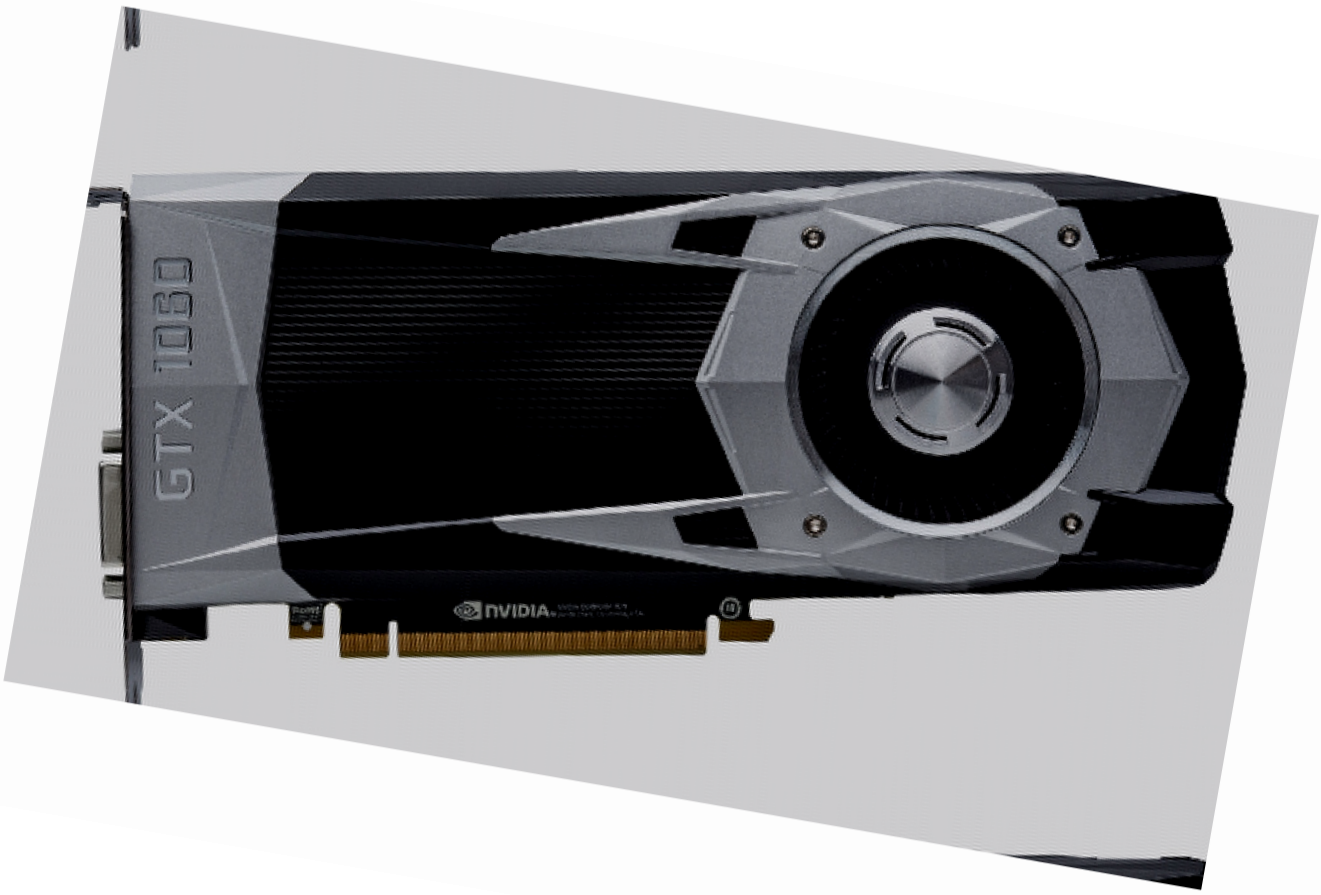
Hector Seminar

3D Ultrasound Computer Tomography

Karlsruher Institut für Technologie

GPU-Implementierung einer nicht-verlustbehafteten 3D-Bildrotation



Abschlussbericht im Rahmen der Kooperationsphase 2021/22

Durchgeführt am Institut für Prozessdatenverarbeitung und Elektrotechnik
Karlsruher Institut für Technologie (KIT)
Betreut durch Michael Zapf

Rouven Kiefer, Kurs KA16
kieferro@hector-seminar.de

Inhaltsverzeichnis

Abstract

Breast cancer is a very common type of cancer for women. The disease can still be treated well before metastases form. This makes it particularly important that early detection of the disease is possible. Mammography is currently used as an imaging method for this purpose. It is based on the use of X-rays and thus exposes the patient to harmful ionising radiation.
In addition, the breast is compressed in this procedure and only two-dimensional projection images are created.
An alternative method that does not have these disadvantages is currently being developed at the Karlsruhe Institute of Technology (KIT). It is based on three-dimensional Ultrasound Computer-Tomography (USCT).
For this procedure, the breast is scanned by hemispherically arranged ultrasound transducers.
Due to the different perspective of the transducers, many image rotations are needed to combine the scan data. Standard image rotation algorithms are based on interpolation and thus have a loss of data. The image reconstruction algorithm is iterative, which causes the error to add up over the large number of required rotations.
A better approach would be lossless image rotation. This has already been developed and implemented as CPU code, but is still too slow for the actual use case of a three-dimensional image with a side length of about 250 pixels.
Since it is a process that can be parallelised well, a GPU implementation could achieve a significant speedup. This paper examines the implementation of an efficient GPU approach at both the algorithmic and hardware architecture levels. Based on the results, the CPU and GPU implementations are then compared.
Lossless image rotation is not only applicable for image reconstruction, but also for image registration and in other areas.

1 Einleitung

Brustkrebs ist bei Frauen eine der am häufigsten auftretenden Krebsarten. Laut Daten des Robert-Koch-Instituts gab es in Deutschland im Jahr 2018 69.900 Neuerkrankungen [1]. Damit machte Brustkrebs bei Frauen ungefähr 30% [1, 2] der Krebsneuerkrankungen 2018 aus.
Für die erfolgreiche Behandlung ist eine frühzeitige Erkennung der Erkrankung entscheidend, da sich Brustkrebs vor einer Ausbildung von Metastasen noch gut therapieren lässt.
Eines der wichtigsten bildgebenden Verfahren zur Brustkrebsfrüherkennung zurzeit ist Mammographie. Dabei wird Röntgenstrahlung eingesetzt, um Auffälligkeiten in der Brust zu erkennen [3].

1.1 Das USCT-Projekt

Am Karlsruher Institut für Technologie wird zurzeit ein Verfahren entwickelt, welches im Gegensatz zur Mammographie auf Ultraschall beruht und damit ohne schädliche Röntgenstrahlung auskommt. Zusätzlich kann bei diesem Verfahren auf eine Kompression der Brust verzichtet werden.
Das bildgebende Instrument ist dabei ein halbkugelförmiges Becken, in dessen Rand Ultraschalltransducer eingelassen sind. Diese sind durch die halbkugelförmige Anordnung in der Lage, den Inhalt des Beckens von allen Seiten abzutasten.
Für die erfolgreiche Erkennung von Krebs wird eine Bildauflösung benötigt, die erst ab Ultraschall im Megahertz-Bereich erreicht werden kann. Zur Übertragung von Ultraschall mit so hohen Frequenzen wird Wasser als Kontaktmittel in das Becken gefüllt.

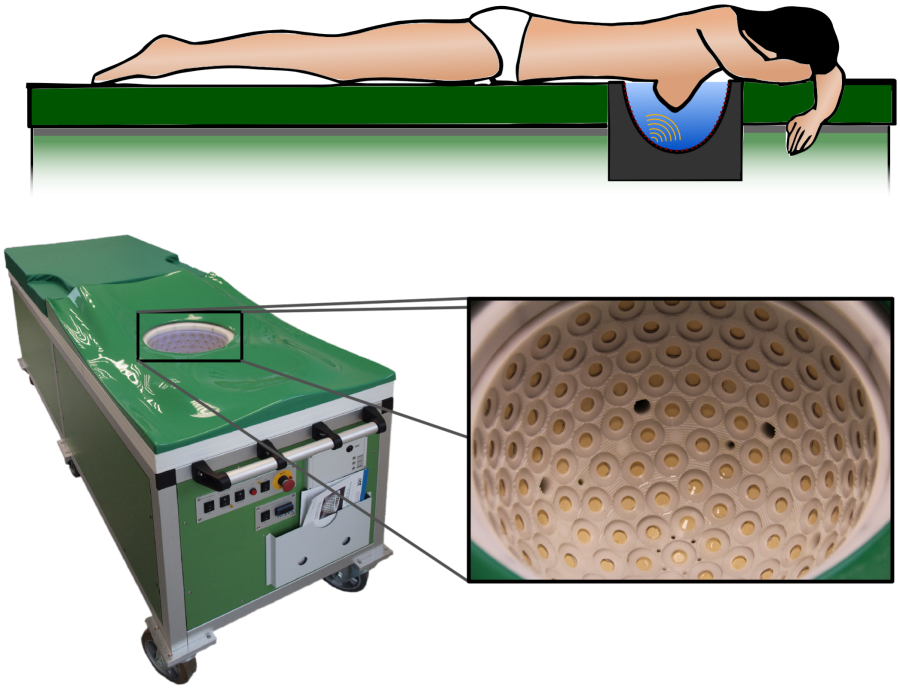


Abbildung 1: Schemazeichnung eines Scans (Oben), Gesamte Scanvorrichtung mit Liege (Unten links), Wasserbecken mit Ultraschalltransducern (Unten rechts)

Bei einer Untersuchung legt sich eine Patientin auf die Liege (Abb. 1), sodass ihre Brust in das Becken ragt und von den Transducern gescannt werden kann.
Die Transducer senden bei einer Messung nacheinander einen Ultraschallimpuls aus, der jeweils von allen anderen Transducern registriert wird. Aus diesen Daten wird dann ein dreidimensionales Bild rekonstruiert, auf welchem Tumore nachgewiesen werden können.

1.2 Motivation dieser Arbeit

Zur Erstellung der 3D-Daten wird ein wellenbasiertes Bildrekonstruktionsverfahren verwendet, welches das zu rekonstruierende Bild orthogonal aus der Sicht des Transducers benötigt. Dadurch muss in jedem Bildgebungsschritt eine Rotation des kartesisch gesampelten, dreidimensionalen Bilds vorgenommen werden.
Auch bei der Kombination der Daten der unterschiedlichen Transducer muss Bildrotation angewendet werden, da diese den Inhalt des Beckens aus einer unterschiedlichen Perspektive scannen.
Zum Drehen von zweidimensionalen Bildern existieren bereits Algorithmen, die sich auch auf dreidimensionale Bilder anwenden lassen, indem jede einzelne Ebene eines dreidimensionales Bild als zweidimensionales Bild interpretiert wird.

