MASTER

GPU-accelerated geodesic fiber tracking for HARDI data using CUDA

Hesselberth, M.

Award date:
2016

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
GPU-Accelerated Geodesic Fiber Tracking for HARDI Data using CUDA

M. Hesselberth

Graduation supervisor
Dr. ir. H.M.M. van de Wetering

Graduation tutor
Dr. N. Sepasian

Supervisor
Dr. A.C. Jalba

December 1, 2016
Contents

1 Introduction ................................................. 3
  1.1 Context .................................................. 3
  1.2 Geodesics in tractography ............................... 4
  1.3 Research goals ........................................... 5
  1.4 Contributions ............................................. 6
  1.5 Document structure ....................................... 7

2 Related Work .................................................. 8
  2.1 DTI and HARDI data ....................................... 8
  2.2 Fiber tracking ............................................. 9

3 Geodesic Fiber Tracking ..................................... 11
  3.1 Introduction to geodesics ................................. 11
    3.1.1 Geodesics in Finsler geometry ....................... 12
  3.2 Fiber tracking implementation ............................ 15
  3.3 Initial fiber directions ................................... 16

4 Introduction to CUDA ....................................... 17
  4.1 Computation model ....................................... 18
  4.2 Memory model ............................................ 19
  4.3 Performance considerations ............................... 20
5 CUDA-Based Geodesic Fiber Tracking

5.1 Implementation ............................................. 22
5.2 Tracking kernel ........................................... 24
5.3 Tracking implementation variants ....................... 25
  5.3.1 Standard .............................................. 26
  5.3.2 Standard per neighbor ................................. 29
  5.3.3 Interpolated data ..................................... 31
  5.3.4 Interpolated data per direction ...................... 32

6 Refraction-Inspired Correction ............................ 34

6.1 Motivation .................................................. 34
6.2 Description ............................................... 35
6.3 Implementation .......................................... 38

7 Ricci Scalars ................................................. 41

7.1 Introduction to Ricci scalars ............................... 42
7.2 Implementation .......................................... 42

8 Results ....................................................... 44

8.1 CPU implementation results ............................... 44
8.2 GPU-related results ...................................... 46
  8.2.1 Texture caching ................................... 47
  8.2.2 Number of tracking steps .......................... 48
  8.2.3 Occupancy .......................................... 49
8.3 CUDA implementation comparisons ..................... 52
8.4 “Refraction” results .................................... 54
8.5 Ricci scalar results ..................................... 57

9 Conclusion ..................................................... 59

9.1 Future work ............................................... 60
Chapter 1

Introduction

1.1 Context

In fibrous tissue, such as white matter of the brain, the fibers restrict diffusion of water molecules in certain directions. Instead of moving freely in any direction through Brownian motion, the water molecules can only travel uninhibited in a direction tangential to the local fiber orientation. At any given position in the fibrous tissue, the greatest diffusion is therefore assumed to be found along the direction of fibers. In Diffusion-Weighted Magnetic Resonance Imaging (DW-MRI), this diffusion of water molecules can be measured in any given direction.

Different methods have been introduced in order to represent diffusion profiles. One such approach is to represent the diffusion as a second-order tensor, known as Diffusion Tensor Imaging (DTI) [1]. Here, the diffusion profile is usually acquired by measuring in at least six directions. The DTI model assumes all fibers in a white matter voxel are oriented in the same direction, which results in inaccurate representation of complex fiber behavior such as branching and crossing structure. To overcome these limitations, a technique has been introduced where measurements are performed in many more directions. This method is known as High Angular Resolution Diffusion Imaging (HARDI) [2], where the diffusion profile is represented as a function on a sphere. This allows for more peaks in the diffusion profile, which can indicate multiple fibers passing through a single point from different directions. These two methods are discussed in more detail in Chapter 2.

Figure 1.1 shows an illustration of a slice of DW-MRI data, where the coloring indicates the direction of maximum diffusion. From this image, one can see that it becomes possible to discern different fiber bundles. This means that this imaging method can be used to non-invasively map the fibrous structure in biological matter such as the brain.

In order for the DW-MRI data to become useful in a clinical application, local diffusion measurements need to be transformed into characteristics that can be clearly visualized. While the method of Figure 1.1 is sufficient to show some large scale region-to-region connectivity in a slice of data, this method falls short when attempting to visualize a full 3D volume. Furthermore, the detailed behavior of fibers in areas where bundles cross or branch out is unclear in such an image. A more suitable approach would be to reconstruct individual fibers and to visualize them as three-dimensional curves.
Figure 1.1: An illustration of a slice of DW-MRI data, where pixels are colored by the dominant diffusion direction at their position. A red color indicates that the diffusion is largest in the left-right direction, green means front-back and blue up-down. [3]

1.2 Geodesics in tractography

Reconstructing the fibers in fibrous tissue imaged through DW-MRI is known as fiber tracking or tractography. One class of tractography methods is geodesic fiber tracking [4][5]. In this type of tractography, a globally optimal path is constructed based on some cost function. Using the assumption that physical fibers coincide with paths of maximum diffusion, this cost function is chosen to be the inverse of the diffusion. In addition to a reduced sensitivity to noise compared to simpler techniques such as streamline tractography, these methods also support more complex fiber structures such as branching or crossing fiber bundles. The main focus of this research is multi-valued geodesic fiber tracking [6], which also offers the ability to identify multiple connections between two points. Variants of this technique have been developed for both DTI [7] and HARDI [8] data.

While multi-valued geodesic fiber tracking has benefits over previous tractography techniques, it suffers from a high computational complexity. Computing the metric that defines the cost function requires many more operations than are involved in other tractography methods such as streamline tracking, where only the direction of maximum diffusion needs to be found. However, as this technique performs the same tracking process for a large number of fibers, the performance can be increased by executing some computations in parallel. Using a Graphical Processing Unit (GPU) has been shown to be able to decrease the running time of the geodesic fiber-tracking method by a factor of up to 60 when using DTI data [9]. A GPU-based implementation for the HARDI variant of the geodesic fiber-tracking technique has not yet been developed.
Another downside of the multi-valued geodesic fiber-tracking technique is the generation of false positives. The output of the geodesic method can contain many fibers that do not coincide with physical fibers in the imaged tissue. This is illustrated in Figure 1.2. This image also shows the common process of post-processing the output of the algorithm generated. For the fibers originating from a seed region, an area within the volume from which the starting positions for tractography are selected, only those fibers that pass through a certain second region are retained. While this approach can remove most of the false positives in the output, it also prevents this tractography method from being used to explore connectivity originating from a single region. It is no longer an option to investigate which parts of the brain are connected to a single region, it is only possible to compare how two regions are connected. Another post-processing technique is generating a measure of confidence in a fiber by computing a ‘connectivity measure’ [10], which indicates how strong the diffusion is along the path of this fiber. A high connectivity measure implies a large number of physical fibers coinciding with the path of the tracked fiber.

1.3 Research goals

The main goal of this research is improving the performance of geodesic fiber tracking for HARDI data by utilizing the parallel computing capabilities of a Graphical Processing Unit (GPU) through the CUDA language. Instead of sequentially tracking single fibers in one CPU thread, the high number of computation cores on a GPU allows us to compute many fiber paths in parallel. While this technique has been shown to be effective at reducing running times when using DTI data [9], an implementation for the HARDI variant of the geodesic fiber-tracking calls for an altered design due to differences in the algorithm. For example, the fact that computation of geodesics in a Finsler manifold is dependent on local fiber direction means that metric tensors cannot be fully preprocessed.
Our second goal is the improvement of the fiber output quality. Rather than removing fibers from the output if they are not likely to coincide with the physical fibrous structure, we aim to design a correction method that redirects the fibers during the tracking process. The anisotropy of the diffusion, which is large when the diffusion has large variations in size in different directions, can be used as an indicator of the output quality. Finding a high density coherent bundle of fibers in an area with low anisotropy should be unlikely, as such a structure in the physical material should result in high anisotropy. The correction method we use is therefore designed to contain fibers within areas of high anisotropy.

Finally, as a starting point for future work, we experiment with a dimension reduction technique. In order to reduce the size and dimensionality of the output, it is possible to represent fiber paths using the local curvature of the fibers rather than a full list of three-dimensional fiber positions [11]. In this research, a well-known intrinsic curvature known as the Ricci scalar [12] has been computed along fiber bundles during the fiber-tracking procedure. In the HARDI model, this scalar value contains information on the shape of the underlying diffusion profiles while also having a dependency on the fiber direction.

### 1.4 Contributions

This report describes modifications to the multi-valued geodesic fiber-tracking algorithm that aim to achieve the goals described in the previous section. In order to show the desired improvements, our suggested modifications have been implemented as plug-ins for the pre-existing visualization tool vIST/e

1. In order to show the effectiveness of the multi-valued geodesic fiber-tracking algorithm for HARDI data, a single-thread CPU-based version of this algorithm has been implemented. This implementation also serves as the baseline for analysis of the performance and output quality of our suggested modifications to the fiber-tracking process.

2. A CUDA-based variant of the geodesic fiber-tracking algorithm has been designed and implemented, which computes many fiber paths in parallel. This implementation is shown to decrease the running time of the fiber-tracking algorithm greatly compared to the CPU-based implementation, improving its scalability with regards to tracking a large number of fibers.

3. In order to reduce the occurrence of false positives in the output, we introduce a novel correction method inspired by the physical concept of refraction. After a number of fiber tracking steps, the last segment of a fiber is ‘refracted’ through a virtual surface based on the HARDI data at its position. This method is designed to restrict computed fibers to areas with a high anisotropy, which correspond to physical areas with high fiber density, resulting in a lower number of false positives in a set of output fibers.

4. Ricci scalars have been computed along fiber bundles during the fiber-tracking procedure. These scalars are shown to be useful in distinguishing between two fiber bundles without having to resort to the full three-dimensional fiber tracts, allowing for future work to investigate fiber bundle comparison and identification using this descriptor.

---

1[http://bmia.bmt.tue.nl/Software/vISTe/](http://bmia.bmt.tue.nl/Software/vISTe/)
1.5 Document structure

In Chapter 2, related work regarding DW-MRI data and white matter fiber tracking is discussed. The mathematical concepts behind geodesics and the geodesic fiber-tracking method are described in Chapter 3. Chapter 4 contains an introduction to CUDA-accelerated computing, followed by a description of the CUDA-based implementation of the geodesic fiber-tracking method in Chapter 5. The refraction-inspired fiber correction technique is introduced in Chapter 6. In Chapter 7, the Ricci scalar and its computation are discussed. In Chapter 8 the results of experiments investigating the performance and effects of our implementations are presented. Finally, Chapter 9 contains some conclusions on this research, as well as suggestions for future work in this field.
Chapter 2

Related Work

2.1 DTI and HARDI data

Information is transmitted in the brain across white matter fibers. Finding the structure of these fibers can aid in diagnosis of neurological disorders such as multiple sclerosis and brain tumors [13]. Diffusion-Weighted Magnetic Resonance Imaging (DW-MRI) is an image acquisition technique that measures diffusion of water molecules in multiple directions to reconstruct these fiber connections. In non-fibrous tissue, Brownian motion of water molecules on a microscopic scale would result in isotropic diffusion on a macroscopic scale, i.e. a diffusion that is equal in size in all directions. The white matter fibers inhibit movement of the water molecules perpendicular to the fiber direction. Measuring the local diffusion in a white matter voxel using DW-MRI therefore gives information on the direction of fibers in this voxel. This allows for a non-invasive, in-vivo method of reconstructing the fibrous connections in white matter.

The DW-MRI data is often fitted to a positive definite second-order diffusion tensor (DT), a method known as Diffusion Tensor Imaging (DTI) [1]. Here, the diffusion profile is acquired by measuring in at least six directions. The three eigenvalues of the DT and their corresponding eigenvectors can be used to visualize the diffusion profile as an ellipsoid, as shown in Figure 2.1(a). This model is able to reveal some parts of the structure of the brain. However, the DTI method is limited due to the fact that it assumes all fibers in a voxel are oriented in the same direction. This prevents the method from accurately representing complex structures such as branching or crossing fiber bundles.

In order to overcome the limitations of DTI, it is possible to measure the diffusion in many more directions. High Angular Resolution Diffusion Imaging (HARDI) [2] has been proposed as a solution where the diffusion profile is represented as a function on a sphere, as shown in Figure 2.1(b). Local maxima of this function indicate maximal diffusion, which is assumed to coincide with local fiber tract orientation. As the HARDI model does not make the assumption of a single dominant diffusion direction, it preserves information on complex fiber structures where fibers pass through a voxel in more than one direction. Multiple models for representing HARDI data have been proposed, such as an Orientation Probability Density Function (OPDF) [14] or Q-ball imaging [15]. Each of these models can also be represented by Higher Order Tensors (HOT) [8], which will be used in the geodesic fiber-tracking method described in Chapter 3.
2.2 Fiber tracking

One of the main applications of DW-MRI data is the reconstruction of the underlying structure of fibrous tissue. In a real-world application, this is done in the tracking of fibers through the white matter of a brain, imaged using DW-MRI. These fibers can be used to visualize parts of the brain structure in three dimensions, forming a more informative image than could be constructed by showing slices of the MRI data alone. Furthermore, the tracked fibers indicate connections between different parts of the brain. This can be used in surgical treatment, for example to predict the volume of tissue activated by deep brain stimulation on white matter tracts [16].

