Bachelorarbeit im Rahmen des Studiengangs Scientific Programming

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 – Medizintechnik und Technomathematik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Simulation von 3D-Polarized Light Imaging mit GPU basiertem Ray Tracing

9. September 2020

Alexander Kobusch
(Matrikelnummer: 3162018)

Selbstständigkeitserklärung

Hiermit versichere ich, dass diese Arbeit von mir selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Bachelorarbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

(Ort, Datum)	(Unterschrift des Studenten)

Diese Arbeit wurde betreut von:

1. Prüfer: Prof. Dr.-Ing. Andreas Terstegge

Fachhochschule Aachen

2. Prüfer: Felix Matuschke, M. Sc.

Forschungszentrum Jülich (INM-1)

Kurzfassung

In der Arbeitsgruppe „Faserbahnarchitektur“ am Institut für Neurowissenschaften und Medizin (INM-1) des Forschungszentrums Jülich wird an der Rekonstruktion der Orientierung von Nervenfasern im Gehirn mithilfe von polarisierter Lichtgebung geforscht. Dazu werden Gehirnschnitte bei der Mikroskopietechnik 3D-Polarized-Light Imaging (3D-PLI) mit polarisiertem Licht durchleuchtet, welches aufgrund der Doppelbrechungseigenschaft der Myelinscheiden, welche Nervenfasern umgeben, in der Polarisation verändert wird. Durch eine geeignete Signalanalyse der Polarisationsveränderungen der Lichtstrahlen, kann auf die Orientierung von Fasern im Gehirn geschlossen werden.

Zur Untersuchung des als Grundlage zur Bestimmung von Faserorientierungen aktuell verwendeten Modells werden Simulationsverfahren eingesetzt, welche das 3D-PLI Verfahren simulieren. Unter Verwendung von synthetisch erzeugten Daten, sollen physikalische Eigenschaften der Mikroskopietechnik untersucht und verbessert werden.

Im Rahmen dieser Arbeit wird auf Grundlage der Implementierung der 3D-PLI-Simulation SimPLI (engl. simulation method for 3D-PLI) eine Simulation entwickelt, welche mithilfe von GPU basiertem Ray Tracing die benötigten Berechnungen der Simulation durchführt. Dabei werden Algorithmen aus dem Bereich der Kollisionskontrolle angewendet und die Berechnungen auf der GPU mithilfe von CUDA parallelisiert. Das Ergebnis ist eine neue Implementierung für eine 3D-PLI Simulation, welche im Gegensatz zu SimPLI keine Diskretisierung des Volumens in Subvoxel vornimmt. Dadurch werden die optischen Eigenschaften des Gewebes an den Positionen der Lichtstrahlen genauer bestimmt, wodurch realistischere Ergebnisse erzeugt werden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das Institut für Neurowissenschaften und Medizin - Strukturelle und funktionelle Organisation des Gehirns (INM-1) hat das Ziel ein hochaufgelöstes 3D-Modell des menschlichen Gehirns auf den verschiedenen organisatorischen und strukturellen Ebenen zu erstellen. Dazu werden Gehirne in wenige Mikrometer dünne Schnitte geschnitten und mit unterschiedlichen Bildgebungsverfahren digitalisiert.

Unter anderem wird das Verfahren 3D-Polarized Light Imaging (3D-PLI) verwendet, bei dem 60 µm dicke Gehirnschnitte mit polarisiertem Licht durchleuchtet werden. Eine Signalanalyse des transmittierten Lichtes ermöglicht aufgrund der doppelbrechenden Eigenschaft der Nervenfasern die Ermittlung ihrer Orientierung.

Das Vorbereiten und die Durchführung einer 3D-PLI-Messung und den Prozessen, welche für die Rekonstruktion der Orientierung der Fasern durchgeführt werden, sind sehr aufwendig und nehmen viel Zeit in Anspruch. Es ist eine Simulation für 3D-PLI entwickelt worden, mit deren Hilfe das Verfahren veranschaulicht wird und Probleme, die bei Messungen auftreten können, untersucht werden.

Die bisherige Simulation wird in die Generierung eines Volumens mit der Berechnung der optischen Eigenschaften der Fasersegmente und die Berechnungen für die Eigenschaften der Lichtstrahlen beim Passieren des Volumens eingeteilt. Dabei wird das Volumen in Voxel unterteilt, welche die Lichtstrahlen der Messung passieren. Für jeden Voxel werden die optischen Eigenschaften im Mittelpunkt bestimmt. Passiert ein Lichtstrahl den Voxel, so werden unabhängig von der Postion im Voxel die zuvor ermittelten optischen Eigenschaften des Mittelpunktes für die Berechnung verwendet. Dadurch treten Fehler bei der Bestimmung der optischen Eigenschaften auf, welche auf die Diskretisierung der optischen Eigenschaften zurückzuführen ist. Außerdem wird für das Nachhalten der optischen Eigenschaften der Voxel viel Speicherplatz benötigt. Aus diesem Grund soll ein neues Verfahren entwickelt werden, welches Diskretisierungsfehler der Ermittlung der optischen Eigenschaften vermeidet und weniger Speicher benötigt. Dabei soll dieses auf der GPU parallelisiert werden, da der Ablauf der Simulation sehr rechenaufwendig ist.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist die Entwicklung eines Algorithmus, der mithilfe von Kollisionserkennung die optischen Eigenschaften an einer Position im Volumen berechnet und diese auf den jeweils betrachteten Lichtvektor anwendet. Dadurch wird die Simulation nicht mehr in die Generierung eines diskretisierten Volumens mit der Berechnung der optischen Eigenschaften und der anschließenden Berechnungen der passierenden Lichtstrahlen unterteilt. Diese Schritte sollen so von jedem Lichtstrahl an jeder erreichten Position im Gewebe ausgeführt werden. Dabei wird das Volumen nicht in Voxel diskretisiert, wodurch die Berechnungen der Simulation sich mehr den realen Messungen annähern sollen und weniger Speicher benötigt wird. Der Ablauf des Algorithmus soll auf der GPU parallelisiert werden, damit die Laufzeit verkürzt wird.

1.3 Aufbau der Arbeit

Zunächst wird ein Einblick in die Durchführung der 3D-PLI-Messung gegeben und der Ablauf der 3D-PLI-Simulation SimPLI erläutert. Anschließend werden die Grundprinzipien der GPU-Programmierung und die Nutzung der Programmierschnittstelle CUDA dargelegt. Der neu entwickelte Algorithmus wird im Nachfolgenden vorgestellt und die Implementierung unter Verwendung der GPU wird beschrieben. Daraufhin werden die ermittelten Ergebnisse mit der vorhandenen Simulation verglichen und die Performanz des neu entwickelten Algorithmus untersucht. Abschließend werden die Vorteile und Nachteile der neu entwickelten Implementierung herausgearbeitet und ein Fazit sowie ein Ausblick für weitere mögliche Verbesserungen gegeben.

2 Grundlagen

2.1 3D-Polarized Light Imaging

Mit der Mikroskopietechnik 3D-PLI [1] werden post mortem Gehirnschnitte untersucht. Das zu untersuchende Gehirn wird zunächst entnommen und in Formaldehydlösung fixiert. Anschließend wird das Gewebe mit Frostschutzmittel versetzt und tiefgefroren. Mit einem Kryostat-Mikrotom, ein heruntergekühltes Schneidegerät, werden 60 µm dünne Gehirnschnitte angefertigt. Diese werden mit Glycerin bedeckt und auf einem Glasobjektträger eingeschlossen.

Die Bestimmung der Orientierung von Nervenfasern beim 3D-PLI basiert auf der doppelbrechenden Wirkung der Myelinscheiden, welche die Axone der Nervenzellen umgibt. Beim Auftreffen von polarisiertem Licht auf die myelinisierten Fasern ändert sich der Polarisationszustand des Lichts abhängig von der optischen Achse der Myelinscheide, welche die Ausrichtung der Faser beschreibt. Durch die Analyse der Polarisationsveränderungen kann die Orientierung der Nervenfasern bestimmt werden.

Der Aufbau des Polarisationsmikroskop setzt sich aus einer Lichtquelle und CCD-Bildsensoren (engl. Charge Coupled Device) zusammen, zwischen denen das Gewebe von zwei Polarisationsfiltern und einer $\lambda/4$-Verzögerungsplatte eingeschlossen wird (Abbildung 1a). Die Lichtquelle emittiert unpolarisiertes Licht, welches durch einen Polarisationsfilter linear polarisiert wird. Beim Passieren der $\lambda/4$-Verzögerungsplatte wird dieses zirkular polarisiert. Das zirkular polarisierte Licht trifft auf das Gewebe und ändert aufgrund der Doppelbrechung der Myelinscheiden den Polarisationszustand zu elliptisch polarisiertem Licht. Ein senkrecht zum Ersten ausgerichteter Polarisationsfilter (Analysator) analysiert das eintreffende Licht. Mithilfe der CCD-Bildsensoren kann abschließend das entstandene Helligkeitsprofil gemessen werden. Für die Rekonstruktion der Orientierung der Nervenfasern werden Aufnahmen benötigt, bei denen das polarisierte Licht unterschiedlich auf das Gewebe trifft. Dazu werden bei den Messungen je nach Bedarf die Polarisationsfilter und die Verzögerungsplatte in äquidistanten 10° oder 20° Schritten bis zu 180° rotiert. Zusätzlich kann auch das Gewebe in der Ebene nach links, rechts, oben und unten gekippt werden [2]. Auf diese Weise ist pro Sensor der Messung ein Helligkeitsverlauf erkennbar. Dieser Helligkeitsverlauf von jedem Bildpunkt kann durch ein im INM-1 entwickelten Modell als sinusförmige Kurve approximiert werden, aus welcher man die Transmittanz, Direktion und Retardierung mithilfe einer Signalanalyse entnehmen kann (Abbildung 1b).

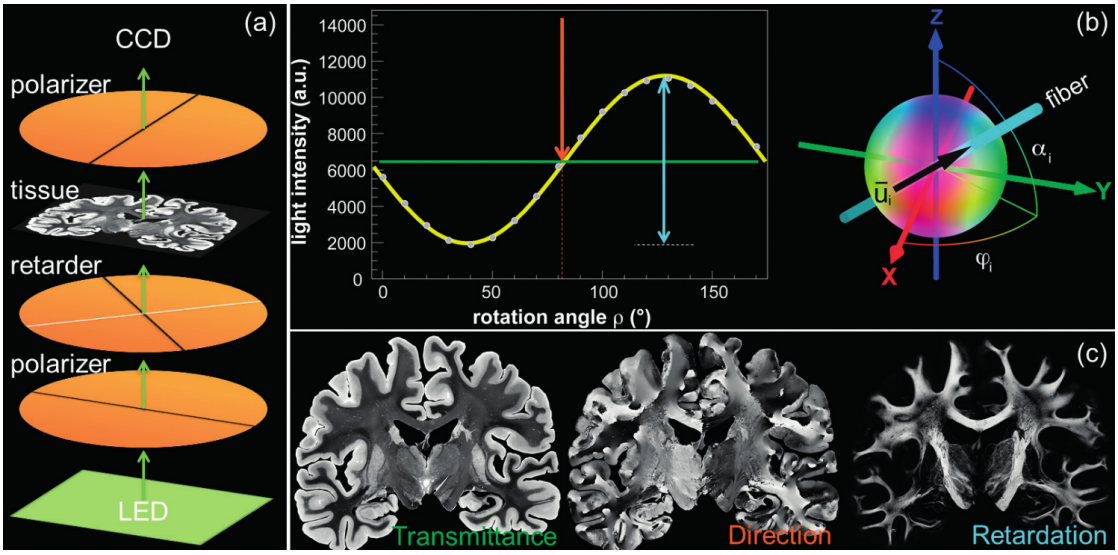


Abbildung 1: (a) Aufbau des PLI
(b, links) Sinusförmige Kurve, aus der Direktion (rot), Transmittanz (grün) und Retardierung (blau) ermittelt werden können
(b, rechts) Direktion φ: Winkel in der Ebene, Inklination α: Winkel aus der Ebene
(c) Parameterkarten der Messung [11]

Die zu erwartende Messkurve für einen Messpunkt mit dem Drehwinkel ρ lautet:

$$I = \frac{I_0}{2} \cdot (1 + sin(2\rho - 2\varphi) \cdot sin(\delta)), \qquad \delta \approx 2\pi \frac{d \cdot \Delta n}{\lambda} \cdot cos(\alpha)^2. \tag{1}$$

Dabei bezeichnet I_0 die Startintensität des emittierten Lichtstrahls, φ die Direktion und δ die Retardierung. Die Retardierung kann unter Verwendung der Dicke des Gewebes d, der Stärke der Doppelbrechung des Gewebes Δn, die Wellenlänge des Lichtes λ und der Inklination α approximiert werden und beschreibt die Phasenverschiebung des polarisierten Lichtes beim Passieren des Gewebes. Die Beschreibung der Orientierung der Nervenfaser kann mit der Transmittanz, der Direktion und der Inklination vorgenommen werden (Abbildung 1c). Dies ist möglich, da die Transmittanz die Lichtdurchlässigkeit des Gewebes beschreibt, die Direktion die Ausrichtung der Nervenfaser innerhalb der Schnittebene darstellt und die Inklination, die Ausrichtung aus der Schnittebene heraus angibt. Die Inklination kann mithilfe der Retardierung berechnet werden [1].

2.2 SimPLI mit Müller-Stokes-Formalismus

SimPLI ist ein im INM-1 entwickeltes Simulationstool zur Analyse von PLI-Messungen. Dieses wird dazu verwendet, um besonders herausfordernde Phänomene bei Messungen, wie kreuzende Nervenfasern, zu untersuchen und Lösungsansätze für diese Phänomene bezüglich der verwendeten Modelle zu entwickeln.

In der ursprünglichen Variante wird die Polarisationsveränderung mithilfe des Jones-Formalismus beschrieben. Dabei wird angenommen, dass die Polarisationsfilter perfekt funktionieren und das Licht kohärent ist [3]. Kohärentes Licht besteht aus elektromagnetischen Wellen, welche eine feste Phasenbeziehung aufweisen. In der Praxis wird jedoch nur ein Polarisationsgrad größer als 99,8 % erreicht und das Licht ist größtenteils inkohärent. Um diese Eigenschaften berücksichtigen zu können, wurden bei einer neueren Implementierung der Simulation zur Beschreibung der Polarisationsveränderung Müller-Matrizen zusammen mit Stokes-Vektoren verwendet.