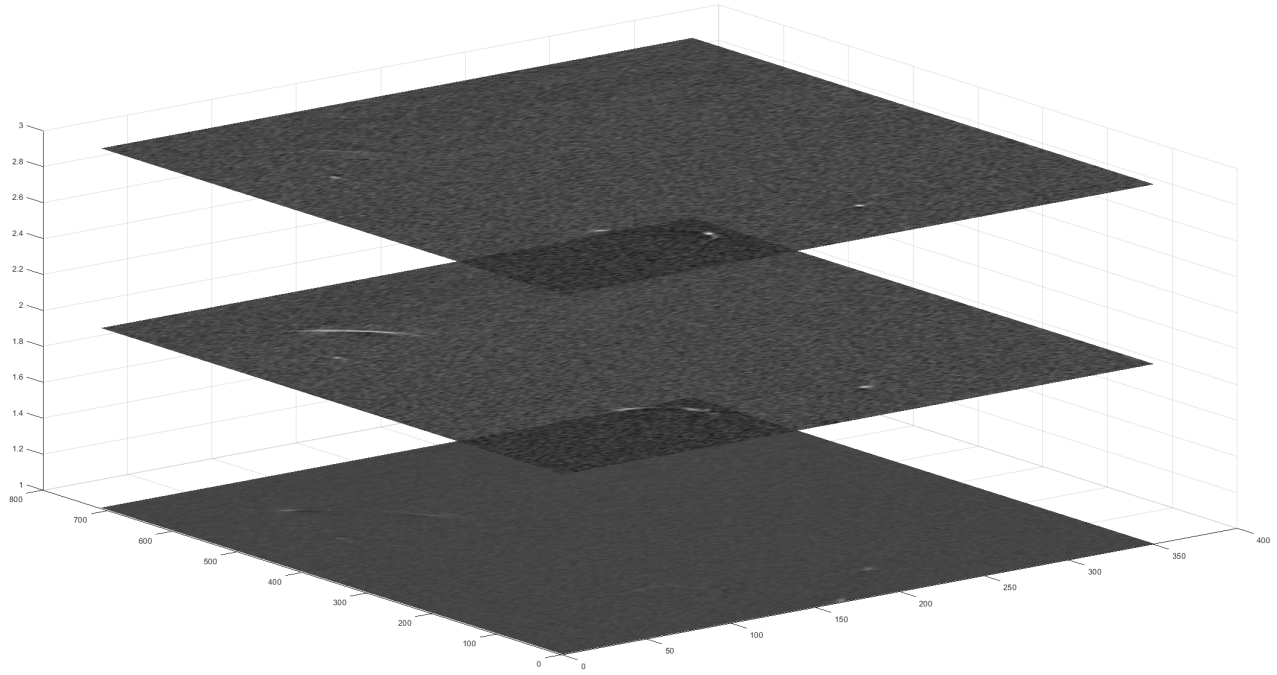


Abbildung 2: Mehrere Ebenen einer dreidimensionalen Aufnahme eines Drahts

Allerdings beruhen diese Standard-Rotationsverfahren auf Interpolation, was zwingend mit Datenverlust verbunden ist. Bei einer einzigen Rotation ist die Datenverfälschung noch vernachlässigbar, aber bei einer mehrfachen Anwendung summieren sich die Effekte auf.
Für viele Anwendungen von Bildrotation kann außerdem immer wieder auf das gleiche Bild als Datengrundlage zurückgegriffen werden, welches dann nur um unterschiedliche Winkel gedreht wird. Dadurch, dass sich die zu rotierenden Bilder beim verwendeten Bildrekonstruktionsverfahren zwischen den Rotationsschritten verändern, ist das hier allerdings nicht möglich.

Für die Rekonstruktion eines dreidimensionalen Bildes sind viele Rotationsschritte notwendig. Bei der Verwendung von Interpolation verliert das Bild mit jedem Rotationsschritt an Genauigkeit (Tab. 1).

Rotationen	0	100	5000
Rotiertes Bild			

Tabelle 1: Darstellung des Datenverlusts bei Bildrotation bei bikubischer Interpolation

Das führt vor allem an den Rändern, aber auch über das gesamte Bild verteilt, zu einem erheblichen Datenverlust (Abb. 3). Durch die Stärke des Datenverlusts ist ein interpolationsbasierter Ansatz für die Bildrekonstruktion nicht anwendbar, weswegen ein nicht-verlustbehafteter Algorithmus benötigt wird.

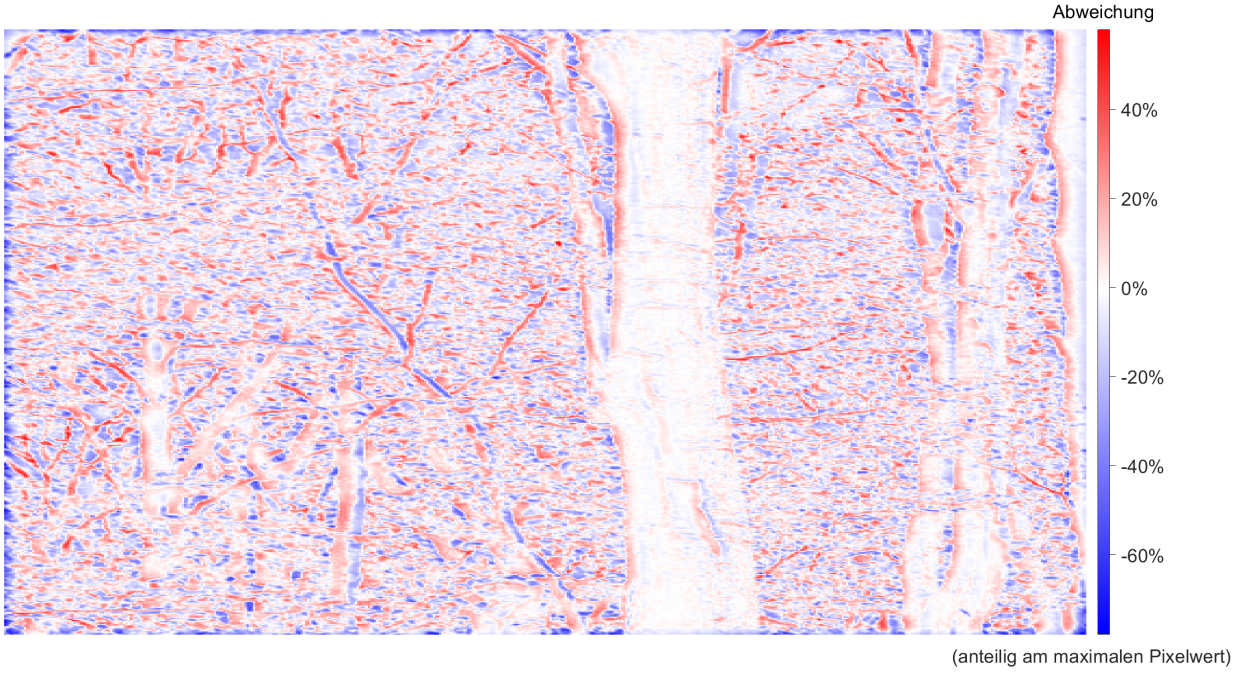


Abbildung 3: Differenzbild nach 5000 Rotationsschritten

Ein anderer Anwendungsfall für Bildrotation neben der Bildrekonstruktion ist der Aufbau und die Kalibrierung des Messsystems. So kann beispielsweise der Winkel zwischen zwei Transducern bestimmt werden, indem die Übereinstimmung der Messungen bei unterschiedlichen Rotationen gegeneinander berechnet wird. Hier kann für die Rotation immer wieder auf die gemessenen Daten zurückgegriffen werden, da diese nicht verarbeitet werden. Dadurch addieren sich zwar keine Interpolationsfehler auf, aber ein nicht-verlustbehaftetes Verfahren liefert trotzdem genauere Ergebnisse.

1.3 Sampling

Bei der Digitalisierung eines analogen Signals müssen immer wieder Messdaten in einem bestimmten zeitlichen Abstand aufgenommen werden. Wie stark der entstehende Graph das Ursprungssignal abbildet hängt stark von der Anzahl an Abtastungen ab.

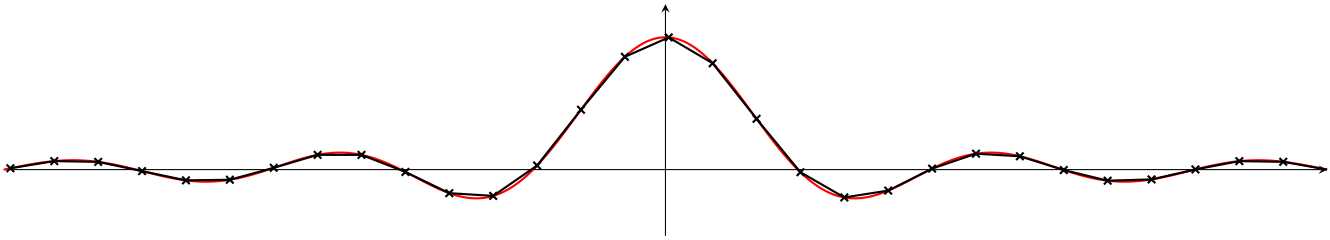


Abbildung 4: Abgetastetes Signal mit vielen Abtastpunkten

Bei einer häufigen Aufzeichnung von Werten in einem kurzen Zeitraum wird das Signal ähnlich abgebildet (Abb. 4). Wenn es allerdings zu wenig Messpunkte gibt, unterscheiden sich der entstehende Messgraph und das ursprüngliche Signal stark (Abb. 5).

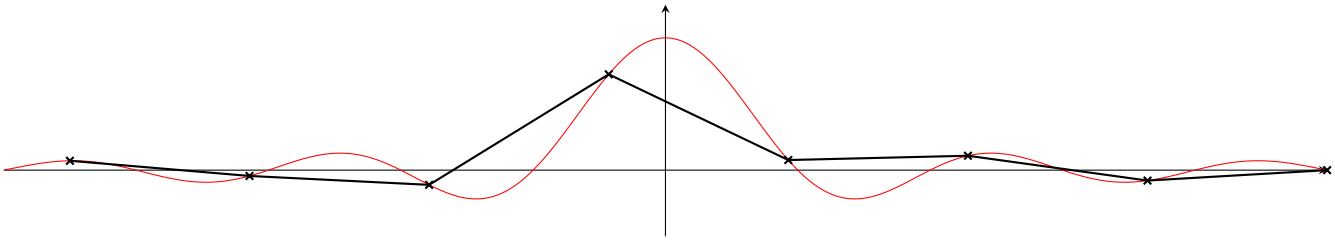


Abbildung 5: Abgetastetes Signal mit wenig Abtastpunkten

Für viele Anwendungsfälle von Sampling soll eine niedrige Abtastrate gewählt werden, bei der der Messgraph noch das Ursprungssignal widerspiegelt.

1.4 Aussage des Nyquist-Shannon-Abtasttheorems

Das Nyquist-Shannon-Abtasttheorem trifft Aussagen über die Rekonstruierbarkeit von Signalen [4]. Es bildet damit die Grundlage für die Digitalisierung von analogen Signalen und wird an unterschiedlichsten Stellen in der Signalverarbeitung und Nachrichtentechnik eingesetzt.
Es gibt die Voraussetzung für eine fehlerfreie Rekonstruktion eines gesampelten Signals an [4].

Satz 1 (Nyquist-Shannon-Abtasttheorem)**.** *Ein Signal, das aus Einzelfrequenzen zusammengesetzt ist, die kleiner oder gleich f_{max} sind, kann exakt rekonstruiert werden, wenn es mit einer Frequenz größer als $2 \cdot f_{max}$ abgetastet wird.*

Zum Verständnis des Abtasttheorems muss zunächst erläutert werden, wie sich ein Signal aus unterschiedlichen Frequenzen zusammensetzen kann.

1.5 Fourier-Transformation

Das Nyquist-Shannon-Abtasttheorem geht von Frequenzen aus, weil sich jedes zeitlich begrenzte Signal aus einer Reihe an Sinuswellen zusammensetzen lässt. Das gilt sowohl für periodische als auch für aperiodische Signale. Dieser Vorgang wird auch als Fourier-Synthese bezeichnet [5].
Die Fourier-Transformation transformiert ein Signal in ein Frequenz-Spektrum, das anzeigt, welche Frequenzen in diesem Signal enthalten sind (Abb. 6).