A number of different approaches exist for the tracking of fibers through DW-MRI data. We will discuss a number of different tractography categories, which are also listed in Table 2.1 with comparisons of their relative performance with regards to some desirable characteristics. The most straightforward algorithms compute streamlines through the data volume. At each position along the computed fiber, the fiber is oriented in the direction of maximal diffusion [17]. While this is a relatively fast method, the fact that it is completely based on local characteristics makes it sensitive to noise. In addition, only considering the direction of maximal diffusion makes it difficult for streamline tractography to effectively cope with branching or crossing fiber bundle patterns, which are known to occur in white matter [18].

Some problems that arise in streamline fiber tracking can be overcome using non-deterministic methods. Probabilistic tractography uses a probability distribution function for fiber orientation [19]. This distribution is then used to construct a connection confidence between any two regions in the volume by evaluating the portion of random paths originating in the first region that would pass through the second region. Beyond this additional information on connection confidence, probabilistic fiber tracking is also suited to handle branching or crossing fiber bundles. Downsides of this method include the high computational costs involved in evaluating probability distributions for a large number of random paths. Furthermore, the output of this approach is an area that is likely to be connected through fibers, rather than showing an individual fiber path.
Another solution to the problems that occur in classic fiber tracking is to use techniques that involve global approaches. Rather than assuming a fiber follows the local maximum diffusion in each position, fiber tracts can be reconstructed by assuming they follow the most efficient overall diffusion path. Geodesics in a Riemannian manifold can be computed to find the most efficient diffusion paths on a Riemannian manifold with a front-propagation algorithm and using this to construct a connectivity measure throughout the volume without explicitly computing fibers. Computing the geodesics on a Riemannian manifold is often done by solving the Hamilton-Jacobi (HJ) equation. One downside of this approach is that solving the HJ equation only yields a single-valued solution [7] which means it can only reconstruct a single fiber that passes through two points, while it is possible for the human brain to exhibit multiple connections between two points [21].

Sepasian et al. [22] presented a geodesic fiber-tracking algorithm for computing geodesics in Riemannian space. In contrast to many other methods based on geodesics, this approach uses tracing of individual fibers, where the trajectory of a fiber is based on the theory of geodesics in a Riemannian manifold. This algorithm is able to reproduce multiple possible paths between two points, rather than the single-valued solutions obtained with geodesics constructed using the HJ equation. A clear downside of this technique is the fact that tracking a large number of fibers with this algorithm is highly computationally expensive. Techniques involving parallel processing using graphics processing units have been shown to reduce the time required to compute these geodesics [9]. Recently, Sepasian et al. [8] discussed an extension to the geodesic fiber-tracking algorithm for HARDI data, using a Finsler metric rather than a Riemannian one. This metric is not only dependent on the position in space, but also on some outgoing direction.

Table 2.1: An overview of different tractography methods with regards to specific desirable properties.
Chapter 3

Geodesic Fiber Tracking

This research is based around the geodesic-based tractography method for HARDI data as presented by Sepasian et al. [8]. A streamline algorithm using local maxima in the HARDI model to estimate fibers would be purely based on local characteristics and therefore sensitive to noise. Geodesic algorithms use a global approach, based on the assumption that physical fibers follow the most efficient diffusion path.

3.1 Introduction to geodesics

Fibers are now considered as geodesics, or shortest paths through a manifold. A manifold is a topological space which is not necessarily Euclidean, although it is presumed to be locally Euclidean. Here, a Riemannian manifold is used, where the geometric properties of the space are described by a so-called metric, which is denoted as a positive-semidefinite 3×3 tensor $g_{ij}$ (where $i$ and $j$ range from 1 to 3). In essence, this metric describes the shape of the space by prescribing a distance measure between two points to compute a distance, through

$$s^2 = g_{11}(dx_1)^2 + g_{12}(dx_1)(dx_2) + g_{22}(dx_2)^2 + \ldots$$

(3.1)

Riemannian manifolds are a generalized form of Euclidean space. For example, the metric that defines Euclidean space as a specific case of a Riemannian manifold, $g_{ij} = \delta_{ij}$ where $\delta$ is the Kronecker delta, is a metric that results in the normal Euclidean distance measure:

$$s^2 = (dx_1)^2 + (dx_2)^2 + (dx_3)^2$$

(3.2)

In this application, the goal is to generate fibers as shortest paths through some non-Euclidean space. The shape of this space should be dependent on the DW-MRI data in a way that results in globally shortest paths that coincide with a route of maximum diffusion. The metric that defines our manifold should therefore yield a short distance between two nearby points if the diffusion along the vector connecting them is large.

In DTI, geodesic fiber tracking uses a Riemannian manifold, with the inverse of the diffusion tensor as the metric tensor [7]. However, the DTI model cannot recover the more complex underlying structures such as crossing and fanning fibers. The HARDI model has been used, as it is better able to recover these structures due to the much higher number of image acquisition directions used. As the HARDI model has a much larger variation in the diffusion in different
directions, the approach used in the context of DTI is extended to use a Finsler manifold. This manifold is a generalization of the Riemannian manifold. A Finsler metric tensor does not merely depend on the position in space, but also has a directional dependency. This metric is better suited for use with the HARDI model, as it more strongly conveys the information available in many directions as found in the input data. We now globally follow the reasoning given in the initial presentation of multi-valued geodesic fiber tracking for HARDI data [6][8] to derive the equations required to implement this method.

3.1.1 Geodesics in Finsler geometry

In order to use a geodesic fiber-tracking algorithm on a Finsler manifold, a choice must be made on what is to be used as the Finsler metric tensor. Again, the metric should scale with the inverse of the diffusion to ensure a low cost of travel in directions of high local diffusion, so we need to define an inversion of a HARDI diffusion profile. The general form of the diffusion profile is an Orientation Distribution Function (ODF) $P_4(x, y)$, which gives a scalar value for the diffusion at a given position $x$ in a given direction $y$. This function is used to create a Higher Order Tensor (HOT) representation. In this research, a fourth order tensor is used to store the diffusion profile at a given position, but using a tensor of higher order is possible. This tensor $D(x)$ can be seen as a four-dimensional array with a length of three elements in each dimension (for a total of $3^4 = 81$ elements) describing the shape of the diffusion profile at position $x$, and can be written as

$$D(x) = D_{\alpha_1\alpha_2\alpha_3\alpha_4}(x), \quad (3.3)$$

where $\alpha_i = 1, 2, 3$ for $i = 1, 2, 3, 4$. The tensor $D$ is symmetrical, such that any ordering of the indices yields the same result. For example, the values of $D_{1123}$, $D_{2113}$ and $D_{1321}$ are all the same. This means the amount of components needed to fully define the tensor is reduced from 81 to 15. By sampling values of $P_4(x, y)$ across many directions $y$ around on a unit sphere $x$, the tensor coefficients of $D(x)$ are chosen to form an approximation of the ODF, such that:

$$P_4(x, y) \approx D_{\alpha_1\alpha_2\alpha_3\alpha_4}(x) y^{\alpha_1} y^{\alpha_2} y^{\alpha_3} y^{\alpha_4}, \quad (3.4)$$

where once again $\alpha_i = 1, 2, 3$ for $i = 1, 2, 3, 4$. Here and from this point onwards, the Einstein summation convention is used, meaning we sum over repeated indices that occur both in subscript and superscript. This equation therefore represents a sum over a multiplication of an element $D_{\alpha_1\alpha_2\alpha_3\alpha_4}$ of the HOT with different combinations of the three components $y^{\alpha_i}$ of the testing direction $y = (y^1, y^2, y^3)$.

An inverse of the diffusion, which we need to construct the Finsler metric, is now defined as $\tilde{P}_4(x, y)$, a spherical inverse of $P_4(x, y)$:

$$\tilde{P}_4(x, y) \approx \frac{\overline{P}_4(x)}{\overline{P}_4(x, y)} = \tilde{D}_{\alpha_1\alpha_2\alpha_3\alpha_4}(x) y^{\alpha_1} y^{\alpha_2} y^{\alpha_3} y^{\alpha_4}, \quad (3.5)$$

where $\tilde{D}$ is the HOT that fits $\tilde{P}_4(x, y)$ in an approximation analogue to (3.4), and $\overline{P}_4(x)$ is the average value of $P_4(x, y)$ integrated across all directions on the unit sphere. This allows us to find the inverse of the diffusion profile, as illustrated in Figure 3.1.
Figure 3.1: An illustration of spherical inversion in two dimensions. The light blue shape represents a spherical inverse of the dark blue diffusion profile. The norm of vector $\overrightarrow{OM'}$ is the size of a spherical inverse of the ODF in the direction of $M$ at position $O$, using the radius of the dashed circle as the average value of the ODF in (3.5). [8].

Now that we have found a way to represent the inverse of the diffusion, the Finsler norm can be defined. This norm $F(x, y)$ is a function that generates a ‘cost’ of moving in a direction $y$ from a position $x$. As mentioned before, we want our metric to return a low cost when computed in a direction that showed high diffusion. We therefore define the Finsler norm through the spherical inversion of the diffusion. Astola et al. [23] proposed the following form for fourth order tensor representations of the diffusion:

$$F(x, y) = (\tilde{P}_4(x, y))^{1/4}. \tag{3.6}$$

This norm can now be used to define the Finsler metric tensor $g_{\alpha\beta}(x, y)$ for a given position $x$ and a direction $y$, where $\alpha$ and $\beta$ are used to index the components of the tensor $g_{\alpha\beta}$. As the metric tensor is symmetric, it can be represented using six elements in the following structure:

$$g_{\alpha\beta}(x, y) = \begin{bmatrix} g_{11} & g_{12} & g_{13} \\ g_{22} & g_{23} \\ g_{33} \end{bmatrix}(x, y).$$

The value of the Finsler metric tensor can be found from the Finsler norm using [8]

$$g_{\alpha\beta}(x, y) = \frac{1}{2} \frac{\partial^2 F^2}{\partial y^\alpha \partial y^\beta}. \tag{3.7}$$

From substituting 3.6 in 3.7, it follows that [8]

$$g_{\alpha\beta}(x, y) = -2\tilde{P}_4(x, y)^{-3/2}(\tilde{D}_{\alpha\alpha_1\alpha_2\alpha_3}(x)y^{\alpha_1}y^{\alpha_2}y^{\alpha_3})(\tilde{D}_{\beta\beta_1\beta_2\beta_3}(x)y^{\beta_1}y^{\beta_2}y^{\beta_3})$$

$$+ 3\tilde{P}_4(x, y)^{-1/2} \tilde{D}_{\alpha\beta\alpha_1\alpha_2}(x)y^{\alpha_1}y^{\alpha_2} \tag{3.8}$$

which is the form of the Finsler metric used in the fiber-tracking process. For each position in space, the metric tensor can be calculated for any given direction from the elements of the
fourth order tensor representation of the diffusion profile.

Now, in order to generate geodesics, a bounded curve $C$ must be generated where the length is minimized. We consider a parametrization $x = \chi(t), a \leq t \leq b$, where $t$ denotes the arc-length. The length of $C$ in Finsler space is then given by

$$J[\chi] = \int_a^b F(\chi(t), \dot{\chi}(t))dt,$$

(3.9)

where $\dot{\chi}(t) = \frac{dx}{dt}$. Minimizing this curve can be done by solving the set of Euler-Lagrange equations [24]:

$$\frac{d}{dt} \left( \frac{\partial F}{\partial y^\alpha} \right) - \frac{\partial F}{\partial x^\alpha} = 0$$

(3.10)

where $y^\alpha = \dot{x}^\alpha = \frac{dx^\alpha}{dt}$.

Substituting the equation for $F$ and solving these equations yields the following geodesic equation for the path of a fiber:

$$\ddot{x}^\alpha + \Gamma^\alpha_{\beta \gamma} \dot{x}^\beta \dot{x}^\gamma = 0$$

(3.11)

where $\Gamma^\alpha_{\beta \gamma}$ is the set of 18 so called Christoffel symbols given by

$$\Gamma^\alpha_{\beta \gamma}(x, y) = \frac{1}{2} g^{\alpha \kappa}(x, y) \left( \frac{\partial g_{\beta \kappa}}{\partial x^\gamma} + \frac{\partial g_{\gamma \kappa}}{\partial x^\beta} - \frac{\partial g_{\beta \gamma}}{\partial x^\kappa} \right)$$

(3.12)

where $g^{\alpha \kappa}(x, y)$ is the inverse tensor of $g_{\alpha \kappa}(x, y)$, computed by using the regular HOT representation of the diffusion in (3.8) rather than a spherical inversion.

With this equation, a numerical solution can be found by computing the Christoffel symbols using the Finsler metric tensors and their partial derivatives for the position and direction of a fiber. These symbols can then be used in conjunction with a standard ordinary differential equation solver, such as the Runge–Kutta method, to find the next position and direction for this fiber.
3.2 Fiber tracking implementation

In order to find geodesic curves that indicate fiber tracts, we need to solve the system (3.11). Introducing \( u^\gamma(t) := \dot{x}^\gamma(t) \) for \( \gamma = 1, 2, 3 \) allows us to rewrite (3.11) as

\[
\begin{align*}
\dot{x}^\alpha &= u^\alpha \\
\dot{u}^\alpha &= -\Gamma^\alpha_{\beta\gamma} u^\beta u^\gamma
\end{align*}
\]  

(3.13)
with \( \Gamma^\alpha_{\beta\gamma} \) as defined in (3.12). Using an ODE solver such as the fourth order explicit Runge-Kutta method, this system can be solved to generate a sequence of fiber positions from a starting location and direction. Partial derivatives of \( g_{\alpha\beta}(x, y) \) required for computation of the Christoffel symbols are approximated using a second order central difference scheme.

