

Deterministische Echtzeit-Phasenrekonstruktion mittels GPGPU

Bachelorarbeit
am Institut für Angewandte Optik und Elektronik
Fakultät für Informations-, Medien- und Elektrotechnik
Fachhochschule Köln

Autor: Jan-Maximilian Beneke
aus Bergisch Gladbach
Matrikelnummer: 110 619 48
06.12.2010

angefertigt am
Institut für Angewandte Optik und Elektronik

Referent: Prof. Dr.-Ing. Stefan Altmeyer
Korreferent: Prof. Dr. rer. nat. Thomas Wilhein

Deterministic realtime phase retrieval by means of GPGPU

Bachelor-Thesis
submitted to the Institute of Applied Optics and Electronics
Faculty of Information, Media and Electrical Engineering
University of Applied Sciences, Cologne

Author: Jan-Maximilian Beneke
from Bergisch Gladbach
Matriculation Number: 110 619 48
06/12/2010

elaborated at the
Institute of Applied Optics and Electronics

First Reviewer: Prof. Dr.-Ing. Stefan Altmeyer
Second Reviewer: Prof. Dr. rer. nat. Thomas Wilhein

Zusammenfassung

Titel:	Deterministische Echtzeit-Phasenrekonstruktion mittels GPGPU
Autor:	Jan-Maximilian Beneke
Referent:	Prof. Dr.-Ing. Stefan Altmeyer
Korreferent:	Prof. Dr. rer. nat. Thomas Wilhein
Zusammenfassung:	<p>Die quantitative Rekonstruktion von Phasenverteilungen ist ein wichtiges Instrument um Brechungsindizes und Wegunterschiede von Objekten mit geringem Amplitudenkontrast darzustellen. Zur Rekonstruktion der Phasenverteilung wird in dieser Arbeit die Intensitäts-Transportgleichung (ITG) genutzt. Sie stellt im Allgemeinen den Zusammenhang zwischen der Intensitätsverteilung des Wellenfelds in unterschiedlichen Fokusebenen und der zugehörigen Phasenverteilung her. Im Mikroskopaufbau lassen sich die Intensitätsaufnahmen in verschiedenen Ebenen durch Versatz der Detektoren in unterschiedliche Fokuslagen mittels eines speziellen Messsystems realisieren. Die Diskretisierung des ITG-Algorithmus zur Phasenrekonstruktion macht den mehrfachen Gebrauch von schnellen Fourier-Transformationen (FFT) nötig, die sich hervorragend zur Berechnung auf Parallelrechnern eignen; Die heutzutage meistverbauten Parallelrechner sind Grafikkartenprozessoren (GPU), die sich in nahezu jedem PC befinden und deren Hauptaufgabe das schnelle und parallelisierte Berechnen von Matrixoperationen ist. Programmierumgebungen wie NVIDIA CUDA ermöglichen die einfache Nutzung herkömmlicher Grafikprozessoren zur Berechnung von gewöhnlichen Gleitkommaoperationen. Unter Verwendung dieser modernen Technologie ist es möglich, den ITG-Algorithmus zu portieren und auf die speziellen Anforderungen von GPUs anzupassen. So kann die unverzögerte, echtzeitfähige Phasenrekonstruktion auch auf Consumer-Rechnersystemen realisiert werden. Aus den gewonnenen Messergebnissen können synchron ausdrucksstarke 3D-Modelle gerendert werden.</p>
Stichwörter:	Intensitäts-Transportgleichung (ITG), deterministische Phasenbestimmung, 3D-Messtechnik, Mikroskopie, Parallel Programming, NVIDIA CUDA, General Purpose Computation on Graphics Processing Unit (GPGPU), digitale Bildverarbeitung
Datum:	06.12.2010

Abstract

Titel: Deterministic realtime phase retrieval by means of GPGPU

Author: Jan-Maximilian Beneke

First Reviewer: Prof. Dr.-Ing. Stefan Altmeyer

Second Reviewer: Prof. Dr. rer. nat. Thomas Wilhein

Summary:

The quantitative reconstruction of phase distributions is an important tool to image refractive index and path difference of an object with low amplitude contrast. In this thesis, the transport of intensity equation (TIE) will be used to retrieve a phase distribution. It shows the abstract correlation between the intensity distribution in different focus layers of a wave field and the corresponding phase distribution. In a common microscope it's possible to achieve the intensity distribution in diverse layers by the displacement of detectors in different focus positions with a special measuring system. The discretization of the TIE-algorithm for phase reconstruction necessitates the multiple use of fast Fourier transforms (FFT), which suites well for calculations on parallel computers; Today, the most installed parallel computers are graphics processing units (GPU). Those you can find in nearly every personal computer and their major task is the fast and parallel computation of matrix operations. Programming environments like NVIDIA CUDA enable the easy use of common graphic processors for the calculations of common floating point operations. By using these modern technology it's possible to port and customize the TIE-algorithm to the particular needs of GPUs. In this way, the undelayed and real-time capable phase reconstruction can be realized even on consumer computer systems. It's further possible to render an expressive live 3D-model from the retrieved results of measurement.

Keywords:

transport of intensity equation (TIE), deterministic phase retrieval, 3D metrology, microscopy, parallel programming, NVIDIA CUDA, General Purpose Computation on Graphics Processing Unit (GPGPU), digital image processing

Date:

06/12/2010

Danksagung

Mein Dank gilt an dieser Stelle jedem, der mich während meines Studiums begleitet und somit dazu beigetragen hat, es mit dieser Arbeit erfolgreich abschliessen zu können.

Ich möchte mich besonders bei Herrn Prof. Dr.-Ing. Stefan Altmeyer für die Möglichkeit bedanken, diese Arbeit zu verfassen. Seine einzigartige Herangehensweise an Problemstellungen und sein Talent auch komplexe Zusammenhänge verständlich darzustellen, hat immer wieder von neuem motiviert und für Inspiration gesorgt.

Bei Herrn Prof. Dr. rer. nat. Thomas Wilhein möchte ich mich für die praxisnah gestalteten Vorlesungen und seinen Langmut im Umgang mit Studenten bedanken.

Ein besonderer Dank gilt M. Eng. Johannes Frank, der jedem Abschlussstudenten mit unendlicher Geduld seinen gesamten Fachverstand zur Verfügung stellt. Zudem besitzt er das Talent, auch bei undurchsichtigen Problemstellungen immer den ausschlaggebenden Anreiz zum nächsten Schritt geben zu können. Er hat mit besonderem Einsatz zum Gelingen dieser Arbeit beigetragen.

Herrn Dipl. Ing. Anton Kraus stand mir bei jeder Frage mit seiner umfassenden praktischen Erfahrung zur Seite. Ihm möchte ich besonders für die hervorragende Zusammenarbeit bei der Praktikumsbetreuung in den Fächern Wellenoptik und Holographie danken.

Außerordentlich danke ich den Kommilitonen, die mich durch mein Studium begleitet haben. Besonders zu erwähnen sind diejenigen, die während der Zeit der Arbeit im Institut beschäftigt waren und immer mit einem offenen Ohr zu Verfügung standen und ebenso für den ein oder anderen Scherz zu haben waren: Dipl.-Ing. Wolfgang Stein, Javed Razzaq, B. Sc. Felix Möllmann, Sebastian Heck, Sabine Hinz, Dipl.-Ing. Jan Matrisch, Sascha Jenderny, Manuel Noppel, Dipl.-Ing. Cenk Kaan, Dipl.-Ing. Sebastian Wette, Dipl.-Ing. André Kluck, Dipl.-Ing. Sven Connemann, Dipl.-Ing. Caroline Daske und Dipl.-Ing. Bernd Bleymehl.

Der wohl größte Dank gilt meinen Eltern, die mein Interesse an der Technik immer gefördert und mich in jeder denkbaren Weise unterstützt haben.

Mein letzter und herzlichster Dank geht an meine Freundin Esther, die es immer wieder schaffte für den nötigen Ausgleich und emotionalen Rückhalt zu sorgen.

Inhaltsverzeichnis

1. Einleitung und Motivation	1
2. Grundlagen	3
2.1. Licht	3
2.1.1. Licht als elektromagnetische Welle	3
2.1.2. Intensität	4
2.1.3. Phase	5
2.1.4. Wellenfront	6
2.1.5. Brechungsindex	7
2.2. Phasenrekonstruktion anhand der Intensitätsverteilung	7
2.2.1. Intensitäts-Transportgleichung	8
2.2.2. Symbolische Lösung	10
2.2.3. Lösung im Frequenzraum	11
2.2.4. Der Lösungsalgorithmus	17
2.3. Mikroskop	21
2.3.1. Köhlersche Beleuchtung	22
2.3.1.1. Durchlichtmikroskop	24
2.3.1.2. Auflichtmikroskop	24
2.3.2. Beugungsbegrenztes Auflösungsvermögen	25
3. Laboraufbau	26
3.1. Konzept	26
3.2. Umsetzung	27
4. Parallel Programming	31
4.1. GPU - Der Grafikprozessor	33
4.2. NVIDIA CUDA	36
4.2.1. Einführung in das Programmiermodell	38
4.2.1.1. Kernels	38
4.2.1.2. Hierarchie der Threads	39
4.2.1.3. Hierarchie des Speichers	41
4.2.1.4. Heterogenität der Programmierung	43
4.2.1.5. Compute Capability	44
4.2.2. Das erste CUDA Programm	45
4.2.3. Fortgeschrittenes Programmieren mit CUDA	47
4.2.3.1. Shared Memory	47

4.2.3.2. Mehrere Devices	51
4.2.3.3. Streams	52
4.2.4. FFT mittels CUDA	54
4.2.4.1. FFT in einem Stream	55
4.2.4.2. Bildshift	55
4.2.4.3. Beispielprogramm einer FFT von Bilddaten	57
4.2.5. Programmoptimierung	58
4.2.5.1. Regeln zur Programmoptimierung	60
4.3. Reduktion als gängiges paralleles Programmierkonzept	61
5. Bildverarbeitung	66
5.1. OpenGL	66
5.1.1. Zustandsautomat und Rendering-Pipeline	67
5.1.2. Zweidimensionales Beispiel	69
5.1.3. Dreidimensionales Beispiel	71
5.1.4. Matrizen	73
5.1.5. Projektion und Perspektive	76
5.1.6. Texturen und 2D-Bildanzeige	78
5.1.7. 3D-Plot einer zweidimensionalen Funktion	80
5.1.8. Interoperabilität zwischen OpenGL und CUDA	81
5.2. Kamerasteuerung	81
5.2.1. Übertragungsbandbreite	82
5.2.2. Synchronisation	82
5.3. Projektive Transformation	82
5.4. Shading Correction	89
5.5. Programmablauf	90
6. Messdurchführung	92
6.1. Kalibrierung	92
6.2. Messung	97
6.3. Messergebnisse	100
6.3.1. Lichtwellenleiter	101
6.3.2. Holographische Probe	102
6.3.3. Diatomeen	103
6.3.4. Microlinse	104
7. Benchmarking	106
7.1. Datentransferrate	108
8. Resümee und Ausblick	109
A. Datenblätter	112
Literaturverzeichnis	117

Abbildungsverzeichnis	121
Tabellenverzeichnis	123
Quellcodeverzeichnis	124

1. Einleitung und Motivation

Erst im 17. Jahrhundert wurde der Wellencharakter des Lichts entdeckt, da die Deutungen der Strahlenoptik nicht ausreichten, um alle optischen Phänomene und Eigenarten umfassend zu erläutern. Das optische Wellenfeld kann gänzlich durch seine Amplitude und Phase beschrieben werden. Bei Wechselwirkung mit einem Objekt wird die Wellenfront moduliert und speichert somit die Informationen des Objekts in Phase und Amplitude. Selbst moderne Detektoren können lediglich die zeitlich gemittelte Intensitätsverteilung, aus der sich die Amplitudenverteilung berechnen lässt, nicht aber die Phasenverteilung eines Wellenfeldes aufnehmen.

Übliche Verfahren zur indirekten Bestimmung der Phasenverteilung basieren auf interferometrischen oder holographischen Messtechniken [1]. Diese Messverfahren benötigen präzise Messaufbauten, die gegen äußere Einwirkungen anfällig sind und eine anschließende Weiterverarbeitung der Bildinformationen zur Gewinnung der Phaseninformationen benötigen.

Zur Bestimmung des Wellenfeldes existieren iterative und deterministische Rechenmethoden, die mit deutlich einfacheren Messaufbauten auskommen. Zu den iterativen Ansätzen zählt der Gerchberg-Saxton Algorithmus (GSA), wie er in Arbeit [2] erläutert wurde. Als deterministisch gilt der in dieser Arbeit verwendete Algorithmus auf Grundlage der Intensitäts-Transportgleichung (ITG). Der ITG-Algorithmus benötigt drei Intensitätsverteilungen als Eingangsgrößen, die eine jeweils andere Fokuslage des Objekts zeigen, aus denen die Phasenverteilung des Wellenfeldes errechnet werden kann. In den vorangegangenen Arbeiten [3] und [4] wurde der Algorithmus bereits angewandt und validiert. Die drei Intensitätsverteilungen wurden jedoch sequenziell über aufwändige Messanordnungen detektiert. In der Arbeit [5] gelang es, einen handlichen Messaufsatz für Standardlabormikroskope zu entwickeln, der die transiente Detektion der benötigten Bilder zu einem Zeitpunkt ermöglicht.

Die bisherigen Softwareimplementierungen benötigten zur Berechnung einer Phasenverteilung mitunter mehrere Sekunden. Um ein solches ITG-Mikroskop für die praktische Anwendung zu realisieren, sollte der Algorithmus in Echtzeit ablaufen und es sollte neben dem Drei-Kamera-System ausschließlich gängige Consumer-Hardware benötigt werden. Durch die große Anzahl an Fourier-Transformationen und den verzweigten Aufbau eignet sich der Algorithmus [4] besonders zur Parallelisierung der einzelnen Arbeitsschritte. Die gängigsten Parallelrechner sind Grafikkartenprozessoren, wie sie mittlerweile in nahezu jedem Anwendercomputer verbaut sind.

Zur Berechnung gewöhnlicher Gleitkommaoperationen auf der Grafikkarte existieren Programmierumgebungen wie CUDA C des Herstellers NVIDIA die eine einfache Portierung eines Problems in viele parallelisierbare Unterprobleme ermöglichen. Durch den Einsatz solcher moderner Programmietechniken und passender Hardware ist es möglich,

den Algorithmus bis zur Echtzeitfähigkeit zu optimieren.

Durch die gewaltige Leistungsfähigkeit moderner Grafikkartenprozessoren ist es sogar möglich, die Phase direkt weiter zu verarbeiten. So bietet es sich an, ein 3D-Modell der soeben errechneten Phasenverteilung darzustellen. Dieses Modell lässt sich im dreidimensionalen Raum frei drehen und skalieren. Es ist möglich, die Struktur von Objekten im Auflicht- bzw. den optischen Wegunterschied im Durchlichtmikroskop direkt als 3D-Modell zu betrachten.

Die im Rahmen dieser Arbeit erstellte Softwarelösung *φScope* stellt alle nötigen Funktionen zur Echtzeit-Phasenrekonstruktion und zur Darstellung eines 3D-Modells für das zuvor entwickelte Drei-Kamera-System zur Verfügung.

Eine solche Mikroskopielösung auf Basis der ITG ist bisher nicht käuflich zu erwerben und bietet das Potential, sich zu einem neuartigen und revolutionären Messprinzip zu entwickeln.

Weitere Inhalte zu dieser Arbeit, wie Videos, Bildmaterial und Beispielcodes, finden sich auf der Homepage <http://www.janbeneke.de/bachelor>.

2. Grundlagen

In diesem Kapitel sollen die benötigten optischen Grundlagen erläutert, die nicht interferometrische Phasenrekonstruktion mittels der Intensitätsverteilung beschrieben sowie der generelle Aufbau eines Mikroskops näher dargelegt werden.

2.1. Licht

Als sichtbares Licht wird das elektromagnetische Spektrum im Wellenlängenbereich von 380nm bis 780nm bezeichnet [6].

Erst im 20. Jahrhundert wurde erkannt, dass Licht Teilcheneigenschaften in Form so genannter Photonen als auch Welleneigenschaften aufweist, welche im Nachfolgenden näher erläutert werden.

Neben Christian Huygens (Kap. 2.1.4) lieferte Thomas Young¹ ein eindeutiges Indiz für den Wellencharakter des Lichts; Er hat mit seinem Doppelspaltexperiment gezeigt, dass sich Licht durch Interferenz auslöschen lässt.

Die Lösung des *Wellen-Teilchen-Dualismus* gelang durch die Entwicklung der Quantendynamik. Der Begriff wurde 1927 von Niels Bohr² und Werner Heisenberg³ in der *Kopenhagener Deutung* festgelegt und umfassend erläutert.

2.1.1. Licht als elektromagnetische Welle

Da Licht ein definierter Bereich des elektromagnetischen Spektrums ist, genügt die Beschreibung durch die Wellengleichung, die sich aus den *Maxwell-Gleichungen*⁴ herleiten lässt. Als einfache Lösung der Wellengleichung gilt die harmonische Welle in folgender komplexer Form:

$$\vec{\underline{E}}(\vec{r}, t) = E_0 e^{i(\omega t \pm \vec{k}\vec{r} + \varphi_0)} \quad (2.1)$$

Wobei E_0 die Amplitude bzw. den Betrag der Welle darstellt.

$$E_0 = \sqrt{\Re\{\underline{E}\}^2 + \Im\{\underline{E}\}^2} \quad (2.2)$$

$$\Re\{\underline{E}\} = E_0 \cdot \cos(\omega t \pm \vec{k}\vec{r} + \varphi_0)$$

$$\Im\{\underline{E}\} = E_0 \cdot \sin(\omega t \pm \vec{k}\vec{r} + \varphi_0)$$

¹Thomas Young (1773-1829), englischer Augenarzt und Physiker

²Niels Henrik David Bohr (1885-1962), dänischer Physiker und Nobelpreisträger der Physik

³Werner Karl Heisenberg (1901-1976), deutscher Physiker und Nobelpreisträger der Physik

⁴benannt nach James Clerk Maxwell (1831-1879), schottischer Physiker

Die Nullphase der Welle φ_0 repräsentiert die Phasenlage zum Zeitpunkt $t = 0s$. Weiterhin stellt ω die Kreisfrequenz in Hertz [Hz] dar, t ist die Zeit und ν die Frequenz.

$$\omega = 2\pi\nu \quad (2.3)$$

Der Ortsvektor \vec{r} gibt den Bezugspunkt in allen drei Raumrichtungen an.

$$\vec{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad (2.4)$$

Der Wellenvektor \vec{k} beschreibt die Ausbreitungsrichtung der Welle, wobei λ die Wellenlänge ist. Das wechselnde Vorzeichen gibt die Ausbreitungsrichtung abhängig vom Wellenvektor an. Ein negatives Vorzeichen symbolisiert die Ausbreitung in Richtung des Wellenvektors.

$$\vec{k} = \begin{pmatrix} k_x \\ k_y \\ k_z \end{pmatrix}; |\vec{k}| = \frac{2\pi}{\lambda} \quad (2.5)$$

Die *Ausbreitungsgeschwindigkeit* der Welle im Vakuum lässt sich über die Phasengeschwindigkeit c beschreiben.

$$c = \lambda \cdot \nu \quad (2.6)$$

2.1.2. Intensität

Die Ausbreitungsgeschwindigkeit der elektromagnetischen Welle im Vakuum beträgt $c \approx 3 \cdot 10^8 \frac{m}{s}$ [7]. Setzt man diesen Wert nun in die Formel der Phasengeschwindigkeit (Gl. 2.6) ein, ergeben sich Frequenzen der Größenordnung von etwa 500THz. Dieser Frequenzbereich ist von Detektoren nicht messbar. Die gemittelte Leistung pro Fläche - *die Intensität* - hingegen ist messbar.

Der *Poynting-Vektor*⁵ kennzeichnet die in Ausbreitungsrichtung fließende Energiestromdichte.

$$\vec{S} = \vec{E} \times \vec{H} = \epsilon c^2 \vec{E} \times \vec{B} \quad (2.7)$$

Wobei \vec{E} die *elektrische Feldstärke*, \vec{H} die *magnetische Feldstärke* und \vec{B} die *magnetische Flussdichte* ist. Weiter ist c die *Phasengeschwindigkeit* bzw. *Lichtgeschwindigkeit*. Die *Intensität* entspricht nunmehr dem zeitlich gemittelten Betrag des *Poynting-Vektors* [8].

$$I = |\vec{S}| = \frac{1}{2} \epsilon c^2 |\vec{E} \times \vec{B}| = \frac{1}{2} \epsilon c^2 |\vec{E}| \cdot |\vec{B}| \cdot \sin \varphi \quad (2.8)$$

⁵benannt nach John Henry Poynting (1852-1914), britischer Physiker

Das elektrische und magnetische Feld stehen senkrecht zueinander und senkrecht auf \vec{k} , wobei $\varphi = \angle(\vec{E}, \vec{H}) = 90^\circ$. Zudem ist hier $|\vec{B}| = \frac{|\vec{E}|}{c}$ [9].

$$I = \frac{1}{2} \varepsilon c \hat{E}^2 \quad (2.9)$$

ε ist die *Permittivität* bzw. *Dielektrizitätszahl*.

$$\varepsilon = \varepsilon_0 \varepsilon_r = 8,8542 \cdot 10^{-12} \frac{As}{Vm} \cdot \varepsilon_r \quad (2.10)$$

ε_r ist hierbei eine materialabhängige Konstante und beschreibt die optischen Eigenschaften des jeweiligen Materials. Im Vakuum ist der Koeffizient $\varepsilon_r = 1$, und somit ergibt sich der Wellenwiderstand im Vakuum Z_0 .

$$Z_0 = \frac{1}{c \cdot \varepsilon} = 377\Omega \quad (2.11)$$

Eingesetzt ergibt sich damit die Intensität im Vakuum:

$$I = \frac{1}{2} \frac{1}{Z_0} \hat{E}^2 \approx \frac{1}{Z_0} \hat{E}^2 \quad (2.12)$$

Der Faktor $\frac{1}{2}$ trägt der Zeitmittelung Rechnung.

2.1.3. Phase

Der komplexe Exponent der harmonischen Welle (Gl. 2.1) wird als Phase φ bezeichnet.

$$\varphi = \omega t \pm \vec{k} \vec{r} + \varphi_0 \quad (2.13)$$

Sie gibt den Schwingungszustand einer Welle an einem bestimmten Raumpunkt \vec{r} (Gl. 2.4) zu einer bestimmten Zeit t an. Die Phase kann einen maximalen Wert von $2\pi \hat{=} 360^\circ$ erreichen. Bei konstanter Amplitude E_0 nimmt die Feldstärke an Orten gleicher Phase den gleichen Wert an.

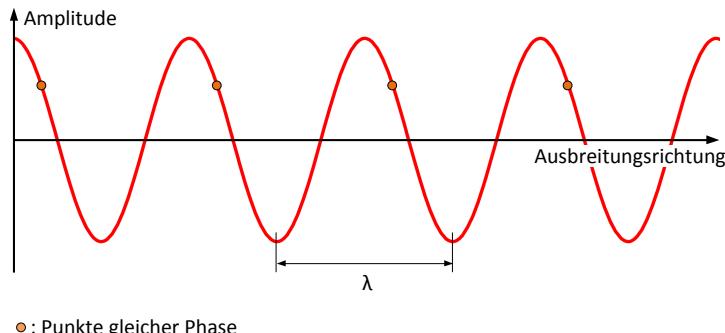


Abbildung 2.1.: Harmonische Welle mit Punkten gleicher Phase

Die Phase gibt somit Auskunft über die zurückgelegte Strecke einer Welle, also eine räumliche Information.

Kann nunmehr die räumliche Phasenverteilung einer Welle gemessen werden, erhält man Informationen über Brechzahlverteilung und Form eines durchdrungenen Objekts (Abb. 2.3).

2.1.4. Wellenfront

Die Welleneigenschaft des Lichts wurde durch das von Christian Huygens⁶ beschriebene *Huygenssche Prinzip* untermauert.

Dieses Prinzip besagt, dass jeder Punkt einer sich ausbreitenden Wellenfront als Ausgangspunkt von Elementarwellen (Kugelwellen) gleicher Frequenz, Wellenlänge und Polarisation betrachtet werden kann. Diese breiten sich selbst wieder mit Lichtgeschwindigkeit aus und deren Einhüllende legt die neue Wellenfront fest [10].

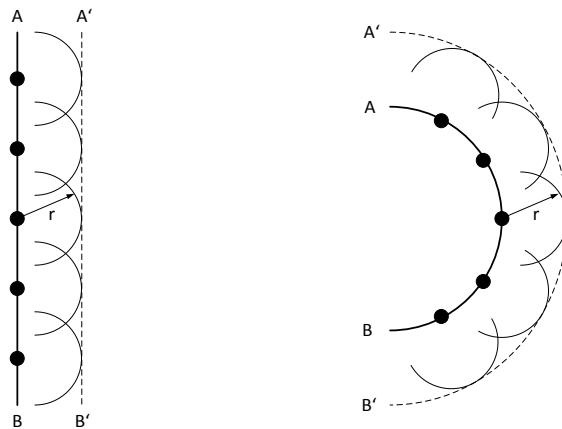


Abbildung 2.2.: Huygenssches Prinzip für eine ebene und für eine Kugelwelle

Als Wellenfront bezeichnet man die Fläche eines Wellenfeldes, auf der jeder Punkt die gleiche Phase besitzt.

Wird eine harmonische Welle von einer Punktquelle in ein isotropes, homogenes Medium emittiert, ist die Wellenfront eine Kugel, die sich mit der Phasengeschwindigkeit in alle Raumrichtungen gleichmäßig ausbreitet. In diesem Fall ergibt sich eine Kugelwelle.

Im Unendlichen ist die Wellenfront als eben anzusehen, da die Krümmung einer Kugel gegen Null geht wenn der Radius gegen unendlich strebt.

⁶Christian Huygens (1629-1695), holländischer Physiker und Mathematiker

2.1.5. Brechungsindex

Ein wichtiger Parameter von transparenten und teiltransparenten Objekten ist der Brechungsindex n . Er gibt das Verhältnis der Ausbreitungsgeschwindigkeit c_n der elektromagnetischen Welle im Medium zur Ausbreitungsgeschwindigkeit c im Vakuum an.

$$n = \frac{c}{c_n} \quad (2.14)$$

Tritt eine elektromagnetische Welle durch ein Medium, dessen Brechungsindex $n > 1$ ist, wird die Phase verzögert, d.h. sie breitet sich im Medium langsamer aus. Die Phasenfront wird verzerrt.

Besitzt ein Objekt nun einen Brechungsindex größer 1 und keine homogene Dicke, so kann bei konstantem Brechungsindex n an der resultierenden Phasenverteilung die lokale Objektdicke abgelesen werden.

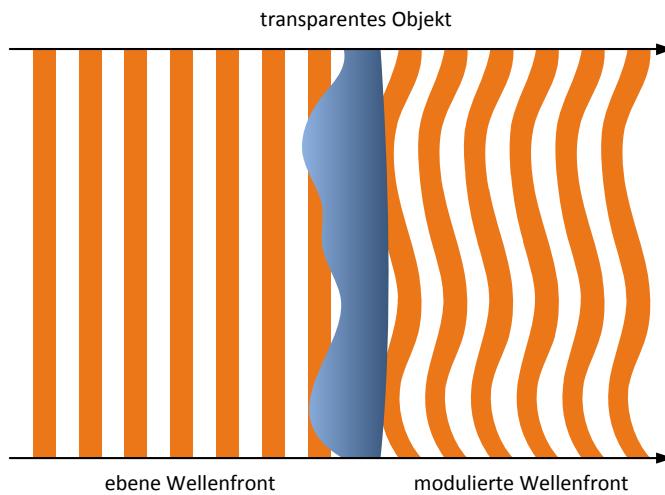


Abbildung 2.3.: Phasenverzerrung beim Durchgang einer ebenen Phasenfront durch ein transparentes Objekt

2.2. Phasenrekonstruktion anhand der Intensitätsverteilung

Ein üblicher optischer Detektor erfasst lediglich die zweidimensionale Intensitätsverteilung. In vielen Anwendungen ist es allerdings von Interesse, das gesamte Wellenfeld, also Amplitude und Phase, zu detektieren. Die Phasenverteilung führt verschiedene Informationen eines be- oder durchstrahlten Objekts mit sich.

Zu den gängisten Verfahren zur Phasengewinnung gehören interferometrische Ansätze [1][11]. Diese erfordern allerdings kohärente Lichtquellen und eine hochempfindliche Messanordnung. Um dies zu umgehen, wird in dieser Arbeit ein Phase-Retrieval Verfahren eingesetzt.

Solche Verfahren ermöglichen die Bestimmung der Phasenverteilung aus der Intensitätsverteilung ohne interferometrischen Aufbau. Es gibt zwei grundsätzliche Ansätze der *Phase-Retrieval* Verfahren. Zum einen den iterativen Ansatz: Er nutzt den Zusammenhang der Intensitätsverteilung in den konjugierten Ebenen. Eine genauere Darstellung ist in [2] zu finden. In dieser Arbeit wird ein deterministischer Ansatz verwendet. Dieser setzt einen direkten Zusammenhang von Intensitäts- und Phasenverteilung voraus. Der mathematische Weg von der Intensitäts- zur Phasenverteilung ist beispielsweise über die *Intensitäts-Transportgleichung* (nachfolgend *ITG* genannt) gegeben.

In diesem Kapitel wird zunächst die allgemeine Form der ITG hergeleitet, die eindeutige Lösung der Phase wird im Abschnitt *Symbolische Lösung* (Kap: 2.2.2) entwickelt und unter *Lösung im Frequenzraum* (Kap. 2.2.3) wird auf die numerische Implementation eingegangen.

Eine ausführlichere Herleitung und Erläuterung der ITG ist in [4] zu finden.

2.2.1. Intensitäts-Transportgleichung

Die ITG ist eine partielle Differentialgleichung, die einen direkten Zusammenhang zwischen Intensität und Phase beschreibt. Erstmals hergeleitet wurde sie 1983 von *M.R. Teague* [12]. Für die Herleitung der ITG gilt der Ansatz der monochromatischen Welle, wobei die spektrale Verteilung der Lichtquelle, wie in mehreren Arbeiten [3] [4] [5] nachgewiesen wurde, keinen großen Einfluss hat.

Grundlage zur Berechnung der ITG bildet die *allgemeine Wellengleichung*, hergeleitet aus den *Maxwell-Gleichungen*. Hier wird der Ansatz der *paraxialen Helmholtz-Gleichung* gewählt, ein Spezialfall der Wellengleichung [13].

$$\nabla^2 U(\vec{r}) + i2k \frac{\partial}{\partial z} U(\vec{r}) = 0 \quad (2.15)$$

$U(\vec{r})$ ist die komplexe Wellenfunktion und k die Wellenzahl (2.5). Die Multiplikation mit der konjugiert komplexen Funktion $U^*(\vec{r})$ ergibt:

$$U^*(\vec{r}) \nabla^2 U(\vec{r}) + i2k U^*(\vec{r}) \frac{\partial}{\partial z} U(\vec{r}) = 0 \quad (2.16)$$

Hieraus wird der konjugiert komplexe Term generiert.

$$U(\vec{r}) \nabla^2 U^*(\vec{r}) + i2k U(\vec{r}) \frac{\partial}{\partial z} U^*(\vec{r}) = 0 \quad (2.17)$$

Nun wird Gleichung 2.17 von 2.16 subtrahiert. Es ergibt sich:

$$U^*(\vec{r}) \nabla^2 U(\vec{r}) - U(\vec{r}) \nabla^2 U^*(\vec{r}) + i2k U^*(\vec{r}) \frac{\partial}{\partial z} U(\vec{r}) + i2k U(\vec{r}) \frac{\partial}{\partial z} U^*(\vec{r}) = 0 \quad (2.18)$$

Mit Hilfe der Summenregel der Differentialrechnung lässt sich die Gleichung weiter vereinfachen:

$$\nabla(U^*(\vec{r})\nabla U(\vec{r})) + i2k\frac{\partial}{\partial z}(U(\vec{r})U'(\vec{r})) = 0 \quad (2.19)$$

Da in diesem Fall die elektrische Feldstärke als komplexes Wellenfeld des Lichts auftritt, wird definiert:

$$U(\vec{r}) = E(\vec{r}) = |E(\vec{r})|e^{i\Phi(\vec{r})}$$

Daraus folgt:

$$\nabla(E^*(\vec{r})\nabla E(\vec{r})) + i2k\frac{\partial}{\partial z}(E(\vec{r})E'(\vec{r})) = 0 \quad (2.20)$$

Durch Anwenden der Kenntnis, dass

$$\nabla E(\vec{r}) = e^{i\Phi}(i|E(\vec{r})|\nabla\Phi(\vec{r}) + \nabla|E(\vec{r})|)$$

und

$$\nabla E^*(\vec{r}) = e^{-i\Phi}(-i|E(\vec{r})|\nabla\Phi(\vec{r}) + \nabla|E(\vec{r})|)$$

folgt:

$$\nabla_{\perp}(|E(\vec{r})|^2\nabla_{\perp}\Phi(\vec{r}) + i2k\frac{\partial}{\partial z}(|E(\vec{r})|^2)) = 0 \quad (2.21)$$

∇_{\perp} ist hier der transversale Nabla-Operator. Unter Berücksichtigung von 2.12 und der Division der Gleichung 2.21 durch $2i$ folgt:

$$\nabla_{\perp}(I(\vec{r})\nabla_{\perp}\Phi(\vec{r}) + k\frac{\partial}{\partial z}(I(\vec{r}))) = 0$$

Oder wie in der Literatur üblich:

$$\nabla_{\perp}(I(\vec{r})\nabla_{\perp}\Phi(\vec{r})) = -k\frac{\partial}{\partial z}(I(\vec{r})) \quad (2.22)$$

Dies ist die *Intensitäts-Transportgleichung*. Hier ist zu erkennen, dass die Phase einer elektromagnetischen Welle nur noch an die Intensität gebunden ist und durch komplexe mathematische Operationen ermittelt werden kann.

2.2.2. Symbolische Lösung

Um die Phase zu erhalten, muss die Gleichung 2.22 nach $\Phi(\vec{r})$ aufgelöst werden.

$$\nabla_{\perp} (I(\vec{r}) \nabla_{\perp} \Phi(\vec{r})) = -k \frac{\partial}{\partial z} (I(\vec{r}))$$

Der Term $I(\vec{r}) \nabla_{\perp} \Phi(\vec{r})$ wird durch die Funktion $\nabla_{\perp} V$ substituiert. Dieser Ansatz wurde von *M. R. Teague* [12] eingeführt und impliziert, dass das Vektorfeld $I(\vec{r}) \nabla_{\perp} \Phi(\vec{r})$ durch Gradientenbildung entsteht und daher rotationsfrei sein muss.

$$\begin{aligned} \nabla_{\perp} (\nabla_{\perp} V) &= -k \frac{\partial}{\partial z} (I(\vec{r})) \\ \nabla_{\perp}^2 V &= -k \frac{\partial}{\partial z} (I(\vec{r})) \\ V &= -k \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \end{aligned} \quad (2.23)$$

Zusätzlich wird ∇_{\perp} angewandt und wieder rücksubstituiert.

$$\begin{aligned} \nabla_{\perp} V &= -k \nabla_{\perp} \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \\ I(\vec{r}) \nabla_{\perp} \Phi(\vec{r}) &= -k \nabla_{\perp} \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \\ \nabla_{\perp} \Phi(\vec{r}) &= -k I(\vec{r})^{-1} \nabla_{\perp} \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \end{aligned} \quad (2.24)$$

Wird nun erneut der ∇_{\perp} Operator angewandt, ergibt sich die Phase eines Wellenfeldes.

$$\begin{aligned} \nabla_{\perp} (\nabla_{\perp} \Phi(\vec{r})) &= -k \nabla_{\perp} \left(I(\vec{r})^{-1} \nabla_{\perp} \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \right) \\ \nabla_{\perp}^2 \Phi(\vec{r}) &= -k \nabla_{\perp} \left(I(\vec{r})^{-1} \nabla_{\perp} \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \right) \\ \Phi(\vec{r}) &= -k \nabla_{\perp}^{-2} \left[\nabla_{\perp} \left(I(\vec{r})^{-1} \nabla_{\perp} \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \right) \right] \end{aligned} \quad (2.25)$$

2.2.3. Lösung im Frequenzraum

Die in Unterkapitel 2.2.2 vorgestellte mathematische Lösung der Phase $\Phi(\vec{r})$

$$\Phi(\vec{r}) = -k\nabla_{\perp}^{-2} \left[\nabla_{\perp} \left(I(\vec{r})^{-1} \nabla_{\perp} \nabla_{\perp}^{-2} \frac{\partial}{\partial z} (I(\vec{r})) \right) \right]$$

sollte in eine Form gebracht werden, die es ermöglicht, sie als Algorithmus in einer Software zu implementieren.

Zur Erinnerung sei erwähnt, dass ∇_{\perp} und ∇_{\perp}^{-2} Differentialoperatoren darstellen. Diese Operatoren lassen sich elegant mittels Fourier-Transformationen darstellen.

Das Fourierintegral

Eine sinus- oder kosinusförmige Funktion mit beliebiger Frequenz, Amplitude und Phase kann als Summe entsprechend gewichteter Kosinus- und Sinusfunktionen dargestellt werden [14].

$$A \cdot \cos(\omega x) + B \cdot \sin(\omega x) = C \cdot \cos(\omega x - \varphi) \quad (2.26)$$

Die *Fourierreihe*⁷ erweitert diese Grundidee auf beinahe jede periodische Funktion $g(x)$, mit der Grundfrequenz ω_0 als Summe von harmonischen Sinusfunktionen.

$$g(x) = \sum_{k=0}^{\infty} [A_k \cos(k\omega_0 x) + B_k \sin(k\omega_0 x)] \quad (2.27)$$

Die konstanten Gewichte A_k und B_k werden als *Fourierkoeffizienten* der Funktion $g(x)$ bezeichnet und lassen sich aus dieser eindeutig mittels *Fourieranalyse* bestimmen. Die Frequenzen der in der Fourierreihe beteiligten Funktionen sind ausschließlich ganzzahlige Vielfache der Grundfrequenz $k\omega_0$, sogenannte harmonische Vielfache.

Fourier fand ebenfalls eine Möglichkeit, *nicht* periodische Funktionen als Summe von Sinus- und Kosinusfunktionen darzustellen. Allerdings erfordert dies nicht nur die harmonischen Vielfachen der Grundfrequenz, sondern im Allgemeinen unendlich viele, dicht aneinander liegende Frequenzen. Die resultierende Zerlegung nennt sich *Fourierintegral*. Es bildet die Grundlage für die Fouriertransformation.

$$g(x) = \int_0^{\infty} A_{\omega} \cos(\omega x) + B_{\omega} \sin(\omega x) d\omega \quad (2.28)$$

A_{ω} und B_{ω} stellen wiederum die Gewichte der zugehörigen Kosinus- und Sinusfunktionen dar.

$$\begin{aligned} A_{\omega} &= A(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x) dx \\ B_{\omega} &= B(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x) dx \end{aligned} \quad (2.29)$$

⁷benannt nach Jean Baptiste Joseph Fourier (1768-1830), französischer Mathematiker und Physiker

Die *Koeffizientenfunktionen* $A(\omega)$ und $B(\omega)$ enthalten eine Verteilung der im ursprünglichen Signal enthaltenen Frequenzkomponenten, das *Frequenzspektrum*.

Die Fourier-Transformation

Die *Fourier-Transformation* betrachtet, im Gegensatz zum *Fourier-Integral*(Gl. 2.28), sowohl die Ausgangsfunktion $g(x)$ als auch das zugehörige *Fourierspektrum* $G(\omega)$ als komplexwertige Funktionen ($g(x), G(\omega) \in \mathbb{C}$).

Der Übergang von der Ausgangsfunktion $g(x)$ zu ihrem Fourierspektrum $G(\omega)$ wird als Fourier-Transformation⁸ (\mathcal{F}) bezeichnet [15].

$$\mathcal{F}\{g(x)\} = G(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot e^{-i\omega x} dx \quad (2.30)$$

Umgekehrt kann die ursprüngliche Funktion $g(x)$ aus dem Fourierspektrum $G(\omega)$ durch die *inverse Fourier-Transformation*⁹ (\mathcal{F}^{-1}) rekonstruiert werden.

$$\mathcal{F}^{-1}\{g(x)\} = g(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(\omega) \cdot e^{i\omega x} d\omega \quad (2.31)$$

Zusammenfassend lässt sich also sagen, dass die Fourier-Transformation ein Signal in seine Frequenzanteile, also sein Spektrum, aufteilt (Beispiele in [14]).

In dieser Arbeit sind die Signale ortsabhängig, da die Intensität vom Detektor als Funktion von x und y aufgenommen wird.

$$\mathcal{F}\{g(x)\} = G(f_x) = \int_{-\infty}^{\infty} g(x) \cdot e^{-i2\pi f_x x} dx \quad (2.32)$$

Auch der zweidimensionale Fall, zum Beispiel ein Bild, lässt sich transformieren.

$$\mathcal{F}\{g(x, y)\} = G(f_x, f_y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(x, y) \cdot e^{-i2\pi(f_x x + f_y y)} dx dy \quad (2.33)$$

Äquivalent hierzu ergibt sich die inverse, zweidimensionale Transformation.

$$\mathcal{F}^{-1}\{G(f_x, f_y)\} = g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} G(f_x, f_y) \cdot e^{i2\pi(f_x x + f_y y)} df_x df_y \quad (2.34)$$

Die diskrete Fourier-Transformation (DFT)

Die Definition der kontinuierlichen Fourier-Transformation ist für die numerische Berechnung nicht direkt geeignet. Kontinuierliche Funktionen können weder dargestellt, noch können die benötigten Integrale tatsächlich berechnet werden. Allerdings liegen die später genutzten Bildinformationen als diskrete Daten vor. Es wird also eine spezielle Art der Fourier-Transformation gefunden werden, die DFT (engl. Discrete Fourier

⁸hier Vorwärtstransformation

⁹hier Rückwärtstransformation

transform).

Zunächst soll eine kontinuierliche Funktion in eine diskrete Funktion umgewandelt werden. Üblicherweise werden hierzu bestimmte Punkte periodischen Abstands aus der kontinuierlichen Funktion extrahiert; Diesen Vorgang nennt man Abtastung beziehungsweise *Sampling*.

Zur formalen Beschreibung benötigen wir das *Dirac-Funktional* $\delta(x)$ ¹⁰. Diese moduliert einen idealen Impuls, das heisst, die Funktion hat abgesehen von ihrem Ursprungspunkt den Wert Null und besitzt eine Fläche von eins.

$$\delta(x) = 0 \text{ für } x \neq 0 \text{ und } \int_{-\infty}^{\infty} \delta(x) dx = 1 \quad (2.35)$$

Wird nun eine kontinuierliche Funktion $g(x)$ mit dem Dirac-Funktional $\delta(x)$ multipliziert, ergibt sich eine neue Funktion $\bar{g}(x)$, die lediglich einen einzigen Puls an der Position $x = 0$ mit der Höhe von $g(0)$ besitzt.

$$\bar{g}(x) = g(x) \cdot \delta(x) = \begin{cases} g(0) & \text{für } x = 0 \\ 0 & \text{sonst} \end{cases} \quad (2.36)$$

Somit wird ein einzelner diskreter Abtastwert generiert. Wird das Dirac-Funktional um eine Distanz x_0 verschoben, kann an jeder beliebigen Stelle $x = x_0$ die Funktion $g(x)$ abgetastet werden.

$$\bar{g}(x) = g(x) \cdot \delta(x - x_0) = \begin{cases} g(x_0) & \text{für } x = x_0 \\ 0 & \text{sonst} \end{cases} \quad (2.37)$$

Darin ist $\delta(x - x_0)$ das um x_0 verschobene Dirac-Funktional.

Um nun mehrere Punkte einer Funktion abzutasten, werden N individuell verschobene Exemplare des Dirac-Funktional jeweils mit der Funktion $g(x)$ multipliziert.

$$\begin{aligned} \bar{g}(x) &= g(x)[\delta(x - 1) + \delta(x - 2) + \dots + \delta(x - N)] \\ &= g(x) \sum_{i=1}^N \delta(x - i) \end{aligned} \quad (2.38)$$

Diese verschobenen Einzelpulse werden als *Pulsfolge* bezeichnet. Wird die Pulsfolge in beide Richtungen bis ins Unendliche erweitert, wird die *Kammfunktion* $\text{III}(x)$ erzeugt.

$$\text{III}(x) = \sum_{i=-\infty}^{\infty} \delta(x - i) \quad (2.39)$$

¹⁰benannt nach Paul Adrien Maurice Dirac (1902-1984), britischer Physiker und Nobelpreisträger der Physik

Die Abtastung eines kontinuierlichen Signals $g(x)$ in regelmäßigen Abständen τ kann nun realisiert werden.

$$\bar{g}(x) = g(x) \cdot \text{III}\left(\frac{x}{\tau}\right) \text{ für } \tau > 0 \quad (2.40)$$

Die Abtastung der kontinuierlichen Funktion hat massive Auswirkungen auf das resultierende diskrete Frequenzspektrum.

Die Fourier-Transformierte der Kammfunktion $\text{III}(x)$ ist wiederum eine Kammfunktion.

$$\text{III}\left(\frac{x}{\tau}\right) \circledast \tau \text{III}(\tau f_x) \quad (2.41)$$

Das diskrete Frequenzspektrum bleibt somit nicht von der Abtastung verschont, da das Produkt zweier Funktionen im Orts- oder Spektralraum einer linearen Faltung im jeweils konjuguierten Raum entspricht.

$$g(x) \cdot \text{III}\left(\frac{x}{\tau}\right) \circledast G(f_x) * \tau \text{III}(\tau f_x) \quad (2.42)$$

Die Faltung einer Funktion $f(x)$ mit einem Dirac-Funktional $\delta(x)$ ist wiederum die ursprüngliche Funktion ($f(x) * \delta(x) = f(x)$). Ist das Dirac-Funktional nun um die Distanz x_0 verschoben, verschiebt sich auch entsprechend die ursprüngliche Funktion.

$$f(x) * \delta(x - x_0) = f(x - x_0) \quad (2.43)$$

Daraus folgt nun, dass das Fourierspektrum des abgetasteten Signals $\bar{G}(f_x)$ das Spektrum $G(f_x)$ des kontinuierlichen Signals an jedem Puls des abgetasteten Spektrums repliziert. Das resultierende Fourierspektrum ist somit periodisch und hat eine Periodenlänge

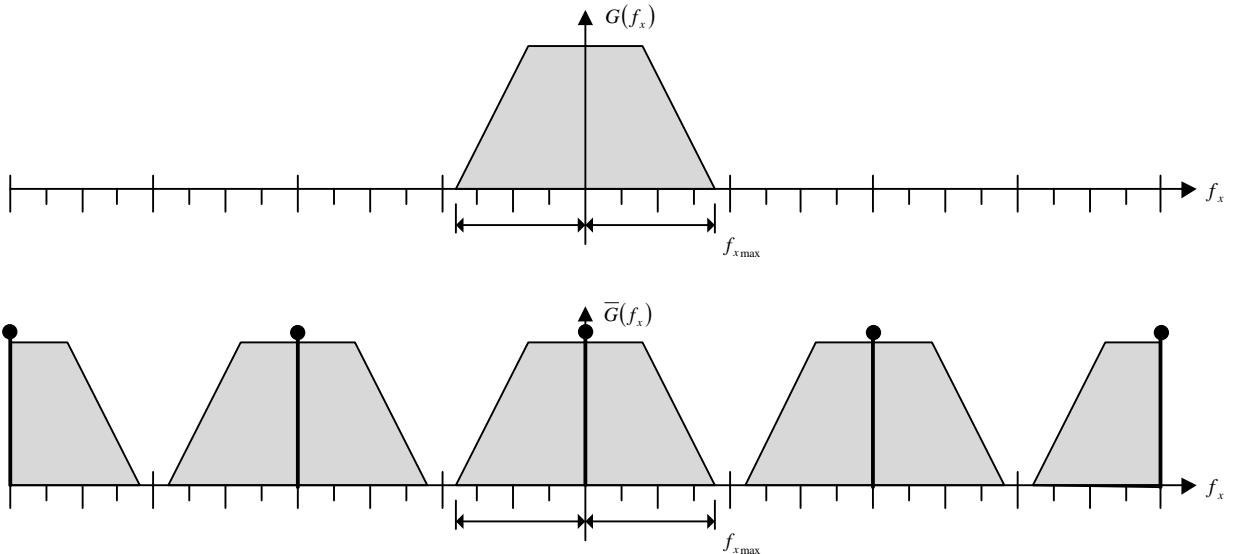


Abbildung 2.4.: Kontinuierliches Frequenzspektrum $G(f_x)$ und Auswirkungen der Abtastung auf ein das Frequenzspektrum $\bar{G}(f_x)$ der diskretisierten Ausgangsfunktion

von $\frac{2\pi}{\tau}$.