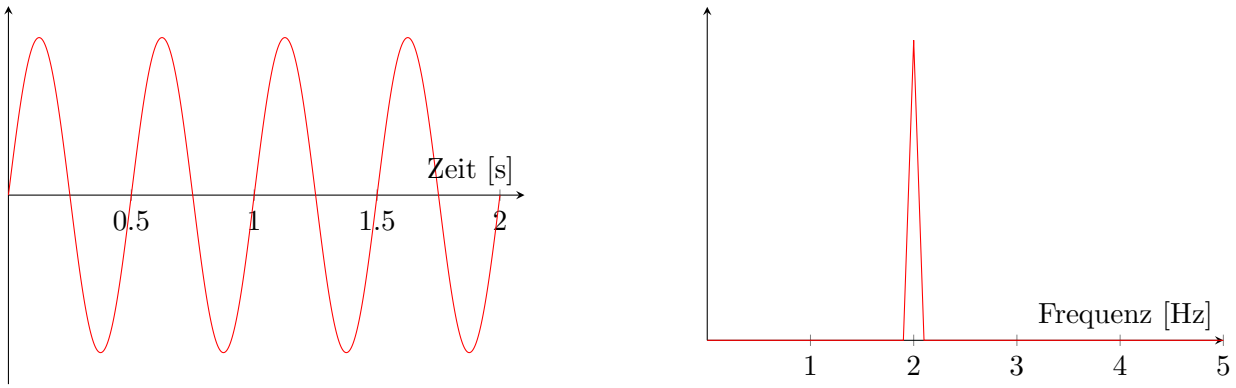


Abbildung 6: Signal (links) mit Frequenzspektrum (rechts)

Hier lässt sich links ein zeitlich begrenztes Signal erkennen, für das die enthaltenen Frequenzen ermittelt werden sollen. Dafür wird dieses mit der Fourier-Transformation transformiert. Rechts ist das resultierende Frequenzspektrum zu sehen. Dieses zeigt an, wie stark die einzelnen Frequenzen vorhanden sind.
Neben der Fourier-Transformation gibt es auch die inverse Fourier-Transformation, die ein Frequenzspektrum wieder in einen Zeitbereich wandelt.
Die Fourier-Transformation bildet damit die Grundlage für das Abtasttheorem, da sie zeigt, dass jedes Signal in ein Frequenzspektrum zerlegt werden kann.
Außerdem lässt sich diese Zerlegung nicht nur auf eindimensionale Daten anwenden, sondern ist auf beliebig viele Dimensionen erweiterbar. Deswegen ist die Fourier-Transformation auch für Bilder relevant.

1.6 Implikation des Nyquist-Shannon-Abtasttheorems

Ausgehend von der Fourier-Transformation kann jedes Signal in unterschiedliche Frequenzen zerlegt werden. Das Nyquist-Shannon-Abtasttheorem erlaubt nun eine Abschätzung der benötigten Abtastfrequenz auf Basis der vorkommenden Frequenzen.
Es kann ermittelt werden, welche Abtastfrequenz für eine verlustfreie Abtastung notwendig ist, indem mit der Fourier-Transformation die größte Frequenz bestimmt wird.
Außerdem spricht das Abtasttheorem von einer Rekonstruktion. Das bedeutet, dass tatsächlich das Ursprungssignal exakt wiederhergestellt werden kann. So wird nicht wie im Beispiel oben (Abb. 4) nur eine Näherung des Signals dargestellt, sondern das tatsächliche Signal. Das funktioniert aber logischerweise nur unter der Bedingung, dass das Signal nyquist-korrekt abgetastet wurde. Wenn das der Fall ist, kann davon ausgegangen werden, dass alle vorkommenden Frequenzen unter der Hälfte der Abtastfrequenz liegen müssen.
Dass das Abtasttheorem für eine vollständige Rekonstruktion gilt, ist auch für die Verlustfreiheit des Bildrotationsverfahrens entscheidend, weil eine vollständige Rekonstruktion keinen Datenverlust aufweist.

1.6.1 Beispiele zum Nyquist-Shannon-Abtasttheorem

Sollte beispielsweise eine Sinusschwingung mit $f_{max} = 1\,\text{Hz}$ mit einer Frequenz von $2{,}5\,\text{Hz}$ abgetastet werden, so ist diese Abtastung nyquist-korrekt und das Signal lässt sich exakt rekonstruieren (Abb. 7) [4].

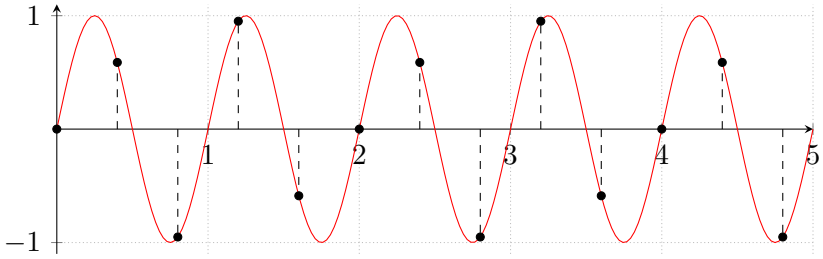


Abbildung 7: Nyquist-korrekt abgetastetes Signal

Bei einer Abtastung mit einer zu niedrigen Frequenz kann es allerdings dazu kommen, dass das Signal falsch rekonstruiert wird (Abb. 8). Dieses Phänomen nennt sich Aliasing. Es entsteht, weil bei der Rekonstruktion von Nyquist-Korrektheit ausgegangen wird.

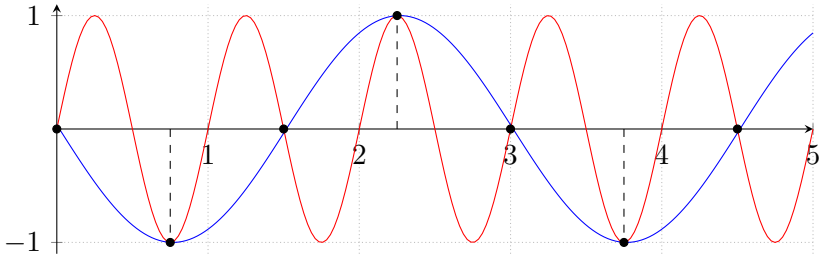


Abbildung 8: Abgetastetes Signal (rot) mit Aliasing bei Rekonstruktion (blau)

Außerdem ist die Beobachtung wichtig, dass die Rekonstruierbarkeit des Signals unabhängig von der Verschiebung der einzelnen Abtaststellen ist. Somit bleibt eine nyquist-korrekte Abtastung korrekt, wenn jeder Abtastpunkt um den gleichen Wert phasen-verschoben wird.

1.7 Das nicht-verlustbehaftete Rotationsverfahren

Durch die Kombination des Abtasttheorems und der Fourier-Transformation lässt sich ein nicht-verlustbehaftetes Bildrotationsverfahren umsetzen. Dafür wird die Ansammlung an Pixeln als periodisches Signal interpretiert.
Dann kann das Signal mittels Fourier-Transformation aus den einzelnen Pixeln, die die Abtastwerte darstellen, rekonstruiert werden. Zur Berechnung der Pixel des rotierten Bildes müssen die Werte von Pixeln berechnet werden, die zwischen den Pixeln des ursprünglichen Bildes liegen. Dabei wird normalerweise Interpolation angewendet. Für die nicht-verlustbehaftete Rotation wird allerdings auf das rekonstruierte Signal zurückgegriffen.
Dadurch, dass das rotierte und das ursprüngliche Bild die gleiche Anzahl an Pixeln haben, bleibt die Abtastfrequenz des Signals die gleiche. Die Abtaststellen sind lediglich unterschiedlich stark phasenverschoben. Nach dem Nyquist-Shannon-Abtasttheorem bleibt das Signal allerdings rekonstruierbar, was verhindert, dass Daten verloren gehen. So bleibt das dahinterliegende System bei jeder Rotation gleich und das Bild wird nicht verfälscht.
Das hat außerdem zur Folge, dass bei diesem Verfahren kein Rand an das rotierte Bild angefügt werden muss. Bei Standardbildinterpolation entsteht bei einem Rotationswinkel, der nicht durch 90 Grad teilbar ist, ein schwarzer Rand, damit das Bild weiterhin rechteckig bleibt. Beim verlustfreien Verfahren ist durch die Interpretation als Signal allerdings gar kein Rand des Bildes vorhanden. Die Ränder des Bildes hängen zusammen. Das führt dazu, dass sich ein Pixel nicht aus dem Bild herausdrehen kann, weil er auf der anderen Seite wieder in das Bild eintritt.
Obwohl das Verfahren in der Theorie nicht verlustbehaftet ist, ist es in der Praxis nicht möglich, eine komplett verlustfreie Implementierung herzustellen. Die Genauigkeit, mit der die Pixelwerte gespeichert werden, ist begrenzt. Somit entsteht bei der Speicherung des rotierten Bildes immer eine kleine Abweichung. Das lässt sich allerdings nicht verhindern, und das Verfahren ist trotzdem weitaus weniger verlustbehaftet als bei einer Standardbildinterpolation.

1.8 Aufgabenstellung

Dieses Verfahren benötigt durch die vielen Berechnungen deutlich mehr Rechenleistung als ein Standard-Bildinterpolations-Verfahren. Der Matlab-Prototyp, der die Rotation implementiert, läuft auf der CPU und ist deutlich zu langsam für den Anwendungsfall eines Bildes mit der Seitenlänge von 250 Pixeln. Durch die Struktur des Verfahrens lässt sich die Bildrotation gut parallelisieren, weil die Werte der neuen Pixel unabhängig voneinander bestimmt werden können. Somit erscheint eine GPU-basierte Implementierung des Verfahrens vielversprechend.
Ziel dieser Arbeit ist es, eine GPU-Implementierung des bereits existierende Matlab-Prototypen zu erstellen und diese auf Laufzeit und Genauigkeit zu untersuchen.

2 Material und Methoden

Vor der Planung der konkreten Software-Architektur, ist es notwendig, unterschiedliche grundsätzliche Entscheidungen zu treffen. Das beinhaltet unter anderem die Wahl der Implementierungsart und die Definition von Metriken zur Evaluierung der Implementierung.
Neben der Planung der Methoden wird hier auch das bereits existierende Material aufgeführt.

2.1 Matlab

Matlab ist eine Entwicklungsumgebung, die sich vor allem für Datenverarbeitung und statistische Analyse eignet.
Der bereits existierende Prototyp ist in Matlab implementiert und auch die Programme, die die Bildrotation einbinden sollen sind in Matlab geschrieben. Die GPU-Implementierung der Bildrotation muss somit die Möglichkeit haben, aus einem Matlab-Programm heraus aufrufbar zu sein.

2.2 Der bereits vorhandene Prototyp

Der vollständige Codes des Matlab-Prototyps ist im Anhang einsehbar.
Der Matlab-Prototyp implementiert das Verfahren der nicht-verlustbehafteten 2D-Bildrotation für die CPU. Als Parameter müssen ein Bild und die Koordinaten der Pixel des neuen Bildes im Verhältnis zum ursprünglichen Bild übergeben werden. Die Funktion führt die Berechnungen aus, die notwendig sind, um das rotierte Bild zu erhalten. Dieses wird dann zurückgegeben.