The different steps of the algorithm are mostly unchanged from the initial presentation by Sepasian et al.[8], only differing in the generation of initial directions for new fibers, leading to the following algorithm:

**Algorithm 1 Geodesic fiber tracking**

1. **function** FIBERTRACKING(S)
2. Generate \( \tilde{D} \), the fourth order tensor fit of the inverse diffusion profile \( \triangleright \) See (3.5)
3. for all seed points \( s \) in \( S \) do
4. \( D \leftarrow \text{GENERATEINITIAlDIRECTIONS}(s) \) \( \triangleright \) See Section 3.3 for details
5. for all directions \( d \) in \( D \) do
6. Initialize a new fiber \( f \)
7. \( x \leftarrow s \)
8. \( y \leftarrow d \)
9. repeat
10. for all eight corner grid points \( x_i \) of the cell containing \( x \) do
11. \( g_i \leftarrow g_{\alpha\beta}(x_i, y) \) \( \triangleright \) Computed using (3.8)
12. Compute \( \frac{\partial g_{\alpha\beta}}{\partial x^j}(x_i, y) \) \( \triangleright \) Requires metrics at neighbors of cell corners
13. Interpolate values of \( g_i \) and \( \frac{\partial g_{\alpha\beta}}{\partial x^j}(x_i, y) \) to find values at \( x \)
14. Compute Christoffel symbols \( \Gamma^\alpha_{\beta\gamma} \) \( \triangleright \) Computed using (3.12)
15. Compute next position \( x \) and direction \( y \) \( \triangleright \) Computed using (3.13)
16. until stopping condition is met for this fiber

Multiple stopping criteria are used for this fiber-tracking algorithm. One is a limit on the angle between two sequential fiber segments, stopping the tracking process when a fiber makes a sharp turn. The second is a lower limit on the anisotropy. If the diffusion at the position of a fiber is too near to being isotropic, the tracking process is stopped for that fiber. The final stopping criterion is fulfilled when the fiber passes through the bounds of the tracked volume.
3.3 Initial fiber directions

One change in this application of the geodesic fiber tracking technique compared to the previous work regarding HARDI data [8] is in the seed point generation. While the positions of seed points are generated as before, there is a difference in their initial directions. Previously, starting directions were uniformly distributed over the unit sphere for each starting position. This means a notable number of fibers are sent in directions where no physical fiber likely exists. In our implementation, one outgoing fiber is generated per local maximum on the ODF. This ensures a lower number of superfluous fiber computations.

However, geodesic fiber tracking is sensitive to initial direction, so two near identical starting vectors can lead to vastly different output fibers. In order to ensure no interesting directions are missed, the option to send additional fibers per local maximum is added. These extra fibers then start along a cone around the local maximum direction, where the width of this cone depends on the sharpness of the ODF peak. This method increases the density of outgoing fibers in directions with high diffusion compared to the previous work.
Chapter 4

Introduction to CUDA

The main downside of geodesic fiber tracking is the amount of computations needed for a single fiber tracking step. Computing 1000 fibers can easily take several minutes for a single-thread CPU implementation of this fiber-tracking algorithm. However, the operations involved in tracking all of these fibers are in essence the same, which means the development of an implementation using a parallel architecture can greatly improve the running times of the tracking algorithm. To this end, a variant of the fiber-tracking algorithm that can be executed on a Graphics Processing Unit (GPU) has been implemented using the CUDA architecture.

CUDA is a parallel computing architecture, developed by the NVIDIA Corporation, designed to allow software developers to produce programs that run on NVIDIA-branded GPUs. This type of hardware, originally designed for highly-parallel processing of computer graphics, contains many more processor cores than a CPU (albeit at lower frequencies). Using CUDA, this highly parallel structure allows GPUs to be applied successfully in the context of parallel processing outside of visual computing.

Figure 4.1: A model of the structures involved in running CUDA programs. The instruction in a kernel are executed on many threads in parallel. These threads are organized in blocks, which are grouped together to form the execution grid.
4.1 Computation model

In CUDA, running code in the form of functions known as *kernels*, is done in parallel with many threads running simultaneously. As illustrated in Figure 4.1, threads are grouped into *blocks*, which are grouped together to form a *grid*. These blocks and grids can be one-, two-, or three-dimensional structures. Each GPU contains a number of Streaming Multiprocessors (SM), each of which contains a number of processors known as Streaming Processors (SP) or CUDA cores. As indicated in Figure 4.2, blocks of threads are assigned to an SM automatically based on the number of threads and memory requirements of each thread. The multiprocessor divides a block up into groups of 32 threads known as *warps*, which are then executed using one CUDA core per thread.

CUDA employs an architecture known as *SIMT* (Single-Instruction, Multiple-Thread). It executes instructions simultaneously on all threads within a warp, allowing for high instruction throughput. If threads diverge from the rest of the warp through a conditional branch, each branching path is executed sequentially, with the threads that are not on this path disabled during the execution of that path. As the code in these branches is then not executed in parallel on all threads, this can greatly increase running times of algorithms using CUDA.

![Diagram of CUDA grid and blocks](image)

Figure 4.2: An illustration of how a grid of blocks can be distributed at runtime. Depending on the total number of Streaming Multiprocessors (SM) available on the device (and memory requirements of the kernel running in these blocks), different numbers of blocks are assigned to each SM.
4.2 Memory model

Memory on a GPU, known as device memory (in contrast to host memory used by the CPU), can be divided into two groups, on-chip and off-chip. The on-chip memory includes the registers and shared memory, while global, local, texture and constant memory resides off-chip. Each type of memory has different characteristics such as size, latency and possibility of caching. An efficient CUDA program design should include a well thought-out distribution of its storage needs across multiple types of memory to leverage the benefits and downsides of each.

Each Streaming Multiprocessor has a bank of 32-bit registers that can be assigned to the threads executing on it. The amount of registers assigned per thread can vary anywhere between 0 and 255 in recent version of CUDA, depending on the needs of the kernel. As there is only a limited number of registers per SM, the amount of registers required per kernel can limit the amount of threads that can be executed concurrently on a multiprocessor.

The other memory residing on-chip is the shared memory. In contrast to the registers, shared memory is assigned per block, meaning the same memory can be accessed by any thread within that block. This allows for inter-thread communication via on-chip memory with low latency and high throughput. Shared memory can be used for sharing of results between threads to prevent redundant calculations. It can also serve as a buffer to temporarily store input data used by multiple threads, to prevent each thread from having to retrieve data from the slower global or texture memory.

Global memory is a large bank of general purpose memory, with sizes generally ranging between 1 and 8 GB for consumer products. Any address in this memory is globally accessible from both CUDA threads and CPU code. With these characteristics, global memory is well suited to be used as storage for input and output data. Due to the fact that this memory lies off-chip, it incurs a large access latency and offers lower throughput than registers or shared memory. In the latest two iterations of the CUDA architecture, the Kepler and Maxwell offer limited caching of global memory loads in the 16 KB L2 cache.

As mentioned, each thread has access to a maximum of 255 registers. If more per-thread memory is needed, additional variables are automatically stored in local memory. The ‘local’
identifier of this 512 KB block of memory refers to the fact that it can only be accessed by a single thread. Local memory resides off-chip, similar to global memory, meaning it suffers from relatively high latency and low throughput.

A type of memory originally designed to store images for graphics applications, texture memory offers some additional functionality over global memory. While it is limited to read-only access from the perspective of CUDA threads, texture memory also offers spatial caching to reduce the costs of reading data from it. The texture cache is designed to benefit from 2D spatial locality. This means that threads within the same warp will encounter greatly reduced latency when loading data from an address ‘near’ the address that other threads are reading from. Furthermore, the graphics applications that texture memory was originally designed for required rendering of pixels that were not grid-aligned with an input image. This lead to the implementation of cheap interpolation mechanisms. Dedicated hardware on the GPU offers nearest-neighbor or linear interpolation for a low computational cost. Texture memory has been modified to also support one- and three-dimensional textures while retaining these two additional functionalities.

Constant memory is designed as a cached piece of memory that is most effective when used to store data that does not change over the course of a kernel execution. This 64 KB memory bank is cached such that for all threads in a half-warp (16 threads), reading from the constant memory is as fast as reading from a register when all threads read from the same address.

4.3 Performance considerations

In order to fully exploit the possibilities of the CUDA architecture, a parallel implementation needs to be designed with some performance considerations in mind. CUDA offers multiple types of memory, each with strengths and weaknesses, that must be used in appropriate parts of the application. Furthermore, while CUDA-enabled GPUs can perform the distribution and execution of thread blocks across CUDA cores without any involvement of the programmer, some aspects of the hardware must be taken into account. Many different GPUs are available on the market, which requires the programmer to ensure that hardware limitations on some platforms do not limit the effective distribution of the workload.

First, the different types of memory should be used in an appropriate context. Global memory is capable of storing large amounts of data, but the low bandwidth from global memory to threads means that simply reading large amounts of data from global memory to each thread will result in a slow CUDA program. When multiple threads in a block use the same part of the global input data, a typical approach is to have these threads load a portion of this input data to the block’s shared memory. The shared memory then acts as a user-managed cache, storing the common data used by multiple threads in the block. If threads in a block show spatial coherence, i.e. use data elements that are located ‘near’ to each other in an input data volume, texture memory can often be used more efficiently than global memory. The in-built texture cache can then perform the caching automatically. The benefits of using texture memory increase even more when input data is required at positions not aligned with the grid. In that case the cheap interpolation functionality can also be used.

Memory operations should ideally also be coalesced. Writing single variables to the global memory from a thread will incur an access latency each time, while writing a sequence of these variables to the global memory will only incur this penalty once. Furthermore, CUDA memory and caching is optimized to ensure lower performance penalties when ‘nearby’ threads, located
in the same warp, access memory addresses that are near to each other.

The main benefit of using a GPU for highly-parallel computing is the number of cores available. An NVIDIA branded GeForce GTX 780 Ti, a relatively high-end consumer GPU, contains 15 multiprocessors. At 192 CUDA cores per multiprocessor, this means this GPU can execute up to 2880 threads in parallel at any given time. In order to fully exploit the computing power of a GPU, the CUDA program must therefore ensure the number of threads available remains high. The total number of threads that can be assigned to a single multiprocessor is 2048. As mentioned in the previous section, a number of registers are assigned to each thread and a an amount of shared memory is assigned per block. As each multiprocessor has a limited amount of both these types of memory, the memory requirements per thread enforce an upper bound on the possible amount of threads running at any given time. This is typically indicated by the occupancy, defined as

\[
\text{Occupancy} = \frac{L_{\text{warp}}}{\max(L_{\text{register}}, L_{\text{shared}})}, \quad (4.1)
\]

where \(L_{\text{warp}}\) is the limit in blocks that can be assigned to each multiprocessor due to the maximum number of warps (and therefore the number of threads) that can be assigned, \(L_{\text{register}}\) is the limit due to the register requirements per thread and \(L_{\text{shared}}\) is the block assignment limit due to the shared memory requirements per block. A low occupancy is indicative of a kernel that requires too many registers or too much shared memory to run. Keeping the required register count low has the added benefit of avoiding the use of (slow) local memory for storage of variables when the maximum register count is exceeded.

As mentioned in Section 4.1, instructions are distributed to all threads in a warp simultaneously. This means a conditional branch can force execution within some threads to be paused while other threads finish the execution of another branch. Afterwards, the branch of the residual threads is executed. This can greatly increase running times of a program, as the number of threads actively executing instructions on the GPU at any given time decreases.

Shared memory can be used for communication between threads, or as a previously described cache for common input data between threads. While such techniques can help improve the performance of a CUDA program, these do not come without costs. When using shared memory as a cache, threads must be sure that all input data has been moved into shared memory by other threads before attempting to use it. This requires thread synchronization, which is implemented in CUDA as the function \_syncthreads(). This function acts as a barrier to ensure all threads within a block have completed the previous instructions before continuing execution. If shared memory is used to communicate between threads, or to store a value that multiple threads modify during the course of the program’s execution, some lock must be put in place to provide exclusive access to a variable for a critical section of a program. To this end, the CUDA language offers atomic operators such as \_atomicAdd() and \_atomicSub(), allowing for modification of variables in one atomic transaction. Both synchronization and atomic operators incur performance costs by holding back execution of some threads to ensure the integrity of the program execution, so an efficient CUDA program design should keep the use of these operators to a minimum.
Chapter 5

CUDA-Based Geodesic Fiber Tracking

Using the CUDA architecture described in Chapter 4, the geodesic fiber tracking algorithm from Chapter 3 can now be implemented as a GPU-accelerated program. This chapter describes the extensions of the algorithm required to accomplish the parallelized computation of geodesics. The first section describes the overall process, including CPU tasks required to trigger computations on the GPU. The second section describes the details of the four fiber-tracking kernels that have been implemented.

5.1 Implementation

While the bulk of the computational work is done by running CUDA kernels on the GPU, the CPU performs related tasks as well, such as data transfer between RAM and GPU memory and triggering the launch of kernels. An illustration of the CPU perspective of the tracking process is shown in Figure 5.1.