Der Müller-Stokes-Formalismus wird zur mathematischen Beschreibung von partiell polarisiertem und nicht kohärentem Licht verwendet. Dabei wird der Polarisationszustand durch einen 4×1 Vektor (Stokes-Vektor) und die optischen Elemente durch eine 4×4 Matrix (Müller-Matrix) dargestellt. Die Veränderung des Polarisationszustandes kann durch die Multiplikation des Stokes-Vektors mit den Müller-Matrizen, welche die optischen Elemente beschreiben, dargestellt werden.

2.2.1 Stokes-Vektoren

Stokes-Vektoren bestehen aus den 4 Komponenten S_0, S_1, S_2, S_3.
S_0 beschreibt die absolute Lichtintensität I, S_1 den größeren Anteil von horizontal (P_{0°) zu vertikal (P_{90°) polarisiertem Licht, S_2 den größeren Anteil von +45° (P_{45°) zu -45° (P_{135°) polarisiertem Licht und S_3 den größeren Anteil von rechtszirkular (P_{RZ}) zu linkszirkular (P_{LZ}) polarisiertem Licht (siehe (2)).

$$\vec{S} = \begin{pmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{pmatrix} = \begin{pmatrix} I \\ P_{0^\circ} - P_{90^\circ} \\ P_{45^\circ} - P_{135^\circ} \\ P_{RZ} - P_{LZ} \end{pmatrix}, \quad S_0^2 \geq S_1^2 + S_2^2 + S_3^2 \tag{2}$$

Der Polarisationszustand eines Lichtstrahls kann so mithilfe eines Stokes-Vektor beschrieben werden. Im Folgenden sind einige Beispiele von Stokes-Vektoren für linear horizontal (LH), linear vertikal (LV), linear um 45° (L+45°), links-zirkular (LZ), rechts-zirkular (RZ) polarisiertes und unpolarisiertes (U) Licht zu sehen:

$$\vec{S}_{LH} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad \vec{S}_{LV} = \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \end{pmatrix}, \quad \vec{S}_{L+45^\circ} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad \vec{S}_{LZ} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}, \quad \vec{S}_{RZ} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \quad \vec{S}_{U} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{3}$$

In Kugelkoordinaten ist der Stokes-Vektor folgendermaßen definiert:

$$\vec{S} = \begin{pmatrix} I \\ I \cdot p \cdot \cos(2\psi) \cdot \cos(2\chi) \\ I \cdot p \cdot \cos(2\psi) \cdot \cos(2\chi) \\ I \cdot p \cdot \sin(2\chi) \end{pmatrix}, \qquad p = \frac{\sqrt{S_1^2 + S_2^2 + S_3^2}}{S_0}. \tag{4}$$

I beschreibt die absolute Lichtintensität eines Lichtstrahls und $p \in [0, 1]$ den Polarisationsgrad. $\psi \in [0, \pi]$ und $\chi \in [-\pi/4, \pi/4]$ geben die Winkel des Vektors (S_1, S_2, S_3) in Polarkoordinaten an [8].

2.2.2 Müller-Matrizen

Eine Müller-Matrix ist eine 4×4 Matrix, welche die Änderung der Intensität und des Polarisationszustandes von polarisiertem oder unpolarisiertem Licht beschreibt. Daher können die Eigenschaften der optischen Elemente bei dem 3D-PLI Messaufbau (Polarisationsfilter, Verzögerungsplatte und Gewebeschnitt) mithilfe von Müller-Matrizen in der Simulation nachgestellt werden. Der grundlegende Aufbau einer Müller-Matrix sieht folgendermaßen aus:

$$M(\delta, D, \tau) = \tau \cdot \begin{pmatrix} 1 & D & 0 & 0 \\ D & 1 & 0 & 0 \\ 0 & 0 & \sqrt{1-D^2} \cdot cos(\delta) & \sqrt{1-D^2} \cdot sin(\delta) \\ 0 & 0 & \sqrt{1-D^2} \cdot sin(\delta) & \sqrt{1-D^2} \cdot cos(\delta) \end{pmatrix}. \tag{5}$$

δ gibt die Phasenverzögerung zwischen horizontal und vertikal polarisiertem Licht an. D stellt die Abschwächung der Intensität vom Licht in Abhängigkeit zur Polarisation und τ die Abschwächung der Intensität ohne Berücksichtigung der Polarisation dar. Die Auswahl der Parameter muss bei dem verwendeten Modell folgende Voraussetzungen erfüllen:
Die X-Achse für die Beschreibung der Polarisationsrichtung muss für eine Verzögerungsplatte so gewählt werden, dass Licht, welches entlang der X-Achse polarisiert ist, die geringste Verzögerung aufweist. Entlang der Y-Achse polarisiertes Licht soll dementsprechend am meisten verzögert werden. Zudem wird eine $\lambda/4$-Verzögerungsplatte eingesetzt, weshalb $\delta = \pi/2$ und $D = 0$ gelten muss. Bei den linearen Polarisationsfiltern, welche nur Licht mit einer bestimmten Polarisationslichtung passieren lassen, muss die X-Achse so gewählt werden, dass entlang dieser Achse das Licht maximal transmittiert, also minimal abgeschwächt wird. Bei perfekten Filtern würde so $\delta = 0$ und $D = 1$ gelten. Auch bei der Verwendung von Matrizen zur Beschreibung der optischen Eigenschaften des Gewebes muss diese Bedingung gelten, sodass Licht entlang der X-Achse minimal abgeschwächt wird. Die Parameter δ und D werden Abhängig von der jeweiligen Faser gewählt, wobei $\delta > 0$ und $D > 0$.

Die Rotation der optischen Elemente gegen den Uhrzeigersinn, welche beim 3D-PLI durchgeführt werden, können mithilfe der folgenden Drehmatrix beschrieben werden:

$$R(\xi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(2\xi) & -sin(2\xi) & 0 \\ 0 & sin(2\xi) & cos(2\xi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{6}$$

Die Müller-Matrix zur Beschreibung einer um den Winkel ξ gegen den Uhrzeigersinn rotierten Verzögerungsplatte oder eines Polarisationsfilters lautet somit:

$$M(\xi, \delta, D, \tau) = R(\xi) \cdot M(\delta, D, \tau) \cdot R(-\xi). \tag{7}$$

Bei der simulierten Messung können die Polarisationszustände der Lichtstrahlen jeweils als Stokes-Vektor und die optischen Elemente als Müller-Matrizen dargestellt werden. Der veränderte Polarisationszustand $\vec{S}'$ eines Lichtstrahls $\vec{S}$ nach dem Passieren eines optischen Elementes kann durch die Multiplikation des Vektors mit einer Müller-Matrix beschrieben werden [8]:

$$\vec{S}' = M \cdot \vec{S}. \tag{8}$$

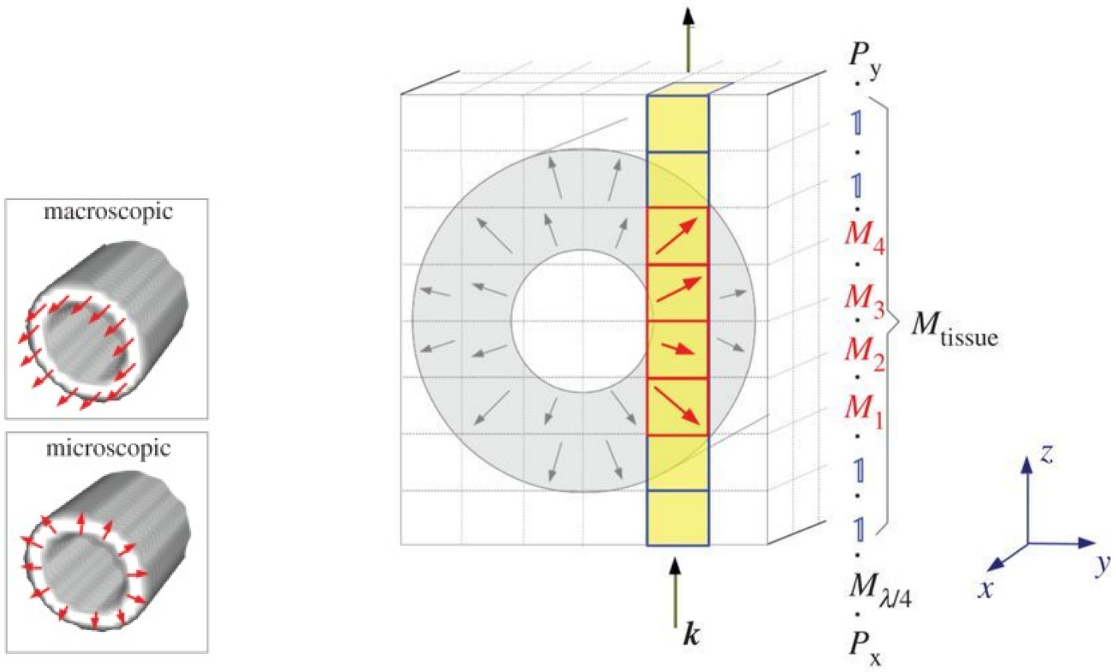
2.2.3 Ablauf SimPLI

SimPLI (engl. simulation method for 3D-PLI) ist ein im INM-1 entwickeltes Simulationstool zur Nachstellung von 3D-PLI Messungen. Das Verfahren besteht aus drei wesentlichen Schritten:

Diskretisierung eines Volumens mit Fasern und Berechnung der optischen Eigenschaften

Bei der Simulation werden zunächst synthetische Fasern in Form von Punkten mit jeweils einem Radius angegeben. Diese modellieren als röhrenartige Strukturen Fasern. Zusätzlich wird eine Volume of Interest definiert, welches im Nachfolgenden den betrachteten Bereich für das simulierte Gewebe darstellt. Anschließend wird das Volumen unter Beachtung der angegebenen Voxelgröße in Subvoxel diskretisiert. Für jeden dieser Subvoxel werden die optischen Eigenschaften in der Mitte des Voxels berechnet.

Dazu werden für jeden Mittelpunkt der Subvoxel alle Fasersegmente in dem Volume of Interest, bestehend aus jeweils zwei Punkten und zwei Radien, überprüft, ob dieser innerhalb eines Fasersegmentes liegt. Ein Fasersegment ist ein röhrenartiger Teil von einer Faser, welcher aus zwei Punkten mit jeweils einem Radius besteht (siehe Abbildung 6a). Dabei kann jede Faser eines Faserbündels unterschiedliche Schichten (Layer) besitzen, welche unterschiedliche optische Eigenschaften aufweisen (siehe Abbildung 6b). Diese sind in Form der Doppelbrechung Δn, der Abschwächung μ und einem Vektor für die optische Achse der Faser gegeben. Ist ein zu überprüfender Punkt nicht innerhalb eines Fasersegments, so wird dieser Subvoxel als Hintergrund klassifiziert. Ansonsten wird von dem nächstliegenden Fasersegment die Schicht bestimmt, welche den Punkt einschließt. Beim Programmstart wurden die Eigenschaften für die einzelnen Schichten der Faserbündel angegeben. Somit kann nach dem Bestimmen der entsprechenden Schicht die Eigenschaften für Δn und μ sowie das Modell zur Simulation der Doppelbrechung ausgelesen werden. Je



(a) (b)

Abbildung 2: Simulationsmethode der Doppelbrechung [9]
(a) Modelle für die Doppelbrechung bei Fasern: Makroskopisch (parallel), Mikroskopisch (radial)
(b) Berechnung der Lichtintensität für einen Lichtvektor k
PLI-Simulation: Gewebe als mehrere doppelbrechende Medien (Myelinschicht) entlang der Z-Achse

nach Doppelbrechungsmodell wird nun der Vektor zur Beschreibung der Doppelbrechung berechnet. Ist die Schicht nicht doppelbrechend, so wird ein Nullvektor zurückgegeben. Bei dem mikroskopischen Modell (radial) wird der Vektor mit dem kürzesten Abstand zum Fasersegment bestimmt, während bei dem makroskopischen Modell der Vektor zur Beschreibung der örtlichen Lage des Fasersegments ermittelt wird. Diese Modelle beschreiben jeweils, wie die optischen Achsen der Fasern angeordnet sind (siehe Abbildung 2a).

Berechnung der Lichtintensität

Die transmittierten Lichtstrahlen beim simulierten 3D-PLI werden als Stokes-Vektoren dargestellt und die Polarisationsfilter sowie die Verzögerungsplatte werden als Müller-Matrizen beschrieben. Die optischen Eigenschaften des Gewebes wurden im ersten Schritt für die Subvoxel berechnet.

Bei der Simulation wird beginnend bei den untersten Subvoxeln jeweils ein Lichtstrahl in Z-Richtung durch das Volumen verfolgt. Zuvor werden die unpolarisier-

ten Lichtstrahlen mit den Müller-Matrizen für den ersten Polarisationsfilter und der Verzögerungsplatte multipliziert. Anschließend bewegt sich der Lichtstrahl mit einer festgelegten Schrittweite durch das Volumen. An jeder so erreichten Position wird der Stokes-Vektor des Lichtstrahls, welcher den Polarisationszustand beschreibt, mit einer Müller-Matrix multipliziert, welche die Polarisationsveränderung durch das Gewebe beschreibt und für den Subvoxel hinterlegt ist. Dabei wird diese Matrix zuvor mit einer Rotationsmatrix transformiert, damit die Drehung des Gewebes beim Versuchsaufbau berücksichtigt wird. Nach dem Passieren des Volumens mit Fasern wird noch die Wirkung des Analysators als Müller-Matrix multipliziert, sodass am Ende die Lichtintensität des Lichtstrahls ausgelesen werden kann (Abbildung 2b). Diese Berechnungen werden für alle Lichtstrahlen durchgeführt, welche von einer zuvor bestimmten Startposition so durch das Volumen verlaufen, dass alle Messpunkte der simulierten CCD-Kamera erreicht werden.
Somit wird für jeden Lichtstrahl die folgende Berechnung durchgeführt:

$$\vec{S} = M_Y \cdot \prod_i (R_i^{-1} M_i R_i) \cdot M_{\lambda/4} \cdot M_X \cdot \vec{S_0} \tag{9}$$

Hinzufügen von Unschärfe und Rauschen

Im letzten Schritt werden den simulierten Messergebnisse noch Unschärfe und Rauschen hinzugefügt, damit diese mit realen Messergebnissen verglichen werden können. Dazu werden die Messergebnisse auf die Pixelgröße des zu entstehenden Bildes skaliert. Anschließend wird für die Unschärfe ein Gauß-Filter und für das Rauschen eine poissonverteilte Zufallsfunktion verwendet [3].