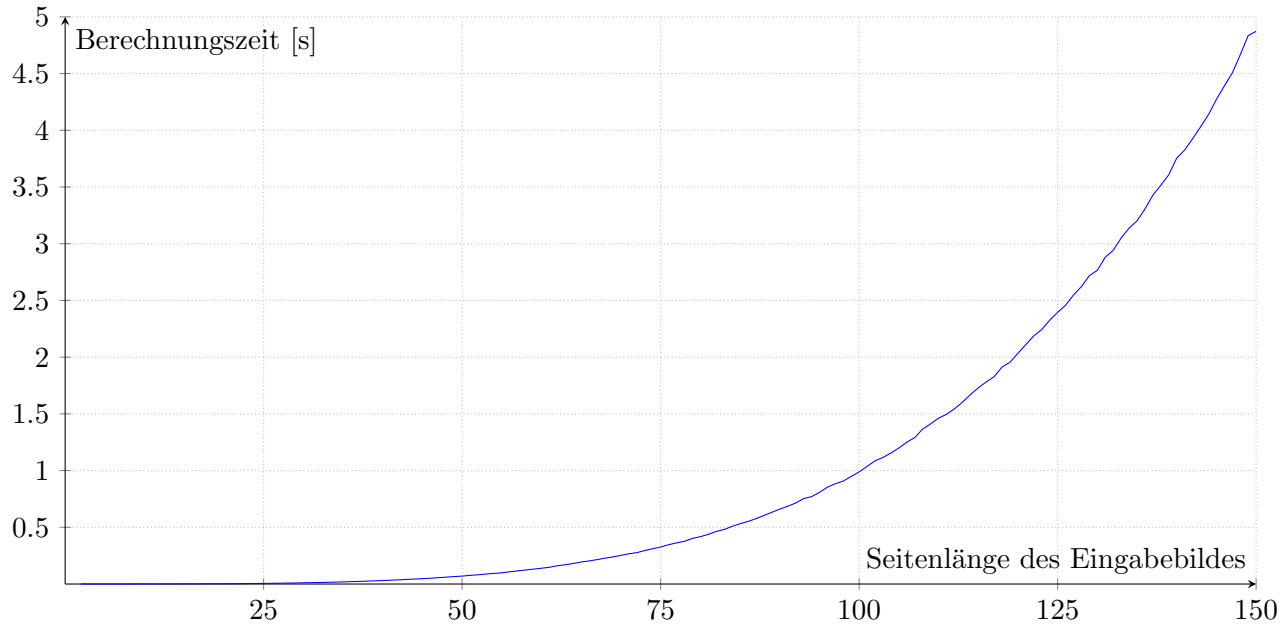


Abbildung 9: Laufzeit des Matlab-Prototyp (Median über 10 Versuche)

Für die Berechnung jedes einzelnen Pixels müssen alle anderen Pixel miteinbezogen werden, was dazu führt, dass der Code eine asymptotische Laufzeit von $\mathcal{O}\left(n^2\right)$ für n Pixel hat (Abb. 9).
Auch der Speicherverbrauch steigt quadratisch zur Bildgröße an. Dadurch übersteigt der benötigte Speicher auch schon bei vergleichsweise kleinen Bildern die maximale RAM-Kapazität. Damit kein *Out-Of-Memory-Error* entsteht, wird das Problem mittels Rekursion in zwei Teilprobleme aufgespalten, falls der Speicherverbrauch zu groß wäre. Die beiden Teilprobleme können dann mit niedrigerem Speicherverbrauch nacheinander berechnet werden und die Ergebnisse werden zusammengeführt.

2.3 Implementierungsmöglichkeiten

Zur Implementierung der Bildrotation auf der GPU müssen zunächst die unterschiedlichen Möglichkeiten einer Implementierung betrachtet werden.

2.3.1 GPU-Coder

Matlab unterstützt mit dem GPU-Coder direkt eine Methode, um Matlab-Code auf die GPU zu übertragen [6, 7]. Dabei wird eine Matlab-Funktion für die GPU übersetzt und kann aus Matlab heraus aufgerufen werden.

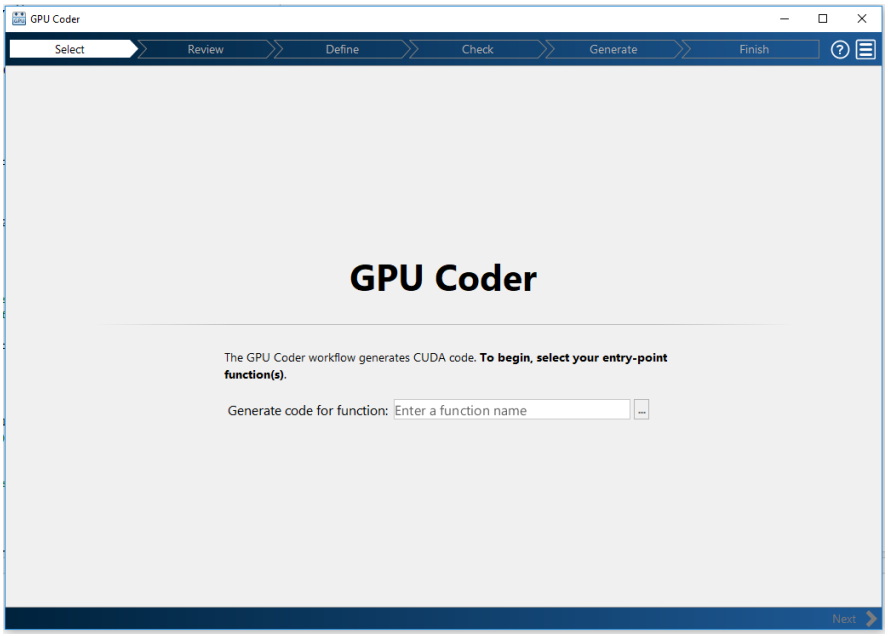


Abbildung 10: Der GPU-Coder

Dieses Verfahren bietet den Nachteil, dass die Parallelisierung automatisiert vorgenommen wird. Dadurch lässt sich nicht genau steuern, wie das Verfahren implementiert wird oder auf welche Weise die Parallelisierung und damit die Effizienzsteigerung erfolgt. Allerdings bietet der GPU-Coder auch die Möglichkeit, einfach Matlab-Code auf die GPU zu übertragen, ohne CUDA-Code schreiben zu müssen.

2.3.2 CUDA

Neben dem GPU-Coder gibt es auch die Möglichkeit, den CUDA-Code direkt selbst zu implementieren. CUDA ist eine NVIDIA-spezifische Programmierschnittstelle, die es ermöglicht, Programme zu schreiben, welche auf GPUs ausgeführt werden. CUDA ist C-ähnlich und spricht die einzelnen Komponenten der GPU direkt an.
Dieses Vorgehen hat zum Vorteil, dass eine filigranere Optimierung möglich ist. Dabei kann die Parallelisierung selbst gewählt werden und Hardware-spezifische Eigenschaften der GPU können beachtet werden.

2.4 Einbindung von CUDA in Matlab

Da der selbst geschriebene CUDA-Code nicht direkt mit Matlab kompatibel ist, muss ein Interface verwendet werden, um diesen trotzdem aus einem Matlab-Skript aufrufen zu können.

2.4.1 Einbindung durch mexcuda

Matlab unterstützt mit *mexcuda* auch das direkte Einbinden von selbst geschriebenen CUDA-Kernels [8]. Dabei sind Matlab und der CUDA-Code allerdings nicht klar voneinander getrennt (Abb. 11).

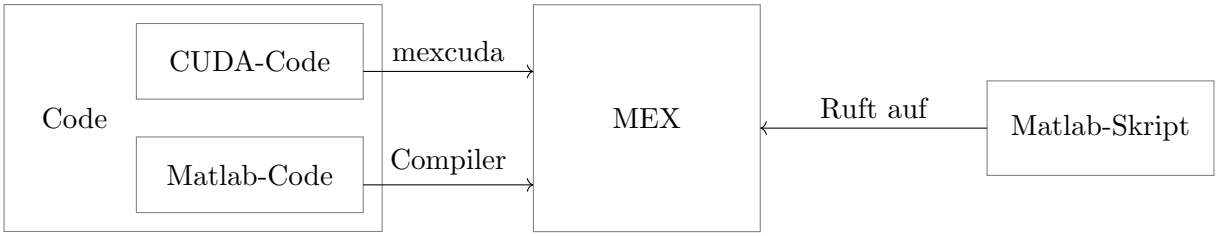


Abbildung 11: Schema der Einbindung mittels mexcuda

Das hat zur Folge, dass es starke Abhängigkeiten gibt, was eine genaue Optimierung erschwert.

2.4.2 Unabhängige Einbindung durch DLL

Es gibt auch die Möglichkeit, Matlab und den CUDA-Code vollständig voneinander zu trennen. Matlab unterstützt das Einbinden externer Programme mit dem MEX-Interface (Matlab executable). Dabei kann ein C oder C++ Programm erstellt werden, welches dann zu einer MEX kompiliert wird und in Matlab verwendet werden kann.
Damit kann auch ein CUDA-Kernel eingebunden werden, wenn dieser zuerst zu einer DLL kompiliert wird [9]. Diese DLL wird unabhängig von Matlab erstellt und dann durch das MEX-Interface von Matlab angesprochen (Abb. 12).

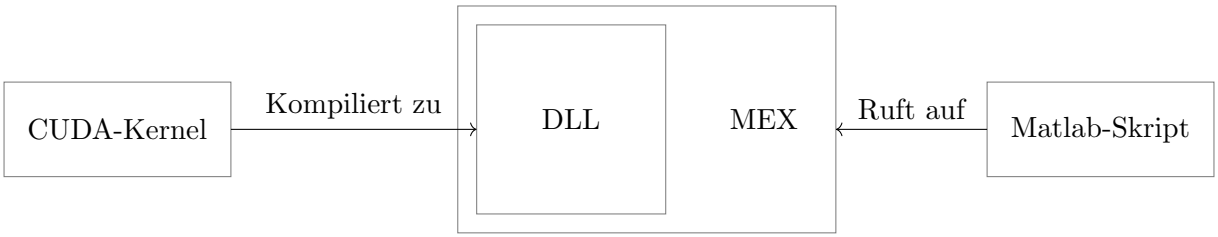


Abbildung 12: Schema der Einbindung durch eine DLL

Bei diesem Ansatz sind der Matlab- und der CUDA-Code unabhängig. Das führt dazu, dass es einfacher ist, den CUDA-Code zu optimieren. Deshalb wurde dieser Ansatz verwendet.

2.5 Evaluierungsmetriken - Profilingmöglichkeiten

Zur Testung eines implementierten Ansatzes ist es notwendig, Metriken zur Evaluierung zu bestimmen. Damit können die einzelnen Ansätze verglichen und beurteilt werden.

Die Effizienz der Implementierung ist eine wichtige Metrik, um zu überprüfen, ob das gesetzte Ziel erfüllt wurde. Neben der Größe der Laufzeit an sich, ist es wichtig, auch einzelne Teile des Codes evaluieren zu können. Dazu gehört zum Beispiel die Zeit- oder Latenzmessung für bestimmte Codezeilen. Dieses Vorgehen nennt sich Profiling.

2.5.1 Profiling mittels Zeitmessung

Eines der einfachsten Verfahren zum Profiling ist das Messen der Laufzeit des Codes. Das lässt sich in diesem Fall einfach über die Matlab-Standard-Library durchführen. Damit können schon Ansätze verglichen werden. Allerdings liefert dieses Verfahren keine tieferen Einblicke für eine weitere Optimierung.

2.5.2 nvprof

Das *nvprof*-Profiling-Werkzeug ermöglicht es, Code-spezifische Profiling-Daten zu sammeln [10]. Diese Daten werden in der Kommandozeile erhoben und sind je nach Einstellung sehr ausführlich und genau. Dieses Tool kann somit gut eingesetzt werden, um ein Profiling durchzuführen.

2.5.3 Der NVIDIA-Visual-Profiler

Der NVIDIA-Visual-Profiler ermöglicht es, CUDA-Kernels bezüglich ihrer Performance zu profilen [11]. Dieser stellt das Profiling im Gegensatz zu *nvprof* visualisiert dar (Abb. 13).

