline tracking* algorithm. This algorithm uses the main eigenvector of the DTI tensors (i.e., the eigenvector corresponding to the largest eigenvalue) as the local direction of the fiber. In other words, the curve $\mathbf{c}(\tau)$ describing the fiber is computed by integration of the main eigenvector $\mathbf{e}_1(\mathbf{c}(\tau))$, i.e.,

$$\mathbf{c}(\tau) = \int \mathbf{e}_1(\mathbf{c}(\tau)) d\tau. \quad (2.4)$$

Here, $\mathbf{e}_1(\mathbf{c}(\tau))$ is the main eigenvector at the position described by $\mathbf{c}(\tau)$. We can solve Equation 2.4 using a Ordinary Differential Equation (ODE) solver, in which the local first derivative of the curve is equal to the main eigenvector at that point. Using Euler's method in combination with a step size h , we can compute the next point of the fiber using the current position, \mathbf{x}_i :

$$\mathbf{x}_{i+1} = \mathbf{x}_i + h\mathbf{e}_1(\mathbf{x}_i) \quad (2.5)$$

For additional accuracy, we can replace Euler’s method by a higher-order ODE solver, such as the second- or fourth-order Runge-Kutta method. Decreasing the step size will also result in a more accurate computation of the curve.

The streamline fiber-tracking algorithm is computationally inexpensive, and is capable of producing good results [10, 6, 35, 64]. However, the method is also very sensitive to noise [29, 50]. A relatively small amount of additive noise in the DTI tensors may cause its main eigenvector to drastically change its magnitude and/or orientation. Since the integration method strictly follows this main eigenvector, this change may lead to large, accumulative errors in the trajectory of the fibers. This process is illustrated in Figure 2.7. Furthermore, this method is unable to compensate for any errors introduced by the partial volume effect (see Section 2.1.5), which severely limits its applicability in complex regions of the white matter.

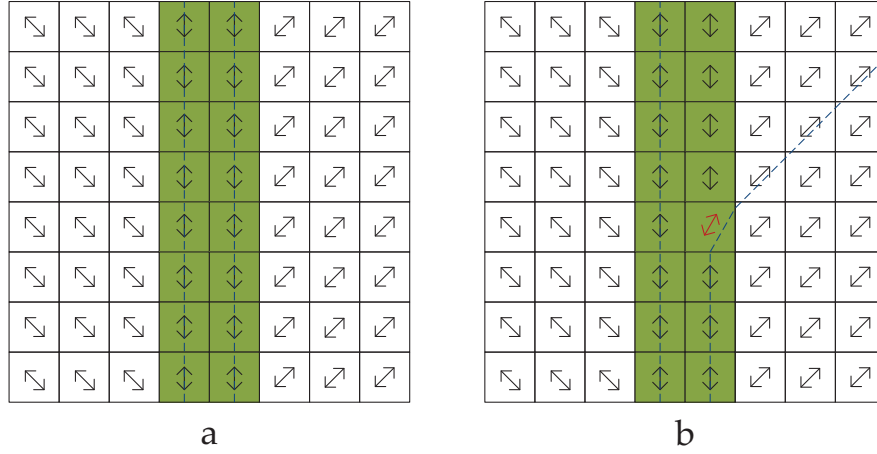


Figure 2.7: A 2D illustration of the effect of noise on the streamline fiber-tracking algorithm. The green squares in the grid represent a fiber, and the arrows represent the main eigenvectors of the DTI tensors. In situation *a*, the streamlines (dashed blue lines) perfectly follow the fiber. However, in situation *b*, a small error in one of the tensors (red arrow) causes one streamline to completely deviate from the fiber.

In order to prevent this sensitivity to noise from creating erroneous fibers, streamline tracking algorithms usually have a number of stopping criteria. The fiber-tracking process will for example terminate when a voxel with low anisotropy is encountered, or when the angle between the current segment and the previous segment of the fiber is larger than a fixed maximum value. However, terminating the tracking process when an isotropic voxel is encountered reduces the applicability of the algorithm in those areas of the brain where the DTI tensors lack a strong directional component.

In an attempt to overcome the noise sensitivity problem of streamline tracking, Lazar *et al.* [29] introduced the *tensor deflection* algorithm, commonly referred to as TEND. Let \mathbf{v}_{i+1} and \mathbf{v}_i be the fiber direction in the next and current integration step, respectively. The tensor deflection algorithm computes the next direction by multiplying the current direction by the entire DTI tensor:

$$\mathbf{v}_{i+1} = \mathbf{D} \cdot \mathbf{v}_i \quad (2.6)$$

In strongly anisotropic voxels, the fiber direction will be deflected towards the main eigenvector, while in largely isotropic voxels, the direction hardly changes. In this way, the TEND algorithm should be able to cross voxels that are (more) isotropic due to noise or the partial volume effect, without deviating too far from the actual fiber path. In this way, it is theoretically possible to traverse voxel containing crossing fibers with losing track of the correct fiber trajectory. The results presented by Lazar *et al.* [29] show that this approach tends to outperform the streamline tracking algorithm, while also remaining computationally lightweight.

A second extension of the streamline fiber-tracking algorithm attempts to overcome its sensitivity to noise by introducing a stochastic element [30, 45]. It does so by randomly varying the direction of the main eigenvectors according to some probabilistic distribution function (PDF). By tracking a large number of fibers from a single seed point and subsequently looking at the density of the computed fibers, we can identify possible fiber trajectories (i.e., if the majority of the fibers follows one path, this path is likely to correspond to an actual fiber bundle). This may even allow the user to identify branching and crossing fibers, even when the partial volume effect has made the corresponding voxels largely isotropic. The obvious downside of this approach is that it requires the computation of a large number of fibers, making it much slower than the previous two algorithms.

2.2.2 Front-Propagation Methods

One of the biggest downsides of streamline-based fiber-tracking algorithm, aside from their general sensitivity to noise and to the partial volume effect, is that, since these algorithms track fibers by locally following the main eigenvectors, they are unable to find globally optimal connections between different regions of the white matter. To overcome this problem, a second group of fiber-tracking algorithms aims to deduce connectivity in the white matter by globally optimizing a certain cost function. This can be done by propagating a front from a user-defined seed point throughout the entire volume. Sethian [56] introduces two distinct approach to front-propagation methods: the *Fast Marching Method*, and the *Level Set Method*. In both cases, the front propagation is described by a speed function F , which may depend on both local characteristics (e.g., tensor data and front characteristics of a single point on the front) and global front characteristics (e.g., size and shape of the front). The nature of the speed function determines the type of the front-propagation method.

Fast Marching Methods assume that the speed of the front F never changes sign (i.e., it is either always positive or always negative). When a point is passed by the front, it is assigned an *arrival time* T , using the relationship $|\nabla T|F = 1$. Since F never changes sign, no point can be passed by the front more than once. When the front has propagated throughout the entire volume, a gradient descent scheme can be used on the values of T to identify possible connections between the seed point and a second point of interest (i.e., the shortest path between the points). Several different research groups have successfully used Fast Marching methods for finding fiber trajectories in white matter [44, 51, 57].

Level Set Methods differ from Fast Marching Methods in the sense that the speed function F is allowed to change sign. As a result, the front may move "backward", meaning that points may be passed by the front more than once. A single arrival time value is now no longer sufficient to describe the evolution of the front. To solve this problem, the initial position of the front is used as the *zero level set* of a higher dimensional function ϕ . The evolution

of the front can now be described through a time-dependant initial value problem, in which the actual front is given by the zero level set of ϕ . From this, the level set equation is derived (see Sethian [56] and Osher *et al.* [43] for details):

$$\phi_t + F|\nabla\phi| = 0 \quad (2.7)$$

By solving this equation using a known value of $\phi(\mathbf{x}, t = 0)$, we can find the evolution of the front, and from this, the possible trajectories of fiber bundles. This technique has successfully been used in the mapping of fiber trajectories in the white matter [9, 21].

2.3 GPU Acceleration of Fiber-Tracking Methods

The research presented in this thesis focuses on the acceleration of a novel fiber-tracking algorithm, which will be introduced in Chapter 3. Specifically, we aim to use the highly parallel architecture of modern Graphics Processing Units (GPU) to speed up the execution of a computationally expensive algorithm. The possibility of using the GPU to accelerate fiber-tracking algorithms has recently been explored by a number of research groups [26, 27, 42, 41]. These implementations use either geometric shaders [26] or fragment shaders [27, 42, 41] to accelerate the streamline tracking algorithm. With the exception of Mittmann *et al.* [42], who introduce a stochastic element, these papers all use the simple streamline method for fiber-tracking, in which the main eigenvector of the DTI tensor is used as the direction vector for the integration step. Finally, the computation capabilities of the GPU has been exploited for the purpose visualization, with the aim of rapidly generating intuitively clear graphical representations of the DTI tensor ellipsoids and/or the generated fibers [8, 48, 52, 59].

In this research, we have accelerated a fiber-tracking algorithm using the *Compute Unified Device Architecture* (CUDA) framework [38], which allows us to write programs for the highly parallel GPU architecture. Recently, Jeong *et al.* [20] have developed a CUDA implementation of a fiber-tracking algorithm based on the Hamilton-Jacobi equation (see Section 2.2.2). Their solution parallelizes the propagation of the front by dividing the DTI image into blocks of 4^3 voxels, after which the front is propagated through a number of such blocks in parallel. This approach has been shown to be 50 to 100 times faster than a sequential implementation on a CPU.

2.4 Conclusion

In this chapter, we have described the context of the research presented in this thesis, by introducing theoretical concepts and citing relevant related work. The focus of our research is the GPU-aided acceleration of a fiber-tracking algorithm, which aims to reconstruct fiber pathways using Diffusion Tensor Imaging (DTI) data. A number of existing fiber-tracking algorithms were discussed, as well as their relative advantages and disadvantages. In the next chapter, we will introduce the geodesic fiber-tracking algorithm, which is the focus of our research. The acceleration of this computationally expensive algorithm will subsequently be discussed in Chapter 4.

Chapter 3

Geodesic Fiber-Tracking

This chapter introduces a geodesic fiber-tracking algorithm, an approach to calculating the trajectories of fibers bundles in DTI data. The main advantages of this algorithm, compared to existing fiber-tracking algorithms, are its low sensitivity to noise, and the fact that it provides a *multivalued* solution (i.e., it allows us to find multiple possible connections between regions). The research presented in the remainder of this report all focuses on the speed and performance of this algorithm. We first introduce the mathematical background of Riemannian manifolds and geodesics, after which we present a numerical approach for computing the geodesic fibers.

3.1 Introduction to Geodesics

A geodesic is defined as the shortest path through a Riemannian manifold. A Riemannian manifold is a real, differentiable manifold, on which each tangent space is equipped with a so-called *Riemannian metric*, which is a positive definite tensor. Roughly speaking, the elements of the metric tensor are an indication of the *cost* of (or *energy* required for) moving in a specific direction. For DTI data, an intuitive choice for the metric is the *inverse* of the DTI tensor. Large values in the DTI tensor correspond to small values in its inverse, indicating low diffusion costs, and vice versa. Locally, a geodesic will tend to follow the direction with the lowest metric value, which is analogous to the direction with the highest diffusion. We define the Riemannian metric as $G = D^{-1}$, where D is the DTI tensor.

A number of research groups, including Lenglet *et al.* [32], have explored the possibilities of using geodesics in Riemannian manifolds to determine fiber trajectories in white matter. This theoretic principle was implemented as a front-propagation fiber-tracking algorithm, based on the Level Set Methods described in Section 2.2.2. Here, the potential fiber trajectories are computed by first propagating a front throughout the entire volume, and subsequently analyzing the front characteristics to locate the shortest path. The Hamilton-Jacobi equation is used to obtain the solution of Equation 2.7.

One of the downsides of geodesic fiber-tracking algorithms based on the Hamilton-Jacobi equations is that each point in the volume is assigned only a single value, which represents a minimization of the cost function. The gradient field of these values may develop discontinuities when the correct solution of the Hamilton-Jacobi equation is multi-valued, i.e., when there is more than one valid connection between the seed point and another point. Research has shown that the white matter does contain multiple fibers connections between two different regions [46]. These structures cannot be captured by the Hamilton-Jacobi-based

front-propagation methods. Furthermore, the solution of the Hamilton-Jacobi will not always correspond to an actual fiber bundle.

Recently, Sepasian *et al.* [54, 55] discussed a novel version of the geodesic fiber-tracking algorithm. In this algorithm, the trajectory of a fiber is computed iteratively by numerically solving a set of Ordinary Differential Equations, similar to the streamline approach introduced in Section 2.2. The ODEs used to compute the trajectory of the fiber are derived from the theory of geodesics in a Riemannian manifold, as shown in section 3.2. The big advantage of this algorithm, compared to those based on the Hamilton-Jacobi equation, is that is able to capture multiple possible connections between regions, thus giving a *multi-valued* solution. This algorithm forms the basis of our research, and will be discussed in detail in the following sections.

3.2 Ordinary Differential Equations

Let $\mathbf{x}(\tau)$ be a smooth, differentiable curve through a volume described by parameter $\tau = [0, T]$, with derivative vector $\dot{\mathbf{x}}(\tau)$. We define the Riemannian length of $\mathbf{x}(\tau)$ as follows:

$$L(\mathbf{x}) = \int_0^T \sqrt{\dot{\mathbf{x}}^T G \dot{\mathbf{x}}} d\tau \quad (3.1)$$

The geodesic is the line that *minimizes* the geodesic length of Equation 3.1. In order to find this minimum, we first define the *energy* of the curve $\mathbf{x}(\tau)$:

$$E(\mathbf{x}) = \int_0^T \dot{\mathbf{x}}^T G \dot{\mathbf{x}} d\tau \quad (3.2)$$

By Hölder's Inequality, we obtain that $L(\mathbf{x})^2 \leq 2T * E(\mathbf{x})$, indicating that the geodesic curve also represents a minimization of the energy functional. From Equation 3.2, we obtain a system of ODEs that allow us to find this minimal curve using the Euler-Lagrange equation:

$$\frac{d}{d\tau} \frac{\partial(\dot{\mathbf{x}}^T G \dot{\mathbf{x}})}{\partial \dot{x}_\gamma} - \frac{\partial(\dot{\mathbf{x}}^T G \dot{\mathbf{x}})}{\partial x_\gamma} = 0 \quad (3.3)$$

Here, x_γ and \dot{x}_γ represent the component of $\mathbf{x}(\tau)$ and $\dot{\mathbf{x}}(\tau)$, respectively, in the γ^{th} dimension (with $\gamma = 1, 2, 3$). To solve the Euler-Lagrange equations, we first compute $(\dot{\mathbf{x}}^T G \dot{\mathbf{x}})$.

$$\dot{\mathbf{x}}^T G \dot{\mathbf{x}} = \sum_{i=1}^3 \left[\dot{x}_i \sum_{j=1}^3 g_{ij} \dot{x}_j \right] \quad (3.4)$$

Here, g_{ij} is the element in the i^{th} row and j^{th} column of the metric tensor G . Please note that, since the DTI tensor is symmetric, its inverse is too, so we have $g_{ij} = g_{ji}$.

By combining Equations 3.4 and 3.3, we can find the ODEs corresponding to the Euler-Lagrange equations.

$$\frac{d}{d\tau} \left(\frac{\partial(\sum_{i=1}^3 \sum_{j=1}^3 g_{ij} \dot{x}_i \dot{x}_j)}{\partial \dot{x}_\gamma} \right) - \frac{\partial(\sum_{i=1}^3 \sum_{j=1}^3 g_{ij} \dot{x}_i \dot{x}_j)}{\partial x_\gamma} = 0,$$

or,

$$\frac{d}{d\tau} \left(\sum_{k=1}^3 g_{k\gamma} \dot{x}_k + \sum_{k=1}^3 g_{\gamma k} \dot{x}_k \right) - \sum_{i=1}^3 \sum_{j=1}^3 \frac{\partial g_{ij}}{\partial x_\gamma} \dot{x}_i \dot{x}_j = 0. \quad (3.5)$$

Next, we use the product rule to compute the derivative $\frac{d}{d\tau}$, i.e.,

$$2 \sum_{k=1}^3 (g_{k\gamma} \ddot{x}_k) + \sum_{k=1}^3 \left(\frac{dg_{k\gamma}}{d\tau} \dot{x}_k \right) + \sum_{k=1}^3 \left(\frac{dg_{\gamma k}}{d\tau} \dot{x}_k \right) - \sum_{i=1}^3 \sum_{j=1}^3 \frac{\partial g_{ij}}{\partial x_\gamma} \dot{x}_i \dot{x}_j = 0,$$

or,

$$2 \sum_{k=1}^3 (g_{k\gamma} \ddot{x}_k) + \sum_{k=1}^3 \sum_{l=1}^3 \frac{\partial g_{k\gamma}}{\partial x_l} \dot{x}_k \dot{x}_l + \sum_{k=1}^3 \sum_{l=1}^3 \frac{\partial g_{\gamma k}}{\partial x_l} \dot{x}_k \dot{x}_l - \sum_{i=1}^3 \sum_{j=1}^3 \frac{\partial g_{ij}}{\partial x_\gamma} \dot{x}_i \dot{x}_j = 0. \quad (3.6)$$

By rewriting the indices, we can combine the three right-most terms, i.e.,

$$2 \sum_{k=1}^3 (g_{k\gamma} \ddot{x}_k) + \sum_{i=1}^3 \sum_{j=1}^3 \dot{x}_i \dot{x}_j \left(\frac{\partial g_{i\gamma}}{\partial x_j} + \frac{\partial g_{\gamma j}}{\partial x_i} - \frac{\partial g_{ij}}{\partial x_\gamma} \right) = 0. \quad (3.7)$$

Next, we multiply the entire equation by $0.5 * g^{\gamma\sigma}$ for $\gamma = 1, 2, 3$, and we sum the result. Here, $g^{\gamma\sigma}$ is equal to $D_{\gamma\sigma}$, where D is the original DTI tensor. Recall that $G = D^{-1}$. The inclusion of this sum allows us to simplify the left-most term to the second derivative of x_γ , as will be shown in Equation 3.10.

$$\sum_{\gamma=1}^3 g^{\gamma\sigma} \left[\sum_{k=1}^3 (g_{k\gamma} \ddot{x}_k) + \frac{1}{2} \sum_{i=1}^3 \sum_{j=1}^3 \dot{x}_i \dot{x}_j \left(\frac{\partial g_{i\gamma}}{\partial x_j} + \frac{\partial g_{\gamma j}}{\partial x_i} - \frac{\partial g_{ij}}{\partial x_\gamma} \right) \right] = 0,$$

or,

$$\sum_{\gamma=1}^3 g^{\gamma\sigma} \left[\sum_{k=1}^3 (g_{k\gamma} \ddot{x}_k) \right] + \frac{1}{2} \sum_{\gamma=1}^3 g^{\gamma\sigma} \left[\sum_{i=1}^3 \sum_{j=1}^3 \dot{x}_i \dot{x}_j \left(\frac{\partial g_{i\gamma}}{\partial x_j} + \frac{\partial g_{\gamma j}}{\partial x_i} - \frac{\partial g_{ij}}{\partial x_\gamma} \right) \right] = 0. \quad (3.8)$$

Using the knowledge that $\sum_{\gamma=1}^3 \sum_{k=1}^3 (g^{\gamma\sigma} g_{k\gamma}) = 1$ if $k = \sigma$ and 0 otherwise, we can simply the left-most term, i.e.,

$$\ddot{x}_\sigma + \frac{1}{2} \sum_{\gamma=1}^3 g^{\gamma\sigma} \left[\sum_{i=1}^3 \sum_{j=1}^3 \dot{x}_i \dot{x}_j \left(\frac{\partial g_{i\gamma}}{\partial x_j} + \frac{\partial g_{\gamma j}}{\partial x_i} - \frac{\partial g_{ij}}{\partial x_\gamma} \right) \right] = 0 \quad (3.9)$$

Finally, after again rewriting the indices, we arrive at the following sets of ODEs,

$$\ddot{x}_\gamma + \sum_{\alpha=1}^3 \sum_{\beta=1}^3 \Gamma_{\alpha\beta}^\gamma \dot{x}_\alpha \dot{x}_\beta = 0, \quad (3.10)$$

where $\Gamma_{\alpha\beta}^\gamma$ are the so-called *Christoffel symbols*, defined as follows:

$$\Gamma_{\alpha\beta}^\gamma = \sum_{\sigma=1}^3 \frac{1}{2} \left[g^{\gamma\sigma} \left(\frac{\partial}{\partial x_\alpha} g_{\beta\sigma} + \frac{\partial}{\partial x_\beta} g_{\alpha\sigma} - \frac{\partial}{\partial x_\sigma} g_{\alpha\beta} \right) \right] \quad (3.11)$$

The geodesic fiber-tracking algorithm computes fiber trajectories by numerically solving the ODEs shown in Equation 3.10, which will be the algorithmic basis for the implementations presented in subsequent sections. A more detailed description of this derivation can be found in Jost *et al.* [23].

3.3 Computing Geodesic Fibers

The computation of the trajectory of the geodesic fibers can be split into two separate stages: a *data-preparation* stage, and a *tracking stage*. In the tracking stage, the system of ODEs presented in Equation 3.10 is solved numerically, using a time integration step. In order to compute the Christoffel symbols defined by Equation 3.11, we first need to compute the derivatives of the inverse DTI tensors; this is done in the data-preparation stage.