2.3 GPU-Programmierung

Die ursprünglich nur zur Bilddarstellung auf dem Monitor verwendete GPU (Graphics Processing Unit) ist ein auf die Berechnung von Grafiken spezialisierter Prozessor. Aufgrund der speziellen Architektur wird diese heutzutage jedoch auch zur Parallelisierung von Berechnungen genutzt.
Im Gegensatz zu CPUs (Central Processing Unit), bestehend aus meist wenigen, leistungsstarken Prozessorkernen, sind in einer GPU sehr viele, jedoch leistungsschwächere Rechenkerne verbaut. Des Weiteren besitzt die Grafikkarte noch einen eigenen Videospeicher, auf den mit einer hohen Bandbreite zugegriffen werden kann. Diese Architektur ist darauf ausgelegt, gleichzeitig, so viele Rechenoperationen wie möglich durchzuführen.

Damit ein Problem auf der Grafikkarte parallelisiert werden kann, müssen folgende Voraussetzungen gegeben sein:
Zum Einen muss die Berechnung des gestellten Problems in gleichartige Teilprobleme teilbar sein, die weitgehenden unabhängig voneinander berechenbar sind. Des Weiteren muss die Rechenkraft eines Threads für ein Teilproblem ausreichend sein, damit die Berechnung in einer sinnvollen Zeit abgearbeitet werden kann. Außerdem sollte die Anzahl der gleichartigen Teilprobleme sehr groß sein, damit die Anzahl der vielen Rechenkerne ausgenutzt werden kann.

2.3.1 CUDA

Die von Nvidia entwickelte Programmierschnittstelle CUDA ermöglicht die Ausführung von Programmcode auf Nvidia GPUs. Die Programmierschnittstelle stellt dazu Threads bereitet, welche zur Abarbeitung von Programmcode auf GPU-Rechenkernen dienen. Dabei sind die Threads in Blöcken organisiert, welche in einem Grid angeordnet sind. Die Anordnung der Threads in einem Block, sowie die Anordnung von Blöcken in dem Grid, kann dabei je nach Angabe des Programmierers entweder eindimensional, zweidimensional oder dreidimensional erfolgen (Abbildung 3). So kann jeder einzelnen Thread mit seiner Position im Grid und im Block angesteuert werden. Jeder Thread hat seinen eigenen privaten Speicher und einen gemeinsamen Speicher mit den Threads aus dem selben Block. Zusätzlich existiert ein globaler Speicher auf den alle Threads zugreifen können. Die GPU arbeitet dabei nach dem Prinzip *Single Programm Multiple Data*. Das bedeutet, dass alle Threads den selben Programmcode abarbeiten, jedoch unterschiedliche Daten verwenden. Dazu muss explizit vom Programmierer Speicher auf der GPU allokiert werden und die Daten anschließend auf die Grafikkarte kopiert werden. Nach dem Durchführen der Operationen auf der GPU müssen die Daten wieder in den Hauptspeicher kopiert werden.
Der Programmcode wird bei CUDA in *Host-Code* und *Device-Code* unterschieden. Die CPU wird als *Host* und die Grafikkarte als *Device* bezeichnet. Um parallelisierten Code auf der GPU auszuführen verwendet man einen *Kernel*, der mit dem Schlüsselwort *global* gekennzeichnet wird. Dieser Kernel enthält *Device-Code* oder *Device-Funktionen* und wird aus dem *Host-Code* aufgerufen. Beim Aufruf des Kernels wird die Anzahl der Blöcke und Threads angegeben, um die Anzahl der verwendeten GPU-Prozesse vorzugeben [10].

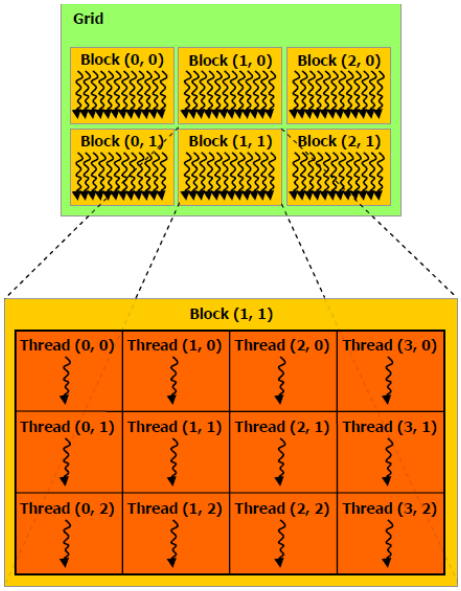


Abbildung 3: Koordination der Threads mit CUDA
Beispiel für zweidimensionale Anordnung eines Grid mit zweidimensionalen Blöcken mit Threads [10]

2.3.2 Thrust

Die speziell zur einfachen Verwendung von CUDA entwickelte Bibliothek Thrust imitiert die *Standard Template Library (STL)* von *C++*. Dabei werden die Daten-Container *host-vector* und *device-vector* eingeführt, welche die normalen Vektoren ersetzen sollen. Dies ist sinnvoll, da im *Device-Code* keine normalen Vektoren verwendet werden können. Auf den von Thrust bereitgestellten Vektoren können bekannte STL-Algorithmen für die Sortierung und zur Veränderung des Containers ausgeführt werden. Dabei werden die Algorithmen je nach verwendetem Vektor auf der CPU oder der GPU ausgeführt. Ein weiterer Vorteil dieser Container ist das Wegfallen vom manuellen Allokieren und Freigeben von Speicher durch den Programmierer. Mithilfe von einfachen Zuweisungen wird Speicher auf der GPU durch Thrust allokiert und die Daten kopiert.

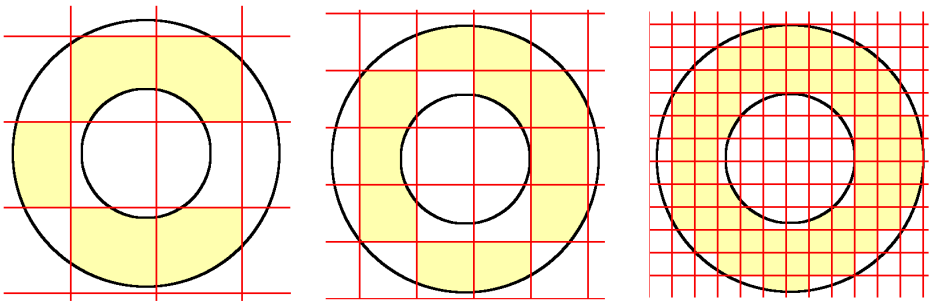


Abbildung 4: Faser im Volumen, dessen optische Eigenschaften je Voxel ermittelt werden. Dazu wird jeweils der Voxelmittelpunkt untersucht.
In gelb sind die Bereiche der Faser eingezeichnet, welche als doppelbrechendes Gewebe erkannt werden. Bestandteile der Faser, welche nicht eingefärbt sind, werden so als Hintergrund klassifiziert. Je kleiner die Voxel für die Diskretisierung gewählt werden, desto genauer werden die optischen Eigenschaften der Faser bestimmt. (Zweidimensionale Darstellung)

3 Realisierung

Wie bereits vorgestellt, kann der Ablauf der Simulation SimPLI in drei grobe Schritte unterteilt werden. Im Rahmen der Seminararbeit [6] wurde bereits der zweite Schritt, die Berechnung der Lichtstrahlen, auf der GPU parallelisiert. Die Diskretisierung des Volumens und die Berechnung der optischen Eigenschaften musste jedoch zuvor auf der CPU durchgeführt werden.
In dieser Arbeit soll das Volumen nicht diskretisiert werden und die Ermittlung der optischen Eigenschaften auf der GPU erfolgen. Dies hat den Grund, dass mit der Diskretisierung des Volumens in Subvoxel Ungenauigkeiten bei der Beschreibung der optischen Eigenschaften entstehen. Bei der Ermittlung der optischen Eigenschaften wird jeweils die Mitte des Subvoxels betrachtet und das Ergebnis auf den gesamten Voxel projiziert. Dabei treten Diskretisierungsfehler auf, die mit sinkender Voxelgröße kleiner werden (siehe Abbildung 4). Jedoch wird gleichzeitig der Aufwand für die Berechnung drastisch erhöht, da sich mit sinkender Voxelgröße die Anzahl der betrachteten Voxel kubisch erhöht. Durch die Anwendung von trilinearer Interpolation können genauere Ergebnisse erzielt werden, welche durch die zuvor möglichen Fehler beim Diskretisieren eingeschränkt werden. Ein weitere Nachteil ist der große Speicheraufwand für die berechneten Eigenschaften für jeden Voxel. Da die optischen Eigenschaften für jeden Voxel gespeichert werden müssen, wächst die benötigte Speichermenge auch kubisch.

```
Startpositionen der einzelnen Lichtstrahlen im Volumen berechnen
  Für jeden Lichtstrahl mit Position (x,y,z) und Stokes-Vektor S:
    Multiplikation von S mit Müller-Matrix für ersten Polarisator
    Multiplikation von S mit Müller-Matrix für Verzögerungsplatte
    Solange z innerhalb des Volumens:
      z += Schrittweite
      Berechne optische Eigenschaften an Stelle (x,y,z)
      Multipliziere S mit Absorption
      Multipliziere S mit Müller-Matrix für die Doppelbrechung des Gewebes
    Multiplikation von S mit Müller-Matrix für Analysator
    Speichern von erster Komponente von S (Intensität)
Rückgabe der gemessenen Intensitäten von allen Lichtstrahlen
```

Abbildung 5: Grober Ablauf der Simulation zur Berechnung der zu messenden Lichtintensitäten.

Der realisierte Ansatz verzichtet auf eine Diskretisierung des Volumens in Subvoxel und verwendet zur Bestimmung der optischen Eigenschaften Kollisionserkennung. Dadurch sollen die Eigenschaften des Gewebes an den von den Lichtstrahlen betrachtet Positionen genau ermittelt werden und so die Berechnungen genauer werden. Die Ausführungszeit des zu entwickelnden Algorithmus soll mittels Parallelisierung der Berechnungen auf der GPU verkürzt werden.

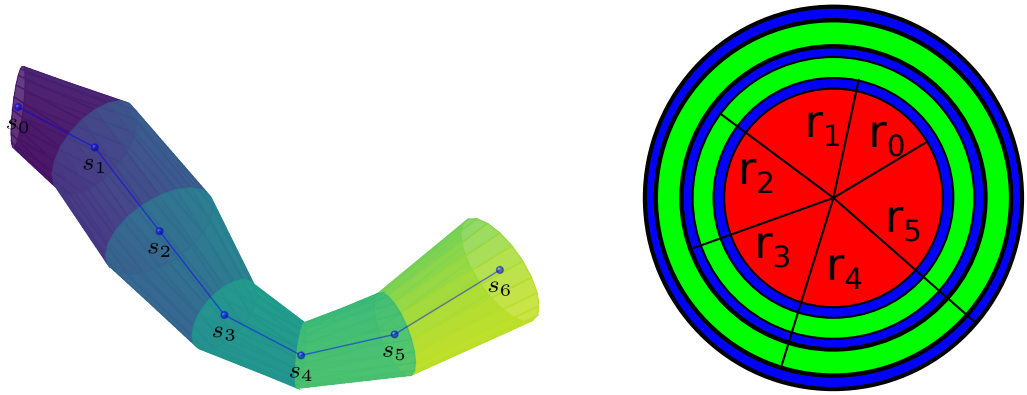
3.1 Ablauf des Algorithmus

Der neu entworfene Algorithmus soll für jeden Lichtstrahl durchgeführt werden und basiert auf den Grundzügen des Ray Tracing. Dabei soll ein Lichtstrahl zunächst den ersten Polarisationsfilter und die Verzögerungsplatte passieren, wodurch sich der Polarisationszustand des zugehörigen Stokes-Vektors verändert. Anschließend verläuft der Lichtstrahl mit einer zuvor angegebenen Schrittweite durch das Volumen. Dabei wird an jeder erreichten Position des Lichtstrahls überprüft, welche optischen Eigenschaften an dieser Stelle vorliegen. Dazu muss eine Kollisionskontrolle mit den Fasersegmenten im Volumen durchgeführt werden. Sofern sich die Position des Lichtstrahls in einem Fasersegment befindet, wird die entsprechende Schicht des Fasersegments ermittelt und die optischen Eigenschaften berechnet (siehe Abbildung 6b). Diese werden nun dazu verwendet, um mithilfe einer Multiplikation des Stokes-Vektors mit einer Müller-Matrix den veränderten Polarisationszustand des Lichtstrahls aufgrund der doppelbrechenden Fasersegmente zu beschreiben. Zudem wird die grundlegende Absorption des Gewebes und die allgemeine Doppelbrechung des Gewebes beachtet und mit dem Vektor multipliziert. Anschließend verschiebt sich der Lichtstrahl an die nächste zu erreichende Position.