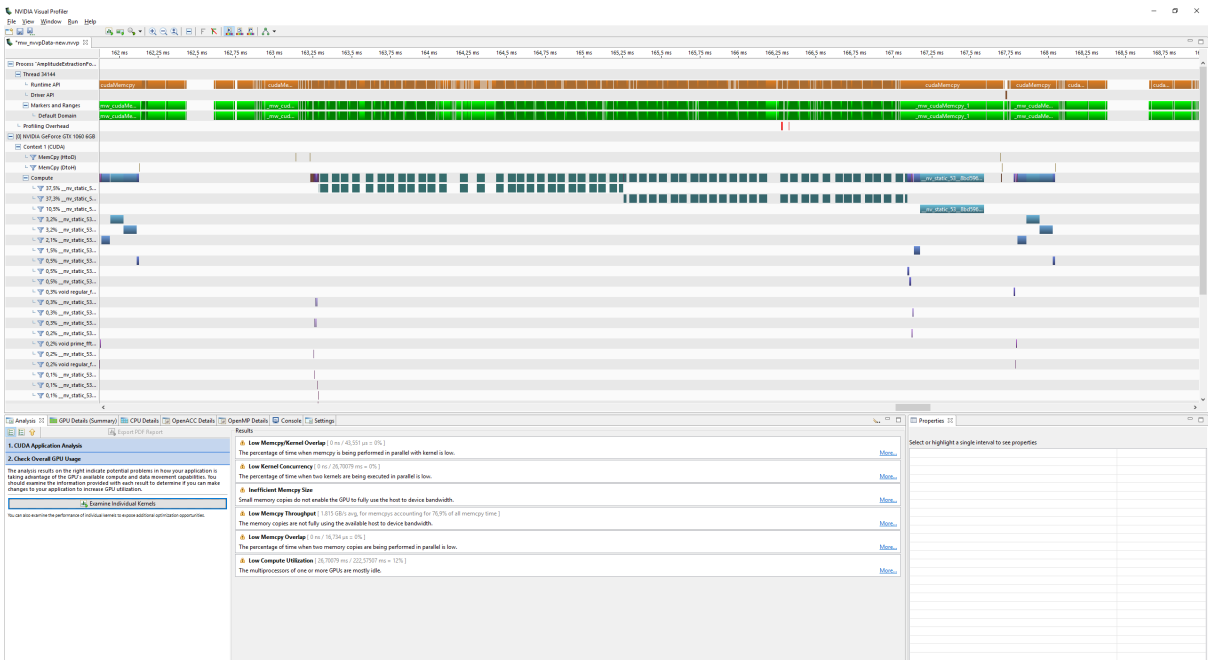


Abbildung 13: Der NVIDIA-Visual-Profiler

Dadurch können die unterschiedlichen Bestandteile der Implementierung auf Effizienz untersucht werden und Optimierungsmöglichkeiten können gefunden werden.

2.6 Evaluierungsmetriken - Genauigkeitsprüfung

Neben der Effizienz ist die Genauigkeit der Ergebnisse eine wichtige Metrik zur Beurteilung der Implementierung. Dafür kann jeder Ausgabepixel mit den bereits geprüften Ausgaben der CPU-Implementierung verglichen werden und für jeden Pixel wird die Abweichung gespeichert. Mit dieser Liste können dann die maximale und die durchschnittliche Abweichung beurteilt werden.

2.7 Hardware

Die verwendete Hardware war bereits gegeben. Nur für die GPU wurden unterschiedliche Modelle getestet und nach Performance bewertet. Die Profiling-Daten wurden auf folgendem System erhoben:

- Desktop-PC (OS: Windows 10)
- Prozessor Intel(R) Core(TME) i7-8700 CPU @ 3.2 GHz, 6 Kerne
- RAM: 32 GB
- GPU: NVIDIA GeForce GTX 1060, 6 GB
- Matlab-Version: 2021a
- CUDA-Version: 11.6.44

3 Softwarearchitektur

Für die Übertragung der CPU-Implementierung auf die GPU, ist es notwendig, die Architektur der Implementierung zu verändern. Die CPU ist mit wenigen Cores ausgestattet [12]. Dadurch werden die meisten Berechnungen auf der CPU seriell ausgeführt. So werden zum Beispiel im konkreten Anwendungsfall alle Schritte zur Berechnung des rotierten Bilds auf der CPU hintereinander ausgeführt, wobei jeder Schritt vom vorherigen abhängig ist.
Eine GPU verfügt hingegen über eine große Anzahl an Cores, von denen jeder parallel zusätzlich mehrere Threads ausführt [12].
Dadurch ist die GPU deutlich stärker für Parallelisierung ausgelegt als die CPU. Allerdings erfordert das effiziente Nutzen der Parallelisierungsmöglichkeit eine Änderung in der Software-Architektur. Die Schritte, die parallelisiert ausgeführt werden sollen, müssen im Gegensatz zur CPU-Software-Architektur unabhängig voneinander sein. Somit muss ein Konzept für eine unabhängige Parallelisierung ausgearbeitet werden, indem ein parallelisierbarer Arbeitsschritt gefunden wird.
Außerdem ist auch nicht jeder Algorithmus darauf ausgelegt, auf einer GPU implementiert zu werden. In manchen Fällen ist eine serielle Vorgehensweise so stark im Algorithmus verankert, dass keine Parallelisierung möglich ist. Bei der Bildrotation sind viele Berechnungen voneinander unabhängig, weswegen überhaupt eine GPU-Implementierung durchgeführt wird.

3.1 Parallelisierung

3.1.1 Parallelisierung über Bildebenen

Einerseits kann die Parallelisierung bei den Bildebenen ansetzen. So wäre es möglich, jede einzelne Bildebene seriell zu berechnen, aber das Berechnen der Bildebenen zu parallelisieren (Abb. 14).

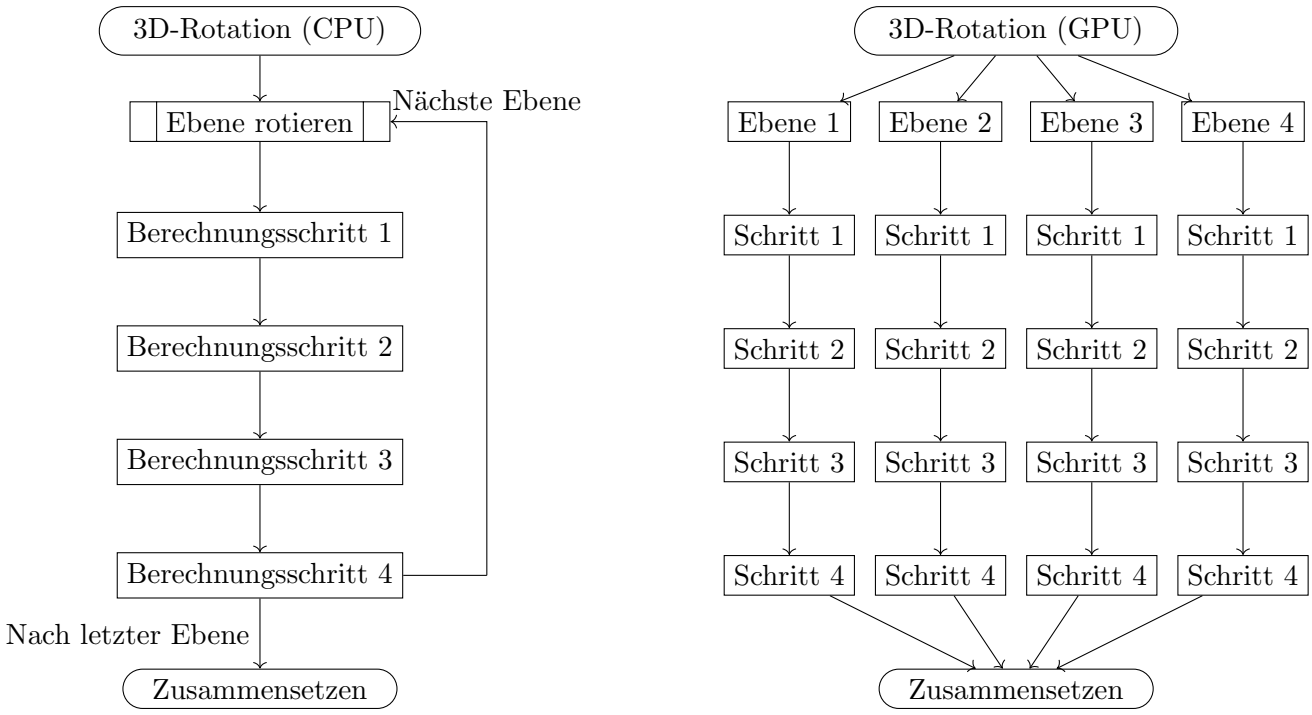


Abbildung 14: Vergleich Software-Architektur CPU und GPU mit Ebenen-Parallelisierung

Diese Parallelisierung lässt sich außerdem gut mit mehreren GPUs umsetzen, da jede einzelne Bildebene einfach an eine GPU übergeben werden kann.

3.1.2 Parallelisierung über Pixel

Da die Berechnungen der einzelnen Pixel innerhalb des rotierten Bildes nicht voneinander abhängen, kann auch diese Berechnung parallelisiert werden. Dabei wird parallel in jedem GPU-Thread ein Pixel berechnet und das zweidimensionale Bild wird erst am Ende zusammengesetzt (Abb. 15).

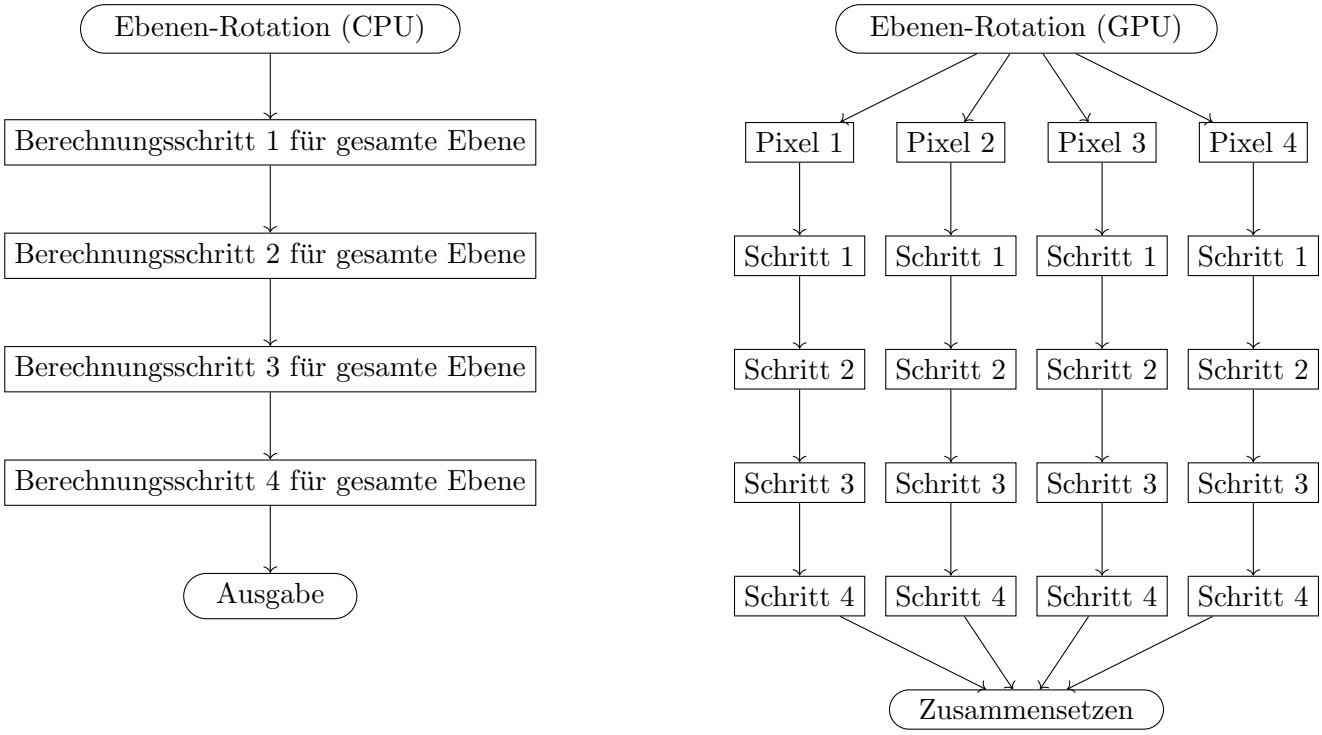


Abbildung 15: Vergleich Software-Architektur CPU und GPU mit Pixel-Parallelisierung

Diese Parallelisierung erfordert eine stärkere Änderung der Software-Struktur als die Parallelisierung über Bildebenen, da hier die kleine Berechnungseinheit parallelisiert wird und somit kein großer Prozess mehr seriell abläuft.
Außerdem erfordert die unabhängige Berechnung jedes Pixels eine Mehrfachberechnung einiger Parameter, die beispielsweise für alle Pixel innerhalb einer Spalte des Bildes gleich wären.