3.3.1 Data-Preparation Stage

In the **data-preparation stage**, we compute the data required to compute the Christoffel symbols by individually processing the tensors of the voxels in the image. The three steps within this stage are *pre-processing the tensors*, *computing the inverse of the tensors*, and *computing the derivatives*. These three steps are described in detail below. Figure 3.1 illustrates the computational flow of this stage.

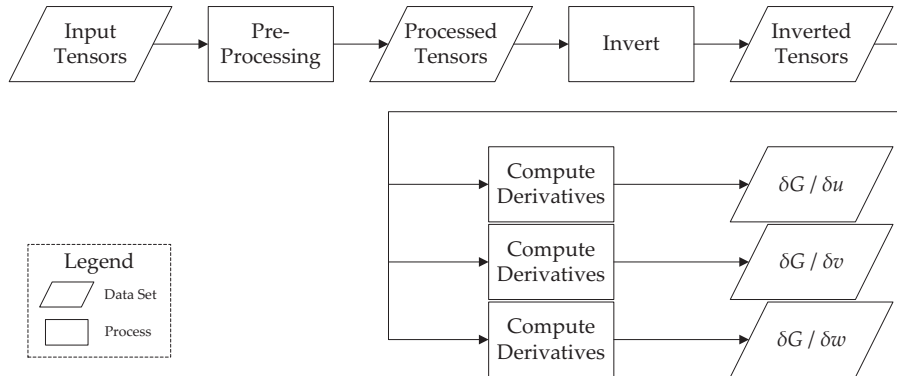


Figure 3.1: Flowchart of the data-preparation stage. The input DTI tensors are first pre-processed, which consists of applying a constant gain and an optional sharpening step. These pre-processed tensors are subsequently inverted. Finally, the derivatives of the inverted tensors are computed for all three dimensions (here referred to as u , v , and w), producing three derivative tensors per voxel. These derivative tensors, in addition to the pre-processed tensor produced by the pre-processing step, will serve as input for the tracking stage.

- The **pre-processing** step performs a number of different action.

- First, it checks if the tensors are nonsingular and positive definite. If either condition fails, we cannot use the tensor for the purpose of geodesic fiber-tracking, and we need to modify it. One simple solution is to replace these tensors by the identity tensor. Tests have shown that the vast majority of tensors for which these conditions do not hold (over 99.9% for whole-brain images) do not have any non-zero elements, which means that the tensor is located in a region without measurable diffusion (usually in the air around the head). Replacing these all-zero tensors with the identity tensor ensures correct behaviour of the tensor inversion step, without meaningfully affecting the trajectories of the fibers (i.e., fibers will not change direction in completely isotropic regions). For the small percentage of tensors that are singular and/or negative definite, but do have non-zero elements, a more sophisticated method may be used to correct the tensor while preserving its information; however, such a scheme has not been included in our implementation.
 - Next, we apply a constant gain factor to all tensor elements. This is done to increase their range, which reduces the impact of hardware-imposed limitations in accuracy. This is especially relevant for those tensors that are nearly singular, as a small error in the computation of the determinant may lead to a completely erroneous inverse tensor.
 - Finally, we may choose to sharpen the tensors. Sharpening can for example be done by exponentiation of the entire tensor by a constant exponent. Doing so may improve the behavior of the fiber-tracking algorithm by increasing their anisotropy. However, since this changes the actual shape of the tensors, sharpening should be applied sparingly, if at all, when dealing with real DTI data.
- **Inverting the tensors** is a fairly straightforward step. Since the pre-processing step has already checked for invertibility (and fixed those tensors that failed the check), we do not need to check for it during this step. Numerically, we compute the inverse by first computing the 3×3 matrix of cofactors of the DTI tensor D , and subsequently dividing the elements in this matrix by the determinant of D .
 - **Computing the derivatives** is done for all three dimensions. The actual computations performed in this step depend on the numerical differentiation scheme and its behavior at the borders. In the implementations presented in this report, we used the two-sided differentiation scheme. At the borders of the image, we instead use one-sided differentiation. See Equations 3.12 and 3.13 for the formulas for two- and one-sided differentiation, respectively. Here, h_γ is the distance between voxels in the γ -dimension, and \mathbf{u}_γ is the unit vector in the γ -dimension.

$$\frac{\partial g_{ij}(\mathbf{x})}{\partial x_\gamma} = \frac{g_{ij}(\mathbf{x} + h_\gamma \mathbf{u}_\gamma) - g_{ij}(\mathbf{x} - h_\gamma \mathbf{u}_\gamma)}{2h_\gamma} \quad (3.12)$$

$$\frac{\partial g_{ij}(\mathbf{x})}{\partial x_\gamma} = \frac{g_{ij}(\mathbf{x}) - g_{ij}(\mathbf{x} \pm h_\gamma \mathbf{u}_\gamma)}{h_\gamma} \quad (3.13)$$

3.3.2 Tracking Stage

In the **tracking stage**, the trajectories of the fibers are computed, using the data generated in the previous phase. This is an iterative numerical integration process, as illustrated by Figure 3.2.

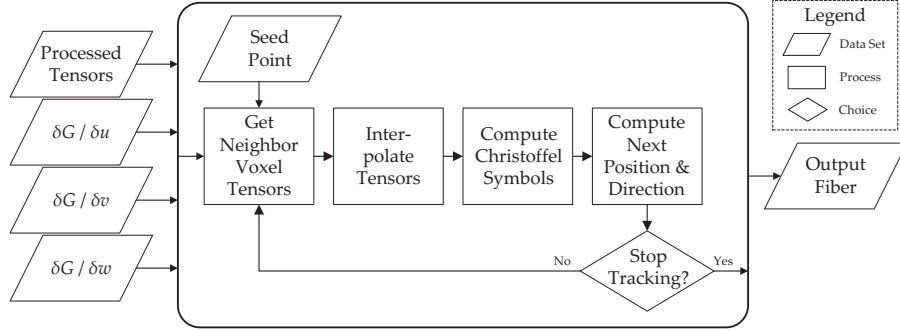


Figure 3.2: Flowchart of the tracking stage. Using the tensors computed in the data-preparation stage and a seed point (which determines the initial position and direction), the fiber trajectory is computed using an iterative process. The tracking process stops when one of the stopping criteria is met, at which point the computed fiber data is sent to the output.

- The (pre-processed) DTI tensor and the three derivatives of its inverse are interpolated at the current position of the fiber head. In all subsequent implementations of the algorithm, the four tensors are interpolated element-wise using trilinear interpolation. Here, the voxels are treated as points, rather than volumetric elements, which are located in the center of their corresponding volumes. The eight voxels closest to the fiber head form a cubic *cell*, and the tensors of this cell's voxels are used in the interpolation process.
- Next, the Christoffel symbols (Equation 3.11) are calculated at the position of the fiber head, using the DTI tensor and the derivative tensors. Note that, since the original DTI tensor and the three derivative tensors of its inverse are all symmetric, some of the symbols are equal. Specifically, $\Gamma_{12}^\gamma = \Gamma_{21}^\gamma$, $\Gamma_{13}^\gamma = \Gamma_{31}^\gamma$, and $\Gamma_{23}^\gamma = \Gamma_{32}^\gamma$. This gives us 18 unique Christoffel symbols.
- Finally, the Christoffel symbols, along with the fiber's current position and direction, are used to compute the next position and direction. This is done using an Ordinary Differential Equation (ODE) solver, which solves the system of ODEs implemented by Equation 3.10. The ODE Solver consists of two first-order time integration steps, which are shown in Equations 3.14 and 3.15.

$$x'_\gamma = x_\gamma + h\dot{x}_\gamma \quad (3.14)$$

$$\dot{x}'_\gamma = \dot{x}_\gamma - h \sum_{\alpha=1}^3 \sum_{\beta=1}^3 \Gamma_{\alpha\beta}^\gamma \dot{x}_\alpha \dot{x}_\beta \quad (3.15)$$

Here, h is a scalar value that represents the step size. These equations effectively implement the Euler ODE solver. ODE solvers of higher order, such as the second- or fourth-order Runge-Kutta solver, can be used to increase the accuracy of the numerical integration scheme, at the cost of an increase in computational complexity.

3.4 Fiber Filtering

When a fiber computed by the geodesic fiber-tracking algorithm connects two different parts of the brain, this fiber does not necessarily correspond to a physical connection in the white matter. By varying the initial direction of a fiber, we can connect practically every point in an image to a single seed point; it is obvious that only a handful of these connections will be valid. Furthermore, the end-user of the algorithm will usually only be interested in a subset of all computed fibers, such as those that connect two user-defined regions in the white matter. A *filtering* technique is needed to remove fibers that are likely to be invalid and/or fibers that are of no interest to the end-user.

In this section, we describe two possible fiber filtering techniques: The Region-to-Region Connectivity method, and the Ray Tracing methods. Both methods aim to identify connections between two or more user-defined regions (or points) of interest. By only showing those computed fibers that are most likely to correspond to actual fibers (using a certain connectivity measure), they constitute a powerful tool for researchers, allowing them to explore the trajectories of fiber bundles in depth. We first discuss the Region-to-Region Connectivity method, which has also been integrated in our implementations of the geodesic fiber-tracking algorithm. The connectivity measure, which quantifies the "strength" of a fiber, is also introduced here. Subsequently, we discuss an alternative approach, the Ray Tracing method introduced by Sepasian *et al.* [55].

3.4.1 Region-to-Region Connectivity

The Region-to-Region Connectivity technique allows us to identify potential fiber bundles that connect two user-defined *Regions of Interest* (ROI). From a number of seed points located within one ROI, we track fiber trajectories using the described methods, and we check whether these fibers intersect with the other ROI. By subsequently only visualizing those fibers that establish a connection between the two regions, we allow the user to intuitively identify the structural topology of the white matter.

One important consideration for this technique lies with the initial direction of the fibers. An intuitive choice for the starting direction is to use the main eigenvector of the DTI Tensor at the position of the seed point. However, as established in Chapter 2.1, the influences of acquisition noise and the partial volume effect introduce an element of uncertainty in the direction and magnitude of the eigenvectors. In order to increase the chances of finding a valid connection between the two regions, we therefore track a large number of fibers, each with a different initial direction, from each seed point. The distribution of initial directions can either be uniform (i.e., in all directions) or based on the local tensor data (e.g., centered around the main eigenvector).

One downside of large numbers of shooting angles, besides the increase in computational complexity, is the fact that erroneous connections between the two regions may occur. While the ability to compute multiple possible connections between regions is one of the main advantages of the described geodesic fiber-tracking algorithm, we do not want to visualize connections that are obviously wrong. In order to be able to quantify the probability that a computed trajectory actually corresponds to a fiber in the white matter, we use a *connectivity measure*, which is a scalar value defined at each point of the computed fiber. By taking the connectivity measure values at the target region and sorting them, we are able to differentiate between strong fibers and weak fibers.

Astola *et al.* [3] discuss a connectivity measure that uses the ratio between the Euclidian and geodesic length of the fiber. Sepasian *et al.* [54] demonstrate that this connectivity measure is indeed able to differentiate between strong and weak connections in synthetic data. Recall the definition of the geodesic length, i.e.,

$$L(\mathbf{x}) = \int_0^T \sqrt{\dot{\mathbf{x}}^T G \dot{\mathbf{x}}} d\tau.$$

The proposed connectivity measure is obtained by computing the ratio between the Euclidian length and this geodesic length, i.e.,

$$CM(\mathbf{x}) = \frac{\int_0^T \sqrt{\dot{\mathbf{x}}^T \dot{\mathbf{x}}} d\tau}{\int_0^T \sqrt{\dot{\mathbf{x}}^T G \dot{\mathbf{x}}} d\tau}. \quad (3.16)$$

By keeping track of the current Euclidian and geodesic length, this measure can be computed for every point along the fiber. The sorting process, which removes weak connections, is then performed as a post-processing step. The Region-to-Region Connectivity method is illustrated in Figure 3.3.

3.4.2 Two-Point Ray Tracing Method

An alternative approach to finding potential connections between two regions is the Two-Point Ray Tracing method proposed by Sepasian *et al.* [55]. Here, the two Regions of Interest between which we want to identify connectivity are shrunk down to two points of interest. From each point, we again track a large number of fibers in all directions. When such a fiber reaches the boundary of the volume, we register its location and the angle under which the line left the volume. After doing so for both points, we plot a graph of the positions and angles of escape of all fibers, and find the points of intersection between the two lines. We then reason that the geodesic corresponding to a point of intersection passes through both points of interest, and we can backtrack these fibers to find all possible connections. Finally, we can use the connectivity measure described in the previous section to rank the connecting lines and filter out the weak ones. The method is illustrated in Figure 3.4.

The Two-Point Ray Tracing method suffers from two main disadvantages. First, the computational complexity of the method illustrated in Figure 3.4 would increase sharply were we to apply it in 3D. A 3D implementation would require a higher-order parameterization of the escape location and angle, making the computation of the points of intersection far more expensive. We argue that the main computational advantage of Two-Point Ray Tracing over the Region-to-Region Connectivity method - the fact that we do not need to check for intersection with a target region at every integration step - may be nullified completely by the increased complexity of the post-processing computations.

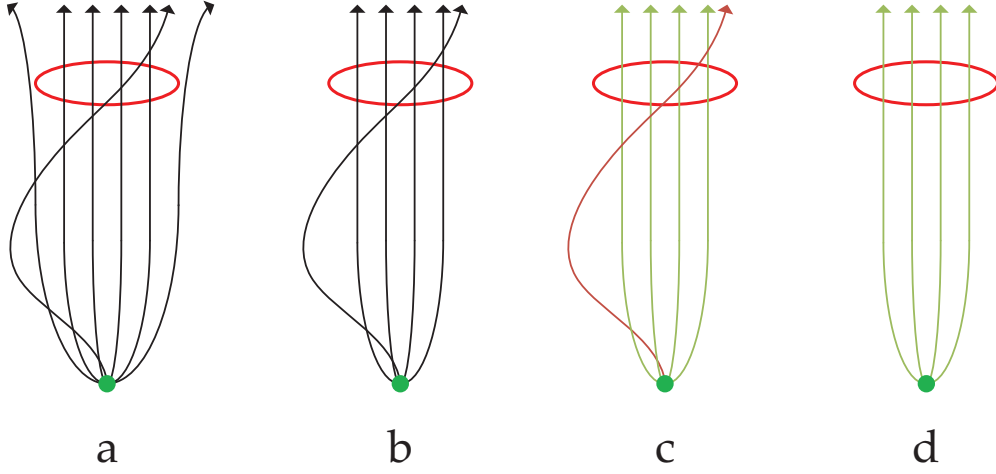


Figure 3.3: Illustration of the connectivity measure filtering of a set of fibers (a) starting at the green seed point. After the first step, only those fibers that intersect the red target region remain (b). Next, the connectivity measure is computed for the remaining fibers (c). Finally, the weak connections are removed, leaving only those fibers that are likely to correspond to actual fibers in the white matter (d).

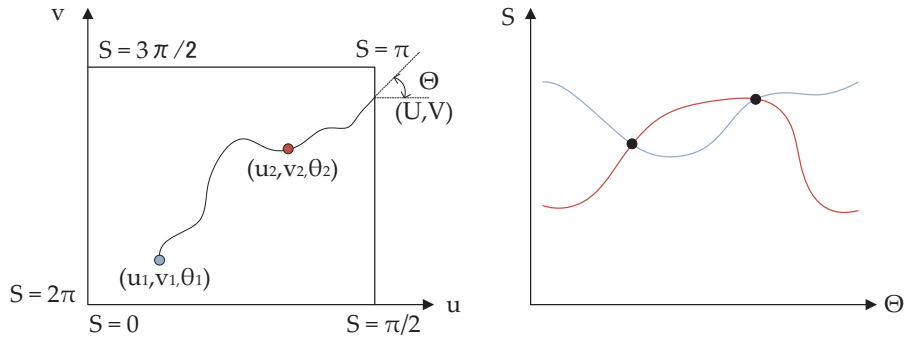


Figure 3.4: Illustration of the Two-Point Ray Tracing Method in 2D. The left graph shows an area with two points interest with position (u, v) and initial direction θ , and a single fiber that passes through both points. When the fiber leaves the area, the angle of escape Θ and the position, mapped to the scalar value $S = [0, 2\pi)$, are registered. On the right, the graph is shown that can be obtained by registering S and Θ for all fibers. The marked points of intersection correspond to lines that connect the two points. Note that every connecting line will produce two points of intersection.

The second, more important disadvantage is that there is little practical purpose to finding the exact connection between two points. Establishing a strong connection between two

points is only possible if the two points are located on the exact same fiber bundle; if the points are not physically connected in white matter, we may find weak connections that do not correspond to physical connections, and as such provide either no information, or erroneous information. By contrast, the Region-to-Region Connectivity method allows the user to identify strong connections between regions, the size and shape of which are determined by the user. As such, it provides far more flexibility and a larger margin of error than the Two-Point Ray Tracing method. For these reasons, we use the Region-to-Region Connectivity method in our implementations.

3.5 Summary

In this chapter, we introduced the geodesic fiber-tracking algorithm, which is the focus of the research presented in this paper. This algorithm uses the tensors obtained through Diffusion Tensor Imaging, which describe the diffusion of water in a small volume called a voxel. Using the knowledge that diffusion is anisotropic in fibrous tissue, such as the white matter in the brain, we can use these DTI tensors to infer the trajectories of fibers within the white matter. The geodesic fiber-tracking algorithm aims to find the shortest path (i.e., the geodesic) through a Riemannian manifold. The algorithm computes fiber trajectories through an iterative procedure which numerically solves a set of Ordinary Differential Equations. Additionally, we presented two possible applications of geodesic fiber-tracking, which both aim to find potential connections between user-defined regions.

Chapter 4

CUDA Implementation

One of the main disadvantages of the geodesic fiber-tracking algorithm described in the previous chapter is that it is relatively slow. Between the interpolation of 24 floating-point values, the computation of 18 Christoffel symbols, and the execution of the integration step, the algorithm has a significantly larger computational load than, for instance, the simple streamline tracking algorithm, which only requires the computation of the main eigenvector followed by a simple integration step. Testbenching shows that a C++ implementation of the algorithm described in Section 3.3 has a throughput of roughly 750,000 integration steps per second on one of the cores of an Intel Core i5 processor with a clock frequency of 2.67 GHz. A more detailed overview of the testbench results for the CPU implementation is provided in Section 5.4.

Practical applications of the geodesic fiber-tracking algorithm, such as the Ray-Tracing Connectivity method and the Region-to-Region Connectivity method (both introduced in Section 3.4), require hundreds of geodesic fibers, all of which may be tracked for a distance of several thousand integration steps, the time it takes to compute all these fiber may become unacceptably large. The remainder of this report focuses on an implementation of the geodesic fiber-tracking algorithm in NVIDIA's *CUDA*, which was developed with the aim of significantly speeding up the algorithm.

This chapter first gives an introduction of *CUDA*, including an overview of important design decisions. Next, the structure of the *CUDA* implementation of the geodesic fiber-tracking algorithm is discussed. Subsequent sections talk in more detail about various aspects of the algorithm, optimization strategies and design decisions relevant to our implementation.

4.1 Introduction to *CUDA*

The *Compute Unified Device Architecture* (*CUDA*) was introduced by NVIDIA in February 2007. It is a parallel computing architecture that allows programmers to develop algorithm for execution on modern Graphics Processing Units (GPU) produced by NVIDIA. The algorithms are programmed in the form of *kernels* using the C/C++ language with a set of *CUDA*-specific extensions. During execution, the computational instructions of a kernel are performed in parallel by a number of *threads*, each of which works on a small range of the input data. The threads are grouped together on multiprocessors, which in turn contain a number of streaming processors. In this way, *CUDA* provides access to a layered system of multiprocessors and memory spaces. By meaningfully parallelizing an algorithm in such a

way that it can exploit the advantages offered by CUDA, a programmer can use this structure to speed up computational tasks of many different varieties.

4.1.1 Similar Technologies

CUDA allows for *General-purpose computing on graphics processing units* (commonly abbreviated as GPGPU), meaning that the computational tasks performed by algorithms programmed using CUDA are not necessarily related to the processing of graphics. Other GPGPU techniques include using a Graphics API, such as DirectX or OpenGL, to perform the computations with *shaders*, and using one of the other available GPGPU languages, such as OpenCL and DirectCompute.

Shaders are highly parallel algorithms that are used in the rendering pipeline of Graphics API's. The two most commonly used Graphics API's, Direct3D and OpenGL, both use three different types of shaders: *Vertex shaders*, which work on individual vertices of the 3D data; *Geometry shaders*, which modify a group of vertices (called a *primitive* or *mesh*, depending on the context); and *Pixel shaders* (or *Fragment shaders*), which create and modify elements of the output 2D image. Figure 4.1 shows the Direct3D 10 pipeline, which uses these three shader types in the order listed.