An dieser werden mittels einer Kollisionskontrolle wieder die optischen Eigenschaften des Gewebes berechnet und die Polarisationsveränderung des Lichtstrahls ermittelt. Dieser Ablauf wiederholt sich so lange, bis das Ende vom Volumen erreicht ist. Nachfolgend wird der Stokes-Vektor des Lichtstrahls noch mit der Müller-Matrix zur Beschreibung des Analysators multipliziert. Als Ergebnis wird je Lichtstrahl die Lichtintensität, also die erste Komponente des Stokes-Vektors, zurückgegeben. Diese Werte stellen die am simulierten CCD-Sensor gemessenen Lichtintensitäten dar (siehe Abbildung 5). Dieser Ablauf wird für jede Kippung des Gewebes wiederholt, wobei die Startpositionen der Lichtstrahlen und die Richtung dieser je nach Kippung transformiert werden. Die Rotation der optischen Elemente bei einer Messung wird in der Simulation durch eine Rotation des Gewebes realisiert. Dabei werden die Eigenschaften des Gewebes mit einer Rotationsmatrix multipliziert. Da sich bei einer Rotation des Gewebes der Verlauf der Lichtstrahlen nicht verändert, werden für einen Lichtstrahl jeweils so viele Stokes-Vektoren betrachtet, wie Rotationen vorhanden sind.

3.2 Konstruktion von Fasersegmenten und Speicherung der Fasereigenschaften

Beim Algorithmus werden die beim Ausführen von SimPLI angegebenen Fasern in Fasersegmente eingeteilt, welche im Nachfolgenden als *Cones* bezeichnet werden. Dabei besteht ein Cone aus zwei *Spheres*, welche jeweils eine (x, y, z)-Koordinate und einen Radius r besitzen (siehe Abbildung 6a).
Zunächst wird eine Liste mit Spheres und eine Liste mit den Eigenschaften der verschiedenen Schichten der Fasern erzeugt. Dazu wird jede Faser eines Faserbündels betrachtet und die einzelnen Spheres, aus denen die Fasern bestehen, in eine Liste hinzugefügt. Allerdings werden nur Fasern berücksichtigt, welche sich in dem angegebenen Volume of Interest befinden.
Die optischen Eigenschaften für die einzelnen Schichten der Fasern sind innerhalb eines Faserbündels identisch. Aus diesem Grund wird eine Liste angelegt, welche die optischen Eigenschaften der Faserschichten für jedes Faserbündel enthält. In diesem werden für jede Schicht die Ausdehnung der Schicht hinsichtlich des Radius, die Stärke der Doppelbrechung Δn, der Absorptionskoeffizient μ und das Doppelbrechungsmodell hinterlegt (siehe Abbildung 6b).
Anschließend werden aus den ermittelten Spheres Cones erzeugt. Jeweils zwei benachbarte Spheres bilden einen Cone, wobei diese Teil der selben Faser sein müssen. Dadurch erhält man eine Liste, die alle Fasersegmente des Volumens enthält. Diese wird im Nach-



(a) (b)

Abbildung 6: Aufbau einer Faser [7]
(a) Faser als Konstrukt aus Fasersegmenten (Cones), welche jeweils aus zwei Punkten mit jeweiligen Radien (Spheres) gebildet werden
(b) Querschnitt eines Fasersegmentes mit 6 unterschiedlichen Faserschichten.
Axon (rot), Myelinscheide (blau), nicht doppelbrechendes Gewebe (grün)
Jede Schicht kann unterschiedliche Eigenschaften für Δn, μ und die optische Achse besitzen

folgenden zusammen mit einer Liste der Fasereigenschaften bei der Kollisionskontrolle benötigt.

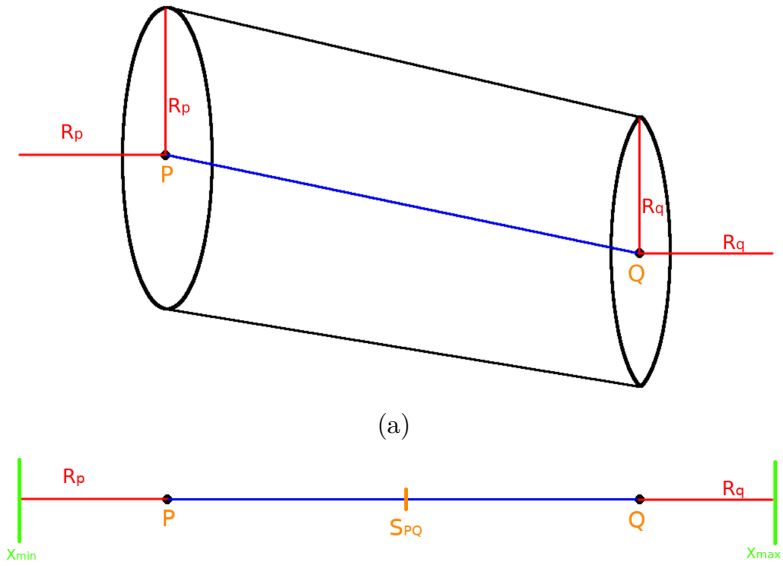
3.3 Ermittlung optischer Eigenschaften einer Koordinate im Volumen

Damit die optischen Eigenschaften an einer beliebigen Position im Volumen ermittelt werden können, wird eine Kollisionskontrolle von der aktuellen Position des Lichtstrahls mit den Cones durchgeführt. Diese wird in eine *Broad-Phase* und eine *Narrow-Phase* eingeteilt. Bei der Broad-Phase werden zunächst nur Fasersegmente ermittelt, die möglicherweise die Position des Lichtstrahles enthalten. In der Narrow-Phase wird überprüft, ob diese Kandidaten tatsächlich mit der betrachteten Position kollidieren.

Sort and Sweep

Für die Durchführung der Broad-Phase wird der *Sort and Sweep*-Algorithmus verwendet [4]. Anhand einer ausgewählten Achse sollen die Cones sortiert und vergleichbar gemacht werden. Als Achse für die Sortierung wurde die X-Achse gewählt, da ein Gewebeschnitt entweder in X oder Y-Richtung die größte Ausdehnung hat und die Varianz der Fasersegmente bezüglich der ausgewählten Achse möglichst groß sein sollte.

Bei Sort and Sweep werden dabei die *Axis-Alligned Bounding Boxes* der Cones betrachtet. Diese stellen jeweils die kleinsten Rechtecke dar, welche die Faserseg-



(a)

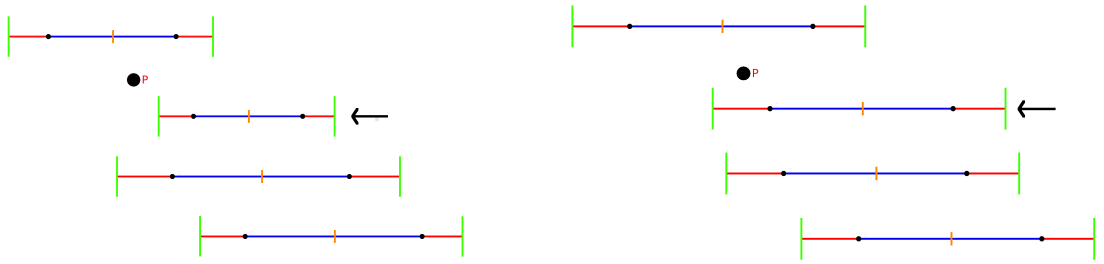
(b)

Abbildung 7: (a) Darstellung eines Cones im Dreidimensionalen, bestehend aus den Spheres P und Q, welche aus einer Position im Raum und jeweils einem Radius gebildet werden.
(b) Darstellung eines Cones projiziert auf die X-Achse. S_{PQ} ist der Schwerpunkt bezüglich der X-Achse. X_{min} und X_{max} sind die Grenzen der Bounding Box $\Delta x = |X_{max} - X_{min}|$.

mente umschließen.

Zunächst sortiert man die Cones, indem man anhand der X-Achse die Schwerpunkte der Cones berechnet und die Schwerpunkte miteinander vergleicht. Der Schwerpunkt ist dabei das arithmetische Mittel zwischen den X-Koordinaten der beiden Spheres, die den Cone bilden. Nun wird in der Liste der Cones nach dem Fasersegment mit der größten Länge bezüglich der X-Achse Δx gesucht (siehe Abbildung 7b). Dieser Wert ist die Größe des Fensters, welches um einen Cone betrachtet werden muss, damit eine mögliche Kollision mit einem Punkt vorliegen kann. Dies ist notwendig, da die alleinige Betrachtung des Schwerpunktes in der sortierten Liste keine genauen Rückschlüsse auf die Ausdehnung eines Cones zulässt. Δx, also die größte Bounding Box bezüglich der X-Achse, wird bei der nachfolgenden Betrachtung als Bounding Box für alle Cones angenommen. Dies ermöglicht das Iterieren durch die sortierte Liste der Cones und das Herausfinden von Grenzen, welche die Cones enthalten, welche potenzielle Kollisionskandidaten einer gegebenen Position sind (siehe Abbildung 8).

Daraufhin wird überprüft, ob Überlappungen der Bounding Boxen der Cones mit der betrachteten X-Position x auftreten. Zunächst wird die Position in der sor-



(a) (b)

Abbildung 8: Darstellung von Bounding Boxen (BB) der Cones bezüglich der X-Achse, welche potenziell mit einem Punkt P kollidieren können.
(a) Die Schwerpunkte lassen keine Rückschlüsse auf die Ausdehnung der ursprünglichen BB zu. Der mit dem Pfeil markierte Cone würde beim Ausführen von Sort and Sweep die Auswahl der Kanditaten nach rechts begrenzen, da dieser den Punkt nicht mehr enthält. Dadurch würde der nachfolgende Cone, welcher den Punkt P enthält, nicht als Kandidat gelten.
(b) Die Vereinheitlichung der BB zu der größten BB der vorhandenen Cones ermöglicht die Anwendung von Sort and Sweep und berücksichtigt jeden potenziellen Kollisionskandidaten.

tierten Liste der Cones gesucht, wo x einsortiert werden müsste. Dies wird mit einer binären Suche realisiert. Ausgehend von dieser Stelle in der sortierten Liste, wird überprüft, ob die Vorgänger x einschließen. Ist die rechte Grenze des Cones größer als x, so schließt dieser die Position ein und ist somit ein potentieller Kollisionskandidat. Es wird so lange durch die Vorgänger iteriert, bis ein Cone auftritt, dessen rechte Grenze x nicht mehr einschließt. Die Position des letzten Cones in der sortierten Liste, der x enthält, wird gespeichert. Danach werden die nachfolgenden Cones der zu betrachtenden Position auf Überlappungen überprüft. Ist die linke Grenze der Cones kleiner als x, so ist dieser ein möglicher Kollisionskandidat. Es wird solange durch die Liste iteriert, bis ein Fasersegment erreicht wird, dessen linke Grenze größer als x ist. Alle Cones zwischen dem letzten Vorgänger von x und dem letzten Nachfolger sind somit mögliche Kollisionskandidaten von x bezüglich der X-Achse (siehe Abbildung 9). Diese Folgerung ist nur möglich, da die Bounding Boxen zuvor angepasst wurden. Ohne eine Vergrößerung dieser, könnte der Fall auftreten, dass ein Cone, welcher die Position einschließt, jedoch in der Liste hinter einem Cone einsortiert ist, der diese nicht umfasst, nicht mehr betrachtet wird (siehe Abbildung 8, schwarzer Pfeil).
Somit wird mit Sort and Sweep eine Liste von Kandidaten ermittelt, die möglicherweise die zu betrachtende Position enthalten. Damit ist die Broad-Phase abgeschlossen.

```
//Sortieren der Cones anhand der Schwerpunkte bezüglich der X-Achse
sortedCones = sort(Cones)
//Größte Bounding Box der Cones bezüglich der X-Achse ermitteln
x_window = maxAABB(Cones)
//Position zum Einfügen von x in sortierte Cones Liste
insertPosition = binary_search(p.x)
//Index des zu überprüfenden Cones
c = insertPosition - 1
//Überprüfe Cones mit Schwerpunkt vor x
Solange x < (sortedCones[c] + x_window/2):
  c -= 1
linkeGrenzeKandidaten = c+1
c = insertPosition
//Überprüfe Cones mit Schwerpunkt nach x
Solange x > (sortedCones[c] - x_window/2):
  c += 1
rechteGrenzeKandidaten = c-1
Rückgabe von linkeGrenzeKandidaten, rechteGrenzeKandidaten
```

Abbildung 9: Broad Phase: Ablauf des Sort and Sweep Algorithmus.

Kollisionen erkennen

Nach Abschluss der Broad-Phase beginnt die Narrow-Phase, in der die zuvor ermittelten Kandidaten auf tatsächliche Kollisionen mit der zu betrachtenden Position kontrolliert werden.

Dazu wird jeweils der Punkt auf der Strecke zwischen den beiden Spheres berechnet, der die kürzeste Distanz zu der Position des Lichtstrahls aufweist. Der kürzeste Abstand bezeichnet die Länge des Lots, welches ausgehend von der Strecke zwischen den beiden Spheres des Cones zum Punkt des Lichtstrahls gefällt wird. Dabei kann es vorkommen, dass ausgehend von der Strecke kein Lot zum Punkt gefällt werden kann, weil der betrachtete Punkt am nächsten zu einem der Spheres liegt. In dem Fall wird die Distanz zwischen der entsprechenden Sphere und der Position des Lichtstrahls bestimmt. An dem so ermittelten Punkt des Cones wird nun der Radius des Fasersegmentes an dieser Stelle berechnet. Es wird angenommen, dass sich der Radius des Fasersegmentes linear zwischen den Spheres annähert, sodass der Radius an einer beliebigen Stelle leicht berechenbar ist. Nun wird auf eine Kollision des Punktes mit dem Cone überprüft, in dem die zuvor ermittelten kürzeste Distanz mit dem Radius des Cones verglichen wird. Ist die Distanz kleiner als der Radius, so ist der betrachtete Punkt in dem Fasersegment und kollidiert. Ist diese größer, so findet keine Kollision statt (siehe Abbildung 10). Die Kollisionsüberprüfung wird für alle Kandidaten durchgeführt. Da die Position des Lichtstrahls in mehreren Cones enthalten sein kann, wird der Cone ausgewählt,

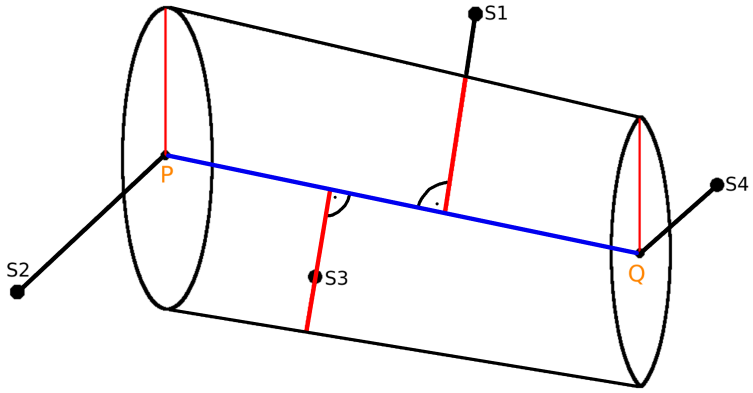


Abbildung 10: Beispiele für die Ermittlung der kürzesten Distanz von einem Punkt zu einem Cone
S1: Die Distanz zum Cone ist größer als der Radius (rot) an der Stelle des gefällten Lotes, wodurch S1 außerhalb des Cones liegt.
S2: Lot fällen nicht möglich, daher wird die Distanz zu Sphere P des Cones ermittelt. Der Radius von der Sphere P ist kleiner als die Distanz zu S2, wodurch der Punkt außerhalb des Cones liegt.
S3: Die Distanz zum Cone ist kleiner als der Radius (rot) an der Stelle des gefällten Lotes, wodurch S3 innerhalb des Cones liegt.
S4: Lot fällen nicht möglich, daher wird die Distanz zu Sphere Q des Cones ermittelt. Der Radius von der Sphere P ist größer als die Distanz zu S4, wodurch der Punkt innerhalb des Cones liegt.

welcher den kürzesten Abstand zum Lichtstrahl aufweist. Da mit euklidischen Distanzen gerechnet wird und der Aufruf der Wurzelfunktion Rechenungenauigkeiten verursacht und zusätzlichen Aufwand bedeutet, wird jeweils mit dem Quadrat der verwendeten Längen (Abstand, Radius) gerechnet. Dabei gehen keine Relationen verloren.