First, the HARDI data needs to be moved to the GPU memory, in order to allow tracking kernels to access it. This data is stored on the GPU in the texture memory. Using texture memory instead of global memory has some benefits for this type of data. Texture memory allows for sampling of data at positions not aligned with the data grid, where the resulting value is computed from a linear interpolation of the surrounding data points, performed on dedicated hardware. This allows for low-cost interpolation of the HARDI data, which is used in two of the four kernel implementations described in the next section. Furthermore, the method of seed point generation, sending multiple fibers out from a seed point position in similar directions, will likely lead to some spatial coherence between fibers. This means this implementation should be able to benefit from the spatial caching available for texture memory. The HARDI data is therefore stored on the GPU in 3D textures. In order to accommodate all fifteen coefficients of the fourth order tensor that forms our data, four textures of the data type float4 are used. This is the largest built-in data type available for texture memory, allowing us to limit the number of texture samples needed for one HARDI data retrieval to four.
After the HARDI data has been transferred to the GPU, a list of seed points is generated in the same way as in the previously mentioned single-thread algorithm. Seed points from a user-supplied region of interest form the initial positions, while the local maxima on the ODF are used to find starting directions for fibers. A subset of these seed points of a certain size is then transferred to the global memory on the GPU. The size of this subset depends on a user parameter, with values generally ranging from 1024 to 4096. As a GPU can execute a large number of threads in parallel, a large number of input seed points ensures the portion of idle threads remains low.

In the next step, up to three kernels are launched. The first is a tracking kernel, which performs a number of iterations of the geodesic fiber-tracking algorithm for the currently active subset of fibers. This kernel uses the seed points and initial fiber directions as input. The output consists of a list of fiber positions that constitute a subsection of a tracked fiber. The final position and direction at the tip of the tracked fiber are then used as the seed point input for the next tracking cycle. The details of the translation from the CPU-based algorithm to a CUDA implementation are discussed in the next section, where four different kernel implementations are presented.

The second kernel that can (optionally) be used applies a correction technique to the output of the previous kernel. If the refraction-inspired correction method (as presented in Chapter 6) is enabled, a ‘refraction’ kernel is launched after the termination of the tracking kernel. This kernel uses the output of the tracking kernel, a list of points (with an outgoing direction) for the next iteration of the algorithm, as input for a correction step. The kernel edits the outgoing direction of the seed point if the current direction is not likely to coincide with the direction of a physical fiber in that location. For each seed point in the global memory, a modified copy of the seed point is stored at the same address. The details of this correction method are given in Chapter 6.

![Flowchart for the CPU-based part of the fiber tracking process.](image-url)

Figure 5.1: Flowchart for the CPU-based part of the fiber tracking process. Arrows to and from the blue box representing the GPU indicate memory transfer between host and device memory. Dotted arrows indicate the triggering of the execution of a CUDA kernel and the return signal from the GPU indicating the completion of the kernel execution. Execution of the process only continues after the execution of a kernel terminates.
Another optional kernel that can be launched after the other kernels have completed is designed to compute a scalar measure at each position along the fiber. This ‘Ricci’ kernel computes the Ricci scalar, which holds information on the curvature of the Finsler geometry at a given position with respect to a given direction. The list of currently-active seed points is used as input to the Ricci scalar computation, as well as the underlying HARDI data. The output of this kernel is a list of scalar values in the global memory, containing one value for each active fiber. The reasoning behind computing Ricci scalars and the implementation in CUDA are given in Chapter 5.

After all kernels have terminated, the global memory of the GPU contains a list of positions per active fiber and possibly a new seed point for the next tracking cycle of each fiber. These two arrays are copied back from the device memory to RAM by the CPU. The new fiber point positions are appended to the ends of partial fibers kept in the host memory. If Ricci scalar values have been computed, the value at each position is linearly interpolated between the Ricci scalar at the last computed position (at the end of the previous tracking cycle) and its value at the newly calculated position.

As mentioned earlier, the tracking kernel stores the fiber position at the end of a tracking cycle in the list of seed points, provided the fiber has not terminated. If the fiber has completed tracking due to fulfilling one of the stopping criteria, an invalid seed point is stored in the global memory instead. This is used by the CPU at the end of a fiber-tracking cycle to replace seed points belonging to terminated fibers by seed points of new fibers. The tracking cycle can then restart with enough fibers being tracked to ensure better utilization of the GPU. If no new seed points are available and the tracking of all active fibers has terminated, the tracking cycle is ended, after which visualization of the resulting fibers can start.

5.2 Tracking kernel

In the previous section, the overall tracking process was given. In this section, we will describe the details of the tracking kernel that will be executed on the GPU. For each fiber, the tracking kernel executes the geodesic fiber-tracking algorithm multiple times to compute a series of new fiber positions before writing the result to the global memory. This is done in an attempt to limit a memory bottleneck that may occur if every thread would write a position vector to global memory at the end of every tracking step. Instead, the tracking kernel executes the algorithm in multitudes of eight iterations. New fiber positions are then written to global memory in blocks of eight at a time.

It may be tempting to increase the number of iterations before a tracking kernel transfers output positions to global memory. As mentioned in Section 4.3, further coalesced memory writing will result in lower running times per kernel. However, this approach has disadvantages. Firstly, the GPU is generally not actively tracking all fibers at the same time. Tracking of new fibers starts when tracking of a previously active fiber terminates. If fibers are tracked for hundreds of integration steps at a time, a large delay between fiber termination and the start of a new fiber can occur. This leads to inactive threads, which is a strong detriment to performance. Furthermore, increasing the length of fiber segments generated in one tracking cycle decreases the accuracy of the results from the two kernels that can be launched after the tracking kernel. The refraction-inspired correction kernel will have a smaller corrective strength if it is very rarely applied. The scalar measure computed by the Ricci kernel is be linearly interpolated between data points, so rarely computing a new value will increase the error in the interpolation.
5.3 Tracking implementation variants

The main difficulty in designing such a kernel is finding the optimal distribution of the workload across CUDA threads. The register limit per thread limits the options of running kernels that require a large amount of memory per thread. As mentioned in Section 4.3, this could lead to low occupancy and longer running times. Spreading the workload of tracking a single fiber across multiple threads in some way could provide a workaround for this problem. Several threads can compute intermediate results and store them in the shared memory of a block. This type of approach can reduce the memory required per thread and allow for more computations to be performed in parallel. However, this does incur some performance costs due to the need for thread synchronization and properly managed access to shared memory.

In order to investigate the effects of the thread configurations, four different approaches have been implemented. Each of these maintains the per-thread nature of the algorithm, tracking a number of fibers in parallel. The way these implementations differ is in the amount of parallelization of computations for a single fiber tracking step. The first implementation, the standard kernel, stays as close as possible to the CPU-based algorithm, running on one CUDA thread per fiber. The standard per neighbor kernel splits the computation of Finsler metrics and their derivatives for the neighboring cell corner voxels across eight threads per fiber, allocating one per voxel. The final two kernels use texture interpolation to find interpolated values of the HARDI data rather than interpolating the Finsler metrics. Of these two, interpolated data per direction attempts to achieve further parallelization over the interpolated data kernel by distributing the computation of the metric derivatives across multiple threads, where each thread computes the partial derivative in one of three directions.

<table>
<thead>
<tr>
<th>Kernel implementation</th>
<th>Number of threads per fiber</th>
<th>Interpolated metrics</th>
<th>Interpolated HARDI data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>1</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Standard per neighbor</td>
<td>8</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Interpolated data</td>
<td>1</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Interpolated data per direction</td>
<td>3</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the characteristics of the tracking kernel implementation variants.
5.3.1 Standard

The first implementation of the geodesic fiber tracking algorithm as a CUDA program adheres closely to the CPU-based variant. Each fiber is tracked by a single CUDA thread, mirroring the behavior of a single CPU core, resulting in the grid structure shown in Figure 5.5.

![Grid layout for the standard fiber-tracking kernel. A block of threads handles computation for multiple fibers. Each thread (marked in yellow) tracks a single fiber.](image)

Figure 5.2: Grid layout for the standard fiber-tracking kernel. A block of threads handles computation for multiple fibers. Each thread (marked in yellow) tracks a single fiber.

The computations involved in the fiber tracking process for this kernel are illustrated in Figure 5.3.

![Flowchart for the standard fiber-tracking kernel.](image)

Figure 5.3: Flowchart for the standard fiber-tracking kernel. The blue box at the top represents the GPU device memory.

First, the position of the seed point and the initial direction are retrieved from the global memory. Next, the Finsler metric tensor $g_{\alpha\beta}(x, y)$ is constructed for each of the eight neighboring voxels forming the corners $V_i$ of the cell that contains the current fiber position, as shown in Figure 5.4.
The HARDI data necessary to compute the metric tensor is retrieved by sampling the texture memory of the GPU, which contains data points defined at the cell corners $V_i$. The metric tensors $g_{\alpha\beta}(V_i, y)$ for the corner voxels are calculated using (3.8), with $y$ the direction of the fiber during this tracking step. With all eight neighboring metric tensors known, the value of $g_{\alpha\beta}$ at the current fiber position $x$ and current direction $y$ can be found by interpolating the eight values found at the cell corners using

$$g_{\alpha\beta}(x, y) = g_{\alpha\beta}(V_0, y)(1 - dx^1)(1 - dx^2)(1 - dx^3) + g_{\alpha\beta}(V_1, y)dx^1(1 - dx^2)(1 - dx^3) + g_{\alpha\beta}(V_2, y)(1 - dx^1)dx^2(1 - dx^3) + g_{\alpha\beta}(V_3, y)dx^1dx^2(1 - dx^3) + g_{\alpha\beta}(V_4, y)(1 - dx^1)(1 - dx^2)dx^3 + g_{\alpha\beta}(V_5, y)dx^1(1 - dx^2)dx^3 + g_{\alpha\beta}(V_6, y)(1 - dx^1)dx^2dx^3 + g_{\alpha\beta}(V_7, y)dx^1dx^2dx^3$$

where $dx^i$ is the distance between the current fiber position $x$ and the lower grid cell boundary along spatial direction $x^i$, as indicated in Figure 5.4. We see here on each line the contribution from one of the eight neighboring voxels to the interpolated total with a weight dependent on the position of the fiber within the grid cell.
The next steps involve computing the partial derivatives of the metric tensor in three directions and interpolating these derivatives in order to compute the Christoffel symbols $\Gamma^\alpha_{\beta\gamma}(x, y)$ (see (3.12)). In order to do this, the derivative is approximated using a central difference scheme. For example, the partial derivative in the first spatial direction is found using

$$\frac{\partial g_{\alpha\beta}}{\partial x^1}(x_1^i, x_j^2, x_k^3, y) \approx \frac{1}{2h^1} \left( g_{\alpha\beta}(x_1^{i+1}, x_j^2, x_k^3, y) - g_{\alpha\beta}(x_1^{i-1}, x_j^2, x_k^3, y) \right),$$

where $x_{ijk} = (x_1^i, x_j^2, x_k^3)$ indicates a grid point position $(i, j, k)$ and $h^i$ is the grid cell size in spatial direction $x^i$.

Computing the partial derivatives with a central difference scheme involves computing the Finsler metric tensor $g_{\alpha\beta}$ in the second order neighbors, one grid cell beyond the cell containing position $x$. As mentioned in Section 4.3, the number of registers used by a kernel is best kept as low as possible. Due to the fact that storing the metric tensors (consisting of six scalar values) of eight voxels in memory already requires 48 registers, storing the metric tensors for all (24) second order neighbors at the same time would likely increase running times considerably, or even exceed the register limit.

To combat this problem, the partial derivatives are computed one-by-one. The metric tensor for a single second order neighbor is computed, which allows the kernel to find a single partial derivative. This value is then added to a running total, by adding the weighted contribution of this partial derivative according to the weight found in the relevant line of the interpolation equation (5.1). This allows the kernel to compute interpolated partial derivatives of the metric tensor using a limited amount of registers, only requiring space for temporary storage of one metric tensor.

When the values of the metric tensor and its partial derivatives are known at the current fiber position, the Christoffel symbols are computed using (3.12). This then enables the kernel to compute the next position and direction of the fiber using an ODE solver. In our implementation, a modified fourth order Runge-Kutta method was used. In order to retain some of the smoothing benefits of the RK4 method without the added cost of fully solving (3.13) multiple times, we assume the Christoffel symbols are constant for the different steps of the RK4 algorithm.

The next position and direction of the fiber are then stored in an output buffer (large enough to accommodate eight steps’ worth of data), which is transferred to global memory when it is completely filled. This allows the kernel to continue to track a fiber for multiple steps before having to perform the costly step of writing to global memory. Finally, if any of the stopping conditions are met, the fiber tracking process of this kernel is stopped and the last fiber position and direction are written to global memory as a new seed point. These include the fiber leaving the data volume, two concurrent fiber segments being oriented at a very sharp angle, the fiber entering an area where the scalar measure for anisotropy (computed using (6.1)) is lower than some user-provided threshold, or exceeding the limit for the number of tracking steps per kernel launch. If none of the stopping criteria are met, the kernel repeats the tracking process.
5.3.2 Standard per neighbor

In the previous section, a basic fiber tracking kernel was introduced. That implementation involved sequentially computing Finsler metric tensors and derivatives for each of the eight voxels forming the cell that contains the fiber position in that step. As these are very similar operations executed sequentially, it is likely that a faster implementation is possible by performing these computations in parallel. To explore this opportunity, we describe an algorithm that uses a set of eight communicating threads. Each of these threads is responsible for computations regarding one of the eight neighboring voxels. This results in the grid configuration as shown in Figure 5.5.

![Figure 5.5: Grid layout for the standard per neighbor fiber-tracking kernel. A block of threads handles computation for multiple fibers. Each column of threads is associated with a single fiber. The eight threads used to track a fiber each compute metrics and partial derivatives at the position of one of the cell corner voxels.](image)

As shown in Figure 5.6, the start and end of a tracking cycle with this kernel are the same as for the standard kernel. The first of the eight threads (the ‘master’ thread) performs the same operations to retrieve a seed point, to compute the next fiber position and to decide whether or not to stop tracking the fiber. However, when computing the Finsler metric tensors, all eight threads are working in parallel to compute one metric tensor each. These metric tensors are then stored in the shared memory of the block. After that, the partial derivatives for each voxel can be computed with the central difference method by retrieving a metric tensor from the shared memory, and computing the metric tensor for the corresponding second order neighbor. With the Finsler metric tensor and partial derivatives known in each voxel, the values can be interpolated by adding them to a shared total with weights according to its corresponding line in (5.1).