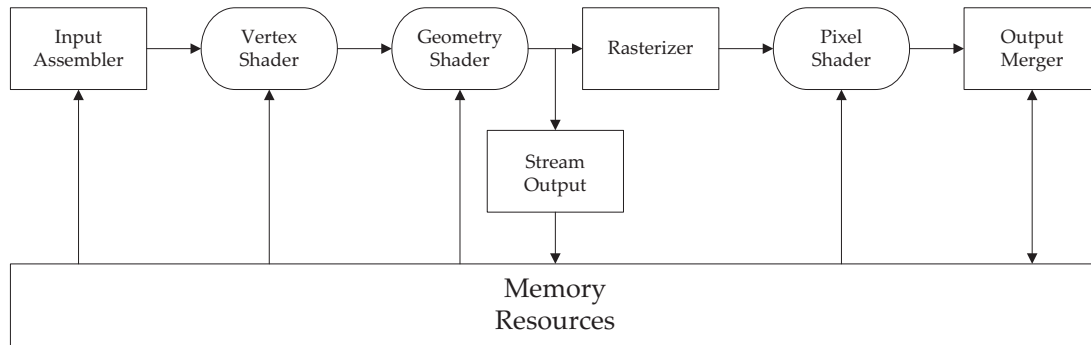


Figure 4.1: Rendering pipeline of Direct3D 10. [34]

Traditionally, these shaders are used to generate graphics: using sets of vertices, textures and other data as input, a 2D image is constructed by executing the shaders in the order specified by the rendering pipeline, after which the image is sent to the screen buffer to be displayed. However, it is also possible to use the parallel computational abilities of these shaders to perform computations that are not necessarily directly related to graphics, such as the construction of fibers from DTI data. Using one of several available shader languages, programmers can create custom shaders that can execute very specific computational tasks while still exploiting the inherent parallelism of the GPU.

When compared to CUDA, shaders have several distinct disadvantages. The first disadvantage is that debugging shaders is far more challenging than debugging CUDA. CUDA includes a device emulation mode, which allows the user to inspect the values of thread variables during run-time. Shader languages generally lack full-featured debug environments, which makes the debugging process less straightforward. This also contributes to the second disadvantage, which is a steeper learning curve. CUDA allows a programmer to parallelize C code using a handful of language extensions, while shader language often require a larger

effort in the areas of code conversion and algorithm (re)design.

An important disadvantage of shaders is that they are generally less flexible than CUDA. When using CUDA, programmers are able to share data between threads and use random-access memory, and these features are not available for shaders, not even when the hardware does support them. In addition, it should be noted that in recent years, the architecture of GPU's has moved away from that of a rendering pipeline optimized for graphics processing, and toward a many-core architecture of generic processors. CUDA, which has been designed specifically for modern graphics card architectures, is able to fully benefit from this shift in design philosophy.

For these reasons, we chose to use CUDA over one of the available shader languages. Please note that this choice was made based on the ease of use and general applicability of these languages, and not on the performance possibilities of either option. In other words, we do not claim that our CUDA implementation is guaranteed to be faster than an implementation using a shader language.

The second group of CUDA alternatives consists of other GPGPU languages, the two main competitors being the Open Computation Language (OpenCL) by the Khronos Group, and Microsoft's DirectCompute. Once again, the reason for choosing CUDA over either alternative lies with the availability of expertise, both internal and on the internet, rather than the performance characteristics of these languages.

4.1.2 Architecture

A CUDA-enabled GPU consists of a large block of RAM called the *device memory*, which is usually 512 MB or 1 GB on modern GPU's, a number of multiprocessors, and some additional memory elements. On the actual GPU, multiprocessors are sometimes grouped in blocks called *Thread Processing Clusters* (TPC's) [37], but this subdivision is neither visible to nor relevant for the CUDA programmer.

The GPU architecture contains a number of different memory spaces, as shown in Figure 4.2. The main memory module of the GPU is a large DRAM block, which is between 512 MB and 2 GB in size on modern devices. This is commonly referred to as the *device memory*. When reading from or writing to the device memory, threads experience a memory access delay of roughly 400 to 600 clock cycles [39]. Four different memory spaces reside within the device memory:

- **Global Memory** works as an uncached RAM block. All threads can write to and read from any location in global memory.
- **Texture Memory** is a read-only memory space that can be accessed by all threads. Texture reads are cached (with a cache size of 8 to 16kB per multiprocessor [15]), allow for efficient interpolation of image data. As such, it offers several possible advantages over global memory. See Section 4.6 for an overview of the features of texture memory.
- **Constant Memory** is a small (64 kB), read-only memory space that can be accessed by all threads, and which has a cache of 8 kB per multiprocessor. Due to its small size and the fact that it is cached, constant memory is best suited for constant values that are used in all threads, such as look-up tables.
- **Local Memory** is used to store the variables of a thread that cannot be stored in the multiprocessor's registry file or shared memory. It is local only in the sense that each

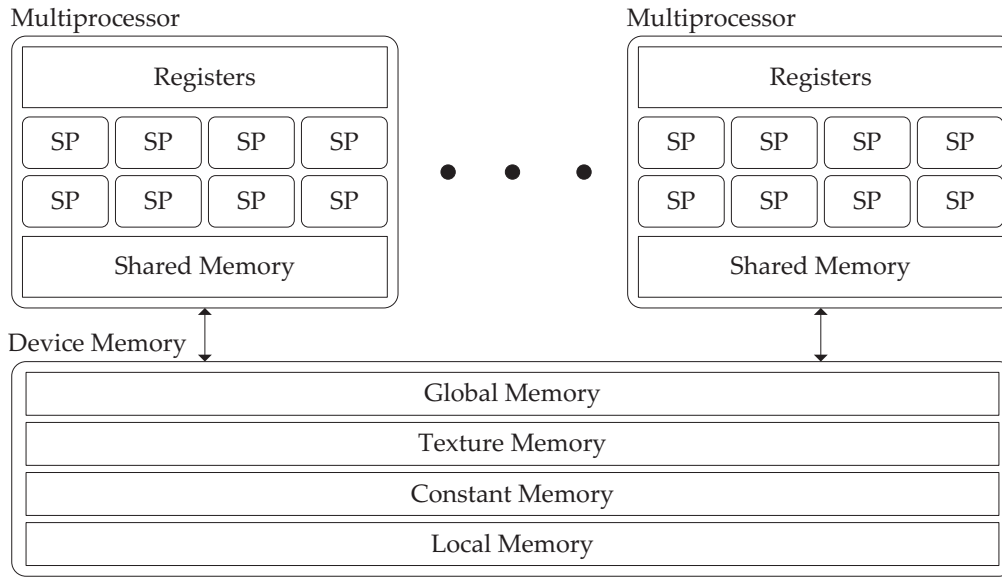


Figure 4.2: Memory structure of CUDA. A distinction is made between memory spaces residing in the device memory of the GPU, and on-chip memory local to the multiprocessors. On the GTX 260, each multiprocessor contains eight Scalar Processors (SP). Cache blocks for the Texture Memory and Constant Memory are not shown, nor are the instruction units of the multiprocessors.

group of thread variables in local memory can only be accessed by the thread they are assigned to. The compiler automatically assigns variables to local memory based on the amount of memory required per thread.

In addition to these four memory spaces and the caches of texture and constant memory, there are two memory spaces that are local to the multiprocessors.

- **Registers** are used to store the thread variables. Accessing a register normally consumes zero extra clock cycles per instruction, although additional latencies may be caused by bank conflicts and read-after-write dependencies.
- **Shared Memory** is a 16 kB block of memory in each multiprocessor that can be used to share data between the threads in one thread block. Access latencies for shared memory are in the order of 100 times lower than those for the device memory. Shared memory is divided into a number of banks; additional access latencies may be caused by bank conflicts.

Each multiprocessor consists of eight *Scalar Processors* (SP's), the shared memory and register file, and an instruction unit. When a block of threads is assigned to a multiprocessor, the instruction unit first groups these threads in sub-blocks called *warps*, each consisting of 32 threads (the term *half-warp* is used to indicate either the first or last 16 threads of a warp). All threads in a warp are then executed using the *Single Instruction Multiple Threads* (SIMT) architecture. This essentially means that all threads in the active warp execute the same instruction at the same time. This type of parallelism is similar to *Single Instruction Multiple*

Data (SIMD), the main difference being that SIMT allows for greater thread-level independence than SIMD. Ideally, all threads in a warp will follow the same execution path, but the scheduler also allows for diverging execution paths within a warp. While having branching execution paths does come at a certain performance cost, it does give the programmer more freedom, as it allows him to program more complex threads. The consequences of having diverging execution paths are discussed in more detail Section 4.1.3.

The size and dimension of each thread block can be set by the programmer. In addition, these thread blocks are arranged in a *grid*, the size and dimension of which can also be set by the programmer. Within a block, each thread has a unique *thread index*, which is the 1D, 2D or 3D integer coordinate (depending on the dimensionality of the thread blocks) of the thread within its block. The location of the thread block within the grid determines its *block index*, which can be seen as the 1D or 2D integer coordinates of the block within the grid (3D grids are not supported in the current version of CUDA). The total number of threads executing a single kernel is $N = (\text{number of threads per block}) * (\text{number of block in the grid})$. By combining the thread index, block index and block size, a unique global thread index between 0 and $(N - 1)$ can be computed for each thread. This grouping of threads into blocks, as well as the scope of the memory spaces available to individual threads, is shown in Figure 4.3.

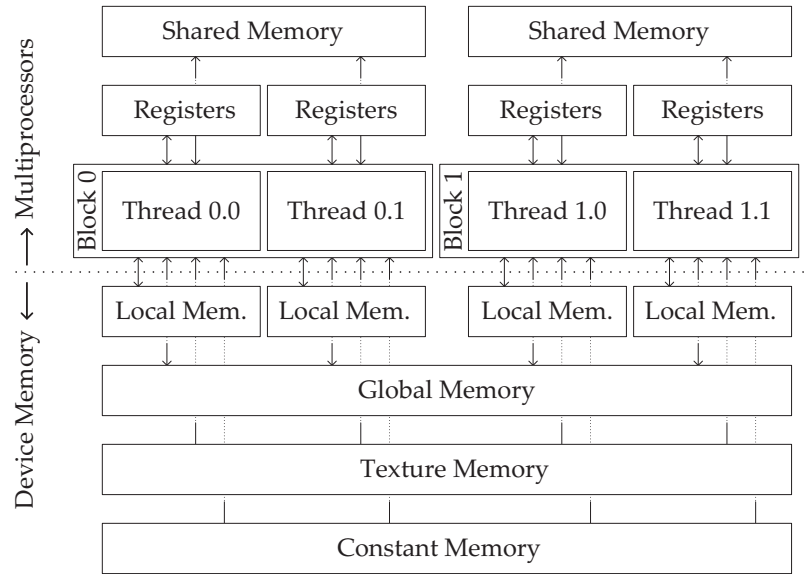


Figure 4.3: Scope of the various memory spaces. Local Memory and the Registers are both local to a single thread, and are free when this thread terminates. Shared Memory is shared between all threads in a single block, and stays allocated until all threads in this block have terminated. The three remaining memory spaces - Global, Texture and Constant - are available to all threads in all blocks, with the last two being read-only.

The control flow of a CUDA program is divided between the CPU and the GPU. The following steps are commonly performed during execution of such a program:

1. The CPU loads or generates an input data set.

2. The input data is copied to the device memory of the GPU.
3. The CPU launches a kernel, which is executed by threads on the GPU.
4. Each thread computes its identifier, based on the index of the thread block it is in, and its own index within the block.
5. Based on its identifier, the thread loads a small part of the input data from the device memory.
6. The thread performs a number of computations on the input data. If necessary, the thread communicates with other threads in its block through the shared memory.
7. The result of the computations is written back to the device memory, and the thread terminates.
8. When all threads are finished, the CPU copies the results to the working memory of the CPU.

4.1.3 Design Considerations

While the relevant design consideration for CUDA differ for each algorithm, there is a set of general guidelines which may be used to ensure optimal performance. These guidelines are discussed in more detail in the *Best Practices* guide for CUDA [39]. Some relevant guidelines are discussed below.

- In general, programmers should try to maximize the amount of threads per multiprocessor. Using the *Occupancy Calculator* made by NVIDIA [36], programmers can optimize the *occupancy* of their algorithm. The occupancy measure corresponds to the percentage of the GPU's maximal computation throughput that a kernel will achieve, and is based on 1) the number of threads per thread block, 2) the number of registers used per thread, and 3) the amount of shared memory used per thread block. Generally speaking, obtaining a high occupancy is one of the main optimization strategies for CUDA. However, it should be noted that the occupancy measure does not incorporate a number of factors that may still influence performance. For example, the amount of memory loaded from global memory by each thread does not influence the occupancy, but may very well influence overall performance when the memory throughput is a bottleneck.
- The throughput between the device memory and the multiprocessor should be minimized. Since accesses to the device memory from the multiprocessor have a latency of about 400 to 600 clock cycles [39], reading more data than necessary from the device memory may result in unnecessary delays in the computation of the threads. In addition, the memory throughput between the device memory and the processors may become a bottleneck when a large number of threads want to read and/or write large amounts of data. Programmers can try to remove redundant memory reads by storing the data locally, and by sharing read data between threads using the shared memory.
- Another important consideration is related to the total amount of memory required by each thread. The maximum number of threads per multiprocessor is limited by the amount of registers per thread, and the total amount of shared memory per thread

block. For example, when each block of threads requires 8 kB of shared memory to share data between the threads in the block, the 16 kB size of the shared memory limits the number of thread blocks per multiprocessor to two. Similarly, the register file - consisting of 8,192 or 16,384 32-bit registers, depending on what GPU is used - can only accommodate a number of active threads. From this, it might be concluded that performance can be increased by lowering the maximal number of registers per thread, as doing so may result in a larger number of active threads, and thus a higher occupancy. However, thread variables that cannot be stored in registers are usually stored in *local memory*. Since local memory resides within the device memory, threads with a large amount of data in local memory may experience severe performance penalties. In the end, increasing the maximum number of registers per thread (thereby potentially lowering the occupancy) may give better results than having a larger number of threads with fewer registers. This trade-off is illustrated in Figure 4.4.

- A number of instruction-related optimization strategies with varying levels of importance exist. One very important strategy is to prevent divergence in the execution paths of the thread within one warp. The multiprocessors in CUDA use the SIMT architecture, which means all threads in a warp execute the same instruction in the same cycle, but on different data. When the execution path diverges - for example because one of the threads enters an `if`-statement - this parallelism can no longer be applied, and the different execution paths must be executed sequentially. This increases the total number of instructions, and may therefore increase the total running time of the algorithm. This process is illustrated in Figure 4.5.
- Data transfer between the CPU and the GPU are slow compared to in-GPU memory transfers, and should therefore be kept to a bare minimum.
- Reads from and writes to the device memory should be *coalesced* where possible. This essentially means that all threads within a half warp (i.e. 16 threads) read from or write to the same block of device memory. These memory accesses are then automatically grouped (coalesced) into a single memory access, which can greatly reduce the amount of memory throughput required by the algorithm. The performance penalty for using non-coalesced memory access may be reduced by using texture memory instead of global memory; see Section 4.6 for details.

The impact of these design considerations on our geodesic fiber-tracking algorithm will be discussed in detail in the relevant sections below.

4.2 Implementation Overview

This section discusses the implementation of the geodesic fiber-tracking algorithm introduced in Section 3.3 in CUDA. As discussed in the introduction of this chapter, the motivation for this implementation is the fact that the geodesic fiber-tracking algorithm is computationally expensive, leading to long running times on single-core processors. We first discuss the specifications of our algorithm in terms of its required functionality. Subsequently, the structure of our CUDA implementation is described. The kernels that implement these different steps of our algorithm are discussed in detail, using pseudo-code to illustrate their computational structure.

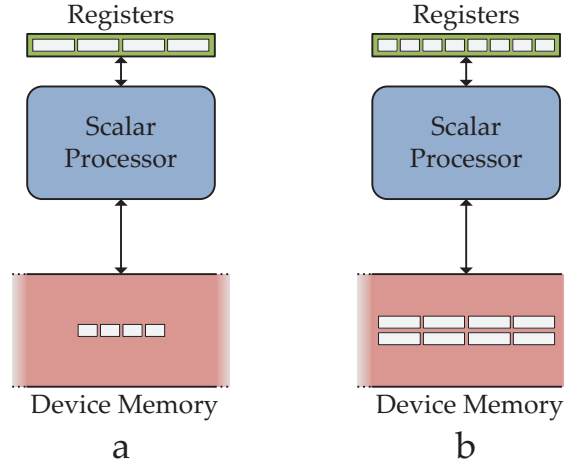


Figure 4.4: Exaggerated example of the possible impact that changing the maximum number of registers per thread can have on the performance of the algorithm. A gray block represents the variables of a single thread in the scalar processor. In situation *a*, the high number of registers per thread limits the number of threads per block, which in turn lowers the occupancy, possibly leading to longer running times. In situation *b*, a lower number of registers per thread allow for more concurrent threads, but also results in a larger amount of memory per thread in local memory, which may become a bottleneck. For certain algorithms, situation *a* may therefore be preferable to situation *b*.

4.2.1 Specifications

The aim of the CUDA implementation of the algorithm is to compute a large number of geodesic fibers using DTI data. Visualization of the computed fibers is *not* included in the algorithm described below. In our project, the vertices describing the fibers were transferred from the GPU to the CPU, after which a separate routine handled the visualization. An additional reason for not including a visualization module is that not all possible applications of geodesic fiber-tracking require visualization of all resulting fibers. For example, when applying one of the post-processing steps described in Section 3.4, the fiber data must be filtered before the results can be visualized. Finally, we argue that the existing CUDA implementation may be extended to included visualization features, should this be necessary. The possibilities for including visualization techniques are discussed in Section 6.1.

As the research presented herein constitutes the first attempt to accelerate geodesic fiber-tracking algorithm under discussion, and as there are no clearly defined performance requirements for the possible applications of this algorithm, it is not possible to set a goal for the performance. Since we cannot meaningfully compare its performance to a set requirement or to the performance of similar implementations, we instead aim simply to make it “as fast as possible” by optimally utilizing the advantages offered by CUDA while dealing with its restrictions. In this section and Chapter 5, we aim to demonstrate that our CUDA implementation makes good use of the parallel architecture of GPU.

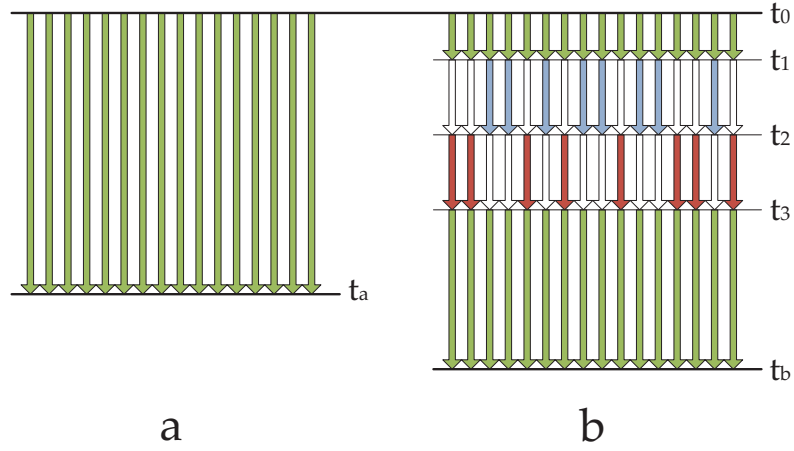


Figure 4.5: Illustration of the effects of execution path divergence (with 16 threads per warp instead of the actual 32). In *a*, all threads follow the same execution path (green), meaning they all execute the same instruction at the same moment. In *b*, there is a branch in the instruction path at $t = t_1$. Subsequently, the two distinct paths (red and blue) following the branch are executed sequentially, until all threads converge back to the same path. During this time, some of the threads will be inactive while another path branch is being executed (white). This sequential execution results in a higher execution time.

4.2.2 Structure

In Section 3.3, the numerical approach for our algorithm was described. We divided the algorithm into two stages:

- The *data-preparation stage*, which computes the pre-processed tensor and the derivatives of its inverse.
- The *tracking stage*, in which we use the pre-preprocessed tensors and the derivatives to compute the trajectories of the fibers.

We have implemented these two stages using three different CUDA kernels: two for the data-preparation stage (the *Pre-Processing kernel* and the *Derivatives kernel*), and one for the tracking stage. The derivation kernel computes the derivatives of the inverse tensors in a single dimension, meaning that in order to compute the derivatives in all dimensions, we need to invoke this kernel three times. The decision to split the data-preparation stage into two kernels with a total of four kernel calls, rather than a single kernel that is executed only once, was motivated by the read-only nature of textures, which are introduced in Section 4.6. Specifically, Section 4.6.7 discusses the advantages of using four kernel invocations instead of one.

The control flow for the three CUDA kernels is shown in Figure 4.6. As we can see, this is very similar to the structure shown in Figure 3.2 in Section 3.3. The main difference is the lack of a separate step for inverting the pre-processed tensors. While we do of course still invert the tensors, we have integrated this step into the Derivatives kernel out of consideration

for the total memory usage in the device memory of the GPU. Computing and storing the inverse of all tensors would require a large amount of device memory (equal to the memory requirements of the input image), and since device memory is strictly limited on the GPU, we have opted not to store the inverse tensors in device memory, instead computing them on-the-fly when computing the derivatives. Doing so increases the number of computations in the Derivatives kernel; however, since the running time of the Derivatives kernel is low compared to the running time of the tracking kernel (see Sections 5.2.2 and 5.3), and since the data-preparation stage only needs to be re-executed when the input data and/or pre-processing parameters change, the impact on the total running time will be small.