Für ein diskretes, periodisches Signal genügt also, eine endliche Abfolge von M Abtast-

werten, um sein diskretes Signal $g(m)$ und das zugehörige Frequenzspektrum $G(k)$ vollständig abzubilden. Die Vorwärtstransformation der diskreten Fourier-Transformation für ein diskretes Signal $g(m)$ der Länge M ($m = 0 \dots M - 1$) ist definiert als

$$\mathcal{F}_D\{g(m)\} = G(k) = \sum_{m=0}^{M-1} g(m) \cdot e^{-i2\pi \frac{mk}{M}}. \quad (2.44)$$

Analog dazu die DFT für den zweidimensionalen Fall, zum Beispiel ein Bild mit den Maßen $M \times N$.

$$\mathcal{F}_D\{g(m, n)\} = G(k, l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} g(m, n) \cdot e^{-i2\pi \left(\frac{mk}{M} + \frac{nl}{N} \right)} \quad (2.45)$$

Wobei k ($k = 0 \dots M - 1$) die Zeilen und l ($l = 0 \dots N - 1$) die Spalten beschreiben. Die Inverse der Transformation (iDFT) wird wie folgt aufgestellt:

$$\mathcal{F}_D^{-1}\{G(k, l)\} = g(m, n) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} G(k, l) \cdot e^{i2\pi \left(\frac{mk}{M} + \frac{nl}{N} \right)} \quad (2.46)$$

Die schnelle Fourier-Transformation (FFT)

Die direkte Anwendung der Gleichungen 2.45 und 2.46 in einer Software würde die praktische Arbeit mit der diskreten Fourier-Transformation nur sehr mühsam ermöglichen. Eine schnelle Implementierung der DFT ist die von Cooley¹¹ und Turkey¹² [16] erstmals publizierte *schnelle Fourier-Transformation* (engl. *fast Fourier transform - FFT*).

Die Berechnungen des FFT-Algorithmus sind so ausgelegt, dass gleichartige Zwischenergebnisse nur einmal berechnet werden und in optimaler Weise mehrfach wieder verwertet werden. Dies hat enorme Auswirkungen auf den Zeitbedarf. Bei einem eindimensionalen Signal der Signallänge $M = 2^n$ reduziert sich der Zeitbedarf von M^2 auf $M \log_2 M$ [17]. Desto größer die Signallänge ist, umso höher ist die Beschleunigung durch den Einsatz des Algorithmus. Beispielsweise wird die Zahl der arithmetischen Operationen bei einem Bild der Größe 1024×1024 Bildpunkten (Pixel) von etwas mehr als 10^{12} auf etwa $2 \cdot 10^7$ verringert.

Nachfolgend soll der grundlegende Algorithmus von Cooley und Turkey erklärt werden. Er bildet nach wie vor die Grundlage für moderne Ansätze der FFT. Dieser Ansatz geht von einer Signallänge aus, die einer Zweierpotenz entspricht, für andere Ansätze gilt diese Limitierung nicht mehr. An dieser Stelle soll nur auf die eindimensionale Variante des Algorithmus eingegangen werden, da die zweidimensionale Berechnung einer Aneinanderreihung eindimensionaler Berechnungen gleich kommt.

¹¹Dr. James W. Cooley (*1926), US-amerikanischer Mathematiker

¹²John Wilder Tukey (1915-2000), US-amerikanischer Statistiker

Ausgangspunkt ist die Gleichung der DFT (Gl. 2.44).

$$\mathcal{F}_D\{g(m)\} = G(k) = \sum_{m=0}^{M-1} g(m) \cdot e^{-i2\pi \frac{mk}{M}}$$

$M = 2^n$ lässt sich nunmehr als $M = 2K$ mit $K > 0$ ausdrücken, und die Gleichung lässt sich mit einer geraden und einer ungeraden Summe definieren.

$$\begin{aligned} \mathcal{F}_D\{g(m)\} = G(k) &= \sum_{m=0}^{2K-1} g(m) \cdot e^{-i2\pi \frac{mk}{2K}} \\ &= \sum_{m=0}^{K-1} g(2m) \cdot e^{-i2\pi \frac{(2m)k}{2K}} + \sum_{m=0}^{K-1} g(2m+1) \cdot e^{-i2\pi \frac{(2m+1)k}{2K}} \\ &= \sum_{m=0}^{K-1} g(2m) \cdot e^{-i2\pi \frac{mk}{K}} + \sum_{m=0}^{K-1} g(2m+1) \cdot e^{-i2\pi \frac{mk}{K}} \cdot e^{-i2\pi \frac{k}{2K}} \end{aligned} \quad (2.47)$$

Somit lässt sich definieren, dass für

$$\begin{aligned} \mathcal{F}_{\text{gerade}}\{g(m)\} &= \sum_{m=0}^{K-1} g(2m) \cdot e^{-i2\pi \frac{mk}{K}} \\ \mathcal{F}_{\text{ungerade}}\{g(m)\} &= \sum_{m=0}^{K-1} g(2m+1) \cdot e^{-i2\pi \frac{mk}{K}} \end{aligned} \quad (2.48)$$

mit $k = 0 \dots K-1$ gilt:

$$\mathcal{F}_D\{g(m)\} = \mathcal{F}_{\text{gerade}}\{g(m)\} + \mathcal{F}_{\text{ungerade}}\{g(m)\} \cdot e^{-i2\pi \frac{k}{2K}} \quad (2.49)$$

Bei Betrachtung der Gleichungen 2.48 und 2.49 ist festzustellen, dass bei einer Transformation mit M Abtastpunkten die Ausgangsgleichung in zwei ähnliche Fragmente (Gl. 2.48) geteilt werden kann. Werden die Fragmente sequentiell berechnet, lassen sich bereits berechnete Operationen wieder verwenden [18].

Auf den tatsächlich implementierten FFT-Algorithmus wird in 4.2.4 näher eingegangen.

2.2.4. Der Lösungsalgorithmus

Da die Fourier-Transformation den Zusammenhang zwischen Orts- und Frequenzraum beschreibt, eignet sie sich gut für die Implementierung eines Algorithmus auf der Grundlage der Intensitäts-Transportgleichung (Gl. 2.22). Differentialoperatoren können geschickt ersetzt werden, da eine Differentiation im Ortsraum eine Multiplikation im Frequenzraum bedeutet und eine Integration im Ortsraum eine Division im Frequenzraum. Als Beispiel soll hier eine eindimensionale Ableitung des Signals $g(x, y)$ durch die Fourier-Transformation dargestellt werden.

$$\begin{aligned}\frac{\partial}{\partial x} g(x, y) &= \frac{\partial}{\partial x} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} G(f_x, f_y) \cdot e^{i2\pi(f_x x + f_y y)} df_x df_y \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} G(f_x, f_y) \cdot e^{i2\pi f_y y} \cdot \frac{\partial}{\partial x} \cdot e^{i2\pi f_x x} df_x df_y \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} i2\pi f_x \cdot G(f_x, f_y) \cdot e^{i2\pi(f_x x + f_y y)} df_x df_y\end{aligned}\quad (2.50)$$

So ist die Ableitung nach x im Ortsraum zu einer Multiplikation mit dem Faktor $i2\pi f_x$ im Frequenzraum geworden. Sie lässt sich also wie folgt ersetzen.

$$\frac{\partial}{\partial x} g(x, y) = \mathcal{F}^{-1} \{ i2\pi f_x \cdot \mathcal{F} \{ g(x, y) \} \} \quad (2.51)$$

Auch eine mehrfache Ableitung lässt sich im Frequenzraum durchführen [19].

$$\frac{\partial^n}{\partial x^n} g(x, y) = \mathcal{F}^{-1} \{ (i2\pi f_x)^n \cdot \mathcal{F} \{ g(x, y) \} \} \quad (2.52)$$

Dementsprechend gilt für die Integration:

$$\int g(x, y) dx^n = \mathcal{F}^{-1} \{ (i2\pi f_x)^{-n} \cdot \mathcal{F} \{ g(x, y) \} \} \quad (2.53)$$

Diese Erkenntnisse sollen nun genutzt werden, um die Differentialoperationen, die im ITG-Algorithmus genutzt werden sollen, zu ersetzen:

$$\nabla_{\perp} = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} = \begin{pmatrix} \mathcal{F}^{-1} \{ i2\pi f_x \cdot \mathcal{F} \} \\ \mathcal{F}^{-1} \{ i2\pi f_y \cdot \mathcal{F} \} \end{pmatrix} \quad (2.54)$$

$$\nabla_{\perp}^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} = \mathcal{F}^{-1} \{ (i2\pi f_x)^2 \cdot \mathcal{F} \} + \mathcal{F}^{-1} \{ (i2\pi f_y)^2 \cdot \mathcal{F} \} \quad (2.55)$$

Dank der Linearität der Fourier-Transformation lässt sich der Operator ∇_{\perp}^2 noch weiter vereinfachen.

$$\begin{aligned}\nabla_{\perp}^2 &= \mathcal{F}^{-1} \{ ((i2\pi f_x)^2 + (i2\pi f_y)^2) \cdot \mathcal{F} \} \\ &= -4\pi^2 \cdot \mathcal{F}^{-1} \{ (f_x^2 + f_y^2) \cdot \mathcal{F} \}\end{aligned}\quad (2.56)$$

Nachfolgend werden die inversen Operatoren dargestellt.

$$\nabla_{\perp}^{-2} = -(2\pi)^{-2} \cdot \mathcal{F}^{-1} \left\{ (f_x^2 + f_y^2)^{-1} \cdot \mathcal{F} \right\} \quad (2.57)$$

Nun können alle Differentialoperatoren durch geeignete Operationen im Frequenzbereich ersetzt werden. Somit kann die Gleichung der symbolischen Lösung der ITG (Gl. 2.25) unter Verwendung der Fourier-Transformation definiert werden.

Zuerst soll die Gleichung in ihre x - und y -Anteile separiert werden.

$$\begin{aligned} \Phi(x) &= \frac{1}{4\pi^2} \cdot \mathcal{F}^{-1} \left\{ \frac{f_x}{f_x^2 + f_y^2} \cdot \mathcal{F} \left\{ I^{-1} \cdot \mathcal{F}^{-1} \left\{ \frac{f_x}{f_x^2 + f_y^2} \cdot \mathcal{F} \left\{ k \frac{\partial I(\vec{r})}{\partial z} \right\} \right\} \right\} \right\} \\ \Phi(y) &= \frac{1}{4\pi^2} \cdot \mathcal{F}^{-1} \left\{ \frac{f_y}{f_x^2 + f_y^2} \cdot \mathcal{F} \left\{ I^{-1} \cdot \mathcal{F}^{-1} \left\{ \frac{f_y}{f_x^2 + f_y^2} \cdot \mathcal{F} \left\{ k \frac{\partial I(\vec{r})}{\partial z} \right\} \right\} \right\} \right\} \\ \Phi(\vec{r}) &= \Phi(x) + \Phi(y) \end{aligned} \quad (2.58)$$

Für die praktische Anwendung, also die Nutzung mit Bilddaten, muss die Gleichung mit geeigneter Pixelindizierung versehen werden,

$$\begin{aligned} \Phi(x) &= k \frac{M^2 \Delta x^2}{4\pi^2} \cdot \mathcal{F}^{-1} \left\{ \frac{x}{x^2 + y^2} \cdot \mathcal{F} \left\{ I^{-1} \cdot \mathcal{F}^{-1} \left\{ \frac{x}{x^2 + y^2} \cdot \mathcal{F} \left\{ k \frac{\partial I(\vec{r})}{\partial z} \right\} \right\} \right\} \right\} \\ \Phi(y) &= k \frac{M^2 \Delta y^2}{4\pi^2} \cdot \mathcal{F}^{-1} \left\{ \frac{y}{x^2 + y^2} \cdot \mathcal{F} \left\{ I^{-1} \cdot \mathcal{F}^{-1} \left\{ \frac{y}{x^2 + y^2} \cdot \mathcal{F} \left\{ k \frac{\partial I(\vec{r})}{\partial z} \right\} \right\} \right\} \right\} \\ \Phi(\vec{r}) &= \Phi(x) + \Phi(y) \end{aligned} \quad (2.59)$$

mit k als Kreiswellenzahl, x und y als Laufvariablen der Pixel, Δx und Δy die geometrischen Pixelgrößen und M als Bildbreite und Bildhöhe.

Als Eingangsgröße erscheint nun bloß die Intensitätsverteilung $I_0(\vec{r})$ und deren Ableitung in Ausbreitungsrichtung $\frac{\partial}{\partial z} I(\vec{r})$. Dabei symbolisiert I_0 eine fokussierte Aufnahme des zu untersuchenden Objekts. $\frac{\partial}{\partial z} I(\vec{r})$ wird praktisch derart umgesetzt, dass zwei defokussierte Aufnahmen des Objekts angefertigt werden, eine mit leichter Defokussierung in Ausbreitungsrichtung ($I_+(\vec{r})$) und die andere mit der gleichen Defokussierung in negativer Ausbreitungsrichtung ($I_-(\vec{r})$). Mit diesen zwei Aufnahmen und deren einseitiger Abstand zur Fokusebene (Defokusdistanz) ergibt sich ein Differenzenquotient, der die Ableitung nähert.

$$\frac{\partial}{\partial z} I(\vec{r}) \approx \frac{I_+(\vec{r}) - I_-(\vec{r})}{2z} \quad (2.60)$$

Somit ergibt sich der endg  lige L  sungsalgorithmus zur Phasenrekonstruktion.

$$\begin{aligned}\Phi(x) &= \frac{k}{2z} \frac{M^2 \Delta x^2}{4\pi^2} \cdot \mathcal{F}^{-1} \left\{ \frac{x}{x^2 + y^2} \cdot \mathcal{F} \left\{ I^{-1} \cdot \mathcal{F}^{-1} \left\{ \frac{x}{x^2 + y^2} \cdot \mathcal{F} \{ I_+(\vec{r}) - I_-(\vec{r}) \} \right\} \right\} \right\} \\ \Phi(y) &= \frac{k}{2z} \frac{M^2 \Delta y^2}{4\pi^2} \cdot \mathcal{F}^{-1} \left\{ \frac{y}{x^2 + y^2} \cdot \mathcal{F} \left\{ I^{-1} \cdot \mathcal{F}^{-1} \left\{ \frac{y}{x^2 + y^2} \cdot \mathcal{F} \{ I_+(\vec{r}) - I_-(\vec{r}) \} \right\} \right\} \right\} \\ \Phi(\vec{r}) &= \Phi(x) + \Phi(y)\end{aligned}\tag{2.61}$$

Zu beachten gibt es bei diesem Algorithmus, dass x und y nicht gleichzeitig Null sein d  rfen. Dabei darf das Feld $I_0(\vec{r})$ nur Werte enthalten, die ungleich Null sind, um eine Division durch Null zu vermeiden. Nachfolgend wird der Algorithmus als Ablaufdiagramm so aufgezeigt, wie er schlie  lich f  r quadratische Bilder ($M = N$) implementiert wird. Eine grundlegende Beschreibung des Algoritihmus ist auch unter [20] zu finden.

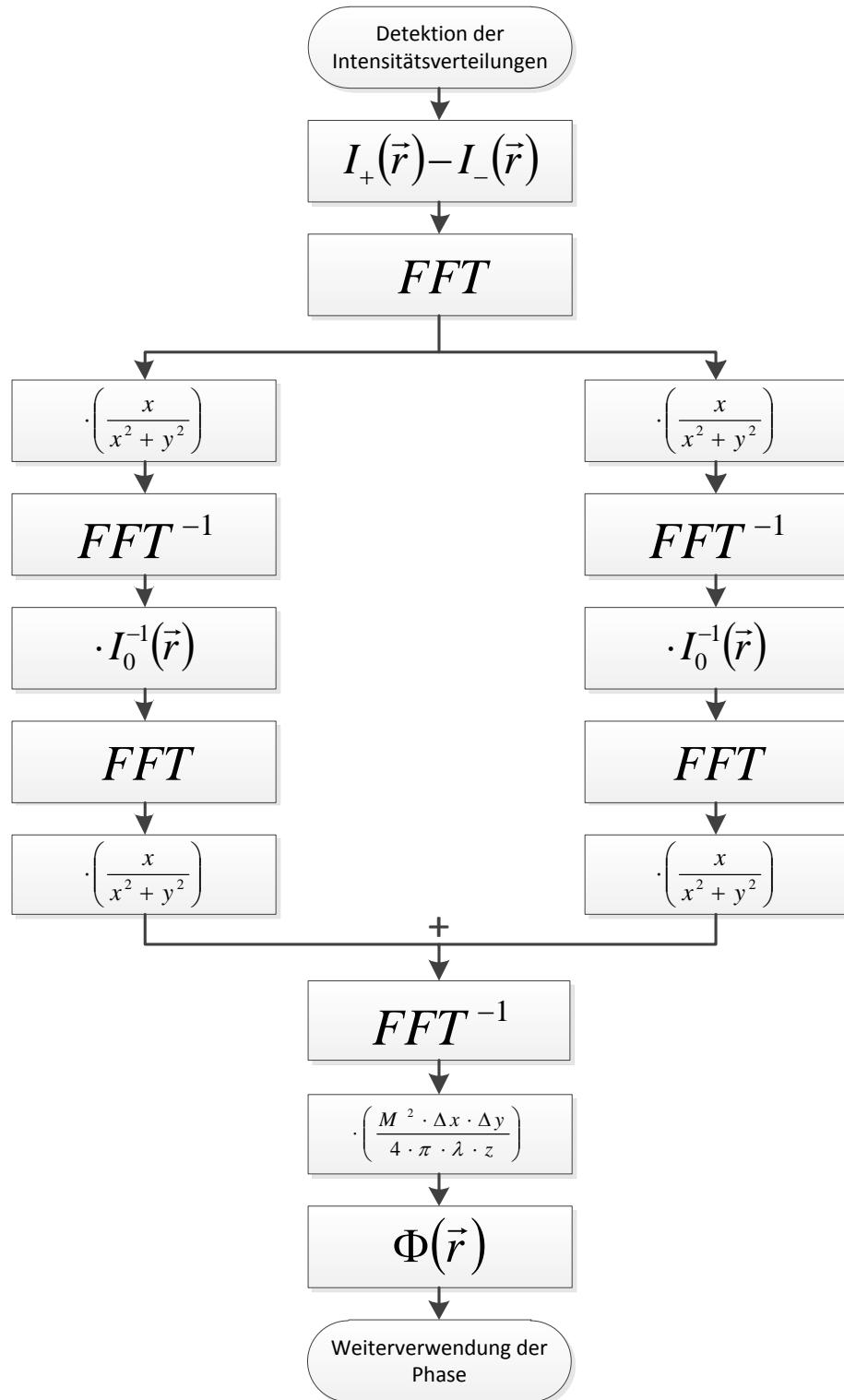


Abbildung 2.5.: Ablaufdiagramm des ITG-Algorithmus' zur Phasenrekonstruktion quadratischer Eingangssignale

2.3. Mikroskop

Ein *Mikroskop* erlaubt es dem Betrachter, Objekte vergrößert darzustellen, deren Größe unterhalb der Auflösungsgrenze des menschlichen Auges¹³ liegt. In diesem Kapitel soll die grundlegende Funktionsweise der Lichtmikroskopie gezeigt werden. Eine nähere Erklärung ist in den Arbeiten [5] und [21] zu finden.

Ein Lichtmikroskop setzt sich aus mehreren optischen Elementen zusammen, die oft in Gruppen zusammengefasst werden. Um unnötige Verwirrungen zu vermeiden, werden nachfolgend Gruppen als einfache Linsen dargestellt.

In Abbildung 2.6 wird der Abbildungsstrahlengang durch ein einfaches Mikroskop schematisch dargestellt, präzisere Darstellungen sind im Unterkapitel 2.3.1 zu finden. In dieser Arbeit wird nur auf Mikroskope mit *Unendlich-Optik* eingegangen. Dabei hat das *Objektiv* die Aufgabe, das in seiner Brennebene befindliche Objekt ins Unendliche abzubilden, d.h hinter dem Objektiv befindet sich ein paralleles Strahlenbündel. In diesem Bereich ist es möglich, ohne Störung der Bildentstehung Filter oder *Strahlteiler* (Kap. 3) einzusetzen, zumal der mechanische und optische Abstand zwischen Objektiv und Tubuslinse frei gewählt werden kann. Erst die Tubuslinse bildet das parallele Strahlenbündel in das Zwischenbild ab. Ein weiterer Vorteil der Unendlich-Optik zeigt sich darin, dass das Zwischenbild bereits durch die Tubuslinse weitestgehend von Abberationen befreit ist [22]. Bereits hier ist das Einsetzen eines digitalen Detektors, zum Beispiel eines *CCD-Chips*, als Aufnahmefläche möglich. Mit dem Okular lässt sich das Zwischenbild direkt betrachten.

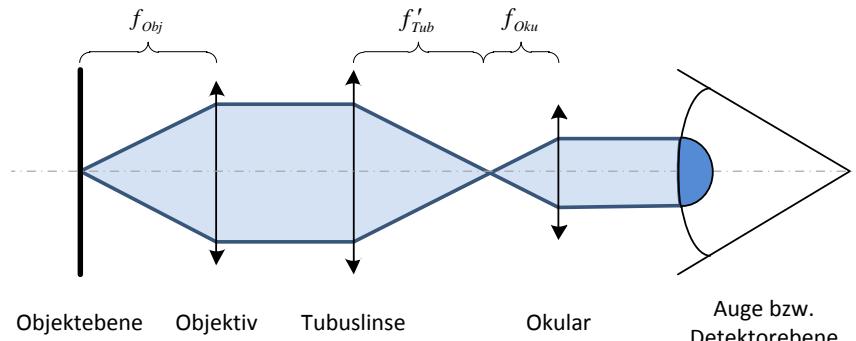


Abbildung 2.6.: Simpler Abbildungsstrahlengang im Mikroskop

Die *Mikroskopvergrößerung* β , also die Vergrößerung des gesamten optischen Systems, ist das Produkt aus den Lupenvergrößerungen von Objektiv β_{Obj} , Okular β_{Oku} und dem Tubusfaktor q_∞ .

$$\beta = \beta_{Obj} \cdot q_\infty \cdot \beta_{Oku} = \frac{250\text{mm}}{f'_{Obj}} \cdot \frac{f'_{Tub}}{250\text{mm}} \cdot \frac{250\text{mm}}{f'_{Oku}} \quad (2.62)$$

¹³Zwei Punkte, die $75\mu\text{m}$ auseinander liegen, sind in der Regel mit 25cm Abstand noch mit dem bloßen Auge erkennbar [10].

Des Weiteren gibt es zwei Arten der Objektbeleuchtung in der normalen Lichtmikroskopie: zum einen die Beleuchtung durch das Objekt hindurch (Durchlichtmikroskopie - Kap. 2.3.1.1), zum anderen kann das Objekt von Oben beleuchtet werden (Auflichtmikroskopie - Kap. 2.3.1.2).

2.3.1. Köhlersche Beleuchtung

Die *Köhlersche Beleuchtung*¹⁴ ist das in der modernen Mikroskopie am weitesten verbreitete Beleuchtungsverfahren. Ziel ist es hierbei, die Lichtleistung der Lichtquelle insofern optimal auszunutzen, dass Objekt und Netzhaut des Betrachters homogen ausgeleuchtet werden [23]. Nur so lässt sich das Auflösungsvermögen des Mikroskops ausschöpfen. Da die Größe der ausgeleuchteten Fläche im Objekt und die der Beleuchtungssapertur unabhängig einstellbar sind, kann die Beleuchtung an das Objekt angepasst werden. Hierdurch wird Streulicht vermieden und nur der gerade sichtbare Bereich des Objekts ausgeleuchtet.

Um die Köhlersche Beleuchtung zu ermöglichen, müssen die folgenden vier Bedingungen erfüllt sein [24]:

- Der Kollektor bildet die Lichtquelle (Lampe) vergrößert ab. Am Ort des Lampenbildes liegt die Aperturblende.
- Das Lampenbild und die Aperturblende liegen in der bildseitigen Brennweite des Kondensors.
- Der Kondensor bildet die Leuchtfeldblende verkleinert in die Objektebene ab.
- Die Leuchtfeldblende und der Kollektor liegen möglichst nah beieinander.

In der Abbildung 2.7 sind beispielhaft Abbildungs- und Beleuchtungsstrahlengang eines Durchlichtmikroskops angegeben, dass der Köhlerschen Beleuchtung genügt.

Im Abbildungsstrahlengang ist zu erkennen, dass die Leuchtfeldblende über den Kondensor in die Objektebene abgebildet wird. Mittels der Leuchtfeldblende lässt sich also der ausgeleuchtete Bereich des Objekts einstellen. Dem Beleuchtungsstrahlengang ist zu entnehmen, dass die Lichtquelle über den Kollektor in die Aperturblende abgebildet wird. Da die Aperturblende in der Brennebene des Kondensors steht, wird das Bild der Lichtquelle ins Unendliche abgebildet und somit die Objektebene homogen ausgeleuchtet.

¹⁴benannt nach August Karl Johann Valentin Köhler (1866-1948), deutscher Professor und Mitarbeiter bei Zeiss (heute: Carl Zeiss) in Jena

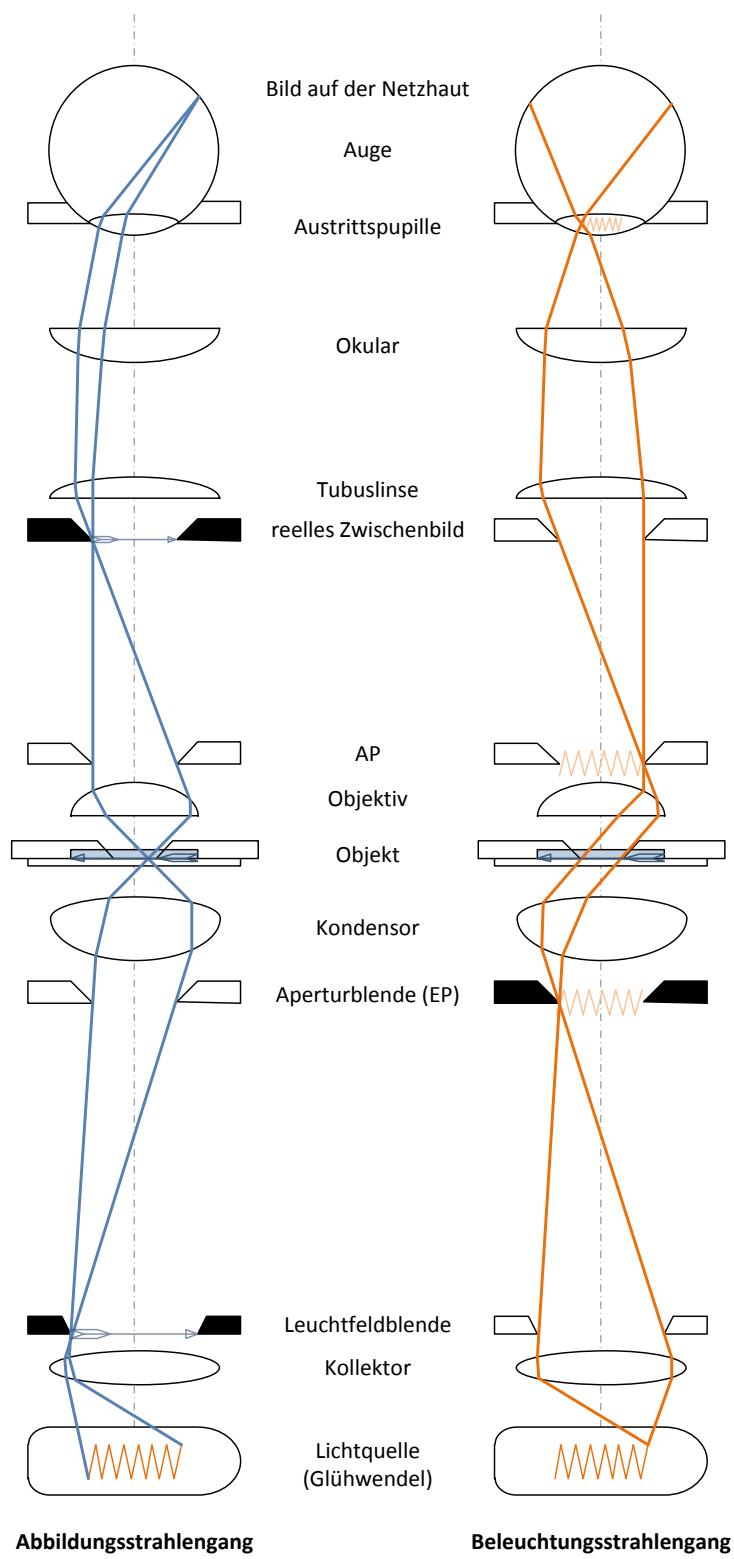


Abbildung 2.7.: Abbildungs- und Beleuchtungsstrahlengang im Durchlichtmikroskop mit Köhlerscher Beleuchtung

2.3.1.1. Durchlichtmikroskop

Ein Mikroskop im Durchlichtaufbau ermöglicht das Betrachten von teiltransparenten Objekten. Hierbei liegt die Lichtquelle im Strahlengang vor dem Objekt und beleuchtet dieses entgegengesetzt der Betrachtungsrichtung, d.h. von unten. Mit speziellen Mikroskopieverfahren ist auch das Betrachten von gleichmäßig transparenten Objekten, sogenannten *Phasenobjekten*, möglich [10].

Beleuchtungsstrahlengang

Abbildungsstrahlengang

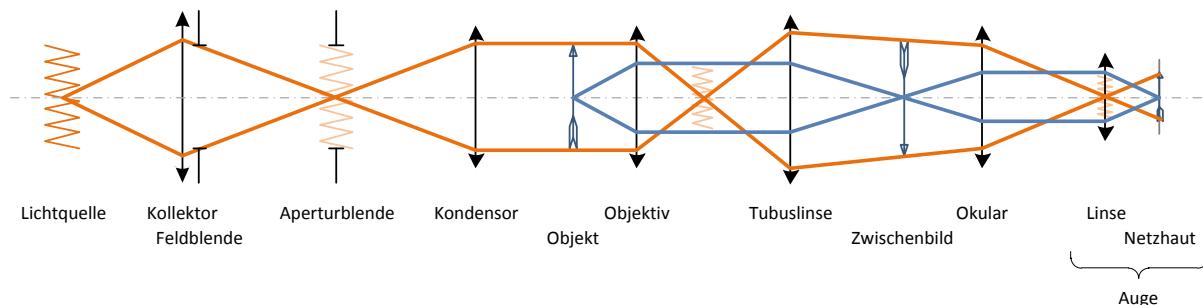


Abbildung 2.8.: Schematischer Abbildungs- und Beleuchtungsstrahlengang eines Durchlichtmikroskops unter Berücksichtigung der Köhlerschen Beleuchtung

2.3.1.2. Auflichtmikroskop

Der Auflichtaufbau ermöglicht die Darstellung nicht oder nur sehr schwach transparenter Objekte. Dabei liegt der Beleuchtungsstrahlengang sozusagen parallel zum Abbildungsstrahlengang. Das Objekt wird in Betrachtungsrichtung, von oben, beleuchtet.

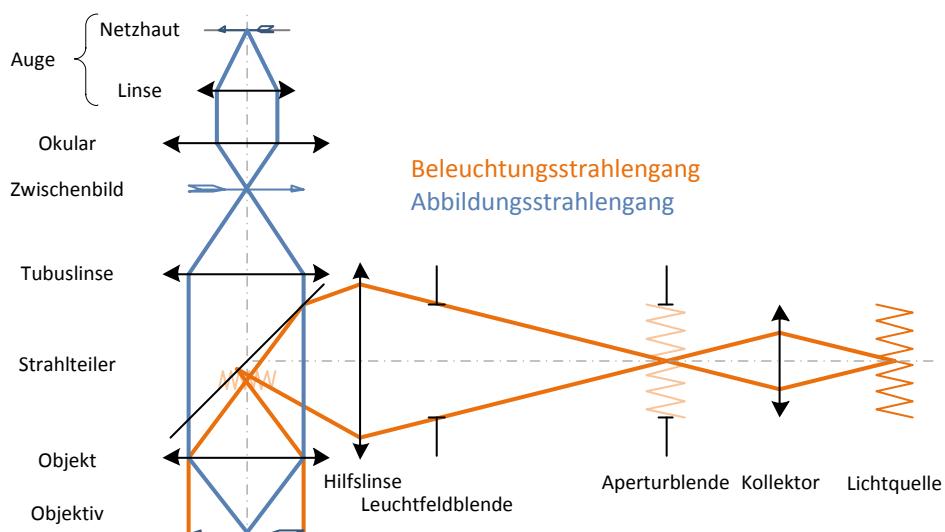


Abbildung 2.9.: Schematischer Abbildungs- und Beleuchtungsstrahlengang eines Auflichtmikroskops unter Berücksichtigung der Köhlerschen Beleuchtung

2.3.2. Beugungsbegrenztes Auflösungsvermögen

Ein Mikroskop ermöglicht die Darstellung von Strukturen, die nahe an die mittlere Wellenlänge des sichtbaren Lichts (Kap. 2.1) kommen. Da hier die Abbildung von Objektpunkten über die Wellenoptik betrachtet werden muss, ist die Beugung zu berücksichtigen, die das optische Auflösungsvermögen von Systemen begrenzt. Die Gestalt der Beugungsfigur wird durch die Form der beugenden Öffnung bestimmt. Im Mikroskop sind lediglich kreisrunde, begrenzende, optische Elemente vorhanden. Die Intensitätsverteilung eines Bildpunktes folgt somit dem Verlauf der *Bessel-Funktion* [10]. Die Form der Beugungsfigur einer kreisrunden beugenden Öffnung ist das sogenannte *Airy-Scheibchen*.

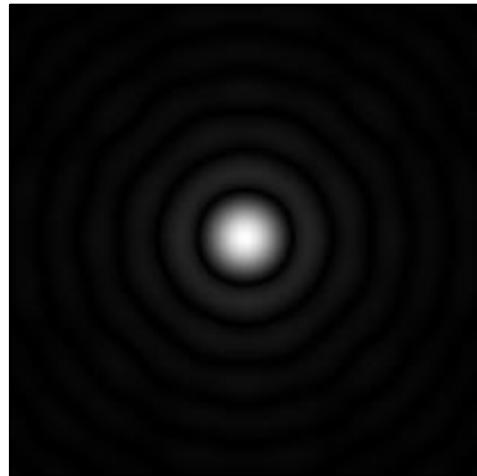


Abbildung 2.10.: Betragbild der Intensität eines Airy-Scheibchens

Der Abstand ρ vom Hauptmaximum zum ersten Minimum entspricht dem Radius des ersten schwarzen "Rings"; Er berechnet sich über die Formel [7]:

$$\rho = 0,61 \cdot \frac{\lambda}{N.A.} \quad (2.63)$$

Die Abbildungsgrenze wird somit bestimmt durch die Wellenlänge λ und die numerische Apertur $N.A. = n \cdot \sin(\sigma_{max})$ (Abb. 2.11) der abbildenden Optik. Bildpunkte unterhalb der Abbildungsgrenze können nicht mehr mit einfachen mikroskopischen Techniken verarbeitet werden.

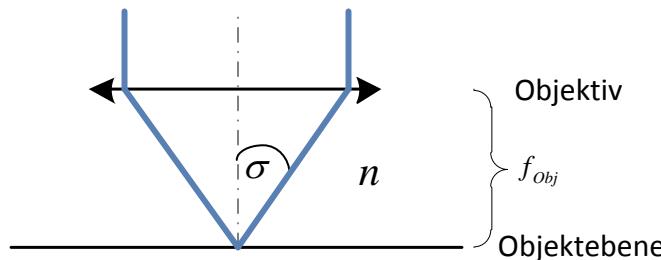


Abbildung 2.11.: Numerische Apertur $N.A.$ am Mikroskopobjektiv

3. Laboraufbau

In diesem Kapitel soll der praktische Aufbau zur Phasenrekonstruktion dargestellt werden. Dieser wurde im Rahmen der Diplomarbeit [5] konstruiert und wird dort näher erläutert.

3.1. Konzept

Wie in Kapitel 2.2.4 bereits erwähnt, wird zur Phasenrekonstruktion mittels der Intensitäts-Transportgleichung neben dem fokussierten Bild je ein um z defokussiertes Bild in Betrachtungsrichtung und in entgegengesetzter Richtung (Gl. 2.60) benötigt.

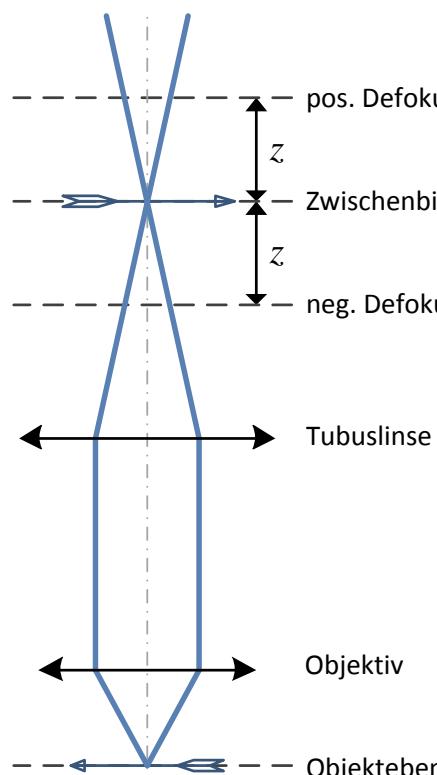


Abbildung 3.1.: Defokusebenen im Abbildungsstrahlengang des Mikroskop

Bei dem genutzten Mikroskop ist die Zwischenbildebene hinter der Tubuslinse bereits ausreichend abberationsfrei und ist somit als Detektorebene geeignet (Kap. 2.3). Der Orte der Zwischenbildebene, der für die Aufnahme des fokussierten Bildes vorgesehen ist und die Ebenen im Abbildungsstrahlengang, in der die defokussierten Aufnahmen erstellt werden sollen, sind in Abbildung 3.1 beispielhaft dargestellt.

Zur Detektion der Intensitätsverteilungen werden FireWire-Kameras mit CCD-Chips eingesetzt. Da diese nicht transparent sind, ist es nicht möglich, die Chips hintereinander in den Strahlengang zu setzen. Der Strahlengang muss also nach der Tubuslinse und vor der Zwischenbildebene in drei Arme aufgespalten werden. Die Abbildungen in diesen Armen sollten möglichst identisch sein und keine zusätzliche Abberation durch die Aufteilung erfahren. Aus Arbeit [5] folgte, dass ein spezieller kubischer Strahlteiler den Anforderungen am besten gerecht wird. Dieser Strahlteiler spaltet das ankommende Strahlenbündel gleichmäßig in drei Bündel auf. Da die drei so entstandenen Arme als identisch anzunehmen sind, kann in die Fokusebene jedes Arms ein Detektor eingebracht werden.

Der schematische Aufbau dieses *Drei-Kamera-Systems* ist in Abbildung 3.2 zu

sehen.

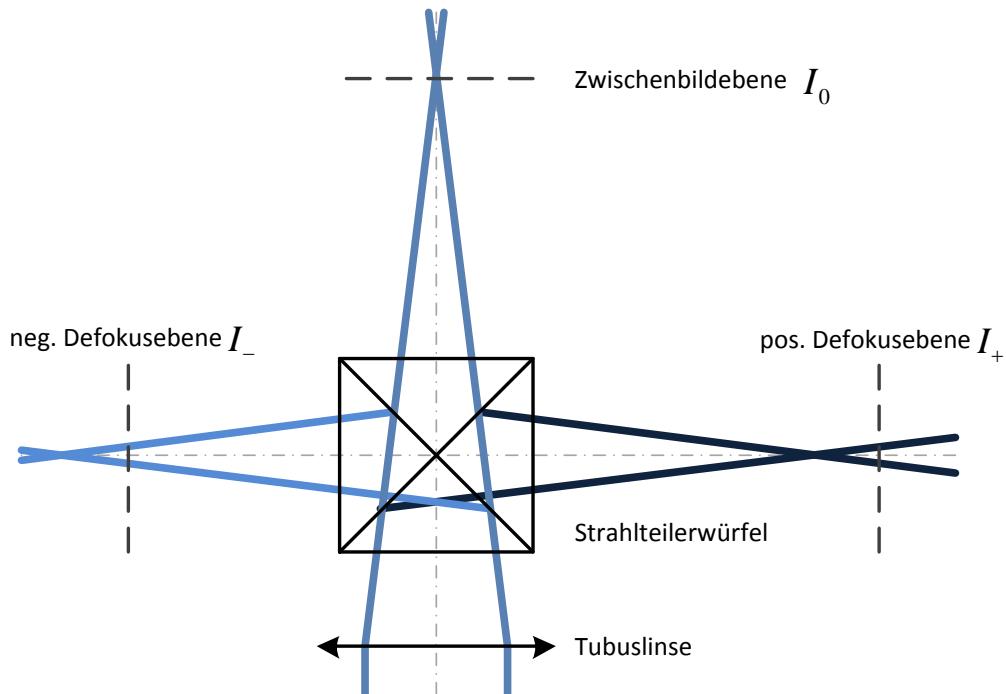


Abbildung 3.2.: Strahlengang des Drei-Kamera-Systems zur Aufnahme der drei benötigten Intensitätsverteilungen

3.2. Umsetzung

Als Mikroskope stehen ein Leica DMRM und ein Leica DMR zur Verfügung. Das Leica DMRM ist mit Auflichtbeleuchtung und das DMR mit Durchlichtbeleuchtung ausgestattet. In der Tabelle 3.1 ist eine Bauteilliste und in Abbildung 3.3 ist eine Skizze des Kamera-Systems zu finden. Weitere Erläuterungen und Datenblätter zum Aufbau sind in der Diplomarbeit [5] zu finden.

Mikroskop	Leica DMR oder Leica DMRM
Kubischer Strahlteilerwürfel	Spezielle Anfertigung
CCD-Kameras	Drei AVT Pike 145B FireWire-Kameras
Skelettwürfel	Zentraler Montagepunkt
Justierbarer Prismenträger	Zur flexiblen Aufnahme des Strahlteilerwürfels
Tuben/ Distanzringe	Einstellung der Defokusdistanz z
Anschlussadapter für Kameras	Montage der Kameras an Skelettwürfel
Anschlussstück zum Mikroskop	Eigenanfertigung Dipl. Ing. Anton Kraus

Tabelle 3.1.: Bauteilliste des Kamera-Systems

Die gewünschte bildseitige Defokusdistanz lässt sich durch Anschrauben von Tubenauf-

sätzen und Distanzringen an den Kameras einstellen. Aus dem axialen Abbildungsmaßstab eines optischen Systems ergibt sich die Berechnung der objektabhängig bildseitigen Defokusdistanz z .

$$z = \frac{\hat{z}}{2} \cdot \beta^2 \quad (3.1)$$

Hierbei ist \hat{z} der Defokus in der Objektebene und β die Mikroskopvergrößerung (Gl. 2.62). Da kein Okular mehr im Aufbau existiert, ergibt sich die Mikroskopvergrößerung mit

$$\beta = \beta_{Obj} \cdot q_\infty = \frac{f'_{Tub}}{f'_{Obj}} . \quad (3.2)$$

Die Kameras werden drehbar in die Anschlussadapter eingesteckt. So kann jede Kamera unabhängig gedreht werden, und die einzelnen Bilder können gegeneinander rotiert werden. Der justierbare Prismenhalter ermöglicht zusätzlich durch das Verschieben des Strahlteilers die Anpassung der Bilder in Richtung der x- und y-Achse aufeinander. Es ist wichtig, dass jedes Bild trotz unterschiedlicher Abbildungsverhältnisse den gleichen Objektausschnitt zeigt. Zudem können die Bilder verzerrt sein. Diese Problematiken lassen sich in einem gewissen Rahmen durch eine *projektive Transformation* (Kap. 5.3) innerhalb der Software lösen.

Es sei an dieser Stelle erwähnt, dass der gewünschte Strahlteiler, der speziell angefertigt werden muss, bis zum Ende des Bearbeitungszeitraums dieser Arbeit nicht verfügbar war. Ersatzweise wurde ein Zwei-Kamera-System mit einem kubischen Strahlteiler mit einem Teilverhältnis 50:50 genutzt. Hierbei wurden lediglich die beiden defokussierten Aufnahmen detektiert. Das fokussierte Bild wurde wahlweise durch den geometrischen oder den arithmetischen Mittelwert der Defokusaufnahmen errechnet. Ein Foto des tatsächlich genutzen Aufbaus ist in Abbildung 3.4 angezeigt.