As mentioned at the start of Section 5.3, running an algorithm on communicating threads requires additional considerations regarding synchronization and memory access. In order to ensure data is available before it is accessed, each block of threads is synchronized before steps that require other threads to store relevant information in shared memory. This is indicated in Figure 5.6 as lightning bolts. Another step where additional modifications are necessary is when the voxel threads add their weighted values of metric tensors and derivatives to the interpolated totals in shared memory. We implement a block-wide reduction, as illustrated in
Figure 5.6: Flowchart for the tracking kernel that computes metric tensors and derivatives for grid cell corners in parallel using additional threads per fiber. The dark blue box at the top represents the GPU device memory. The operations in the light blue box are executed for each of the eight threads in parallel, while the other operations are executed only by one of the eight threads.

Figure 5.7, where the list of tensors and derivatives is halved in size by adding pairs of values together in each step. As we only need to combine eight values together, we can be sure that all threads accessing the data associated with a fiber are located within the same warp. The fact that instructions are synchronized across a warp means this approach will not encounter race conditions, without having to resort to (costly) block-wide thread synchronization.

Figure 5.7: Illustration of the shared memory operations required for the reduction of the eight metric tensors and partial derivatives produced by the standard per neighbor kernel.

The beneficial results of added parallelization of metric tensor computations are shown in Chapter 8, where we see these improvements outweighing the costs of added thread synchronization.
5.3.3 Interpolated data

In the previous two kernel implementations, the Finsler metric tensor and its partial derivatives in three directions were computed by linear interpolation of the metric tensors found in the eight corners of the surrounding grid cell. Computing partial derivatives was done using a central difference method, which required more metric tensors to be computed for second order neighbors. This means that computing the metric tensors and derivatives in the fiber position requires a total of 32 metric tensors to be calculated. As computing one of these metrics is one of the most computationally expensive steps in the fiber tracking process, it could be beneficial to explore opportunities to reduce the amount of metric tensors needed.

The HARDI data used for the fiber tracking has been stored in texture memory, making use of the caching mechanism to reduce the cost of retrieving the tensor coefficients. The dedicated interpolation mechanism for sampling data at positions that are not grid-aligned can now be used to reduce the amount of metric tensors that are needed. Instead of interpolating the metric tensors at the position of the fiber, we modify the original geodesic fiber tracking algorithm to interpolate the HARDI tensor coefficients. As illustrated in Figure 5.8 for a derivative in the horizontal direction, the HARDI data is sampled at the position of the fiber and in each neighboring cell, at a distance of one cell width from the fiber position. The Finsler metric tensors are computed in these positions. This means the central difference method from (5.2) can be used to compute the partial derivatives in the position of the fiber directly. In total, only seven metric tensors need to be calculated, rather than 32. Of course, this approach will give slightly different results. The error caused by interpolating the metrics for each neighboring voxel will be different from the error observed when interpolating the HARDI data before computation of metrics. The metrics obtained by these two techniques should be compared to see whether one yields more accurate results than the other, regardless of performance gains.

![Figure 5.8: An illustration of the positions where metric tensors are computed when computing the partial derivative in the horizontal direction. The HARDI data is sampled at positions one grid cell width on either side of the fiber position x.](image)
Figure 5.9: Flowchart for the ray-tracing kernel based on interpolated HARDI data. The dark blue box at the top represents the GPU device memory.

Figure 5.10: Grid layout for the interpolated data fiber-tracking kernel. A block of threads handles computation for multiple fibers. Each thread tracks a single fiber.

5.3.4 Interpolated data per direction

The previous section described a fiber tracking kernel that interpolated HARDI data before computing Finsler metric tensors, in order to reduce the computation workload and better utilize dedicated interpolation architecture in the GPU. Similar to our previous description of a kernel that used additional threads for one fiber, we can find symmetry in the computations involved in fiber tracking. In this case, the computations of partial derivatives along three axes are nearly identical and largely independent of each other. We can therefore use the same approach as before, now using three threads per fiber. This results in the grid structure as pictured in Figure 5.12.

First, the central inverse metric tensor is computed by one thread. Next, for each axis, the thread responsible for this direction computes the Finsler metric tensors in the two cells neighboring the fiber position along the axis. With these tensors, the thread then computes the partial derivative in this direction using (5.2) and stores it in the shared memory. When all three partial derivatives have been computed, the first thread can compute the Christoffel symbols and perform the integration step to find the next position and direction of the fiber.
The Christoffel symbol computations and integration steps show an opportunity to be split across three threads as well. The 18 Christoffel symbols could be distributed in three groups of six, whereas each thread could compute one coordinate of the next position and direction in the integration steps. However, no further parallelization is implemented with regards to computing Christoffel symbols or performing the integration, as this was not expected to yield noticeable speed improvements. As mentioned in Section 4.3, an important part of optimizing a CUDA program is ensuring that threads do not exhibit branching behavior. While the metric derivative computation can be easily separated for each axis, only requiring one lookup of a direction vector, the steps involved in computing Christoffel symbols and integration are less symmetric. The resulting code would require more hard-coded lookups to prevent branching behavior, becoming less readable in the process. Furthermore, the workload involved in these computations is much smaller than that of computing the Finsler metric tensors, leading to a low upper bound on the possible improvement.

Figure 5.11: Flowchart for the fiber-tracking kernel based on interpolated HARDI data and computing derivatives in three directions in parallel. The dark blue box at the top represents the GPU device memory. The operations in the light blue box are executed for each of the three threads in parallel, while the other operations are executed only by one of the three threads.

Figure 5.12: Grid layout for the interpolated data per direction fiber-tracking kernel. A block of threads handles computation for multiple fibers. Each column of threads is associated with a single fiber. The three threads used to track a fiber each compute partial derivatives of the metric tensor in one of three spatial directions.
Chapter 6

Refraction-Inspired Correction

A downside of the geodesic fiber-tracking technique is the occurrence of false positives in the set of output fibers. The fiber-tracking algorithm can estimate the existence of fibers in directions or positions where they are unlikely to exist. While methods exist to filter these false positives out from the final result, such as comparing the diffusive strength along geodesics connecting two regions [10], a method of correcting this behavior beforehand would be preferred. This chapter introduces a correction method that adjusts the direction of fibers during the tracking stage to improve the overall quality of the output fibers.

6.1 Motivation

One clear situation where false positives occur in the output of the fiber-tracking algorithm is when fiber paths lead into near-isotropic parts of the domain. Here, where the Orientation Distribution Function (ODF) is nearly spherical, water molecules exhibit near-identical diffusion in all directions, which means the probability of any physical fibers existing in such a location is very low. When using a streamline fiber-tracking technique, a computed fiber will follow a path with sharp angles, as noise in the data becomes a significant factor in determining a local maximum of the ODF. Filtering out fibers with sharp angles from the final result is a simple method to remove these false positives when using streamline tracking. As illustrated in Figure 6.1, this method would not suffice for geodesic fiber tracking using a Finsler metric. Because of the dependency on incoming direction, a fiber moving through an area with low anisotropy will mostly retain its direction. While this retention of direction is highly beneficial when dealing with crossing fiber bundles, it does not serve a clear purpose in near-isotropic parts of the domain. The resulting fibers appear to be part of a consistent bundle, becoming more difficult to identify as false positives.
We aim to reduce the number of false positives in the output of the geodesic fiber-tracking technique. One approach is to apply a different type of filtering, where fiber tracking is halted whenever the Generalized Fractional Anisotropy, a measure for the anisotropy of the diffusion at a given position, is below a certain threshold at the tip of the fiber. In the case where the diffusion profile is represented as a fourth order tensor, the GFA can be computed from the 15 tensor coefficients $c_i$ using

$$GFA = \sqrt{1 - \frac{(c_0)^2}{\sum_{i=0}^{14} (c_i)^2}}. \quad (6.1)$$

Aggressively filtering fibers based on this value will prevent the problem shown in Figure 6.1 at the cost of breaking off computation on a high number of fibers. We aim to introduce a smooth corrective adjustment to the fiber direction, to keep fibers within fiber bundles rather than stopping the tracking procedure at the boundary of such a bundle.

![Figure 6.1](image)

Figure 6.1: Illustration of fiber tracking through near-isotropic areas, shown as blue shapes representing near-circular Orientation Distribution Functions, when using (a) streamline tracking or (b) geodesic fiber tracking using Finsler metrics. The direction-dependent nature of Finsler metrics makes the fibers less sensitive to small deviations in ODF shape.

### 6.2 Description

Our correction method is inspired by the physical concept of refraction. Refraction is a phenomenon where a wave undergoes a change in propagation direction due to a change in the refractive index of the transmission medium. When a wave hits a surface between two media at an angle, the change in phase velocity affects the direction of the wave. The most common observation of refraction is the effect of light rays changing direction when traveling across a boundary between air and water or glass. The amount of change in the direction is dependent on the ratio of refractive indices $n_i$ in the media on both sides of the incident plane, which is also equal to the inverse ratio of phase velocities $v_i$, as represented in Snell’s law:

$$\frac{\sin \theta_1}{\sin \theta_2} = \frac{n_2}{n_1} = \frac{v_1}{v_2}, \quad (6.2)$$

where $\theta_1$ and $\theta_2$ are the angle with the surface normal of the incoming and refracted ray.
This physical concept is the inspiration for our correction technique. We can use the adjustment of a fiber direction due to a change in HARDI data as an analogue for the change in propagation direction of a wave due to a change in refractive index. Figure 6.2 shows an illustration of one correction step. At a given fiber position \( p \), the current direction vector \( \overrightarrow{p_0p_1} \) is known. A virtual interface plane is constructed between two propagation media along \( r_{\text{max}} \), the local maximum of the ODF in \( p \) that is nearest to the current direction vector. Using vector \( \overrightarrow{p_0p} \) as the incoming ray, we can compute refraction based on some material properties \( n_1 \) and \( n_2 \). The refracted ray \( \overrightarrow{pp_1'} \) can then be used as a new direction at fiber position \( p \).

Figure 6.2: Illustration of refraction-inspired adjustment of a fiber at fiber position \( \mathbf{x} \) for \( n_1 > n_2 \), based on the ODF (indicated by the peanut-shaped curve) interpolated at \( \mathbf{x} \). The imaginary plane of refraction (the dashed line between the blue and yellow area) is aligned with the nearest radial vector in the direction of the local maximum of the ODF. The direction \( \mathbf{y} \) of the fiber at the current computation step is used as the incoming ray \( \overrightarrow{p_0x} \). The refracted ray vector \( \overrightarrow{xp_1'} \) forms the ‘corrected’ direction for this fiber. The lengths of the ‘radial vectors’ \( \mathbf{r} \) and \( r_{\text{max}} \) are the size of the ODF in their respective directions.

Using a refraction-inspired correction offers some useful characteristics. In our case, the interface surface is never a sharp jump in refractive index, but rather a relatively smooth transition between two values (due to linear interpolation of values between grid points). This ensures that fibers will not be subjected to sharp correctional rotations, instead being adjusted slightly during multiple steps. Furthermore, refraction includes the concept of a ‘critical angle’, where the angle of the incoming wave with the surface normal is so large that a refracted ray would not travel into the second medium at all. The ray is then completely reflected instead. This
occurs at an angle $\theta_1 \geq \theta_c$ when the outgoing direction is along the surface, i.e. $\theta_2 = \frac{\pi}{2}$, such that

$$\sin \theta_c = \frac{n_2}{n_1}. \quad (6.3)$$

This critical angle is therefore dependent on the change in refractive index. This offers a natural threshold for the geodesic fiber correction. Any fiber where the angle $\theta_1$ is at an angle larger than $\theta_c$ is considered to be an accurate representation of a physical fiber, as its local direction nearly coincides with the direction of a local maximum of the ODF, and therefore does not have its direction changed. Using the same material properties to generate a threshold for correction eliminates the need for another ad-hoc decision on a threshold value, instead linking it to the same input used before. This allows for easier tuning of the correction strength for different tracking configurations, as only one variable can change at a time.

We have now defined a use of Snell’s law to enforce a change in fiber direction based on changing properties of the medium. What remains is the choice of material property to base the refraction indices on in the context of geodesic fiber tracking. We consider two options, the ODF radii and GFA values.

When using ODF radii to construct a refraction index, $n_1$ is set to the size $||r_{max}||$ of the nearest local maximum on the ODF and $n_2$ is set as the ODF value $||r||$ in the current fiber direction. As $n_2$ will always be smaller than $n_1$ (so long as the fiber is not moving exactly along the local maximum vector), the correction will direct fibers towards the local maximum direction. The results of redirecting fibers in the direction of local maxima on the ODF will become more similar to the streamline tracking results as the correction strength is increased.

The second approach is to use the fractional anisotropy as the refraction index at a given position. Rather than attempt to adjust the results of the fiber tracking to be more similar to the results of a streamline algorithm, this technique aims to directly achieve the desired result of keeping fibers within bundles through highly anisotropic parts of the domain. If we consider the refractive indices $n_1$ and $n_2$ to be GFA values in the two media of Figure 6.2, the refraction shown corresponds with a fiber moving across a boundary between a high- and low-anisotropy medium. The fiber is rotated towards the nearest local maximum of the ODF, redirecting it to move closer to parallel with the surface. If this boundary is replaced by a smooth transition from an anisotropic to an isotropic medium, this fiber would be gradually rotated to move along the fiber bundle that would be found in the anisotropic medium.