3.2 Portierung

Zur Übertragung des Codes auf die GPU müssen neben der Planung der Parallelisierung noch weitere Problematiken untersucht werden. Diese entstehen entweder durch den Parallelisierungsansatz oder verhalten sich auf der GPU anders als auf der CPU.

3.2.1 Speicherproblem

Bei der Wahl der Parallelisierungsmöglichkeit ist außerdem die Problematik zu beachten, dass der Speicherbedarf asymptotisch mit $\mathcal{O}\left(n^2\right)$ ansteigt (Kap. 2.2).
Diese Problem lässt sich auf der GPU nicht so einfach lösen wie auf der CPU. Die Rekursions-Lösung lässt sich nicht auf die GPU übertragen, also muss eine andere Möglichkeit gefunden werden, den Speicherbedarf zu senken, damit auch große Bilder verarbeitet werden können.
Der Grund für den hohen Speicherbedarf ist, dass für die Berechnung jedes einzelnen Pixels jeder andere Pixel betrachtet werden muss und so für jede Pixelkombination ein Wert bestimmt wird. Für den endgültigen Wert eines einzelnen Pixels werden dann alle Werte, die bei der Betrachtung der anderen Pixel berechnet wurden, addiert. Das Speichern der Werte für die Ergebnispixel würde eigentlich nur so viel Speicherplatz verbrauchen wie das Speichern des Eingabebildes, da diese dieselbe Größe haben. Allerdings müssen alle berechneten Werte zwischengespeichert werden, so dass ein dreidimensionales Array entsteht, welches n^2 Werte speichern muss und nicht nur n wie die Ausgabe.
Bei einer Parallelisierung über die Bildebenen bleibt dieses Problem bestehen. Wenn allerdings eine Parallelisierung über die Pixel erfolgt, müssen die einzelnen Werte nicht alle zwischengespeichert werden, sondern können direkt auf das Ausgabearray addiert werden. So ist der Speicherverbrauch nur der für die Speicherung der Eingabearrays und des Ausgabearrays.
Dieses Konzept löst das Speicherproblem für die GPU vollständig und beschränkt den Speicherverbrauch auf das absolute Minimum, das bei jedem Ansatz gebraucht wird: Die Speicherung von Input und Output.

Da sich das Speicherproblem nur so umgehen lässt, muss eine Parallelisierung über die Pixel erfolgen. Auf diese Parallelisierung konzentriert sich diese Arbeit. Es ist logischerweise trotzdem möglich, eine Parallelisierung über die Bildebenen vorzunehmen. Allerdings eignet sich diese vor allem für einen Multi-GPU-Ansatz. Dieser erfordert keinen anderen Quellcode, da die Implementierung der Bildrotation für zweidimensionale Bilder vorgenommen wird und dreidimensionale Bilder somit sowieso zerlegt werden müssen. Der Code ist also mit und ohne Parallelisierung über die Bildebenen anwendbar.

3.2.2 Zusammenführen der berechneten Daten

Für jeden Pixel werden bei einer Parallelisierung über Pixel mehrere Einzelberechnungen parallel durchgeführt. Diese Einzelberechnungen müssen zur Generierung des Ausgabebildes wieder zusammengeführt werden. Dieser Schritt lässt sich nicht in die Parallelisierung integrieren, weil hier die Daten aus jedem parallelen Thread addiert werden müssen.
Es können aber auch nicht einfach alle einzelnen Werte aus den Threads zurückgegeben werden, da dies wieder Speicherprobleme hervorrufen würde. Dieses Problem lässt sich umgehen, indem eine Struktur genutzt wird, bei der Threads mit sogenannten *atomic functions* zusammenarbeiten, obwohl sie parallel laufen [13]. Dabei schreiben mehrere Threads in eine gemeinsame Variable, ohne, dass Konflikte entstehen.

3.3 Float und Double

Beim Entwickeln der GPU-Software muss entschieden werden, welcher Datentyp zum Speichern von Reellen Zahlen verwendet werden soll. Dabei gibt es float und double, die sich in der Anzahl an Bits zum Speichern einer Zahl und somit in der Genauigkeit unterscheiden, mit der eine Zahl gespeichert werden kann [14].
Für das Verfahren ist eine möglichst hohe Genauigkeit von Vorteil. Allerdings bedeutet höhere Präzision auch mehr Rechenzeit. Insbesondere auf GPUs unterstützen doubles nicht alle Funktionen und können je nach Baureihe deutlich langsamer sein (Abb. 16).

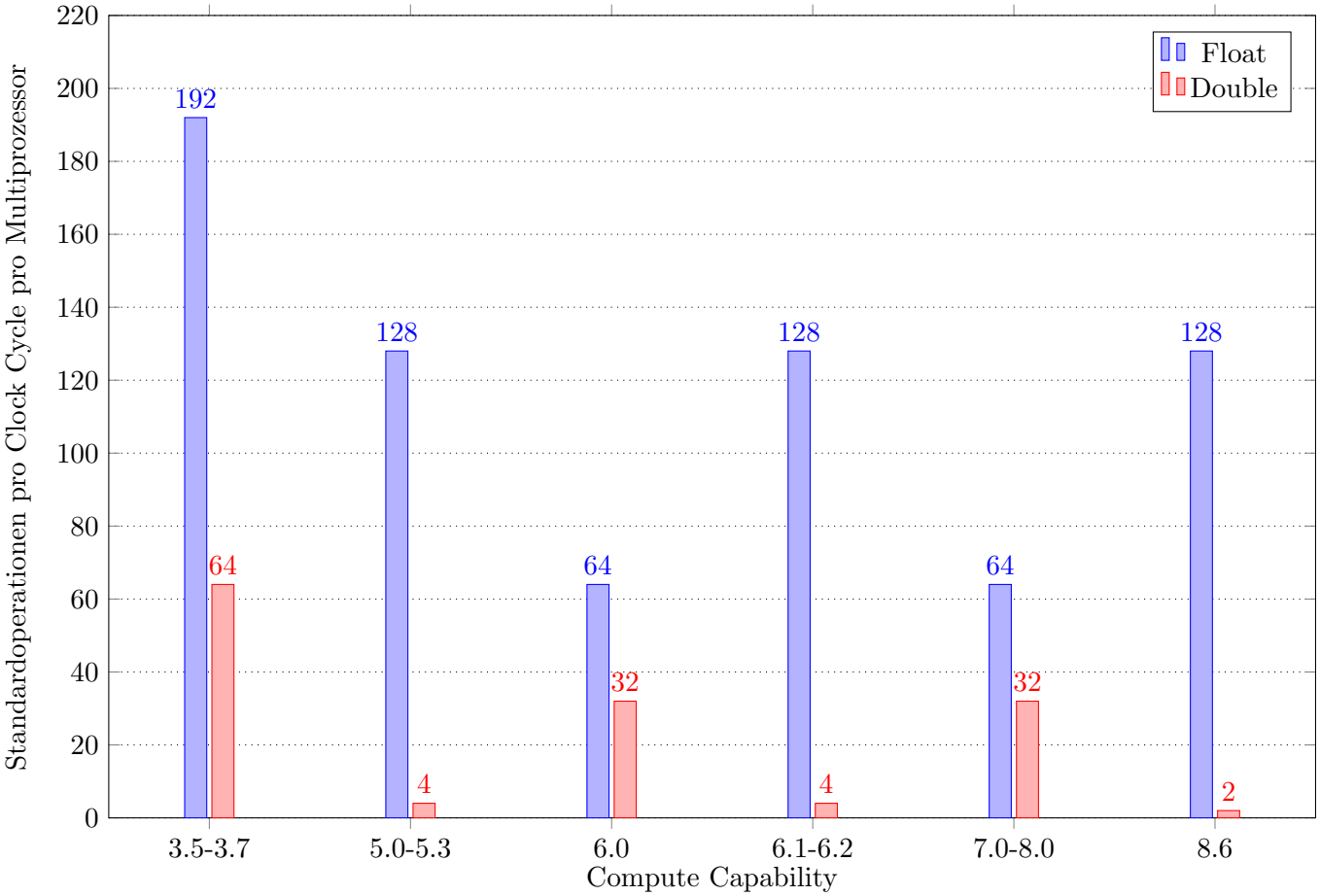


Abbildung 16: Vergleich des Rechendurchsatzes von doubles und floats bei NVIDIA-GPUs [15]

Durch den großen Performance-Unterschied zwischen Doubles und Floats, der insbesondere bei der verwendeten GPU (Compute Capability 6.2) vorliegt, ist es eher unwahrscheinlich, dass sich eine Implementierung mit Doubles lohnt.
Allerdings unterscheidet sich der Code für Doubles und Floats nur wenig. Das ermöglicht es, zwei unterschiedliche Versionen zu erstellen, von denen eine schneller und eine präziser ist. Dafür wurde die Einstellung, ob Floats oder Doubles verwendet werden an einer Stelle im Code gesetzt, so dass eine Umstellung ohne Probleme durchführbar ist.

4 Implementierung mittels GPU-Coder

4.1 Erster Ansatz: Naive Implementierung

4.1.1 Konzept

Der erste Ansatz ist die Verwendung der native Implementierung des Matlab-Prototyps (Kap. 8.1) für den GPU-Coder.
Dabei müssen einige Änderungen vorgenommen werden, um den Code GPU-kompatibel zu machen. Die Rekursion wird entfernt, da diese vom GPU-Coder nicht unterstützt wird. Damit bleibt das Speicherproblem vorerst bestehen. Sollte sich die Implementierung mittels GPU-Coder lohnen, muss noch ein Weg gefunden werden, dieses zu lösen.
Außerdem unterstützt der GPU-Coder im Gegensatz zu nativem Matlab-Code keine Multiplikation von unterschiedlich großen, aber trotzdem kompatiblen Matrizen. Somit müssen einige Teile des Codes wie zum Beispiel Zeile 80-82 leicht verändert werden, damit die Matrizen tatsächlich die gleich Größe haben.

4.1.2 Laufzeitdaten

Die vom GPU-Coder erstellte MEX wird für unterschiedliche Problemgrößen ausgeführt und die Laufzeiten werden gemessen.