Ermittlung der Optischen Eigenschaften

Zuletzt wird beim nächsten Cone geprüft, in welcher Faserschicht die Lichtposition vorliegt. Dazu werden die einzelnen Ausdehnungen der Faserschichten betrachtet und diese mit dem Abstand des Punktes verglichen (siehe Abbildung 6b). Nun können die optischen Eigenschaften der Schicht aus den zuvor ausgelesenen Fasereigenschaften entnommen werden (siehe Kapitel 3.2). Dazu zählen die Stärke der Doppelbrechung Δn, der Absorptionskoeffizient μ und das Simulationsmodell für die Doppelbrechung. Auf Basis des Doppelbrechungsmodells wird nun ein dreidimensionaler Vektor bestimmt, der die Doppelbrechung der Faser aufgrund ihrer Orientierung beschreibt (siehe Abbildung 2a). Ist das Doppelbrechungsmodell *None*, so ist dies ein Null-Vektor, da das Gewebe an dieser Stelle nicht doppelbrechend ist. Bei dem *radialen* Doppelbrechungsmodell wird der Vektor mit dem kürzesten Abstand zum Fasersegment zurückgegeben. Ansonsten wird beim *parallelen* Dop-

pelbrechungsmodell der Vektor vom ersten Sphere des Fasersegmentes zum zweiten Sphere zurückgegeben. Dies entspricht der örtlichen Lage.
Somit werden die Stärke der Doppelbrechung Δn, der Absorptionskoeffizient μ und ein dreidimensionaler Vektor zur Beschreibung der Doppelbrechung ermittelt, welche die optischen Eigenschaften des Gewebes an der Position eines Lichtstrahls beschreiben.

3.4 Berechnungen der Lichtstrahlen

Ein Lichtstrahl wird durch eine örtliche Position (x, y, z) und den vierdimensionalen Stokes-Vektor für die Beschreibung des Polarisationszustandes dargestellt. Nachdem die Lichtstrahlen bereits den ersten Polarisationsfilter und die Verzögerungsplatte passiert haben, wurde der ursprüngliche Stokes-Vektor verändert. Ausgehend von einer Startposition am Beginn des unteren Ende des Volumens und einer Schrittrichtung, welche zuvor für jeden Lichtstrahl berechnet wurden, werden die Lichtstrahlen durch das Volumen ausgesendet. Die Schrittrichtung ist für jeden Lichtstrahl gleich und hängt genauso von der Ausrichtung des Volumens ab, wie die Startposition. Ist das Gewebe bei der Messung gekippt, so verschieben sich die Koordinaten der Startpositionen und die Schrittrichtung im Gewebe.
Die Lichtstrahlen durchlaufen mit einer zuvor angegebenen Schrittweite das Gewebe. Dabei werden nach jeder Iteration an der aktuellen Position die optischen Eigenschaften des Volumens wie in Kapitel 3.3 ermittelt. Die so erhaltenen Parameter Δn, μ und der dreidimensionale Vektor zur Beschreibung der Doppelbrechung der Faser aufgrund ihrer Orientierung, werden im Anschluss mit dem Stokes-Vektor verrechnet. Dazu wird zunächst die Abschwächung aus dem Absorptionskoeffizient und der Dicke des Gewebes berechnet. Ist an der aktuellen Position Gewebe vorhanden, so wird die Abschwächung mit dem Stokes-Vektor multipliziert. Außerdem wird bei gekippten Gewebe der Vektor für die Doppelbrechung mithilfe einer Rotationsmatrix transformiert. Dies liegt daran, dass in der Simulation im Gegensatz zum realen Messaufbau nicht das Gewebe gekippt, sondern die Lichtstrahlen transformiert werden. Das Volumen bleibt somit unverändert, wohingegen der Einfallswinkel der Lichtstrahlen und dessen Verlauf dem Tilting (Kippen des Gewebes) entsprechen.
Anschließend werden die einzelnen Parameter der Müller-Matrix auf Basis des zuvor berechneten dreidimensionalen Vektors zur Beschreibung der Doppelbrechung, aufgrund der Faserorientierung im Gewebe, ermittelt. Dazu wird unter Anderem die Retardierung approximiert (siehe (1)). Die Multiplikation des Stokes-Vektors mit der in Kugelkoor-

dinaten umgerechneten Müller-Matrix entspricht so der Polarisationsveränderung des Lichtstrahls nach Passieren des Gewebes an der aktuellen Position.
Die Lichtstrahl iterieren mit der Schrittweite solange durch die verschiedenen Lichtpositionen, bis diese das Volumen verlassen. Ist dies der Fall, so wird jeder Lichtstrahl mit der Müller-Matrix für den Analysator multipliziert. Abschließend wird für jeden Lichtstrahl die erste Komponente ausgelesen, welche der verbleibenden Lichtintensität entspricht.

3.5 GPU Parallelisierung des Algorithmus

3.5.1 Anwendbarkeit der Parallelisierung

Der zuvor vorgestellte Algorithmus für die Durchführung der Simulation SimPLI basierend auf Ray Tracing mit Kollisionskontrolle, eignet sich zur Parallelisierung auf der GPU.
Da die für jeden Lichtstrahl auszuführenden Berechnungen von einem einzelnen GPU-Thread in einer sinnvollen Zeit abgearbeitet werden können, kann man die Berechnungen für einen einzelnen Lichtstrahl jeweils von einem GPU-Thread durchführen lassen. Des Weiteren müssen für jeden Lichtstrahl die selben Operationen mit unterschiedlichen Daten durchgeführt werden. Diese sind unabhängig voneinander, sodass keine *Race-Conditions* zwischen den Threads auftreten und so unerwartete Ergebnisse produzieren. Da schon bei einer simulierten Messung mit einer 240×240 CCD-Sensoren großen Kamera 57.600 Lichtstrahlen simuliert werden, ist die Anzahl der gleichartigen Teilprobleme gegeben. Bei einem größeren Versuchsaufbau würde sich diese Anzahl nochmal deutlich steigern.
Des Weiteren können die Operationen, welche für die Kollisionskontrollen notwendig sind auch parallelisiert werden. So ist das Durchführen von Listenoperation wie Maxima zu finden oder zu sortieren auch auf einer GPU realisierbar.

3.5.2 Implementierung auf der GPU mit CUDA

Der entwickelte Algorithmus wurde mit der Programmiersprache C++ und der Programmierschnittstelle CUDA implementiert. Zusätzlich wurde die Bibliothek Thrust verwendet.
Zunächst werden auf dem Host alle Parameter bereitgestellt, welche im weiteren Verlauf benötigt werden. Dazu zählen beispielsweise die Matrizen für die Polarisationsfilter,

die Verzögerungsplatte und die Rotationen sowie die anfänglichen Stokes-Vektoren für die Lichtstrahlen und die Startpositionen. Außerdem werden die Eigenschaften der Faserschichten und die Spheres der Fasern jeweils in einen Vektor hinterlegt. Die Spheres werden dabei als Struct behandelt, welches eine Position und einen Radius in Form eines *float4*, einen Faserbündelindex und einen Faserindex besitzt. Nachfolgend wird ein Vektor erstellt, der die Spheres zu passenden Cones zusammenfügt. Ein Cone wird als Struct behandelt, welches einen Pointer auf den Vektor mit Spheres hat und die Position der ersten Sphere des zu modellierenden Cones enthält. Dies reicht aus, da jeweils zwei nachfolgende Spheres im Vektor jeweils einen Cone bilden. Zuvor wird jedoch überprüft, ob diese zur selben Faser gehören.
Da im Device-Code nicht auf Host-Speicher, sondern nur auf den Speicher der GPU zugegriffen werden kann, müssen die Daten auf die GPU kopiert werden. Dazu werden die standard C++ Vektoren zunächst einem Thrust Host-Vector zugewiesen. Die anschließende Zuweisung zu einem Thrust Device-Vector sorgt dafür, dass die Bibliothek selbständig Speicher auf der GPU allokiert und die Daten kopiert. Zu den Daten zahlen sowohl die Spheres, die Eigenschaften der Faserschichten, die Cones, die Stokes-Vektoren der Lichtstrahlen als auch die Matrizen für den Analysator und die Rotationen. Die Stokes-Vektoren wurden zuvor schon mit den Matrizen für den ersten Polarisationsfilter und der Verzögerungsplatte multipliziert.
Nach den Vorbereitungen können nun Berechnungen auf der GPU durchgeführt werden. Die für die Durchführung von Sort and Sweep benötigte Sortierung des Cones-Vektors werden mithilfe der von Thrust bereitgestellten Sortierfunktion realisiert. Dazu wird der Sortierfunktion eine Ordnung übergeben, welche zwei Cones bezüglich des Schwerpunktes auf der X-Achse vergleicht. Ist die X-Koordinate eines Schwerpunkt eines Cones kleiner als die eines anderen, so wird dieser weiter vorne einsortiert. Als Ergebnis der Sortierfunktion erhält man einen Vektor der die Indizes der Cones in der Reihenfolge enthält, sodass diese aufsteigend sortiert sind. Da die Operation auf einem Device-Vektor arbeitet, wird diese von Thrust parallel auf der GPU ausgeführt.
Danach wird die maximale Bounding Box eines Cones entlang der X-Achse gesucht. Dazu wird die *max_element*-Funktion von Thrust verwendet. Als Kriterium wird die Bounding-Box bezüglich der X-Achse von zwei zu vergleichenden Cones berechnet und die jeweils größte zurückgegeben. Auch diese Funktion wird von Thrust parallel auf der GPU ausgeführt und gibt so die Position des Cones mit dem größten Δx zurück.

Nachdem die für Sort and Sweep benötigten Vorbereitungen getroffen wurden, werden nun die Berechnungen für die einzelnen Lichtstrahlen durchgeführt. Es wird eine Kernel-

Funktion aufgerufen, welche die Berechnungen für die einzelnen Threads enthält. Sofern die GPU genügend Threads zur Verfügung stellen kann, sollen die Berechnungen für einen Lichtstrahl in der Simulation von einem Thread ausgeführt werden. Ist die Anzahl der zu berechnenden Lichtstrahlen jedoch größer als die Anzahl der zur Verfügung stehenden Threads, wird mithilfe einer *Grid-Stride-Loop* sichergestellt, dass ein Thread auch mehrere Lichtstrahlen nacheinander abarbeiten kann. Dabei wird die Arbeitslast möglichst gleichmäßig verteilt.
Jeder Thread iteriert mit einer vorgegebenen Schrittweite durch die verschiedenen Positionen des Lichtstrahls im Volumen. An jeder Stelle wird eine *Device-Funktion* aufgerufen, welche die optischen Eigenschaften des Volumens an dieser Stelle ermittelt. Diese Funktion führt eine Kollisionskontrolle mit dem Sort and Sweep-Algorithmus durch, um zunächst mögliche Cones zu ermitteln, die kollidieren könnten. Diese Kandidaten werden im Nachfolgenden auf den kürzesten Abstand zur aktuellen Position untersucht. Ist der Abstand kleiner als der Radius des Cones, so liegt eine Kollision vor. Bei dem Cone mit dem kürzesten Abstand wird die genaue Faserschicht ermittelt und dessen Eigenschaften ermittelt. Je nach Simulationsmodell wird nun die doppelbrechende Wirkung des Gewebes, sowie die Absorption und Stärke der Doppelbrechung zurückgegeben. Im Kernel werden diese wie in Kapitel 3.4 beschrieben nun mit dem Stokes-Vektor des Lichtstrahl verrechnet. Dieser Ablauf wiederholt sich so lange, bis das Ende vom Volumen erreicht wird. Zuletzt wird noch die Wirkung des Analysators berücksichtigt und die letztlich gemessene Lichtintensität gespeichert. Jeder Lichtstrahl führt die Berechnungen an den Position für alle möglichen Rotationen des Gewebes durch. Die abschließend gemessenen Lichtintensitäten für jeden Lichtstrahl unter Beachtung der Rotation des Gewebes werden vom Speicher der GPU in den Hauptspeicher kopiert. Anschließend können diese von der Simulation weiterverarbeitet werden.