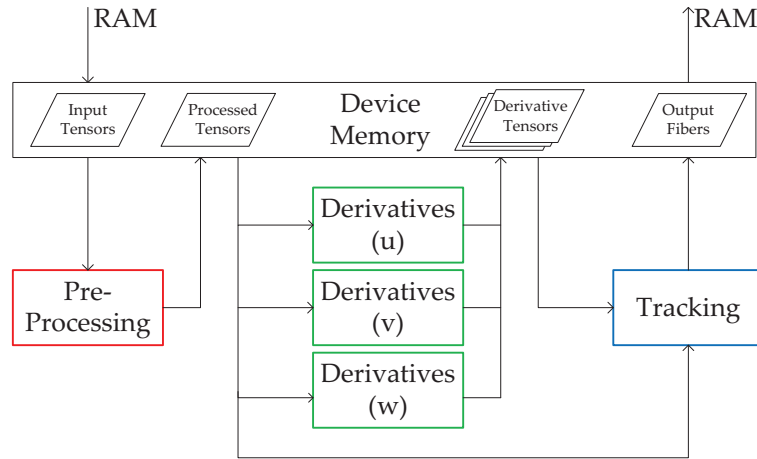


Figure 4.6: The controls flow of our CUDA implementation. Each of the colored rectangles represents an invocation of a single kernel: The Pre-Processing kernel (red), the Derivatives kernel (green, one invocation per dimension), and the Tracking kernel (blue). The contents of the device memory are also shown, although it should be noted that the final algorithm uses a slightly different memory management scheme, as described in Section 4.6.7.

4.3 Pre-Processing Kernel

The pre-processing step can very easily be parallelized, as the computations performed on a single tensor are completely independent of those performed on the tensors around it. Translating this to the context of CUDA, this means that we can use a single thread per voxel, which loads the voxel's tensor from the device memory, pre-processes this tensor based on a set of user-defined parameters, and writes the resulting output tensor back to the device memory. Since individual threads at no point need to communicate or wait for other threads, and since all threads execute the exact same code, we are able to achieve high levels of parallelism. This process is illustrated in Figure 4.7.

As described in Section 3.3, the pre-processing step consists of three distinct actions:

- Application of a constant gain to all elements in the tensor.

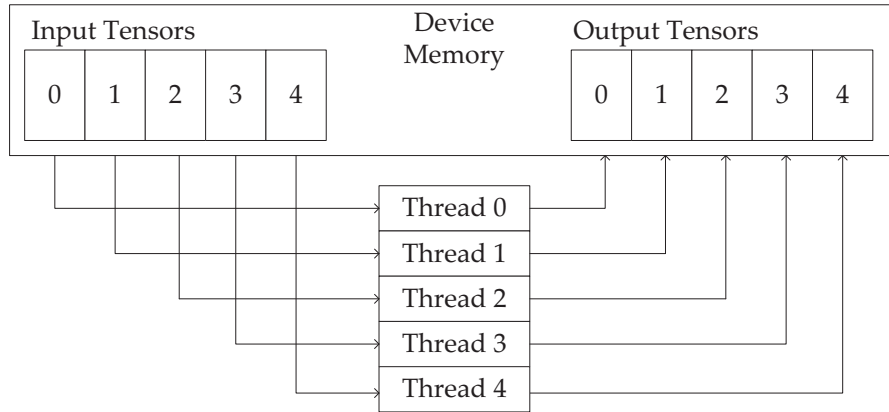


Figure 4.7: Illustration of the thread-level parallelism of the Pre-Processing kernel, where each thread processes one input tensor, and writes the pre-processed tensor back to the device memory.

- Checking if the tensor can be inverted, and if it is positive-definite. If either check fails, replace the tensor with the identity tensor. As mentioned in Section 3.3.1, over 99.9% of tensors for which these conditions do not hold only have elements equal to zero. Replacing these tensors with the identity tensor makes them nonsingular while creating a region of minimal anisotropy.
- If required, sharpen the tensor by exponentiating it using a user-defined exponent. Sharpening is only performed when the anisotropy measure of the tensor (see Section 2.1.4) is lower than a user-defined threshold value.

We now give the pseudo-code program that implements these three steps. Let $D(x)$ be the input DTI tensor at the coordinates $x = (u, v, w)$, and let idx be the thread index, which is an integer value between 0 and N , where N is the total number of voxels in the image. Finally, let $O(x)$ be the output tensor. Algorithm 1 shows the pseudo-code for the Pre-Processing kernel. Here, A , B and C are intermediate tensors. The practical implementations of the functions in the pseudo-code are discussed below.

Algorithm 1

(* Pre-processes DTI tensors. *)

1. $[u, v, w] \leftarrow \text{ComputeCoordinates}(idx)$
2. $A \leftarrow D([u, v, w])$
3. $B \leftarrow A * \text{TensorGain}$
4. **if** B is singular **or** B is not positive definite
5. $B \leftarrow 3 \times 3$ identity tensor
6. $FA \leftarrow \text{ComputeFractionalAnisotropy}(B)$
7. **if** $FA < \text{Threshold}$
8. $C \leftarrow \text{SharpenTensor}(B, \text{TensorExponent})$
9. $O([u, v, w]) \leftarrow C$

- We test for singularity by computing the determinant of \mathbb{B} and checking whether or not it is equal to zero (or close to zero, depending on a small margin of error).
- Testing for positive-definiteness is done using Sylvester's criterion, which states that a square matrix is positive-definite if and only if all leading principal minors (i.e., the determinant of the tensor, the determinant of the 2×2 upper-left sub-matrix, and the top-left element) are all positive.
- Computing the Fractional Anisotropy as defined by Equation 2.3 is done using the method described by Hasan *et al.* [17]. In this method, the fractional anisotropy is directly calculated from the tensor elements, without first having to compute its eigenvalues. Specifically, $FA = \sqrt{1 - I_2 I_4^{-1}}$, where $I_4 = I_1^2 - 2I_2$, $I_1 = D_{11} + D_{22} + D_{33}$, and $I_2 = D_{11}D_{22} + D_{11}D_{33} + D_{22}D_{33} - D_{12}^2 - D_{13}^2 - D_{23}^2$.
- Sharpening the tensor through exponentiation is done by multiplying the tensor by itself a number of times. For example, when the exponent is 3, we essentially compute $C = B * B * B$.

4.4 Derivatives Kernel

As shown in Figure 4.6, we use the same Derivatives kernel three times to compute the derivative of the inverse DTI tensor in all three dimensions. Since derivation works the same way in all directions, we can accomplish this by simply including an integration variable `dir` (with `dir = 0, 1, 2`), which determines the direction of derivation. For example, if (i, j, k) is the 3D index of the current voxel, the derivation kernel will use the voxels with indices $(i+1, j, k)$ and $(i-1, j, k)$ when `dir` is zero, and those with indices $(i, j+1, k)$ and $(i, j-1, k)$ when `dir` is one.

Our approach for the Derivatives kernel is to have each kernel compute the derivative for a single voxel. To do so, it will need to load and invert two pre-processed tensors. If the target voxel has a voxel on either side in the derivation direction, we use the two-sided derivation scheme as seen in Equation 3.12; if it is located on a border, we instead use one-sided interpolation, see Equation 3.13. The inverse is calculated by dividing the matrix of cofactors of a tensor by the determinant of that tensor. The pseudo-code for this kernel is shown in Algorithm 2.

Algorithm 2

(* Computes the derivatives of inverse DTI tensors. *)

1. **if** voxel is located on lower boundary
2. $A \leftarrow \text{GetTensor}(\text{this voxel})$
3. $B \leftarrow \text{GetTensor}(\text{next voxel})$
4. $O \leftarrow (\text{inv}(B) - \text{inv}(A)) / d[\text{dir}]$
5. **else**
6. **if** voxel is located on upper boundary
7. $A \leftarrow \text{GetTensor}(\text{this voxel})$
8. $B \leftarrow \text{GetTensor}(\text{previous voxel})$
9. $O \leftarrow (\text{inv}(A) - \text{inv}(B)) / d[\text{dir}]$
10. **else**
11. $A \leftarrow \text{GetTensor}(\text{next voxel})$

```

12.          B ← GetTensor(previous voxel)
13.          O ← (inv(A) - inv(B)) / (2 * d[dir])

```

Here, `next voxel` and `previous voxel` refer to the neighboring voxel in positive and negative direction, respectively, along the direction of derivation, while `d[dir]` is the distance between voxels in this direction. Subtraction and subsequent division of the inverse tensors are performed element-wise.

4.5 Tracking Kernel

While the previous two kernels allowed for a straightforward, intuitive translation between the elements of an image and the threads of the corresponding CUDA program, a good parallelization scheme is less obvious for the tracking kernel. Specifically, we want to determine what each thread should calculate, using what data.

Recall that fiber trajectories are computed using a series of integration steps, each solving a numerical integration step to compute the next location and direction of the fiber, based on its current location and direction and the surrounding image data. The fact that these steps are by definition executed sequentially makes step-level parallelism impossible.

Furthermore, the three actions executed within each step - interpolating the image data at the current location, computing the Christoffel symbols, and solving the integration step - also have to be performed sequentially. One could argue that computationally intensive tasks, such as the computation of the Christoffel symbols, could be split between several threads, but the overhead resulting from communication between and synchronization of these threads would likely negate any possible improvement of the running time. Communication between threads can only be done through the shared memory, the limited size of which will limit the maximum number of active threads, and the synchronization barriers needed to prevent memory conflicts will reduce the computational throughput of the algorithm.

We therefore conclude that each thread should track a single fiber. Most applications of geodesic fiber-tracking require the computation of several thousand fiber trajectories (see Section 3.4), and since the computation of one fiber is completely independent of the computation of other fibers, we can run large numbers of threads in parallel without the need for synchronization. Each thread loads the data of a single seed point from the device memory, after which it traces this fiber for a certain length inside a big loop, loading the surrounding image data and interpolating it as necessary. Each thread writes its output - three coordinates per point along the fiber - to a specific location in the device memory. Since threads cannot dynamically allocate memory on the GPU, we limit the number of integration steps, meaning that threads either terminate when they have performed a set number of steps, or when their fiber has left the image volume.

4.5.1 Tile-Based Approach

One possible strategy would be to group the threads based on the position of the fibers. If all fibers corresponding to the threads within a single thread block are located within the same region of the input image (i.e. the same *tile*, we can load this region into the memory, allowing the thread in this block to access this data, thus removing the number of redundant reads from the device memory. This process is illustrated in Figure 4.8.

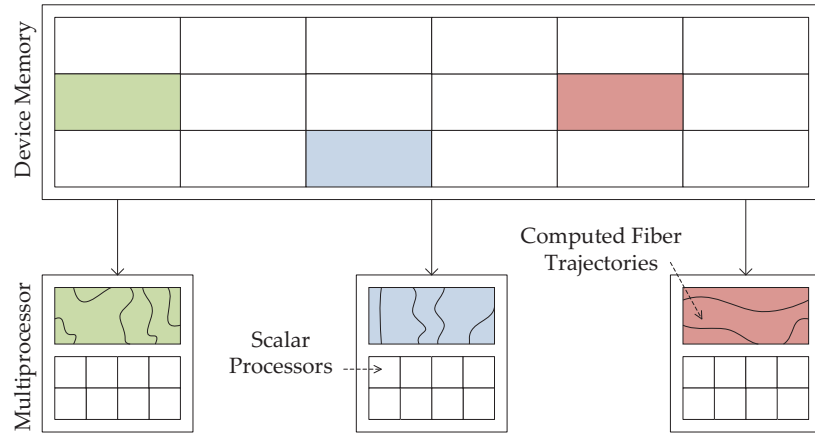


Figure 4.8: Illustration of the Tile-Based Approach. Each multiprocessor loads a small sub-part of the image (i.e. a *tile* into its shared memory, after which all threads trace their corresponding fibers through this tile.

There are, however, two problems with this approach. First of all, the shared memory of a multiprocessor is only 16 kB. Since we need to store four tensors per voxel (the pre-processed tensor and the three derivatives of its inverse), each voxel takes up $4 \text{ tensors} * 6 \text{ unique elements per tensor} * 4 \text{ bytes per element} = 96 \text{ bytes of data}$. This means that the shared memory can hold roughly 170 voxels; if we were to store a cubic part of the image in shared memory, this sub-volume could be at most 5 voxels wide on each side.

The second problem with this approach is that it would be impossible to guarantee reasonable levels of occupancy. As mentioned in Section 4.1.3, the occupancy of a CUDA algorithm is an important measure for its performance. This occupancy depends directly on the number of active threads in a thread block, and there is no way to guarantee that a sub-volume (particularly a sub-volume of at most $5 \times 5 \times 5$ voxels) will always contain at least a certain number of active fibers. In fact, as we move away from the seed points, the probability of encountering a sub-volume with only one or two active fibers becomes very large, especially when the range of initial directions per seed points is very large. In this case, an entire multiprocessor would be dedicated to tracking a single fiber through an otherwise empty sub-volume, which is obviously a huge waste of resources.

For these reasons, we have chosen not to implement the tile-based approach. Instead, we let each fiber load the data it needs at its current position, without trying to guarantee any form of spatial coherence between the fibers. The obvious downside of this approach is that the amount of memory that needs to be transferred between the device memory and the multiprocessors is larger than for the tile-based approach, as no steps are taken to avoid redundant reads. However, the big advantage is that we will be able to maintain a higher degree of occupancy, since almost all threads in a thread block (except for those that have terminated due to their fiber leaving the volume) will be active at all times.

4.5.2 Kernel Structure

We now describe the structure of the implemented kernel using pseudo-code, using the control flow presented in Figure 3.2 as a basis. Let $D([u, v, w])$ be the (pre-processed) tensor at the coordinates $\mathbf{x} = (u, v, w)$, and let $dGdu(\mathbf{x})$, $dGdv(\mathbf{x})$ and $dGdw(\mathbf{x})$ be the three derivative tensors of its inverse at that position. Let idx be the global index of a thread, which is computed using the index and size of the thread block it is in and its local index within this box. For example, if we have five blocks of four threads each, the second thread in the third block will have $idx = 2 * 4 + 1 = 9$, as both the block index and the local thread index start at zero. Finally, let $Seed(idx)$ be the seed point data of the fiber, consisting of both a position and an initial direction and $O(idx, n)$ the n^{th} output coordinate set. The pseudo-code for this kernel is shown in Figure 3.

Algorithm 3

(* Track fibers in DTI data. *)

1. $S \leftarrow Seed(idx)$
2. $currentPosition \leftarrow S.position$
3. $currentDirection \leftarrow S.direction$
4. $n \leftarrow 0$
5. **while** ($n < MAXIMUM_NUMBER_OF_STEPS$)
6. $W \leftarrow D(currentPosition)$
7. $X \leftarrow dGdu(currentPosition)$
8. $Y \leftarrow dGdv(currentPosition)$
9. $Z \leftarrow dGdw(currentPosition)$
10. $C \leftarrow ComputeChristoffelSymbols(W, X, Y, Z)$
11. $[nextPosition, nextDirection] \leftarrow SolveODE(C,$
12. $currentPosition, currentDirection)$
13. $currentPosition \leftarrow nextPosition$
14. $currentDirection \leftarrow nextDirection$
15. $O(idx, n) \leftarrow currentPosition$
16. $n \leftarrow n + 1$

This code does not include the statement that checks if the fiber has left the volume. When this happens, the thread will break out of the `while`-loop, after which it will terminate. This will lead to the situation where some of the threads in a thread block will terminate before others, leading to a reduction in the overall occupancy, but this can never be completely avoided. One solution would be to make the maximum number of steps relatively low (i.e., no more than a hundred steps), which reduces the probability of having underused thread blocks for large amounts of steps. We could, for example, write the current position and direction back to the `Seed` array when the maximum number of steps is reached. After one kernel call, we could then remove from the seed point list those fibers that terminated due to leaving the volume, after which we could continue tracking the remaining fibers where we left off. However, doing so would increase the amount of overhead for data transfers and kernel launches, making the actual benefit of this strategy questionable at best.

4.5.3 Optional Stopping Criteria and Connectivity Measure

As noted in Section 3.3, our algorithm has three optional stopping criteria, in addition to the essential criterion that the fiber must remain within the volume, and the criterion that all

fibers must terminate after a set number of steps. Depending on which of these additional criteria are enabled, the fiber will also stop when the local anisotropy measure drops below a certain value, when the fiber exceeds a certain length, and/or when the angle between two subsequent fiber segments exceeds a certain maximum.

Intuitively, limiting both the number of steps and the maximum fiber length may seem redundant. However, there are two important differences between these stopping criteria. The first is the fact that the number of steps does not directly correspond to the fiber length, since the length of a single step is not known beforehand. The second difference is in their purpose: limiting the number of steps is necessary since CUDA does not allow the threads to dynamically allocate data, while limiting the maximum length is done to prevent infinite loops in the (rare, but possible) case that a fiber trajectory is circular.

These additional stopping criteria can be implemented in a number of ways. The most intuitive solution is to directly include them in the tracking kernel, i.e., break from the `while`-loop when any of the stopping criteria holds. The downside of this approach is that implementing them in CUDA increases the resource requirements of our kernel. The computation of the Fractional Anisotropy, in particular, is quite expensive, and would increase both the amount of memory per thread (in registers and/or local memory), and the number of instructions. Additionally, allowing more threads to terminate prematurely decreases the overall occupancy, reducing the effectiveness of our algorithm.

The second approach would be to include these stopping criteria as a post-processing step, i.e. walk through the computed fibers and break when one of the additional criteria holds. This approach would result in lightweight, highly parallel kernels that would likely be able to process all fibers very quickly (expected running times are in the order of milliseconds), and intuitively appears to be favorable over the first option. However, since the inclusion of these stopping criteria is completely optional and in no way part of the core functionality of our algorithm, we have not implemented this step in CUDA, opting instead for a post-processing step in C++.

Similarly, the Connectivity Measure, which quantifies the strength of computed fibers (see Section 3.4.1), can be either computed within the tracking kernel, or by means of a post-processing step; the same design consideration apply here as for the additional stopping criteria. Again, we have only implemented the post-processing step in C++, with the footnote that it would benefit greatly from a CUDA implementation. A kernel performing the connectivity measure computation would load the data of a computed fiber and cycle through its point, fetching and inverting the DTI tensor at each point, and computing and storing the *CM* value as defined by Equation 3.16.

4.6 Introducing Textures

This section introduces the concepts of using textures in our CUDA implementation. In early versions of the implementation, the input images were stored in global memory. This approach had several downsides, which will be described in detail below. The use of textures in our algorithm improves its performance by increasing the efficiency of memory transfers and by outsourcing the interpolation process to dedicated hardware, allowing for smaller kernels, in terms of both memory requirements and number of instructions. Subsequent sections describe the advantages of using textures, as well as a number of relevant design considerations related to our implementation.

4.6.1 Non-Coalesced Memory Accesses

The CUDA programming guide states that the creation of coalesced memory accesses is a very important optimization strategy [38]. A memory is coalesced when all threads within a half-warp (i.e., 16 threads within the same thread block) all try to read from the same block of 128 bytes in the global memory. For example, assume that the input of a kernel is an array \mathbb{I} of floating-point values, and that the first 16 threads of this kernel have a unique thread index idx between 0 and 15. Fetching a floating-point value A using $A = \mathbb{I}[\text{idx}]$ now creates a coalesced memory read, meaning that all sixteen memory fetches of threads 0 to 15 are grouped into a single memory read, thus increasing the efficiency of the read operation.

However, as discussed in Section 4.5.1, we have opted not to group together threads based on the locations of their fibers. In other words, there's no way to guarantee that all sixteen threads - or any two threads, for that matter - in a half-warp will try to read from the same 128 bytes in global memory. This makes coalescing of memory reads practically impossible for our application. However, the CUDA Best Practices guide states that the use of textures may allow for a more efficient utilization of memory throughput of non-coalesced memory access patterns [39].

4.6.2 Texture Cache

One potential advantage of using texture memory reads over global memory reads stems from the fact that texture reads are cached. The NVIDIA GPU architecture contains a number of small caches (around 8kB per multiprocessor, although this may differ per device [15]), which are used to cache the data read from texture memory. This can be especially advantageous when a large number of active threads are reading from the same region of memory, as it reduces the required throughput between the multiprocessors and the device memory.

We will now illustrate the effect of texture caching on the running time of our geodesic fiber-tracking algorithm. Specifically, we will show that the spatial locality of the fiber heads will greatly influence the running time. For the purpose of this experiment, we first create a synthetic data set of size $64 \times 256 \times 64$ voxels. The original DTI tensor is the same for all voxels:

$$D = \begin{bmatrix} 0.001 & 0 & 0 \\ 0 & 0.003 & 0 \\ 0 & 0 & 0.001 \end{bmatrix}$$

This corresponds to a uniform volume of tensors with main eigenvectors in the direction parallel to the longest side of the volume. In addition, all seed points within this volume have the initial direction $\dot{\mathbf{X}}_0 = [0 \ 1 \ 0]^T$. This way, all computed fibers will be parallel to the longest side of the volume for their entire length. Finally, the locations of the seed points are chosen such that all fibers will run for the defined maximal number of integration steps (i.e., no fibers will terminate due to reaching the border of the volume). In all cases described below, the total number of seed points was 4096, divided over 64 thread blocks of 64 threads each. The volume used during this benchmark is equal to the synthetic data set described at the start of Chapter 5 and illustrated in Figure 5.1.

The running time of the tracking algorithm was measured for five different seed point patterns, denoted A to E, see Table 4.1. In this table, the function $\text{random}(x, y)$ returns a random real value from the range $[x, y]$. The variables ti and bi are the thread index and the block index, respectively. The thread index is the index of a thread within its thread block ($ti = 0, 1, \dots, 63$), and bi is the index of the thread block within the grid ($bi = 0, 1, \dots, 63$).

	u_0	v_0	w_0
A	32	1	32
B	ti	1	32
C	ti	1	bi
D	random(0, 63)	random(0, 1)	random(0, 63)
E	random(0, 63)	random(0, 127)	random(0, 63)

Table 4.1: Seed point coordinates for the experiments described in section 4.6.2. Here, ti and bi are the index of the thread and its thread block, respectively, and **A** to **E** are the seed point patterns for which we measure the running time.