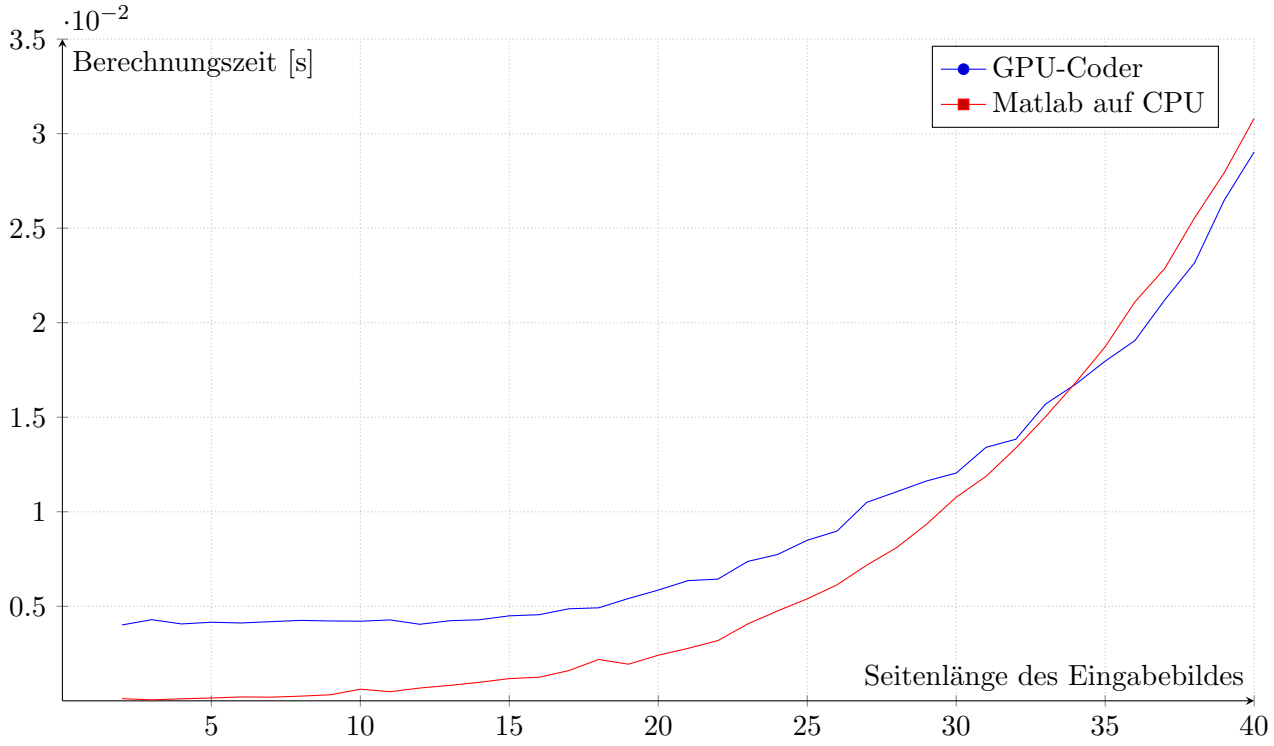


Abbildung 17: Laufzeiten des GPU-Coder-Programms (Median über 10 Versuche)

Es lässt sich erkennen, dass der GPU-Code einen deutlich größeren Overhead als der CPU-Code hat, da der GPU-Code für kleine Fälle weniger performant ist und dann an Effizienz gewinnt.
Trotzdem liefert dieser erste Ansatz für eine GPU-Implementierung keinen erkennbaren Speedup.

4.1.3 Profiling

Die Durchführung eines Profilings mit dem NVIDIA-Visual-Profiler ermöglicht es, genauer herauszufinden, an welchen Stellen es noch Optimierungspotential gibt.

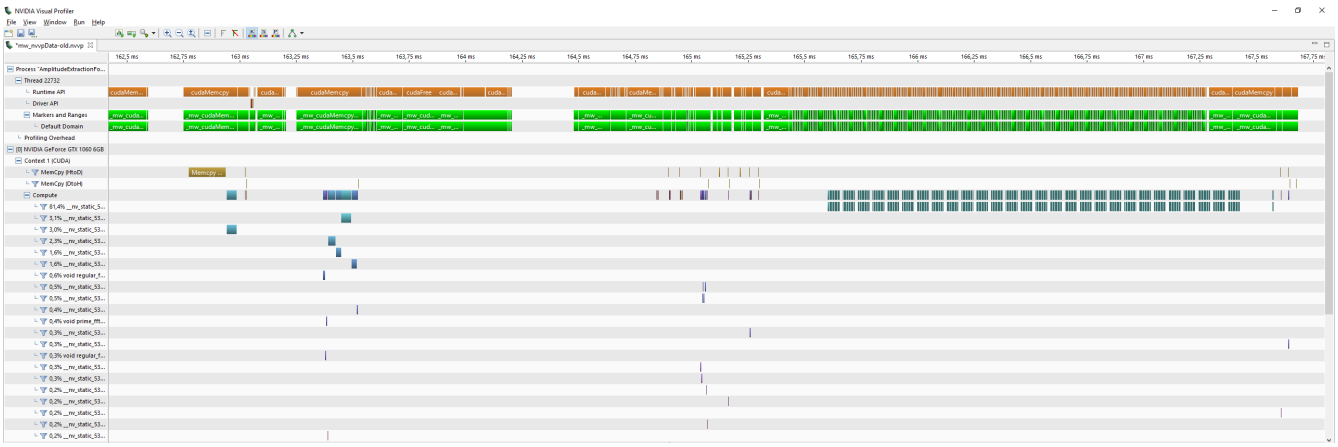


Abbildung 18: Profiling des ersten Ansatzes im NVIDIA-Visual-Profiler

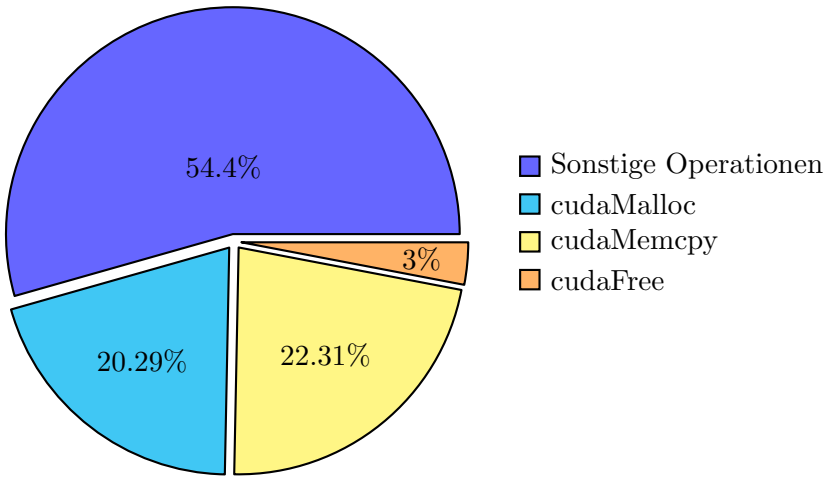


Abbildung 19: Laufzeitanteile der Speicher-Operationen

Im Profiling lässt sich erkennen, dass die GPU nicht optimal ausgenutzt wird. Einerseits zeigt der NVIDIA-Visual-Profiler an, dass selten Kernels parallel laufen und dass die Multiprozessoren der GPU nur wenig ausgelastet sind.
Andererseits ergibt eine statistische Analyse der Profiling-Daten, dass die GPU fast zur Hälfte der Zeit mit Speicher-Operationen beschäftigt ist (Abb. 19). Dazu zählt zum Beispiel das Allokieren von Speicherplatz oder das Kopieren von Daten.
Der Laufzeitanteil an Speicher-Operationen sollte möglichst niedrig gehalten werden, da diese nicht parallelisiert ausgeführt werden und nichts mit den eigentlichen Berechnungen zu tun haben. Dass die Speicher-Operationen so präsent sind, erklärt den großen Overhead und den fehlenden Speedup.

4.2 Zweiter Ansatz: Anpassung der Speicher-Operationen

4.2.1 Konzept

Ausgehend von den Ergebnissen des ersten Ansatzes sollte unnötige Speicherallokation und -verwendung möglichst vermieden werden. Für die Umsetzung dieser Anforderungen mit dem GPU-Coder, muss der zugrundeliegende Matlab-Code verändert werden.
Bei einer näheren Untersuchung des Codes fällt auf, dass an vielen Stellen *repmat*-Operationen verwendet werden, die es ermöglichen, eine Matrix oder einen Vektor in eine Dimension zu vervielfältigen. Allerdings muss die entstehende Matrix dann gespeichert werden, obwohl eigentlich ein Großteil der gespeicherten Informationen redundant ist.
Für die Matlab-Implementierung ist es effizienter, die Matrizen oder Vektoren zu vervielfältigen, um dann zum Beispiel eine Multiplikation zweier Matrizen auszuführen. Das liegt daran, dass Matlab darauf ausgelegt ist, solche Operationen schnell auszuführen. Für den GPU-Coder scheint es allerdings sinnvoller für das Sparen von Speicherplatz auf die *repmat*-Operationen zu verzichten und stattdessen mit Schleifen zu arbeiten.
Außerdem können Variablen, die die gleiche Größe wie anderen Variablen haben, wiederverwendet werden, um unnötige Speicherallokationen zu vermeiden.
Entsprechend dieser Ansätze zum Vermeiden von Speicher-Operationen wird der Matlab-Prototyp umgeschrieben (Quellcode im Anhang einsehbar) und wieder mit dem GPU-Coder kompiliert.

4.2.2 Laufzeitdaten

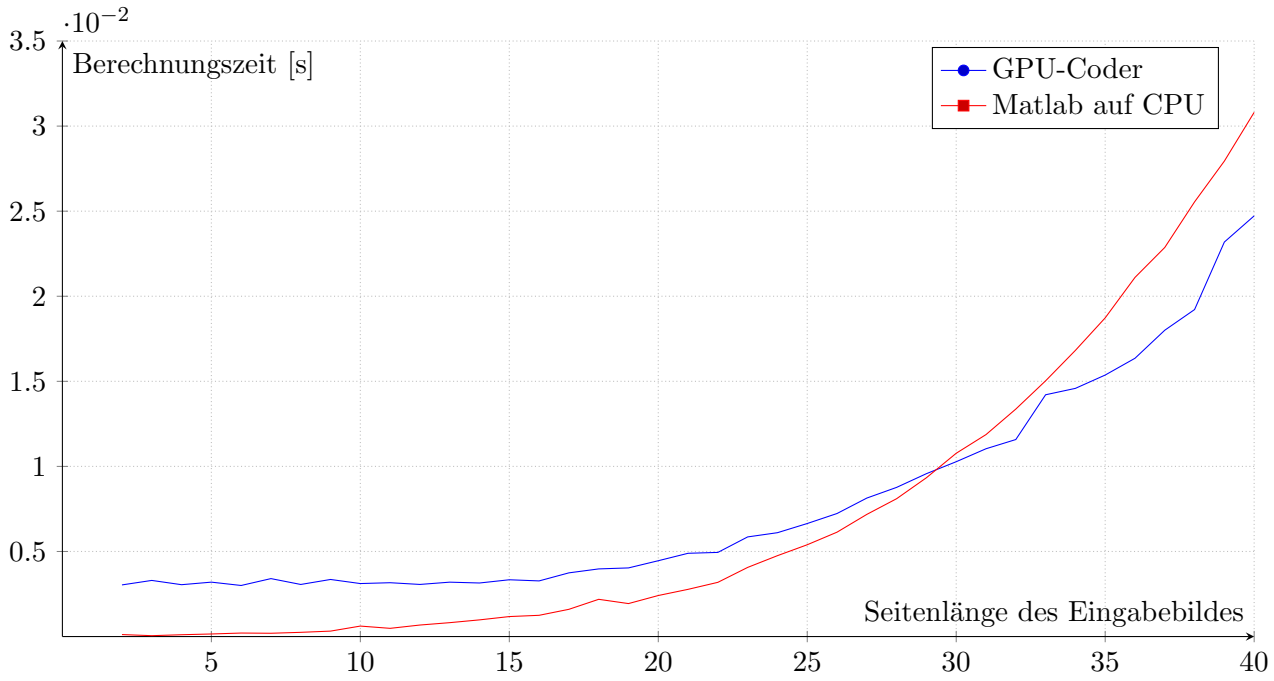


Abbildung 20: Laufzeiten des zweiten GPU-Coder-Programms (Median über 10 Versuche)

Es lässt sich erkennen, dass das Einsetzen von Schleifen die Ausführung des Programms beschleunigt. Einerseits läuft der zweite Ansatz für alle Problemgrößen schneller als der erste (Abb. 21).
Andererseits wird der Overhead im Vergleich zur Matlab-Version kleiner und somit tritt der Punkt, an dem GPU- und CPU-Implementierung gleich schnell sind, früher auf (Abb. 20).

Dadurch bildet dieser Ansatz ab einer Bildseitenlänge von über 30 tatsächlich eine signifikante Beschleunigung.
Trotzdem ist der Speedup nicht besonders hoch und das Speicherproblem besteht weiterhin.