In order to find representative values for the fractional anisotropy in the two virtual media involved in the refraction, we use the value at estimated positions during the previous and next tracking steps, as illustrated in Figure 6.2. The first value $n_1$ is chosen as the square of the fractional anisotropy at a position one Euler integration step backwards from the current position $x$ with respect to the current direction $y$, such that $n_1 = \text{GFA}^2(p_0)$. The value of $n_2$ is found one Euler integration step forwards from $x$, leading to $n_2 = \text{GFA}^2(p_1)$. The values are squared to increase the effects of the correction, as the differences in GFA are relatively small when measured on the scale of integration step lengths.

In our implementation, the choice has been made to use the GFA values as a source of refraction indices. The approach based on ODF radii may be used to remove false positives, but it may also reintroduce the problems that streamline-based tractography has. Using GFA values as refraction indices should help contain the computed fibers to high-anisotropy bundles, without interfering with the crossing and branching structures found within bundles.
6.3 Implementation

The implementation of the refraction-inspired correction technique consists of two major phases.
As described in the previous section, a virtual interface plane is used to perform the refraction.
The first phase of the correction method therefore involves the search for all local maxima on the
ODF. In the second phase, the nearest of these maxima is used to perform the actual correction
of a fiber traversing a given position in a given direction, based on fractional anisotropy as a
measure that replaces the refraction indices. These two phases are described in more detail in
Algorithms 2 and 3, describing the correction process for a fiber at position $x$ and current local
direction $y$.

Algorithm 2 Refraction-inspired correction

1: function RefractionCorrection($x, y, iterationLimit, stepsize$)
2:   maxima $\leftarrow$ FindLocalMaxima($x$)  \Comment{Returns list of vectors of unit length}
3:   $m$ $\leftarrow$ FindNearestMaximum(maxima, $y$)  \Comment{Returns vector with smallest angle to $y$}
4:   $i$ $\leftarrow$ 0
5:   for $i < iterationLimit$ do
6:     nextGFA $\leftarrow$ GFA($x + stepsize \cdot y$)\Comment{Computed using equation (6.1)}
7:     prevGFA $\leftarrow$ GFA($x - stepsize \cdot y$)
8:     ratio $\leftarrow$ ($nextGFA / prevGFA$)$^2$
9:     $\theta \leftarrow \cos^{-1}(y \cdot m)$\Comment{Note: $|y| = |m| = 1$}
10:    if $\theta < \sin^{-1}(n_2/n_1)$ then\Comment{Angle is smaller than critical angle}
11:       $\Delta_\theta = \theta - \sin^{-1}(ratio \cdot \theta)$
12:       Rotate $y$ by an amount $\Delta_\theta$ towards $m$ around vector perpendicular to $y$ and $m$
13:    $i$ $\leftarrow$ $i + 1$

The $iterationLimit$ input variable in Algorithm 2 is used in the CUDA-based implementation
to compensate for a lower correction frequency. As noted in Chapter 5, the GPU-accelerated
variant of the geodesic fiber-tracking algorithm applies a multiple of eight tracking steps at a
time. While this reduces the delays in computation due to memory limitations, it also limits the
effectiveness of the refraction-inspired correction. The correction is only applied once for every
eight or more tracking steps. The correction can therefore be applied multiple times to increase
the effects without having to adjust the equations used to compute the simulated refractive
indices. In each iteration, the adjusted new fiber direction is used to recompute expected values
for the GFA. The adjustment in direction should remain relatively small for each iteration, such
that the nearest local maximum remains the same. This way the fiber direction can be adjusted
multiple times at a single position, without having to perform the costly computations involved
in finding the nearest local maximum multiple times.
Algorithm 3 Local maxima extraction

1: function FindLocalMaxima(x) \(\triangleright\) Returns local maxima of ODF at position \(x\)
2: \(U \leftarrow\) Directions \(u_i\) towards vertices on tessellated sphere
3: \(ODF \leftarrow\) Interpolate ODF at position \(x\)
4: for all vertices \(u_i\) on tessellated sphere do \(\triangleright\) Parallel threads when using CUDA
5: \(v_i = ODF(u_i)\) \(\triangleright\) In shared memory when using CUDA
6: for all tessellation triangles do
7: if the triangle contains \(u_i\) then
8: if \(v_i\) is not the largest value in the triangle then
9: \(U \leftarrow U - \{u_i\}\) \(\triangleright\) Remove this vertex from output
10: return \(U\) \(\triangleright\) Vectors remaining in \(U\) are local maxima vectors

Computing the set of local maxima of the ODF is done by sampling the value of the ODF in the direction of vertices positioned on the surface of a tessellated unit sphere. Such a sphere, as pictured in Figure 6.3, consists of a set of triangles. We first find the value of the ODF at all vertices on this sphere. In order to find a local maximum, we need to find the vertices on the tessellated sphere where the ODF value is higher than the neighboring values. For each vertex, we therefore need to iterate over all triangles containing this vertex, as illustrated in Figure 6.4. If the ODF values at all neighbors of a vertex are lower than the value at this vertex, it is a local maximum.

Figure 6.3: An illustration of the type of tessellated sphere used for sampling ODF values to find local maxima. In our implementation, this sphere consists of 164 vertices.

This correction method is implemented using two CUDA kernels, one for each of the two described algorithms. The maxima-extraction algorithm, as described in Algorithm 3, is executed using one block of threads for each fiber, where the number of threads is equal to the number of vertices on the tessellated sphere used to sample ODF values. This means that the value in the direction of each vertex is computed in parallel and stored in the shared memory of the block. These values are then used in the next step, where all threads traverse the set of triangles in parallel. While computing the ODF values for each vertex in parallel means this part of the algorithm is relatively fast, the maxima-extraction kernel is the most computationally expensive of the two. This is largely due to the branching behavior that occurs when traversing through the list of triangles. Only vertices that are contained in a triangle perform computations on it. The threads associated with vertices that are not included in the triangle remain idle. This part of the algorithm could possibly be replaced by a more sophisticated computation of local maxima involving a dimensional reduction in the extraction process [26].
Figure 6.4: An illustration of a part of the process of finding local maxima of the ODF. After finding the value of the ODF in the direction towards each of the vertices on the tessellated sphere, we check whether the ODF value at a vertex is higher than that of all its neighbors. Here, the value of the ODF in the central direction $\mathbf{u}_i$ is compared to the values found at the other vertices in the blue triangle, before comparing it to the values at the other pictured triangles.

The second correction kernel computes the corrected direction based on the fiber position, direction and the output of the maxima-extraction kernel, as described in Algorithm 2. This kernel is executed in blocks of 512 threads where one thread applies the correction for one fiber. Each thread retrieves the current fiber position and direction, as well as the list of local maxima for this fiber as produced by the previous kernel. It then executes the iterative correction as described in Algorithm 2 and writes the corrected direction as the new starting direction of the fiber at this location.
Chapter 7

Ricci Scalars

In this chapter, we introduce a novel way to represent fibers in a lower dimensional space. The output from our fiber tracking algorithm is a sequence of 3D positions for each fiber. Since we are aiming to track a large number of fibers, loading and processing of this output data can become costly enough to be a noticeable detriment to performance. Furthermore, comparison of fibers is not entirely straightforward, as any similarity measure between 3D curves can only be fair if the underlying DW-MRI data is completely aligned between scans.

In order to reduce the impact of these issues, fibers can be represented in a lower dimension through some measure other than 3D position. For example, mapping the curvature as a function of the distance traveled along a fiber reduces the data from four to two dimensions per fiber [27], as a list of 3D fiber points is replaced by a list of scalar values. In this research, a similar approach is used. Rather than representing fibers as a sequence of 3D positions, the Ricci scalar is computed along each fiber. This Ricci scalar is a curvature measure defined for Riemannian and Finsler manifolds, allowing us to reduce the dimensionality of our output data to a single scalar value as a function of the distance traveled along the fiber. These 2D curves can then be used for further processing, allowing for clearer two-dimensional visualization or smaller input size for other applications such as machine learning.
7.1 Introduction to Ricci scalars

The Ricci scalar is a measure in Riemannian geometry that describes a curvature of a manifold. It measures the change in volume of a ball on the manifold compared to a Euclidean ball with the same radius. The distorted shape of the ball in a Riemannian geometry results in a change of this volume, leading to positive or negative non-zero values of the Ricci scalar. In DTI, using Riemannian geometry, the Ricci scalar is computed via a sum of Riemannian curvatures. Here, a negative value of Ricci scalar implies a structure of crossing fiber bundles. In Finsler geometry, where metrics also depend on a direction rather than just a position in space, the Ricci scalar for HARDI data $\text{Ric}(x,y)$ [12] at a position $x$ with regards to a direction $y$ is defined as

$$R(x,y) = R^i_i(x,y), \quad (7.1)$$

where

$$R^i_i(x,y) = 2 \frac{\partial G^i(x,y)}{\partial x^k} - y^j \frac{\partial^2 G^i(x,y)}{\partial x^j \partial y^k} + 2G^i(x,y) \frac{\partial G^i(x,y)}{\partial y^j} \frac{\partial G^i(x,y)}{\partial y^k}. \quad (7.2)$$

The geodesic coefficient $G^i$ in the previous equation is defined as

$$G^i(x,y) = \frac{1}{4} \gamma^{jl}(x,y)(2 \frac{\partial g_{jl}(x,y)}{\partial x^k} - \frac{\partial g_{jk}(x,y)}{\partial x^l})y^j y^k, \quad (7.3)$$

where $\gamma_{\alpha\beta}(x,y)$ is the Finsler metric tensor computed from the HARDI data the same way as in the fiber-tracking process, using (3.8).

From these equations, we see the dependency of the Ricci scalar on both the input direction $y$ and the Finsler metric tensor $\gamma_{\alpha\beta}(x,y)$. While it is possible to compute the Ricci scalar for any direction, this work focuses on computing Ricci scalars with this vector $y$ chosen in the tangential direction along the geodesics. By doing this, we compute a curvature measure that contains both information regarding the direction of an individual fiber as well as the underlying geometry of the HARDI data.

7.2 Implementation

From the equations in the previous section, it becomes clear that Ricci scalars are computationally expensive. The computation of each value requires the partial derivatives of $G^i(x,y)$ in each direction, as well as second derivatives in each direction. In total, this means the Ricci scalar can only be computed by finding all values of $G^i(x,y)$ in a neighborhood of 25 cells. The total number of Finsler metric tensors required for this is 63, which indicates the complexity of these computations when compared to the 32 metrics needed for a step of the geodesic fiber-tracking algorithm. Due to these large costs involved in computing Ricci scalars for all fibers, their values are only computed once after each tracking cycle. A fiber is tracked for at least eight steps before another run of the Ricci kernel is started. The values between measurements is filled using linear interpolation between the two known values.
Ricci scalars are computed using 25 threads for each fiber, which perform four steps of computation to find the Ricci scalar value at the current position of a fiber (also illustrated in Figure 7.1):

- First, all 25 threads each compute one of the geodesic coefficients $G$ in the neighborhood similarly to way the kernel described in Section 5.3.3 computes Christoffel symbols, interpolating HARDI data and computing the Finsler metric in neighboring cells and using those values to compute metric derivatives needed to find $G$. The values of $G$ in all 25 cells are then stored in shared memory.

- In the next step, four threads remain inactive while the remaining 21 use the data in shared memory to compute all 21 derivatives $\frac{\partial G}{\partial x^i}$ resulting from computing three partial derivatives in seven cells. These derivatives are then also stored in shared memory.

- The partial derivatives of $G$ in shared memory are then used by nine out of the 25 threads to compute all second derivatives at the position of the fiber.

- Finally, one thread uses all values available in shared memory to compute the Ricci scalar at this position, storing it in the global memory to be used externally in visualization or further processing.

![Diagram of computation steps](image)

Figure 7.1: The four steps involved in computing the Ricci scalar for a fiber. Each step stores its result in shared memory and uses the result of previous steps for its computation.

As mentioned, computing Ricci scalars in our implementation requires 25 threads per fiber. Only during the first step, computing the geodesic coefficients $G$, all threads are active. Up to 24 threads are inactive during the later steps. However, basic measurements indicate that the computation of geodesic coefficients requires nearly 1000 times more time than computing the partial derivatives and Ricci scalar sum. This means that any inactive thread is still utilized for near 99.9% of the total running time. Splitting the different computation steps into different kernels would therefore likely offer no noticeable improvement. Furthermore, the additional memory transfers required to share the geodesic coefficients via global memory may even negatively affect performance.
Chapter 8

Results

In this chapter, the performance of the geodesic fiber-tracking algorithm is discussed in both a CPU and a GPU context. First, the single-threaded CPU-based implementation is used to demonstrate the benefits of this method of tractography. This will also highlight the downsides of this method such as long running times and a tendency to generate false positives. In the next section, the performance of the GPU-accelerated CUDA implementation is examined, showing that it is able to reduce the running time of the fiber-tracking algorithm. The third section contains a demonstration of the refraction-inspired correction technique that reduces the number of false positives in the output by means of a soft constraint keeping fibers within bundles through regions with high anisotropy. The final section presents the preliminary results on Ricci scalars, showing that this measure can be used as a lower-dimensional fiber bundle comparison technique.