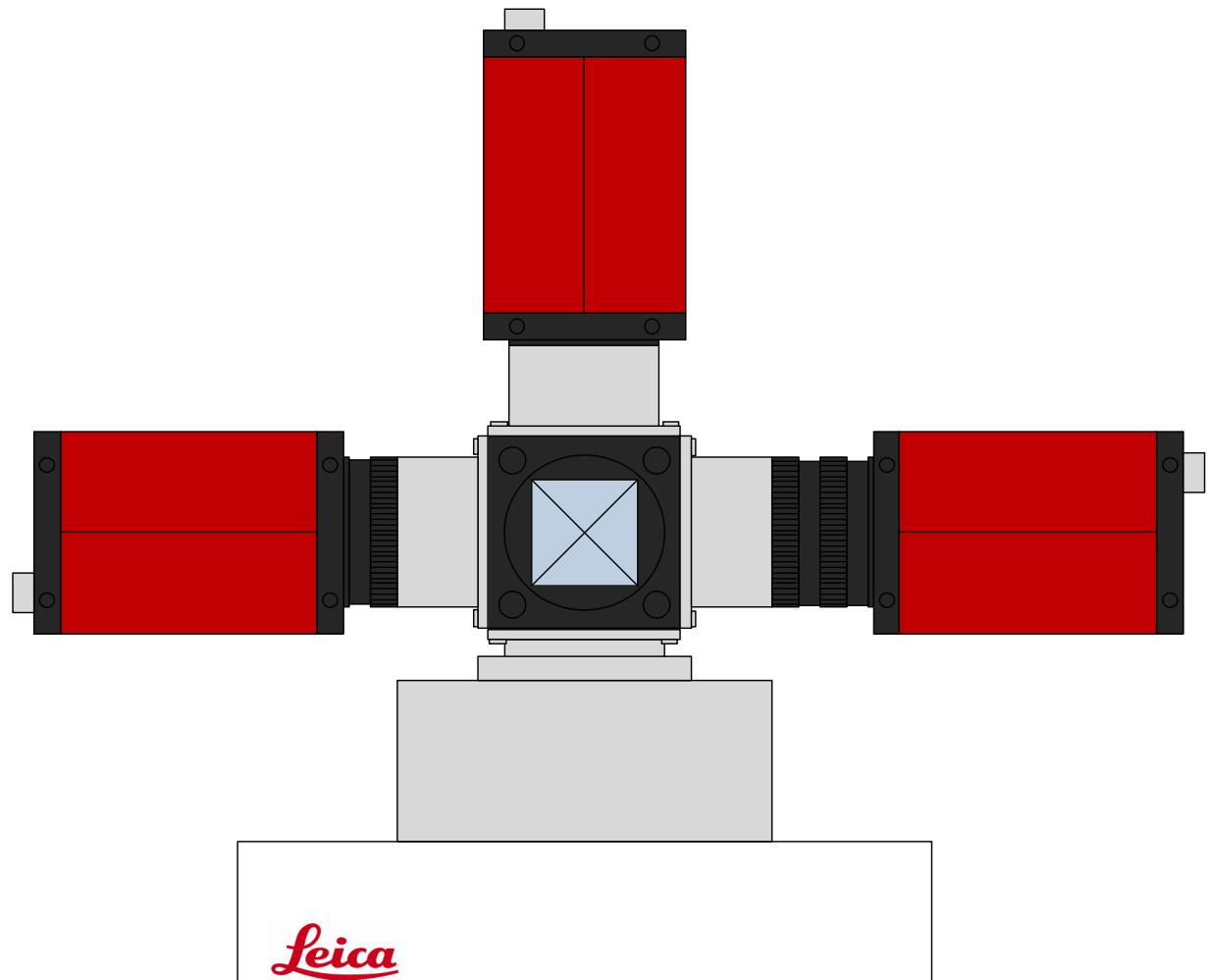


Abbildung 3.3.: Skizze des Kamera-Systems auf dem Mikroskop. Die linke Kamera steht in der Zwischenbildebene, die Kamera auf der rechten Seite nimmt das positiv defokussierte Bild auf, die obere Kamera das negativ defokussierte. In der Mitte ist der Skelettwürfel mit kubischem Strahlteiler zu sehen; Der Prismenträger ist hier nicht abgebildet.



Abbildung 3.4.: Foto des Laboraufbaus mit dem geplanten Kamera-System am Leica DMRM

4. Parallel Programming

Mikroprozessoren, die auf einer zentralen Recheneinheit (CPU - central processing unit) basieren, wie zum Beispiel die gängigen CPUs der Intel Pentium und AMD Opteron Familien, haben, wie durch das *Moore'sche Gesetz*¹ zu erwarten war, in den letzten zwei Dekaden eine exponentielle Leistungssteigerung und gleichzeitig eine Kostenreduzierung erfahren. Die Mehrheit der erstellten Programme erzielten Leistungssteigerungen durch den Einsatz neuer Prozessorgenerationen. Seit 2003 allerdings sorgen Energieverbrauch und Wärmeableitung für eine Limitierung der Taktraten² und des Grades der erreichten Aktivität pro Taktperiode innerhalb eines Rechenkerns.

Aufgrund dieser Umstände gingen scheinbar alle Hersteller von Mikroprozessoren dazu über, mehrere Recheneinheiten in einem Chip zu verbauen, um eine weitere Leistungssteigerung zu erreichen. Sogenannte *Multi-Core-Prozessoren* erobern den Markt. Diese neuen CPUs übten großen Druck auf die Softwareentwickler aus [26].

Bis zu diesem Zeitpunkt wurde Software normalerweise als sequenzielle Programme erstellt, wie es die *Von-Neumann-Architektur*³ [27] beschreibt (Abb. 4.1). Bei einem sequenziellen Programmablauf wird jeder Befehl in der vorgegebenen Reihenfolge Schritt für Schritt ausgeführt. Die Leistungssteigerung eines sequenziellen Programms durch eine neue Prozessorgeneration ist bei Mehrkern-Prozessoren nicht mehr zwingend gegeben, da das Programm nur auf einer Recheneinheit ausgeführt werden kann. Um neue Funktionen in eine bestehende Software zu implementieren bzw. neue Möglichkeiten bei der Programmentwicklung zu schaffen, musste die Computerindustrie lernen, die neuen Prozessoren erschöpfend zu nutzen.

Um den Programmablauf auf mehrere Recheneinheiten zu verteilen, müssen aus dem vorangehenden Ausgangsproblem kleinere Unterprobleme gebildet werden. Diese Unterprobleme können dann in sogenannten *Threads* realisiert werden und auf verschiedene Recheneinheiten verteilt, *parallelisiert*, werden (Abb. 4.1). Ein Thread beschreibt einen Ausführungsstrang bei der Abarbeitung eines Programms. Damit Software weiterhin durch Technologiefortschritt eine Leistungssteigerung erfährt, beginnt nun ein Trend, bestehende Software zu parallelisieren, die sogenannte *Concurrency Revolution* [28] (deut. Revolution der Nebenläufigkeit) beginnt.

Die Anwendung des *Parallel Programmings* (deut. Parallelprogrammierung) ist keines-

¹benannt nach Gordon Earle Moore, Mitbegründer des Halbleiterherstellers Intel.

Moore's Law of complexity [25] besagt, in der ursprünglichen Interpretation, dass sich die Anzahl der Schaltkreiskomponenten auf einem Computerchip innerhalb von zwei Jahren verdoppeln.

²Die maximal erreichte Taktfrequenz einer CPU im Consumer-Bereich ist das Modell *Intel Pentium 4 Extreme Edition Prescott 2M* mit 3,73GHz.

³benannt nach John von Neumann (1903-1957), US-amerikanischer Mathematiker österreichisch-ungarischer Herkunft

wegs neu. Im Bereich des HPC (engl. high-performance computing - deut. Hochleistungsrechnen), besonders in Verbindung mit *Supercomputern*⁴ oder *Computerclustern*⁵ [29], werden bereits seit Jahrzehnten rechenintensive Aufgaben parallelisiert gelöst. Nur wenige, meist wissenschaftliche Programme können den Einsatz eines solchen teuren Systems rechtfertigen und machten den Bereich des Parallel Programmings lange zur Spezialaufgabe. Die Entwicklung der neuen Mikroprozessoren ermöglicht die Anwendung parallelisierter Programmabläufe bis in den Consumer-Bereich. Eine weitere Effektivitätssteigerung im Parallel Programming wird durch den Einsatz von Grafikprozessoren (Kap. 4.1) zur Berechnung von parallelisierbaren Problemen erreicht.

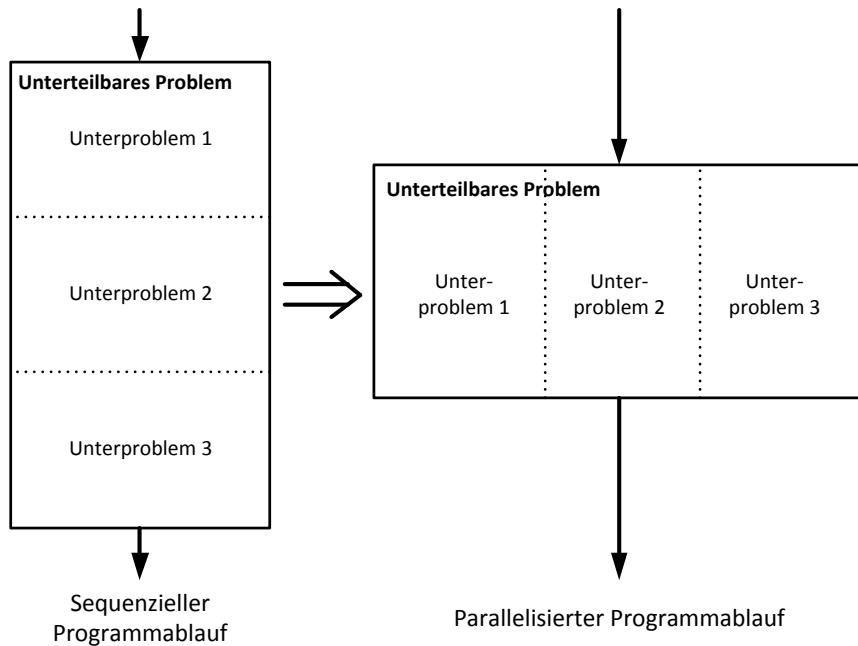


Abbildung 4.1.: Parallelisierung eines Programmablaufs

Ziel dieser Arbeit ist es, den vorgestellten Algorithmus zur Phasenrekonstruktion (Abb. 2.5) so zu realisieren, dass er einerseits auf gängiger, kostengünstiger Consumer-Hardware ausführbar ist, andererseits soll er echtzeitfähig sein. Echtzeitfähigkeit bedeutet, dass das menschliche Auge keine Verzögerung in der Bildentstehung wahrnimmt. Eine gute Einheit um dies zu messen ist die Bildfrequenz, abgekürzt mit *fps* (engl. frames per second

⁴Hochleistungsrechner, der zum Zeitpunkt seiner Einführung im obersten realisierbaren Leistungsbereich arbeitet. Ein typisches Merkmal eines modernen Supercomputers ist seine große Anzahl an Prozessoren, die auf gemeinsame Peripheriegeräte und einen teilweise gemeinsamen Hauptspeicher zugreifen können.

⁵Rechnerverbund, eine Anzahl von vernetzten Computern, die von außen in vielen Fällen als ein Computer gesehen werden können. In der Regel sind die einzelnen Elemente eines Clusters untereinander über ein schnelles Netzwerk verbunden.

- deut. Bilder pro Sekunde). Sie beschreibt, wie viele Einzelbilder pro Sekunde generiert und dargestellt werden. Das menschliche Gehirn nimmt ab einer Bildfrequenz von etwa 24 fps [30] eine scheinbar flüssig bewegte Szene wahr, ein Wert, der auch in dieser Arbeit die mindestens zu erreichende Grenze bildet. 24 fps entsprechen $41,66ms \approx 41ms$. In dieser Zeit müssen die drei Eingangsbilder synchronisiert aufgenommen, die Bildverarbeitung des Algorithmus abgeschlossen und das generierte Bild dargestellt worden sein. Um dies zu erreichen, werden die weit verbreiteten Grafikprozessoren und Schnittstellen des Herstellers NVIDIA verwendet. Eine genaue Auflistung der im Laboraufbau genutzten Hardwarekomponenten ist im Anhang A zu finden.

Eine Einführung zur Nutzung von Grafikprozessoren sowie zur praktischen Anwendung des Parallel Programmings mittels NVIDIA CUDA soll in diesem Kapitel erfolgen.

4.1. GPU - Der Grafikprozessor

2003 teilte sich die Halbleiterindustrie in zwei Lager [26]. Das Mehrkern-Lager hat sich darauf spezialisiert, CPUs mit immer mehr Rechenkernen zu entwickeln und die Leistung jedes einzelnen Rechenkerns zu optimieren. Ein modernes Beispiel hierfür sind die *Intel Core i7* CPUs mit vier bis acht Recheneinheiten und darauf optimierter Architektur. Das Vielkern-Lager hingegen hat sich darauf konzentriert, möglichst viele Berechnungen in einem Prozessor parallel auszuführen. Die ersten Generationen von Vielkern-CPUs besaßen noch eine überschaubaren Anzahl von Kernen, aber auch hier verdoppelt sich die Menge der Rechenkerne mit jeder neuen Generation.

Aktuell besteht der Trend, *GPUs* in diesem Bereich einzusetzen. Als *GPU* (graphics processing unit - deut. Grafikprozessor) wird eine Recheneinheit bezeichnet, die speziell dazu designed wurde, Berechnungen zur Erstellung und Darstellung von Computergrafiken durchzuführen. Auf die Grafikerzeugung und -wiedergabe mittels GPU wird in Kapitel 5.1 eingegangen. Viele moderne GPUs lassen sich als leistungsstarke Parallelrechner einsetzen. Die GPUs der Hersteller *NVIDIA* und *AMD/ATI* bieten diese Möglichkeit auf fast allen aktuell angebotenen Grafikkarten. Ebenfalls gibt es bereits GPU-Platinen die ausschließlich als Parallelrechner ausgelegt sind und keine Grafikausgabe mehr ermöglichen [31]. Zur Grafikausgabe wird dann eine weitere Grafikkarte benötigt.

In dieser Arbeit werden GPUs des Herstellers *NVIDIA* genutzt, die sowohl die Grafikausgabe sowie den Einsatz als Parallelrechner auf einer Grafikkarte ermöglichen. Die gängigsten Consumer-Grafikkarten des Herstellers finden sich in der Produktreihe *GeForce*. Für professionelle Anwendungen besonders in der Wissenschaft existieren die Produktreihen *QUADRO* und *TESLA*.

Als Grafikkarte im *Workstation*-System⁶ wird in dieser Arbeit ein *GeForce GTX295* Chip eingesetzt. Er besitzt 480 Rechenkerne, verteilt auf zwei GPUs. Jeder dieser Kerne ist ein stark auf Multithreading⁷ ausgelegter, einzeln ansprechbarer Prozessor und teilt

⁶Im Vergleich zum normalen Heim- oder Bürocomputer besonders leistungsfähiger Arbeitsplatzrechner.

⁷Bezeichnet das gleichzeitige Abarbeiten mehrerer Threads (Unterprobleme).

sich den Kontroll- und Anweisungscache⁸ mit sieben weiteren Kernen.

Zur Geschwindigkeitbestimmung von Prozessoren und deren Systemen wurde die Maßeinheit *FLOPS* (engl. Floating Point Operations Per Second - deut. Gleitkommaoperationen pro Sekunde) eingeführt. Ein FLOPS steht für eine Gleitkommaoperation (Addition oder Multiplikation), die das System pro Sekunde ausführen kann.

Seitdem die Entwicklung im Vielkern-Bereich 2003 an Fahrt gewann, führen GPUs das Rennen um die meisten erreichten FLOPS an [26]. In Abb 4.2 zeigt sich zudem, dass die Leistungssteigerung bei Multi-Core-CPUs langsam vorangeht, wobei GPUs weiterhin einen deutlichen Leistungsanstieg verzeichnen. Das Verhältnis der maximalen FLOPS zwischen Mehrkern-CPUs und Vielkern-GPUs liegt 2009 bei 1 zu 10. Auch im Bereich der Supercomputer wird der Einsatz von GPUs immer beliebter. So ist der momentan weltweit zweitschnellste Supercomputer (Stand Juni 2010) der Rechner *Nebulae* des *National Supercomputing Centre in Shenzhen (NSCS)* in China und verwendet 120640 Rechenkerne auf *NVIDIA TESLA C2050* GPUs [32].

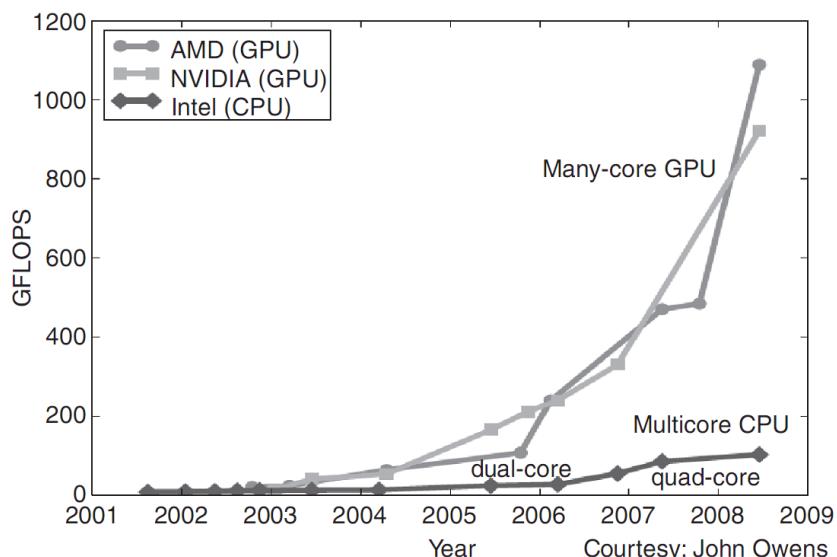


Abbildung 4.2.: Leistungssteigerung mit fortschreitender Entwicklung von GPUs und CPUs [26]

Die Leistungsunterschiede zwischen den beiden Prozessortypen lassen sich am unterschiedlichen Designansatz festmachen. Um den prinzipiellen Aufbau eines Prozessors anschaulich zu machen, lässt er sich in vier Komponenten unterteilen: Zum einen der eigentliche Rechenkern oder ALU (engl. arithmetic logic unit), hier werden die zugewiesenen Rechenoperationen ausgeführt. Der *Cache* dient als *Puffer-Speicher* und ermöglicht den wiederholten Zugriff auf bereits genutzte Inhalte. Ein weiteres Element übernimmt die Kontrolle des Datenflusses (Controlcache). Das gesamte Datenvolumen, vor allem

⁸Als Cache wird ein *Puffer-Speicher* bezeichnet, der Kopien von Inhalten anlegt, um diese schnell wieder abrufbar zu machen. Hier wird der Cache als Hardware-Element realisiert.

die Inhalte, die zur momentanen Berechnung nicht benötigt werden, werden im ausgelagerten Speicher *DRAM* (dynamic random access memory) gesammelt.

Die GPU wurde speziell für rechenintensive und stark parallelisierte Berechnungen - genau darum geht es bei Grafikberechnungen - konzipiert. Es wird die Mehrzahl an Transistoren der Datenverarbeitung zugewiesen, die Flusskontrolle sowie der Anteil an Cache-Bausteinen fällt im Vergleich zur CPU eher sparsam aus (Abb. 4.3).

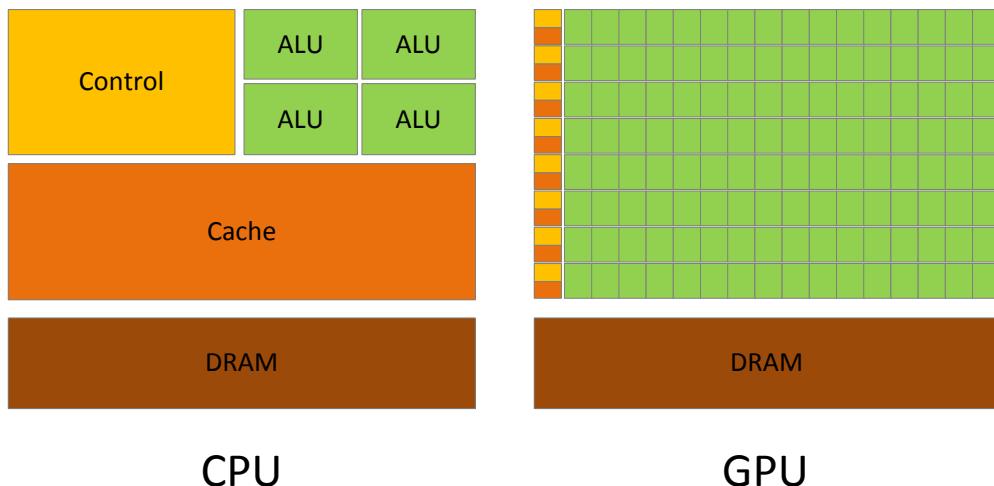


Abbildung 4.3.: Schematischer Aufbau von CPU und GPU [33]

Bei genauerer Betrachtung fällt auf, dass sich GPUs sehr gut für Probleme eignen, die sich mit der Berechnung paralleler Datensätze lösen lassen. Es werden also die gleichen aufwändigen Berechnungen auf mehreren Recheneinheiten an unterschiedlichen Daten aus einem Datensatz durchgeführt. Aufwändige Berechnungen lassen sich so verstehen, dass der Aufwand an arithmetischen Operationen größer ist als der Aufwand der Speicherbewegungen. Da die gleichen Operationen auf immer andere Datenelemente angewendet werden, ist die Anforderung an eine komplexe Flusskontrolle eher gering. Durch die großen Mengen an Daten und die hohen Rechenintensitäten lassen sich die Verzögerungszeiten, die der Zugriff auf Speicher verursacht, sogenannte Latenzen, hinter der Vielzahl an Berechnungen verstecken. So lassen sich große Puffer-Speicher vermeiden. Parallelrechnung ermöglicht es also, die gleiche komplexe Berechnung an vielen Datenelementen auf parallel ausgeführte *Threads* aufzuteilen.

Viele Anwendungen, die große Datenmengen verarbeiten, können mittels parallelisierter Berechnungen Leistungssteigerungen erzielen, und moderne GPU-Technologien ermöglichen die kosteneffiziente Realisierung. Der Einsatz von Grafikkartenprozessoren zur Berechnung von Aufgaben, die herkömmlicher Weise von CPUs übernommen werden, lassen sich unter dem Begriff *GPGPU* (engl. General-purpose computing on graphics processing units - deut. universelle Berechnungen auf Grafikprozessoren) zusammenfassen. Neben den üblichen Einsatzgebieten wie dem 3D-Rendering oder anderen Medienbehandlungen etabliert sich GPGPU auch immer mehr in anderen datenintensiven Bereichen der angewandten Informatik, so zum Beispiel bei der allgemeinen Signalverarbeitung, Finanzmathematik, Bioinformatik oder auch der Simulation und Anwendung

in vielen Bereichen der Physik.

GPUs sind als treibende Kraft für numerische Berechnungen designed worden und können nicht jede Aufgabe alleine übernehmen, die bisher eine CPU ausgeführt hat. Die meisten Anwendungen können somit nicht ausschließlich auf der GPU ausgeführt werden, sondern sie müssen mit einer CPU zusammenarbeiten. Eine solche Schnittstelle zwischen CPU und GPU ist *NVIDIA CUDA*.

4.2. NVIDIA CUDA

Die von NVIDIA 2006 [33] erstmals vorgestellte *Compute Unified Device Architecture*, kurz *CUDA*, ist eine parallele Rechnerarchitektur, die es ermöglicht, viele komplexe Probleme auf einem NVIDIA Grafikprozessor effektiver zu lösen als auf einem herkömmlichen Prozessor. Hierzu wurde ein neuer Programmierstil⁹ und neuartiger Befehlssatz¹⁰ eingeführt, die dem geübten Programmierer den unbeschwerteren Zugang zum Parallel Programming ermöglichen soll.

Die Softwareumgebung von CUDA ermöglicht es dem Programmierer durch wenige zusätzliche Syntaxelemente die höhere Programmiersprache¹¹ *C* zum Parallel Programming zu nutzen. Wie in Abbildung 4.4 dargestellt, erlaubt CUDA auch die Verwendung anderer Programmiersprachen bzw. *Programmierschnittstellen*, kurz *APIs* (engl. application programming interface - deut. Schnittstelle zur Anwendungsprogrammierung), wie z.B: OpenCL, DirectCompute und CUDA Fortran. Ein besonderes Blick bei der

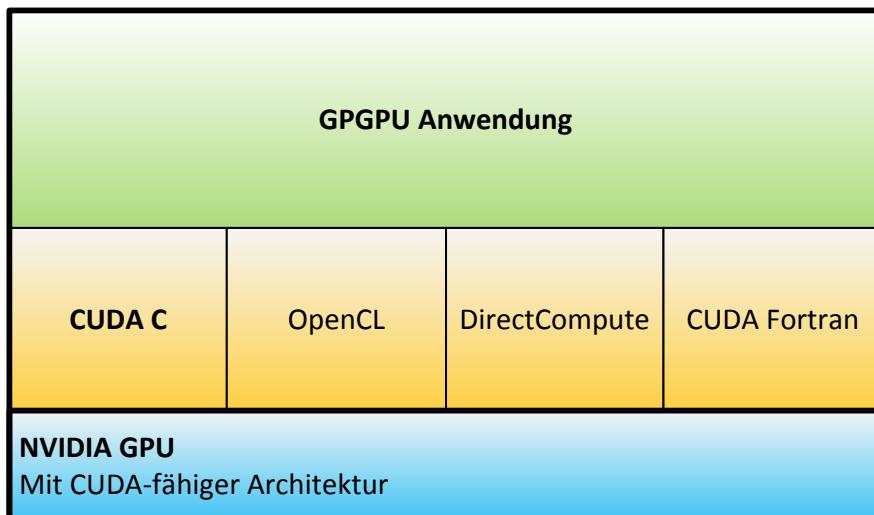


Abbildung 4.4.: CUDA stellt mehrere Schnittstellen zur Unterstützung von Programmiersprachen und APIs bereit

Entwicklung paralleler Anwendungen sollte darauf fallen, den Grad der Parallelisierung

⁹Satz von Regeln, die dem Programmierer vorgegeben werden

¹⁰Menge der Maschinenbefehle eines Mikroprozessors

¹¹Programmiersprache, die das akstrakte Abfassen eines Computerprogramms ermöglicht. Für den Menschen ist diese Sprache verständlich, zur Verarbeitung muss sie erst in einen maschinenlesbaren Code umgewandelt werden.

der Zahl verfügbarer Rechenkerne anzupassen, gerade unter Berücksichtigung des Moorschen Gesetzes ein wichtiger Anhaltspunkt. Der von CUDA vorgegebene Programmierstil wurde speziell entwickelt, um dieser Herausforderung gerecht zu werden, ohne den Programmierer damit zu überfordern.

Grundlegend bietet CUDA drei Abstraktionsstufen, die dem Programmierer eine minimale Anzahl von Spracherweiterungen bieten: Hierarchien von *Thread*-Gruppen, verschiedene *Speicherebenen* und *Synchronisationsgrenzen*.

Diese Abstraktionen ermöglichen den feinaufgelösten Umgang mit parallelen Datenmengen, verschachtelt in größeren Anweisungen zur Lösung des Problems. So wird der Programmierer angeleitet seine Problemstellungen in kleine Unter-Problemstellungen aufzuteilen, die unabhängig parallel in *Blocks* gelöst werden können. Als Block wird eine Menge von zusammenhängenden, gruppierten Threads betitelt. Diese zerlegte Darstellung ermöglicht Threads die Zusammenarbeit bei der Lösung eines Unter-Problems und gleichzeitig ist eine automatisierte Anpassung auf den Aufbau der genutzten GPU möglich. Somit kann jeder Block jedem verfügbaren Rechenkern zugewiesen werden in jeder Reihenfolge, gleichzeitig oder sequenziell. Ein kompiliertes CUDA-Programm kann somit auf GPUs mit einer beliebigen Anzahl von Rechnernkernen ausgeführt werden (Abb. 4.5). Das ausführende System muss lediglich die Anzahl vorhandener Rechenkerne kennen.

Dieses Programmiermodell erlaubt es CUDA zu einer großen Auswahl von Hardware kompatibel zu sein, von den Hochleistungs-GPUs GeForce für den Einsatz mit Computerspielen über die professionellen Lösungen Quadro, Tesla und Fermi bis zu einer großen Anzahl preisgünstiger, weitverbreiteter GeForce GPUs wie sie bereits in handelsüblichen Bülorechnern eingesetzt werden. Eine Liste der CUDA-fähigen Produkte ist im auf der Homepage des Herstellers, unter http://www.nvidia.com/object/cuda_gpus.html, zu finden.

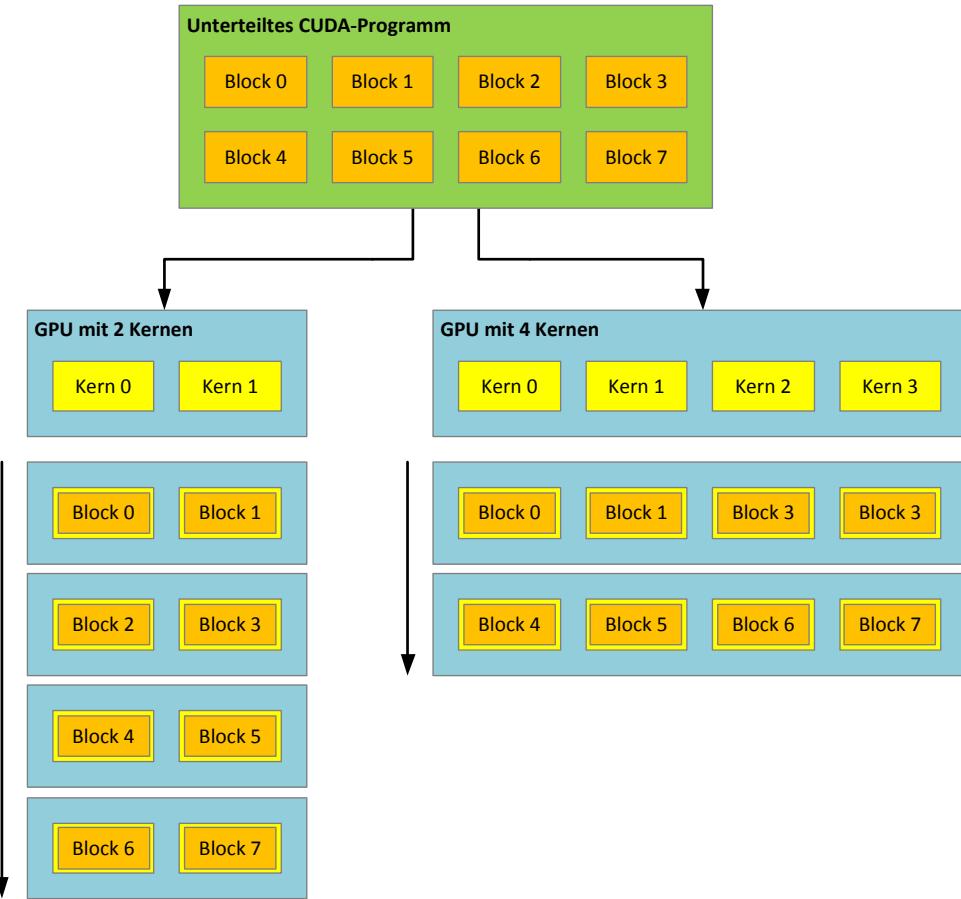


Abbildung 4.5.: Automatisierte Skalierbarkeit von kompilierten CUDA-Programmen auf die Anzahl der verfügbaren Rechenkerne

4.2.1. Einführung in das Programmiermodell

In dieser Arbeit wird nur auf die *CUDA C API* eingegangen. Sie ermöglicht einen einfachen Einstieg, vorausgesetzt der Umgang mit der Programmiersprache *C* bzw. *C/C++* ist bereits vertraut. CUDA C bietet eine minimale Anzahl an Erweiterungen zur Syntax der Sprache C und eine notwendige Laufzeitbibliothek (*cudart*). Die Funktionen der API haben den Prefix *cuda*.

Alternativ hierzu ist die *CUDA Driver API* verfügbar. Sie ermöglicht eine exakte Kontrolle aller Vorgänge, ist aber auch entsprechend komplexer im Umgang und nicht zum Einstieg geeignet. Die hier implementierte dynamische Bibliothek heisst *nvcuda*, die Funktionen haben den Prefix *cu*. Nähere Informationen zur Driver API und auch zu CUDA C sind in [33] zu finden.

4.2.1.1. Kernels

CUDA C erweitert C um die Möglichkeit, eine Funktion zu definieren, die M mal von N verschiedenen CUDA-*Threads* ausgeführt wird. Diese Funktionen werden *Kernel* ge-

nannt und durch die Deklaration `__device__` definiert. Die Anzahl an Threads, die diesen Kernel ausführen sollen und wie sie in Blocks organisiert sind, wird durch eine neue Konfigurationssyntax zum Kernelaufruf (Quellcode 4.9) spezifiziert:

```
1 Funktion <<< ... >>> () ;
```

Jeder Thread bekommt eine eigene *ID* zugewiesen, die innerhalb des Kernels über die vorhandene Variable `threadIdx` verfügbar ist. Ebenso ist die ID des Blocks, in dem der jeweilige Thread gruppiert ist, über `blockIdx` abrufbar.

Zur Veranschaulichung soll im Folgenden Quellcode der Kernel zur einfachen Vektoraddition dienen.

```
1 // Definition des Kernels
2 __global__ void VecAdd(float* A, float* B, float* C)
3 {
4     int i = threadIdx.x;
5     C[i] = A[i] + B[i];
6 }
7
8 int main()
9 {
10     ...
11     // Aufruf des Kernels mit N Threads
12     VecAdd<<<1, N>>>(A,B,C);
13     ...
}
```

Quellcode 4.1: Addition der Vektoren A und B der Größe N

4.2.1.2. Hierarchie der Threads

Die Variable `threadIdx` ist ein Vektor aus drei Komponenten. So kann ein *Thread* als eindimensional, zweidimensional oder dreidimensional identifiziert werden und mit weiteren Threads einen ein- bis dreidimensionalen *Block* bilden. Dies ermöglicht, Berechnungen in den Definitionsbereichen von Vektoren, Matrizen oder Volumen anschaulich darzustellen.

Es existiert ein direkter Zusammenhang zwischen dem Index des aktuell ausgeführten Threads und der zugehörigen ID des Blocks, dem er zugewiesen ist: Für einen eindimensionalen Block ist der Index gleich der ID; Bei einem zweidimensionalen Block der Größe (N_x, N_y) ergibt sich die ID des Threads mit $(x + yN_x)$, wobei (x, y) die aktuellen Indizes sind; Ein dreidimensionaler Block (N_x, N_y, N_z) hat an der Position (x, y, z) die ID $(x + yN_x + zN_xN_y)$.

Das Beispiel der Addition quadratischer Matrizen soll den Zusammenhang anschaulich machen.

```

1 // Definition des Kernels
2 __global__ void MatAdd(float A[N][N],float B[N][N],float C[N][N])
{
3     int i = threadIdx.x;
4     int j = threadIdx.y;
5     C[i][j] = A[i][j] + B[i][j];
6 }
7
8 int main()
9 {
10     ...
11     // Aufruf des Kernels mit N * N * 1 Threads
12     int numBlocks = 1;
13     dim3 threadsPerBlock(N,N);
14     MatADD<<<numBlocks, threadsPerBlock>>>(A,B,C);
15     ...
16 }

```

Quellcode 4.2: Addition der quadratischen Matrizen A und B der Größe $N \times N$

Die Anzahl an Threads pro Block ist limitiert, da alle Threads eines Blocks auf dem gleichen Rechenkern erwartet werden und sich dort den limitierten Speicher des Kerns teilen müssen. Die aktuelle CUDA-Architektur ermöglicht die Ausführung von bis zu 1024 Threads pro Block. Ein Kernel kann aber in mehreren gleichgroßen Blocks und deren Threads ausgeführt werden. Somit ist die Anzahl der ausgeführten Threads gleich dem Produkt der Threads pro Block und der Anzahl der Blocks.

Blocks werden in einem ein- oder zweidimensionalen Raster, *Grid* genannt, verschachtelt (Abb. 4.6). Die Anzahl an Blocks pro Grid wird für gewöhnlich durch die Größe der zu berechnenden Daten oder der Anzahl an verfügbaren Prozessoren vorgegeben.

Die Anzahl an Threads pro Block und die Anzahl an Blocks pro Grid werden in der Syntax des Kernelaufrufs definiert und können vom Typ `int` oder `dim3` (dreidimensionale Integer-Variable) sein. Jeder Block in einem Grid kann durch ein- oder zweidimensionale Indizes identifiziert werden. Innerhalb des Kernels ist dieser Index über `blockIdx` und die Dimension des Blocks über `blockDim` abfragbar.

Das vorherige Beispiel `MatAdd()` (Code 4.2) soll nun so erweitert werden, dass die Berechnung in mehreren Blocks erfolgen kann.

```

1 // Definition des Kernels
2 __global__ void MatAdd(float A[N][N],float B[N][N],float C[N][N])
3 {
4     int i = blockIdx.x * blockDim.x + threadIdx.x;
5     int j = blockIdx.y * blockDim.y + threadIdx.y;
6     if(i < N && j < N)
7         C[i][j] = A[i][j] + B[i][j];
8 }
9
10 // Code auf CPU
11 int main()
12 {
13     ...
14     // Aufruf des Kernels
15     dim3 threadsPerBlock(16,16);
16     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
17     MatADD<<<numBlocks, threadsPerBlock>>>(A,B,C);
18     ...
19 }

```

Quellcode 4.3: Addition der quadratischen Matrizen A und B der Größe $N \times N$ in mehreren Blocks

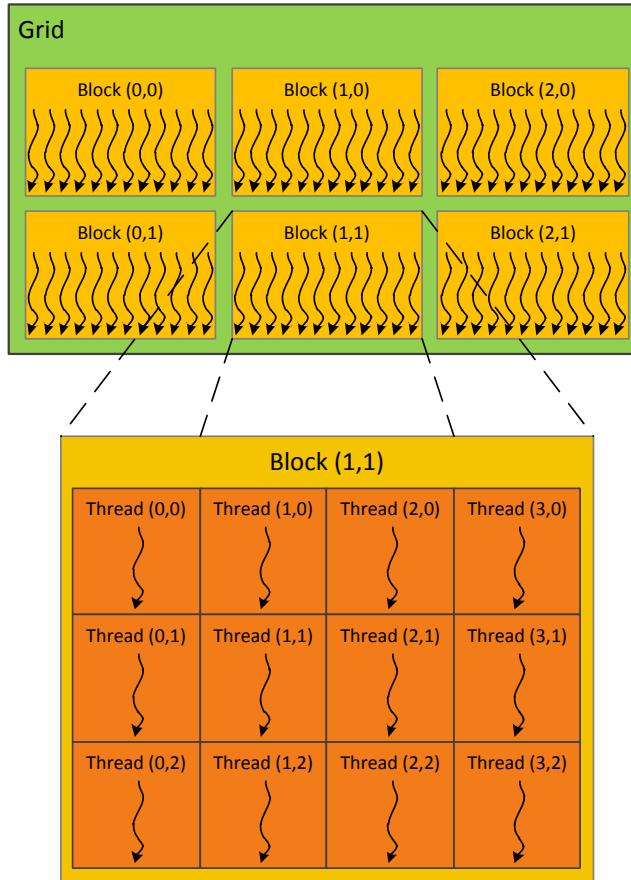


Abbildung 4.6.: Aufbau eines Grid aus mehreren Thread-Blocks

Die beliebige Größe des Blocks von 16×16 ($16 \cdot 16 = 256$ Threads) ist eine übliche Wahl. Wie zuvor wird das Grid hier mit einer ausreichenden Anzahl an Blocks erstellt, so dass jedes Element der Matrix C in einem separaten Thread berechnet werden kann.

Jeder Block muss unabhängig, also in jeder beliebigen Reihenfolge, egal ob parallel oder sequenziell ausführbar sein. Dies ermöglicht das wunschgemäße Zuweisen der Blocks in einer beliebigen Reihenfolge auf verschiedenen Rechenkernen (Abb. 4.5). Mit diesem Konzept ist das Erstellen skalierbarer Software kein Problem mehr.

Die Threads innerhalb eines Blocks können über einen gemeinsamen Speicher, dem sogenannten *Shared Memory* (Kap. 4.2.3.1), zusammenarbeiten. Die Speicherzugriffe lassen sich über gezielte Synchronisationen kontrollieren. Der Funktionsaufruf `__syncthreads()` erzeugt einen Synchronisationspunkt für alle Threads des aktiven Kernels. An diesem Punkt müssen alle Threads eines Blocks abgearbeitet sein, bevor nachfolgende Anweisungen ausgeführt werden können.

4.2.1.3. Hierarchie des Speichers

Threads können während ihrer Ausführung auf Daten aus verschiedenen Speicherbereichen zugreifen (Abb. 4.7). Jeder Thread besitzt einen lokalen Speicher, den *Local*

Memory, auf den kein anderer Thread zugreifen kann und der hauptsächlich zur temporären Ablage von übergebenen oder zwischengespeicherten Werten dient. Blocks besitzen einen *Shared Memory*, auf den alle Threads des Blocks uneingeschränkt zugreifen können und der so lange existiert wie der Block selbst. Alle Threads haben Zugriff auf den globalen Speicher, *Global Memory*, der viel Platz bietet, aber langsam ist. Zusätzlich ist der private und unveränderbare *Constant Memory* und der spezielle Texturen-Speicher für Threads lesbar. Der globale, konstante und der Texturspeicher sind beständig für jeden Kernel-Aufruf eines Programms. Auf die verschiedenen Speicherbereiche wird in Kapitel 4.2.3.1 näher eingegangen.

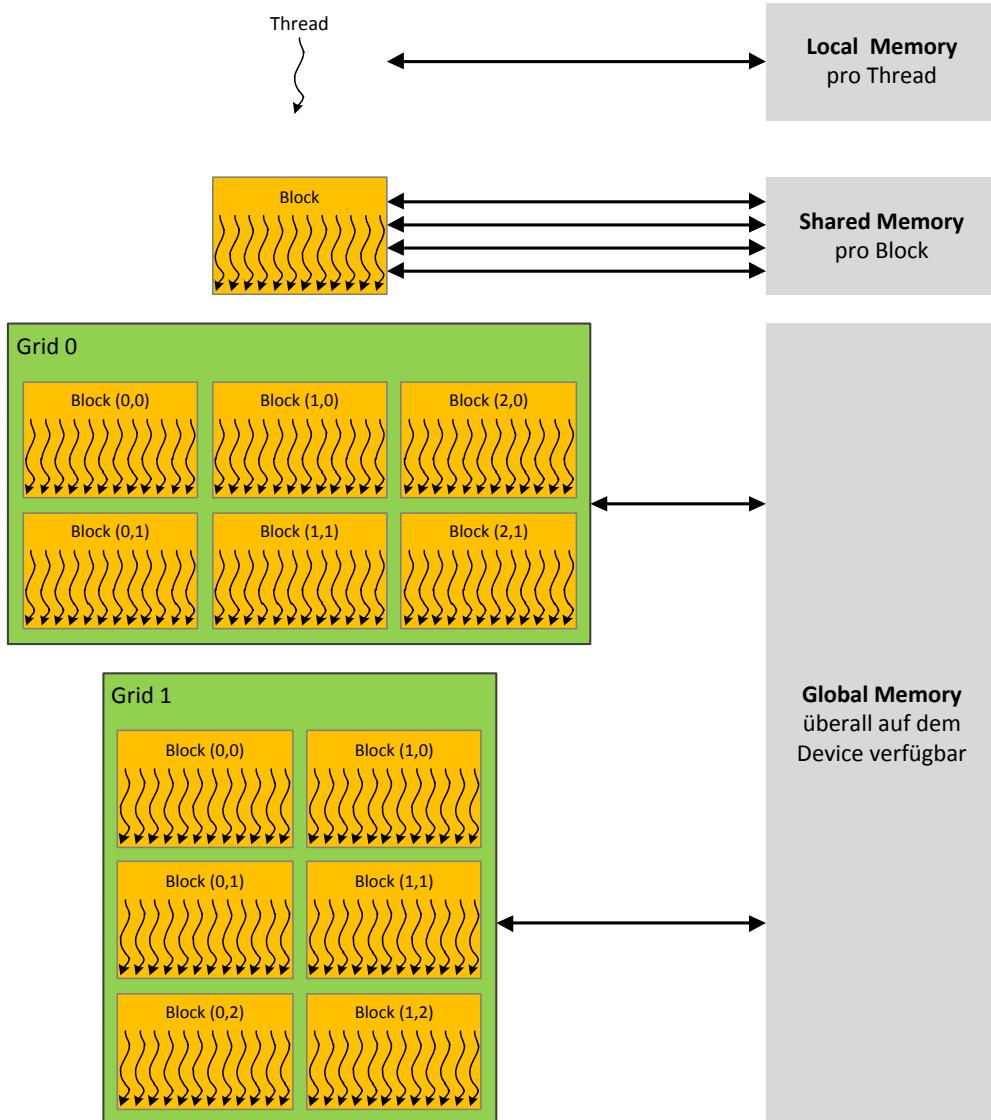


Abbildung 4.7.: Speicherhierarchie in CUDA

4.2.1.4. Heterogenität der Programmierung

Das Programmiermodell von CUDA geht davon aus, dass alle Threads auf einem fiktiven Hardwareelement, dem *Device*, ausgeführt werden und sieht diese als Erweiterung zum Hauptprozessor, dem *Host*, an. Auf dem Host wird das eigentliche C-Programm ausgeführt. Somit ergibt sich ein heterogenes Modell, das für beide Komponenten, Host und Device, gilt (Abb. 4.8). So wird im vorangegangenen Beispiel (Code 4.3) der Kernel `MatAdd()` auf dem Device und das restliche Programm vom Host ausgeführt.

Ebenfalls wird davon ausgegangen, dass Host wie Device eigene Speicherbereiche im DRAM besitzen, entsprechend bezeichnet als *Host Memory* und *Device Memory*. So mit muss das Programm die Verwaltung der globalen, konstanten, gemeinsamen und Texturenspeicherbereiche bereitstellen, der für den Kernel verwendbar sein soll, über die *CUDA Runtime* abstrahiert durch CUDA C Funktionen. Dazu gehören Belegung und Freigabe von Speicherbereichen sowie der Transport von Daten zwischen den Speicherbereichen von Host und Device.

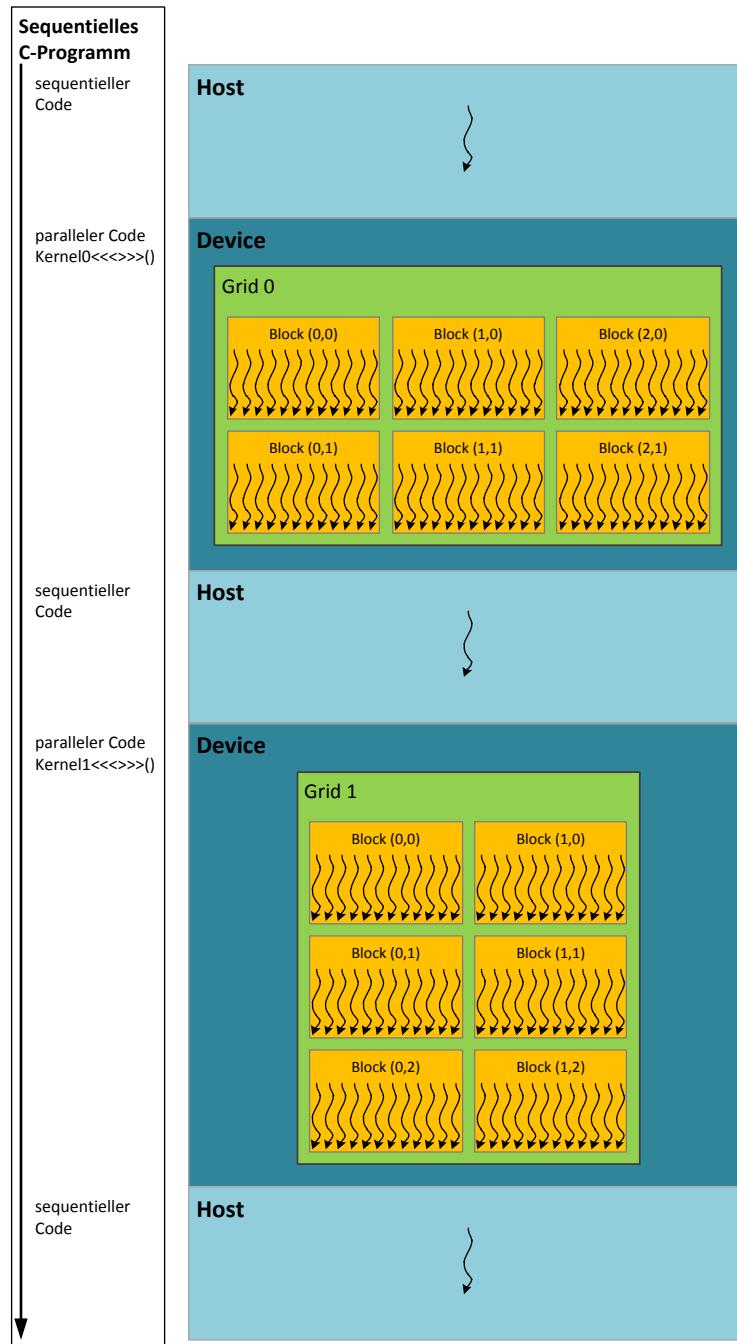


Abbildung 4.8.: Heterogenität des Programmiermodells

4.2.1.5. Compute Capability

Die *Compute Capability* eines Device, also die Leistungsfähigkeit und das Entwicklungsstadium einer GPU, wird durch eine *Major Revision Number* und eine *Minor Revision Number* angegeben. Die Major Revision Number beschreibt die zugrundeliegende Architektur der GPU. So haben Devices, die auf der Fermi Architektur basieren, die Nummer 2, ältere die Compute Capability 1.x.