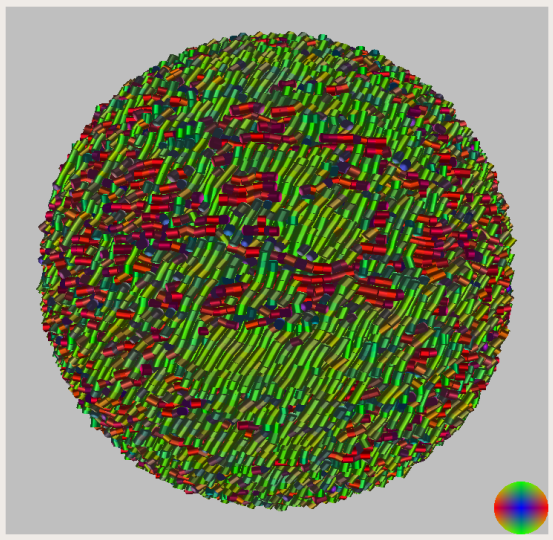


Abbildung 11: Ein synthetisches kugelförmiges Volumen mit zwei kreuzenden Faserbündeln dient als Gewebe. Insgesamt werden 2.632 Fasern betrachtet, wobei 30% zum ersten Faserbündel gehören. Somit sind 115.784 Spheres und 113.152 Cones vorhanden.

4 Ergebnisse

Zur Validierung des neu entwickelten Algorithmus (GPUSimPLI) wird ein Vergleich mit den Ergebnissen von SimPLI durchgeführt und die Laufzeit sowie die Speichernutzung untersucht. Dazu wird ein kugelförmiges Volumen mit kreuzenden Fasern betrachtet (siehe Abbildung 11). Die Fasern der Faserbündel bestehen jeweils aus drei unterschiedlichen Faserschichten, welche von innen nach außen betrachtet folgende Eigenschaften besitzen:

Layer	**Anteil am Faserradius**	Δn	μ	**Doppelbrechungsmodell**
1	0.333	-0.004	10	Parallel
2	0.666	0	5	Nicht doppelbrechend
3	1	0.004	1	Radial

Tabelle 1: Eigenschaften der simulierten Faserschichten

Das betrachtete *Volume of Interest* stellt einen $120 \times 120 \times 120$ µm großen Würfel dar, in dessen Mitte das kugelförmige Volumen mit Fasern angeordnet ist. Je nach angegebener Voxelgröße, wird die Anzahl der auszusendenden Lichtstrahlen bestimmt. Die Einteilung in Voxel findet nur beim ursprünglichen SimPLI statt und dient nur als Koordinatensystem für die Anordnung der Fasern im Volumen und die Anzahl der CCD-Sensoren. Bei jeder Messung werden insgesamt 6 Rotationen betrachtet, die in äquidistanten 30°-

Schritten von 0° bis 150° durchgeführt werden. Außerdem wird das Gewebe in der Ebene in jede Richtung um 5.5° gekippt, sodass eine ungekippte Messung und jeweils 4 gekippte Messungen simuliert werden.
Die Tests wurden auf einem Rechner mit einer Intel Core i7-8700K 3.70GHz CPU, 16 GB Arbeitsspeicher und einer Nvidia GeForce GTX 1080 unter dem Betriebssystem Ubuntu 18.04 durchgeführt.

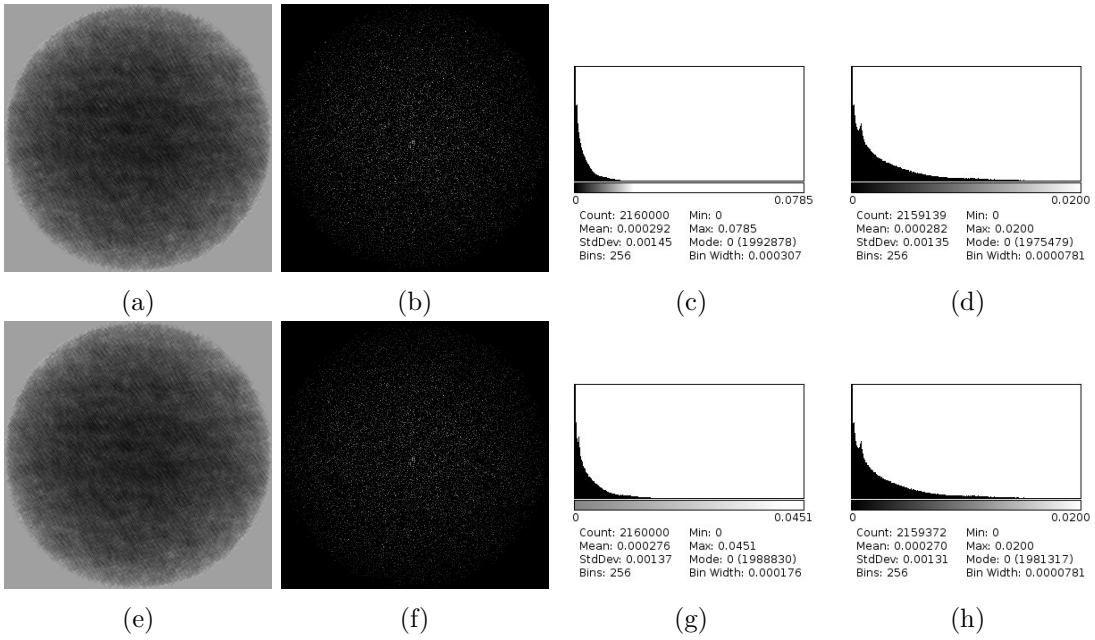


Abbildung 12: Unterschiede zwischen SimPLI und GPUSimPLI je nach implementierter Auswahl der Cones (a-d) ursprüngliche Implementierung, (e-h) angepasste Implementierung
Auf der linken Seite ist eine ungekippte Messung von SimPLI mit einer Voxelsize von 0,2 µm mit 0° Rotation aufgeführt. (a) zeigt das Ergebnis der ursprünglichen Implementierung von SimPLI und (e) zeigt das Ergebnis der abgeänderten Implementierung, bei der im Falle von gleich nahen Cones, der Cone, mit dem größten Schwerpunkt bezüglich der X-Achse ausgewählt wird. Dies entspricht der Auswahl der Cones bei GPUSimPLI.
Daneben sind in (b) und (g) die relativen Abweichungen zu einer GPUSimPLI Messung mit den selben Parametern aufgeführt.
Die zugehörigen Histogramme für die relative Abweichung zu GPUSimPLI sind in (c) und (g) aufgeführt.
(d) und (h) sind Ausschnitte aus den Histogrammen im Bereich $[0.0, 0.02]$.
Die Colorbar für die relative Abweichung lautet $[0.0, 0.02]$. Jeder komplett weiße Punkt entspricht einer Abweichung von mindestens 2%.

4.1 Verifizierung der Ergebnisse

Zunächst werden die ungekippten Messungen miteinander verglichen. Dabei sollten die gleichen Messergebnisse generiert werden, da beim neuen Algorithmus wie beim diskretisierten SimPLI jeweils die gleichen Positionen mit den Lichtstrahlen abgeschritten werden und diese jeweils in der Mitte des diskretisierten Volumens liegen. Somit werden an den gleichen Positionen im Gewebe die optischen Eigenschaften bestimmt.
Bei der Ermittlung der optischen Eigenschaften wird wie in Kapitel 2.2.3 beschrieben jeweils das Fasersegment (Cone) mit der kürzesten Distanz zum betrachteten Punkt berücksichtigt. Dabei kann die kürzeste Distanz eines Cones der Abstand zu einer Sphere sein (siehe Abbildung 10). Da eine Sphere jeweils zwei Cones zugeordnet werden kann, ist die Bestimmung des nächsten Cones nicht eindeutig. Während beim GPUSimPLI

immer der Cone ausgewählt wird, welcher den größten Schwerpunkt bezüglich der X-Achse aufweist, werden bei SimPLI die einzelnen Cones so durchlaufen, wie sie in der Liste der übergebenen Faserbündeln vorkommen und der letzte Cone, der die Position enthält zurückgegeben. Dadurch werden von den Programmen an manchen Positionen unterschiedliche optische Eigenschaften ermittelt. Um diese unterschiedlichen Entscheidungen zu vermeiden, wird im Nachfolgenden jeweils eine Version von SimPLI verwendet, welche auch jeweils den Cone mit dem größten Schwerpunkt bezüglich der X-Achse auswählt, wenn mehrere Cones die selbe Distanz zu einer betrachteten Position haben.
Durch die Anpassung der Auswahl der gleichen Cones zur Bestimmung der optischen Eigenschaften nähern sich die Ergebnisse von SimPLI und GPUSimPLI weiter an. Zwar ist die Veränderung an der gesamten Messung mit durchschnittlich 0.0016% und einer Verringerung der Standardabweichung um 0,008% sehr gering, da der Sonderfall, dass mehrere Cones im gleichen Maße als Kandidat für die Bestimmung der optischen Eigenschaften verwendet werden, nicht sehr häufig auftritt, jedoch kann das Auftreten von größeren Ausreißern bei der abschließend gemessenen Intensität verringert werden. So treten höchstens Abweichungen von 4,51% im Gegensatz zu Abweichungen von 7,85% auf (siehe Abbildung 12).

Bei ungekippten Messungen mit einer anfänglichen Lichtintensität von 26.000 und durchschnittlichen Lichtintensitäten von circa 3.726 sind die Abweichungen zwischen den Simulationen gering. So beträgt bei einer Voxelsize von 0,5 µm, mit insgesamt 57.600 Lichtstrahlen pro Rotation, also 345.600 gemessenen Intensitäten pro Messung, die relative Abweichung je gemessener Intensität durchschnittlich nur 0,00161% und die Standardabweichung 0,0451%. Es sind zwar wenige Ausreißer mit bis zu 4,77% Abweichung vorhanden, welche aber nicht ins Gewicht fallen, da die meisten Abweichungen wesentlich kleiner als 1% sind (siehe Abbildung 13b).
Wird die Voxelsize auf 0,25 µm halbiert, so werden 230.400 Lichtstrahlen pro Rotation, also 1.382.400 Intensitäten gemessen. Wie zuvor ist die durchschnittliche Abweichung mit 0,00162% und die Standardabweichung mit 0,035% gering. Durch die größere Anzahl der gemessenen Intensitäten steigt auch die Anzahl der Abweichungen und die Anzahl der Ausreißer (siehe Abbildung 13e).
Mit einer Voxelsize von 0,2 µm erhöht sich die Anzahl der Lichtstrahlen pro Rotation auf 360.000 Lichtstrahlen und somit 2.160.000 gemessenen Intensitäten. Die durchschnittliche Abweichung weist weiterhin mit 0,0276% einen geringen Wert auf, wobei sich die Standardabweichung auf 0,137% erhöht hat. Anhand der Messbilder ist nun auch deutlich zu erkennen, dass mehrere Unterschiede zwischen den Programmen erkennbar sind.

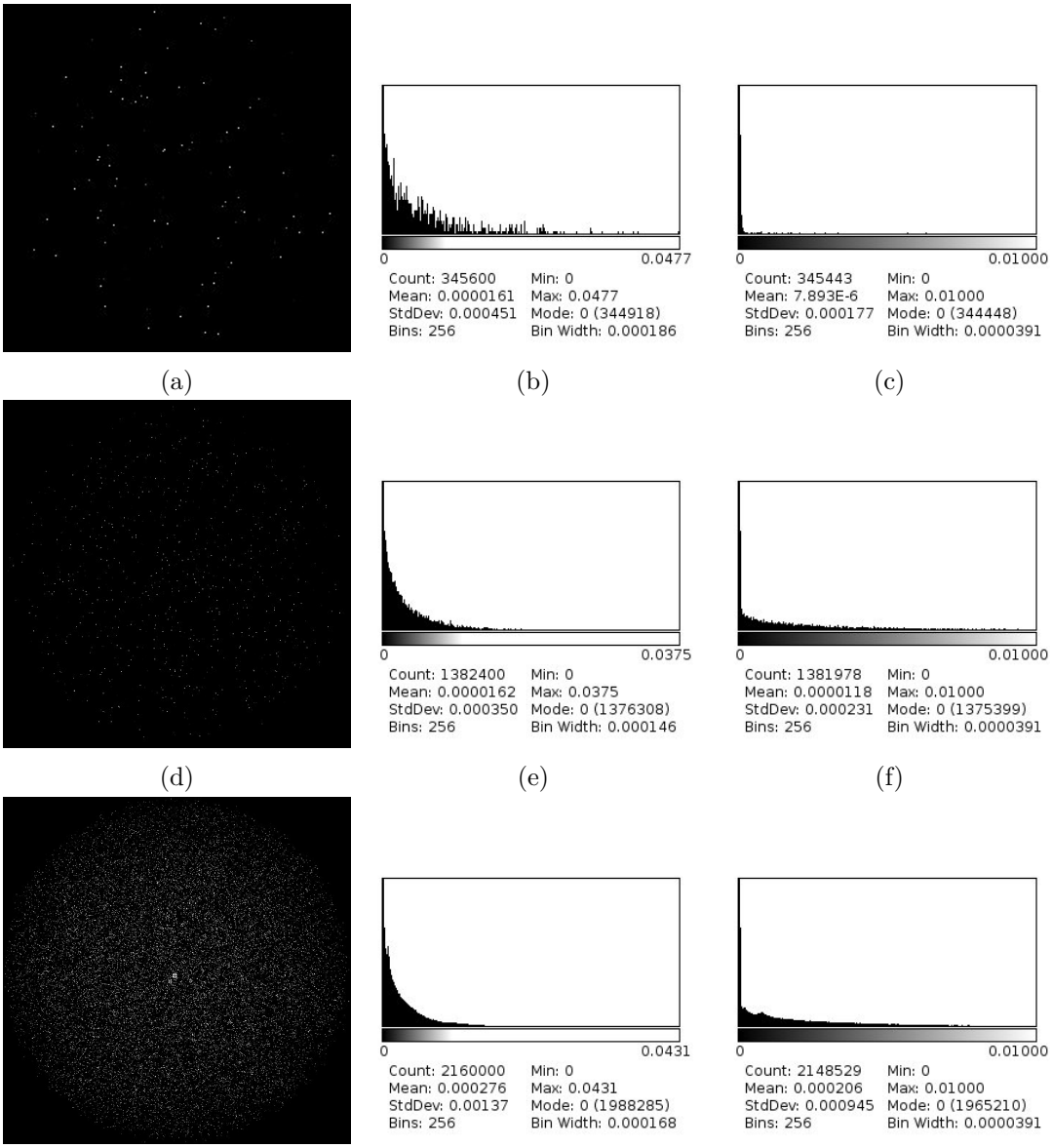


Abbildung 13: Vergleich von ungekippten Messungen zwischen GPUSimPLI und SimPLI mit Colorbar $[0.0, 0.01]$
Auf der linken Seite sieht man die relative Abweichung des Vergleichs von ungekippten Messungen zwischen GPUSimpli und Simpli mit 0° Rotation und unterschiedlicher Voxelsize (a)0,5 µm, (d)0,25 µm und (g)0,2 µm, eingefärbt mit der Colorbar $[0.0, 0.01]$. Jeder komplett weiße Punkt entspricht einer Abweichung von mindestens 1%.
In der Mitte sind die zugehörigen Histogramme für die relativen Abweichung der Voxelsize (b)0,5 µm, (e)0,25 µm und (h)0,2 µm für alle 6 Rotationen gegeben.
Auf der rechten Seite sind Ausschnitte im Bereich $[0.0, 0.01]$ der Histogramme für die relativen Abweichung der Voxelsize (b)0,5 µm, (e)0,25 µm und (h)0,2 µm für alle 6 Rotationen gegeben.