8.1 CPU implementation results

First, we examine whether our CPU implementation of the geodesic fiber-tracking algorithm exhibits the behavior expected of the geodesic approach. To this end, a bundle of corpus callosum fibers has been tracked using an Intel i7-3770 CPU at 3.90 GHz. The results of this experiment are shown in Figure 8.1. In the top-left of this image, the fibers in this bundle start branching off, as is expected from the geodesic fiber-tracking algorithm. Where a deterministic streamline algorithm would only reveal the main arc of the corpus callosum, the branching behavior of the geodesic approach reveals the multi-valued solution including connections to other parts of the brain.
Figure 8.1: A bundle of corpus callosum fibers tracked using the geodesic fiber-tracking algorithm, in a volume of 31x13x9 voxels from part of the centrum semiovale. The fibers were filtered through a region at the right-hand side of the corpus callosum to produce a clear image with a reduced number of false positives.

In order to show the effectiveness of geodesic fiber-tracking in areas containing crossing fibers, a set of fibers in the cingulum region of the brain has been computed in addition to the previous bundle. The results of this experiment are shown in Figure 8.2. We see a subset of these bundles where fibers continue through the overlapping region uninhibited, showing that our geodesic fiber-tracking implementation can deal effectively with crossing fiber structures.

Figure 8.2: A bundle of corpus callosum fibers in purple and a bundle of cingulum fibers in green partially crossing through the bundle.
The geodesic fiber-tracking algorithm generally produces a fairly large number of false positives. In order to counteract this, fibers are often selected such that all rendered fibers also pass through a second filtering region. This technique was applied to obtain the fibers shown in Figure 8.1. In order to ensure enough fibers are present in the filtered output set, a larger number of fibers must be computed. However, the geodesic fiber-tracking algorithm is computationally very expensive. This means that it becomes impractical to track a large enough number of fibers to obtain a clear image. As an indication, Figure 8.6 shows that computing more than 1500 fibers using the CPU can take a few minutes, even in this relatively small volume where fibers are cut off at the boundaries. The GPU-accelerated algorithm is therefore highly useful to allow users to use this geodesic fiber-tracking algorithm practically, without having to wait for long periods of time for a set of fibers to be generated after a small change in input parameters.

8.2 GPU-related results

In the following section, the performance of the GPU-accelerated implementation of the geodesic fiber-tracking algorithm is discussed. In the following experiments, the GPU used in testing is the GeForce GTX 780 Ti, a relatively high-end desktop GPU released in late 2013, with additional testing on a relatively low-end GeForce GTX 840M notebook GPU released in early 2014. The most important specifications of these two GPUs are listed in Table 8.1.

<table>
<thead>
<tr>
<th></th>
<th>GeForce GTX 780 Ti</th>
<th>GeForce GTX 840M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprocessors</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>CUDA cores</td>
<td>2880</td>
<td>384</td>
</tr>
<tr>
<td>GPU clock rate</td>
<td>928 MHz</td>
<td>1124 MHz</td>
</tr>
<tr>
<td>Global memory</td>
<td>3072 MB</td>
<td>2048 MB</td>
</tr>
<tr>
<td>Memory clock rate</td>
<td>3500 MHz</td>
<td>900 MHz</td>
</tr>
<tr>
<td>Memory bus width</td>
<td>384-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>12 Mb</td>
<td>8 Mb</td>
</tr>
</tbody>
</table>

Table 8.1: Specifications for the two GPUs used in the GPU-related experiments.
### Table 8.2: Comparison of total running times for tracking a set of fibers, with the seed points either in the order that they are generated or in a randomized order.

<table>
<thead>
<tr>
<th>Kernel type</th>
<th>Running time (s) in regular order</th>
<th>Running time (s) in random order</th>
<th>Running time reduction in regular order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>97.07</td>
<td>106.13</td>
<td>9.07</td>
</tr>
<tr>
<td>Standard per neighbor</td>
<td>45.17</td>
<td>54.55</td>
<td>9.37</td>
</tr>
<tr>
<td>Interpolated</td>
<td>14.54</td>
<td>15.46</td>
<td>0.92</td>
</tr>
<tr>
<td>Interpolated per direction</td>
<td>13.70</td>
<td>15.54</td>
<td>1.85</td>
</tr>
</tbody>
</table>

#### 8.2.1 Texture caching

First, the effect of the GPU texture cache on the fiber-tracking process is shown. As mentioned in Chapter 4, this cache is designed to benefit from 2D spatial locality. Texture memory was used to store the HARDI data based on the assumption that nearby threads would be tracking fibers that are positioned close together. This assumption was based on the expectation that some fibers from nearby seed points would remain relatively close throughout the tracking phase. Furthermore, if the user chooses to utilize the option to send out fibers in a cone around each local maximum, these fibers could stay close together for a long distance.

In order to test this assumption of spatial coherence, each kernel variant was used to track a set of fibers. First, the fiber-tracking algorithm was executed as before. Seed points were generated for a region of interest, with one fiber per local maximum on the ODF and sixteen additional outgoing fibers in a tight cone around it. The resulting 5236 seed points were used as input in the order that they were generated. In the second run, these seed points were put in a randomized order. A comparison between the running times of these two runs is shown in Table 8.2.

From the results in Table 8.2, it appears that at least some spatial coherence can be found in our input. If the input is shuffled into a randomized order, the performance deteriorates by up to 17%. Fibers originating from nearby seed points, or seed points with the same position but slightly different fiber starting direction, seem to remain close enough together to allow for some data caching for texture sampling. Furthermore, we see an increased performance gain for the kernel variants that utilize multiple threads per fiber. This can be explained by the fact that these kernels already guarantee some spatial locality between threads. The eight threads for a standard per neighbor kernel will always sample the texture memory within an area of 2x2x2 voxels. The interpolated data per direction kernel uses three threads that each sample data from positions at a distance of one grid cell in each direction. The two kernels that utilize multiple threads per fiber therefore benefit more from texture caching than the kernels that do not. This indicates that increasing parallelization is not only useful to improve execution of floating point operations, but can also contribute to more efficient memory access.
8.2.2 Number of tracking steps

As described in Section 5.3, the tracking kernels all perform multiple tracking steps in sequence before writing the resulting fiber positions to global memory in blocks of eight at a time. Writing to global memory in batches coalesces the memory accesses, reducing the effects of latency incurred from using global memory. In order to examine whether increasing the size of these batches would benefit performance, the time needed to track a set of approximately 34,000 fibers has been measured for different amounts of tracking steps per kernel run. The results of this experiment are shown in Figure 8.3. Here we can see that the running time does not decrease when performing more tracking cycles per kernel execution. The possible benefits of writing to memory in larger batches is outweighed by the negative effects of having a lower refill rate of CUDA threads. When a kernel finishes tracking a fiber, a larger number of cycles per kernel run means the thread will stay inactive for more cycles until it loads a new seed point.

![Figure 8.3: Measurements of the total running time of tracking about 34,000 fibers using the interpolated per direction kernel, for different amounts of tracking steps per execution of the tracking kernel. The dotted line indicates the running time for eight steps per kernel run.](image)

The fact that threads stay inactive for longer periods of time when using larger batches can also be seen from the number of kernel executions. The number of executions required to complete the tracking of all fibers is shown in Figure 8.4. From this graph, we can see that if the amount of tracking steps per kernel run is doubled, the total number of kernel executions remains just over half the previous value. CUDA threads remaining inactive for more tracking steps means that we require more kernel executions than expected to complete the tracking of all fibers. This increases the running time enough to outweigh the possible benefits from the more efficient memory accessing.
Figure 8.4: Measurements of the amount of kernel runs needed to complete the tracking of approximately 34,000 fibers using the \textit{interpolated per direction} kernel, for different amounts of tracking steps per execution of the tracking kernel. Also indicated is the number of kernel launches that would be expected given the number of launches needed at eight tracking steps per run.

\subsection{8.2.3 Occupancy}

Now, we aim to find an optimal configuration for the CUDA launch parameters. As mentioned in Section 4.3, the occupancy can be used as an indication of GPU utilization. We therefore measure the occupancy and average running time per kernel execution for all four variants described in Section 5.3, for different block sizes. The results of these measurements can be found in Figure 8.5.

For the \textit{standard} kernel, we see that the maximum occupancy found is 0.5, found at 64, 128 or 256 fibers per block. From the CUDA Occupancy Calculator\footnote{developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls} we can find that this maximum value is equal to the theoretical limit of 50\% occupancy for the 61 registers required to run this kernel. As the running time was found to be lower for 128 fibers per block, this value was chosen for the configuration of this kernel used in all other experiments.

The \textit{standard per neighbor} kernel uses eight threads per fiber to compute metric tensors and partial derivatives for each cell corner in parallel. Communicating values between these threads requires a large amount of shared memory per fiber. This shows in the occupancy measurements, as the maximum value here is approximately 0.4, whereas other kernels approach an occupancy of 0.5 for some configurations. The large amount of shared memory required means that it is impossible to assign 128 or more fibers to a single block, as the total amount of memory required would exceed the 48KB limit on shared memory available per block.
As the interpolated data kernel uses a similar setup to the standard kernel of one CUDA thread per fiber, we see a similar pattern in the occupancy. Again, we find the maximum occupancy of 0.5 (equal to the theoretical limit) and the minimum running time at 128 fibers per block.

The occupancy and running time of the interpolated per direction kernel start to deteriorate for more than 96 fibers per block. As mentioned before, this kernel uses three threads for each fiber that share information via the shared memory. As this kernel requires three times as many threads as fibers per block as the the interpolated data kernel, this kernel is affected by the limited number of registers per block when more than 96 fibers per block, whereas the interpolated data kernel is not.

The runtime configuration as used in the next experiments has been based on these measurements, and has lead to the configuration as shown in Table 8.3.

<table>
<thead>
<tr>
<th>Tracking kernel implementation</th>
<th>Number of fibers per block</th>
<th>Block dimensions $(\text{dim}_x, \text{dim}_y)$</th>
<th>Shared memory per block (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>128</td>
<td>$(128, 1)$</td>
<td>0</td>
</tr>
<tr>
<td>Standard per neighbor</td>
<td>96</td>
<td>$(96, 8)$</td>
<td>38016</td>
</tr>
<tr>
<td>Interpolated data</td>
<td>128</td>
<td>$(128, 1)$</td>
<td>0</td>
</tr>
<tr>
<td>Interpolated data per direction</td>
<td>96</td>
<td>$(96, 3)$</td>
<td>10368</td>
</tr>
</tbody>
</table>

Table 8.3: CUDA runtime configuration for the four tracking kernel variants with regards to block dimensions and shared memory allocation.
Figure 8.5: Measurements of occupancy and average running times of a single kernel execution for different choices of the amount of fibers assigned to one CUDA block. The dotted line for the standard per neighbor kernel indicates the maximum number of fibers that could be assigned to a block due to the limited amount of shared memory available per block.
8.3 CUDA implementation comparisons

Now, we compare the performance of the CUDA kernel variants introduced in Chapter 5 to each other and the single-thread CPU implementation. The focus is the total running time of the algorithm, as this is the largest downside of classic geodesic fiber tracking as well as the most noticeable performance metric from a user’s perspective. As mentioned in the previous section, the runtime configuration with respect to block dimensions is shown in Table 8.3.

First, we examine the running times of all previously mentioned implementations for different numbers of outgoing fibers from a single region in the corpus callosum. The number of outgoing fibers is controlled by varying the number of additional directions sent out in a cone around each local maximum on the diffusion ODF. Figure 8.6 shows the time in seconds required for each implementation to finish the tracking algorithm for fiber counts up to 2500.

![Figure 8.6: Running times for each implementation for up to 2500 outgoing fibers.](image)

It can be easily observed from Figure 8.6 that the single-threaded CPU implementation does not scale as efficiently as GPU implementations with regards to the number of tracked fibers. The Standard kernel has a lower total running time when tracking over 300 fibers, while the other kernels perform better for anything over 100 fibers. We now focus on the detailed performance of the GPU kernels. We therefore repeat the previous experiment using all GPU kernel implementations for a larger number of fibers. The results of this experiment are shown in Figure 8.7.

As seen in Figure 8.7, the different implementations all show a near-linear increase in running time with increasing fiber count, provided the number of fibers tracked is larger than 3000. Below this value, the running time for the standard kernel decreases less with a decreasing number of fibers. This likely occurs due to a lack of occupancy of CUDA cores. The GTX 780 Ti used for this experiment contains 15 multiprocessors and 2880 CUDA cores. In order to have enough seed points available to occupy the threads running on these CUDA cores, threads were created for 4096 fibers at a time. This ensures that fibers which terminate early can be replaced by new seed points to keep CUDA cores occupied, provided new seed points are available. In the case where only a small number of fibers is tracked, too few seed points are available to keep the cores occupied. This will lead to a part of the CUDA cores remaining idle, which does not lead to any improvement to the running time at this low fiber count. The SIMT architecture
Figure 8.7: Running times for all four CUDA kernel implementations for up to 27000 outgoing fibers.

described in Section 4.1 means that executing the algorithm on a single CUDA core takes the same amount of time as executing the same algorithm on all CUDA cores at the same time.

The standard per neighbor kernel appears to suffer less problems when tracking a smaller number of fibers. This further indicates that a lack of CUDA core occupancy is a limiting factor for the standard kernel, as the standard per neighbor kernel uses eight threads per fiber. It can therefore saturate all available threads for an input size that is eight times smaller. The result is that the running times for this kernel show near-linear behavior for almost the full range of input sizes.

From Figure 8.7, we can see that the use of interpolated HARDI data improves performance over the method that first computes Finsler metric tensors and derivatives before interpolating those. As expected, the reduction in the number of metrics that need to be computed from 32 to 7 leads to a large reduction in running time. The Standard kernel requires close to five times as much time as the Interpolated data kernel to complete the fiber tracking process. From the reduction in computed metrics we would expect a reduction factor in running time of $32/7 \approx 4.6$. The actual reduction is slightly larger, which can be explained by the fact that in addition to the number of computed metrics being reduced, texture interpolation on a GPU is computationally cheaper than in-kernel interpolation [9].