Die Minor Revision Number steht für die fortschreitenden Verbesserungen in der Architektur der Kerne, gegebenenfalls werden hierbei neue Features eingeführt. Die Compute Capability versteht sich als abwärtskompatibel.

4.2.2. Das erste CUDA Programm

Bei den bisher gezeigten Code-Beispielen wurde in der Hauptfunktion des Host `main()` lediglich der Aufruf des Kernels angegeben. Jetzt soll ein vollständig lauffähiges Beispiel erstellt werden. Es basiert auf dem Beispielcode 4.3, der Kernel bleibt unverändert zu dem im vorherigen Quellcode.

```

1 // Device Code
2 __global__ void MatAdd(float* A, float* B, float* C, int N)
3 {
4     int i = blockIdx.x * blockDim.x + threadIdx.x;
5     int j = blockIdx.y * blockDim.y + threadIdx.y;
6     if(i < N && j < N)
7         C[i+j*N] = A[i+j*N] + B[i+j*N];
8 }
9
10 // Host Code
11 int main()
12 {
13     int N = ...;
14     size_t size = N * N * sizeof(float);
15
16     // Belegen des benötigten Speichers für die Matrixen h_A, h_B und h_C im Host
17     // Memory mit der Größe (N*N)
18     float* h_A = (float*)malloc(size);
19     float* h_B = (float*)malloc(size);
20     float* h_C = (float*)malloc(size);
21
22     // Initialisieren der Eingangsmatrizen, hier werden Zufallswerte generiert
23     for(int i = 0; i < N*N; i++)
24     {
25         h_A[i] = rand();
26         h_B[i] = rand();
27     }
28
29     // Belegen des benötigten Speichers im Device Memory
30     float* d_A;
31     cudaMalloc(&d_A, size);
32     float* d_B;
33     cudaMalloc(&d_B, size);
34     float* d_C;
35     cudaMalloc(&d_C, size);
36
37     // Kopieren der Eingangsmatrizen h_A und h_B aus dem Host Memory in den Device
38     // Memory d_A und d_B
39     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
40     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
41
42     // Aufruf des Kernels
43     dim3 threadsPerBlock(16,16); // 16 x 16 = 256
44     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
45     MatAdd<<< numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, N);
46
47     // Kopieren des Ergebnisses aus dem Device Memory d_C in den Host Memory h_C
48     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
49
50     // Freigeben des belegten Device Memory
51     cudaFree(d_A);

```

```

51     cudaFree(d_B);
52     cudaFree(d_C);

53     // Freigeben des belegten Host Memory
54     free(h_A);
55     free(h_B);
56     free(h_C);
57 }
```

Quellcode 4.4: Das erste lauffähige CUDA Programm

Wie bereits in Abbildung 4.7 beschrieben, unterscheidet CUDA zwischen *Host Memory* und *Device Memory*. Kernel können nur auf Speicherbereiche im Device Memory zugreifen. Lediglich die Werte von Variablen (nicht die Werte, auf die ein Pointer zeigt), die an den Kernel übergeben werden, müssen nicht gesondert in den Device Memory kopiert werden, sondern werden für jeden Thread automatisch zur Verfügung gestellt. Im Beispiel betrifft dies die Variable \mathbf{N} .

Device Memory kann als ordinärer *linearer Speicher* oder als *CUDA Array* angelegt werden.

CUDA Arrays haben einen verschachtelten, undurchsichtigen Aufbau und sind optimiert zum Abruf von Texturen. Weitere Informationen hierzu finden sich in [33] und [34]. Ein linearer Speicher auf dem Device wird typischerweise mit `cudaMalloc()` reserviert und mit `cudaFree()` wieder freigegeben. Kopiervorgänge zwischen den Speichern von Host und Device werden mit der Funktion `cudaMemcpy()` realisiert. Weitere Möglichkeiten im Umgang mit Speicher wird im *CUDA Reference Manual* [35] beschrieben.

```

1  cudaError_t cudaMalloc (void** devPtr, size_t size);
// **devPtr := Pointer auf den Pointer der Variable
3  // size := Größe des Datenbereichs

5  cudaError_t cudaFree (void* devPtr);
// *devPtr := Pointer auf Variable
7

9  cudaError_t cudaMemcpy (void* dst, const void * src, size_t size, enum cudaMemcpyKind
// kind := Kopierrichtung: cudaMemcpyHostToHost, cudaMemcpyHostToDevice,
11 // cudaMemcpyDeviceToHost oder cudaMemcpyDeviceToDevice
```

Quellcode 4.5: Host-Funktionen zum Management von Global Memory

Zum Ausführen des Programms muss der Quellcode wie üblich kompiliert werden. Allerdings muss bei einem CUDA-Programm der in C/C++ programmierte *Host-Code* getrennt vom *Device-Code* der GPU betrachtet werden. CUDA stellt hier den eigenen Compiler-Treiber `nvcc` [36] bereit. Er ist dafür zuständig, den in einer CUDA-Quellcode-Datei (.cu oder .cu.h) gemischten Code aufzutrennen. Der gewöhnliche C/C++-Code wird dann mit einem externen Compiler in eine Objektdatei übersetzt, `nvcc` übernimmt diese Arbeit für den Teil des Codes, der später auf der GPU ausgeführt wird. Während des *Linken* der Objektdateien werden spezielle CUDA-Laufzeitbibliotheken eingebunden. Nach dem Ausführen des *Makers* steht ein eigenständig lauffähiges CUDA-Programm, beispielsweise in Form einer .exe-Datei, bereit.

4.2.3. Fortgeschrittenes Programmieren mit CUDA

Die grundlegende Funktionsweise von CUDA wurde in den vorangegangenen Abschnitten erläutert. Um ein leistungsfähiges parallelisiertes Programm zu erstellen, bietet CUDA noch einige weitere Werkzeuge. Die im Rahmen dieser Arbeit genutzten Module und Funktionsweisen sollen hier grundlegend erläutert werden. Weitere Informationen zum optimalen Arbeiten mit CUDA bieten die Quellen [33], [34], [35] und [37] des Herstellers NVIDIA.

4.2.3.1. Shared Memory

In den vorangegangenen Beispielen wurden alle Daten, die in einem Kernel verarbeitet wurden, aus dem globalen Speicher (Global Memory) bezogen, vom Thread verarbeitet und dort auch wieder abgespeichert. Allerdings ist der Transfer vom globalen Speicher in den jeweiligen Thread und wieder zurück sehr zeitaufwändig. Zudem erhält jeder Thread seine eigene Kopie aller Daten, was viel Speicherplatz unnötig verschwendet. Der Registerspeicher und der *Shared Memory* (deut. gemeinsam genutzter Speicher) be-

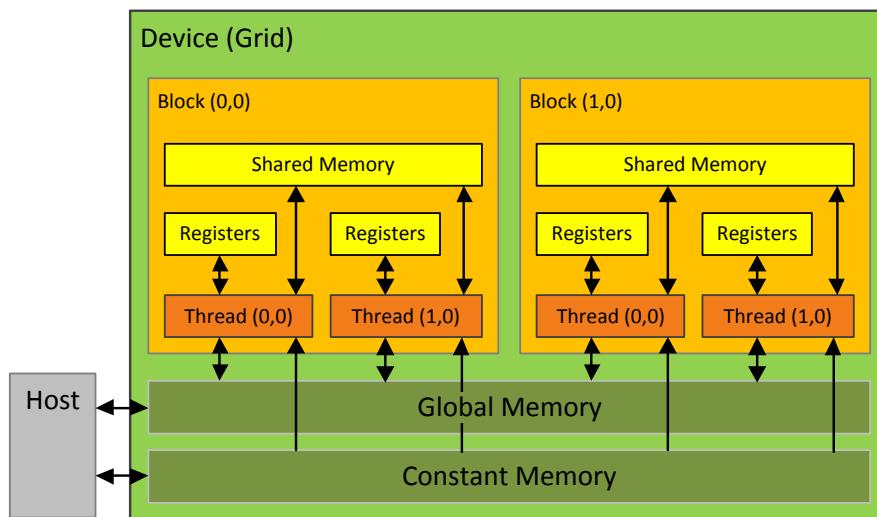


Abbildung 4.9.: Speichermodell auf CUDA GPUs

finden sich direkt auf dem Chip und sind somit sehr schnell zugänglich. Der bisher nicht erwähnte Registerspeicher ist jeweils einem Thread zugeordnet und auch nur von diesem erreichbar. Üblicherweise wird dieser Speicher vom Kernel benutzt, um oft genutzte Variablen eines Threads zu speichern. Shared Memory ist je einem Block zugewiesen und die Threads innerhalb eines Blocks können auf diesen uneingeschränkt zugreifen. Er eignet sich hervorragend zur Zusammenarbeit der Threads in einem Block. Welche Daten in welchem Speicherbereich abgelegt werden sollen, geht aus der Kennzeichnung der Variablendeklaration hervor (Tab. 4.1).

Wird vor der Variable `__shared__` angegeben, werden die Daten der entsprechenden Variable im Shared Memory abgelegt. Typischerweise werden solche Variablen im Kernel deklariert. Der Geltungsbereich beläuft sich dann auf einen Block, d.h. alle Threads dieses Blocks können auf den Speicherbereich der Variable zugreifen. Die Daten bestehen

Variablen-deklaration	Speicherbereich	Geltungsbereich	Lebensdauer
Automatische Variablen	Registers	Thread	Kernel
Automatische Array-Variablen	Local Memory	Thread	Kernel
<code>__shared__ int sharedVar;</code>	Shared Memory	Block	Kernel
<code>__device__ int globalVar;</code>	Global Memory	Grid	Anwendung
<code>__constant__ int constVar;</code>	Constant Memory	Grid	Anwendung

Tabelle 4.1.: Kennzeichnung der Variablen-deklaration

während des Kernelaufrufs, und für jeden Block wird eine eigene Kopie angelegt. Der gravierende Vorteil im Vergleich zum Global Memory ist die extrem geringe Zugriffszeit und der schnelle Datentransfer zum Thread.

Variablen im *Constant Memory*, gekennzeichnet mit `__constant__`, müssen außerhalb von Funktionsaufrufen deklariert werden. Die Elemente jedes Grids können auf die jeweils persönliche Kopie dieser Variable zugreifen und der Wert der Variable bleibt über die gesamte Programmlaufzeit erhalten. Diese Variablen werden oft zur Übergabe von Daten an Kernelfunktionen genutzt. Sie werden im globalen Speicher abgelegt, aber beim Kernelaufruf wird je eine private Kopie im Cache von jedem Thread zur Verfügung gestellt. Momentan stehen für die Variablen im Constant Memory lediglich 65536 Bytes = 64 KB Speicherplatz zur Verfügung.

Variablen, die lediglich mit `__global__` gekennzeichnet sind, werden im *Global Memory* abgelegt. Dieser ist zwar langsam, besitzt dafür aber viel Speicherplatz und alle Threads können jederzeit darauf zugreifen. Somit können globale Variablen zur Zusammenarbeit mehrerer Blocks genutzt werden. Momentan ist es jedoch nicht möglich, den Zugriff auf den Global Memory von verschiedenen, gleichzeitig ausgeführten Blocks zu beeinflussen. Der Umgang mit globalem Speicher wurde in Abschnitt 4.2.2 dargestellt.

Im nachfolgenden Beispiel (Quellcode 4.7) soll ein Kernel zur Matrixmultiplikation durch den Einsatz von Shared Memory optimiert werden. Der Host-Code entspricht dem aus Quellcode 4.4.

Es ist nun bekannt, dass der Global Memory groß, aber langsam ist und der Shared Memory klein, aber sehr schnell. Daraus ergibt sich eine gängige Strategie im Parallel Programming: Die Datenmenge wird in kleinere Submengen, sogenannten *Tiles*, unterteilt. Die Voraussetzung dabei ist, dass die Berechnungen mit Tiles innerhalb eines Kernels unabhängig von anderen Tiles erfolgen können.

Die nachfolgende Kernel-Funktion (Quellcode 4.6) zeigt die Matrixmultiplikation $A \cdot B = C$, die in mehreren Blocks ausgeführt wird.

```

// Device Code
2 __global__ void MatAdd(float* A, float* B, float* C, int N)
{
4     // Berechnen des Reihenindex für Elemente von A und C
5     int row = blockIdx.y * TILE_SIZE + threadIdx.y;
6     // Berechnen des Zeilenindex für Elemente von B und C
7     int col = blockIdx.x * TILE_SIZE + threadIdx.x;
8
9     float V = 0;
10
11     // Jeder Thread berechnet ein Element der Submatrix eines Blocks
12     for(int k = 0; k < TILE_SIZE; k++)
13     {
14         V += A[row * N + k] * B[k * N + col];
15     }
16
17     // Übertragen der Werte aus den Submatrizen in die Ergebnismatrix C
18     C[row * N + col] = V;
}

```

Quellcode 4.6: Kernel zur Matrixmultiplikation in Blocks

Es wird davon ausgegangen, dass ein Block aus 2×2 Threads besteht. Abbildung 4.10 stellt die Berechnungen der vier Threads in Block (0,0) dar. Die rechte Seite der Abbildung zeigt den Speicherzugriff der Threads auf den Global Memory. Dabei sind die Threads horizontal aufgelistet und die Speicherzugriffe in zeitlichem Ablauf in vertikaler Richtung. Jeder Thread greift bei der Ausführung auf je vier Elemente von A und B im globalen Speicher zu. In den vier Threads zeigen sich mehrere Überlappungen im

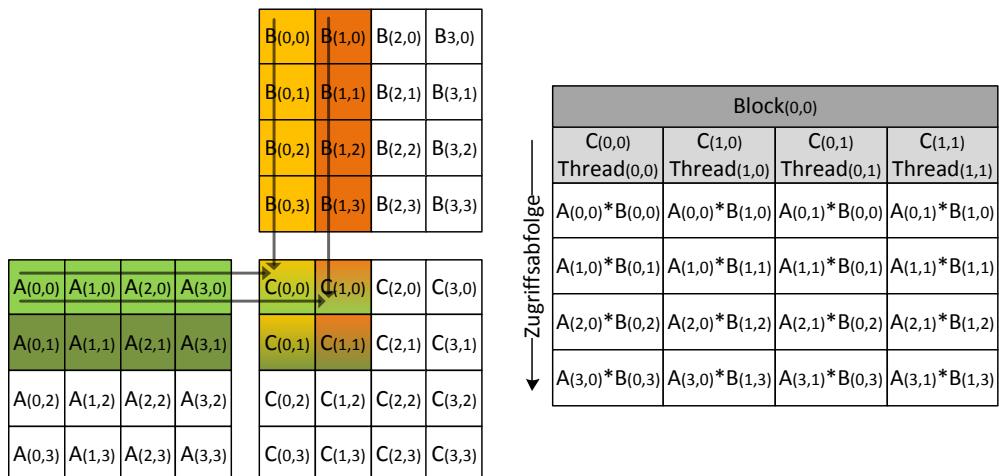


Abbildung 4.10.: Matrixmultiplikation in verschiedenen Blocks mit Global Memory

Speicherzugriff. So greifen beispielsweise $\text{Thread}(0,0)$ und $\text{Thread}(1,0)$ beide auf $A(1,0)$ zu. Der Kernel in Quellcode 4.6 lässt $\text{Thread}(0,0)$ und $\text{Thread}(1,0)$ aus dem globalen Speicher auf die Elemente der Reihe 0 von A zugreifen. Gelingt es nun, dass gemeinsam genutzte Werte in den vier Threads eines Blocks nur einmal aus dem Global Memory geladen werden, kann die Zahl an Speicherzugriffen halbiert werden, da jeder Wert zweimal aufgerufen wird. Das Optimierungspotential wächst hierbei mit der Anzahl der genutzten Blocks, so dass sich die Speicherzugriffe bei $N \times N$ Blocks um den Faktor N verringern.

Die grundlegende Idee besteht darin, die in einem Block von Threads relevanten Daten vorher einmal in den gemeinsamen Shared Memory zu kopieren, bevor die individuell benötigten Werte von den einzelnen Threads genutzt werden. Es muss beachtet werden, dass der Shared Memory recht klein ist und seine Größe beim Einladen von A und B nicht überschritten werden darf. Um dies zu verhindern, teilt man die Matrizen in kleinere Tiles, die in den Speicher passen. In Abbildung 4.11 werden die Matrizen in

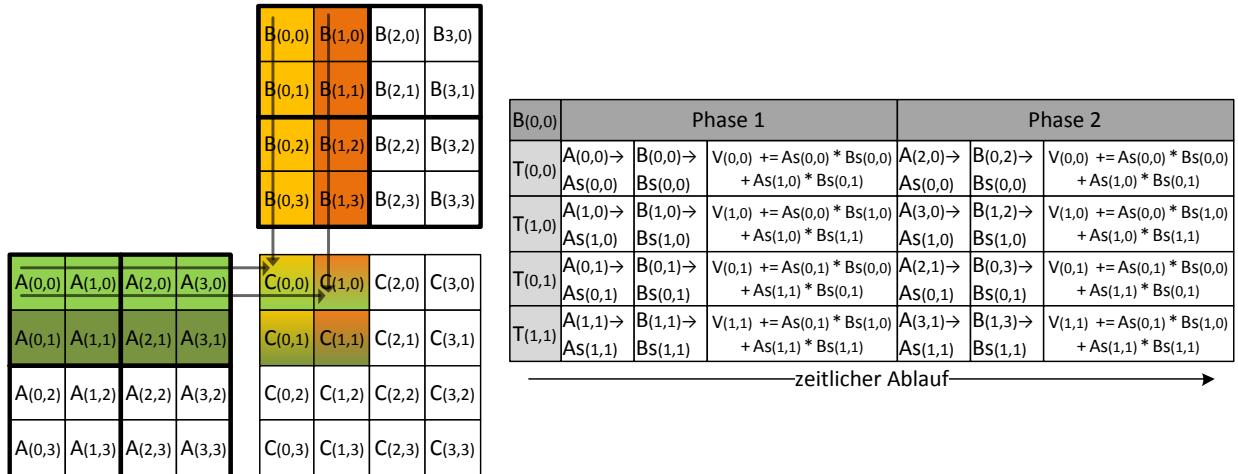


Abbildung 4.11.: Matrixmultiplikation mit Shared Memory

2×2 Tiles unterteilt. Die Berechnungen des Punktprodukts lassen sich jetzt in Phasen unterteilen. In einer Phase lädt zunächst jeder Thread ein Element des relevanten Tiles aus A und anschließend aus B in den Shared Memory (Quellcode 4.7, Zeile 20 bis 25); anschließend kann jeder Thread auf die benötigten Daten aus dem Shared Memory zugreifen. Jeweils nach den Kopiervorgängen und der Berechnung müssen die Threads eines Blocks synchronisiert werden (`__syncthreads()`). Der Ablauf ist auf der rechten Seite in Abbildung 4.11 dargestellt. In jeder Phase wird ein Tile von A und B in den Shared Memory geladen.

In jeder Phase wird das Produkt der Summe zweier Matrixelemente in der temporären Variablen v abgelegt. Für eine Matrix der Größe $M \times M$ und Tiles von `TILE_SIZE` \times `TILE_SIZE`, werden $M/TILE_SIZE$ Berechnungsphasen benötigt. Da immer das jeweils benötigte Tile passend in den Shared Memory eines Blocks geladen wird, bleibt die jeweilige Berechnung gleich und die Phasenzahl definiert sich über die Initialisierung der jeweiligen Schleife. Die Betrachtung von jeweils nur einem Unterproblem je Phase wird in der Informatik als *Lokalität* bezeichnet.

Quellcode 4.7 zeigt nun den modifizierten Kernel zur Matrixmultiplikation.

```

1 // Device Code
2 __global__ void MatAdd(float* A, float* B, float* C, int N)
3 {
4     // Deklarieren der Variablen für A und B im Shared Memory
5     __shared__ float As[TILE_SIZE][TILE_SIZE];
6     __shared__ float Bs[TILE_SIZE][TILE_SIZE];
7
8     int bx = blockIdx.x;
9     int by = blockIdx.y;
10    int tx = threadIdx.x;
11
12    for (int i = 0; i < N; i += TILE_SIZE)
13    {
14        for (int j = 0; j < N; j += TILE_SIZE)
15        {
16            float v = 0.0;
17
18            for (int k = 0; k < N; k += TILE_SIZE)
19            {
20                v += As[i][k] * Bs[k][j];
21            }
22
23            C[i + by][j + bx] = v;
24        }
25    }
26}
```

```

11     int ty = threadIdx.y;
13
14     // Berechnen der Reihen- und Zeilenindizes der Matrizen
15     int row = by * TILE_SIZE + ty;
16     int col = bx * TILE_SIZE + tx;
17
18     float V = 0;
19
20     // Schleife über die relevanten Elemente von A und B um C zu berechnen
21     for(int m = 0; m < N/TILE_SIZE; m++)
22     {
23         // Gemeinschaftliches Kopieren der Elemente vom Global in den Shared Memory
24         As[ty][tx] = A[row * N + (m * TILE_SIZE + tx)];
25         Bs[ty][tx] = B[(m * TILE_SIZE + ty) * N + col];
26         __syncthreads();
27
28         // Berechnung des Matrixprodukts aus Elementen im Shared Memory
29         for(int k = 0; k < TILE_SIZE; k++)
30         {
31             V += As[ty][k] * Bs[k][tx];
32         }
33         __syncthreads();
34     }
35 }
```

Quellcode 4.7: Optimierter Kernel zur Matrixmultiplikation

4.2.3.2. Mehrere Devices

Das Host-System kann über mehrere Devices verfügen, sprich das System besitzt mehrere GPUs auf einer oder mehreren Grafikkarten, die Anzahl an verfügbaren Devices kann mit der Funktion `cudaGetDeviceCount()` in Erfahrung gebracht werden. Die Funktion `cudaGetDeviceProperties()` liefert die Eigenschaften eines Devices. Das zu nutzende Device kann mit `cudaSetDevice()` festgelegt werden.

Ein Thread des Hosts kann zu jeder Zeit nur Device-Code auf dem jeweils zugewiesenen Device ausführen. Folglich müssen verschiedene Host-Threads ihren Code auf verschiedenen Devices ausführen.

Der Aufruf von `cudaThreadExit()` säubert alle für die Laufzeit relevanten Ressourcen, die mit dem aufrufenden Host-Thread verbunden sind. Jeder nachfolgende API-Aufruf aus diesem Host-Thread initialisiert die Laufzeitumgebung neu.

```

1 // Anzahl verfügbarer Devices
2 int deviceCount;
3 cudaGetDeviceCount(&deviceCount);
4 int device;
5 for(device = 0; device < deviceCount; device++)
6 {
7     // Eigenschaften der verfügbaren Devices
8     cudaDeviceProp deviceProp;
9     cudaGetDeviceProperties(&deviceProp, device);
10    if(deviceProp.major == 9999 && deviceProp.minor == 9999)
11        printf("Es ist kein CUDA-fähiges Gerät verfügbar.");
12    else if(deviceCount == 1)
13        printf("Es ist ein CUDA-fähiges Gerät verfügbar.");
14    else
15        printf("Es sind %d CUDA-fähige Geräte verfügbar.", deviceCount);
16 }
17 // Zuletzt aufgeführtes verfügbares Device nutzen
18 cudaSetDevice(deviceCount - 1);

```

Quellcode 4.8: Abfrage der Anzahl der Devices und deren Eigenschaften

4.2.3.3. Streams

Nebenläufigkeiten innerhalb eines parallelen Algorithmus werden durch CUDA-*Streams* ermöglicht, wie in Abbildung 4.12 dargestellt wird. Jeder Stream besteht dabei aus einer Reihe von sequenziell auszuführenden Befehlen.

Ein Stream wird durch das Erstellen eines Stream-Objekts mittels `cudaStreamCreate()`, vom Typ `cudaStream_t` definiert. Auflösen lässt er sich mit `cudaStreamDestroy()`. Dieses Objekt kann dann u.a. einem Kernel oder einer FFT (Kap. 4.2.4) zugewiesen werden:

```

Kernel<<< gridDim, blockDim, sharedSize, Stream >>>(...);
1 // gridDim (dim3) := Dimensionen und Größe des Grids in x- und y-Richtung, z = 0
2 // blockDim (dim3) := Dimensionen und Größe jedes Blocks in x-, y- und z-Richtung
3 // sharedSize (size_t) := optional, gibt die Anzahl an dynamisch festgelegten Bytes des
4 // Shared Memory pro Block an
// Stream (cudaStream_t) := optional, gibt den zugewiesenen Stream an

```

Quellcode 4.9: Vollständige Beschreibung eines Kernel-Aufrufs

Zu beachten ist, dass verschiedene Streams ihre Befehle ohne Rücksicht auf andere Streams ausführen. Sollen die Ergebnisse von zwei Streams zur weiteren Verarbeitung genutzt werden, müssen sie synchronisiert werden. Hierzu dient `cudaStreamSynchronize()`.

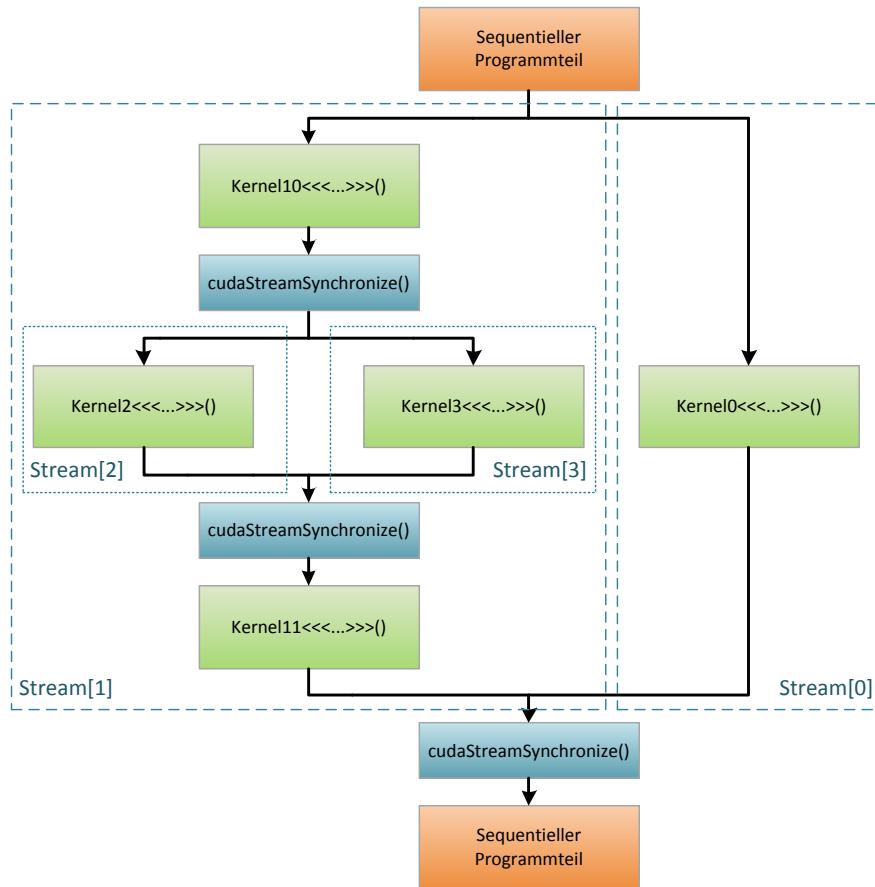


Abbildung 4.12.: Realisierung eines nebenläufigen Programms mit vier Streams

```

1 // Device Code
2 __global__ void Kernel0(...){...};
3 __global__ void Kernel10(...){...};
4 __global__ void Kernel11(...){...};
5 __global__ void Kernel2(...){...};
6 __global__ void Kernel3(...){...};
7
8 // Host Code
9 int main()
{
10 ...
11 ...
12 ...
13 // Bestimmen der Streams
14 cudaStream_t Stream[4];
15 for(int i = 0; i < 4; i++)
16     cudaStreamCreate(&Stream[i]);
17 ...
18 ...
19 ...
20 ...
21 // Ausführen der Kernel und Synchronisieren der Streams
22 Kernel10<<< gridDim, blockDim, 0, Stream[1]>>>(...);
23 cudaStreamSynchronize();
24 Kernel2<<< gridDim, blockDim, 0, Stream[2]>>>(...);
25 Kernel3<<< gridDim, blockDim, 0, Stream[3]>>>(...);
26 cudaStreamSynchronize();
27 Kernel11<<< gridDim, blockDim, 0, Stream[1]>>>(...);
28 Kernel0<<< gridDim, blockDim, 0, Stream[0]>>>(...);
29 cudaStreamSynchronize();

```

```

29
31     ...
32
33     // Auflösen der Streams
34     for(int i = 0; i < 4; i++)
35         cudaStreamDestroy(Stream[i]);
36
37 }
```

Quellcode 4.10: Beispiel zur Anwendung von Streams (zu Abb. 4.12)

Auch das nebenläufige Kopieren von Daten ist beispielsweise mit `cudaMemcpyAsync()` möglich.

4.2.4. FFT mittels CUDA

Die schnelle Fourier-Transformation, beschrieben in Kapitel 2.2.3, stellt ein zentrales Element des ITG-Algorithmuses (Abb. 2.5) dar. Für CUDA existiert die *CUFFT API* [38] als Implementierung eines Teile-und-herrsche-Algorithmus¹² zur parallelen Lösung der FFT auf CUDA GPUs. Er basiert auf der *FFTW API* [17] und bietet folgende Funktionalitäten:

- 1D, 2D und 3D Transformation komplexer und realwertiger Daten
- Stapeltransformationen zur mehrfach parallelisierten Transformation einer beliebigen Dimension
- 1D Transformationen mit bis zu 8 Millionen Elementen
- 2D und 3D Transformationen im Intervall von [2, 16384] Elementen in jeder Dimension
- In-place und out-of-place Transformationen komplexer und realwertiger Daten
- Transformationen mit doppelter Genauigkeit auf kompatibler Hardware
- Unterstützung für die Ausführung von Transformationen in Streams

Tabelle 4.2.: Fähigkeiten der CUFFT API

Zum Ausführen einer FFT muss zunächst ein *Plan* des Typs `cufftHandle` erstellt werden. Dazu gibt es die Funktionen `cufftPlan1d()`, `cufftPlan2d()` und `cufftPlan3d()` für die jeweilig benötigte Dimension. Mehr Flexibilität liefert die Funktion `cufftPlanMany()`. Der erstellte Plan beinhaltet Informationen zum optimalen Ausführen - Ausführen in möglichst wenigen FLOPS - der FFT für die gewünschte Art der FFT, die Datengröße

¹²Der Grundsatz *teile und herrsche* beschreibt in der Informatik das Prinzip, eine Problemstellung so lange in Unterprobleme zu zerlegen, bis diese überblickt ("beherrscht") werden kann. Letztendlich werden alle Teillösungen zur Lösung des Gesamtproblems rekombiniert.

und den Datentyp. Vorteil des Plans ist es, dass er nur einmal erstellt werden muss und danach beliebig oft bis zum Aufruf von `cufftDestroy()` genutzt werden kann.

```

1 // Arten der Transformation
2 typedef cufftType_t {
3     CUFFT_R2C, // Realwertig zu komplexwertig (überlappend)
4     CUFFT_C2R, // Komplexwertig (überlappend) zu realwertig
5     CUFFT_C2C, // Komplexwertig zu komplexwertig, interleaved
6     CUFFT_D2Z, // Double-realwertig zu double-komplexwertig
7     CUFFT_Z2D, // double-komplexwertig to double-realwertig
8     CUFFT_Z2Z // double-komplexwertig to double-komplexwertig
9 } cufftType;
10
11 // Zugehörige Datentypen
12 typedef float cufftReal;
13 typedef double cufftDoubleReal;
14 typedef cuComplex cufftComplex;
15 typedef cuDoubleComplex cufftDoubleComplex;
```

Quellcode 4.11: Mögliche Transformationen mittels CUFFT und zugehörige Datentypen

Die Ausführung der CUFFT erfolgt über den jeweils zum Plan passenden Funktionsaufruf: `cufftExecC2C()`, `cufftExecR2C()`, `cufftExecC2R()`, `cufftExecZ2Z()`, `cufftExecD2Z()` und `cufftExecZ2D()`. Neben dem Plan und den Pointern auf die Datenarrays für Eingangs- und Ausgangsdaten muss noch die Transformationsrichtung, `CUFFT_FORWARD` oder `CUFFT_INVERSE`, angegeben werden.

4.2.4.1. FFT in einem Stream

Bei Betrachtung des Ablaufdiagramms des ITG-Algorithmus (2.5) fällt das Potential der Nebenläufigkeit auf. Somit kommt es gelegen, dass sich jeder CUFFT-Plan mit der Funktion `cufftSetStream()` an einen Stream des Datentyps `cudaStream_t` binden lässt und somit die nebenläufige Ausführung mehrerer Transformationen ermöglicht.

4.2.4.2. Bildshift

Eine wichtige Eigenschaft der Fourier-Transformierten (Kap. 2.2.3; Gl.2.33) und ihrer Inversen (Gl.2.34) ist deren unendliche Periodizität in f_x - und f_y -Richtung [18].

Überlicherweise befindet sich eine Periode einer eindimensionalen Fourier-Transformierten im Intervall $[-\frac{M}{2}, \frac{M}{2}]$ (Abb. 4.13 oben). Bei Bilddaten liegt die erste Periode im Intervall $[0, M]$, also um $\frac{M}{2}$ in Richtung f_x verschoben (Abb. 4.13 unten). Es ist bekannt, dass

$$\mathcal{F} \left\{ g(x) e^{i 2\pi \frac{x \cdot x_0}{M}} \right\} = G(f_x - x_0) . \quad (4.1)$$

Durch Multiplikation mit dem Exponentialterm im Ortsbereich lässt sich also der Wert des Ursprungs $G(0)$ im Frequenzraum auf den Ort x_0 verschieben. Setzt man nun $x_0 = \frac{M}{2}$, so wird der Exponentialterm zu $e^{i\pi x}$, dies entspricht $(-1)^x$, da x eine reelle Zahl ist.

$$\mathcal{F} \{ g(x) (-1)^x \} = G(f_x - \frac{M}{2}) \quad (4.2)$$

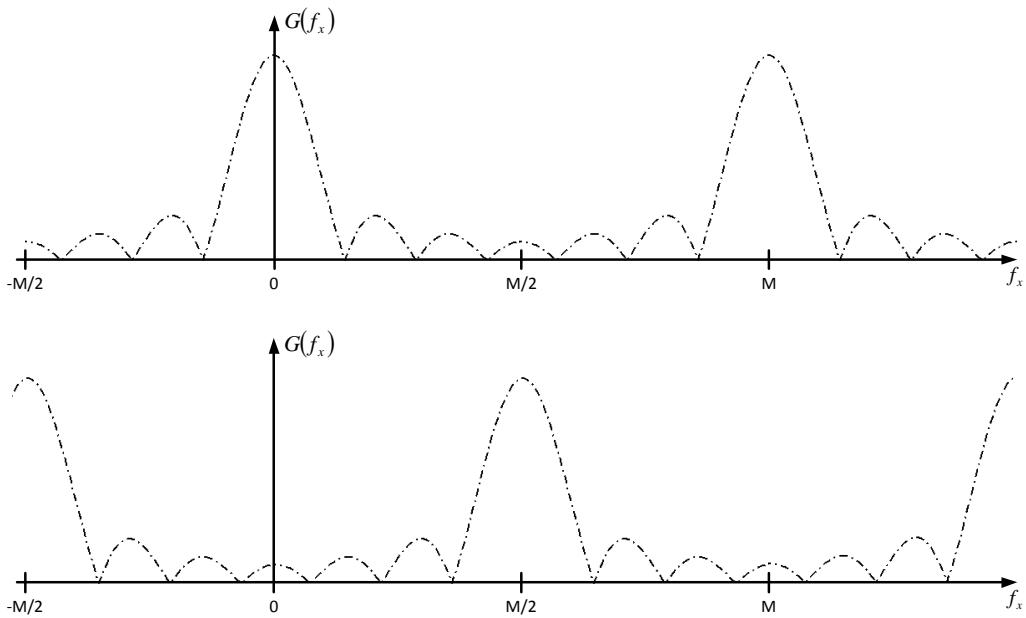


Abbildung 4.13.: Periodizität einer eindimensionalen Fourier-Transformierten, im unteren Bild wurde $f(x)$ vor der Transformation mit dem Faktor $(-1)^x$ multipliziert

Nach Anwendung der Gleichung 4.2 befindet sich der Wert $G(0)$ wie gewünscht in der Mitte des Intervalls $[0, M]$.

Die Betrachtung im Zweidimensionalen ist schlechter darstellbar, lässt sich aber analog zum eindimensionalen Fall betrachten. In Abbildung 4.14 ist zu erkennen, dass es erstrebenswert ist, den Wert $G(0, 0)$ an die Position $(\frac{M}{2}, \frac{N}{2})$ zu verschieben. Analog zu Gleichung 4.2 ergibt sich daraus

$$\mathcal{F} \left\{ g(x)(-1)^{x+y} \right\} = G(f_x - \frac{M}{2}, f_y - \frac{N}{2}) . \quad (4.3)$$

Diese Methode wird als Quadrantenshift bezeichnet. Anschaulich betrachtet werden hier jeweils die beiden diagonal gegenüberliegenden Quadranten miteinander vertauscht (Abb. 4.15).

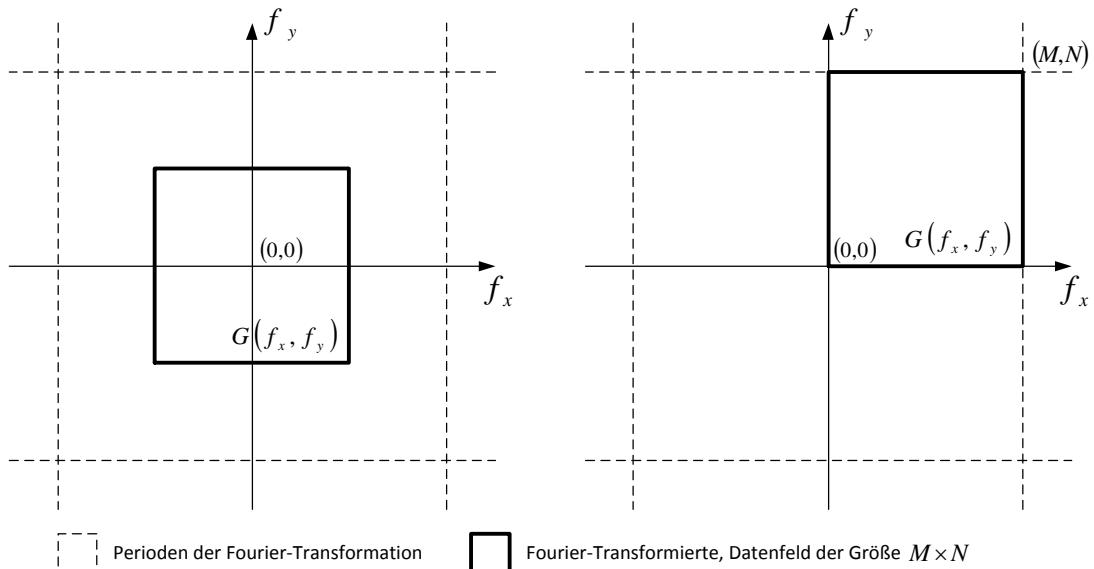


Abbildung 4.14.: Periodizität einer zweidimensionalen Fourier-Transformierten, rechts wurde $f(x, y)$ vor der Transformation mit dem Faktor $(-1)^{x+y}$ multipliziert

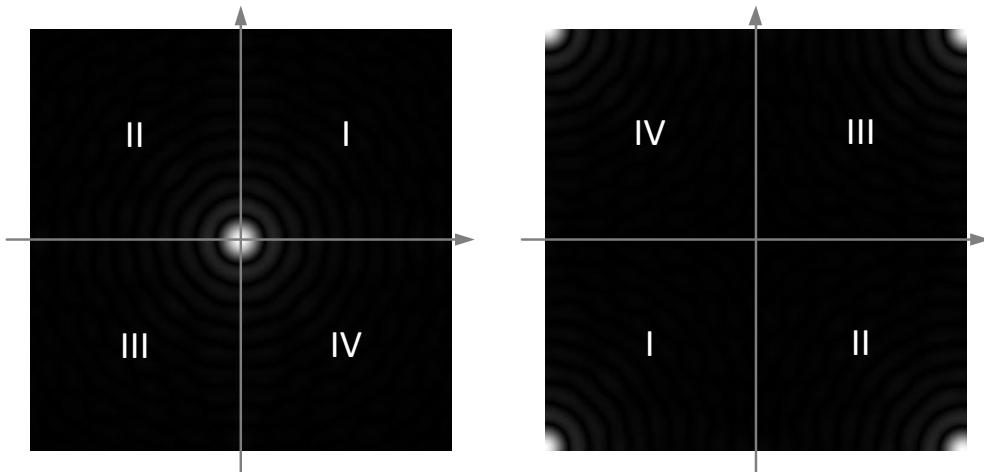


Abbildung 4.15.: Quadrantenshift eines Airy-Scheibchens

4.2.4.3. Beispielprogramm einer FFT von Bilddaten

```

1 #define NX 1024 // Bildweite
2 #define NY 1024 // Bildhöhe
3
4 // Kernel zum Quadrantenshift
5 __global__ void shiftImage(cufftComplex data)
6 {
7     int i = blockIdx.x * blockDim.x + threadIdx.x;
8     int j = blockIdx.y * blockDim.y + threadIdx.y;
9     data.x[i+j*NX] *= (-1)^(x+y)      // Quadrantenshift des Realteils
10    data.y[i+j*NX] *= (-1)^(x+y)      // Quadrantenshift des Imaginärteils
11 }
12
13 // Hauptprogramm
int main()

```

```

15  {
16      // CUFFT-Plan
17      cufftHandle plan;
18
19      // Arrays für Eingangs- und Ausgangsbilddaten
20      cufftComplex *indata, *outdata;
21
22      // Belegen des benötigten Speichers
23      cudaMalloc((void**)&indata, NX * NY * sizeof(cufftComplex));
24      cudaMalloc((void**)&outdata, NX * NY * sizeof(cufftComplex));
25
26      // Erstellen des zweidimensionalen Plans
27      cufftPlan2D(&plan, NX, NY, CUFFT_C2C);
28
29      // Einladen der Bilddaten in indata
30      getImage(indata);
31
32      // Quadrantenshift der Bilddaten
33      dim3 threadsPerBlock(16,16);
34      dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
35      shiftImage<<<numBlock, threadsPerBlock>>>(indata);
36
37      // Nutzen des CUFFT Plans zur Vorwärtstransformation des Signals, out-of-place
38      cufftExecC2C(plan, indata, outdata, CUFFT_FORWARD);
39
40      // Quadrantenshift der transformierten Daten
41      shiftImage<<< numBlock, threadsPerBlock>>>(outdata);
42
43      // Ausgabe des Bildes
44      showImage(outdata);
45
46      // Auflösen des Plans
47      cufftDestroy(plan);
48
49      // Freigabe des Speichers
50      cudaFree(indata);
51      cudaFree(outdata);
52 }

```

Quellcode 4.12: Komplexe 2D-Transformation eingeladener Bilddaten

4.2.5. Programmoptimierung

Im Idealfall würde die Beschleunigung durch Parallelisierung linear verlaufen. Durch Verdopplung der verfügbaren Rechenkerne sollte die Laufzeit halbiert werden, eine zweite Verdopplung sollte die Laufzeit erneut halbieren. Leider kann kein implementierbarer paralleler Algorithmus diesem Anspruch genügen.

Eine Einschätzung über die tatsächlich realisierbare Beschleunigung auf einem Parallelrechner gibt das *Amdahlsche Gesetz*¹³ [39] an. Es geht davon aus, dass kein Programm vollständig parallelisiert werden kann, ein sequentieller Anteil wird immer vorhanden sein. P beschreibt nachfolgend den parallelisierten Anteil an der Programmlaufzeit und $(1 - P)$ beschreibt den sequentiellen Anteil. Die Gesamtlaufzeit auf einem Kern ergibt sich dann aus der Summe:

$$1 = (1 - P) + P \quad (4.4)$$

¹³benannt nach Gene Myron Amdahl (*1922), US-amerikanischer Computerarchitekt und Hi-tech-Unternehmer norwegischer Abstammung

Die mögliche Beschleunigung B hängt weiter von der Anzahl verfügbarer Rechenkerne N ab.

$$B \leq \frac{1}{(1 - P) + \frac{P}{N}} \leq \frac{1}{1 - P} \quad (4.5)$$

Der Quotient $\frac{P}{N}$ gibt den beschleunigten parallelen Anteil an. Die Abbildung 4.16 bildet diese Gesetzmäßigkeit für verschiedene große parallelisierte Programmanteile ab. Das Am-

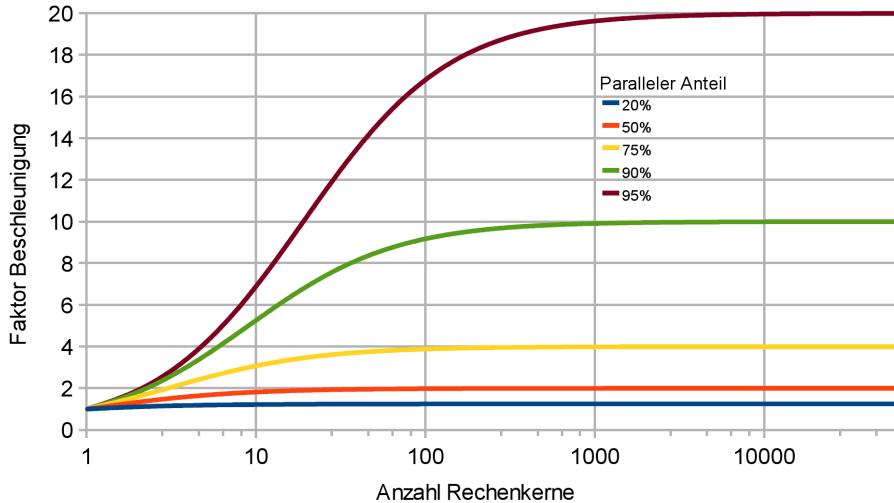


Abbildung 4.16.: Amdahlsches Gesetz: Geschwindigkeitsgewinn mit steigender Anzahl Prozessoren, abhängig vom parallelisierten Anteil eines Problems.

dahlsche Gesetz geht davon aus, dass die verarbeitbare Menge an Daten proportional zur Anzahl von Rechenkernen ansteigt oder anders ausgedrückt: Der parallelisierte Anteil an der Programmlaufzeit bleibt konstant. Das darauf aufbauende *Gustafsons Gesetz*¹⁴ [40] berücksichtigt hingegen die Verhältnismäßigkeit zwischen Datenmenge und Rechenkernen. Die Variable s gibt prozentual den zeitlich sequentielle Anteil der Laufzeit an dem parallelen Programm an.

$$B \leq N + (1 - N)s \quad (4.6)$$

Es lässt sich sagen, dass das Amdahlsche Gesetz (Gl. 4.5) die maximale Optimierbarkeit eines sequenziellen Codes durch Parallelisierung angibt und das Gustafsonssche Gesetz (Gl. 4.6), wie weit ein paralleles Programm durch die Nutzung von mehr Rechenkernen weiter beschleunigt werden kann.