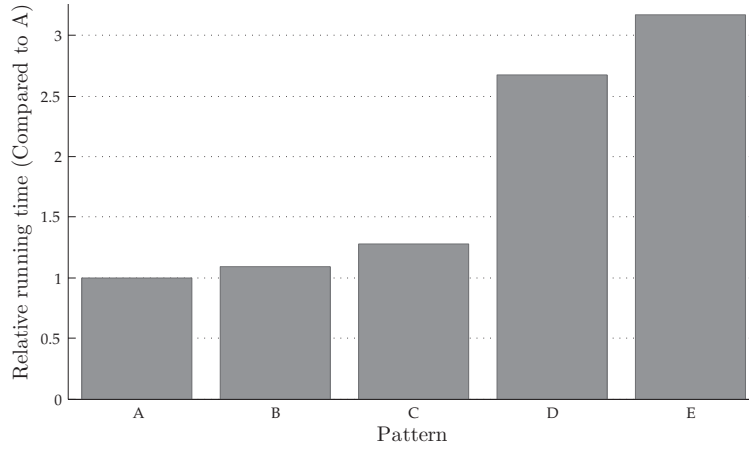


Figure 4.9: Relative running times for the five different seed point patterns (**A**, **B**, ..., **E**) introduced in Table 4.1. All running times are relative to the running time for pattern **A**.

The relative running times, compared to that of pattern **A**, are shown in Figure 4.9. We can see that a reduction in the spatial locality of the seed points will lead to increased running times. The difference in terms of running times between patterns **B** and **C** shows that cached texture variables can be shared between threads of different thread blocks. Indeed, closer inspection of the NVIDIA GTX 200 GPU architecture shows that texture caches are shared between a number of multiprocessors, which explains these findings; see also Volkov *et al.* [61].

Since we cannot predict the trajectory of the geodesic fibers, grouping them together in blocks to ensure spatial locality is practically impossible. When shooting a large number of fibers from a single seed point in different directions, the effects of texture caching will be most noticeable when the fibers are still close to the seed point. At this time, all fiber heads will still be grouped within a small region, creating a pattern similar to patterns **A** and **B**. However, after a large number of integration steps, this spatial locality will have disappeared almost completely, and the locations of the fiber heads will begin to resemble patterns **D** and **E**. We therefore conclude that, while texture caching would improve performances in the early stages of the fiber-tracking process, its effectiveness would be reduced for long fibers.

It should be noted at this point that the differences in running time shown in Figure 4.9 may be caused in part by factors other than texture caching. It is known that DDR memory,

such as the device memory in GPU, experiences less overhead for bank switches when the memory access patterns exhibit high spatial locality, which is the case for **A** and **B**. By contrast, patterns **D** and **E** require more bank switches, leading to a larger overhead in access times. Furthermore, NVIDIA claims that textures are optimized for 2D spatial locality [38]. While the exact method by which this optimization process takes place does not appear to be mentioned in any official sources, we expect that some sort of *space-filling curves* may be used to achieve this effect. Therefore, both the inherent benefit of high spatial locality when using DDR memory and NVIDIA's space-filling curves may contribute to the results of the above experiment.

4.6.3 Texture-Filtering Interpolation

A third important advantage of using textures for fetching the required data is the fact that CUDA supports *texture-filtering*. Since texture fetches in CUDA can be addressed using floating-point coordinates, the requested data may be located in between two (for 1D textures), four (2D), or eight (3D) texels. In such a scenario, the texture-filtering mode determines the way in which the required data is fetched. When using the default mode, `cudaFilterModePoint`, the returned value is that of the texel closest to the specified texture coordinates, which effectively implements nearest-neighbor interpolation. The second mode, `cudaFilterModeLinear`, automatically applies low-precision trilinear interpolation on the eight surrounding texels (for 3D textures), and returns the result of this interpolation process.

The entire interpolation process, including the computation of the required texel coordinates, fetching the data of these texels, and finally computing the output value, is performed on dedicated hardware blocks. Each multiprocessor or block of multiprocessor contains a number of these texture processing units, which can be used by all threads within the multiprocessor(s). This configuration offers a number of advantages in the context of our implementation:

- Calculation of the coordinates of the surrounding voxels no longer needs to be performed within the kernel. Not only does this reduce the number of required computations per thread, it also allows for a more straightforward and intuitive way of fetching the data.
- Computation of the weights, multiplication of the voxel data with these weights and summation of the multiplication results occurs on dedicated hardware, rather than on the processors. Trilinear interpolation of the four symmetric tensors required for the geodesic tracking integration process requires at least $8 * 24 = 192$ floating-point multiplications and $7 * 24 = 168$ additions, which is excluding the computation of the interpolation weights. By performing these instructions on the dedicated texture hardware, we save at least 360 floating-point operations per thread per integration step. This is a huge improvement, since the algorithm now performs only 256 floating-point operations per step (as will be shown in Section 5.3.4).
- The values of the eight surrounding voxels no longer need to be stored in the kernels. This is an important advantage for our implementation, since the relatively large size of its main tracking kernel means that most of these values would otherwise be stored in local memory. Since local memory resides within the device memory, its relatively high access latencies can be a big bottleneck for kernels that store a large number of

$$\begin{aligned}
\mathbf{V}(i, 0) &= [D_{1,1}(i) & D_{1,2}(i) & D_{1,3}(i) & D_{2,2}(i)] \\
\mathbf{V}(i, 1) &= [D_{2,3}(i) & D_{3,3}(i) & \frac{\delta G_{1,1}}{\delta u}(i) & \frac{\delta G_{1,2}}{\delta u}(i)] \\
\mathbf{V}(i, 2) &= [\frac{\delta G_{1,3}}{\delta u}(i) & \frac{\delta G_{2,2}}{\delta u}(i) & \frac{\delta G_{2,3}}{\delta u}(i) & \frac{\delta G_{3,3}}{\delta u}(i)] \\
\mathbf{V}(i, 3) &= [\frac{\delta G_{1,1}}{\delta v}(i) & \frac{\delta G_{1,2}}{\delta v}(i) & \frac{\delta G_{1,3}}{\delta v}(i) & \frac{\delta G_{2,2}}{\delta v}(i)] \\
\mathbf{V}(i, 4) &= [\frac{\delta G_{2,3}}{\delta v}(i) & \frac{\delta G_{3,3}}{\delta v}(i) & \frac{\delta G_{1,1}}{\delta w}(i) & \frac{\delta G_{1,2}}{\delta w}(i)] \\
\mathbf{V}(i, 5) &= [\frac{\delta G_{1,3}}{\delta w}(i) & \frac{\delta G_{2,2}}{\delta w}(i) & \frac{\delta G_{2,3}}{\delta w}(i) & \frac{\delta G_{3,3}}{\delta w}(i)]
\end{aligned}$$

Table 4.2: Distribution of the four symmetric tensor field used as input for the Tracking kernel over six textures, each with a four-element vector at each tensor.

variables in local memory. By reducing the memory requirements per thread, dedicated interpolation hardware may reduce the influence of this memory latency.

One potential downside of using texture-filtering interpolation instead of in-kernel interpolation is that it requires a larger memory throughput between the multiprocessors and the device memory. When we store the values of the eight surrounding voxels locally, we can reuse them for several steps of the integration process, since we do not need to load new values from global memory unless the fiber head moves from one cell to the next. The average number of steps needed by a fiber to cross a single cell depends on the integration step size and the voxel spacing, and linearly controls the required memory throughput for the device memory. By contrast, a tracking algorithm that uses texture-filtering interpolation needs to fetch the data of the eight surrounding texels in every single integration step. If, for example, a fiber requires an average of ten integration steps to cross a single cell, the device memory throughput of the in-kernel interpolation scheme will be roughly ten times lower than that of the texture-filtering interpolation scheme (not accounting for the effects of texture caching).

4.6.4 Texture vectors

Each texel in a CUDA texture contains either a scalar value, or a vector of two or four values. In our algorithm, we have four tensors of six elements each per texel, and all 24 values need to be interpolated. Therefore, we can group these values into six textures, each with a four-element vector per texel. Generally, reading N four-element vectors from texture memory is generally faster than reading $2N$ two-element vectors or $4N$ scalar values [15]. Based on these findings, we have decided to group together elements from different tensors in a single vector, thereby ensuring that we only use four-element vectors.

Let $\mathbf{V}(i, T)$ be the four-element vector at location i in texture T , let $D(i)$ be the pre-processed DTI tensor at location i , and let $\frac{\delta G}{\delta u}(i)$, $\frac{\delta G}{\delta v}(i)$, $\frac{\delta G}{\delta w}(i)$ be the three derivatives of its inverse at location i . The contents of the six textures used in our program are shown in Table 4.2.

4.6.5 Boundary conditions

CUDA textures are able to automatically correct coordinates that exceed the limits of the texture volume. These coordinates can either *wrapped* to the other side of the volume, or *clamped* to the nearest boundary. For example, in a volume of 10 by 10 by 10 voxels, the coordinate $(-1, 5, 5)$ would either be wrapped around to $(9, 5, 5)$, or clamped to $(0, 5, 5)$.

Since the DTI images used as the input for our algorithm do not wrap, we use the clamping option.

4.6.6 Addressing

One final option of CUDA textures is the addressing mode. The texels can be addressed using either absolute coordinates or relative coordinates. Absolute coordinates are in the range $[0, N_x)$, where N_x is the number of texels in the relevant dimension. Relative coordinates, on the other hand, have the range $[0, 1)$, with zero corresponding to the lower boundary of the volume, and one corresponding to its upper boundary. For the purpose of our algorithm, absolute coordinates are the best option, as they allow for a straightforward and intuitive addressing scheme.

4.6.7 Writing to textures

In CUDA, textures are created by binding a *texture reference* to a range of device memory. The way in which this memory space is allocated influences the functionality of the texture. For example, when it is allocated as linear memory using the simple `cudaMalloc()` function, both the interpolation functionality and the boundary conditions are disabled, and texture only serves as a cache. Using pitched linear 3D memory (which aligns the start of each row to ensure optimal row addressing, as described in the CUDA Programming Guide [38]) enables these features, but has the disadvantage that pitched linear memory does not currently support 3D textures. A solution would be to use 2D textures bound to 2D pitched memory, but this does not allow for trilinear interpolation. This means that (part of) the interpolation steps would need to be performed within the kernel, which would reduce the efficiency of using textures.

For these reasons, we are limited to the third option, which is to use CUDA arrays. The advantages of CUDA arrays are that they support the additional textures features, such as interpolation, and that they allow for 3D textures. The main downside of these arrays, however, is the fact that they cannot be written to by the kernels. Data can only be written to CUDA arrays using host-controlled memory copies from either the host memory or another location in the device memory. This is not an issue for the actual tracking algorithm, as it does not need to write to the textures, but the kernels that prepare the data needed by the tracking algorithm (the Pre-Processing kernel and the three invocations of the Derivatives kernel) do need to store their output data in textures. We solve this by having these kernels write to another location in the device memory, and subsequently copying this data to the CUDA array to which the textures are bound. Since the intra-device memory bandwidth is very high, these extra steps add only a couple of milliseconds to the total pre-processing time. The latency of the intra-device copy action is benchmarked in detail in Section 5.2.4.

The process of storing a kernel's output in an output array before copying it to a texture is illustrated in Figure 4.10. The four relevant data sets are shown in global memory, as well as the six textures that are used to store them. Note that, since the output data set of each kernel has been distributed over two actual textures, the Output Array must be big enough to contain two textures.

This figure also motivates our decision to split the data-preparation stage into two kernels with a total of four kernel invocations, rather than a single big kernel that is executed only once. Let N be the total number of voxels in an image. Using the scheme shown in Figure 4.10, the amount of device memory needed to store the textures and the output array is

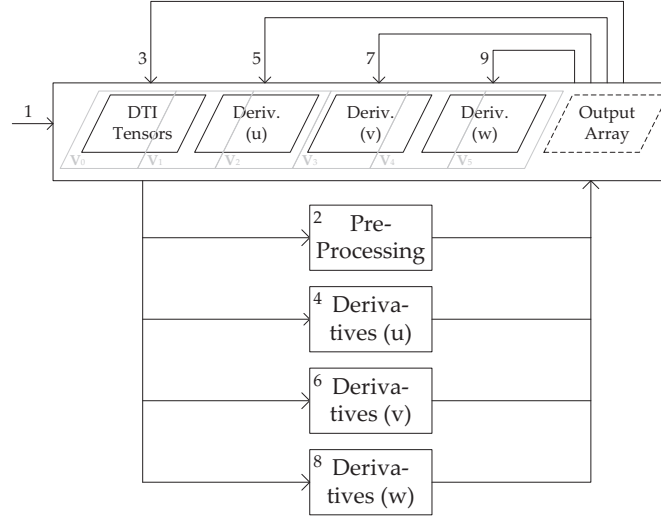


Figure 4.10: Illustration of the memory management scheme for the data-preparation stage. First (1), the CPU copies the input data to the *DTI Tensors* textures. The Pre-Processing kernel loads this data, and stores its results in the output array (2), which is then copied back to the *DTI Tensors* textures. Subsequently, the Derivatives kernel is executed three times (4, 6, 8), once for each direction. The kernel stores its results in the output array, after which the data is copied to the corresponding textures (5, 7, 9). Once this is done, we may free the output array, and launch the tracking kernel. The gray lines indicate the six textures between which the four tensors have been distributed, see Section 4.6.4.

$5\frac{1}{3}N$: four data sets of N each, divided over six textures, and an output array big enough to contain two textures of $\frac{4}{6}N$ each. If we were to use a single, big kernel, we would need to have an output array of size $4N$, giving a total memory usage of $8N$, which would decrease the maximum image size by a third.

4.6.8 Conclusion

CUDA textures offer a number of advantages over global memory arrays. Of particular interest for our project are the texture-filtering interpolation, which allows us to greatly reduce the size of the threads, and the significant speed-up gained by having a high spatial locality between fibers. In addition, the use of textures helps to reduce the penalty imposed by using non-coalesced memory reads, which are unavoidable in our algorithm. One of the downsides of using textures is that their read-only nature forces us to perform an intra-device memory copy, but as Section 5.2.4 shows, this adds a delay of only a couple of milliseconds to the pre-processing phase. We have also demonstrated the advantages of using texture-filtering interpolation instead of in-kernel interpolation.

In general, we conclude that the use of texture is very advantageous for our algorithm, in the sense that implementations using textures are significantly faster than those using global memory. These claims will be supported by the benchmarks presented in Chapter 5.

4.7 Summary

In this chapter, we discussed a CUDA implementation of the geodesic fiber-tracking algorithm introduced in Chapter 3. The highly parallel GPU architecture allows us to accelerate both the data-preparation stage and the tracking stage of the algorithm. We have shown that parallelization of the computations in the data-preparation stage is straightforward, as it intuitively allows for voxel-level parallelism. The tracking stage has been parallelized by having each thread compute a single fiber. Doing so allows us to maintain a high level of occupancy; the main downside is that we cannot guarantee any spatial locality between the different threads in a thread block, which lowers the efficiency of memory transfers. Finally, we introduced textures, which offer two distinct advantages for our algorithm: it enables texture-filtering interpolation on dedicated hardware, allowing for a smaller kernel in terms of both memory requirements and computational complexity, and it caches memory reads, which may increase performance when a lot of fibers are located in a relatively small region of the image.

Chapter 5

Results

In this chapter, we discuss the performance of the CUDA implementation introduced in Chapter 4. Our focus in this case is the *running time* of the algorithm, the reduction of which was the main motivation for our research. We first demonstrate the correctness of the algorithm itself, as well as the results of the implemented fiber filtering technique. Next, we discuss the running time of the data-preparation stage, followed by a description of the results of various benchmarks performed on the tracking kernel. In addition to showing the running time as a function of a number of factors, we aim to quantify the result of using texture memory instead of global memory for storing the input data (as discussed in Section 4.6), and to demonstrate the advantage of using texture-filtering interpolation instead of in-kernel interpolation. Finally, we compare the performance of our CUDA implementation to that of a sequential C++ implementation, to show that the GPU allows for significant acceleration of the algorithm without loss of accuracy.

The benchmarking platform was the NVIDIA GTX 260 GPU, a mid-range model with 24 multiprocessors (i.e. 192 scalar processors), 1 GigaByte of DDR device memory, and a processor clock speed of 1242 MHz. Additional specifications of this GPU are listed in Table 5. The unofficial CUDA decompiler `decuda` [28] was used in some cases to analyze the machine-code generated by the CUDA compiler, allowing us to quantify the computational throughput of the program. Finally, we used the CUDA Occupancy Calculator, which is described in detail in Section 4.1.3.

Note that the memory size of 1 GB limits the maximum image size to roughly 190 MB (since we need to store $5\frac{1}{3}$ full images, see Section 4.6.7). With six floating-point values per voxel, an image may contain at most roughly 8 million voxels, which corresponds to a cube of 200 voxels on each side. Larger images are not supported; if an input image exceeds 190 MB, a relevant sub-volume of 190 MB or less should be selected before execution of the CUDA program.

The data set used in all benchmarks described below is a volume of $64 \times 256 \times 64$ voxels. The tensors in this volume are all equal and anisotropic, meaning that the direction of the fibers does not change from its initial direction. The initial direction of all fibers is $[010]^T$, i.e. one in the direction with 256 voxels, and zero in the other two directions. The real coordinates of the seed points are randomly distributed over the volume $([0, 63], [0, 1], [0, 63])$. The random distribution ensures low spatial locality between the different fibers, which is also the case for fiber-tracking in real data, as discussed in Section 4.6.2. The step size is chosen such that each fiber terminates only when a maximum number of integration steps have been performed (i.e. no fibers terminate due to leaving the volume); this maximum number of steps is set to 2048. The benchmarking data set is illustrated in Figure 5.1.

Scalar Processors	192
Graphics Clock	576 MHz
Processor Clock	1242 MHz
Memory Clock	999 MHz
Memory Interface Width	448 bit
Memory Bandwidth	111.9 GB/s

Table 5.1: Specifications of the NVIDIA GTX 260 GPU [40].

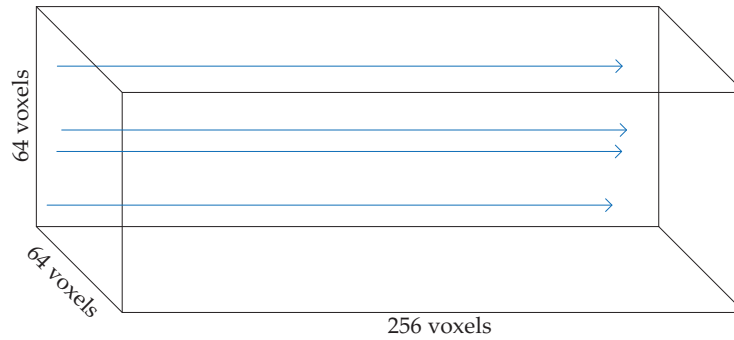


Figure 5.1: Synthetic data set used for benchmarking. Some examples of resulting fibers are shown in blue. All fibers run in the same direction for the same amount of steps; their seed points are randomly distributed over a small volume on the left-hand side of the image.

Note that the described data set is not supposed to represent a realistic scenario. While the random distribution of the seed points does aim to mimic the low spatial locality between fibers in real data, the fact that all fibers run for the maximum number of steps does not reflect a realistic scenario. It does, however, allows us to benchmark the performance of the algorithm under constant maximum occupancy. We argue that in a more realistic scenario, where some fibers will terminate prematurely due to reaching the boundaries of the volume, the average memory throughput will be slightly lower than reported in this section, which may lead to a small decrease in running times.

5.1 Algorithm

In this section, we aim to demonstrate the correctness of the algorithm, by showing that the fibers it computes do indeed correspond to actual fiber bundles. Additionally, we show the effects of applying the Region-to-Region Connectivity method, as introduced in Section 3.4. The Region-to-Region method has been implemented as a sequential post-processing step, which filters out those fibers that do not intersect with the target Region of Interest (ROI), after which it computes the Connectivity Measure value (see Section 3.4.1) for the remaining fibers, and removes those fibers with low connectivity measure values. The images shown in this section have been made using DTITool, a research tool developed at the Eindhoven

University of Technology, which served as the main implementation environment for our CUDA program.

Figure 5.2 shows a synthetic data set, which demonstrates that the algorithm is capable of constructing fibers that correctly follow structural features, such as a U-shaped fiber. To demonstrate the correctness of the algorithm for real data, Figure 5.3 shows a group of computed fibers in a section of the Corpus Callosum. In both cases, the Region-to-Region Connectivity filter was used to narrow down the set of fibers computed by the CUDA algorithm. First, those fibers that do not intersect the target regions are removed, resulting in the fibers shown in the left-hand side of both images. Subsequently, The connectivity measure is used to distinguish between strong fibers and weak fibers, resulting in the set of fibers shown in the right-hand side of the images.