The measurements for the two kernels based on interpolated HARDI data are shown in detail in Figure 8.8. As with the two standard kernels, we see that spreading the computations involved in tracking one fiber across multiple threads improves performance. Furthermore, we can also observe the same behavior as before with regards to low fiber counts. The regular interpolated data kernel does not show linearly decreasing running times when reducing fiber counts to under 2500, whereas the kernel that uses three threads per fiber achieves running time reductions for smaller input sizes.

Now, the performance of the GPU-accelerated implementation is examined on different hardware. Figure 8.9 shows the running times for the Interpolated data kernel on two GPUs, a GTX 840M and GTX 780 Ti, as well as the running time for the single-threaded CPU technique.

As could be expected from a relatively low-end graphics card such as a GTX 840M, the
Figure 8.8: Running times for the kernel implementations based on interpolated HARDI data for up to 27000 outgoing fibers.

performance of the algorithm is lower than on a GTX 780 Ti. This is to be expected when considering the specifications of the two devices, as listed in Table 8.1. The GTX 780 Ti has about five times the number of multiprocessors of the other GPU and 7.5 times the number of CUDA cores. Using the low-end GPU will therefore lead to running times many times larger. However, even on this GPU the CUDA-based technique still scales much better with increasing fiber count than the CPU-based variant. In terms of performance for lower fiber counts, we see that the linear increase of running time starts at a lower number of tracked fibers on the lower-end GPU. This can be explained by difference in CUDA cores of the two GPUs. The GTX 840M only has a total of 384 CUDA cores. This means that all CUDA cores can already be saturated by 400 seed points, leading to an earlier increase in running time.

8.4 “Refraction” results

In this section, the effectiveness of the refraction-inspired correction method presented in Chapter 6 is shown. We compute a bundle of fibers in a small volume of HARDI data to show the problems of geodesic fiber tracts in areas of low anisotropy. The same bundle is then computed with the correctional method enabled, resulting in higher quality fibers that require less filtering before being able to be used in further practical applications.

First, a bundle of fibers has been computed through the corpus callosum region of the brain using the GPU-based implementation. The result is shown in Figure 8.10. While there are fibers that follow the arced shape of the corpus callosum from the starting region to the top right of the image, a large portion of the fibers travels outside the desired bundle. These fibers end up in an area with low anisotropy, as can be seen in more detail in Figure 8.11 where fibers are colored by the fractional anisotropy at the fiber positions.

As fiber density correlates with fractional anisotropy [28], we can assume that the high number of fibers in the area at the top of Figure 8.11 with low anisotropy is indicative of these fibers being false positives. We would expect a larger number of fibers following a path along the
Figure 8.9: Running times for the same *Interpolated data* implementation on two different GPU devices, as well as for the single-threaded CPU technique.

Figure 8.10: A set of 7000 fibers originating from a region on one side of the corpus callosum, computed by the GPU-accelerated algorithm.

corpus callosum arc to the top right of the image.

In order to reduce the number of false positives, we enable the refraction-inspired correction while computing a new set of fibers from the same starting region. As we use the GPU-accelerated algorithm to compute these fibers, which tracks fibers for eight steps before returning a sequence of eight new positions, we apply the correction with eight refraction iterations. This counteracts the fact that we apply the correction only once every eight tracking steps. The resulting fibers are shown in Figure 8.12.

From these newly computed fibers, it becomes clear that this correction method is well suited to reduce the generation of false positives in areas of low anisotropy. A large number of fibers that would previously exit the main bundle have been redirected along the path of the corpus callosum. As desired, this increases the number of valid fibers that end up in the top right of the image. Furthermore, the fiber structure to the left of the starting region are mostly unaffected. This shows that this correction method does not interfere with the branching behavior of the fibers computed using geodesic fiber tracking.
Figure 8.11: The same fibers as shown in Figure 8.10, colored according to the fractional anisotropy at each fiber position, with blue representing a high anisotropy and red representing a near-isotropic area. The zero-anisotropy areas at the bottom and right-hand side are an artifact of the anisotropy computation at the boundary.

Figure 8.12: A set of 7000 fibers originating from the same as the fibers shown in Figure 8.10, with 8 iterations of refraction-inspired correction applied after each 8 fiber-tracking steps.

In terms of performance, the cost of performing this correction is almost completely dependent on the time needed to find the local maxima on the ODF required to construct the virtual plane of refraction. In Table 8.4, the running times of three different kernels involved in the tracking process are listed. In this table, we can see that the accuracy with which the local maxima are computed can be disastrous for the overall performance. The local maxima are found by iterating over triangles of a tessellated sphere. Using 3\textsuperscript{rd} order tessellation results in 164 vertices on the surface of the sphere, which has a relatively small impact on the total running time. However, with the 642 vertices that appear when using 4\textsuperscript{th} order tessellation, the amount of triangles generated results in a correction algorithm that takes more than twice as long as eight tracking steps. The fibers in Figure 8.12 were generated with 3\textsuperscript{rd} order tessellation, indicating that the refraction-inspired correction method can be sufficiently accurate to generate satisfactory results without reducing performance by more than 25%.
### 8.5 Ricci scalar results

Finally, we aim to show the value of Ricci scalars in identifying fiber bundles. The two main criteria for this method would be its ability to distinguish a difference between two different fiber bundles and its ability to recognize similarity between two fiber bundles in the same region of the brain.

As an example of using Ricci scalars to identify fiber bundles, Figure 8.13 illustrates two cingulum bundles colored by their Ricci scalars. While the shapes of these two bundles are relatively similar, the Ricci scalar values along the fibers on either side are noticeably different. This shows that the Ricci scalar contains information beyond pure three-dimensional shape.

![Figure 8.13: Visual indication of the possibility of identifying fibers via Ricci scalars. Pictured are two cingulum bundles colored by the Ricci scalar values at the fiber positions, where red coloring indicates a positive value and a blue color indicates a negative value.](image)

As mentioned in Chapter 7, computing a Ricci scalar requires the computation of a large number of Finsler metric tensors. Similarly to the computation of the Finsler metric in the fiber-

---

**Table 8.4: Running times for a single execution of different kernels, for 3rd and 4th order tessellation of the ODF sphere.**

<table>
<thead>
<tr>
<th>Kernel</th>
<th>3rd order tessellation Running time (ms)</th>
<th>4th order tessellation Running time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolated data tracking (8 steps)</td>
<td>294.15</td>
<td>294.18</td>
</tr>
<tr>
<td>Maximum finder</td>
<td>60.82</td>
<td>569.19</td>
</tr>
<tr>
<td>Refraction correction (8 iterations)</td>
<td>0.76</td>
<td>0.76</td>
</tr>
</tbody>
</table>
tracking process, we compute the Ricci scalar in a varying direction along the geodesics, removing the option of preprocessing values. Table 8.5 shows the performance of a single execution of the kernel that computes the Ricci scalar value for a given position and direction. It is clear that the computation of the Ricci scalar is computationally very expensive, much more so than the computation of the Christoffel symbols for fiber-tracking. However, the improvement in performance achieved using the CUDA architecture ensure that it remains feasible to compute Ricci scalars in a reasonable time. This creates an opportunity for further research into the use of Ricci scalars in tractography, whereas a CPU-based implementation would likely be too impractical due to long running times.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Running time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolated data tracking (8 steps)</td>
<td>294.78</td>
</tr>
<tr>
<td>Ricci computation</td>
<td>631.09</td>
</tr>
</tbody>
</table>

Table 8.5: Running times for a single execution of a tracking kernel and the Ricci computation kernel.

The main task that remains in a practical application of Ricci scalars to process fibers in a lower-dimensional space is using these values for positive identification. Matching two similar fiber bundles through the same region of the brain as near identical remains an open task. For this, some form of a similarity measure is required to express how likely it is that two sets of Ricci scalar data represent the same set of physical fibers. If we consider the \( n \) Ricci scalars for each position of one fiber as an \( n \)-dimensional vector, it would be possible to take a simple Euclidean vector distance as the similarity measure. However, this would limit the input data to only support fibers of the same length, as the vector distance is only defined for vectors of the same dimension. Furthermore, it would require the fiber data to be completely aligned when comparing two fibers, such that each data point \( R_i \) in the first vector represents data at the same spatial position as \( R'_i \) in the second vector.

An alternative similarity measure considers the Ricci data for one fiber as a statistical distribution of values. It is then possible to take a probabilistic distance or divergence measure [29] as a way to indicate similarity. Using these types of similarities to cluster or classify fibers does not require the input data for all fibers to be physically aligned or to have the same number of data points. One probabilistic divergence measure is the Kullback-Leibler divergence, which has been applied in the context of machine learning to cluster data in multiple domains [30][31][32]. Such a method may be used in this context as well, combining probabilistic divergence with machine learning techniques to identify similar fibers or classify a set of tracked fibers.
Chapter 9

Conclusion

In this research, the behavior of the multi-valued geodesic fiber-tracking algorithm for HARDI data has been verified by implementing a single-threaded CPU variant of the algorithm. This implementation exhibits the expected behavior with regards to complex structures involving branching or crossing fiber bundles. Downsides of this algorithm include a high running time and the generation of many false positives in the output.

In order to improve the performance issues of the technique, a CUDA-based GPU-accelerated implementation of the geodesic fiber-tracking algorithm has been developed. Leverage of GPU architecture to increase performance has been increased by experimenting with four different implementations. The experiments with these four variants show that use of texture memory for interpolation of HARDI data is preferable over using grid-aligned HARDI data points to compute metric tensors and interpolating those. Furthermore, using multiple threads per computed fiber can improve performance to a varying degree as well. Computing parts of the algorithm in parallel and using shared memory as a communication method will improve the running time of the algorithm, provided the number of registers required for each thread and the amount of shared memory needed are limited.

Another downside of the geodesic fiber-tracking algorithm is the generation of false positives. In order to improve the overall quality of the output fibers, a novel correction method inspired by the physical concept of refraction has been presented. This technique adjusts the direction of fibers during the tracking phase to contain the fibers to regions with high anisotropy. The resulting output has been shown to exhibit less false positives in regions of low anisotropy. While the adjustments to the fiber direction mean the resulting fibers are not technically geodesics anymore, the correction method appears to retain the behavior of the geodesic fiber-tracking algorithm in regions where the computed tracts are already valid. This technique can reduce performance of the fiber-tracking process by up to 25%, a cost that is often outweighed by the benefits of not having generate a larger number of fibers to offset a post-processing filter.

Using the GPU-accelerated implementation of the fiber-tracking algorithm, it becomes possible to output tens of thousands of fibers in under a minute. With such a large output size, further analysis of this output can become highly complex. In order to make the results more manageable, dimensional reduction can be used. In this work, we have proposed the use of the Ricci scalar as a curvature measure that also contains information on the underlying geometry. This scalar measure can provide a method to distinguish two bundles of fibers that exist in different regions of the brain while being similar in shape. The computation of Ricci scalars is computationally expensive, even when compared to the complex fiber-tracking algorithm. While
not fully optimized, the CUDA-based implementation we presented is able to reconstruct fibers including their Ricci scalar measure while maintaining greatly improved performance over the CPU-based tractography.

9.1 Future work

As mentioned in the section on tracking kernel variants, in our GPU-based implementation only the computation of the Finsler metric tensor and its partial derivatives has been parallelized across multiple threads. It may be possible that further performance improvements can be obtained by spreading other parts of the algorithm across multiple threads, such as the computation of individual metric elements $g_{ij}$ or the Christoffel symbols. While tracking a large number of fibers already scales linearly with the amount of fibers, increasing the number of threads involved with tracking a single fiber can increase the utilization of the GPU for lower fiber counts.

While our correction method appears to reduce the number of false positives in the output greatly, some fibers that do not correspond to physical tracts remain. Adding post-processing techniques that are more sophisticated than simple region-to-region filtering could improve the output quality as well. For example, it would be interesting to implement the connectivity measure $[10]$, used to give a confidence score to each computed fiber based on the size of the diffusion along the geodesic, in this context of Finsler geometry. Furthermore, in order to cope with the large amount of fibers that can be generated by our fiber-tracking algorithm, this post-processing method would likely benefit from a highly parallel CUDA-accelerated implementation as well.

As mentioned in the previous chapter, the extraction of local maxima on the diffusion ODF is the bottleneck in the execution of the refraction-inspired correction. In our implementation, we iterate over triangles on a tessellated sphere to compare ODF values to those of the neighboring vertices. This is a relatively slow method of finding local maxima. Aganj et al. $[26]$ have proposed a technique to extract local maxima on an ODF represented by spherical harmonics through an analytical dimension reduction. It could be beneficial to include such a method in our correction technique to improve its performance.

Using Ricci scalars as a dimensional reduction shows promise in the context of distinguishing between fiber bundles. However, the fact that this measure is costly to calculate means that further optimization of the algorithms used to calculate these values would be very useful. The potential of using Ricci scalars to identify and match two near-identical bundles passing through the same brain region to each other is unconfirmed as of yet. One path for further research is the application of probabilistic divergence measures such as the Kullback-Leibler divergence $[30][31][32]$ as a similarity measure for clustering and classification using machine learning methods. The physical interpretation of a positively or negatively valued Ricci scalar in Finsler geometry also remains unknown. The Ricci scalar could be a highly useful tool in automated analysis of fiber bundles generated by different tractography methods, making it an interesting subject for further research.
Bibliography