Ein weiterer interessanter Wert ist die erreichte Effizienz, also die tatsächliche Ausnutzung der Systemressourcen. Die prozentuelle Effizienz E lässt sich aus dem Quotienten aus beobachteter Beschleunigung B_{real} und der Anzahl genutzter Kerne N_{real} bestim-

¹⁴benannt nach John L. Gustafson (*1955), US-amerikanischer Informatiker und Geschäftsmann

men.

$$E = \frac{B_{real}}{N_{real}} \quad (4.7)$$

4.2.5.1. Regeln zur Programmoptimierung

Die Optimierung eines bestehenden Programms ist ein zeitaufwändiger Prozess und bringt die Gefahr mit sich, neue Programmierfehler, sogenannte *Bugs*, in ein bisher fehlerfreies Programm einzufügen. In diesem Abschnitt sollen einige gängige Optimierungsansätze in der Entwurfs- und Quellcodeebene für CUDA-Programme angegeben werden.

Optimierungsstrategien für parallelisierte Programme lassen sich in drei Ansätze unterteilen [33]:

- Maximale parallele Ausführung zur maximierten Ausnutzung der Ressourcen
- Optimierte Speichernutzung für maximalen Speicherdurchsatz
- Optimierte Anweisungen für maximalen Anweisungsdurchsatz

Welche der Strategien den größten Erfolg verspricht, hängt vom konkreten Problem ab und sollte durch Messungen eruiert werden. Eine gute Methode zur Performancemessung bietet das Programm *CUDA profiler* aus dem *CUDA Toolkit* [37].

Hier sollen nun die in der Software berücksichtigten Optimierungsansätze aufgezeigt werden.

- Kopiervorgänge vom Host-Memory in den Global Memory des Devices mit der Funktion `cudaMemcpy()` sind sehr zeitaufwändig und sollten so selten wie möglich ausgeführt werden. Um Kopiervorgänge möglichst effektiv vorzunehmen, sollten mehrere Datenfelder zu einem großen Datenfeld zusammengefasst werden. Weiter kann das asynchrone Ausführen der Kopiervorgänge mit `cudaMemcpyAsync()` sinnvoll sein.
- Wann immer möglich, sollte der Einsatz von Global Memory minimiert und lieber im Shared Memory gearbeitet werden. CUDA-Kernel lassen sich in den meisten Fällen durch den geschickten Einsatz der verschiedenen Speicherebenen (Kap. 4.2.3.1, Kap. 4.3) beschleunigen.
- CUDA bietet spezielle mathematische Funktionen (*fast math library*) für Kernel, die schneller berechenbar sind als die üblichen Funktionen der `math.h` [33]. Beispiele sind `__powf()`, `__[u]mul24()`, `__sinf()`, `__logf()`, `__fdividef()` oder auch `__expf()`.
- *Bitshifts* sind schneller als Divisionen. Der Shift `i >> 1` entspricht zwar `i / 2`, kann aber bedeutend schneller berechnet werden.
- Eine Division, die keiner Zweierpotenz entspricht, sollte durch reziprokes Radizieren realisiert werden. `i / u` entspricht somit `i * rsqrtf(u * u)`.

- Wenn möglich sollte ein Algorithmus in Streams unterteilt werden (Kap. 4.2.3.3).
- Synchronisationen mit `__syncthreads()` in Kernel nur ausführen, wenn sie wirklich nötig sind.
- Double-Precision Gleitkommavariablen benötigen viel Speicherplatz und sind nur selten nötig. Sie lassen sich aber nicht immer vermeiden wie beispielsweise bei der Nutzung der `e`-Funktion. Genauso sollte die automatische Konvertierung von `double` zu `float` vermieden werden.
- Bei Kernel mit vielen übergebenen Argumenten sollten einige im Constant Memory abgelegt werden, um den Speicherzugriff zu vereinfachen und so Zeit einzusparen.
- Die Anzahl an Threads pro Block sollte einem Vielfachen von 32 Threads entsprechen.
- Zählervariablen in Schleifen sollten dem Typ `int` anstatt `unsigned int` entsprechen.

Neben den hier angegebenen Optimierungsstrategien gibt es noch einige weitere Ansätze, die bei der Entwicklung der Software zu dieser Arbeit nicht verwendet wurden; Sie werden größtenteils durch die Quellen [33], [34], [26] und [41] abgedeckt.

4.3. Reduktion als gängiges paralleles Programmierkonzept

Ein oft genutztes Programmierkonzept beim Parallel Programming, besonders im Bereich der Bildverarbeitung, ist die Reduktion. An ihrem Beispiel soll gezeigt werden, wie komplex es sein kann, eine im Sequenziellen einfache Aufgabe effektiv in eine Kernel-Funktion zu übertragen.

Als Beispiel soll eine Funktion zur Summenbildung dienen. Die einfachste sequenzielle Variante besteht aus einer `for`-Schleife, in der alle Werte des Arrays `values` aufsummiert werden.

```

1 float sumFunc(float values[])
2 {
3     // Berechnung der Anzahl der Werte im Array values
4     int n = sizeof(values)/sizeof(float);
5
6     // Temporäre Variable
7     float sum = 0;
8
9     // Summenbildung
10    for(int i = 0; i < n; i++)
11        sum += values[i];
12
13    // Rückgabe des Ergebnisses
14    return sum;
}
```

Quellcode 4.13: Einfache sequenzielle Summenbildung

Eine Reduktion besteht aus einer Sequenz von mehreren Schritten. In jedem Schritt wird ein Wert an einer Stelle $i < n/2$ mit dem Wert an Stelle $i + n/2$ addiert, wobei n die Anzahl an Elementen im Array ist. So wird die relevante Größe des Arrays mit jedem Schritt halbiert. Dieser Vorgang wird schrittweise wiederholt, bis im Array an der Stelle 0 die Summe steht. Der Quellcode 4.14 und die Abbildung 4.17 verdeutlichen das Prinzip.

```

1 float sumReduce(float values[])
{
3     int n = sizeof(values)/sizeof(float);
5     // Reduktionsschleife
7     do{
        // Halbieren der relativen Arraygröße
        n /= 2;
9
        // Summenbildung in einem Schritt
11    for(int i = 0; i < n; i++)
        values[i] += values[i + n];
13
    }while(n > 0);
15
17    return values[0];
}

```

Quellcode 4.14: Einfache sequentielle Funktion zur Reduktion

Die Herausforderung liegt nun in der Übersetzung dieser simplen Funktion in einen effizienten Kernel [42]. Die nachfolgenden Beispiele wurden stark vereinfacht, um sich auf das Wesentliche konzentrieren zu können.

Die parallelisierte Reduktion auf dem Device wird durch Quelltext 4.15 angegeben. Die Host-Funktion `do_reduce()` (Zeile 37) liest ein `float`-Array `indata` ein und gibt die Summe der Werte aller Elemente des Arrays an den Host zurück. Die Reduktionsschleife ist vergleichbar mit der in Quellcode 4.14, nur dass die eigentliche Reduktion hier möglichst in Threads des Kernels `reduce()` (Zeile 5) abläuft. Das Verfahren der Reduktion mit Threads innerhalb eines Blocks gibt die Abbildung 4.17 wieder. Um die Ergebnisse der einzelnen Reduktionen in einem Block weiter zu reduzieren, ist ein erneutes Abarbeiten der Reduktionsschleife und eine temporäre Variable `outdata` nötig. Die Reduktionsschleife wird hierbei entscheidend seltener durchlaufen als zuvor. Der eigentliche Device-Kernel wurde darauf ausgelegt, $\frac{n}{2}$ Threads zu nutzen. Zudem wurde wieder mit Shared Memory gearbeitet (Kap. 4.2.3.1), und der erste Reduktionsschritt wird bereits beim Kopieren der notwendigen Daten in den Shared Memory ausgeführt (Zeile 17). Der Hauptteil der Reduktion soll dann im Shared Memory mehrerer Blocks geschehen (Zeile 23).

```

1 #define SIZE ...
3 // Device-Code
// Reduktionskernel
5 __global__ void reduce(float *indata, float *outdata)
{
7     int n = SIZE;
8     __shared__ float sdata[SIZE];
9
11    // erster Teil der Reduktion: Kopieren der Daten von Global zu Shared Memory und
12    // erste Reduktion
13    unsigned int tid = threadIdx.x;
14    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
15
16    float mySum = (i < n) ? indata[i] : 0;
17
18    if (i + blockDim.x < n)
19        mySum += indata[i+blockDim.x];
20
21    sdata[tid] = mySum;
22    __syncthreads();
23
24    // Reduktion im Shared Memory
25    for(unsigned int s=blockDim.x/2; s>0; s>>=1)
26    {
27        if (tid < s)
28            sdata[tid] = mySum = mySum + sdata[tid + s];
29
30        __syncthreads();
31    }
32
33    // Ergebnisse des Blocks in Global Memory schreiben
34    if (tid == 0) outdata[blockIdx.x] = sdata[0];
35}
36
37 // Host-Code
// Funktion zum Aufruf der Reduktion vom Host
38 float do_reduce(float *indata)
{
39     // Variablen deklarieren und initialisieren
40     ...
41
42     // Block- und Gridgröße festlegen
43     dim3 dimBlock(threads, 1, 1);
44     dim3 dimGrid(blocks, 1, 1);
45
46     // Reduktionsvariable
47     int s = blocks;
48
49     // Reduktionsschleife
50     do{
51         // Schrittweiser Aufruf des Reduktionskernels
52         reduce<<< dimGrid, dimBlock, smemSize >>>(indata, outdata);
53
54         // Reduzieren der Reduktionsvariablen
55         s = (s + (threads*2-1)) / (threads*2);
56     }while(s > 1)
57
58     // Kopieren des Ergebnisses auf Host und Rückgabe
59     cudaMemcpy( &result, outdata, sizeof(float), cudaMemcpyDeviceToHost);
60     return result;
61 }
```

Quellcode 4.15: Erster Reduktions-Kernel

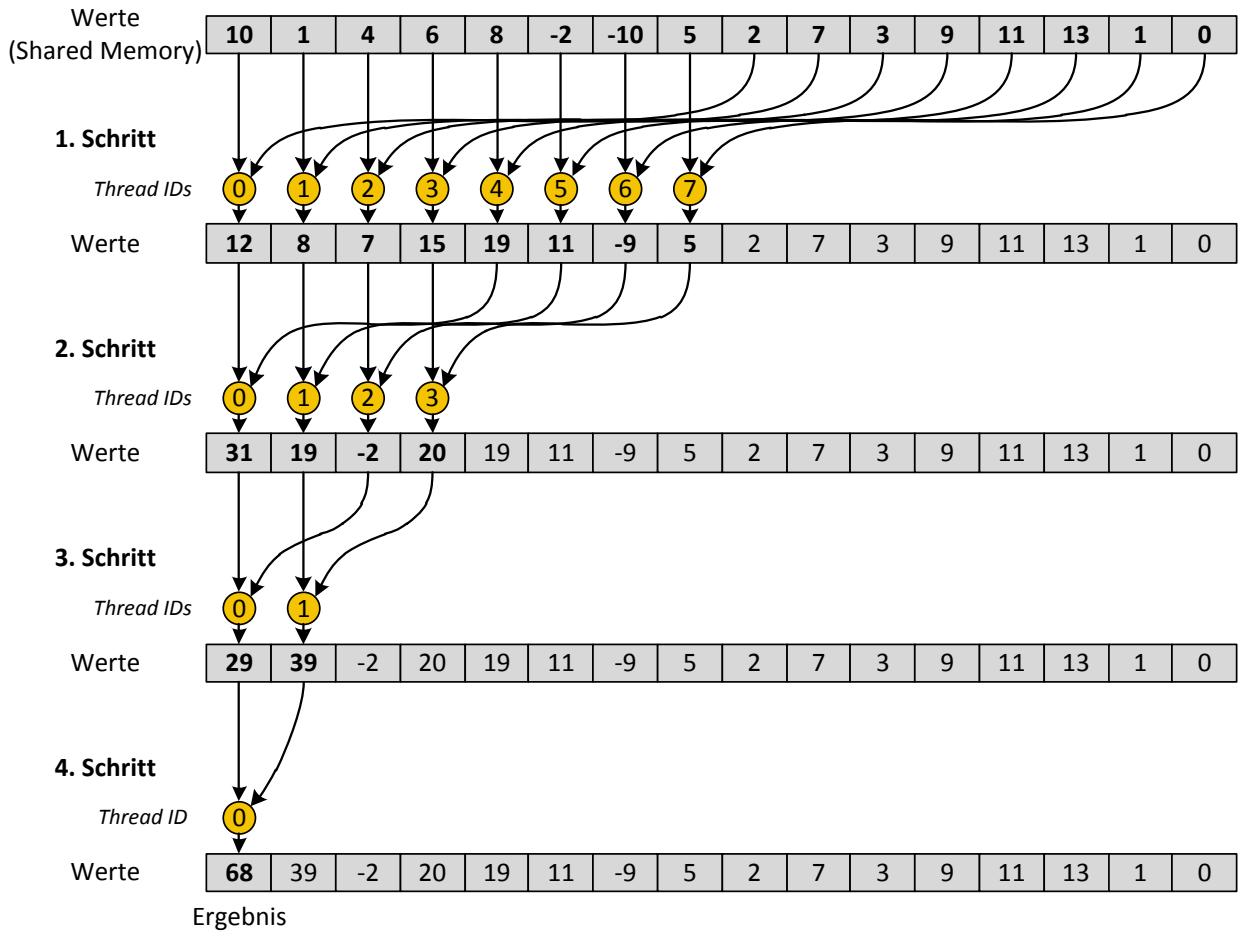


Abbildung 4.17.: Ablauf einer Reduktion von 16 Werten mit bis zu 8 Threads

Der CUDA-Kernel ermöglicht bereits die Reduktion eines Arrays mit 2^{22} Elementen in ca. 1ms [42]. Ein interessanter Ansatz zur Optimierung des Kernels ist das Entfalten der Reduktionsschleife. Hierbei wird die Reduktion in Abhängigkeit von der Größe des aktiven Blocks ausgeführt (Quellcode 4.16 ab Zeile 16). Der Vorteil hierbei ist, dass bereits beim Kompilieren alle if-Abfragen ausgewertet werden. Bei der eigentlichen Laufzeit wird dann jedem Block der passende Kernel mit der passenden Reduktion zugewiesen. Somit ergibt sich eine sehr effektive Schleife, die die Reduktion der 2^{22} Elemente in ca. 0,4ms [42] ermöglicht. Der Kernel wurde durch diese Optimierung um den Faktor 2,5 beschleunigt.

```

1  __global__ void reduce(float *indata, float *outdata, unsigned int blockSize)
{
3      int n = SIZE;
5      __shared__ float sdata[SIZE];
7      unsigned int tid = threadIdx.x;
8      unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;

9      float mySum = (i < n) ? indata[i] : 0;
10     if (i + blockSize < n)
11         mySum += indata[i+blockSize];

13     sdata[tid] = mySum;
14     __syncthreads();
15
16     if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum + sdata[tid + 256]; } __syncthreads(); }
17     if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum + sdata[tid + 128]; } __syncthreads(); }
18     if (blockSize >= 128) { if (tid < 64) { sdata[tid] = mySum + sdata[tid + 64]; } __syncthreads(); }
19
20     if (tid < 32)
21     {
22         volatile float* smem = sdata;
23         if (blockSize >= 64) { smem[tid] = mySum + smem[tid + 32]; }
24         if (blockSize >= 32) { smem[tid] = mySum + smem[tid + 16]; }
25         if (blockSize >= 16) { smem[tid] = mySum + smem[tid + 8]; }
26         if (blockSize >= 8) { smem[tid] = mySum + smem[tid + 4]; }
27         if (blockSize >= 4) { smem[tid] = mySum + smem[tid + 2]; }
28         if (blockSize >= 2) { smem[tid] = mySum + smem[tid + 1]; }
29     }
30
31     if (tid == 0) outdata[blockIdx.x] = sdata[0];
}

```

Quellcode 4.16: Optimierter Reduktions-Kernel

Analog zur Summenbildung lassen sich auch Funktionen zur Mittelwertbildung oder zum Auffinden von Minimal- und Maximalwerten erstellen. Eine ausführliche Beschreibung dieser Reduktion ist in [42] zu finden.

5. Bildverarbeitung

Die digitale Bildverarbeitung beeinhaltet die Aufarbeitung, Verarbeitung, Speicherung und Darstellung von digitalisierten, visuellen zweidimensionalen oder dreidimensionalen Daten. Sie dient der Informationsgewinnung aus digitalen Ursprungsdaten.

In diesem Kapitel soll dementsprechend auf die in dieser Arbeit relevanten Methoden zur Aufbereitung von digitalen Bilddaten eingegangen werden. So beschäftigt sich Abschnitt 5.1.6 mit der genutzten Programmierschnittstelle zum Erstellen und Ausgeben von Computergrafiken. Die ledigliche Ausgabe von zweidimensionalen Bilddaten wird in 5.1.6 beschrieben, wogegen 5.1.7 die Konvertierung einer zweidimensionalen Phasenverteilung in ein dreidimensionales Modell beschreibt. Als manipulative Methode der Bildverarbeitung wird die *projektive Transformation* in Unterkapitel 5.3 erläutert. In 5.5 wird ein Überblick des Ablaufs der erstellten Software *φScope* gegeben. Vervollständigend zeigt Abschnitt 5.2, welchen Bedingungen das genutzte Kamerassystem genügen muss und wie dies erreicht wird.

5.1. OpenGL

OpenGL ist eine plattform- und programmiersprachenunabhängige Programmierschnittstelle. Sie ermöglicht durch ein abstraktes Programmiermodell mit über 250 verschiedenen Funktionen die Entwicklung von Programmen mit komplexen dreidimensionalen Echtzeitcomputergrafiken für eine Vielzahl unterschiedlicher Hardwareplattformen und Betriebssysteme [43].

Der offene Industriestandard OpenGL wurde von Silicon Graphics Inc. entwickelt und 1992 erstmals vorgestellt [44]. Seit 2006 wird die Weiterentwicklung der OpenGL-API von dem Non-Profit-Technologiekonsortium *Khronos Group* betreut [45]. Zu dem Konsortium gehören mehr als einhundert Soft- und Hardwareunternehmen, darunter NVIDIA, AMD/ATI, Intel Corporation, Apple Inc., Dell, Google, Epic Games, id Software, Motorola, Nokia, Oracle/ Sun Microsystems und Texas Instruments, um nur einige zu nennen.

Die OpenGL-API ermöglicht dem Entwickler das *Rendern*¹ einer dreidimensionalen Szene. Hierzu müssen zunächst Objekte durch geometrische Grundformen konstruiert werden, um daraus später eine mathematische Beschreibung der Objekte liefern zu können. Zu den verfügbaren Grundformen (*Primitives*) gehören Punkte (*Points*), Linien (*Lines*), Vielecke (*Polygones*) und Binärbilder (*Bitmaps*), die durch ihre *Vertices*² beschrieben

¹ auch Bildsynthese, beschreibt das Erzeugen eines Bildes aus Rohdaten.

² Ein Vertex ist ein Eck- bzw. Scheitelpunkt eines Primitives und enthält neben der Positionsangabe in einem 3D-Vektor auch weitere Informationen wie beispielsweise Farbe oder Transparenz.

werden. Die so erstellten Objekte werden im dreidimensionalen Raum platziert, und die später wiedergegebene Perspektive wird durch das Darstellungsfeld³ (*Viewport*) definiert. Die Farbanteile, in denen ein Objekt erscheinen wird, richten sich nach der explizit zugewiesenen Farbe, der eventuell "aufgeklebten" *Textur*⁴ und nach der optional beschriebenen Beleuchtungssituation und zugehörigen Materialeigenschaften. OpenGL kennt zum Zeitpunkt der Ausführung die mathematische Beschreibung der vorhandenen Objekte, kann bestimmen welche Objekte sichtbar sind, simuliert das Aussehen der Oberflächen und Materialeigenschaften (*Shading*) und berechnet die Lichtverteilung innerhalb der Szene, die sich durch direkte und indirekte Beleuchtung ergibt. So lässt sich ein dreidimensional wirkendes Bild synthetisieren und ausgeben. Das erstellte Bild besteht aus Pixeln⁵, die im Speicher als *Bitplanes* abgelegt werden. Ein Bitplane enthält jeweils eine Information wie z.B. den Rotanteil eines Pixels. Diese Elemente sind wiederum gruppiert in einem *Framebuffer*, der Informationen über die Helligkeit aller Pixel enthält und später zur Anzeige genutzt wird.

5.1.1. Zustandsautomat und Rendering-Pipeline

Der endliche Zustandsautomat ist ein Modell der Informatik, dessen Verhalten sich mit einer endlichen Anzahl an Zuständen, Zustandsübergängen und Aktionen vollständig beschreiben lässt. Auch OpenGL ist ein Zustandsautomat, und somit bleibt jeder eingesetzte Zustand bestehen, bis er explizit geändert wird [46]. Beispielsweise ist die aktuelle Farbe eines Vertex ein Zustand. Ist die Vertexfarbe einmal gewählt, bleibt sie für alle nachfolgend erstellten Vertices bestehen, bis eine neue Farbe zugewiesen wird. Auf die gleiche Weise können z.B. Lichtquellen ein- und ausgeschaltet werden. Da jede Änderung im Zeichenmodus eine aufwändige Neuordnung der Grafikpipeline erzwingt, lassen sich durch den Automaten redundante Neuordnungen in der Rendering-Pipeline vermeiden. Die *Rendering-Pipeline* ist eine Modellvorstellung des Ablaufs beim Synthetisieren einer Computergrafik, sowohl auf Software- wie auf Hardwareseite. Dieser Ablauf, wie Abbildung 5.1 ihn zeigt, ist nicht unbedingt das strikte Vorgehen der OpenGL-Implementierung, veranschaulicht aber die Arbeitsweise von OpenGL beim Rendern.

Geometrische Daten wie Punkte, Linien und Polygone durchlaufen die Schritte für Auswertung und Vertexoperationen, wenngleich Pixel-Daten (Pixel, Bilder, Binärbilder) lediglich den Block für Pixeloperationen durchlaufen müssen. Beide Typen von Eingangsdaten passieren die gleichen finalen Schritte (Rasterung und Fragmentoperationen), bevor die endgültigen Pixel-Daten in den Framebuffer geschrieben werden und somit zur Ausgabe bereitstehen.

³Der Viewport beschreibt den sichtbaren Bildausschnitt einer 3D-Szene. Hierzu wird die Position und der Blickwinkel des "Betrachters" definiert.

⁴Texturen sind Bilder, die auf der Oberfläche eines virtuellen Objekts dargestellt werden. Ein Pixel der Textur wird als *Texel* bezeichnet.

⁵Ein Pixel beschreibt das kleinste Element eines Bildrasters. Es ist ein Kunstwort aus den englischen Wörtern *pictures*, umgangssprachlich "pix", und *element* und lässt sich mit *px* abkürzen.

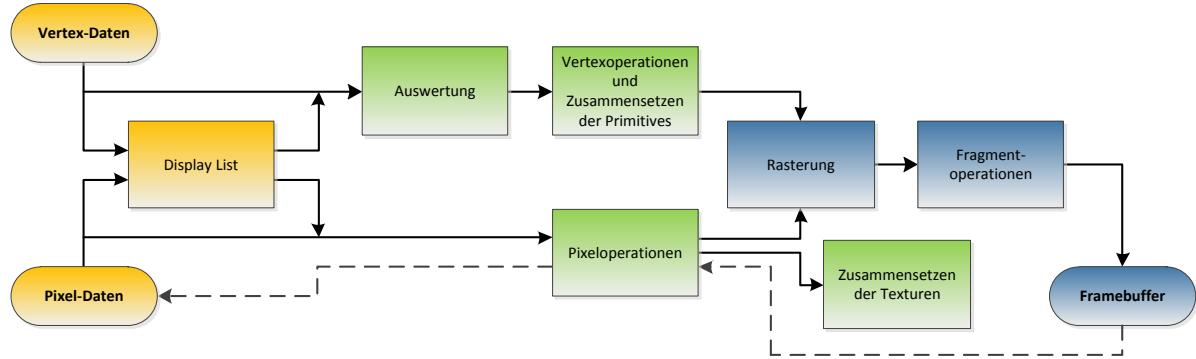


Abbildung 5.1.: Ablauf der Rendering-Pipeline von OpenGL

Display Lists

Alle Daten, egal ob geometrisch oder Pixel, können in einer Display List zwischengespeichert und später wieder von dort abgerufen werden. Bei einmaliger Verwendung der Daten (*immediate mode*) ist dieser Schritt überflüssig.

Auswertung

Geometrische Primitives werden durch Vertices beschrieben. Die Auswertung ermöglicht es, aus den angegebenen Eckpunkten die nötigen Oberflächen-Vertices abzuleiten. Hierbei handelt es sich um eine mehrgliedrige Abbildung der Oberfläche, wobei die Oberflächennormalen, Texturkoordinaten, Farbwerte und die räumlichen Koordinaten der erstellten Objekte zur Verfügung gestellt werden können.

Vertexoperationen

Dieser Schritt fasst die zuvor erstellten Vertices zu Primitives zusammen. Einige Vertices werden mit 4×4 – Gleitkommamatrizen transformiert (Kap. 5.1.4), also verschoben, rotiert oder skaliert. Die so gewonnenen räumlichen Daten werden dann auf das Darstellungsfeld projiziert.

Wenn weitere Funktionen wie z.B. die Nutzung von Texturen oder Lichtquellen vorgesehen sind, werden auch die dazu nötigen Berechnungen hier ausgeführt.

Zusammensetzen der Primitives

Ein wichtiger Teil in diesem Schritt ist das *Clipping*. Hier werden alle Geometrien, die nicht dargestellt werden, abgeschnitten. Ebenso werden hier Berechnungen durchgeführt, um einem Objekt den richtigen räumlichen Eindruck in der Tiefe zu verleihen.

Aus diesem Schritt werden die vollständigen geometrischen Primitives zurückgegeben. Sie wurden bereits transformiert und beschnitten, besitzen Angaben zur Farbe und Ausbreitung in der Tiefe und eventuell auch Koordinaten für Texturen und hinreichende Anweisungen für die Rasterung.

Pixeloperationen

Zunächst werden hier die Pixel aus ihrem ursprünglichen Speicherplatz geholt und in ein verarbeitbares Format überführt. Anschließend werden die Daten zerteilt, verzerrt und in eine *Pixel-Map*, ein adäquates Abbild mit passender Pixelauflösung, geschrieben. Das Ergebnis wird aufgespannt und in den Texturen-Speicher geschrieben oder an die Rasterung weitergegeben.

Daten die auf umgekehrtem Weg aus dem Framebuffer geladen werden, sind bereits bearbeitet und können nach passender Typenumwandlung an das System zurückgegeben werden.

Zusammensetzen der Texturen

Bei der Verwendung mehrerer Texturen ist es sinnvoll, sie in Texturobjekte zu fassen, um einfach Zugriff auf sie zu haben. Alle dazu nötigen Operationen und weitere Bearbeitungen der Texturen werden in diesem Schritt vollzogen.

Rasterung

Bei der Rasterung werden geometrische Daten und Pixel-Daten zusammengefasst und in quadratische Fragmente konvertiert. Ein Fragment steht später für einen Pixel im Framebuffer, und das gesamte Pixel-Raster ergibt das Bild.

Fragmentoperationen

Bevor die Daten endgültig im Framebuffer abgelegt werden, können verschiedenste Operationen die Fragmente verändern oder sogar aussondern. Beispielsweise werden hier die Texel einer Textur aus dem Texturenspeicher generiert und auf ein Fragment abgebildet. Berechnung von Elementen wie Nebel oder der Test, ob ein Objekt in der Tiefe sichtbar ist (*Depth-Test*), werden hier angewandt. Maskierungen mit Binärbildern oder Farbmischungen finden ebenfalls statt. Sollte einer dieser Schritte fehlschlagen, kann das jeweilige Fragment nicht weiter verarbeitet werden und wird von OpenGL ignoriert. Die endgültig fertigen Fragmente werden in den entsprechenden Buffer geschrieben und entsprechen jeweils einem Pixel im Ausgabebild.

5.1.2. Zweidimensionales Beispiel

Auch wenn OpenGL alle Objekte in einem dreidimensionalen Raum ablegt, ist es möglich, eine zweidimensionale Objektdarstellung zu realisieren. Dies wird beispielsweise zur zweidimensionalen Bildanzeige oder zur Ausgabe von Text benötigt.

Der nachfolgende Quelltext 5.1 zeigt einen übersichtlichen OpenGL-Code, der ein weißes Quadrat auf schwarzem Grund erzeugt. Die Ausgabe ist in Abbildung 5.2 zu sehen.

```

1 #include <GL/gl.h>
2
3 int main()
4 {
5     openAWindow();
6
7     glClearColor(0.0, 0.0, 0.0, 0.0);
8     glClear(GL_COLOR_BUFFER_BIT);
9     glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
10    glColor4f(1.0f, 1.0f, 1.0f, 0.0f);
11    glBegin(GL_QUADS);
12        glVertex3f(-0.5f, 0.5f, 0.0f);
13        glVertex3f( 0.5f, 0.5f, 0.0f);
14        glVertex3f( 0.5f,-0.5f, 0.0f);
15        glVertex3f(-0.5f,-0.5f, 0.0f);
16    glEnd();
17    glFlush();
18 }
```

Quellcode 5.1: Zweidimensionales OpenGL-Beispiel

Die erste Zeile von `main()` ruft die Funktion `openAWindow()` auf. Sie soll ein Fenster auf dem Bildschirm erzeugen, in dem später die Ausgabe von OpenGL angezeigt werden kann. Im Anschluss wird mit `glClearColor(r, g, b, a)` bestimmt, mit welcher Farbe die Ausgabe beim Aufruf von `glClear(GL_COLOR_BUFFER_BIT)` geleert werden soll. Diese entspricht der späteren Hintergrundfarbe. Die Angabe der Farben erfolgt in der Reiheinfolge rot `r`, grün `g`, blau `b`, alpha `a`, wobei jeder Wert im Intervall $[0.0, 1.0]$ liegt und alpha die Transparenz wiederspiegelt. Da in diesem Beispiel die räumliche Tiefe einer 3D-Szene nicht benötigt wird, erzeugt `glOrtho(l, r, b, t, sn, sf)` einen orthogonalen zweidimensionalen

Rendermodus (Kap. 5.1.5). Hier werden die Eckkoordinaten für die Szenerie in der Reihenfolge links `l`, rechts `r`, unten `b`, oben `t`, hinten `sn`, vorne `sf` festgelegt. In diesen Dimensionen können nur Vertices angelegt werden, die anschließend gerendert werden. Da ein weißes Quadrat erstellt werden soll, muss zunächst äquivalent zu `glClearColor()`, die Farbe der Vertices mit `glColor4f(r, g, b, a)` festgelegt werden. Der Suffix `4f` steht hierbei für den Datentyp (Tab. 5.1) und die Anzahl der Parameter, die an die Funktion übergeben werden müssen. Das zu erzeugende Quadrat ist ein Primitiv des Typ `GL_QUAD` und muss durch vier Vertices definiert werden. Innerhalb der Aufrufe `glBegin(GL_Quad)` und `glEnd()` werden mit `glVertex3f(x, y, z)` die vier Eckpunkte des Quadrats in der Reihenfolge unten links, unten rechts, oben links, oben rechts erstellt. Die Argumente der Funktion geben die jeweilige Position innerhalb des Darstellungsfelds an, definiert durch `glOrtho()`. Der Befehl `glFlush()` erzwingt schließlich die Ausführung aller anstehenden OpenGL-Befehle.

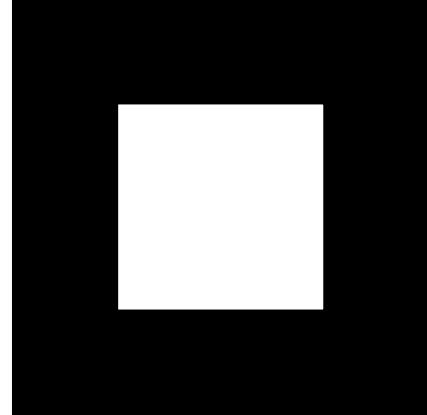


Abbildung 5.2.: Ausgabe zu Quellcode 5.1

Suffix	Datentyp	Definition in C	Definition in OpenGL
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned long	GLuint, GLenum, GLbitfield

Tabelle 5.1.: Suffixe und Datentypen in OpenGL

5.1.3. Dreidimensionales Beispiel

Im Gegensatz zum vorherigen Beispiel soll hier eine 3D-Szene realisiert werden. Zusätzlich soll eine einfache Animation implementiert sein.

Räumliche Objekte lassen sich, wie in nebenstehender Abbildung 5.3 dargestellt, in einem dreidimensionalen Koordinatensystem anordnen und durch Matrixoperationen bewegen und rotieren. Die Position und der Blickwinkel des "Betrachters" (Viewport) lassen sich in gleicher Weise verändern. Ist der konkrete Viewport ein punktförmiges Objekt und besitzt eine Richtungsangabe, die das Gesichtsfeld des Betrachters repräsentiert, wird er an der Position (0.0, 0.0, 1.0) vorgegeben. OpenGL lässt sich leicht durch das Einbinden aufbauender Bibliotheken erweitern. So beispielsweise mit dem *OpenGL Utility Toolkit*, kurz *GLUT*, das elementare Ein- und Ausgabeoperationen plattformunabhängig zur Verfügung stellt. Mit GLUT soll in diesem Beispiel das Anzeigefenster realisiert werden.

Im nachfolgenden Quellcode soll ein vorgefertigter, oranger Quader `glutSolidCube()` stark um die z-Achse und schwächer um die y-Achse rotieren. Der Körper soll somit taumeln. Die Animation soll in einem GLUT-Fenster dargestellt werden.

```

1 #include <GL/gl.h>
2 #include <GL/glut.h>
3
4 GLfloat angle = 0.0f;
5
6 void renderScene(void)
7 {
8     glEnable(GL_DEPTH_TEST);
9     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
10    glPushMatrix();
11        glRotatef(angle, 0.0f, 0.5f, 1.0f);

```

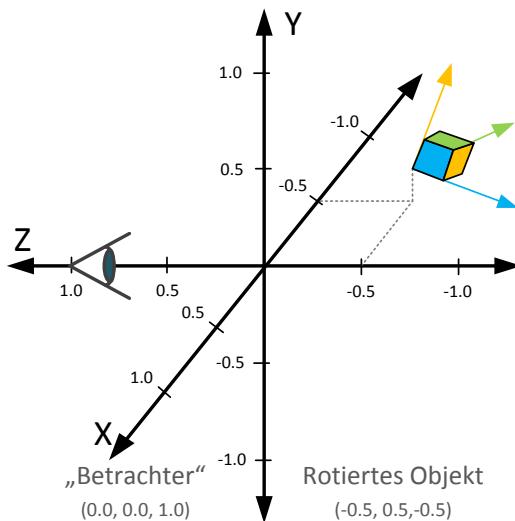


Abbildung 5.3.: 3D-Koordinatensystem

```

12         glColor4f(1.0f, 0.5f, 0.0f, 0.0f);
13         glutSolidCube(0.8);
14     glPopMatrix();
15     angle++;
16 }

18 void keys(unsigned char key, int x, int y)
{
20     if (key == 27) // 27 entspricht der 'ESC'-Taste
21         exit(0);
22 }

24 int main( int argc, char **argv )
{
25     glutInit(&argc, argv);
26     glutInitWindowSize(800, 800);
27     glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH);
28     glutCreateWindow("3D-Beispiel");
29     glutDisplayFunc(renderScene);
30     glutIdleFunc(renderScene);
31     glutKeyboardFunc(keys);
32     glutMainLoop();
33 }

```

Quellcode 5.2: Dreidimensionales OpenGL-Beispiel mit GLUT

Durch den Einsatz von GLUT zur Darstellung der gerenderten Szene muss der Programmablauf abgeändert werden.

In `main()` finden sich jetzt nur noch Funktionen der GLUT-Bibliothek, angefangen mit `glutInit()`, die die API initialisiert. Die Fenstergröße wird mit `glutInitWindowSize(x, y)` und der Anzeigemodus mit `glutInitDisplayMode()` festgelegt. `GLUT_RGBA` gibt an, dass wir ein Fenster mit Farbdarstellung wünschen und `GLUT_DEPTH` aktiviert den Tiefen-Buffer zur räumlichen Objektdarstellung. Der Fenstername `3D-Beispiel` wird mit `glutCreateWindow("3D-Beispiel")` angegeben. GLUT erfordert, dass alle kontinuierlich auszuführenden Befehle in weitere Funktionen ausgelagert werden, die dann von Callback-Funktionen⁶ aufgerufen werden. So wird die eigentliche, zu rendernde Szene in die Funktion `renderScene()` ausgelagert, und `glutDisplayFunc()` definiert die Funktion als Renderkontext. Die Funktion `keys()` wird beim Betätigen der Tastatur aufgerufen und überprüft ob die Taste `ESC`, zum Beenden

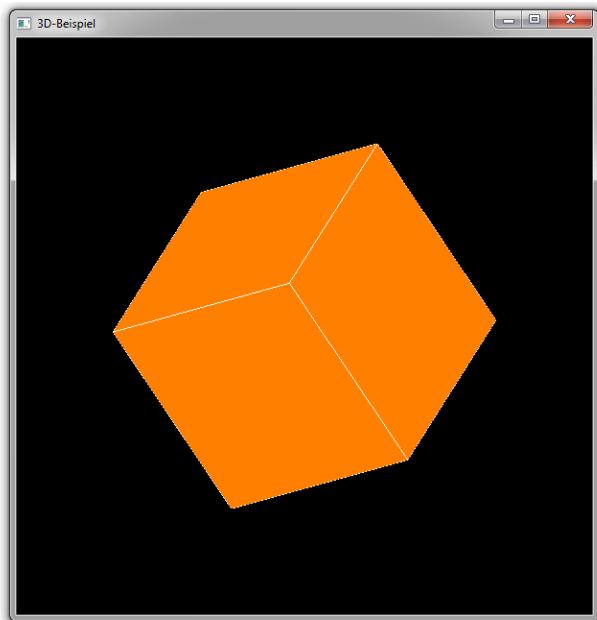


Abbildung 5.4.: Screenshot zu Quellcode 5.2

⁶Rückruffunktionen sind Funktionen, denen eine andere Funktion als Parameter übergeben wird und von dieser unter gewissen Bedingungen aufgerufen wird.

des Programms, aufgerufen wurde. Der Aufruf `glutMainLoop()` führt zum Betreten der Ereignisschleife. Diese Schleife wird bis zum Programmende nicht mehr verlassen und führt alle registrierten Callback-Funktionen nach einem gegebenen Muster aus.

In der Render-Funktion, `renderScene()`, wird zunächst der `DEPTH_TEST`, also der Tiefen-Buffer, aktiviert. Er ermöglicht, dass jedem Fragment im Framebuffer ein Tiefenwert zugewiesen werden kann. So kann bestimmt werden, ob das zu zeichnende Objekt in einer sichtbaren Distanz liegt. Der Aufruf von `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` leert diesmal Ausgabe sowie Tiefen-Buffer. Durch `glPushMatrix()` werden Matrizenoperationen auf die bis zum Aufruf von `glPopMatrix()` folgenden Elemente möglich, ohne außerhalb liegende Objekte zu beeinflussen (Kap. 5.1.4). Somit kann der Quader rotieren, ohne ein anderes Objekt wie beispielsweise das Sichtfeld zu beeinflussen. Die Rotation des Objekts wird durch `glRotatef(angle, x, y, z)` ausgeführt. Das Objekt wird um den angegebene Winkel `angle` um den mit `x, y, z` beschriebenen Vektor gedreht (Kap. 5.1.4). Die Objektfarbe wird wie gehabt mit `glColor4f()` festgelegt, und die Hilfsfunktion `glutSolidCube()` stellt die Primitives für den Quader zur Verfügung. Der Quader ließe sich auch durch sechs `GL_QUADS`-Primitives, wie in Beispiel 5.1 genutzt, erzeugen. Die Anweisung `angle++` sorgt für eine Drehung des Objekts um 1° bei jedem neu gerenderten Bild. Der Aufruf `glFlush()` wird, dank GLUT, nicht weiter benötigt. Ein Screenshot des Programms ist in Abbildung 5.4 zu sehen.

5.1.4. Matrizen

Die räumlichen Informationen jedes Elements im dreidimensionalen Raum von OpenGL (Abb. 5.3), sei es ein Vertex, die Ecke einer Textur oder die Position des Betrachters, lassen sich durch den Vektor

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \\ h \end{pmatrix} \quad (5.1)$$

beschreiben, wobei x, y und z die Positionskoordinaten sind und h üblicherweise den Wert 1 besitzt (Kap. 5.3).

Soll nun ein Primitive, bestehend aus einer Menge von Vektoren, räumlich verändert, also bewegt, gedreht, skaliert oder verzerrt werden, ist dies durch die Multiplikation mit einer 4×4 – Matrix M möglich.

$$M = \begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix} \quad (5.2)$$

Geht man davon aus, dass jedes Element ein eigenes Koordinatensystem besitzt, kann man die Position und Form dieses Systems mit einer Matrixmultiplikation verändern.

$$\vec{v}' = M \cdot \vec{v} \quad (5.3)$$

OpenGL kennt drei Arten von Matrizen, die mit der Funktion `glMatrixMode()` aktiviert werden können:

- **GL_MODELVIEW** - die Objektmatrix

Sie legt die Größe und Position eines Primitives fest; Objekte können mit dieser Matrix manipuliert werden.

- **GL_PROJECTION** - die Projektionsmatrix

Sie beeinflusst die Perspektive, also die Position und den Blickwinkel des Betrachters.

- **GL_TEXTURE** - die Texturmatrix

Diese Matrix wird auf die Texturkoordinaten angewandt und ermöglicht somit das Bewegen von Texturen auf Oberflächen.

Jede dieser Matrizen enthält nach Programmstart die Identitätsmatrix

$$M_{\text{Identität}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (5.4)$$

die jederzeit mit `glLoadIdentity()` wieder geladen werden kann.

Eine beliebige Matrix kann mit `glLoadMatrix()` eingeladen werden. Hierzu muss der Funktion der Pointer auf eine 4×4 – Matrix in Form eines Gleitkomma-Arrays übergeben werden.

Zur Multiplikation der vorhandenen Matrix mit einer benutzerdefinierten Matrix dient der Aufruf `glMultMatrix()`.

Um nicht für jede räumliche Veränderung eines Primitives eine eigene Matrix erstellen zu müssen, bietet OpenGL drei Funktion an, die dies für den Programmierer übernehmen. Eine Verschiebung (Translation) eines Elements aus dem momentanen Koordinatenursprung in den Punkt (x, y, z) lässt sich mit dem Funktionsaufruf `glTranslate(x, y, z)` erledigen. Dabei wird die vorhandene Matrix M mit einer Translationsmatrix M_T multipliziert und ergibt die neue Matrix M' :

$$M' = M \cdot M_T = M \cdot \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.5)$$

Analog zur Translation gibt es eine Funktion zur Rotation von Vertices. Die Funktion `glRotate(alpha, x, y, z)` benötigt einen Winkel `alpha`, um den gedreht werden soll und einen Vektor, beschrieben durch die Argumente `x`, `y` und `z`, der als Rotationsachse dient. Die Funktion nimmt eine Multiplikation der Rotationsmatrix M_R mit der vorhandenen

Matrix M vor:

$$M' = M \cdot M_R = M \cdot \begin{pmatrix} x^2(1 - c) + c & xy(1 - c) - zs & xz(1 - c) + ys & 0 \\ yx(1 - c) + zs & y^2(1 - c) + c & yz(1 - c) - xs & 0 \\ xz(1 - c) - ys & yz(1 - c) + xs & z^2(1 - c) + c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.6)$$

mit $c = \cos(\alpha)$ und $s = \sin(\alpha)$

Die dritte vorhandene Funktion `glScale(x, y, z)` erlaubt das unabhängige Skalieren und Spiegeln jeder Achse. Wird für x , y und z jeweils der Wert 1 angegeben, so bleibt das Objekt in der Ursprungsgröße. Soll eine Spiegelung an der x -Achse erfolgen, muss x den Wert -1 erhalten. Werte > 1 wirken sich in einer Vergrößerung aus, Werte < 1 bewirken eine Verkleinerung, negative Werte äußern sich in einer Spiegelung. Der Wert 0 sollte vermieden werden, da die Matrix sonst singulär wird. Auch hier ergibt die Multiplikation der vorhandenen Matrix M mit der Skalierungsmatrix M_S die neue, skalierte Matrix M' :

$$M' = M \cdot M_S = M \cdot \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{für } x, y, z \neq 0 \quad (5.7)$$

Ist die Matrix `GL_MODELVIEW` oder `GL_PROJECTION` aktiviert, bewirkt der Aufruf einer Matrixfunktion die Manipulation aller nachfolgend gerenderten Objekte. Da dies nicht immer erwünscht ist, existieren die Funktionen `glPushMatrix()` und `glPopMatrix()`.

Jeder Matrizenmodus besitzt einen eigenen Stapel von Matrizen, *Stack* genannt, der eine bestimmte Anzahl Matrizen aufnehmen kann. Die aktuelle Matrix ist jeweils die, die oben auf dem Stapel liegt. `glPushMatrix()` "drückt" die aktuelle Matrix um eine Stelle nach unten in den Stapel und hinterlässt an der vorherigen Position eine Kopie. Die obere Matrix ist dann identisch mit der darunter. `glPopMatrix()` ersetzt die oben liegende Matrix mit der darunter liegenden.

Soll beispielsweise nur ein bestimmtes Primitive bewegt werden, wird erst `glPushMatrix()` aufgerufen, die Matrixmultiplikation durchgeführt, die Vertices erstellt und anschließend durch `glPopMatrix()` die ursprüngliche Matrix wieder geladen.

Die Kombination mehrerer Matrizenoperationen ist nicht kommutativ, kann also nicht beliebig vertauscht werden. So ist es relevant, ob ein Primitive erst bewegt und dann rotiert oder erst rotiert und anschließend bewegt wird. Die resultierende Position und Ausrichtung des Primitives ist in beiden Fällen völlig verschieden.

$$M \cdot M_T \cdot M_R \neq M \cdot M_R \cdot M_T \quad (5.8)$$

Abbildung 5.5 zeigt die Kombination mehrerer Matrizenoperationen auf quadratischen 2D-Primitives.