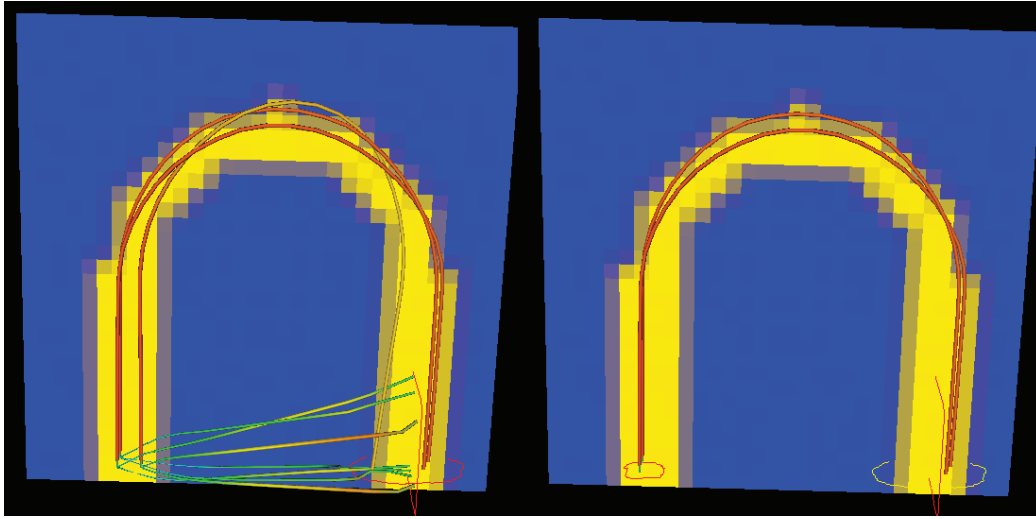


Figure 5.2: Computed fibers in a synthetic U-Fiber data set. The blue region is completely isotropic, while in the yellow region, the main eigenvector is oriented in the direction of the U-shape. When we shoot in all directions from a seeding region located in one of the stems of the U, some fiber will directly cross the isotropic void between the two stems (left). However, these fiber will have a lower connectivity measure value than those that do follow the U-shape, and a subsequent Region-to-Region filtering step will remove them (right). In this picture, the fibers have been colored according to their local connectivity measure value (with red signifying a high CM value, and green representing low CM values).

From the results shown in Figures 5.2 and 5.3, we can conclude that the algorithm can correctly deduce the fibrous structures from the DTI data, for both noiseless synthetic data and noisy real data. Additionally, these images demonstrate the purpose of both the multi-valued solutions produced by the algorithm, and the fiber filtering technique used to find the strongest connections. In both cases, multiple connections between the target regions were constructed, after which the Region-to-Region Connectivity filter narrowed them down to only the strongest connections (i.e., those most likely to correspond to a physical fiber connection).

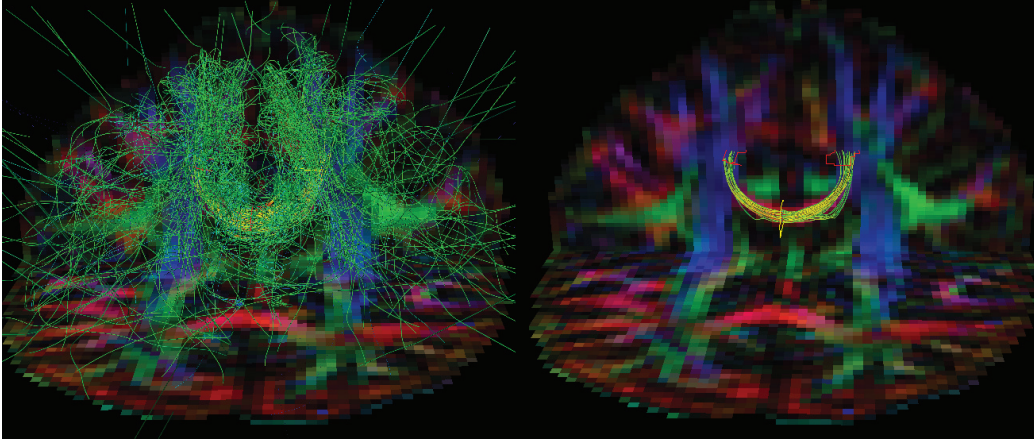


Figure 5.3: Computed fibers showing a section of the Corpus Callosum. The fibers are traced from a single seeding region (yellow outline) to two filtering regions (red outlines, visible on the right). Since we track fibers in all directions, and we do not stop tracking when regions of low anisotropy are encountered, the initial set of fibers computed by the CUDA algorithm is very large (left). After using the Region-to-Region Connectivity filter to remove weak connections, we are left with a structure that strongly resembles the Corpus Callosum (right).

5.2 Data-Preparation Stage

The *data-preparation stage* of our CUDA implementation consists of two kernels, the *Pre-Processing kernel* and the *Derivatives kernel*. In this section, we aim to find a formula for the running time of the data-preparation stage. This running time includes the running time of the Pre-Processing kernel, three times the running time of the Derivatives kernel (since we run it three times, once for each direction, see Section 4.4), as well as a certain overhead caused by memory transfers. We first present benchmarks for these individual elements, after which we combine their results to find a formula for the total running time. We will show that this running time depends linearly on the size of the input image.

It should be noted at this point that during the research project, the main focus of our optimization efforts was on the Tracking kernel, meaning that the data-preparation kernels benchmarked below are not necessarily optimal. We do argue, however, that the kernels still experience significant speed-up factors when compared to a sequential CPU implementation.

5.2.1 Pre-Processing Kernel

We first obtain the running time of the pre-processing algorithm. This kernel loads the six elements of the input tensors from two different textures (see Section 4.6.4), after which it writes the pre-processed tensors to the output array. The results of this particular benchmark were obtained with the tensor sharpening option disabled, but additional benchmarking revealed that enabling this option does not significantly impact the running time: for tensor exponents between 2 and 10 - which is already a far greater range than what should be used for practical purposes - we did not register a noticeable increase in running time. This

suggests that the performance of this kernel is limited by the memory throughput rather than the computational throughput, even though - as the results will show - the current memory throughput does not correspond to a known hardware limitation.

In order to obtain the results described below, we used a block size of 128 threads. This number was reported by the CUDA Occupancy Calculator [36] as one of the possible optimal configuration given the other contributing factors: each thread uses 24 registers, and a block of thread only requires 72 bytes of shared memory. No local memory is used. The Occupancy Calculator reports that for 128 threads per block, each multiprocessor can run five thread blocks in parallel, with the number of registers as the main limiting factor. Experimentation confirms that 128 threads per block is indeed optimal in this case.

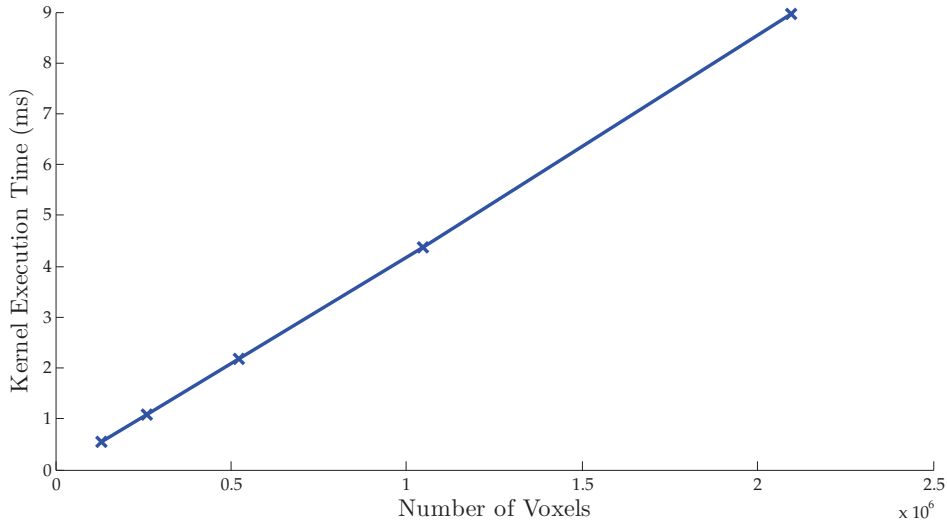


Figure 5.4: Running time of the Pre-Processing kernel as a function of the number of voxels.

The results of this benchmark are shown in Figure 5.4. The running time depends linearly on the number of voxels in the image, i.e.,

$$T_{pp}(N) = T_e * N + T_0. \quad (5.1)$$

Here, $T_{pp}(N)$ is the running time as a function of the number of voxels N , T_e the execution time per voxel (and, since each thread processes a single voxel, also the time per thread), and T_0 the launch overhead of the kernel. Since the launch overhead is in the order of $5 \mu s$ [61], we have $T_0 = 5 * 10^{-6}$ seconds, and $T_e \approx 4.274 * 10^{-9}$ seconds per voxel, by a linear fit of the data. Additional benchmarking has shown that this linear behavior continues at least up to the maximum image size of approximately 8.3 million voxels.

From these results, we conclude that the performance of the Pre-Processing kernel is currently not optimal. The total memory throughput (reading and writing) to the device memory is roughly 15 GB/s ($(256 \times 64 \times 64)$ voxels per texture $\times 2$ textures $\times 4$ vector elements per voxel $\times 4$ bytes per element $\times 2$ (reading and writing) $\times \frac{1}{T_e}$), which is less than the maximum bandwidth of the GTX 260. Furthermore, `decuda` revealed 232 floating-point operations per tensor, giving us a total computational throughput of 54 GFLOPS, which is again lower than the theoretical maximum. However, this computational throughput is still

well above the maximum computational throughput of a single core of a modern CPU; while the efficiency of the kernel may not be optimal, this result at least validates the parallelization of the pre-processing step.

5.2.2 Derivatives Kernel

Next, we measure the running time of the Derivatives kernel. Our implementation of this kernel uses 28 registers per threads, and 100 bytes of shared memory per thread block. Based on these two specifications, the Occupancy Calculator reports that a block size of 128 threads is optimal. We therefore use the same thread configuration as for the Pre-Processing kernel. Recall that we launch the same kernel three times in a row, changing only the direction of derivation each time, see Section 4.4. Since each invocation of the kernel processes all voxels, its running time does not depend on the current direction, nor does it feature user-defined parameters that may influence its performance.

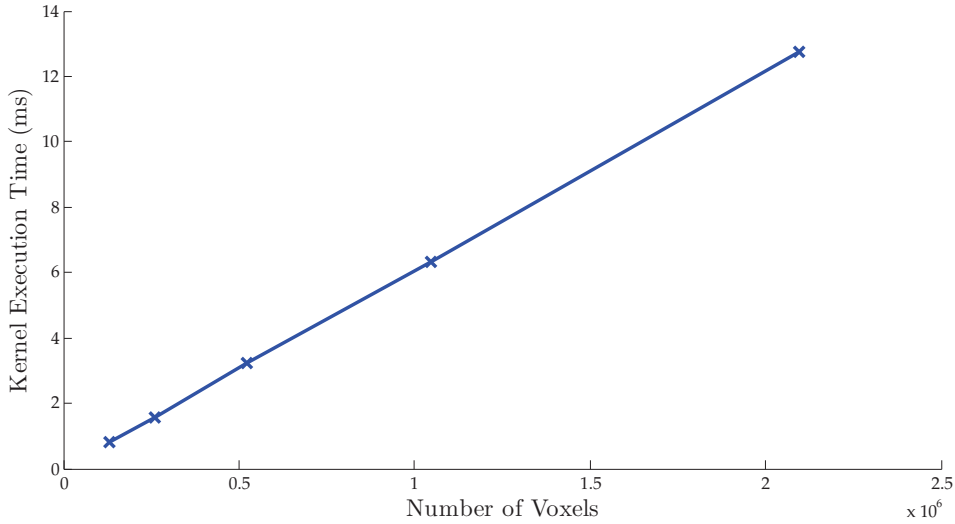


Figure 5.5: Running time of the Derivatives kernel as a function of the number of voxels.

The results of this benchmark are shown in Figure 5.5. We express the running time $T_d(N)$ as a linear equation of the same form as Equation 5.1, again using $T_0 = 5 \times 10^{-6}$ seconds as the kernel launch overhead, and with $T_e \approx 6.082 \times 10^{-9}$ seconds per voxel. Like with the Pre-Processing kernel, we conclude that the derivatives kernel is less than optimal. Its effective memory throughput is roughly 17 GB/s, while its computational throughput is approximately 24.3 GFLOPS (based on 157 floating-point operations per tensor). However, its computational throughput is still larger than that of the fastest sequential algorithm on a CPU by a wide margin.

5.2.3 Memory Setup

Before we can execute the Pre-Processing kernel and the three Derivatives kernels, we first need to upload the input data to the GPU. Specifically, we identify three different steps required to set up the device memory:

- Declare six textures (see Section 4.6.4), and allocate room for them in the device memory.
- Allocate an output array in the device memory (since the textures are read-only).
- Copy the input tensors to the corresponding texture.

All three steps require communication between the CPU and the GPU. Since this inter-device communication is much slower than intra-device communication for both the CPU and the GPU, the influence of these steps on the total running time cannot be assumed to be irrelevant.

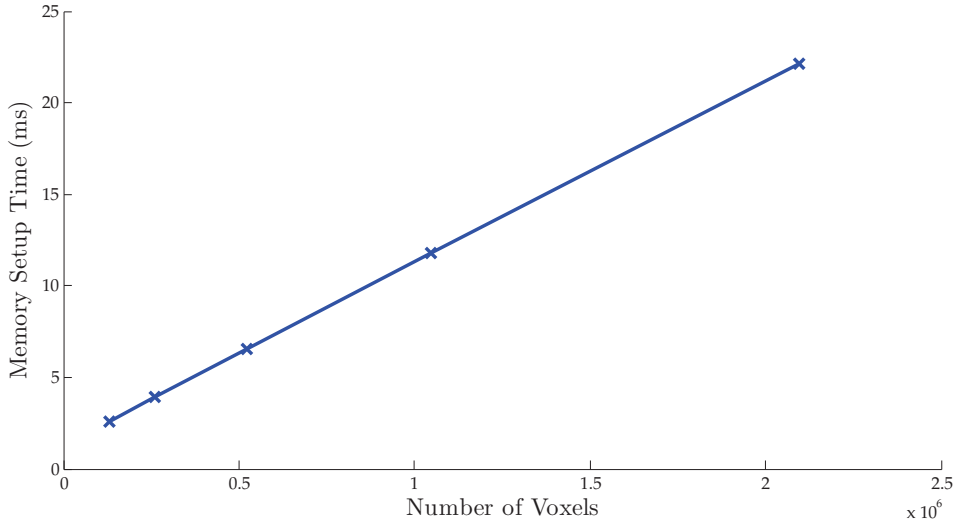


Figure 5.6: Time required to setup the device memory as a function of the number of voxels.

As Figure 5.6 shows, the time required to setup the memory $T_{setup}(N)$ again depends linearly on the number of voxels in the input image. Applying (again) a linear fit, we obtain the following formula:

$$T_{setup}(N) \approx (9.899 * 10^{-9}) * N + 1.359 * 10^{-3}. \quad (5.2)$$

The relatively large offset of over one millisecond could be explained by the overhead for CPU-to-GPU communication.

5.2.4 Intra-Device Copying

As described in section 4.6.7, the main disadvantage of using CUDA texture to store the data needed by the fiber-tracking algorithm is the fact that the textures are read-only. In order to store the output of the Pre-Processing and Derivatives kernels in these textures, we need to copy this output data from a range of linear memory to the CUDA array associated with the required texture. This process needs to be performed for the Pre-Processing kernel and the three instantiations of the Derivatives kernel, meaning that we need to perform a total of four intra-device copies in between kernel calls.

Also note that, because the six tensor elements per voxel are stored in two separate four-element vector textures (as mentioned in Section 4.6.4), copying an image with N voxels requires $2N$ four-element vectors to be copied, for a total of $(2 \text{ vectors} \times 4 \text{ elements} \times 4 \text{ bytes per floating-point value} \times N \text{ voxels}) = 32N$ bytes. Recall from Section 4.6.4 that the total storage needed in the device memory (excluding the storage required by the output fibers) is $5\frac{1}{3}N$ bytes.

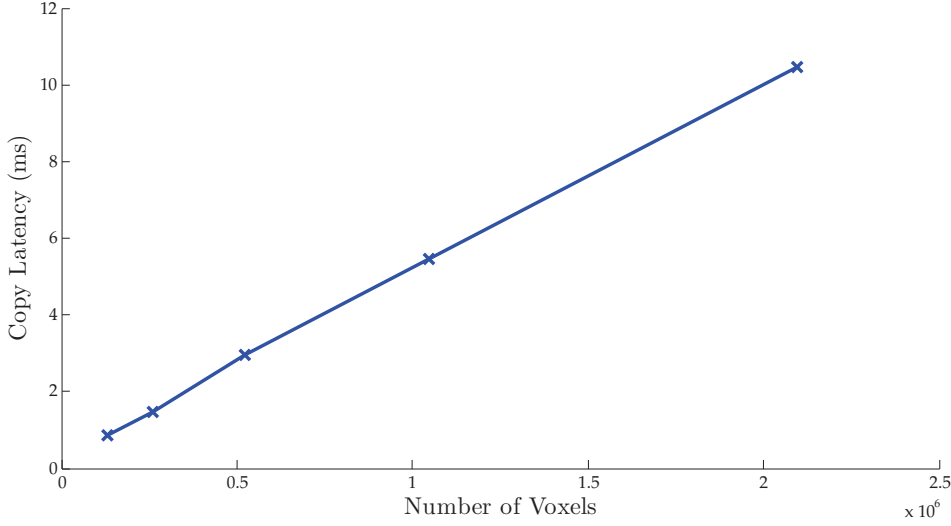


Figure 5.7: Intra-device copy latency as a function of the number of copied voxels. The amount of bytes copied during this memory transfer can be obtained by multiplying the number of voxels by $24 \times 4 = 96$ (number of floating-point values per voxel times the size in bytes of a floating-point value).

Figure 5.7 shows the latency of copying the output results from a range of linear device memory to a CUDA array for several different values of N . As was to be expected, this latency increases linearly with the number of voxels. Let $T_c(N)$ be the time it takes to copy an images of N voxels, with two four-element vectors per voxel, from linear memory to a CUDA array. We can express $T_c(N)$ using the following formula:

$$T_c(N) = T_v * N + T_0 \quad (5.3)$$

Here, T_v is the copying latency per voxel, and T_0 represent the overhead latency created by performing the copying actions. Fitting a linear function to the data in Figure 5.7, we get $T_v \approx 4.879$ ns per voxel and $T_0 \approx 0.267$ ms. Using T_v , we compute that the memory throughput for writing to 3D CUDA arrays is approximately $\frac{1}{4.879 \times 10^{-9}} * 2 * 4 * 4 = 6.558$ GB/s. Since the voxels also need to be read from the device memory, the total device memory throughput during the copying operation is $2 * 6.558 \approx 1.3$ GB/s. The fact that this throughput is far below the theoretical maximum bandwidth for the NVIDIA GTX 260 can be explained by the assumption that CUDA's texture do indeed use space-filling curves, as mentioned at the end of Section 4.6.2. In this case, the construction of the space-filling curve, coupled with the fact that the data must be copied in small blocks, may explain the difference between the theoretical bandwidth and actual memory throughput.

5.2.5 Total Data-Preparation Time

By combining the results of the four benchmarks presented in this section, we can construct a formula for the total running time of the data-preparation stage. Accordingly, the total running time depends *linearly* on the number of voxels in the input image, N , i.e.,

$$T_{total}(N) \approx (51.935 * 10^{-9}) * N + 1.636 * 10^{-3}. \quad (5.4)$$

Benchmarking has shown that this equation is valid at least up to the maximum image size. Equation 5.4 confirms that the influence of the data-preparation stage on the total running time is quite small. Even for an image of $256 \times 256 \times 512$ voxels - which exceeds the total device memory of our GTX 260 - the running time of the data-preparation stage would be less than one second. Since the data-preparation stage does not need to be repeated unless either the input data or one of the pre-processing parameters changes, we regard $T_{total}(N)$ as a start-up interval, after which we can perform the tracking stage as often as is required without additional overhead.

5.3 Tracking Stage

We now discuss the performance of the tracking kernel, which computes the actual fiber trajectories, based on the data computed in the data-preparation stage. We start by discussing the optimal configuration of our final algorithm, in terms of the number of threads per block. Using this optimal configuration, we next demonstrate that using the interpolation functionality provided by texture-filtering interpolation functionality does indeed reduce our total running time, as was first stated in Section 4.6.3. Next, we show that, for our algorithm, using textures is preferable of using global memory. We will also show that the algorithm is limited by the memory throughput (as opposed to the computational throughput), and that our implementation is significantly faster than a sequential C++ implementation.

5.3.1 Configuration

As stated before, the occupancy of a CUDA algorithm is an important measure for its performance. In this section, we aim to achieve the highest possible occupancy, and we use benchmarking to show that this configuration is indeed optimal. Recall that the occupancy, as reported by the Occupancy Calculator, depends on three factors: the number of threads per thread block, the number of registers per thread, and total amount of shared memory used per thread block. The number of registers per thread depends on the algorithm, and is largely beyond our control. In our case, the number of registers per thread is equal to 43.

With regards to the shared memory, we have two options: we can either leave it empty, or we can use it to store some of the local data used by the threads. In the latter case, we would essentially use the shared memory as an extension to the register file. We may, for example, choose to store a small output buffer in shared memory, which could be used to group together writes to the device memory (i.e., the output buffer can contain x coordinate sets, and after every x integration steps, the kernel copies the entire buffer to output memory).

However, benchmarking has shown that storing variables in shared memory is of little practical use. When a thread stores more than a certain amount of data in the shared memory, the size of the shared memory becomes the limiting factor for the occupancy, instead of the number of available registers. For 64 threads per block, this amount of 48 bytes, or only 12

floating-point variables. Storing such a small number of variables in the shared memory has very little impact on the number of registers per thread, and the fact that the shared memory may experience bank conflicts may reduce the computational throughput of the kernel. For these reasons, we only use the shared memory to store the input parameters (like the step size), with the footnote that using the shared memory to store variables may lead to small improvements in running times, at the cost of scalability.