Dennoch sind die gemessenen Unterschiede weiterhin gering (siehe Abbildung 12g). Generell erhöht sich mit der Anzahl der gemessenen Intensitäten die Anzahl der Abweichungen. Da mit einer Verkleinerung der Voxelsize und somit auch einer Verkleinerung der Schrittweite der Lichtstrahlen mehr Berechnungen durchgeführt werden, sind diese anfälliger für Rechenungenauigkeiten. Treten bei der Ermittlung der optischen Eigenschaften float-Ungenauigkeiten auf, so setzen sich die Fehler bei den nachfolgenden Multiplikationen weiter fort. Die Abweichungen sind dabei vermutlich auf die Verwendung von *single precision* (32 Bit) bei der GPU Implementierung im Gegensatz zu der 64 Bit *double precision* der CPU zurückzuführen. Dennoch sind die betrachteten Auswirkungen vergleichsweise gering, da die Intensitätswerte kaum um mehr als 1% abweichen.

Bei gekippten Messungen ist zwar der Verlauf der Lichtstrahlen der beiden Programme gleich, jedoch wird bei GPUSimPLI für jede erreichte Position eine Ermittlung der optischen Eigenschaften mithilfe einer Kollisionskontrolle durchgeführt. Im Gegensatz dazu wird bei SimPLI überprüft, in welchem Voxel die erreichte Position liegt und welche Eigenschaften zuvor für den Mittelpunkt des Voxels errechnet wurden. Dabei können Diskretisierungsfehler auftreten und Eigenschaften für das Gewebe zurückgegeben werden, welche an der aktuellen Position nicht vorliegen (siehe Abbildung 4).
Als Beispiel für die gekippten Messungen wird die simulierte Messung eines Gewebes betrachtet, welches um 5,5° in der Ebene nach oben gekippt wurde. Dabei wurden wie bei der nicht gekippten Messungen unterschiedliche Voxelgrößen untersucht.
Bei einer Voxelsize von 0,5 µm beträgt die relative Abweichung der Intensität durchschnittlich 5,04% und die Standardabweichung 6,64%, wodurch deutliche Abweichungen erkennbar sind (siehe Abbildung 14b). Diese treten an Stellen auf, an denen Lichtstrahlen Gewebe passieren. Außerdem sind 6.943 Ausreißer über 25 % erkennbar (siehe Abbildung 14c). Nach dem Halbieren der Voxelsize auf 0,25 µm bleiben weiterhin viele Abweichungen erkennbar, jedoch hat sich die durchschnittliche Abweichung auf 2,63% und die Standardabweichung fast auf 3,48% halbiert. Die maximalen Ausreißer weisen geringere Werte auf und der Anteil der Abweichungen unter 25% steigt (siehe Abbildung 14e). Die Verkleinerung der Voxelsize auf 0,2 µm setzt den zuvor erkennbaren Trend fort. Die Beträge der Ausreißer verringern sich und die Standardabweichung nimmt weiterhin auf 2,77% ab. Auch die durchschnittliche Abweichung beträgt so 2,1%. Wie bei den anderen Differenzbildern ähnelt die Verteilung der Abweichungen einer Normalverteilung (siehe Abbildung 14h). Dies könnte daran liegen, dass durch die Diskretisierungsfehler bei SimPLI an manchen Stellen im Gewebe eine falsche Klassifizierung von Hintergrund durchgeführt wird. Dadurch kann in einigen Fällen fälschlicher Weise das Gewebe als

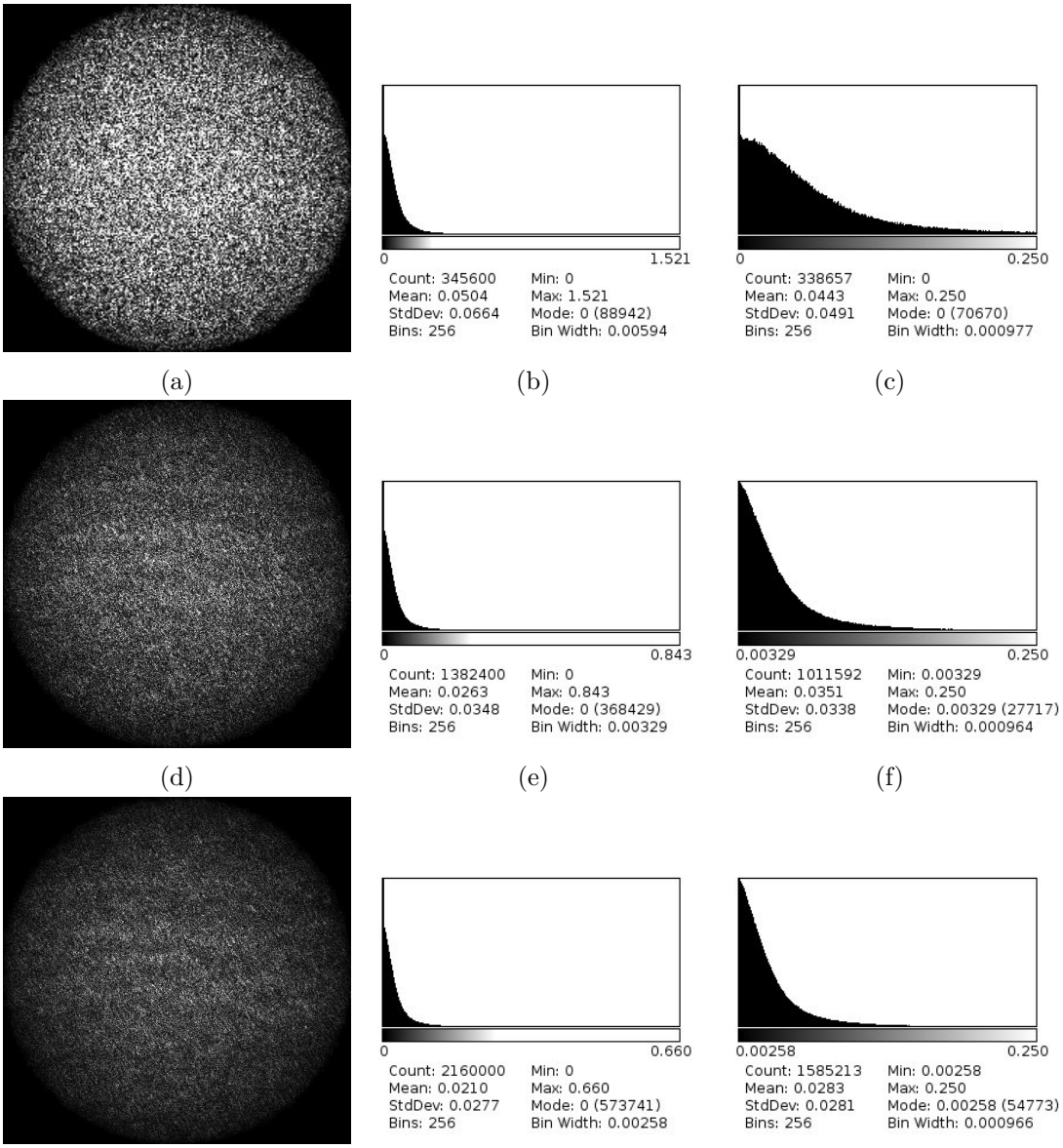


Abbildung 14: Vergleich von nach oben gekippten Messungen zwischen GPUSimPLI und SimPLI mit Colorbar $[0.0, 0.25]$
Auf der linken Seite sieht man die relative Abweichung des Vergleichs von nach oben gekippten Messungen zwischen GPUSimpli und Simpli mit 0° Rotation und unterschiedlicher Voxelsize (a)0,5 µm, (d)0,25 µm und (g)0,2 µm, eingefärbt mit der Colorbar $[0.0, 0.25]$. Jeder komplett weiße Punkt entspricht einer Abweichung von mindestens 25%.
In der Mitte sind die zugehörigen Histogramme für die relativen Abweichung der Voxelsize (b)0,5 µm, (e)0,25 µm und (h)0,2 µm für alle 6 Rotationen gegeben.
Auf der rechten Seite sind Ausschnitte im Bereich $[0.0, 0.25]$ der Histogramme für die relativen Abweichung der Voxelsize (b)0,5 µm, (e)0,25 µm und (h)0,2 µm für alle 6 Rotationen gegeben.

Hintergrund klassifiziert werden, wobei in anderen Fällen Hintergrund fälschlicher Weise als doppelbrechende Faserschicht klassifiziert wird. Dadurch können sowohl Effekte auftreten, die eine negative Differenz der Messungen von SimPLI zu GPUSimPLI aufweisen oder auch eine positive. Da die Lichtstrahlen parallel durch das Gewebe verlaufen, bei der Kippung die einzelnen betrachteten Position jedoch in der Z-Achse versetzt sind, können je nach Verlauf durch das Gewebe solche Effekte entstehen. Je kleiner die Voxelsize gewählt wird, desto geringer werden die Diskretisierungsfehler von SimPLI und die gemessenen Intensitäten nähern sich den Ergebnissen von GPUSimPLI an.

4.2 Performanzanalyse

Da die implementierten Verfahren unterschiedliche Anzahlen von Berechnungen durchführen, sind die Laufzeiten von GPUSimPLI und SimPLI unterschiedlich.
Bei SimPLI werden zu Beginn des Programms einmal die Bounding Boxen von allen vorhandenen Fasersegmente durchlaufen, um die optischen Eigenschaften der Mittelpunkte aller Subvoxel des Volumens zu bestimmen. Dabei wird der Abstand der Cones zu den betrachteten Punkten bestimmt und auf eine Kollision überprüft. Wurden die Eigenschaften einmal bestimmt, so werden nur noch die Berechnungen für alle Lichtstrahlen der Rotationen und einzelnen Tilts durchgeführt.
Bei GPUSimPLI werden hingegen für jede erreichte Position eines Lichtstrahls zunächst die möglichen Kandidaten für eine Kollision berechnet und anschließend für jeden von diesen eine Kollisionskontrolle durchgeführt. Daher werden die Berechnungen für die optischen Eigenschaften wesentlich häufiger ausgeführt und müssen auch im Gegensatz zu SimPLI bei jeder Kippung wieder ausgeführt werden. In der Tabelle 2 sind die Anzahlen der Aufrufe für die Kollisionskontrollen eines betrachteten Punktes bei den beiden Implementierungen für unterschiedliche Voxelsizes aufgeführt.
Es ist erkennbar, dass bei GPUSimPLI wesentlich häufiger Kollisionskontrollen durchgeführt werden. Bei einer Voxelsize von 0,5 µm und somit einer Stepsize von 0,5 µm sind dies mit 1.457.204.160 circa 55-mal mehr Kollisionskontrollen als bei SimPLI. Mit klei-

Voxelsize	0,5 µm	0,25 µm	0,2 µm	0,125 µm
SimPLI	26.543.512	187.371.663	356.637.342	
GPUSimPLI	1.457.204.160	5.829.726.720	9.107.260.800	23.316.603.840

Tabelle 2: Vergleich der Anzahl der Kollisionskontrollen bei unterschiedlicher Voxelsize.
Die Berechnungen mit 0,125 µm waren für SimPLI auf dem Testrechner nicht möglich, da nur 16 GB zur Verfügung standen (siehe Tabelle 4).

ner werdender Voxelsize verringert sich zwar das Verhältnis zwischen der Anzahl der Kollisionskontrollen bei 0,25 µm auf den Faktor 25, jedoch ist die absolute Anzahl an Kollisionskontrollen mit 9.107.260.800 immer noch deutlich höher.
Aufgrund der unterschiedlichen Anzahl an Kollisionskontrollen weicht die Ausführungszeit der beiden Implementierung deutlich voneinander ab. Während bei einer Voxelsize von 0,25 µm die Ausführungszeit bei SimPLI circa 58 Sekunden beträgt, benötigt GPUSimPLI circa 400 Sekunden. Auch bei einer Voxelsize von 0,2 µm ist die CPU Implementierung mit 117 Sekunden circa 7-mal schneller als die GPU Implementierung mit 820 Sekunden. Dabei ist jedoch zu beachten, dass die GPU circa 25-mal mehr Kollisionskontrollen durchführt und die Ermittlung der Kollisionskandidaten dabei nicht eingeschlossen ist.
Die Laufzeit der GPU Implementierung setzt sich hauptsächlich aus der Ausführung des Kernels zusammen. So dauert das Konstruieren und Sortieren der Cones bei dem betrachteten Beispiel mit 115.784 Cones nur 1,6354 Sekunden. Bei einer Voxelsize von 0,2 µm dauert das Ausführen des Kernels, der für eine Kippung die Messung simuliert, circa 126 Sekunden. Somit werden 810 Sekunden der 820 Sekunden Gesamtlaufzeit für die Berechnung des Kernels aufgewendet, da in dem Beispiel 5 Kippungen durchgeführt werden. Die Laufzeit des Kernels resultiert dabei aus der hohen Anzahl an Kollisionskontrollen.