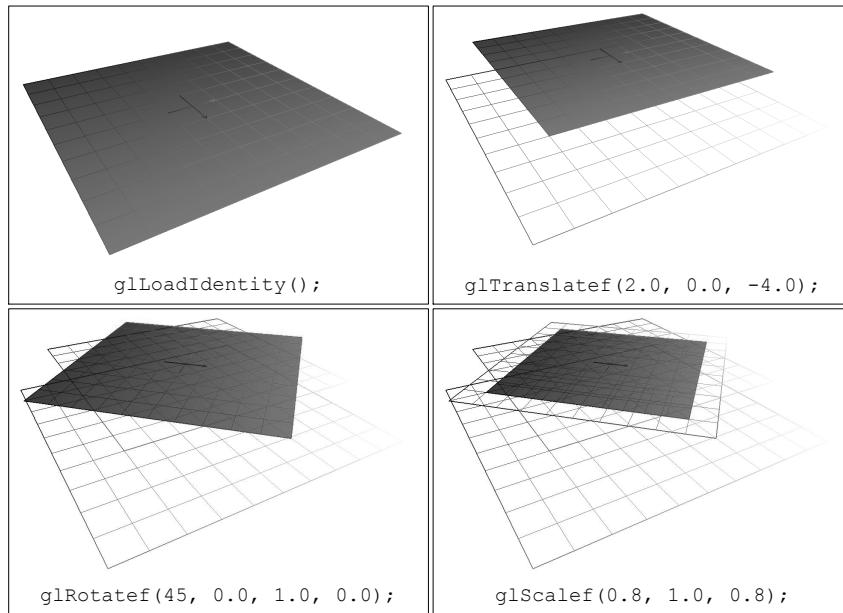


Abbildung 5.5.: Kombinierte Matrixoperationen: Ausgehend von der Identitätsmatrix wird das Primitive zuerst verschoben, anschließend rotiert und als letztes skaliert. Der rote Pfeil zeigt die x -Achse und der blaue Pfeil die z -Achse an, die y -Achse steht senkrecht darauf.

5.1.5. Projektion und Perspektive

Als Frustum⁷ bezeichnet man in der 3D-Grafik einen Körper, dessen Inhalt auf eine Bildebene projiziert wird. Ein perspektivisches Frustum, wie es für die dreidimensionale Darstellung benötigt wird, hat die Form einer Pyramide und beschränkt den sichtbaren Bereich der Szene. Die Pyramidenspitze entspricht der Position des "Betrachters".

Für die Projektionsmatrix `GL_PROJECTION` existieren zwei Funktionen, die eine gezielte Anpassung des Frustums ermöglichen. Das Frustum wird dabei begrenzt durch fünf Schnittflächen, sogenannte *Clippingplanes*, und eine Projektionsfläche, die der späteren Bildebene entspricht (Abb. 5.6). Clippingplanes begrenzen das Sichtfeld. Alle Objekte oder auch Teilobjekte hinter einem Clippingplane, also ausserhalb des Frustum, werden nicht gerendert.

`glFrustum(l, r, b, t, zn, zf)` dient zur Definition des Frustums mittels sechs Parametern. Zunächst lässt sich die Größe der Projektionsfläche durch die Position der linken (`l`) und rechten (`r`) Kante auf der x -Achse und die untere (`b`) und obere (`t`) Kante auf der y -Achse festlegen, wobei der Mittelpunkt der Projektionsfläche immer auf der z -Achse liegt. Die Position der Projektionsfläche lässt sich mit dem Parameter `zn` und das Ende des Frustums mit `zf` definieren. Die Matrix des neuen Frustums M_{Frustum} wird dabei mit

⁷Frustum steht in der Geometrie für den Stumpf eines Kegels oder einer Pyramide.

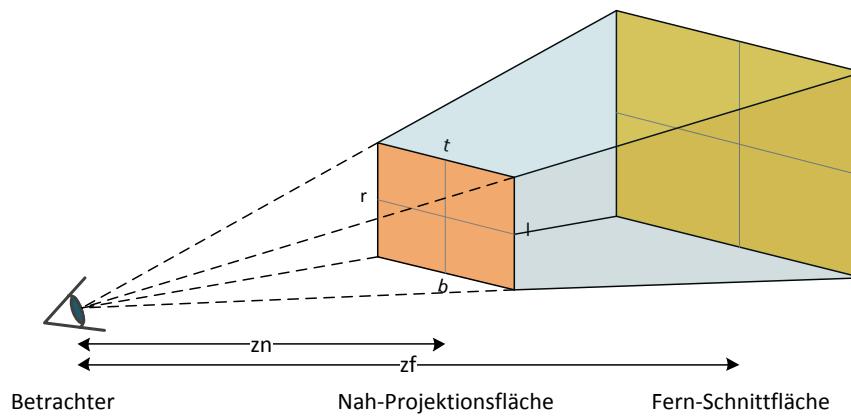


Abbildung 5.6.: Perspektivisches Frustum zur dreidimensionalen Darstellung

der aktuellen Matrix multipliziert.

$$M_{\text{Frustum}} = \begin{pmatrix} \frac{2z_n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2z_n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(z_f+z_n)}{z_f-z_n} & \frac{-2z_f z_n}{z_f-z_n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (5.9)$$

Soll OpenGL zur Darstellung von zweidimensionalen Flächen genutzt werden, dargestellt im Quellcodebeispiel 5.1, sollte das Frustum die Form eines Quaders wie in Abbildung 5.7 haben. Hierfür existiert die Funktion `glOrtho(l, r, b, t, zn, zf)`. Sie lässt

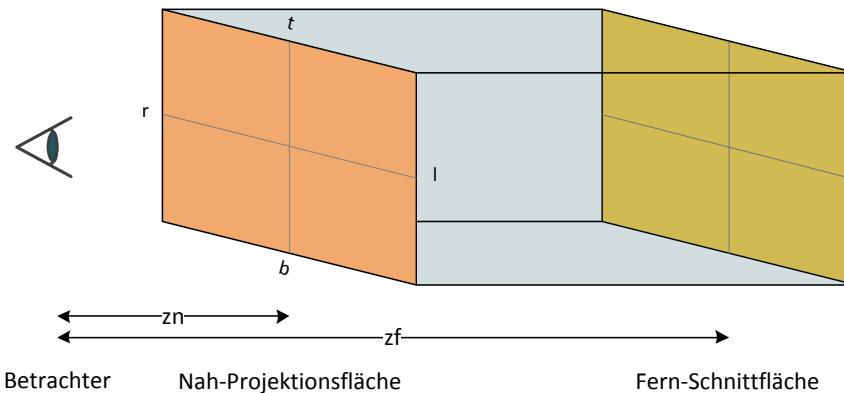


Abbildung 5.7.: Orthogonales Frustum zur Darstellung zweidimensionaler Objekte

sich anwenden wie `glFrustum()`, nur dass die Projektionsfläche und die gegenüberliegende Clippingplane gleich groß sind. Den Unterschied im entstehenden Bild soll Abbildung 5.8 verdeutlichen.

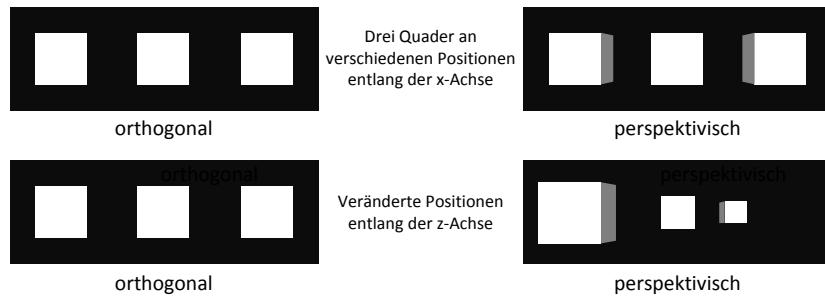


Abbildung 5.8.: Vergleich zwischen orthogonaler und perspektivischer Darstellung von dreidimensionalen Primitiven

Auch hier wird die neue Matrix $M_{\text{Orthogonal}}$ für ein orthogonales Frustum mit der vorherigen Projektionsmatrix multipliziert.

$$M_{\text{Orthogonal}} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{z_f-z_n} & -\frac{z_f+z_n}{z_f-z_n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.10)$$

5.1.6. Texturen und 2D-Bildanzeige

Die Software φ Scope soll die Phaseninformationen, die auf der GPU mit dem ITG-Algorithmus berechnet wurden, als zweidimensionales Bild anzeigen. Um keinen weiteren Kopiervorgang von Host zu Device vornehmen zu müssen, können die Daten von der Grafikkarte mit OpenGL direkt als Textur weiterverarbeitet werden (Kap. 5.1.8). Diese Textur lässt sich dann direkt aus dem globalen Speicher der Grafikkarte auf einem Primitive darstellen. Im orthogonalen Darstellungsmodus reicht somit ein quadratisches Primitive, wie es in Kapitel 5.1.2 gezeigt wurde, das der Größe der Projektionsfläche entspricht und mit einer Textur zur einfachen Bildanzeige versehen wird.

Zur Positionierung einer Textur verwendet man keine globalen Koordinaten, sondern definiert für jede Ecke des betreffenden Objekts eine Texturkoordinate, sogenannte *UV-Koordinaten*. OpenGL berechnet dann den Teil der Textur, der auf das Objekt projiziert werden soll. Dieser Vorgang wird *Texture Mapping* genannt. Die Breite und Höhe jeder Textur wird in dem Intervall $[0, 1]$ definiert, dabei entspricht die Koordinate $(0, 0)$ der oberen linken Ecke, $(0, 1)$ der unteren linken und $(1, 1)$ der oberen rechten Ecke (Quellcode 5.3). Durch Verwendung von UV-Koordinaten muss die Größe der Textur nicht bekannt sein und das Wiederholen (Koordinaten > 1 oder < -1) oder Spiegeln (Koordinaten $[0, -1]$) von Texturen ist intuitiv möglich. Eine UV-Koordinate lässt sich mit `glTexCoord()` festlegen. Jedem nachfolgenden Vertex wird diese Texturkoordinate zugewiesen, bis sie wieder geändert wird (Kap. 5.1.1).

```

1  glBegin(GL_QUADS);
2   //links oben
3   glTexCoord2f(0, 0); glVertex2f(-1, 1);
4   //links unten
5   glTexCoord2f(0, 1); glVertex2f(-1, -1);
6   //rechts unten
7   glTexCoord2f(1, 1); glVertex2f(1, -1);
8   //rechts oben
9   glTexCoord2f(1, 0); glVertex2f(1, 1);
10 glEnd();

```

Quellcode 5.3: UV-Koordinaten auf einem Quadrat

Um die Nutzung von Texturen zu ermöglichen, muss `glEnable(GL_TEXTURE_2D)` vor der Definition der UV-Koordinaten aufgerufen werden. `glDisable(GL_TEXTURE_2D)` deaktiviert die Texturierung wieder.

Jeder Textur muss zunächst mit `glGenTexture(n, &texID)` ein eigener Index zugewiesen werden und mit `glBindTexture(GL_TEXTURE_2D, texID)` wird festgelegt, dass alle nachfolgenden Änderungen an `GL_TEXTURE_2D` für die Textur mit dem Index `texID` gelten. Die eigentliche Übergabe der zuvor eingeladenen Bildinformationen an OpenGL erledigt die Funktion `glTexImage2D()`. Quellcode 5.4 zeigt einen Renderkontext, der eine aus dem Hauptspeicher geladene Textur auf der gesamten Projektionsfläche darstellt. Das Beispiel baut auf Quellcode 5.1 und Quellcode 5.2 auf.

```

1  GLuint texID;
2  GLubyte *bitmap;
3  GLsizei width = ..., height = ...;
4
5  void renderScene(void)
6  {
7      glClearColor(0.0, 0.0, 0.0, 0.0);
8      glClear(GL_COLOR_BUFFER_BIT);
9      glOrtho(-1.0, 0.0, -1.0, 1.0, -1.0, 1.0);
10     glEnable(GL_TEXTURE_2D);
11     glGenTextures(1, &texID);
12     glBindTexture(GL_TEXTURE_2D, texID);
13     loadBitmap(bitmap);
14     glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
15                 bitmap);
16     glBegin(GL_QUADS);
17     //links oben
18     glTexCoord2f(0, 0); glVertex2f(-1, 1);
19     //links unten
20     glTexCoord2f(0, 1); glVertex2f(-1, -1);
21     //rechts unten
22     glTexCoord2f(1, 1); glVertex2f(1, -1);
23     //rechts oben
24     glTexCoord2f(1, 0); glVertex2f(1, 1);
25  glEnd();
26  glDisable(GL_TEXTURE_2D);
27  glFlush();
28  glDeleteTexture(1, &texID);
}

```

Quellcode 5.4: Ausgabe einer zweidimensionalen Textur

5.1.7. 3D-Plot einer zweidimensionalen Funktion

Die Software zu dieser Arbeit soll es ermöglichen, die berechnete Phasenverteilung nicht nur als zweidimensionales Graustufenbild zu betrachten, sondern auch durch einen dreidimensionalen Plot⁸ der Funktion einen räumlichen Eindruck der Phasenverteilung vermitteln. Die zweidimensionale Phasenverteilung wurde zur Darstellung auf einen Da-

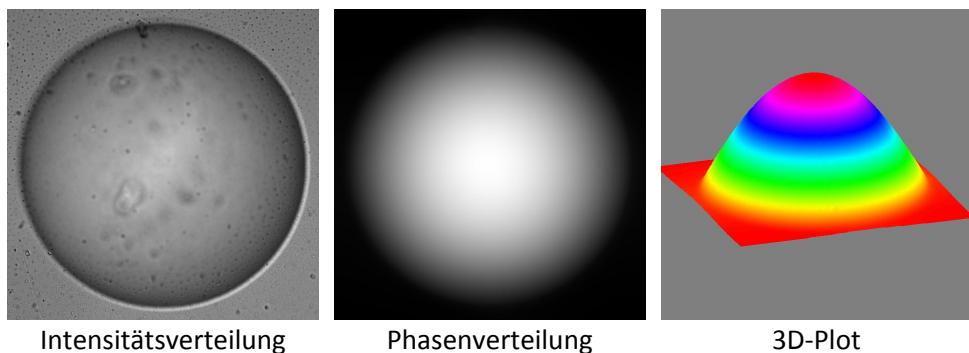


Abbildung 5.9.: Intensität-, Phasenverteilung und 3D-Plot einer Microlinse

tenbereich von 8 Bit⁹ Graustufen normiert. Die benötigte dritte Dimension lässt sich aus diesen Graustufen ablesen. Der Wert 0 entspricht dem am tiefsten liegenden Punkt und der Wert 255 dem höchstgelegenen des Phasebildes. So lässt sich für jedes Pixel der Phasenverteilung eine überschlägige Höhe abtragen. Dieses Höhenprofil gibt einen räumlichen Eindruck der Phasenverzögerung wieder.

Erstellt man nun für jedes Pixel ein Punkt-Primitive, wobei die Lage auf der y -Achse durch den Graustufenwert angegeben wird, erhält man eine dreidimensionale Verteilung aller Pixel der Phasenverteilung (Abb.5.10). Verbindet man nun die einzelnen Pixel durch Linien-Primitives ergibt sich ein Drahtgittermodell, das bereits die Oberfläche erahnen lässt. Durch Einfügen eines quadratischen Flächen-Primitives zwischen jeweils vier Pixeln ergibt sich ein Oberflächenmodell der rekonstruierten Phase.

Die Anwendung einer Color-Lookup-Tabelle kann die Visualisierung des Modells weiter verbessern. Eine solche Tabelle enthält für jeden Graustufenwert einen definierten RGB-Farbwert, der den jeweiligen Vertices zugewiesen wird.

Eine weitere anschauliche Methode ist die Darstellung von Niveaulinien, sogenannte *ISO-Lines*. Sie ergeben ein Höhenprofil durch die Bezeichnung benachbarter Punkte gleicher Höhe in einer Linienschar.

Um die Performance der 3D-Darstellung zu verbessern, empfiehlt es sich nicht jeden Pixel im 3D-Modell zu berücksichtigen. Es sollte ein Pixel-Offset n eingeführt werden. Dieser legt fest, dass nur jeder n -te Pixel betrachtet werden soll.

⁸Als Plot bezeichnet man die Darstellung einer diskreten Menge von Punkten einer Funktion in einem ein- oder mehrdimensionalen Koordinatensystem.

⁹8 Bit entsprechen 256 Grautönen, bzw. dem Intervall [0, 255].

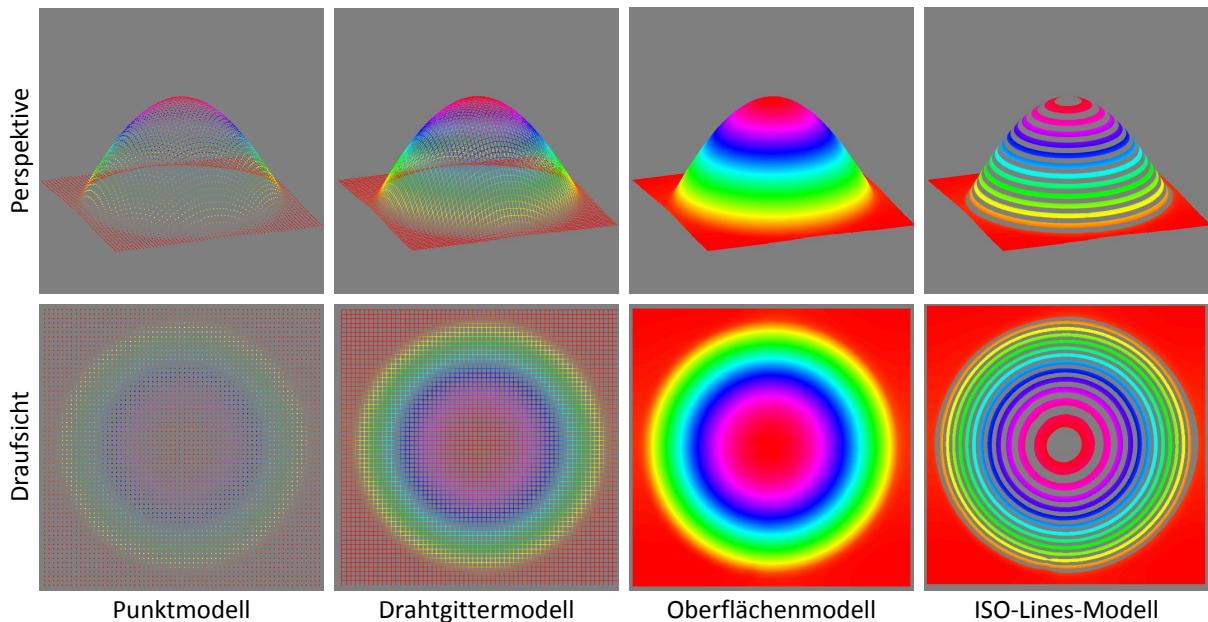


Abbildung 5.10.: Verschiedene 3D-Plots einer Microlinse mit einem Pixel-Offset von 16 bei einer Auflösung der Phasenverteilung von 986px × 986px

5.1.8. Interoperabilität zwischen OpenGL und CUDA

Manche Ressourcen von OpenGL können im Speicherbereich von CUDA abgebildet werden. Dies ermöglicht CUDA den Zugriff auf Daten von OpenGL, und umgekehrt kann OpenGL auf Daten von CUDA zugreifen.

Für die Zusammenarbeit zwischen CUDA und OpenGL muss zunächst das zu nutzende CUDA-Device mit `cudaGLSetGLDevice()` zugewiesen werden.

Mit der Funktion `cudaGraphicsGLRegisterBuffer()` lässt sich ein Bufferobjekt¹⁰ des Typs `struct cudaGraphicsResource` erstellen. In CUDA entspricht dies einem Device-Pointer und ist entsprechend in Kernen oder auch mit `cudaMemcpy()` zugänglich. Der Aufruf `cudaGraphicsMapResources()` bildet die erstellten Bufferobjekte beliebig oft in den Speicherbereich von CUDA ab. Mit `cudaGraphicsUnregisterResource()` kann dieses Abbild wieder aufgehoben werden, um es mit OpenGL verarbeiten zu können.

5.2. Kamerasteuerung

Die Bildaufnahme der nötigen Intensitätsverteilungen in drei Fokuslagen erfolgt mit Kameras vom Typ *PIKE F-145B* des Herstellers *Allied Vision Technologies* (Datenblatt A.1). Sie ermöglichen die Aufnahme von Bildern mit 14 Bit bzw. 16384 Graustufen, gekapselt in einem 16 Bit-Datentyp bei einer maximalen Auflösung von 1388px × 1038px. Jedes Pixel ist $6,45\mu m \times 6,45\mu m$ groß. Der Anschluss der Kameras erfolgt über *FireWire*

¹⁰ Objekt zur temporären Datenspeicherung, hier Bilddaten.

800 bzw. IEEE 1394b-Anschlüsse¹¹ und bietet eine Bildwiederholrate von bis zu 30fps bei voller Auflösung.

Der Hersteller liefert eine eigene API, die *UniAPI* [47]. Sie ermöglicht die vollständige Steuerung aller Kamerafunktionen durch Funktionsaufrufe, denen jeweils ein konkret zugewiesener Kameraindex übergeben werden muss.

5.2.1. Übertragungsbandbreite

Der IEEE 1394b Standard garantiert eine Übertragungsbandbreite von 800 Mbit/s bei maximal 16 angeschlossenen Verbrauchern pro Bus¹² (Datenblätter A.8 und A.9). Somit würde sich bei drei Kameras an einem Bus eine maximal verfügbare Bandbreite von 266 Mbit/s pro Kamera ergeben. Ein Bild der Dimensionen $1024\text{px} \times 1024\text{px} : 16\text{Bit}$ ist 16 Mbit groß. Bei angestrebten 24 Bildern pro Sekunde ergibt sich ein notwendiger Datendurchsatz von 384 Mbit/s für eine Kamera bzw. 1152 Mbit/s für drei Kameras. Tatsächlich ist mit diesem Aufbau gerade einmal eine Bildwiederholrate von 10fps möglich.

Um eine Wiederholrate von mindestens 24 Bildern pro Sekunde zu erreichen, ist es nötig, jeder Kamera einen eigenen FireWire-Bus zur Verfügung zu stellen. Dies lässt sich beispielsweise über eine eigene FireWire-Steckkarte im System erreichen (Datenblatt A.2).

5.2.2. Synchronisation

Der ITG-Algorithmus (Kap. 2.2.4) benötigt drei zeitgleich aufgenommene Intensitätsaufnahmen als Eingangsgrößen. Die Bildaufnahme muss also kontrolliert, sprich synchronisiert, geschehen.

Das Auslösen, *Triggern*, der angeschlossenen Kameras funktioniert bei der UniAPI, indem ein Triggersignal über einen bestimmten Bus geschickt wird. Alle angeschlossenen Kameras lösen aus und stellen daraufhin die aufgenommene Intensitätsverteilung zum Abruf bereit. Es existiert ebenfalls ein sogenannter *Multibus-Trigger*, der das Senden eines Triggersignals synchron auf mehreren Bus-Systemen ermöglicht.

So ist es möglich die drei Bilder in einem Intervall von mehreren μs aufzunehmen. Sie liegen dann in den Speichern der Kameras bereit und können parallel über den FireWire-Bus abgerufen werden.

5.3. Projektive Transformation

Geometrische Bildoperationen, wie sie bereits für den dreidimensionalen Raum von OpenGL in Kapitel 5.1.4 thematisiert wurden, ermöglichen die *Verformung* eines zwei-

¹¹Der Standard IEEE 1394b legt eine Übertragungsbandbreite von 800 Mbit/s (100 MByte/s) fest. Angeschlossene Geräte können über das FireWire-Kabel mit 8 bis 33V DC und 1,5A bei max. 48W versorgt werden. Es können bis zu 16 Geräte an einem FireWire-Bus angeschlossen werden.

¹²System zur Datenübertragung zwischen mehreren Teilnehmern über einen Übertragungsweg

dimensionalen Bildes mittels Matrixoperationen, d.h. die Pixel des Bildes werden neu angeordnet. In dieser Arbeit wird die projektive Transformation verwendet, um die drei detektierten Bilder aufeinander abzulegen. So werden die beiden defokussierten Aufnahmen auf die fokussierte abgebildet. Dazu ist es lediglich notwendig für jede Kamera einmal vier charakteristische Punkte zu markieren (Kap. 6).

Grundsätzlich erzeugt eine geometrische Bildoperation aus einem Bild I ein neues Bild I' ,

$$I(x, y) \rightarrow I'(x', y') \quad (5.11)$$

wobei nicht die Werte sondern die Koordinaten der Bildelemente verändert werden. Zweidimensionale Koordinatentransformationen in Form von Matrizenoperationen ermöglichen die einfachen Bildtransformationen Translation, Rotation, Skalierung und Transvektion [14]:

Translation (Verschiebung) um die Koordinaten (x_0, y_0) :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \quad (5.12)$$

Rotation (Drehung) um den Koordinatenursprung mit Winkel α :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad (5.13)$$

Skalierung (Streckung oder Stauchung) um den Faktor s_x bzw. s_y :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad (5.14)$$

Transvektion (Scherung) in x - und y -Richtung um die Faktoren t_x und t_y :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & t_x \\ t_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad (5.15)$$

Die projektive Transformation ist die Zusammenfassung der einfachen Koordinatentransformationen. Hierzu können die einzelnen Koordinatentransformationen in einer 3×3 - Transformationsmatrix M_T kombiniert werden. Sie besitzt folgende Form:

$$\begin{pmatrix} \hat{x} \\ \hat{y} \\ h' \end{pmatrix} = \begin{pmatrix} h'x' \\ h'y' \\ h' \end{pmatrix} = M_T \cdot \begin{pmatrix} x \\ y \\ h \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ h \end{pmatrix} \quad (5.16)$$

Da die Translation, die durch eine Vektoraddition beschrieben wurde, nicht direkt durch eine Matrixoperation beschrieben werden kann, zeigt sich die Einführung von *homogenen Koordinaten* als sinnvoll [14]. Dazu wird jeder Vektor um eine zusätzliche Komponente h bzw. h' erweitert. h ist dabei ein beliebiger Wert aus der Menge der natürlichen Zahlen \mathbb{N} und wird vorzugsweise mit 1 gewählt. In der Matrix wird die homogene Koordinate

h' durch a_{31} , a_{32} und a_{33} repräsentiert, wobei $a_{33} = h$ ist.

Die Translation entspricht den Paramtern a_{13} und a_{23} . Rotation, Skalierung und Transvektion spiegeln sich gemeinsam in a_{11} , a_{12} , a_{21} und a_{22} wieder.

Die projektive Transformation, auch *Vierpunkt-Abbildung* genannt, ermöglicht die allgemeine Verformung von Vierecken. So kann beispielsweise ein orthogonales Bild auf eine beliebig geformte viereckige Ebene projiziert werden.

Die projektive Transformationsmatrix lässt sich mit einer *zweistufigen Abbildung* über das Einheitsquadrat E berechnen. Im ersten Schritt wird E auf ein beliebig geformtes Zielviereck T abgebildet (Abb. 5.11). Zur Ermittlung der Matrixparameter wer-

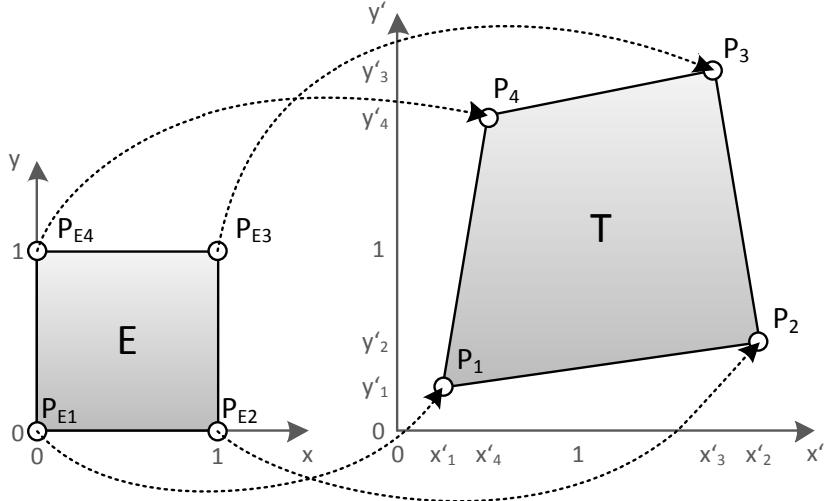


Abbildung 5.11.: Projektive Abbildung des Einheitsquadrat E auf ein beliebiges Quadrat T

den die Koordinatenpaare $P_{E1\dots E4}$ beziehungsweise $P_{T1\dots T4}$ der jeweils vier Eckpunkte des Einheitsquadrats und des Zielvierecks benötigt. Das Einheitsquadrat besteht aus den Punkten $P_{E1} = (x_1, y_1) = (0, 0)$, $P_{E2} = (x_2, y_1) = (1, 0)$, $P_{E3} = (x_2, y_2) = (1, 1)$ und $P_{E4} = (x_1, y_2) = (0, 1)$. Das Zielviereck lässt sich durch Eckkoordinaten der Form $P_1 = (x'_1, y'_1)$, $P_2 = (x'_2, y'_2)$ usw. beschreiben.

Die Parameter der Matrix errechnen sich aus diesen Koordinaten wie folgt [14]:

$$\begin{aligned} a_{31} &= \frac{(x'_1 - x'_2 + x'_3 - x'_4) \cdot (y'_4 - y'_3) - (y'_1 - y'_2 + y'_3 - y'_4) \cdot (x'_4 - x'_3)}{(x'_2 - x'_3) \cdot (y'_4 - y'_3) - (x'_4 - x'_3) \cdot (y'_2 - y'_3)} \\ a_{32} &= \frac{(y'_1 - y'_2 + y'_3 - y'_4) \cdot (x'_2 - x'_3) - (x'_1 - x'_2 + x'_3 - x'_4) \cdot (y'_2 - y'_3)}{(x'_2 - x'_3) \cdot (y'_4 - y'_3) - (x'_4 - x'_3) \cdot (y'_2 - y'_3)} \quad (5.17) \\ a_{11} &= x'_2 - x'_1 + a_{31}x'_2 \quad a_{12} = x'_4 - x'_1 + a_{32}x'_4 \quad a_{13} = x'_1 \\ a_{21} &= y'_2 - y'_1 + a_{31}y'_2 \quad a_{22} = y'_4 - y'_1 + a_{32}y'_4 \quad a_{23} = y'_1 \end{aligned}$$

Die aus diesen Parametern gewonnene Matrix M_T (Gl. 5.16) ermöglicht die Transformation von E nach T . Die Inverse M_T^{-1} ermöglicht die gegenläufige Transformation von Viereck T auf das Einheitsquadrat E .

Eine projektive Transformation von einem gegebenen Viereck T auf ein beliebig geformtes Zielviereck T' ermöglicht die bereits erwähnte zweistufige Abbildung. Zunächst

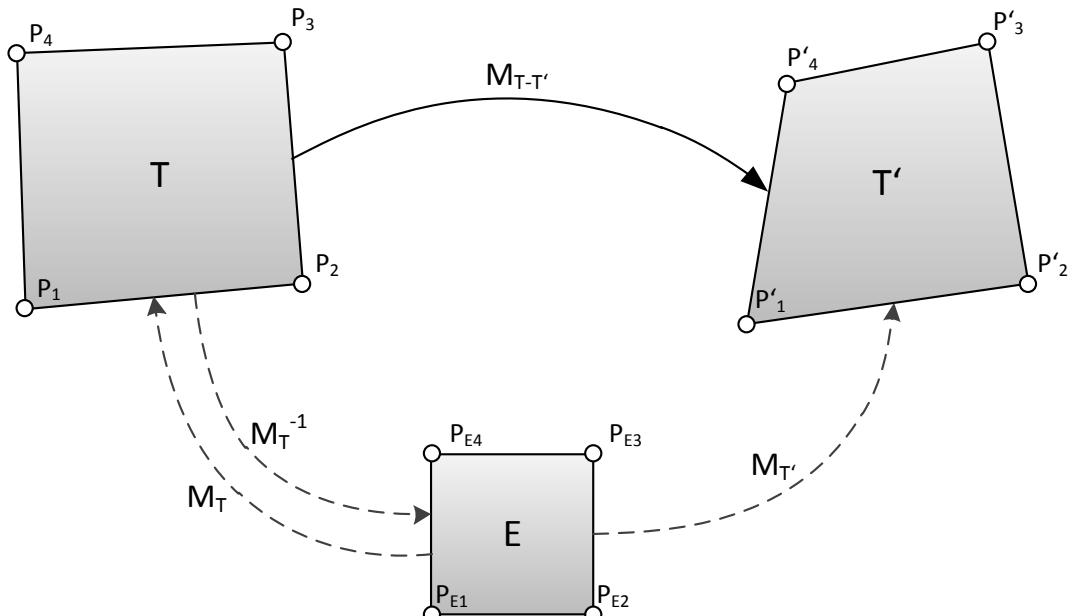


Abbildung 5.12.: Projektive Abbildung eines beliebigen Vierecks T auf eine Referenzfläche T'

müssen hierzu die Koordinaten P_n der Eingangsfläche T in die Koordinaten P_{En} des Einheitsquadrats E mit der inversen Matrix M_T^{-1} überführt werden. Diese Koordinaten lassen sich wiederum in die Koordinaten P'_n der Zielfläche T' mit der Matrix $M_{T'}$ transformieren (Abb. 5.12). Durch Kombination der beiden Matrizen M_T^{-1} und $M_{T'}$ ergibt sich eine Matrix $M_{T-T'}$ (Gl. 5.18). Sie ermöglicht die direkte Koordinatentransformation von einer beliebigen viereckigen Ausgangsfläche auf eine Zielfläche, ohne den Umweg über die Einheitsmatrix zu gehen.

$$P'_n = M_{T'} \cdot P_{En} = M_{T'} \cdot M_T^{-1} \cdot P_n = M_{T-T'} \cdot P_n \quad (5.18)$$

Bei der projektiven Abbildung werden die zuvor diskreten Positionen der Pixel in kontinuierliche Positionen überführt, die außerhalb des diskreten Rasters liegen können (Abb. 5.13). Eine ganz ähnliche Problematik erzeugt die Skalierung, bei der der Abstand der Pixel verändert wird, ohne das zugrundeliegende Raster anzupassen. Um Bildfehler durch diese Problematik möglichst gering zu halten, muss ein passender Wert für jede Position des diskreten Zielrasters gefunden werden. Der Wert jedes neuen Pixels basiert dann auf den Werten der umliegenden Bildpunkte ausserhalb des Rasters.

Das einfachste Interpolationsverfahren ist die *Nearest-Neighbor-Interpolation* oder *Pixelwiederholung*. Hierbei wird jedem Pixel der Wert des nächstgelegene Pixels aus dem Originalraster zugewiesen.

$$I'(x_0, y_0) = I(u_0, v_0), \text{ mit } \begin{cases} u_0 = \text{Round}(x_0 - 0,5) = \lfloor x_0 \rfloor \\ v_0 = \text{Round}(y_0 - 0,5) = \lfloor y_0 \rfloor \end{cases} \quad (5.19)$$

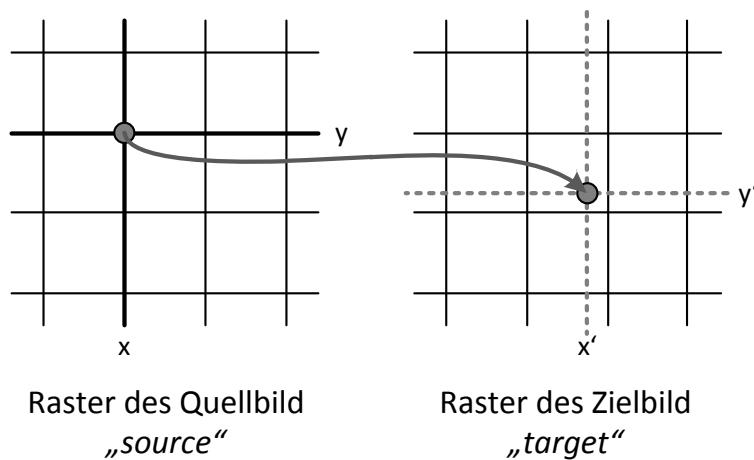


Abbildung 5.13.: Quelle-Ziel Abbildung: Transformierte Positionen des Quellrasters fallen auf Punkte zwischen den neuen Rasterpostionen

Bei dieser Methode können Bildinformationen wegfallen, und es kann zu starken Alias-Effekten¹³ kommen.

Genauer arbeitet da die *bilineare Interpolation*, wie sie auch in der Software zu dieser Arbeit verwendet wurde. Hierbei wird der Wert des Ausgabepixels aus den vier umliegenden Pixeln des Quellbild interpoliert. Bei der Variation der Punktdichte, beispielsweise bei Skalierungen, werden so auch markante Wertetendenzen berücksichtigt. Die Arbeitsweise der bilinearen Interpolation stellt die Abbildung 5.14 dar.

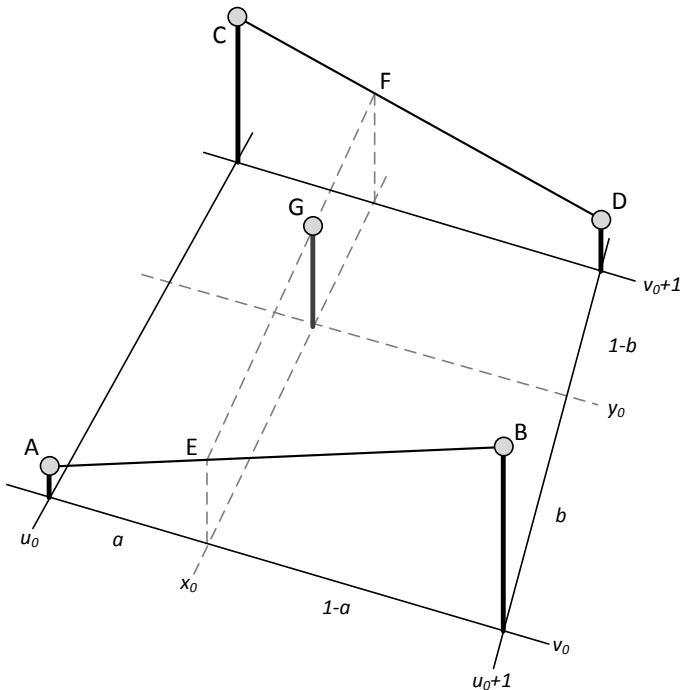


Abbildung 5.14.: Veranschaulichung zur bilinearen Interpolation

¹³ Alias-Effekte entstehen durch die Nichtbeachtung des Abtasttheorems, also zu geringer Abtastfrequenz, beim digitalen Abtasten von Signalen.

Dabei werden zunächst die zur neuen Koordinate (x_0, y_0) nächstliegenden Bildpunkte A, B, C und D im Quellbild mit

$$\begin{aligned} A &= I(u_0, v_0) & B &= I(u_0 + 1, v_0) \\ C &= I(u_0, v_0 + 1) & D &= I(u_0 + 1, v_0 + 1) \end{aligned} \quad \text{mit} \quad \begin{cases} u_0 = \lfloor x_0 \rfloor \\ v_0 = \lfloor y_0 \rfloor \end{cases} \quad (5.20)$$

ermittelt. Nun werden die vier Bildpunkte interpoliert, d.h. der dazwischen liegende Wert wird errechnet. Dazu werden die beiden Zwischenwerte E und F aus dem Abstand $a = (x_0 - u_0)$ errechnet.

$$\begin{aligned} E &= A + (x_0 - u_0) \cdot (B - A) = A + a \cdot (B - A) \\ F &= C + (x_0 - u_0) \cdot (D - C) = C + a \cdot (D - C) \end{aligned} \quad (5.21)$$

Der endgültige Interpolationswert $G = I'(x_0, y_0)$ ergibt sich unter Zuhilfenahme des Abstands $b = (y_0 - v_0)$.

$$\begin{aligned} I'(x_0, y_0) &= G = E + (y_0 - v_0) \cdot (F - E) = E + b \cdot (F - E) \\ &= (a - 1)(b - 1)A + a(1 - b)B + (1 - a)bC + abD \end{aligned} \quad (5.22)$$

Weitere bekannte Interpolationsverfahren sind die *bikubische Interpolation* und die *Lanczos-Interpolation*. Sie liefern zwar bessere Ergebnisse, sind allerdings bedeutend rechenintensiver. Eine Einführung in die weiteren Verfahren ist in Quelle [14, S. 377ff] zu finden.

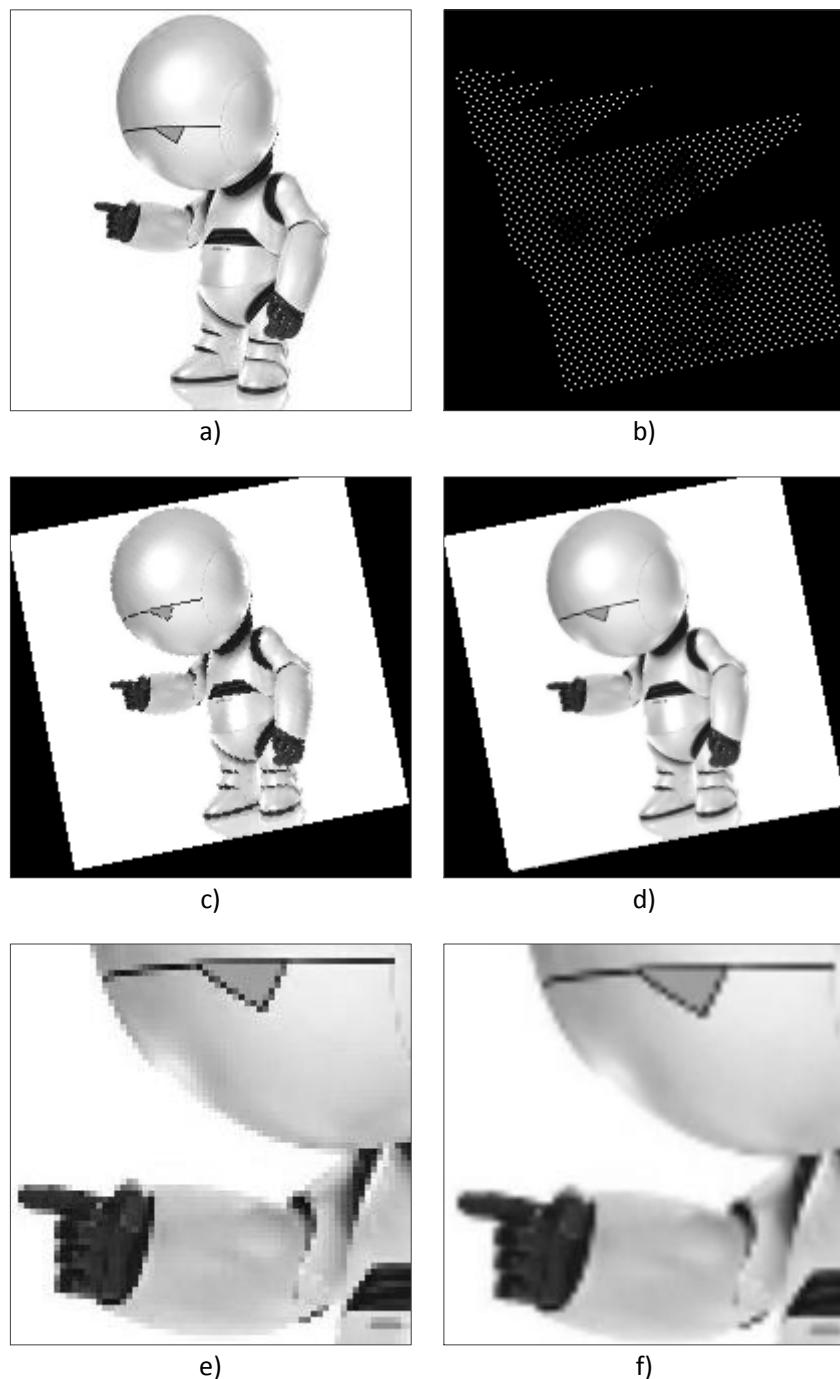


Abbildung 5.15.: Das Ausgangsbild a) wurde zunächst lediglich projektiv transformiert und nicht interpoliert b). Abbildung c) zeigt die identische Transformation mit Nearest-Neighbor-Interpolation und d) die bilineare Interpolation. Bei extremen Vergrößerungen zeigt die Nearest-Neighbor-Interpolation e) deutlich mehr Alias-Effekte als die bilineare Interpolation f). (Bildquelle: <http://alturl.com/2ee6e>)

5.4. Shading Correction

Auch unter den besten Abbildungs- und Beleuchtungsbedingungen ist die Ausleuchtung im Mikroskopaufbau inhomogen. Hinzu kommen Verunreinigungen und Reflexe an optischen Bauteilen, Vignettierungen, sowie Staubablagerungen im Abbildungsstrahlengang. Die *Shading Correction* (deut. Korrektur der Bildabschattung) wird zur Kompensation der Inhomogenität im Bildfeld B angewandt [48]. Das korrigierte Bild B' wird direkt auf der Kamera berechnet.

Zur Korrektur wird jedes Pixel $B(x, y)$ mit einem Faktor $S(x, y)$ des Intervalls $[1, 2]$ multipliziert.

$$B' = \sum \sum B(x, y) \cdot S(x, y) \quad (5.23)$$

Das *Shadingbild* S weist jedem Pixel $B(x, y)$ einen Faktor $S(x, y)$ zu, der jeden Pixel auf den Wert des hellsten Pixels anheben soll. Zur Generierung des Shadingbilds werden bis zu 256 Bilder detektiert. Während der Aufnahmen darf der hellste Pixel einen Grauwert von 255 nicht überschreiten, und es sollte nur der Hintergrund sichtbar sein.

Die Shading Correction wird immer vor dem Binning¹⁴ oder Beschneiden des detektierten Bilds ausgeführt. Somit sollten die Aufnahmen für das Shadingbild bei voller Auflösung gemacht werden.



Abbildung 5.16.: Shading Correction im Hellfeld-Mikroskop

¹⁴Unter *Binning* versteht man das Zusammenfassen von benachbarten Pixeln im CCD-Sensor zu Pixelblöcken.

5.5. Programmablauf

An dieser Stelle soll der Programmablauf von φ Scope aufgezeigt werden. Abbildung 5.17 stellt das Programm als Ablaufdiagramm dar. Der Programmablauf, wie der Benutzer ihn erfährt, ist im nachfolgenden Kapitel 6 dargestellt.

Nachdem die grafische Benutzeroberfläche¹⁵ gestartet ist und die genutzten Bibliotheken initialisiert wurden, wird ein Host-Thread erstellt, in dem alle Aktivitäten der Kameras mittels *UniAPI* parallel zum restlichen Programm, das im Hauptthread ausgeführt wird, ablaufen. Sollten keine Kameras angeschlossen sein, werden Bilder von der Festplatte (HDD - engl. hard disk drive) eingelesen. Sind Kameras vorhanden, wird ein Triggersignal gesendet, das die Kameras zum Auslösen bringt. Nach erfolgreicher Bildaufnahme und Shading Correction (Kap. 5.4) wird wiederum ein Triggersignal an den Host gesendet und die Bildinformationen können über den FireWire-Bus in den Speicher des Host geladen werden. Nach Abschluss des Transfers startet der Thread erneut.

Sobald die Bildinformationen zur Verfügung stehen, werden sie vom Hauptthread in den globalen Speicher des Devices kopiert. Anschließend wird die projektive Transformation (Kap. 5.3), sowie alle nachfolgenden Arbeitsschritte in einem CUDA-Kernel angewendet. Der Algorithmus teilt sich an dieser Stelle in die drei zur Verfügung stehenden Ausgabemodi auf.

- Zur Ausgabe der *Intensitätsverteilung* wird das ausgewählte Kamerabild lediglich auf 8 Bit Graustufen normiert und mit OpenGL ausgegeben (Kap. 5.1.6).
- Die Ausgabe der *Phasenverteilung* erfordert den Durchlauf des ITG-Algorithmus (Abb. 2.5). Anschließend wird das gewonnene Phasenbild normiert und ausgegeben.
- Für das *3D-Modell* wird ebenfalls das normierte Phasenbild benötigt. Aus diesem wird dann das Höhenprofil der Probe ermittelt und im dreidimensionalen Raum dargestellt (Kap. 5.1.7).