Thus, we have fixed the number of registers to 43, and we only use 100 bytes of the shared memory per thread block. The top graph of Figure 5.8 shows the occupancy as reported by the Occupancy Calculator, expressed as the number of active warps (32 threads) in a multiprocessor. The graph shows four optimal configurations: at 64 and 80 threads per block, and at 304 and 320 threads per block. At more than 320 threads per block, a single thread block no longer fits in the register file, essentially making execution impossible.

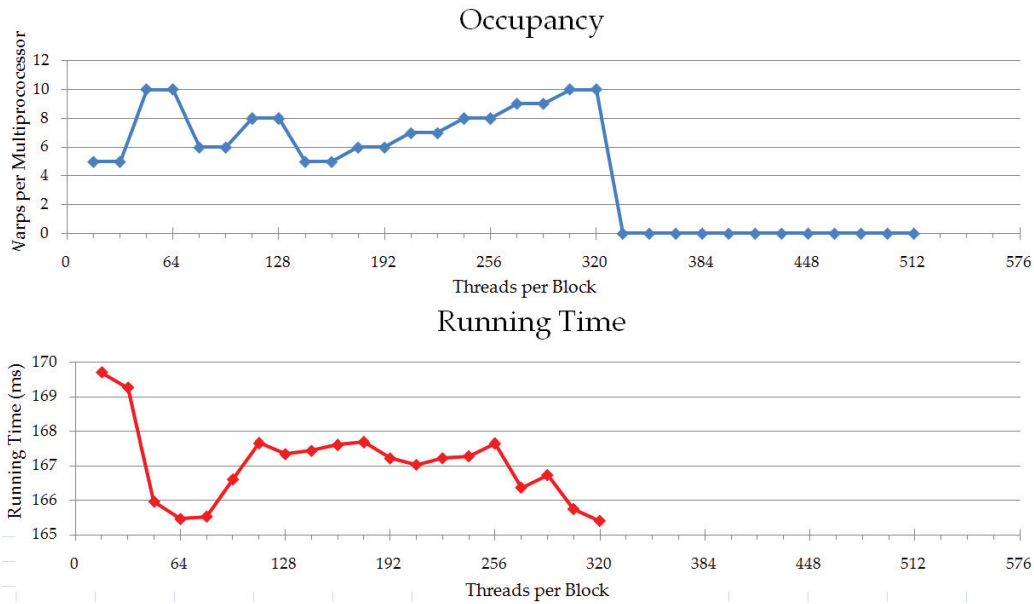


Figure 5.8: Occupancy and running times for different configurations.

In the bottom graph of Figure 5.8, the running time is shown for the different configurations. To measure these running times, we made the total number of threads equal to the number of active threads per microprocessor (i.e., the number of active warps times 32) multiplied by the number of multiprocessor, which is 24 for the GTX 260. In this way, we ensured maximum occupancy while avoiding sequentially executed thread blocks. For example, when the maximum number of active threads is 320, and we have a block size of 64 threads, we launch $(320 * 24) / 64 = 120$ thread blocks.

From these results, we conclude that a block size of 64 is optimal for the current version of the algorithm. In this configuration, we have ten active warps per multiprocessor, for a total of 320 active threads. This means that the number of active thread blocks per multiprocessor is equal to five. The Occupancy Calculator reports that this number is limited by the space available in the register file. In subsequent benchmarks, we will use 64 threads per block as our configuration. Finally, we can conclude from the results that the occupancy has very little effect on the actual running time of the algorithm. In Section 5.3.4, we will show that

this is because the performance of the algorithm is limited by its memory throughput, rather than its computational throughput.

5.3.2 Texture-Filtering versus In-Kernel Interpolation

Recall that we have two possible ways of interpolating the image data at the current fiber position (see Section 4.6.3): we can either store the values in local memory, and perform the interpolation computations inside the kernel (*in-kernel interpolation*), or we can use the built-in *texture-filtering interpolation*. In Section 4.6.3, we stated that the latter would likely work better in our algorithm than the former. In this section, we show this by testing the running times of both implementations.

The main potential advantage of in-kernel interpolation over texture-filtering interpolation is that we can store the values of the eight surrounding voxels, meaning that we only have to load new data when the fiber head moves to a new cell. Doing so will reduce the total memory throughput used by our algorithm, potentially leading to a reduction on running times. The actual effectiveness of this concept, however, depends largely on the step size. If the step size is small, each thread spends a lot of steps inside a single cell, during which time it does not need to load new data. A large step size, on the other hand, results in fibers traversing a cell in only two or three steps, largely negating the potentially advantageous effects of in-kernel interpolation. Therefore, texture-filtering interpolation is not expected to benefit from a smaller step size. This is illustrated in Figure 5.9.

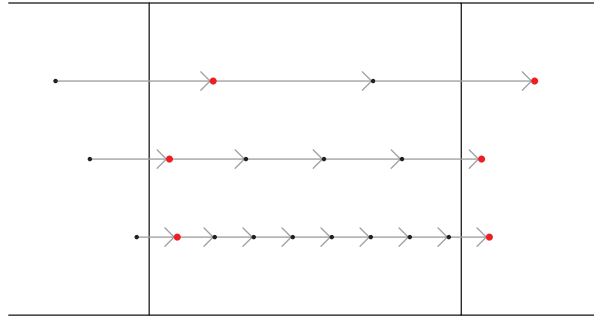


Figure 5.9: Illustration in 2D of the influence of the step size on the number of steps per cell, for $h = 0.5$ (top), $h = 0.25$ (middle) and $h = 0.125$ (bottom). Each arrow represents a single time integration step. When in-kernel interpolation is used, new data only needs to be loaded at the points marked in red; when using texture-filtering interpolation, we need to load data from the device memory in every single step.

For these reasons, we use the step size as the main variable for this benchmark. The synthetic data set described in the introduction of this chapter ensures that the distance travelled in a single step by each fiber is equal to the step size. The distance between voxels is one in all directions, making the number of steps needed to traverse a single cell in our synthetic data set equal to $1/h$, where h is the step size. Both the in-kernel interpolation (IKI) and the texture-filtering interpolation (TFI) versions of the algorithm use 64 threads per block, which was found to be the optimum both through experimentation and by the Occupancy Calculator. The maximum number of steps was set to 512, to ensure that fibers could not reach the

borders of the volume for large step sizes.

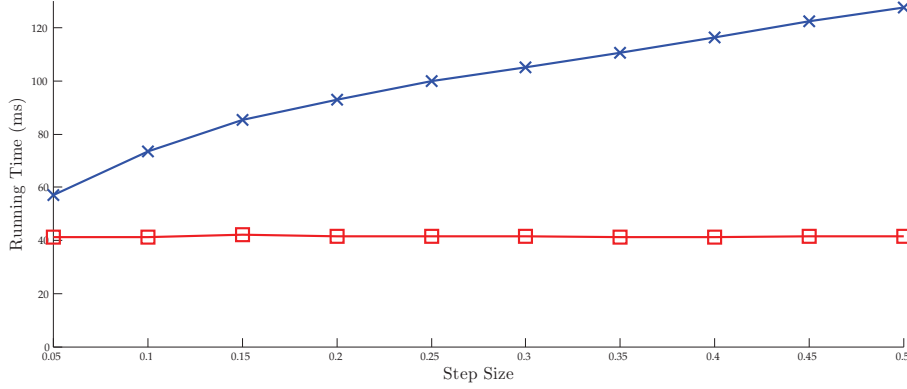


Figure 5.10: Running times of in-kernel interpolation (IKI) and the texture-filtering interpolation (TFI) as a function of the step size.

Figure 5.10 shows the results of this benchmark. As we can see, the texture-filtering interpolation version of our algorithm is indeed independent of the step size. In-kernel interpolation, on the other, is influenced greatly by the step size, as expected. The trends of these running times show that, for very small step sizes, in-kernel interpolation maybe indeed be able to overtake texture-filtering interpolation. However, we should note that a step size of 0.05 is already smaller than what we would normally use (i.e., $h = 0.1$); for step sizes lower than 0.1, the potential increase in accuracy no longer validates the increase in processing times. We must therefore conclude that texture-filtering interpolation is the better choice for our algorithm.

5.3.3 Texture Memory versus Global Memory

In Section 4.6, we introduced the possibility of using textures to store the data needed by the Tracking kernel. In this section, we show that using textures is indeed beneficial to the speed of our program. To this end, we compare the performance of the in-kernel interpolation method described in the previous chapter, in which textures are used to store the data, to a version of the algorithm in which the data is stored in local memory. In this second version, interpolation is also performed inside the kernel, as global memory does not support texture-filtering functionalities. Using the Occupancy Calculator, we identified the optimal configuration for both versions as 64 threads per block, with 64 registers per block. The step size h was fixed to 0.01.

The running times for this experiment are shown in Table 5.2. From this, we can conclude that using textures does indeed reduce the running times of our algorithm by a small

Number of seed points	Running time with textures (ms)	Running time without textures (ms)
1024	74.3	83.5
2048	151.5	167.2
3072	227.9	250.1
4096	302.4	335.6

Table 5.2: Running times of the algorithm using in-kernel interpolation, using textures (left) and global memory (right).

margin. This difference can be explained in part by the caching and 2D spatial locality optimization of the textures, although the random distribution of the seed points severely limits their influence (as demonstrated in Section 4.6.2). Additionally, the CUDA Programming Guide suggests that textures provide better throughput when the memory reads are not fully coalesced, which is the case for our algorithm. Note that in our final version, we use texture-filtering interpolation rather than in-kernel interpolation, which means that the difference in running times between global memory and texture will be even larger than those reported in Table 5.2.

5.3.4 Limiting Factor

The performance of CUDA algorithms may be limited by one or more factors. The two most common limiting factors are the *memory throughput* between the processors and the device memory, and the *computational throughput* in terms of *Floating-Point Operations per Second* (FLOPS). In this section, we aim to find the limiting factor of our algorithm. For this purpose, we use a set of 4096 seed points (i.e., 4096 threads), each of which is propagated through a synthetic data volume for exactly 2048 integration steps. The threads are divided into 64 blocks of 64 threads each, which was identified as the optimal configuration, both by the Occupancy Calculator and through benchmarking. The 4096 seed points are randomly distributed throughout the volume.

In each step of the integration process, a thread interpolates four tensors of six elements each at the current fiber position. For this interpolation, we first need to fetch the four tensors of the eight voxels surrounding the fiber head. Ignoring the influence of the texture cache, this means that in each step, a thread reads $8 \text{ voxels} * 4 \text{ tensors} * 6 \text{ elements} * 4 \text{ bytes} = 768$ bytes of data from the texture memory. With 4096 threads performing 2048 steps each, the algorithm as a whole reads a total of $768 * 2048 * 4096 = 6,442,450,944$ bytes. We ignore the data transfer resulting from reading seed point data and writing output data, since both are much smaller than the amount of texture data transferred. With a running time of approximately 165 milliseconds, this gives us a memory throughput of approximately **39 GB/s**. Since this is well below the theoretical maximum bandwidth of the GTX 260 (111.9 GB/s), we cannot conclude that the algorithm is bandwidth-limited, based on this benchmark.

Next, we compute the number of FLOPS for our algorithm. Using the `decuda` decompiler [28], we decompile the kernel code, and count the number of floating-point operations in the assembly code. Specifically, we count the number of floating-point operations used in the computation of the Christoffel symbols and the execution of the second-order Runge-Kutta step. Between them, these two steps create all floating-point operations within the main integration loop; the other instructions in the main loop body are all concerned with

fetching from or writing to memory. Inspection of the assembly code reveals that the floating-point operations used are:

- Addition (`add`)
- Multiplication (`mul`)
- Multiplication + Addition, i.e. $o = a * b + c$ (`mad`)
- Base-2 Logarithm (`lg2`)
- Base-2 Exponentiation (`ex2`)

With the exception of `mad`, all operations are counted as one FLOP; `mad` is counted as two FLOP's. Counting these instructions revealed the following results:

```

32  add's
52  mul's
75  mad's
6   lg2's
13  ex2's

```

This gives us a total of 256 floating-point operations per integration step. Since we have 4096 threads with 2048 steps each, the complete algorithm performs 2,147,483,648 FLOP's. With a running time of 165 milliseconds, this gives us a total of roughly 13 GFLOPS, which is well below the theoretical maximum of 715 GFLOPS [40]. Note that we did not count those floating-point operations performed in the interpolation process; since we use texture-filtering, these operations are performed by the dedicated texture-filtering hardware rather than the processors. To our best knowledge, texture-filtering operations do not count towards the maximum amount of FLOPS reported by nVIDIA either.

Since both the memory throughput and the computational throughput are well below their respective maximum, it is not yet clear which of the two is the limiting factor. To determine the limiting factor, we propose two experiments. In the first experiment, the amount of data read from the textures is reduced while keeping the number of floating-point operations constant, and in the second experiment, we reduce the number of FLOP's without reducing the number of texture fetches.

Experiment 1

To reduce the amount of data read per voxel without changing the number of floating-point operations, we use the knowledge that in our synthetic data set, all elements of the derivative tensors are zero in all directions (i.e. all tensors are the same). Thus, we can replace certain derivative tensor values in the computation of the Christoffel symbols by other derivative tensor values, allowing us to reduce the number of texture reads without changing the number of FLOP's. Recall from Section 4.6.4 that the four tensors of a voxel are stored in six different textures, each with a four-element vector per texel. The number of vectors read per voxel is varied from six (i.e. all elements) down to three.

The results of this experiment are shown in Table 5.3. From this, it can be concluded that reducing the total amount of data transferred between the processors and the device memory strongly influences the running time of the algorithm, meaning that the memory

throughput is indeed a limiting factor. Also note that the amount of GFLOPS increases as the running time goes down, indicating (as will be confirmed in the second experiment) that the computational throughput is not a limiting factor.

Vectors	Running Time (ms)	Bandwidth (GB/s)
6	165.133	38.8
5	134.124	40.0
4	103.143	41.6
3	90.554	35.6

Table 5.3: Effects on the running time and total memory throughput of changing the amount of data read per voxel.

Experiment 2

To reduce the number of floating-point operations without changing the amount of data read per point, we simply replace the second-order Runge-Kutta method by Euler’s method. Due to the nature of our synthetic data field and the initial directions of our fibers, doing so does not affect the fiber trajectories. Inspection of the the assembly code produced by *decuda* reveals that the computation of the Christoffel symbols and the integration step now takes 32 *add*’s, 39 *mul*’s, 57 *mad*’s, 3 *lg2*’s, and 7 *exp2*’s, for a total of 195 FLOP’s. However, the running time is not impacted by this change: 165.133 ms for Runge-Kutta versus 165.347 ms for Euler’s method. The total amount of GFLOPS is reduced from 13 to 10. From this, we can conclude that the computational throughput is indeed not a limiting factor.

Conclusion

We have shown that the limiting factor of our algorithm’s performance is the memory throughput. This explains the results shown in Figure 5.8: since the computational throughput does not limit the performance of our algorithm, the occupancy has very little influence on the total running time. The actual memory throughput obtained in Experiment 1 is significantly lower than the maximum bandwidth of the GTX 260 (39 GB/s versus 111.9 GB/s). However, we do note that this result was obtained by using randomly distributed seed point locations. As shown in Section 4.6.2, the spatial locality of the seed points is of great influence of the running time. In fact, when we replace the seed points used to obtain the results of Experiment 1 with a more dense distribution, running times do indeed go down, in accordance with the results shown in Figure 4.9.

The fact that the spatial distribution of the running time has a noticeable influence on the running time allows us to attribute the fact that the actual memory throughput is well below the maximal bandwidth to the following potential contributing factors:

- As stated in Section 4.6.2, DDR memory experiences a certain overhead when subsequent reads access different parts of a data range. Since a low spatial locality of the seed points will lead to such scattered access pattern, this overhead may explain our relatively low memory throughput.
- Section 4.6.2 also states the assumptions that CUDA textures have been optimized for 2D spatial locality using some sort of space filling curve. The absence of spatial locality

prevents us from fully taking advantages of these optimization methods.

- The throughput of the texture fetching and filtering units may become a limiting factor when large numbers of voxels are involved. The documentation of the GTX 260 states that it should be able to process 36.9 billion texels per second [37], while our implementation only loads 39 billion bytes (of multi-byte texels) per second. However, this figure is based on 2D bilinear interpolation, while we use 3D trilinear interpolation. Furthermore, the definition of a texel seems to be uncertain (e.g. if a texture contains a 4-element vector at each voxel, can we process 36.9 billion vectors per second, or 36.9 billion vector elements?).
- Finally, the fact that our memory reads are still very much uncoalesced probably does prevent us from reaching higher levels of efficiency for the memory transfers. While the CUDA Programming Guide [38] does state that textures can help to reduce the effects of memory reads that are not fully coalesced, there is no reason to assume that texture completely remove the need for coalesced access patterns.

5.4 GPU versus CPU

In this section, we aim to find the speed-up factor of our CUDA algorithm compared to a C++ implementation. Since it was established in Section 5.2.5 that the running time of the data-preparation stage is of lesser important when the performance of the algorithm as a whole is considered (as the data-preparation stage only needs to be executed once, after which the tracking stage can be executed an arbitrary number of times), we only focus on the tracking algorithm itself. Furthermore, we consider a basic version of the algorithm, without the optional stopping criteria, and without the computation of the connectivity measure. The C++ version of the algorithm is a bare version made specifically for this benchmark; by using this simple, efficient C++ implementation, we ensure that the CPU version experiences no unnecessary computational overhead. The computations performed by the C++ version are largely the same as the performed by the GPU version, with the main difference being the interpolation: since we cannot use texture-filtering interpolation in the C++ version, we are forced to do the interpolation ourselves.

As mentioned before, the GPU used during this benchmark is an NVIDIA GTX 260, which has a total of 192 scalar processors (i.e. 24 multiprocessors), a processor clock speed of 1242 MHz, and 1 GigaByte of device memory. The CPU used is an Intel Core i5 750, which has four cores (of which only one is used), and a clock frequency of 2.67 GHz. The PC containing this CPU also contains 4 GB of RAM, of which only 3.2 GB can be used due to the restrictions of a 32-bit Operating System. Overhead related to memory transfer and allocation is not included in the testbench; we argue that we can ignore this factor, given the facts that this overhead is relatively small (see Section 5.2) and that the difference in running times is very large (as will be shown in this section). We use the number of seed points as the main variable in this benchmark.

Table 5.4 shows the running times for the two implementations. In both cases, the running time depends linearly on the number of seed points. As we can see, the speed-up factor of the GPU algorithm, compared to the C++ version, is roughly 60. In Section 5.3.4, we computed that the GPU algorithm processes roughly 13 billion Floating-Point Operations per Second (FLOPS). In the disassembly code of the C++ implementation, we count a total of 814 floating-point operations, giving the CPU version of the algorithm a total computational

Number of Seed Points	Running Time GPU (ms)	Running Time CPU (s)	Speed-Up
1024	41.187	2.406	58.417
2048	82.337	4.889	59.378
3072	123.230	7.373	59.831
4096	160.168	9.732	60.761

Table 5.4: Benchmark results for GPU and CPU implementation of the geodesic fiber tracking algorithm.

throughput of roughly 0.7 GFLOPS, 18.6 times lower than that of the GPU version. Finally, we can divide the amount of GFLOPS by the processor clock frequency to obtain the number of floating-point instructions per clock cycle, which is roughly 10 for the GPU, and 0.26 for the CPU.

Parallel processing theory teaches us that a parallel implementation is efficient if the running time of the parallel algorithm multiplied by the number of processors is less than the running time of the fastest sequential algorithm. With 192 scalar processors in the GTX 260, our algorithm does not meet this definition of efficiency. However, we note here that the definition assumes fully autonomous processors, while the Scalar Processors in the GTX 260 are Single Instruction, Multiple Threads (SIMT) processors, which means that a single instruction unit is shared between eight processors. Thus, the standard definition of efficiency is not fully applicable to our implementation.

The speed-up factor of roughly 60 times also applies for real data: On a CPU, the construction of all fibers used to construct Figure 5.3 took 43.6 seconds, while the CUDA algorithm did it in 0.711 seconds. No visible differences in fiber trajectories were noticed between the results of the CPU version and those of the GPU version. While floating-point computations in CUDA are potentially less accurate than those on a CPU [38], these differences do not have a significant impact on the overall accuracy of the algorithm.

5.5 Conclusions

In this section, we have shown that the CUDA implementation of the geodesic fiber-tracking algorithm is indeed capable of significantly reducing the running time. When looking only at the tracking stage, we have achieved a speed-up factor of roughly 60 when compared to a C++ implementation of the same algorithm. We have also obtained a formula which expresses the running time of the data-preparation stage as a function of the image size. Finally, we have used benchmarks to quantify the influences of optimization strategies and additional features, as introduced in Chapter 4, on the total running time.

The performance of our algorithm has been shown to be limited by the memory throughput. We sped up the algorithm by reducing the amount of data read by each thread while keeping the amount of instructions constant, thus showing that memory transfers are indeed our main bottleneck. Further, we have offered a number of possible explanations for the fact that the memory throughput of our algorithm is still only roughly a third of the maximum bandwidth of the GPU. However, since we have also demonstrated that texture addressing in combination with texture-filtering interpolation is always preferable to the alternatives (global memory addressing, and texture memory addressing with in-kernel interpolation), and since we cannot reduce the amount of memory read by each thread without affecting the

validity and accuracy of our algorithm, we include that the implementation presented in the previous chapter does indeed represent a good solution to our problem. The large memory requirements and computational complexity of our Tracking kernel, coupled with the large amount of data needed per time integration step and the low spatial locality of the fibers, limits the potential efficiency of our algorithm, making its CUDA implementation less than straightforward.