Im Gegensatz zu der erhöhten Laufzeit wird bei GPUSimPLI wesentlich weniger Speicher benötigt. Bei SimPLI wird wegen der Diskretisierung für jeden Subvoxel die Stärke der Doppelbrechung, der Absorptionskoeffizient und ein Vektor für die Wirkung der Doppelbrechung aufgrund der Orientierung der Faser gespeichert. Dies entspricht einer Speichermenge pro Voxel von 5 32-bit floats. Wie in der folgenden Tabelle zu sehen wächst die Speichermenge damit kubisch (siehe Tabelle 4).
Da bei GPUSimPLI keine Diskretisierung in Subvoxel stattfindet, sondern die optischen Eigenschaften an einer Position bestimmt und anschließend direkt mit dem zugehörigen Lichtstrahl verrechnet werden, wird dieser Speicher nicht benötigt. Stattdessen müssen die Fasersegmente und Eigenschaften auf die GPU kopiert werden. Zusammen mit dem

Voxelsize	**Laufzeit SimPLI**	**Laufzeit GPUSimPLI**
0,5 µm	8,46 s	56 s
0,25 µm	58 s	400 s
0,2 µm	117 s	820 s
0,125 µm		3200 s

Tabelle 3: Laufzeit bei SimPLI und GPUSimPLI bei unterschiedlicher Voxelsize.

benötigten Speicher für die Berechnungen werden bei einer Voxelsize von 0,2 µm und 2.632 Fasern so nur circa 75 MB auf der GPU allokiert. Soll ein Volumen mit einer Voxelsize von 0,125 µm bei SimPLI für die Messung verwendet werden, so werden 16.875 MB Speicher für die Speicherung der optischen Eigenschaften der Subvoxel benötigt. Da auf dem Testrechner nur maximal 16 GB Hautspeicher vorhanden waren, konnte die Messung nicht durchgeführt werden. Stattdessen konnte GPUSimPLI ausgeführt werden und benötigte dabei nur 180 MB auf der GPU.

4.3 Diskussion

Die mit GPUSimPLI ermittelten Ergebnisse entsprechen bei ungekippten Messungen annähernd den Ergebnissen von SimPLI. Dabei treten aufgrund von Rechenungenauigkeiten teilweise Abweichungen auf. Im Verhältnis zu den gemessenen Intensitätswerten sind diese jedoch sehr gering und scheinen damit für die Simulation von 3D-PLI Messungen geeignet zu sein. Die Übereinstimmung der ungekippten Messungen weist dabei auf eine korrekte Implementierung des mathematischen Modells der simulierten PLI-Messungen hin.
Im Gegensatz zu den ungekippten Messungen weichen die ungekippten Messungen stark voneinander ab. Die Abweichungen der Intensitätsmessungen sind sehr groß, wobei sich mit Verringerung der Voxelsize die Ergebnisse wieder einander annähern. Dies lässt darauf schließen, dass die Anzahl der bei SimPLI vorkommenden falschen Klassifizierung von Gewebe mit der Steigerung der Anzahl der Subvoxel abnimmt. Dennoch ist die so verbleibende Abweichung immer noch ziemlich groß, sodass die Voxelsize noch um einiges verringert werden müsste, um ähnliche Ergebnisse zu erzeugen. Die Voxelsize kann bei SimPLI jedoch nur eingeschränkt verringert werden, da die benötigte Speichermenge kubisch wächst und so Rechner verwendet werden müssen die einen großen Hauptspeicher besitzen. Die benötigte Speichermenge bei der Durchführung von GPU-

Voxelsize	**Anzahl Subvoxel**	**Speichermenge**
0,5 µm	13.824.000	263,67 MB
0,25 µm	110.592.000	2.109,38 MB
0,2 µm	216.000.000	4.119,87 MB
0,125 µm	884.736.000	16.875,00 MB
0,0625 µm	7.077.888.000	135.000,00 MB

Tabelle 4: Benötigte Speichermenge bei SimPLI für die Speicherung der optischen Eigenschaften für Subvoxel bei unterschiedlicher Voxelsize.

SimPLI ist vergleichsweise gering und stellt keine Einschränkung dar. Stattdessen ist die Laufzeit aufgrund von einer wesentlich größeren Anzahl an Kollisionskontrollen deutlich höher. Dies ist jedoch notwendig, da so die Genauigkeit der Ermittlung der optischen Eigenschaften im Volumen erhöht wird und Fehler, die durch das Einteilen des Gewebes in Voxel entstehen, vermieden werden. Während die Laufzeit von GPUSimPLI bei den durchgeführten Messungen circa 7-mal größer ist, ist die Anzahl der Kollisionskontrollen zunächst 55-mal höher und fällt bei eine Voxelsize von 0,2 µm auf das 25-fache. Damit scheint trotz der hohen Anzahl der Berechnungen durch die hohe Parallelisierbarkeit der GPU eine sinnvolle Abarbeitung stattzufinden. Müsste die hohe Anzahl der Berechnungen von einer CPU ausgeführt werden, so wäre die Laufzeit vermutlich größer, da nur wenige Kerne für die Berechnungen zur Verfügung stehen und so ein hoher Task-Scheduling Aufwand erwartet werden müsste. Außerdem ist die Parallelisierung der Berechnungen der einzelnen Lichtstrahlen ohne die Durchführung der Kollisionskontrollen auf der GPU performanter als auf der CPU. Dies wurde bereits in der Seminararbeit gezeigt [6].
Somit bietet GPUSimPLI eine Alternative zu SimPLI, welche zwar eine längere Laufzeit hat, jedoch Diskretisierungsfehler beim Ermitteln der optischen Eigenschaften an Positionen der Lichtstrahlen vermeidet und wesentlich weniger Speicher benötigt. Dadurch können simulierte Messungen an Rechner durchgeführt werden, welche eine geeignete Grafikkarte besitzen, anstatt Rechner mit sehr großem Hauptspeicher zu verwenden. Die Vermeidung von Diskretisierungsfehlern erhöht die Genauigkeit der Messungen und liefert so gemessene Intensitäten, welche realen Messungen ähnlicher sind.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine parallele Implementierung einer 3D-PLI Simulation auf der GPU vorgestellt, welche mithilfe von Ray Tracing funktioniert. Dabei werden die Lichtstrahlen einzeln verfolgt und an den zu erreichenden Positionen Kollisionskontrollen mit einem Volumen von Fasern durchgeführt. Mögliche Kollisionskandidaten werden in der Broad-Phase mit dem Sort and Sweep-Algorithmus herausgefiltert und anschließend in der Narrow-Phase auf eine vorhandene Kollision überprüft. Das Fasersegment, welches am nächsten zur betrachteten Position liegt und diese enthält, wird zur Ermittlung der optischen Eigenschaften verwendet. Dazu wird die Faserschicht ermittelt, die den Punkt enthält und die Parameter für die Berechnungen der veränderten Eigenschaften des Lichtstrahls bestimmt.
Durch diese Art der Berechnung werden im Vergleich zu SimPLI die optischen Eigenschaften nicht Voxelweise im Volumen, sondern an jeder mit einem Lichtstrahl betrachteten Position genau bestimmt. Dadurch werden fehlerhafte Bestimmungen der Eigenschaften des Gewebes vermieden, wodurch die Berechnungen mit einer höheren Genauigkeit die realen Messungen beschreiben. Aufgrund der wesentlich größeren Anzahl an Kollisionskontrollen im Vergleich zu SimPLI, ist die Laufzeit des Programms höher.

Das implementierte Verfahren könnte noch auf unterschiedliche Arten verbessert werden.
So wird die erhöhte Laufzeit des Programms durch die hohe Anzahl an Kollisionskontrollen hervorgerufen. Bei der Auswahl der Kollisionskandidaten mit dem Sort and Sweep-Algorithmus werden die Kandidaten nur anhand der X-Achse herausgefiltert. Dadurch tritt zum Einen der Fall bei ungekippten Messungen auf, dass für jeden Lichtstrahl an jeder Position die selben Kandidaten ermittelt werden, obwohl die Ermittlung nur einmal notwendig wäre. Dies liegt daran, dass die Lichtstrahlen bei einer ungekippten Messung nur vertikal verlaufen und ihre X-Position nicht verändern. Daher könnte bei ungekippten Messungen ein Sonderfall implementiert werden, um viele Berechnungen zu sparen. Des Weiteren könnte ein anderes Verfahren zur Kollisionskandidatenbestimmung genutzt werden, welches mit einer höheren Genauigkeiten Kandidaten herausfiltert. Bei der bisherigen Implementierung werden für eine betrachtete Positionen viele Kollisionskandidaten ermittelt, die nicht kollidieren, da als einziges Kriterium die Anordnung auf der X-Achse betrachtet wird. Diese ist jedoch nur ein grobes Kriterium. Besser wäre die Einteilung in eine *bounding volume hierarchy* (BVH), welche besonders für die Minimierung der Anzahl von möglichen Kollisionskandidaten geeignet ist [5]. Die Anordnung

der Fasersegmente in einer BVH konnte so die Anzahl fälschlicher Kollisionskandidaten verringern und somit die benötigte Rechenzeit verkürzen.
Bei der GPU Programmierung sind die Threads in *warps* strukturiert, welche die selben Codeabläufe befolgen. Dabei wird jede mögliche Verzweigung innerhalb von Device-Code von allen 32 Threads eines warps durchlaufen, wobei die nicht zum Berechnen vorgesehenen Ergebnisse verworfen werden. Zur Verbesserung der Performanz sollte das entwickelte Programm auf Kontrollstrukturen untersucht werden und die Anzahl dieser verringert werden.

6 Anhang

```
# Setup Simpli for Tissue Generation
simpli = fastpli.simulation.Simpli()
simpli.omp_num_threads = 1
# voxel_size varies between [0.5, 0.25, 0.2]
simpli.voxel_size = 0.5  # in micro meter
simpli.set_voi([-60] * 3, [60] * 3)  # in micro meter
simpli.fiber_bundles = fastpli.io.fiber_bundles.load(
    os.path.join(FILE_PATH, 'Testdaten/cube_2pop_psi_0.30_omega_60.00_r_1
    .00_v0_105_.solved.h5'))

simpli.fiber_bundles_properties = [[(0.333, -0.004, 10, 'p'),
                                    (0.666, 0, 5, 'b'),
                                    (1.0, 0.004, 1, 'r')],
                                   [(0.333, -0.004, 10, 'p'),
                                    (0.666, 0, 5, 'b'),
                                    (1.0, 0.004, 1, 'r')]]
# (_0, _1, _2, _3)
# _0: layer_scale times radius
# _1: strength of birefringence
# _2: absorption coefficient mu: I = I*exp(-mu*x)
# _3: model: 'p'-parallel, 'r'-radial or 'b'-background

# Simulate PLI Measurement
simpli.filter_rotations = np.deg2rad([0, 30, 60, 90, 120, 150])
simpli.light_intensity = 26000  # a.u.
simpli.interpolate = "NN"
simpli.wavelength = 525  # in nm
simpli.pixel_size = 20  # in micro meter
simpli.sensor_gain = 3
simpli.optical_sigma = 0.71  # in voxel size
simpli.tilts = np.deg2rad([(0, 0), (5.5, 0), (5.5, 90), (5.5, 180),
                           (5.5, 270)])
```

Abbildung 15: Parameter für die Simulation von SimPLI und GPUSimPLI

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Markus Axer u. a. “A novel approach to the human connectome: Ultra-high resolution mapping of fiber tracts in the brain”. In: *NeuroImage* 54.2 (2011), S. 1091–1101. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2010.08.075.

[2] Markus Axer u. a. “High-Resolution Fiber Tract Reconstruction in the Human Brain by Means of Three-Dimensional Polarized Light Imaging”. In: *Frontiers in Neuroinformatics* 5 (2011), S. 34. ISSN: 1662-5196. DOI: 10.3389/fninf.2011.00034. URL: https://www.frontiersin.org/article/10.3389/fninf.2011.00034.

[3] Melanie Dohmen u. a. “Understanding fiber mixture by simulation in 3D Polarized Light Imaging”. In: *NeuroImage* 111 (2015), S. 464–475. ISSN: 1053-8119. DOI: 10.1016/j.neuroimage.2015.02.020. URL: http://juser.fz-juelich.de/record/188078.

[4] Karras, Tero (Nvidia Corporation). “Thinking Parallel, Part I: Collision Detection on the GPU”. In: (2012). [Online; accessed August 18, 2020]. URL: https://developer.nvidia.com/blog/thinking-parallel-part-i-collision-detection-gpu/.

[5] Karras, Tero (Nvidia Corporation). “Thinking Parallel, Part III: Tree Construction on the GPU”. In: (2012). [Online; accessed September 7, 2020]. URL: https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/.

[6] Kobusch, Alexander. “Parallele Implementierung einer Polarisationsmikroskopie-Simulation zur Licht-Hirngewebe-Wechselwirkung auf der GPU”. In: (2020).

[7] Matuschke, Felix. *Fiber Architecture Simulation Toolbox for PLI*. [Online; accessed August 18, 2020]. 2020. URL: https://github.com/3d-pli/fastpli.

[8] Miriam Menzel. “Finite-difference time-domain simulations assisting to reconstruct the brain’s nerve fiber architecture by 3D polarized light imaging”. Auch veröffentlicht auf dem Publikationsserver der RWTH Aachen University. - Ausgezeichnet mit der Borchers-Plakette.; Dissertation, RWTH Aachen University, 2018. Dissertation. Jülich: RWTH Aachen University, 2018, ix, 296 S. ISBN: 978-3-95806-368-6. DOI: 10.18154/RWTH-2018-230974. URL: http://publications.rwth-aachen.de/record/750948.

[9] Miriam Menzel u. a. “A Jones matrix formalism for simulating three-dimensional polarized light imaging of brain tissue”. In: *Journal of the Royal Society Interface* 12.111 (2015), S. 20150734.

[10] Nvidia Corporation. *Nvidia CUDA Programming Guide*. [Online; accessed December 5, 2019]. 2019. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[11] Nicole Schubert u. a. “3D Polarized Light Imaging Portrayed: Visualization of Fiber Architecture Derived from 3D-PLI”. In: *High-Resolution Neuroimaging*. Rijeka: IntechOpen, 2018. Kap. 3. DOI: `10.5772/intechopen.72532`.