Nach der jeweiligen Ausgabe durch OpenGL werden die nächsten detektierten Bildinformationen auf gleiche Weise verarbeitet.

¹⁵Nachfolgend auch *GUI* (engl. Graphical User Interface) genannt. Zum Zeitpunkt der Veröffentlichung wurde die Bibliothek *GLUI*, eine Bibliothek die auf OpenGL aufbaut, zum Erzeugen des GUI genutzt.

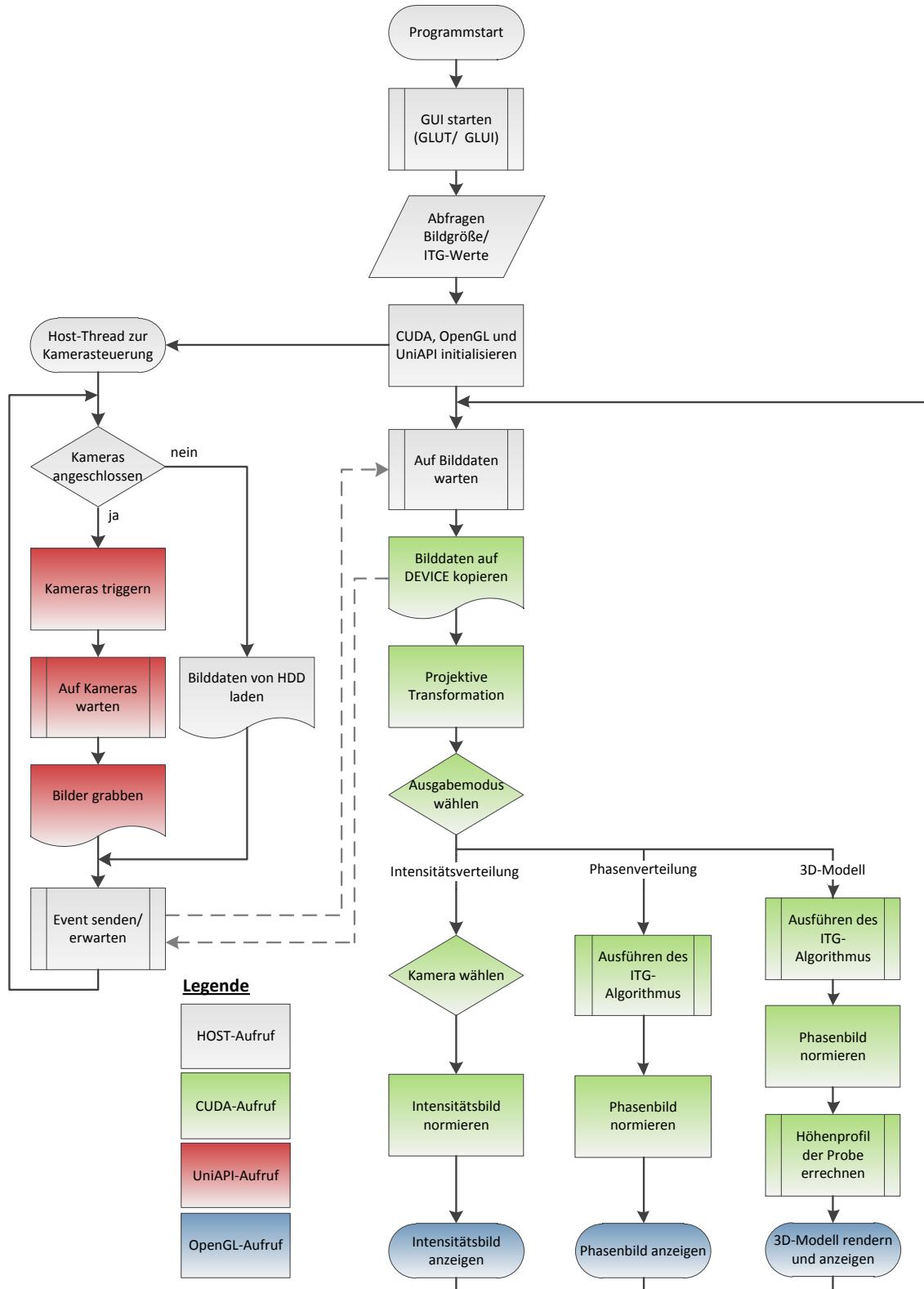


Abbildung 5.17.: Programmablaufplan

6. Messdurchführung

In diesem Kapitel soll die Anwendung der Software *φScope* erklärt werden.

Im ersten Abschnitt wird auf die nötigen Schritte zur Kalibrierung eingegangen. Anschließend wird der Leistungsumfang der Software aufgezeigt und somit das Vorgehen bei einer Messung beschrieben. Abschließend werden einige beispielhafte Messergebnisse aufgezeigt.

Unter der Internetadresse <http://www.janbeneke.de/bachelor> steht Videomaterial zu diesem Kapitel bereit.

6.1. Kalibrierung

Die Kalibrierung bereitet das Messsystem auf die nachfolgende Messung vor. Sie ist nach jeder Änderung im Abbildungsstrahlengang, z.B: Wechsel des Objektivs oder Änderung der Defokusdistanz (Gl. 3.1) zu wiederholen.

Einstellen des Mikroskops

1. Köhlersche Beleuchtung einstellen (Kap. 2.3.1)
 - a) Gewünschtes Objektiv einschwenken. Präparat auf den Objekttisch legen. Licht einschalten.
 - b) Hellfeldkondensor einbringen.
 - c) Apertur- und Leuchtfeldblende öffnen.
 - d) Kondensorkopf in den Strahlengang schwenken und zum oberen Anschlag drehen.
 - e) Objekt fokussieren.
 - f) Leuchtfeldblende schließen.
 - g) Kondensor senken bis Leuchtfeldblende abgebildet wird, anschließend zentrieren.
 - h) Leuchtfeldblende öffnen.
 - i) Okular aus dem Tubus ziehen und hintere Brennebene des Objektivs beobachten.
 - j) Aperturblende so weit schließen, dass sie gerade die Objektivbrennebene frei gibt und zentrieren.
 - k) Okular wieder einsetzen und Leuchtfeldblende so weit schließen, dass sie gerade das Bildfeld frei gibt.

2. Streuscheibe unter den Objektträger in den Beleuchtungsstrahlengang einfügen, um eine homogene Ausleuchtung zu gewährleisten.
3. Okularaufsatz abnehmen und Kamerasytem (Abb. 3.3) in Tubus einsetzen.

Einstellen des Kamerasytems

1. Berechnen der benötigten bildseitigen Defokusdistanz für das zu messende Präparat (Gl. 3.1).
2. Einstellen der Defokusdistanz durch Einbringen von Tuben und Distanzringen zwischen Anschlussgewinde der Kamera und dem Adapter am Skelettwürfel.
3. Ein geeignetes Kalibrierobjekt auf den Objekttisch legen. Mit diesem Objekt wird später die projektive Transformation kalibriert.

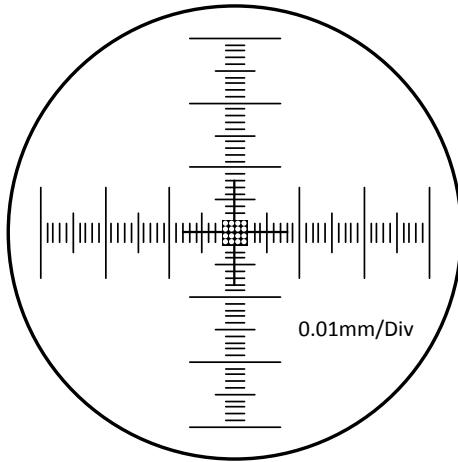


Abbildung 6.1.: Kalibrierobjekt

4. Die Software *φScope* starten.
5. Im ersten Dialog *phiScope - Settings* (Abb. 6.2) ist nun die Auflösung und Bittiefe der Bildinformationen der Kameras zu wählen. Um korrekte Messergebnisse zu erhalten, sollten auch die Angaben unter *TIE properties* entsprechend angepasst werden.
 - **wavelength:** Die genutzte Wellenlänge in *nm*. Bei Weißlicht empfiehlt sich der Wert *550nm* als mittlere Wellenlänge des Spektrums.
 - **pixel width/ height:** Pixelbreite und Pixelhöhe in *μm*. Bei der *AVT PIKE F-145* (Datenblatt A.1) beträgt Höhe wie Breite *6,45μm*.
 - **magnification:** Die Lupenvergrößerung des Mikroskops, hier *20x*.
 - **defocus distance:** Die eingestellte objektseitige Defokusdistanz in *mm*, hier *7,0mm*.
 - **transmitted light/ reflected light:** Mikroskopotyp, *transmitted light* steht für ein Durchlichtmikroskop und *reflected light* für ein Auflichtmikroskop.

Die hier vorgenommenen Einstellungen bleiben beim Neustart des Programms erhalten. Zum Fortfahren auf `calibrate` klicken.

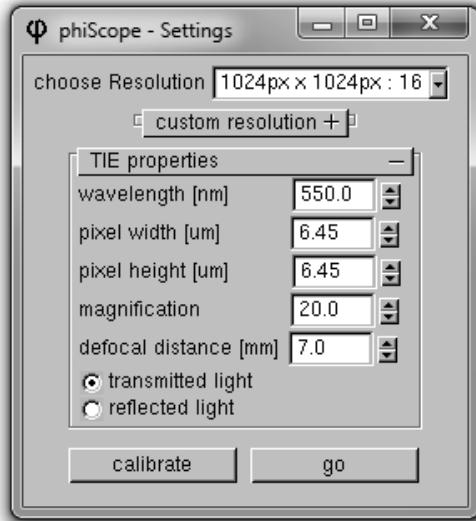


Abbildung 6.2.: Benutzeroberfläche nach Programmstart

6. Es erscheinen zwei Fenster, `phiScope - Display` dient als Ausgabefenster und das links daneben befindliche Fenster enthält alle Kontrollelemente (Abb. 6.3):

- `select camera`: Auswahl der aktiven Kamera.
- `properties`: Enthält die Kontrollelemente zur Kamerasteuerung. Es kann notwendig sein, ein Kamerabild vertikal zu spiegeln, hierzu steht die Funktion `mirror image` zur Verfügung.
- `image focus`: Hier ist jeder Kamera die zugehörige Defokus- bzw. Fokusebene zuzuweisen. Sollten nur zwei Kameras zur Verfügung stehen, muss `use two images` markiert werden.
- `compare images`: Ermöglicht den Vergleich von zwei Kamerabildern.
- `show histogram`: Zeigt ein Histogramm an.
- `show crossfade`: Zeigt ein zentriertes Fadenkreuz im Ausgabefenster an.

Für jedes Kamerabild muss nun die entsprechende Fokus- bzw. Defokusebene festgelegt werden.

Vorgehen am Beispiel eines Zweikamerasystems: Zunächst wird das Bild der aktuell ausgewählten Kamera fokussiert. Anschließend wird die andere Kamera selektiert und das entsprechende Bild fokussiert. Wurde zum Fokussieren der Objekttisch nach unten verschoben, detektiert diese Kamera die negative Defokusaufnahme. Andernfalls entspricht das Bild dieser Kamera der positiven Defokusaufnahme. Das Vorgehen lässt sich analog hierzu auch auf das Drei-Kamera-System übertragen.

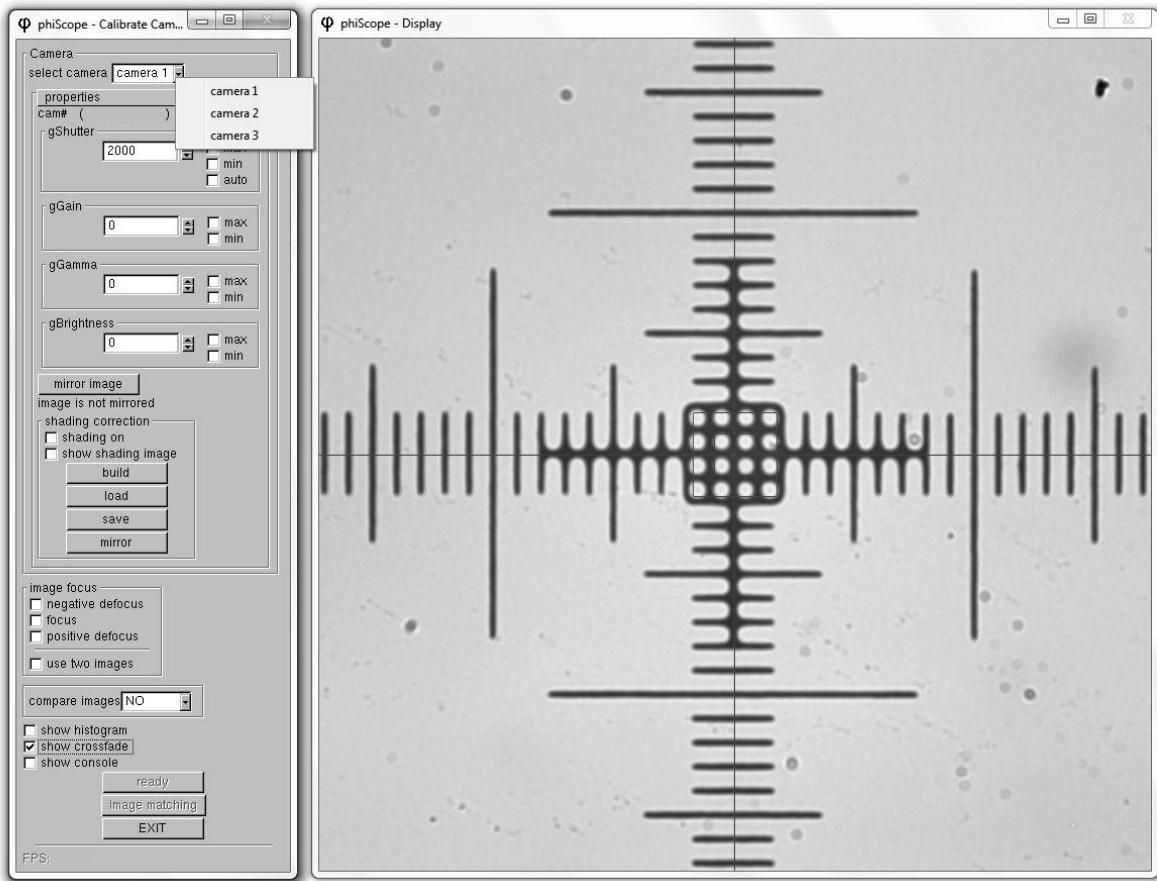


Abbildung 6.3.: Benutzeroberfläche im Kalibriermodus

Einstellen der Bildlage

1. Die Kameras sind üblicherweise gegeneinander rotiert. Unter Zuhilfenahme des Fadenkreuzes lassen sich die Kamerabilder durch einfaches Drehen der Kameras auf die gleiche Achse bringen.
2. Der Strahlteiler, der sich auf dem Prismenträger im Skelettwürfel befindet, muss so verschoben werden, dass sich die Bilder aller Kameras überlagern. Hierzu ist es sinnvoll die Funktion `compare images` zu nutzen (Abb. 6.4). Es werden jeweils zwei Kamerabilder subtrahiert und Verschiebungen sowie Rotationen werden sichtbar und können, durch Verschieben des Prismenträgers korrigiert werden.
3. Der nächste Schritt liefert die Eingangsdaten für die projektive Transformation (Kap. 5.3), die die unterschiedlichen Abbildungsmaßstäbe zwischen den Bildern aufeinander abgleichen soll. Ist eine projektive Transformation nicht erwünscht, kann sie mit einem Klick auf `go` umgangen werden.
Zur Berechnung einer Transformationsmatrix (Gl. 5.16) werden jeweils vier Koordinaten für die gleichen markanten Punkte im jeweiligen Objektbild benötigt. Hierzu eignet sich besonders ein spezielles Kalibrierobjekt, wie es in Abbildung 6.1

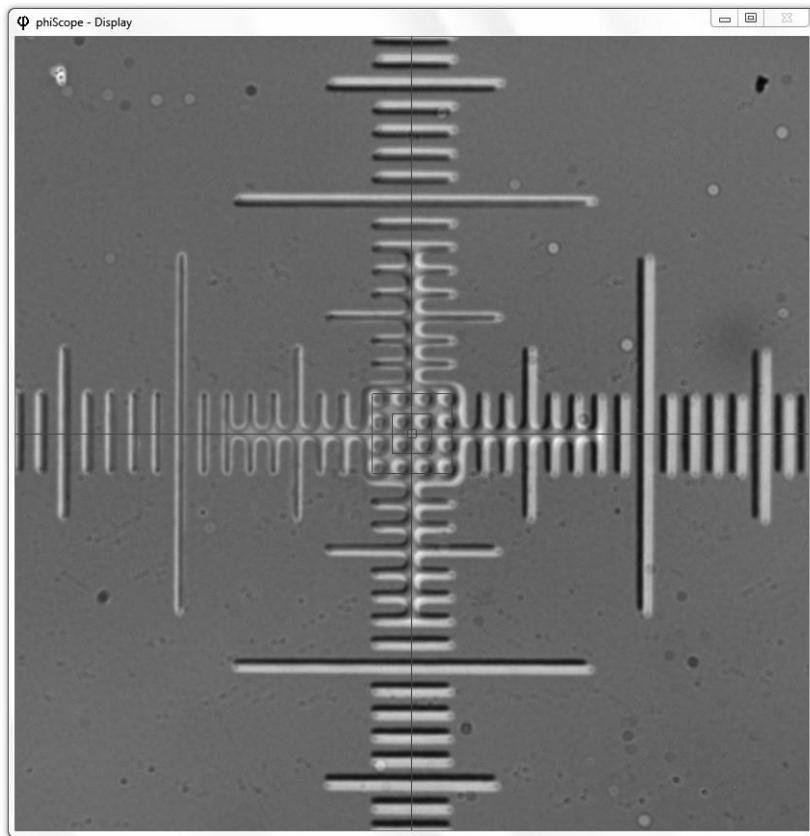


Abbildung 6.4.: Vergleich von zwei Kamerabildern

dargestellt ist.

Die Methode `image matching` zeigt nacheinander die Quadranten jedes Kamerabildes an. In jedem Quadranten muss der jeweilige markante Punkt, zum Beispiel das Ende einer Linie oder einer Ecke, durch Anklicken markiert werden (Abb. 6.5). Diese Punkte bilden dann die Koordinaten, aus denen die Transformationsmatrizen gebildet werden, die alle Bilder auf das Bild mit dem kleinsten Abbildungsmaßstab transformieren.

4. Nach Durchlauf des Vorgangs ist die Kalibrierung abgeschlossen. Alle vorgenommenen Einstellungen bleiben auch bei Neustart des Programms erhalten und werden erst bei erneutem Betreten des Kalibriermodus überschrieben.

Es wird automatisch der Messmodus gestartet.

Wird ein Durchlichtmikroskop genutzt, sollte zunächst eine *Shading Correction* (Kap. 5.4) durchgeführt werden. Hierzu muss das Objekt aus dem Strahlengang entfernt werden. Die Funktionen der Shading Correction finden sich im Menü `properties` unter `shading correction` (Abb. 6.6(a)).

- `shading on`: Aktivieren bzw. Deaktivieren der Shading Correction.
- `show shading image`: Zeigt das Weißbild an.

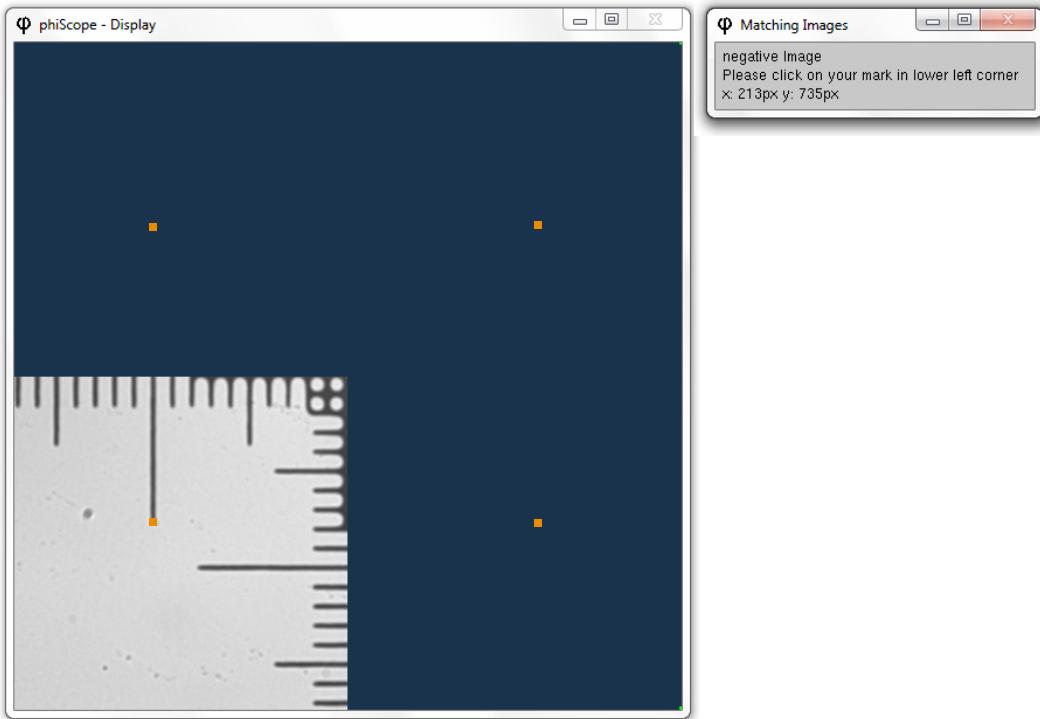


Abbildung 6.5.: Kalibrieren der projektiven Transformation

- **build:** Nimmt 256 Weißbilder auf und mittelt daraus das zur Korrektur verwendete Bild.
- **load:** Lädt ein zuvor gespeichertes Weißbild.
- **save:** Speichert das aktuelle Weißbild im Ordner `setting/ shading_correction`.
- **mirror:** Ermöglicht das vertikale Spiegeln des aktuellen Weißbilds.

6.2. Messung

Auch im Messmodus gibt es neben dem Ausgabefenster eine Benutzeroberfläche `phiScope - ControlPanel` (Abb. 6.6). Sie beinhaltet alle Steuerlemente, die für den aktuellen Modus `Mode` relevant sind:

- **display:** *Ausgabe* eines der Kamerabilder und Anpassen der Kameraeinstellungen.
- **TIE:** *Phasenrekonstruktion* mittels ITG-Algorithmus in Echtzeit.
- **3D:** *3D-Plot* der rekonstruierten Phasenverteilung in Echtzeit.

Neben der Moduswahl sind die folgenden Elemente in jedem Modus vorhanden:

- **show histogramm:** Histogramm im Ausgabefenster anzeigen.

- `show crossfade`: Zentriertes Fadenkreuz anzeigen.
- `show console`: Zusätzliche Systemhinweise und Fehlermeldungen im Konsolenfenster ausgeben.
- `save image`: Speichert den aktuellen Inhalt des Ausgabefensters im Ordner `output`. Außer im Modus `display` wird auch eine Textdatei mit Messdaten gespeichert.
- `Exit`: Beenden des Programms.

Ausgabe

Der Modus `display` (Abb. 6.6(a)) zeigt das bereits transformierte und normierte Livebild der aktivierten Kamera an. Die Auswahl der Kamera erfolgt im Feld `select camera` als `negative image`, `focused image` oder `positive image`. Sollten nur zwei Bilder detektiert werden, zeigt `focused image` das arithmetische Mittel aus `negative image` und `positive image` an.

Neben der Bildausgabe können in diesem Modus die Einstellungen der zu dem jeweiligen Bild gehörenden Kamera im Feld `camera` verändert werden:

- `gShutter`: Gibt die Verschlusszeit der Kamera und somit indirekt die Belichtungsdauer an.
- `gGain`: Verstärkungsmaß des CCD-Chips, die Anwendung führt zu Messfehlern.
- `gGamma`: Gammakorrektur, die Anwendung führt zu Messfehlern.
- `gBrightness`: Helligkeitsanhebung, die Anwendung führt zu Messfehlern.
- `mirror image`: Bild vertikal spiegeln.
- `shading correction`: Shading Correction, siehe Kapitel 6.1.

Phasenrekonstruktion

Im Modus `TIE`¹ (Abb. 6.6(b)) gibt die mit dem ITG-Algorithmus (Abb. 2.5) rekonstruierte Phasenverteilung das Objekt als Graustufenbild aus. Auch hier existiert ein spezielles Feld `properties TIE`:

- `path difference`: Gibt den Gangunterschied in der Phasenverteilung an. Die Einheit lässt sich als Bogenmaß in Radiant `rad`, als Längenmaß in `µm um` oder als Vielfaches der Wellenlänge `wavelength` wählen.
- `invert output`: Invertiert die Ausgabe.

3D-Plot

Der dreidimensionale Plot der rekonstruierten Phasenverteilung `3D` (Abb. 6.6(c)) ermöglicht die räumliche Veranschaulichung der Phasenverteilung (Kap. 5.1.7). Die Rotation des Modells erfolgt durch Mausbewegung bei gedrückter linker Maustaste im Ausgabefenster, eine Verschiebung des Modells ist bei gedrückter rechter Maustaste möglich und eine Skalierung erfolgt bei gedrückter mittlerer Maustaste. Die weitere Anpassung des Modells ermöglicht das Feld `properties 3D`:

¹ *Transport of intensity equation* kurz *ITG*, deut. Intensitäts-Transportgleichung

- **render mode:** Das Modell kann nicht nur als Oberflächenmodell `surface` sondern auch als Punktmodell `points`, Linienmodell `lines`, Drahtgittermodell `wireframe` oder als ISO-Lines-Modell `isolines` angezeigt werden (Abb. 5.10).
- **look up table:** Weist jeder Höhe einen eigenen Farbwert zu und hilft somit, die räumliche Verteilung anschaulicher zu machen.
- **grid resolution:** Ermöglicht einen Pixel-Offset von n Pixel zur Geschwindigkeitssteigerung. Es gehen Details verloren (Kap. 5.1.7).
- **height ratio:** Erlaubt das Stauchen des Modells.
- **isoline width:** Gibt die Breite einer ISO-Line bei der Nutzung des ISO-Lines-Modells an.
- **invert output:** Invertiert die Höhe des Modells.
- **show grid:** Skala im dreidimensionalen Raum an- bzw. abschalten.
- **reset view:** Vordefinierte Ansichten abrufen.
 - `default`: Perspektivische Ansicht, Modell mit 20° um y - und z -Achse rotiert.
 - `front`: Vorderansicht.
 - `side`: Seitenansicht.
 - `top`: Draufsicht.
- **lighting:** Die globale Beleuchtung des Modells wird durch einen gerichteten Spot ersetzt.
 - `light source`: Position der Lichtquelle verschieben in x -, y - und z -Richtung.
 - `light target`: Zielpunkt der gerichteten Lichtquelle verschieben in x -, y - und z -Richtung.

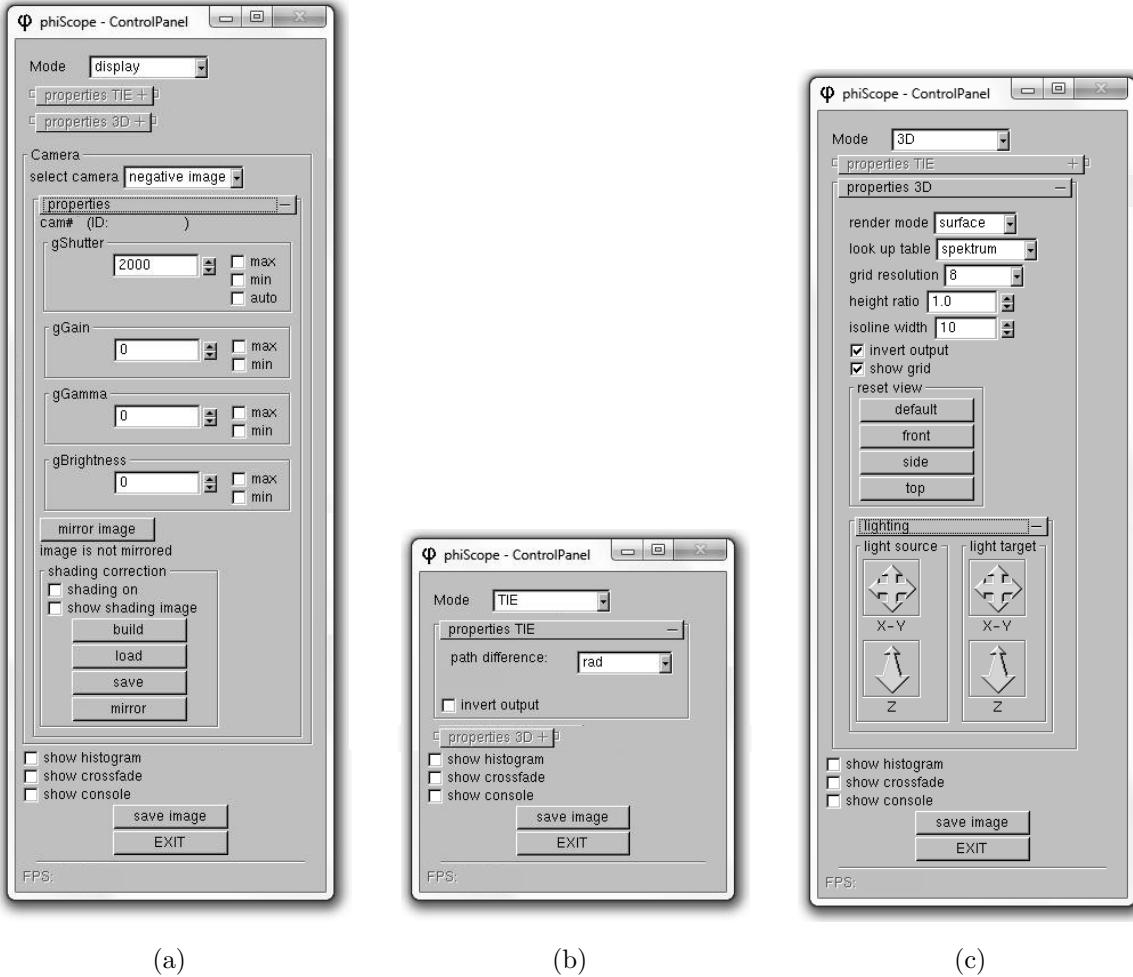


Abbildung 6.6.: Benutzeroberflächen im Messmodus

6.3. Messergebnisse

In diesem Abschnitt sollen exemplarische Messergebnisse präsentiert werden. Eine quantitative Aussage über die Messungen wurde bereits in den vorangegangenen Arbeiten [4], [3] und [5] getroffen und die Messergebnisse des ITG-Algorithmus somit validiert. Da zum Zeitpunkt der Messungen der speziell anzufertigende Strahlteilerwürfel nicht zur Verfügung stand, wurden alle Intensitätsverteilungen mit einem Zwei-Kamera-System erfasst. Das Fokusbild wird durch den Mittelwert der defokussierten Aufnahmen approximiert.

In den nachfolgenden Abbildungen 6.7 bis 6.10 gibt (a) die negativ und (c) die positiv defokussierte Intensitätsverteilung an. Die Bilder (b) zeigen den arithmetischen Mittelwert aus (a) und (c), der die Intensitätsverteilung des Fokusbilds repräsentiert. Die berechnete Phasenverteilung aus dem ITG-Algorithmus zeigt Abbildung (d). Ein dreidimensionaler Plot ist in (e) zu sehen. Alle Intensitätsverteilungen wurden mit einer Auflösung von $1024px \times 1024px$ bei 16 Bit am *Leica DMR* bzw. *Leica DMRM* aufgenommen.

6.3.1. Lichtwellenleiter

Mikroskopieverfahren:	Durchlicht/ Hellfeld
Objektiv:	Leica HC PL FLUOTAR 20 × /0,50 PH 2
Wellenlänge:	550nm
objektseitige Defokusdistanz:	20µm
bildseitige Defokusdistanz:	4mm
gemessener max. Wegunterschied:	2,73µm

Tabelle 6.1.: Messdaten zum Lichtwellenleiter Fibertech AS 3401198

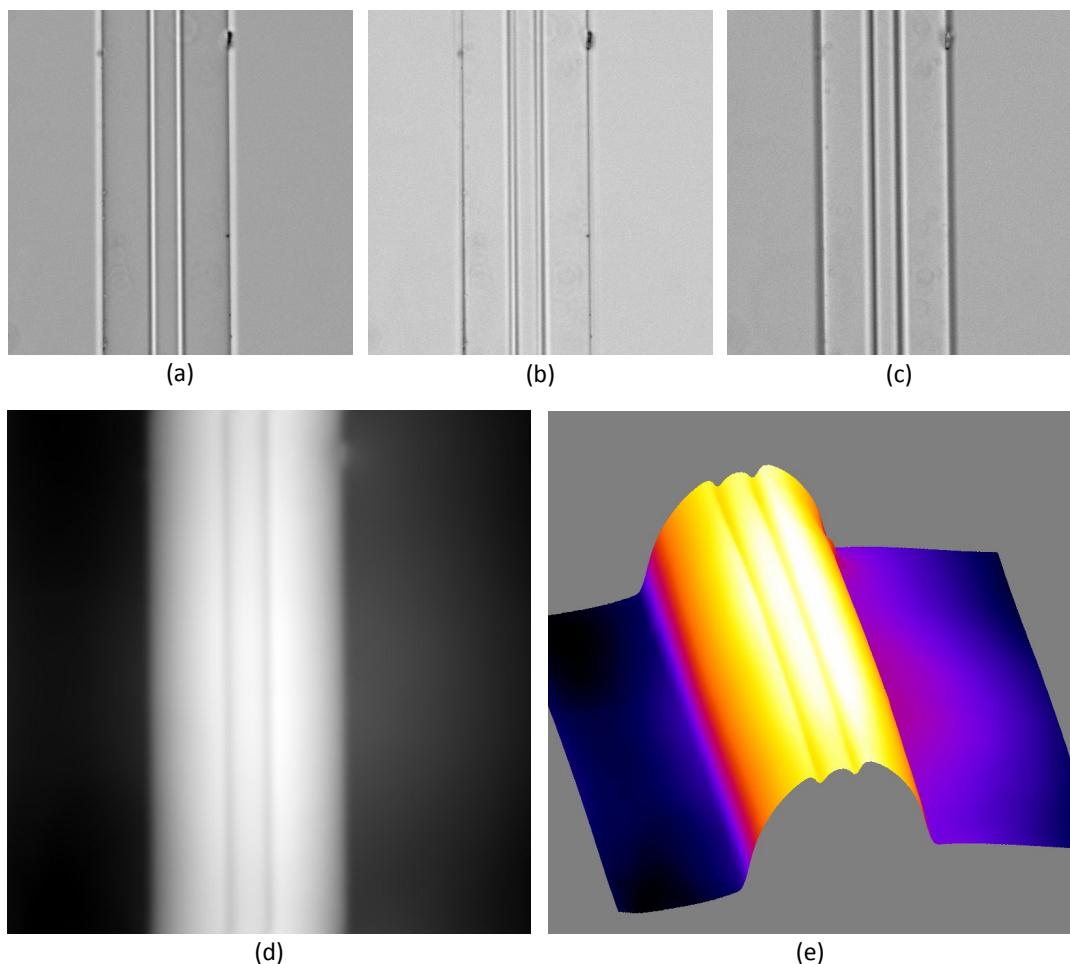


Abbildung 6.7.: Messung eines Fibertech AS 3401198 Lichtwellenleiters

Step-index Lichtwellenleiter sind zylinderförmig und bestehen aus zwei transparenten Materialien, Kern und Mantel, wobei der Mantel einen niedrigeren Brechungindex besitzt. Besonders im 3D-Modell lässt sich der lokale Phasenunterschied, der durch die unterschiedlichen Dicken und verschiedenen Materialien herbeigeführt wird, gut differenzieren.

6.3.2. Holographische Probe

Mikroskopieverfahren:	Durchlicht/ Hellfeld
Objektiv:	Leica HC PL FLUOTAR $20 \times /0,50$ PH 2
Wellenlänge:	550nm
objektseitige Defokusdistanz:	$20\mu\text{m}$
bildseitige Defokusdistanz:	4mm
gemessener max. Wegunterschied:	$0,32\mu\text{m}$

Tabelle 6.2.: Messdaten zur holographischen Probe

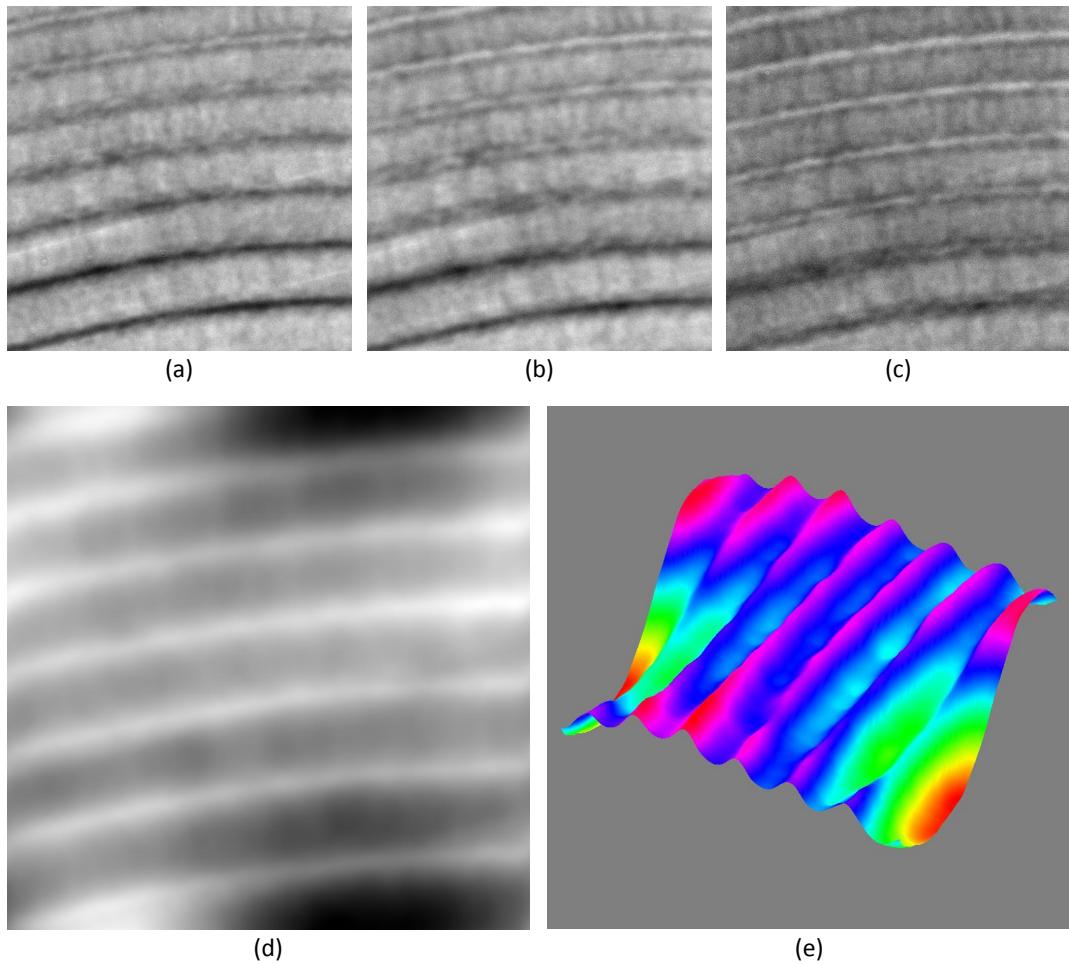


Abbildung 6.8.: Messung einer holographischen Probe

Die hier vermessene Fresnel-Zonenplatte ist ein rein phasenmodulierendes Hologramm mit konzentrischen Gitterstrukturen. Die Gitterkonstante nimmt mit zunehmendem Radius ab. Dies führt dazu, dass monochromatisches Licht auf einen Punkt fokussiert wird.

6.3.3. Diatomeen

Mikroskopieverfahren:

Durchlicht / Hellfeld

Objektiv:

Leica HC PL FLUOTAR 20 × /0,50 PH 2

Wellenlänge:

550nm

objektseitige Defokusdistanz:

20µm

bildseitige Defokusdistanz:

4mm

gemessener max. Wegunterschied:

2,73µm

Tabelle 6.3.: Messdaten zur Diatomee *Triceratium favus*

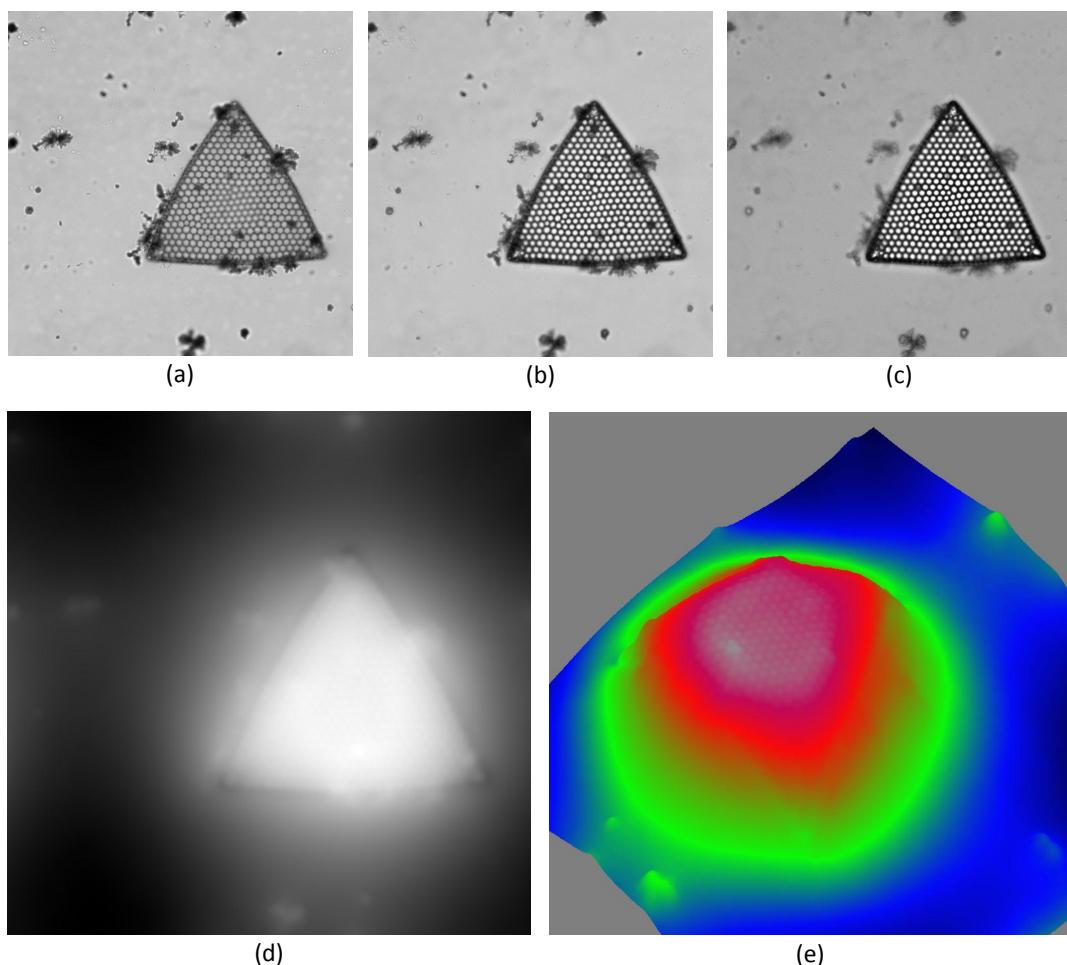


Abbildung 6.9.: Messung einer Diatomee *Triceratium favus*

6.3.4. Microlinse

Mikroskopieverfahren:	Auflicht/ Hellfeld
Objektiv:	Leitz PLAN 50 × /0,75 D
Wellenlänge:	550nm
objektseitige Defokusdistanz:	10µm
bildseitige Defokusdistanz:	12,5mm
gemessener max. Wegunterschied:	2,84µm

Tabelle 6.4.: Messdaten zur Microlinse

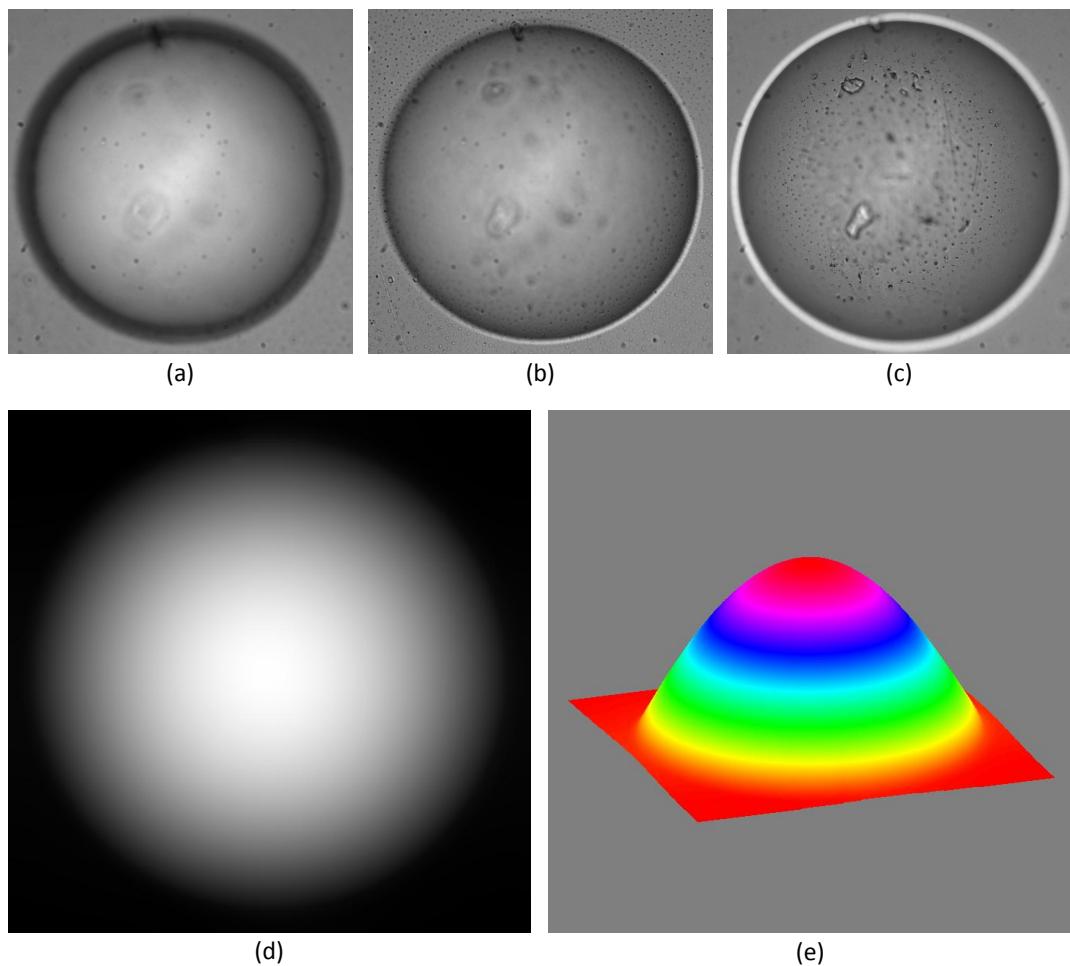


Abbildung 6.10.: Messung einer Microlinse

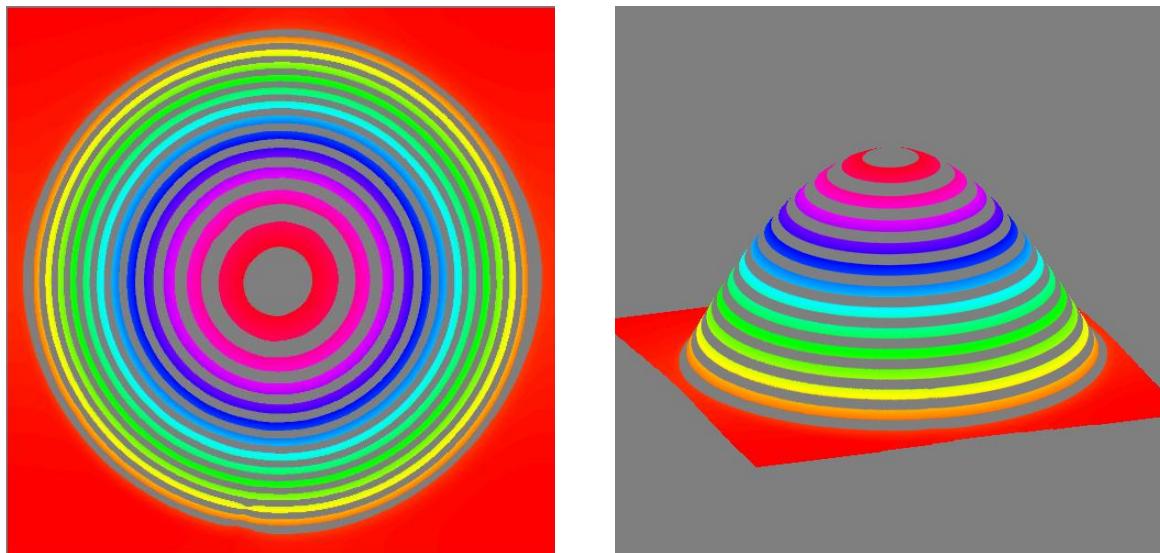


Abbildung 6.11.: ISO-Lines-Modell der Microlinse

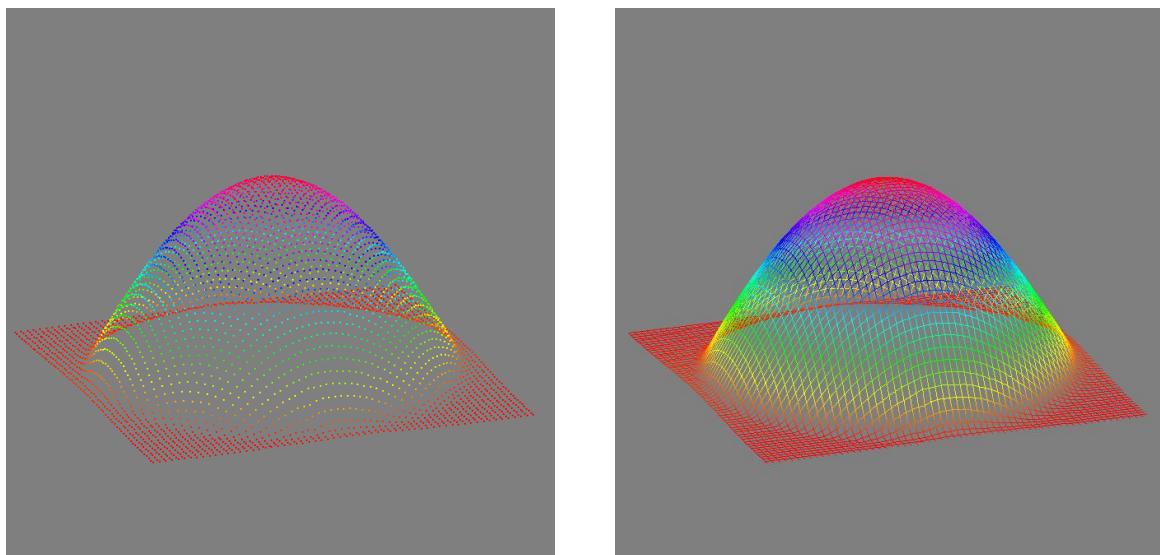


Abbildung 6.12.: Punktmodell und Gitternetzmodell der Microlinse

7. Benchmarking

Als *Benchmarking* bezeichnet man eine Methode zum Leistungsvergleich verschiedener Systeme. Hier sollen die Berechnungszeiten des ITG-Algorithmus auf der CPU und auf der GPU verglichen werden. Im weiteren Verlauf des Kapitels wird dann der Leistungsunterschied zwischen den beiden Referenzsystemen dargestellt.