The final speed-up factor of our CUDA implementation, compared to a sequential implementation on a modern CPU, is roughly 60. This means that users can now compute in seconds what would previously take minutes. As stated before, the term "real-time" has little meaning for our algorithm, as there is no clear goal in terms of, for example, the amount of fibers that must be computed for a certain application. However, since running times of up to fifteen minutes were reported for some applications, the benefits of accelerating these computations on a GPU are obvious.

Chapter 6

Conclusion

In this thesis, we have described the design and implementation of the GPU-based acceleration of a DTI fiber-tracking algorithm. Such an algorithm uses the Diffusion Tensor Imaging Data, which models the diffusion of water molecules within the human body, to deduce the structure of fibrous biological tissues. The area of application for the algorithm described in this algorithm is the white matter, which is a giant, complex network of fibrous neurons. Fiber tracking algorithms can be used to find connections between different regions of the brain, thus providing insight into its physical structure. Furthermore, these algorithms may be used to identify anomalies in the white matter, which aids the diagnosis and treatment of a number of neurological diseases.

The fiber-tracking algorithm discussed in this thesis is based on the theory of geodesics. A geodesic is defined as the shortest path through a tensor-warped space. Within DTI data, such a shortest path is likely to correspond to a physical connection in the brain. This algorithm has been shown to be more resistant to noise than similar algorithms and allows us to find multiple valid connections between regions. From the definition of the geodesic, we derive a set of Ordinary Differential Equations, which allow us to iteratively construct geodesic paths by numerically solving a time integration step. We can use a connectivity measure to quantify the strength of a connection between two different regions of the brain. The Region-to-Region Connectivity measure allows us to find potential connections between regions by first tracking a large number of fibers from a seed point in all directions, and subsequently selecting only those fibers that 1) intersect the target region, and 2) have a large connectivity measure value at the point of intersection.

One downside of the geodesic fiber-tracking algorithm is its computational complexity, which is far greater than that of more straightforward algorithms. Especially when using the Region-to-Region Connectivity measure, which requires the computation of a large number of fibers, the running times of the algorithm becomes impractically large. For this reason, we have accelerated the algorithm using CUDA. This language, developed by GPU manufacturer NVIDIA, allows us to write software for the highly parallel architecture of modern Graphics Processing Units. Both the actual fiber-tracking process, which solves the system of ODEs by means of a time integration step, and the computation of the data needed by the tracking process, were implemented as CUDA kernels. While the kernels of the data-preparation stage have been shown to be significantly faster than a sequential CPU implementation, the focus of our optimization effort has been the Tracking kernel.

Acceleration of the tracking process presented a number of challenges. Due to the computational complexity of the fiber tracking algorithm, the Tracking kernel is quite large, both in terms of memory requirements and in terms of the number of computations. It provides

fiber-level parallelism (i.e., each thread computes a single fiber), but does not attempt to guarantee spatial locality between the threads in one thread block, resulting in scattered and inefficient memory access patterns. Finally, the algorithm requires the trilinear interpolation of 24 individual values per step, which has been shown to make the performance of the kernel limited by the memory throughput. Textures have been used to alleviate some of these problems: texture caching allows for more efficient data transfer, while the texture-filtering interpolation functionality outsources the interpolation process to dedicated blocks of hardware, thus allowing for smaller and faster kernels.

Although the limiting factor of the performance of the algorithm has not been fully identified, we have shown that each of the design decisions made during development has resulted in an overall increase of speed. While we do not claim that the solution presented herein is optimal, we do believe that the final results demonstrate the algorithm's overall efficiency. The speed-up factor when comparing our solution to a sequential CPU implementation is roughly 60, which makes the algorithm far more suitable for practical applications than before. As such, we conclude that the research presented herein constitute a successful attempt to accelerate a complex algorithm using both the advantages and limitations of the highly parallel GPU architecture.

6.1 Future Work

Since both the development of fiber-tracking algorithms for DTI data and the acceleration of data using GPGPU languages remain active areas of research, there is a substantial body of potential future work. When we limit ourselves to the CUDA-aided acceleration of the geodesic fiber-tracking algorithm under discussion, we identify the following possible improvements to the work presented herein:

- The **further optimization** of the CUDA kernels discussed in this thesis may lead to an additional increase in performance. As we have shown in the previous chapter, the performance of especially the two data-preparation kernels is still less than optimal. Additional optimization efforts may reveal bottlenecks that were not uncovered in this thesis, allowing for further reduction of the running time.
- A number of **post-processing** steps may also be implemented in CUDA. Currently, the Region-to-Region Connectivity method has only been implemented on the CPU: Once the CUDA algorithm has terminated, all computed fibers are copied back to the CPU, where a sequential algorithm filters out those fibers that do not intersect the target region, and those fibers that have low connectivity measure values at the target region. Checking for intersection with a target region and computing the connectivity measure may very well be accelerated using CUDA, as it intuitively allows for fiber-level parallelism.
- The **visualization** of the resulting fibers may also be directly integrated in then CUDA implementation. Currently, the DTITool research software containing our CUDA implementation copies the computed fibers back to the CPU, and after some post-processing, again copies them to the GPU for visualization. Since CUDA does allow us to write to the screen buffer, we may be able to skip these unnecessary steps, allowing us to complete the entire fiber tracking process - from computation to visualization - on the GPU.

Bibliography

- [1] A. L. Alexander, K. M. Hasan, M. Lazar, J. S. Tsuruda and D. L. Parker. "Analysis of Partial Volume Effects in Diffusion-Tensor MRI," in *Magnetic Resonance in Medicine*, Volume 45, Issue 5, Pages 770-780, 2001.
- [2] H-E. Assemlal, D. Tschumperlé and L. Brun. "Fiber Tracking On HARDI Data Using Robust ODF Fields," in *Proceedings of IEEE International Conference on Image Processing (ICIP)*, Pages 133-136, 2007.
- [3] L. J. Astola, L. M. J. Florack and B. M. Ter Haar Romeny. "Measures for Pathway Analysis in Brain White Matter using Diffusion Tensor Imaging," in *Proceeding of Information Processing in Medical Imaging*, Lecture Notes in Computer Science, Volume 4584, Pages 642-649, 2007.
- [4] P. J. Basser, J. Mattiello and D. LeBihan. "Estimation of the Effective Self-Diffusion Tensor from the NMR Spin Echo," in *Journal of Magnetic Resonance*, Volume 103, Issue 3, Pages 247-254, 1994.
- [5] P. J. Basser and C. Pierpaoli. "Microstructural and Physiological Features of Tissues Elucidated by Quantitative-Diffusion-Tensor MRI," in *Journal of Magnetic Resonance*, Volume 111, Issue 3, Pages 209-219, 1996.
- [6] P. J. Basser, S. Pajevic, C. Pierpaoli, J. Duda and A. Aldroubi. "In Vivo Fiber Tractography Using DT-MRI Data," in *Magnetic Resonance in Medicine*, Volume 44, Issue 4, Pages 625-632, 2000.
- [7] P. J. Basser and S. Pajevic. "Spectral Decomposition of a 4th-order Covariance Tensor: Applications to diffusion tensor MRI," in *Signal Processing*, Volume 87, Pages 2202-2236, 2007.
- [8] R. Brecheisen, A. Vilanova, B. Platel and B. M. ter Haar Romeny. "Flexible GPU-Based Multi-Volume Ray-Casting," in *Proceedings of Vision, Modeling and Visualization 2008*, Pages 301-312, 2008.
- [9] J. S. W. Campbell, K. Siddiqi, B. C. Vemuri and G. B. Pike. "A Geometric Flow for White Matter Fiber Tract Reconstruction," in *Proceedings of IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI)*, Pages 505-508, 2002
- [10] T. E. Conturo, N. F. Lori, T. S. Cull, E. Akbudak, A. Z. Snyder, J. S. Shimony, R. C. McKinstry, H. Burton and M. E. Raichle. "Tracking Neuronal Fiber Pathways in the Living Human Brain," in *Proceedings of the National Academy of Sciences of the United States of America*, Volume 96, Issue 18, Pages 10422-10427, 1999.

- [11] H. Y. Carr and E. M. Purcell. "Effects of Diffusion on Free Precession in Nuclear Magnetic Resonance Experiments," in *Physical Review*, Volume 94, Issue 3, Pages 630-638, 1954.
- [12] A. F. M. DaSilva, D. S. Tuch, M. R. Wiegell and N. Hadjikhani. "A Primer on Diffusion Tensor Imaging of Anatomical Substructures," in *Neurosurg. Focus*, Volume 15, Issue 1, Pages 1-4, 2003.
- [13] P. van Gelderen P, M. H. de Vleeschouwer, D. DesPres, J. Pekar, P. C. van Zijl, C. T. Moonen. "Water Diffusion and Acute Stroke," in *Magnetic Resonance in Medicine*, Volume 31, Issue 2, Pages 154-163, 1994.
- [14] M. Guye, G. J. M. Parker, M. Symms, P. Boulby, C. A. M. Wheeler-Kingshott, A. Salek-Haddadi, G. J. Baker and J. S. Duncan. "Combined Functional MRI and Tractography to Determine the Connectivity of the Human Primary Motor Cortex In Vivo," in *NeuroImage*, Volume 19, Pages 1349-1360, 2003.
- [15] R. Farber. "CUDA, Supercomputing for the Masses," from *drdobbs.com*, 2009.
- [16] M. Filippi, M. Cercignani, M. Inglese, M. A. Horsfield and G. Comi. "Diffusion Tensor Magnetic Resonance Imaging in Multiple Sclerosis," in *Neurology*, Volume 56, Pages 304-311, 2001.
- [17] K. M. Hasan and P. A. Narayana. "Computation of the Fractional Anisotropy and Mean Diffusivity Maps Without Tensor Decoding and Diagonalization: Theoretical Analysis and Validation," in *Magnetic Resonance in Medicine*, Volume 50, Issue 3, Pages 589-598, 2003.
- [18] E. L. Hahn. "Spin Echoes," in *Physical Review*, Volume 80, Issue 4, Pages 580-594, 1950.
- [19] S. Jbabdi, P. Bellec, R. Toro, J. Daunizeau, M. Pélérini-Issac, H. Benali. "Accurate Anisotropic Fast Marching for Diffusion-Based Geodesic Tractography," in *International Journal of Biomedical Imaging*, 2008.
- [20] W.-K. Jeong, P. T. Fletcher, R. Tao and R. T. Whitaker. "Interactive Visualization of Volumetric White Matter Connectivity in DT-MRI Using a Parallel-Hardware Hamilton-Jacobi Solver," in *Proceedings of IEEE Visualization 2007*, Pages 1480-1487, 2007.
- [21] L. Jonasson, P. Hagmann, X. Bresson, R. Meuli, O. Cuisenaire and J.-P. Thiran. "White Matter Mapping in DT-MRI Using Geometric Flows," in *Computer Aided Systems Theory - EUROCAST 2003*, Lecture Notes in Computer Science, Volume 2809, Pages 585-596, 2004.
- [22] D. K. Jones, D. Lythgoe, M. A. Horsfield, A. Simmons, S. C. Williams, and H. S. Markus. "Characterization of White Matter Damage in Ischemic Leukoaraiosis with Diffusion Tensor MRI," in *Stroke*, Volume 30, Pages 393-397, 1999.
- [23] Jürgen Jost. "Riemannian Geometry and Geometric Analysis," Springer, 5th Edition, Pages 13-28, 2008.
- [24] C. G. Koay, L. -C. Chang, J. D. Carew, C. Pierpaoli, P. J. Basser. "A Unifying Theoretical and Algorithmic Framework for Least Squares Methods of Estimation in Diffusion Tensor Imaging," in *Journal of Magnetic Resonance*, Volume 182, Pages 115-125, 2006.

- [25] M. A. Koch, D. G. Norris and M. Hund-Georgiadis. "An Investigation of Functional and Anatomical Connectivity using Magnetic Resonance Imaging," in *NeuroImage*, Volume 16, Pages 241-250, 2002.
- [26] A. Köhn, J. Klein, F. Weiler and H.-O. Peitgen. "A GPU-based Fiber Tracking Framework using Geometry Shaders," in *Proceedings of SPIE Medical Imaging 2009*, Volume 7261, Pages 72611J-172611J-10, 2009.
- [27] P. Kondratieva, J. Krüger and R. Watermann. "The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization," in *Proceedings of IEEE Visualization 2005*, 2005.
- [28] W. J. van der Laan. "decuda 0.4.2", from wiki.github.com/laanwj/decuda/, 2010.
- [29] M. Lazar, D. M. Weinstein, J. S. Tsuruda, K. M. Hasan, K. Arfanakis, M. E. Meyerand, B. Badie, H. A. Rowley, V. Haughton, A. Field and A. L. Alexander. "White matter tractography using diffusion tensor deflection," in *Human Brain Mapping*, Volume 18, Issue 4, Pages 306321, 2003.
- [30] M. Lazar and A. L. Alexander. "White Matter Tractography Using Random Vector (RAVE) Perturbation," in *Proceedings of 10th ISMRM Annual Meeting*, Page 539, 2002.
- [31] D. Le Bihan, E. Breton, D. Lallemand, P. Grenier, E. Cabanis and M. Laval-Jeantet. "MR Imaging of Intravoxel Incoherent Motions: Application to Diffusion and Perfusion in Neurologic Disorders," in *Radiology*, Volume 161, Issue 2, Pages 401-407, 1986.
- [32] C. Lenglet, R. Deriche and O. Faugeras. "Inferring White Matter Geometry from Diffusion Tensor MRI: Application to Connectivity Mapping," in *Proceedings of Computer Vision - ECCV 2004*, Lecture Notes in Computer Science, Volume 3024, Pages 127-140, 2004.
- [33] L. Marner, J. R. Nyengaard, Y. Tang, B. Pakkenberg. "Marked Loss of Myelinated Nerve Fibers in the Human Brain with Age," in *The Journal of Comparative Neurology*, Volume 462, Issue 2, Pages 144-152, 2003.
- [34] Microsoft Corporation. "Pipeline Stages (Direct3D 10)," in *Programming Guide for Direct3D 10*, MSDN, 2010.
- [35] S. Mori, B. J. Crain, V. P. Chacko and P. C. M. van Zijl. "Three-Dimensional Tracking of Axonal Projections in the Brain by Magnetic Resonance Imaging," in *Annals of Neurology*, Volume 45, Issue 2, Pages 265-269, 2001.
- [36] NVIDIA. "CUDA Occupancy Calculator," from developer.nvidia.com, 2007.
- [37] NVIDIA. "Technical Brief - NVIDIA GeForce GTX 200 GPU Architectural Overview," from nvidia.com, 2008.
- [38] NVIDIA. "CUDA - Programming Guide," from nvidia.com, Version 2.3.1, 2009.
- [39] NVIDIA. "CUDA - C Programming Best Practices Guide," from nvidia.com, 2009.
- [40] NVIDIA. "GeForce GTX 260 - Specifications," from nvidia.com, 2010.

- [41] T. McGraw and M. Nadar. "Stochastic DT-MRI Connectivity Mapping on the GPU," in *IEEE Transactions on Visualization and Computer Graphics*, Volume 13, Issue 6, Pages 1504-1511, 2007.
- [42] A. Mittmann, E. Comunello and A. von Wangenheim. "Diffusion Tensor Fiber Tracking on Graphics Processing Units," in *Computerized Medical Imaging and Graphics*, Volume 32, Issue 7, Pages 521-530, 2008.
- [43] S. Osher and J. A. Sethian. "Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations," in *Journal of Computational Physics*, Volume 79, Pages 1249, 1988.
- [44] G. J. M. Parker, C. A. Wheeler-Kingshott and G. J. Barker. "Distributed Anatomical Connectivity Derived from Diffusion Tensor Imaging," in *Proceedings of Information Processing in Medical Imaging*, Lecture Notes in Computer Science, Volume 2082, Pages 106120, 2001.
- [45] G. J. M. Parker and D. C. Alexander. "Probabilistic Monte Carlo Based Mapping of Cerebral Connections Utilising Whole-Brain Crossing Fibre Information," in *Information Processing in Medical Imaging*, Volume 18, Pages 684-695, 2003.
- [46] G. Parker, S. Luzzi, D. Alexander, C. Wheeler-Kingshott, O. Ciccarelli and M. Lambon-Ralph. "Lateralization of Ventral and Dorsal Auditory-Language Path-Ways in the Human Brain," in *NeuroImage*, Volume 24, Pages 656-666, 2007.
- [47] T. H. J. M. Peeters, A. Vilanova, G. J. Strijkers and B. M. ter Haar Romeny. "Visualization of the Fibrous Structure of the Heart," in *Proceedings of Vision, Modeling and Visualization 2006*, 2006.
- [48] T. H. J. M. Peeters, A. Vilanova and B. M. ter Haar Romeny. "Visualization of DTI fibers using Hair-Rendering Techniques," in *Proceedings of Twelfth Annual Conference of the Advanced School for Computing and Imaging (ASCI)*, Pages 66-73, 2006.
- [49] M. Perrin, C. Poupon, Y. Cointepas, B. Rieul, N. Golestani, C. Pallier, D. Rivière, A. Constantinesco, D. Le Bihan and J.-F. Mangin. "Fiber Tracking in q-Ball Fields Using Regularized Particle Trajectories," in *Information Processing in Medical Imaging*, Lecture Notes in Computer Science, Volume 3565, Pages 52-63, 2005.
- [50] C. Poupon, J.-F. Mangin, C. A. Clark, V. Frouin, J. Régis, D. Le Bihan, I. Bloch. "Towards Inference of Human Brain Connectivity from MR Diffusion Tensor Data," in *Medical Image Analysis*, Volume 5, Issue 1, Pages 1-15, 2001.
- [51] E. Prados, C. Lenglet, J.-P. Pons, N. Wotawa, R. Deriche, O. Faugeras and S. Soatto. "Control Theory and Fast Marching Techniques for Brain Connectivity Mapping," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Volume 1, Pages 1076-1083, 2006.
- [52] G. Reina, K. Bidmon, F. Enders, P. Hastreiter and T. Ertl. "GPU-Based Hyperstreamlines for Diffusion Tensor Imaging," in *Proceedings of Eurographics - IEEE-VGTC Symposium on Visualization 2006*, Pages 35-42, 2006.

- [53] S. E. Rose, F. Chen, J. B. Chalk, F. O. Zelaya, W. E. Strugnell, M. Benson, J. Semple and D. M. Doddrell. "Loss of Connectivity in Alzheimer's Disease: An Evaluation of White Matter Tract Integrity with Colour Coded MR Diffusion Tensor Imaging," in *Journal of Neurology, Neurosurgery & Psychiatry*, Volume 69, Issue 4, Pages 528-530, 2000.
- [54] N. Sepasian, J. H. M. ten Thijs Boonkcamp, A. Vilanova and B. M. ter Haar Romeny. "Multi-valued Geodesic Based Fiber Tracking for Diffusion Tensor Imaging," in *MIC-CAI Workshop on Computational Diffusion MRI (CDMRI)*, Pages 148-158, 2009.
- [55] N. Sepasian, A. Vilanova, L. Florack, and B. ter Haar Romeny. "A Ray Tracing Method for Geodesic-Based Tractography in Diffusion Tensor Images," in *Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA)*, 2008.
- [56] J. A. Sethian. "Level Set Methods and Fast Marching Methods," Cambridge University Press, 1999.
- [57] P. Staempfli, T. Jaermann, G. R. Crelier, S. Kollias, A. Valavanis and P. Boesigera. "Resolving Fiber Crossing using Advanced Fast Marching Tractography Based on Diffusion Tensor Imaging," in *NeuroImage*, Volume 30, Issue 1, Pages 110-120, 2006.
- [58] E. O. Stejskal and J. E. Tanner. "Spin Diffusion Measurements: Spin Echoes in the Presence of a Time-Dependent Field Gradient," in *Journal of Chemical Physics*, Volume 42, Issue 1, Pages 288-292, 1965.
- [59] P. Svetachov, M. H. Everts and T. Isenberg. "DTI in Context: Illustrating Brain Fiber Tracts In Situ," in *Proceedings of Eurographics - IEEE-VGTC Symposium on Visualization 2010*, Volume 29, Issue 3, 2010.
- [60] D. S. Tuch, "Q-Ball Imaging," in *Magnetic Resonance in Medicine*, Volume 52, Pages 1358-1372, 2004.
- [61] V. Volkov and J. W. Demmel. "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," EECS Department, University of California, Berkeley, 2008.
- [62] B. Wandell, A. Sherbondy, R. Dougherty, M. Ben-Shachar. "Diffusion Tensor Imaging and Fiber Tractography of Human Brain Pathways," Psychology Department, Stanford University.
- [63] M. R. Wiegell, H. B. W. Larsson and V. J. Wedeen. "Fiber Crossing in Human Brain Depicted with Diffusion Tensor Imaging," in *Radiology*, Volume 217, Pages 897-903, 2000.
- [64] D. Xu, S. Mori, M. Solaiyappan, P. C. M. van Zijl and C. Davatzikos. "A Framework for Callosal Fiber Distribution Analysis," in *NeuroImage*, Volume 17, Issue 3, Pages 1131-1143, 2002.