Die Tabelle 7.1 zeigt die Zeit, die für die Berechnung einer FFT auf dem Host mit der *FFTW3*-Bibliothek [17] und auf dem Device mit der *CUFFT*-Bibliothek [38] benötigt wird. Dabei zeigt sich eine Beschleunigung um den Faktor 3500 auf dem Device. Selbst bei dem gesamten benötigten Zyklus, bestehend aus der FFT, dem Bildshift und Datentypkonvertierungen (Kap. 4.2.4), zeigt sich ein Geschwindigkeitsunterschied zwischen CPU und GPU von Faktor 84.

	einfache FFT	gesamter Zyklus
FFTW3 (Host)	527ms	596ms
CUFFT (Device)	0,15ms	7,13ms
Faktor	≈3500	≈84

Tabelle 7.1.: Berechnungsdauer einer FFT auf Host und Device. Es wurden simulierte Bilder der Dimension $1024px \times 1024px : 16$ bei 1000 Durchläufen auf der Workstation (Datenblatt A.2) genutzt.

Diese Werte untermauern bereits die Wahl des *Parallel Programming*, wie es in Kapitel 4 erläutert wurde.

Eine Zeitmessung aller Schritte des ITG-Algorithmus (Abb. 2.5) und der zugehörigen parallelisierbaren Bildverarbeitungsschritte (Abb. 5.17) zeigt die Tabelle 7.2. Die Messungen wurden auf beiden Referenzsystemen (Datenblätter A.2 und A.3) und jeweils in der Ausführung mit FFTW3 und CUFFT durchgeführt. Es zeigt sich auch hier eine beachtliche Beschleunigung durch den Device-Algorithmus um etwa Faktor 272.

Der Vergleich der beiden Rechnersysteme zeigt die Skalierbarkeit des CUDA-Programmiermodells. Die Notebookgrafikkarte *GeForce 9600M GT* besitzt 32 Rechenkerne, die Grafikkarte des Workstationrechners *GeForce GTX 295* besitzt 240 Kerne pro GPU. Die Messungen zeigen einen resultierenden Geschwindigkeitsunterschied von etwa Faktor 10 zwischen den GPUs, da die einzelnen Problemstellungen auf mehr Rechenkernen parallelisiert werden können.

Die theoretisch erreichbare Bildwiederholrate beträgt $47,35fps$ auf dem Workstationsystem. Tatsächlich wird eine Wiederholrate von $32,5fps$ für die zweidimensionale Darstellung erreicht, das 3D-Modell wird mit $23,9fps$ dargestellt. Der Algorithmus ist somit echtzeitfähig.

	FFTW3 (Host)		CUFFT (Device)	
	Workstation	Notebook	Workstation	Notebook
Proj. Transf. und Kopiervorg.	267,9991ms	310,0467ms	7,9171ms	152,6269ms
Differenzbild berechnen	11,1204ms	13,7517ms	0,4773ms	8,2164ms
FFT Differenzbild	58,1628ms	165,1595ms	0,2030ms	0,3151ms
Normierung FFT	318,7501ms	352,8856ms	0,9616ms	5,4090ms
Kopieren Differenzbild	8,9037ms	10,7320ms	0,0075ms	0,0411ms
Filter multiplizieren	639,8759ms	670,4188ms	2,4436ms	4,0398ms
FFT	86,6754ms	179,0923ms	0,3071ms	0,4450ms
Normierung FFT	637,6078ms	706,0109ms	1,8945ms	10,2479ms
Division durch Fokusbild	608,8577ms	550,6554ms	0,6679ms	5,8167ms
FFT	116,5904ms	325,2914ms	0,3046ms	0,4892ms
Normierung FFT	638,0638ms	705,3370ms	1,8852ms	10,3072ms
Filter multiplizieren	640,3497ms	670,8973ms	0,4882ms	2,8934ms
Summieren	226,5127ms	246,5146ms	1,2728ms	6,7192ms
FFT	43,6335ms	90,1924ms	0,2110ms	0,3338ms
Vorfaktor multiplizieren	217,4992ms	238,8250ms	0,2688ms	1,2401ms
Bildausgabe vorbereiten	1223,7881ms	1324,4834ms	1,8085ms	12,8827ms
Summe	5744,39ms	6560,29ms	21,12ms	222,02ms
fps	0,17	0,15	47,35	4,50
Faktor	≈272	≈310	1	≈10

Tabelle 7.2.: Berechnungszeiten des ITG-Algorithmus. Alle Werte entsprechen dem Mittelwert nach 1000 Durchläufen und wurden auf beiden Referenzsystemen mit Daten der Dimension $1024px \times 1024px : 16$ durchgeführt.

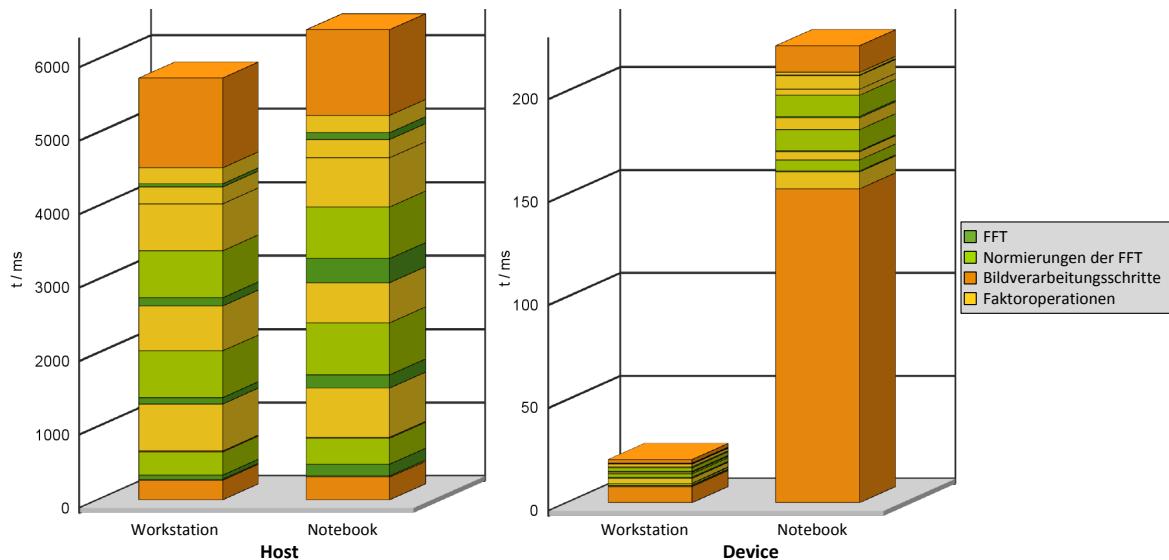


Abbildung 7.1.: Grafik aus den Benchmarkergebnissen in Tabelle 7.2. Ähnliche Operationen wurden gruppiert.

Eine Veranschaulichung der Geschwindigkeitsunterschiede bietet die Abbildung 7.2, hier wurden ähnlich aufgebaute Verarbeitungsschritte farblich zusammengefasst.

7.1. Datentransferrate

Die *Datentransferrate* beschreibt die Größe der übertragbaren Datenmenge pro Zeiteinheit. Sie ist mit das wichtigste Bewertungskriterium für die Leistungsfähigkeit eines Systems und somit auch einer Grafikkarte. Bei allen Änderungen im Programmcode sollte der Einfluss auf den Datentransfer bedacht werden [34].

Die theoretisch erreichbare Datentransferrate B_t in $\frac{GB}{s}$ lässt sich aus den Spezifikationen der Speichertaktrate f_S in Hz und der Bitbreite n_{Bit} in $Byte$ des Speichermoduls aus dem Datenblatt einer Grafikkarte errechnen.

$$B_t = \frac{f_S \cdot n_{Bit}}{1024^3} \quad (7.1)$$

Das Speichermodul des *GeForce GTX295* Grafikchips besitzt eine Speichertaktrate von $999MHz$ und eine Bitbreite von $448Bit$ pro GPU (Datenblatt A.5). Zudem arbeitet der Chip mit *GDDR-RAM*¹ und verdoppelt somit den Datendurchsatz. Daraus ergibt sich eine theoretische Datentransferrate von

$$B_t = \frac{999 \cdot 10^6 Hz \cdot (448Bit/8) \cdot 2}{1024^3} = 104,20 \frac{GB}{s} \quad (7.2)$$

pro GPU. Die *GeForce 9600M GT* (Datenblatt A.4) erreicht eine theoretische Datentransferrate von lediglich $23,84 \frac{GB}{s}$.

Die effektive Datentransferrate B_0 in $\frac{GB}{s}$ lässt sich aus der Anzahl gelesener Bytes pro Kernel N_r , die ausgegebene Anzahl Bytes pro Kernel N_w und die benötigte Zeit t in s errechnen.

$$B_0 = \frac{N_r + N_w}{1024^3 \cdot t} \quad (7.3)$$

Im konkreten Beispiel aus Tabelle 7.2 werden Daten der Größe $1024px \times 1024px$ verarbeitet. Dabei entspricht jedes Pixel im Datenformat `float 4Bytes` und jeder Kernel die Daten liest und wieder schreibt (Faktor 2). Die für die FFT-Ausführung benötigte Zeit beträgt im Mittel $170,95\mu s$. Es ergibt sich eine überschlägige effektive Datentransferrate von

$$B_0 = \frac{1024^2 \cdot 4Byte \cdot 2}{1024^3 \cdot 170,95\mu s} = 45,70 \frac{GB}{s} . \quad (7.4)$$

Die effektiv genutzte Speicherbandbreite von $45,70 \frac{GB}{s}$ liegt somit nicht einmal bei der Hälfte der theoretisch nutzbaren Speicherbandbreite von $104,20 \frac{GB}{s}$ und besitzt Potential zur weiteren Optimierung. Da das System mit *GeForce 9600M GT* im Mittel $395,78\mu s$ für eine FFT benötigt, ergibt sich hier eine effektive Speicherbandbreite von $19,74 \frac{GB}{s}$ und liegt somit am theoretisch erreichbaren Wert von $23,84 \frac{GB}{s}$. Der Datenmengentransport erschöpft hier bereits die Möglichkeiten des Systems.

¹Der *Graphics Double Data Rate Random Access Memory*, kurz GDDR-RAM, ist ein spezielles Datenspeichermodul für Grafikkarten.

8. Resümee und Ausblick

Ziel dieser Arbeit war die Entwicklung der Software φ Scope, die die Messung der Phase eines mikroskopischen Objekts in Echtzeit ermöglichen sollte. Zu diesem Zweck wurde der zuvor validierte Algorithmus zur Phasenrekonstruktion auf Grundlage der Intensitäts-Transportgleichung (Kap. 2.2) zur Ausführung auf Grafikprozessoren portiert. Die Programmierschnittstelle *CUDA C* ermöglicht das parallele Ausführen von gewöhnlichen Gleitkommaoperationen auf den GPUs des Herstellers *NVIDIA* (Kap. 4).

Besonders die schnellen Fourier-Transformationen, die einen Großteil des ITG-Algorithmus ausmachen, eignen sich zur Zerlegung in parallel berechenbare Unterprobleme und sorgen somit für eine deutliche Beschleunigung. Die FFT der CUDA-Bibliothek *CUFFT* ist um Faktor 3500 schneller als gängige CPU gestützte Implementierungen (Kap. 7). Ebenso lassen sich alle weiteren benötigten Filter und Berechnungsschritte durch Parallelisierung und den geschickten Einsatz verschiedener Speicherebenen (Kap. 4.2.1.3) deutlich beschleunigen.

Der charakteristische Aufbau des Algorithmus, wie er in Abbildung 2.5 dargestellt ist, eignet sich besonders zur Zerlegung in mehrere nebenläufige Zweige (Kap. 4.2.3.3). In jedem Zweig befindet sich dann eine sequenzielle Abfolge von Anweisungen, die parallel zu weiteren Zweigen ablaufen können und somit die Leistungsfähigkeit der GPU auslasten.

Die Kopiervorgänge vom Arbeitsspeicher des Computers in den Speicher der Grafikkarte erwiesen sich als besonders zeitintensiv. Durch den Einsatz von OpenGL (Kap. 5.1), einer Schnittstelle zur Programmierung von 2D- und 3D-Computergrafiken, erübriggt sich der abschließende Kopiervorgang des Algorithmusergebnisses von der Grafikkarte in den Arbeitsspeicher durch den Gebrauch von texturierten Objekten zur Bildausgabe.

Der Algorithmus ist von der Aquirierung der Bilddaten bis zur Ausgabe der errechneten Phasenverteilung auf dem Referenzsystem mit 32,5 Bildern die Sekunde möglich. Das System ermöglicht somit die Phasenrekonstruktion in Echtzeit.

Zur Verdeutlichung des Messergebnisses aus dem Algorithmus ist es wünschenswert, ein frei skalierbares dreidimensionales Modell mit OpenGL zu rendern. Neben frei im Raum rotierbaren 3D-Modellen ist es möglich, das Höhenprofil einer Probe durch die Darstellung von ISO-Lines zu verdeutlichen.

Selbst das Rendern eines 3D-Modells ist bei entsprechenden Einstellungen auf dem Referenzsystem mit 23,9 Bildern pro Sekunde, also in Echtzeit, möglich.

Zur Detektion der benötigten Intensitätsverteilungen in den jeweiligen Fokus- und Defokusebenen wurde das in der vorangegangenen Arbeit [5] entwickelte Drei-Kamerasystem mit *AVT Pike 145B* CCD-Kameras genutzt (Kap. 3). Da bis zum Abschluss dieser Arbeit der spezielle kubische Strahlteiler nicht zur Verfügung stand wurde lediglich ein Zwei-Kamera-System mit kubischem 50:50-Strahlteilerwürfel verwendet. Zu diesem Zweck

enthält φ Scope die Möglichkeit, das Fokusbild aus dem arithmetischen Mittelwert der beiden Defokusaufnahmen zu errechnen. So werden, wenn auch auf Kosten der Messgenauigkeit, lediglich zwei CCD-Kameras zur Aufnahme benötigt.

Um den Transport der Bilddaten von den Kameras in den Arbeitsspeicher des Rechnersystems echtzeitfähig zu gestalten, war es nötig, jeder Kamera einen eigenen *FireWire-Bus* zur Verfügung zu stellen (Kap. 5.2). Auf Seiten der Software sind alle Funktionen zur Kameraansprache in einem eigenen Host-Thread integriert, der parallel zum restlichen Programm läuft. So ist es möglich, bereits neue Bilder zu detektieren, während unabhängig dazu der ITG-Algorithmus und die Bildausgabe berechnet werden.

Die Abbildungsverhältnisse des optischen Systems bewirken für jedes Kamerabild verschiedene Abbildungsmaßstäbe und Verschiebungen gegen die ideale Bildposition. Ein Abgleich der verschiedenen Intensitätsverteilungen aufeinander ist mit projektiven Transformationen möglich (Kap. 5.3). Inhomogenitäten in der Abbildung, wie sie beispielsweise durch verschmutzte optische Komponenten entstehen, können durch eine *Shading Correction* ausgeglichen werden (Kap. 5.4).

Zusammenfassend lässt sich sagen, dass die Softwarelösung φ Scope die Phasenrekonstruktion von Objekten im Hellfeld von Auflicht- und Durchlichtmikroskopen in Echtzeit ermöglicht. Hierzu werden neben dem Kamerasytem lediglich gängige Consumer-Hardware und ein gewöhnliches Labormikroskop benötigt.

Die praktische Anwendung des Mikroskopsystems hat gezeigt, dass besonders das Kalibrieren des Messaufbaus sehr aufwändig ist und eine lange Einarbeitungszeit erfordert. Zudem muss nach jeder Veränderung der Abbildungsbedingungen, z.B. Objektivwechsel oder Verändern der bildseitigen Defokusdistanz, der gesamte Kalibriervorgang wiederholt werden. Abhilfe würde ein *automatisiertes System* schaffen, basierend auf präzisen Linearmotoren. Hierzu wäre ein spezieller Kalibrieralgorithmus nötig, der jeden CCD-Chip an die korrekte Position verschiebt und die Einstellung der projektiven Transformation übernimmt. Dadurch würde die Benutzung des Systems bedeutend praxistauglicher und eine intuitivere Nutzung ohne lange Einarbeitungsphase wäre möglich.

Die *Kontrastierungsverfahren* Dunkelfeldbeleuchtung, Phasenkontrast und differentieller Interferenzkontrast ließen sich, wie in Arbeit [4] beschrieben, leicht in die Software implementieren und würden die Einsatzbereiche des Systems enorm vergrößern. Für viele Anwender könnte eine entsprechende Software die Anschaffung teurer spezieller Mikroskope ersparen. Ebenso wäre es möglich, einmal als Bild oder Video detektierte Proben nachträglich den verschiedenen Verfahren zu unterziehen oder das entsprechende 3D-Modell noch einmal aus beliebiger Perspektive zu betrachten.

Die Ausdrucksstärke des dreidimensionalen Modells ließe sich durch den Einsatz von modernen *stereoskopischen 3D-Darstellungen* deutlich steigern. Eine stereoskopische Darstellung lässt sich am einfachsten durch anaglyphe Bilder erzeugen, zu deren Betrachtung eine Rot-Zyan-Filterbrille benötigt wird. Die momentan gängigste Technik im Consumer-Bereich setzt auf 120Hz-Monitore oder Projektoren zusammen mit Shutter-Brillen. Hierzu wird für den Blickwinkel von jedem Auge ein gesondertes Bild erzeugt und nacheinander angezeigt. Die Shutterbrille gibt zum passenden Zeitpunkt das entsprechende Auge des Betrachters frei und dunkelt das jeweils andere Auge ab [49].

Zusätzlich zum ITG-Algorithmus bietet es sich an, alternative Algorithmen zur Phasenrekonstruktion zu implementieren, die das vorhandene Kamera-System nutzen könnten. Algorithmen, die auf der ersten Green'schen Identität beruhen [50], besitzen einen ähnlichen Aufbau und stützen sich ebenfalls stark auf Fourier-Transformationen. Zudem sind verschiedene Randbedingungen anwendbar, die zum Objekt passend gewählt werden können.

Die Funktionsstruktur zur Phasenrekonstruktion, wie sie in *φScope* genutzt wird, ließe sich in einer eigenen API aufbauen. Mit solch einer Bibliothek wäre es ein Leichtes, die Funktionalitäten des Programms in bestehende Mikroskopiesoftwarelösungen zu integrieren. Im Rahmen dieses Schritts wäre abzuwägen, ob ein Wechsel von CUDA C zu der Programmierplattform *OpenCL* [45] sinnvoll wäre. OpenCL besitzt einen ähnlichen Funktionsumfang wie CUDA, ist allerdings nicht an einen Hersteller gebunden und macht somit in der Auswahl der Hardware flexibler.

Wäre es möglich, die Methoden zur Phasenrekonstruktion in den makroskopischen Bereich zu transportieren, könnten viele aufwändige auf Interferometrie oder Holographie basierende Messverfahren durch den Einsatz des ITG-Algorithmus kosteneffizient ersetzt werden. Auch die Anwendung der Technologie in der herkömmlichen Video- und Fototechnik ist denkbar, wenn auch sehr optimistisch. Bisher konnten keine positiven Erfahrungen bei der Vermessung streuer Proben gemacht werden.

Momentan denkbare Einsatzgebiete finden sich vor allem in der *Online-Prozesskontrolle* im Produktionszyklus technischer Komponenten. Beispielsweise könnten die optischen Eigenschaften einer gerade gezogenen Glasfaser online geprüft oder bei der Produktion diffraktiver Elemente die entstandene Struktur betrachtet und ausgewertet werden.

Weiter ist es denkbar, typische Proben in den Bereichen *Biologie*, *Medizin* und *Chemie* mit dem System zu vermessen. Auch Proben, die sich zeitlich verändern, wie Reaktionsvorgänge in der Chemie oder Zellteilungen in der Biologie, wären in Echtzeit betrachtbar. Diese Proben könnten gleichzeitig digital gespeichert und immer wieder auf vielfältige Weise reproduziert und ausgewertet werden.

A. Datenblätter



Resolution:	$1388px \times 1038px$
Max. frame rate:	$30fps$
Sensor type:	CCD Progressive
Interface:	IEEE 1394b - 800 Mb/s, 2 ports, daisy chain
ADC:	14 Bit
Output:	8-14 Bit
Sensor size:	Type2/3
Sensor:	Sony ICX285
Pixel size:	$6.45\mu m \times 6.45\mu m$
On-board FIFO:	64MB
Body dimensions:	$96.8 \times 44 \times 44mm$ including connectors

Tabelle A.1.: Datenblatt: Allied Vision Technologies PIKE F-145

Prozessor	Intel Core i7 920 - QuadCore 2,67 GHz
Mainboard	ASUS P6T SE
Arbeitsspeicher	12 GB RAM 2,6 GHz
Grafikkarte	ASUS GeForce GTX295 2DI - 1792 MB GDDR3
Festplatte	500 GB S-ATA 7200rpm
Betriebssystem	Windows 7 Professional 32Bit
CUDA API	NVIDIA CUDA 3.1 32Bit
Kamera API	AVT FirePackage 2.11 32Bit
IDE	Microsoft Visual Studio 2008 32Bit
FireWire-Adapter	1× AVT FWB-PCIE1x11B, 2× Delock 89109

Tabelle A.2.: Systemkomponenten des Workstation-Systems

Notebook	Apple MacBook Pro 15" - Generation 7 (Juni 2009)
Prozessor	Intel Core2Duo P8800 - DualCore 2,66 GHz
Arbeitsspeicher	4 GB RAM
Grafikkarte	NVIDIA GeForce 9600M GT - 512 MB GDDR3
Festplatte	500 GB S-ATA 7200rpm
Betriebssystem	Windows 7 Professional 32Bit
CUDA API	NVIDIA CUDA 3.1 32Bit
Kamera API	AVT FirePackage 2.11 32Bit
IDE	Microsoft Visual Studio 2008 32Bit
FireWire-Adapter	onboard IEEE 1394b

Tabelle A.3.: Systemkomponenten des mobilen Systems

GPU Engine Specs:	
CUDA Cores	32
Graphics Clock (MHz)	120 MHz
Memory Specs:	
Standard Memory Config	512 MB GDDR3
Memory Clock (MHz)	800 MHz
Memory Interface Width	128-bit
Max. Memory Config	1024 MB GDDR2 oder GDDR3
Feature Support:	
NVIDIA PureVideo Technology	HD
HybridPower Technology	yes
Certified for Windows 7	yes

Tabelle A.4.: Datenblatt: NVIDIA GeForce 9600M GT

GPU Engine Specs:	
CUDA Cores	480 (240 per GPU)
Graphics Clock (MHz)	576 MHz
Processor Clock (MHz)	1242 MHz
Texture Fill Rate (billion/sec)	92.2
Memory Specs:	
Memory Clock (MHz)	999
Standard Memory Config	1792 MB GDDR3 (896MB per GPU)
Memory Interface Width	896-bit (448-bit per GPU)
Memory Bandwidth (GB/sec)	223.8
Feature Support:	
NVIDIA SLI-ready	Quad
NVIDIA 3D Vision Ready	yes
NVIDIA PureVideo Technology	HD
NVIDIA PhysX-ready	yes
NVIDIA CUDA Technology	yes
Microsoft DirectX	10
OpenGL	2.1
Certified for Windows 7	yes
Display Support:	
Maximum Digital Resolution	2560x1600
Maximum VGA Resolution	2048x1536
Standard Display Connectors	Two Dual Link DVI
HDMI	yes
Multi Monitor	yes
HDCP	yes
HDMI	yes
Audio Input for HDMI	SPDIF
Standard Graphics Card Dimensions:	
Height	4.376 inches (111 mm)
Length	10.5 inches (267 mm)
Width	Dual-slot
Thermal and Power Specs:	
Maximum GPU Temperature (in C)	105 C
Maximum Graphics Card Power (W)	289 W
Minimum Recommended System Power (W)	680 W
Supplementary Power Connectors	6-pin and 8-pin
High Dynamic-Range Lighting	128bit
Unified Architecture	yes

Tabelle A.5.: Datenblatt: NVIDIA GeForce GTX295

Essentials:	
Status	Launched
Launch Date	Q4 2008
Processor Number	i7-920
Number of Cores	4
Number of Threads	8
Clock Speed	2.66 GHz
Max Turbo Frequency	2.93 GHz
Intel Smart Cache	8 MB
Bus/Core Ratio	20
Intel QPI Speed	4.8 GT/s
Number of QPI Links	1
Instruction Set	64-bit
Instruction Set Extensions	SSE4.2
Embedded Options Available	No
Supplemental SKU	No
Lithography	45 nm
Max TDP	130 W
VID Voltage Range	0.800V-1.375V
Tray 1ku Budgetary Price	\$284.00
Memory Specifications:	
Max Memory Size (dependent on memory type)	24 GB
Memory Types	DDR3-800/1066
Number of Memory Channels	3
Max Memory Bandwidth	25.6 GB/s
Physical Address Extensions	36-bit
ECC Memory Supported	No
Package Specifications:	
Max CPU Configuration	1
TCASE	67.9°C
Package Size	42.5mm x 45.0mm
Processing Die Size	263 mm ²
Number of Processing Die Transistors	731 million
Sockets Supported	FCLGA1366
Halogen Free Options Available	Yes

Tabelle A.6.: Datenblatt: Intel Core i7 920

Essentials:	
Status	Launched
Launch Date	Q2 09
Processor Number	P8800
Number of Cores	2
Number of Threads	2
Clock Speed	2.66 GHz
L2 Cache	3 MB
Bus/Core Ratio	10
FSB Speed	1066 MHz
Instruction Set	64-bit
Embedded Options Available	No
Supplemental SKU	No
Lithography	45 nm
Max TDP	25 W
VID Voltage Range	1.00V-1.25V
Tray 1ku Budgetary Price	\$241.00
Package Specifications:	
TJUNCTION	105°C
Package Size	35mm x 35mm
Processing Die Size	107 mm ²
Number of Processing Die Transistors	410 million
Sockets Supported	BGA479, PGA478
Halogen Free Options Available	Yes

Tabelle A.7.: Datenblatt: Intel Core2Duo P8800

Controller	Texas Instruments XIO2213A
Interface	PCI Express 1.1, 1-lane
Anschlüsse	3× IEEE 1394b, verschraubar, Spannungsvers. (max. 1500mA)
Geschwindigkeit	100, 200, 400, 800 MBit/s
Bandbreite	100/80/64 MB/s @ 800MBit/s
Externe Spannungsversorgung	12 – 32V DC

Tabelle A.8.: Datenblatt: AVT FWB-PCIE1x11B

Interface	PCI Express 1.1, 1-lane
Anschlüsse	2× IEEE 1394b, 1× IEEE 1394a
Geschwindigkeit	100, 200, 400, 800(1394b) MBit/s
Externe Spannungsversorgung	12 – 32V DC

Tabelle A.9.: Datenblatt: Delock 89109

Literaturverzeichnis

- [1] WERNICKE, G. ; OSTEN, W.: *Holografische Interferometrie*. Weinheim, Germany : Physik-Verlag, 1982. – ISBN 3–87664–066–0
- [2] KLUCK, A.: *Iterative Algorithmen zur Phasenrekonstruktion*, Fachhochschule Köln, Diplomarbeit, 2009
- [3] HORTSMANN, J.: *Anwendung der Intensitätstransportgleichung auf die 3D-Messtechnik*, Fachhochschule Köln, Diplomarbeit, 2008
- [4] MATRISCH, J.: *Anwendung der Intensitätstransportgleichung und Evaluierung darauf beruhender Messverfahren*, Fachhochschule Köln, Diplomarbeit, 2009
- [5] WETTE, S.: *Mehr-kamerabasierende 3D-Echtzeitmikroskopie auf Grundlage der Intensitäts-Transportgleichung*, Fachhochschule Köln, Diplomarbeit, 2010
- [6] NORMUNG, Deutsches I. (Hrsg.): *DIN 5031 - Teil 7: Strahlungsphysik im optischen Bereich und Lichttechnik - Benennung der Wellenlängenbereiche*. Januar 1984
- [7] BERGMANN, L. ; SCHÄFER, C.: *Lehrbuch der Experimentalphysik, Band 3, Optik*. 9th. Berlin, Germany : Walter de Gruyter, 1993. – ISBN 3–11–012973–6
- [8] HECHT, E.: *Optik*. 4. München, Deutschland : Oldenbourg Wissenschaftsverlag, 2005. – ISBN 3–48–627359–0
- [9] BORN, M. ; WOLF, E.: *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light*. 7th. Cambridge, UK : Cambridge University Press, 1999. – ISBN 0–52–164222–1
- [10] PEDROTTI, F. ; PEDROTTI, L. ; BAUSCH, W. ; SCHMIDT, H.: *Optik für Ingenieure - Grundlagen*. 4. Berlin, Germany : Springer-Verlag, 2007. – ISBN 3–540–73471–6
- [11] KREIS, T.: *Handbook of Holographic Interferometry*. Weinheim, Germany : Wiley-VCH, 2005. – ISBN 3–527–40546–1
- [12] TEAGUE, Michael R.: Deterministic phase retrieval: a Green's function solution. In: *J. Opt. Soc. Am.* Volume 73 (1983), November, Nr. 11, S. 1434–1441
- [13] SALEH, Bahaa E. A. ; TEICH, Malvin C.: *Fundamentals of Photonics*. 2nd. New Jersey, USA : John Wiley and Sons, 2007. – ISBN 0–471–35832–0

- [14] BURGER, W. ; BURGE, M. J.: *Digitale Bildverarbeitung: Eine Einführung mit Java und ImageJ*. 2. Berlin, Germany : Springer, 2006. – ISBN 3–540–30940–3
- [15] GOODMAN, J. W.: *Introduction to Fourier*. 3rd. Englewood, USA : Roberts and Company, 2005. – ISBN 0–974–70772–4
- [16] COOLEY, J. W. ; TUKEY, J. W.: An Algorithm for the Machine Calculation of Complex Fourier Series. In: *Mathematics of Computation* Volume 19 (1965), Nr. 90, 297–301. <http://www.amath.washington.edu/~narc/win08/papers/cooley-tukey.pdf>
- [17] FFTW.ORG (Hrsg.): *Fastest Fourier Transform in the West - FFTW*. <http://www.fftw.org>. Version: Oktober 2010
- [18] GONZALEZ, R. C. ; WOODS, R. E.: *Digital Image Processing*. 3rd. New Jersey, USA : Prentice Hall International, 2007. – ISBN 0–131–68728–x
- [19] JÄHNE, B.: *Digitale Bildverarbeitung*. 6. Berlin, Germany : Springer, 2005. – ISBN 3–540–24999–0
- [20] ALLEN, L. ; OXLEY, M.: Phase retrieval from series of images obtained by defocus variation. In: *Optics Communications* Volume 199 (2001), Nr. 1-4, S. 65–75
- [21] MÖLLMANN, F.: *Wellenfront-Korrektur im Mikroskop mit Hilfe phasenmodulierender Flüssigkristalldisplays*, Fachhochschule Köln, Bachelorarbeit, 2010
- [22] CHRISTIAN LINKENHELD (Hrsg.): *Pfad durch die Lichtmikroskopie*. <http://www.mikroskopie.de/pfad/architekturen/drei.html>. Version: September 2010
- [23] KÖHLER, A.: Ein neues Beleuchtungsverfahren für mikrophotographische Zwecke. In: *Zeitschrift für wissenschaftliche Mikroskopie* X (1893), Nr. 4, S. 433–440
- [24] S. ALTMAYER: *Vorlesungsskript - Lichtmikroskopie*. FH-Köln, 2009/2010
- [25] MOORE, G.: Cramming more components onto integrated circuits. In: *Electronics* 19 (1965), Nr. 3, 114-117. <http://download.intel.com/research/silicon/moorespaper.pdf>
- [26] KIRK, D. ; HWU, W.-M.: *Programming Massively Parallel Processors: A Hands-On Approach*. 1st. Sebastopol, USA : O'Reilly Media, 2009. – ISBN 0–596–52153–7
- [27] NEUMANN, J. von: *First draft of a report on the EDVAC*. 1945
- [28] SUTTER, H.: The Concurrency Revolution. In: *C/C++ Users Journal* 23 (2005), Februar, Nr. 2. <http://www.gotw.ca/publications/concurrency-ddj.htm>
- [29] RAGSDALE, S.: *Parallele Programmierung: Grundlagen, Anwendungen, Methoden*. 1. Berlin, Germany : McGraw-Hill Book Company GmbH, 1992. – ISBN 3–890–28397–7

- [30] POYNTON, C.: *A Technical Introduction to Digital Video*. 1st. New Jersey, USA : John Wiley and Sons, 1996. – ISBN 0-471-12253-X
- [31] NVIDIA CORPORATION: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. (2010). http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
- [32] TOP500 PROJECT (Hrsg.): *TOP500 List - June 2010*. <http://www.top500.org/list/2010/06/100>. Version: Juni 2010
- [33] NVIDIA CORPORATION: NVIDIA CUDA C Programming Guide (Version 3.1.1). (2010). http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf
- [34] NVIDIA CORPORATION: NVIDIA CUDA C Best Practices Guide (Version 3.1). (2010). http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf
- [35] NVIDIA CORPORATION: NVIDIA CUDA Reference Manual (Version 3.1). (2010). http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CudaReferenceManual.pdf
- [36] NVIDIA CORPORATION: The CUDA Compiler Driver NVCC (Version 3.1). (2010). http://developer.nvidia.com/object/cuda_3_1_downloads.html
- [37] NVIDIA CORPORATION (Hrsg.): *NVIDIA GPU Computing Developer Home Page*. <http://developer.nvidia.com/object/gpucomputing.html>. Version: Oktober 2010
- [38] NVIDIA CORPORATION: CUDA CUFFT Library). (2010). http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUFFT_Library_3.1.pdf
- [39] ANDREWS, G.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. 1st. Massachusetts, USA : Addison Wesley, 2000. – ISBN 0-201-35752-6
- [40] BRESHEARS, C.: *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. 3rd. Massachusetts, USA : Morgan Kaufman Publ Inc, 2010. – ISBN 0-123-81472-3
- [41] SANDERS, J. ; KANDROT, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Massachusetts, USA : Addison Wesley, 2010. – ISBN 0-131-38768-5
- [42] M. HARRIS; NVIDIA CORPORATION: Optimizing Parallel Reduction in CUDA. (2010). http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

- [43] KHRONOS GROUP (Hrsg.): *OpenGL - The Industry's Foundation for High Performance Graphics.* <http://www.opengl.org/>. Version: Oktober 2010
- [44] NEIDER, J. ; DAVIS, T. ; Woo, M.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1.* 1st. Massachusetts, USA : Addison Wesley, 1993 <http://glprogramming.com/red/>. – ISBN 0-201-63274-8
- [45] KHRONOS GROUP (Hrsg.): *KHRONOS GROUP: Open Standards for Media Authoring and Acceleration.* <http://www.khronos.org/>. Version: Oktober 2010
- [46] SILICON GRAPHICS, INC.: The OpenGL Machine. (1996). <http://www.opengl.org/documentation/specs/version1.1/state.pdf>
- [47] ALLIED VISION TECHNOLOGIES GMBH: AVT FirePackage - für FireWire Kameras. (2010), Oktober. <http://www.alliedvisiontec.com/de/produkte/software/windows/avt-firepackage.html>
- [48] ALLIED VISION TECHNOLOGIES GMBH: AVT Pike - Technical Manual V4.1.0. (2008), August. http://www.alliedvisiontec.com/fileadmin/content/PDF/Products/Technical_Manual/Pike/Pike_TechMan_V4.1.0_en.pdf
- [49] NVIDIA CORPORATION (Hrsg.): *NVIDIA 3D Vision.* <http://www.nvidia.de/object/3d-vision-main-de.html>. Version: November 2010
- [50] FRANK, J. ; ALTMAYER, S. ; WERNICKE, G.: Non-interferometric, non-iterative phase retrieval by Green's functions. In: *J. Opt. Soc. Am. A* Volume 27 (2010), September, Nr. 10, 2244-2251. <http://www.opticsinfobase.org/abstract.cfm?URI=josaa-27-10-2244>

Abbildungsverzeichnis

2.1. Harmonische Welle mit Punkten gleicher Phase	5
2.2. Huygenssches Prinzip	6
2.3. Phasenverzerrung am transparenten Objekt	7
2.4. Auswirkungen der Abtastung auf Frequenzspektrum	14
2.5. Ablaufdiagramm des ITG-Algorithmus	20
2.6. einfacher Mikroskopstrahlengang	21
2.7. Strahlengang: Köhlersche Beleuchtung	23
2.8. Strahlengang: Durchlichtmikroskop	24
2.9. Strahlengang: Auflichtmikroskop	24
2.10. Airy-Scheibchen	25
2.11. Numerische Apertur am Mikroskopobjektiv	25
3.1. Defokusebenen im Abbildungsstrahlengang des Mikroskop	26
3.2. Strahlengang des Kamera-Systems	27
3.3. Skizze des Kamera-Systems	29
3.4. Foto des Laboraufbaus	30
4.1. Parallelisierung	32
4.2. Leistungssteigerung von GPUs und CPUs	34
4.3. Schematischer Aufbau von CPU und GPU	35
4.4. Von NVIDIA CUDA unterstützte Sprachen und APIs	36
4.5. Skalierbarkeit von CUDA-Programmen auf eine Anzahl Rechenkerne	38
4.6. Aufbau eines Grid aus mehreren Thread-Blocks	41
4.7. Speicherhierarchie in CUDA	42
4.8. Heterogenität des Programmiermodells	44
4.9. Speichermodell auf CUDA GPUs	47
4.10. Matrixmultiplikation in verschiedenen Blocks	49
4.11. Matrixmultiplikation mit Shared Memory	50
4.12. Nebenläufiges Programm mittels Streams	53
4.13. Periodizität eindimensionaler Fourier-Transformierter	56
4.14. Periodizität zweidimensionaler Fourier-Transformierter	57
4.15. Quadrantenshift eines Airy-Scheibchens	57
4.16. Amdahlsches Gesetz	59
4.17. Ablauf einer Reduktion	64
5.1. OpenGL Rendering-Pipeline	68
5.2. Ausgabe des Quelltext 5.1	70

5.3.	3D-Koordinatensystem	71
5.4.	Screenshot zu Quellcode 5.2	72
5.5.	Kombinierte Matrixoperationen	76
5.6.	Perspektivisches Frustum	77
5.7.	Orthogonales Frustum	77
5.8.	Vergleich zwischen orthogonaler und perspektivischer Darstellung	78
5.9.	Intensität, Phase und 3D-Plot einer Microlinse	80
5.10.	Verschiedene 3D-Plots einer Microlinse	81
5.11.	Projektive Abbildung des Einheitsquadrat	84
5.12.	Projektive Abbildung beliegiger Vierecke	85
5.13.	Quelle-Ziel Abbildung	86
5.14.	Bilineare Interpolation	86
5.15.	Vergleich verschiedener Interpolationsverfahren	88
5.16.	Shading Correction	89
5.17.	Programmablaufplan	91
6.1.	Kalibrierobjekt	93
6.2.	φ Scope - Benutzeroberfläche nach Programmstart	94
6.3.	φ Scope - Benutzeroberfläche im Kalibriermodus	95
6.4.	φ Scope - Vergleich von Kamerabildern	96
6.5.	φ Scope - Kalibrieren der projektiven Transformation	97
6.6.	φ Scope - Benutzeroberflächen im Messmodus	100
6.7.	Messergebnis: Lichtwellenleiter	101
6.8.	Messergebnis: Holographische Probe	102
6.9.	Messergebnis: Diatomeen	103
6.10.	Messergebnis: Microlinse	104
6.11.	Messergebnis: Microlinse - ISO-Lines-Modell	105
6.12.	Messergebnis: Microlinse - Punkt- und Gitternetzmodell	105
7.1.	Benchmark des ITG-Algorithmus	107

Tabellenverzeichnis

3.1. Bauteilliste des Kamera-Systems	27
4.1. Kennzeichnung der Variablen Deklaration	48
4.2. Fähigkeiten der CUFFT API	54
5.1. Suffixe und Datentypen in OpenGL	71
6.1. Messdaten: Lichtwellenleiter	101
6.2. Messdaten: Holographische Probe	102
6.3. Messdaten: Diatomeen	103
6.4. Messdaten: Microlinse	104
7.1. Benchmark FFTW versus CUFFT	106
7.2. Benchmark des ITG-Algorithmus	107
A.1. Datenblatt: AVT PIKE F-145	112
A.2. Systemkomponenten des Workstation-Systems	113
A.3. Systemkomponenten des mobilen Systems	113
A.4. Datenblatt: NVIDIA GeForce 9600M GT	113
A.5. Datenblatt: NVIDIA GeForce GTX295	114
A.6. Datenblatt: Intel Core i7 920	115
A.7. Datenblatt: Intel Core2Duo P8800	116
A.8. Datenblatt: AVT FWB-PCIE1x11B	116
A.9. Datenblatt: Delock 89109	116

Quellcodeverzeichnis

4.1.	Addition der Vektoren A und B der Größe N	39
4.2.	Addition der quadratischen Matrizen A und B der Größe $N \times N$	40
4.3.	Addition der quadratischen Matrizen A und B der Größe $N \times N$ in mehreren Blocks	40
4.4.	Das erste lauffähige CUDA Programm	45
4.5.	Host-Funktionen zum Management von Global Memory	46
4.6.	Kernel zur Matrixmultiplikation in Blocks	49
4.7.	Optimierter Kernel zur Matrixmultiplikation	50
4.8.	Abfrage der Anzahl der Devices und deren Eigenschaften	52
4.9.	Vollständige Beschreibung eines Kernel-Aufrufs	52
4.10.	Beispiel zur Anwendung von Streams (zu Abb. 4.12)	53
4.11.	Mögliche Transformationen mittels CUFFT und zugehörige Datentypen .	55
4.12.	Komplexe 2D-Transformation eingeladener Bilddaten	57
4.13.	Einfache sequenzielle Summenbildung	61
4.14.	Einfache sequentielle Funktion zur Reduktion	62
4.15.	Erster Reduktions-Kernel	63
4.16.	Optimierter Reduktions-Kernel	65
5.1.	Zweidimensionales OpenGL-Beispiel	70
5.2.	Dreidimensionales OpenGL-Beispiel mit GLUT	71
5.3.	UV-Koordinaten auf einem Quadrat	79
5.4.	Ausgabe einer zweidimensionalen Textur	79

Erklärungen

Eidesstattliche Erklärung:

Ich versichere hiermit, die vorliegende Arbeit in dem gemeldeten Zeitraum ohne fremde Hilfe verfasst und mich keiner anderen als der angegebenen Hilfsmittel und Quellen bedient zu haben.

Köln, den 06.12.2010

Jan-Maximilian Beneke

Weitergabeerklärung:

Ich erkläre hiermit mein Einverständnis, dass das vorliegende Exemplar meiner Diplomarbeit oder eine Kopie hiervon für wissenschaftliche Zwecke verwendet werden darf.

Köln, den 06.12.2010

Jan-Maximilian Beneke

Sperrvermerk

Die Einsicht in die vorgelegte Arbeit unterliegt einem Sperrvermerk bis zum 31.12.2012.