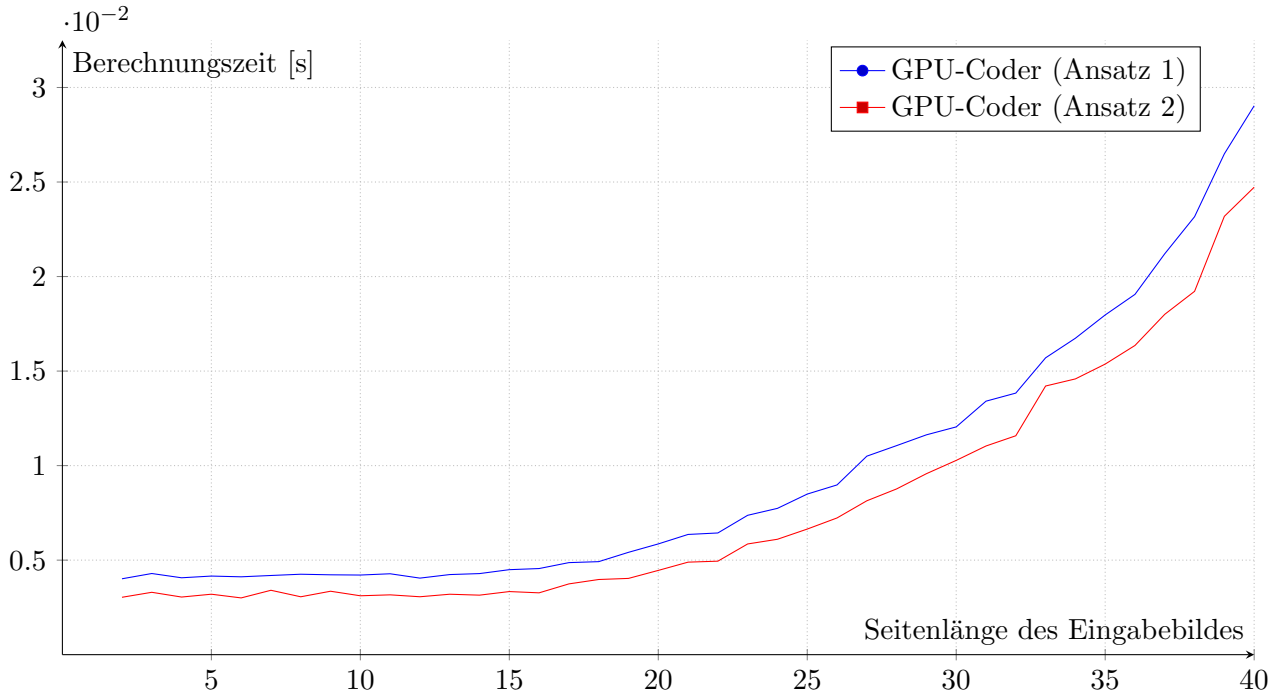


Abbildung 21: Laufzeiten der unterschiedlichen Ansätze (Median über 10 Versuche)

4.2.3 Profiling

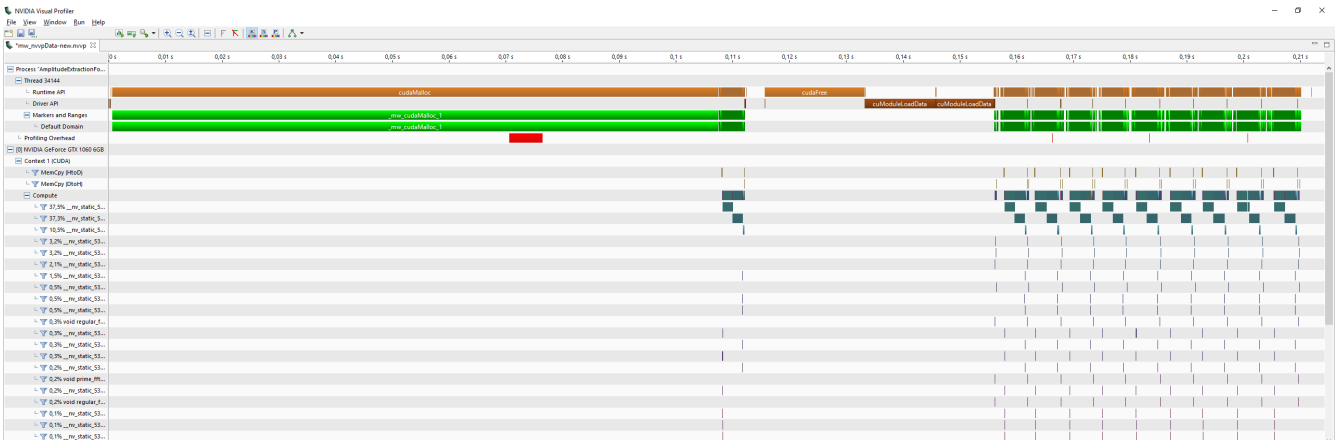


Abbildung 22: Profiling des zweiten Ansatzes mit 10 Durchgängen im NVIDIA-Visual-Profiler

Das Profiling ergibt eine bessere Nutzung der Multiprozessoren. So werden bei diesem Ansatz laut Profiler 12% der Multiprozessor-Kapazitäten genutzt. Allerdings werden trotzdem selten viele parallelisierte Threads ausgeführt.
Zusätzlich lässt sich im Gegensatz zum ersten Ansatz keine eindeutige Quelle für den niedrigen Speedup ausmachen.

5 Implementierung mittels CUDA

Da der GPU-Coder keinen direkten Zugriff auf die einzelnen Komponenten der GPU und die detaillierte Programmierung bietet, lässt sich der GPU-Coder-Code nicht kleinteilig genug gestalten, um die gewünschte Effizienzsteigerung zu erzielen.
Mittels CUDA können die einzelnen Komponenten der GPU direkt angesprochen werden. Dies birgt mehr Optimierungspotential als der GPU-Coder.

5.1 Einbindung in Matlab

Das MEX-Interface transportiert die Ein- und Ausgabedaten zwischen Matlab und CUDA. Für die Einbindung des CUDA-Kernels ist zusätzlich eine Header-Datei (Kap. 8.6) notwendig, die die Hauptfunktion importierbar macht.
Der Matlab-Code wird dann mit dem Befehl

```
nvcc -o cuda_kernel.dll --shared cuda_kernel.cu
```

zu einer DLL kompiliert werden und die MEX-C-Datei wird mit dem Befehl

```
mex mex_interface.c -I"NVIDIA Toolkit\\" -L./ -l./cuda_kernel
```

in Matlab zu einer MEX kompiliert.

5.2 Der CUDA-Kernel

Der CUDA-Code ist im Anhang einsehbar.
Der Code implementiert die nicht-verlustbehaftete Bildrotation nach dem Konzept der Parallelisierung der Pixelberechnungen. Dadurch ist das Speicherproblem bei diesem Lösungsansatz nicht mehr vorhanden. Zusätzlich kann genau bestimmt werden, wie das Verfahren abläuft.
Trotz der unabhängigen Berechnung der Pixel enthält die ursprüngliche Implementierung eine Vorberechnung, die außerhalb der Berechnung der Pixel durchgeführt wird (Kap. 5.4).

5.3 Portierung

Bei der Portierung von CPU auf GPU müssen einige Konzepte, die auf der CPU verwendet wurden, anders umgesetzt werden. Das liegt unter anderem daran, dass einige Funktionen auf der GPU nicht unterstützt sind oder, dass durch die Parallelisierung manche Konzepte nicht mehr funktionieren.

5.3.1 atomicAdd

Am Ende des Rotations-Algorithmus werden alle Berechnungen für jeden Pixel addiert. Das lässt sich auf der CPU durch die serielle Durchführung mit einer einfachen Summe lösen. Auf der GPU muss allerdings *atomicAdd*[13] verwendet werden. Damit kann eine Summe über parallel laufende Threads gebildet werden.

5.3.2 IFFT

Bei der Rotation wird im ersten Schritt die inverse, zweidimensionale Fourier-Transformation auf das Eingangsbild ausgeführt. Auf der GPU ließe sich das mit der cuFFT-library [16] durchführen. Beim Testen mit der verwendeten GPU war es nicht möglich diese Library einzubinden.
Die erstellte MEX wird noch in eine Matlab-Funktion eingebunden, die die Fourier-Transformation einmal auf das Bild anwendet und das Ergebnis an die MEX übergibt. Dadurch kann die Fourier-Transformation trotzdem ausgeführt werden.
Obwohl die Fourier-Transformation auf der CPU ausgeführt werden muss, hat sie nur einen kleinen Anteil an der verbrauchten Laufzeit. Eine Implementierung mittels cuFFT wäre idealer, weil sich die Laufzeit dadurch verbessern würde. Die Einbindung in eine zusätzliche Matlab-Funktion kann so auch vermieden werden.

5.4 Ergebnisse

5.4.1 Laufzeitmessungen

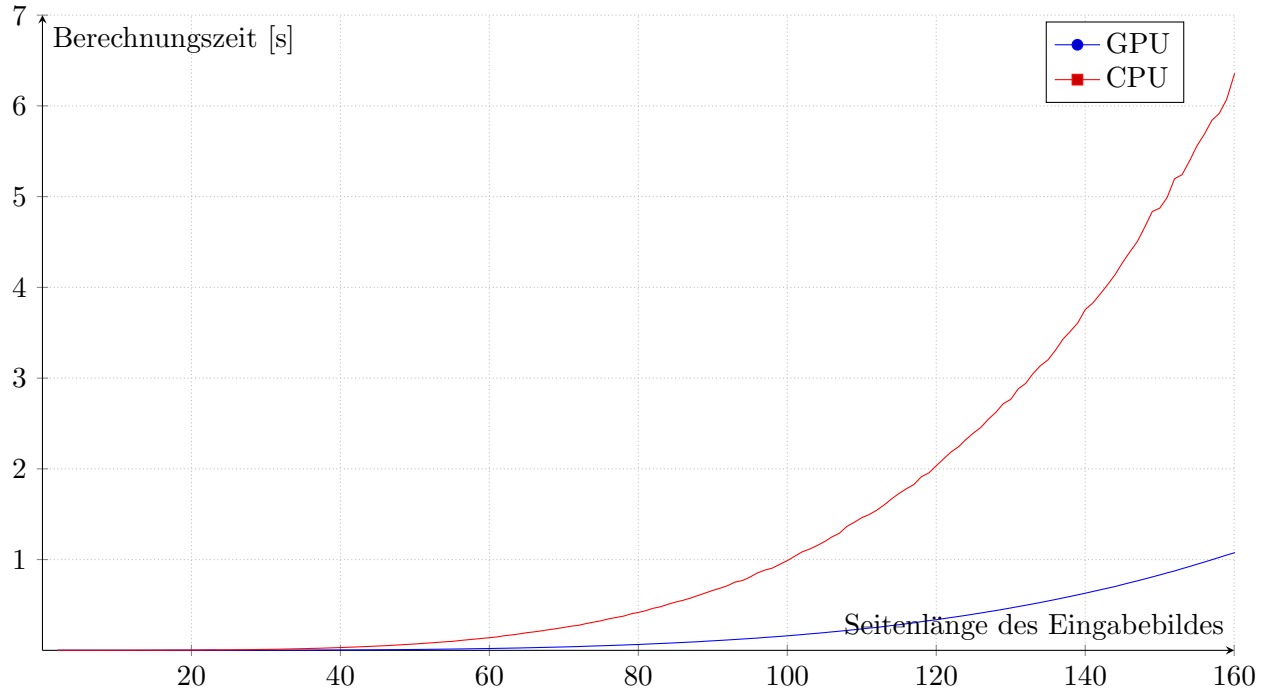


Abbildung 23: Vergleich Laufzeit Matlab und CUDA (Median über 10 Versuche)

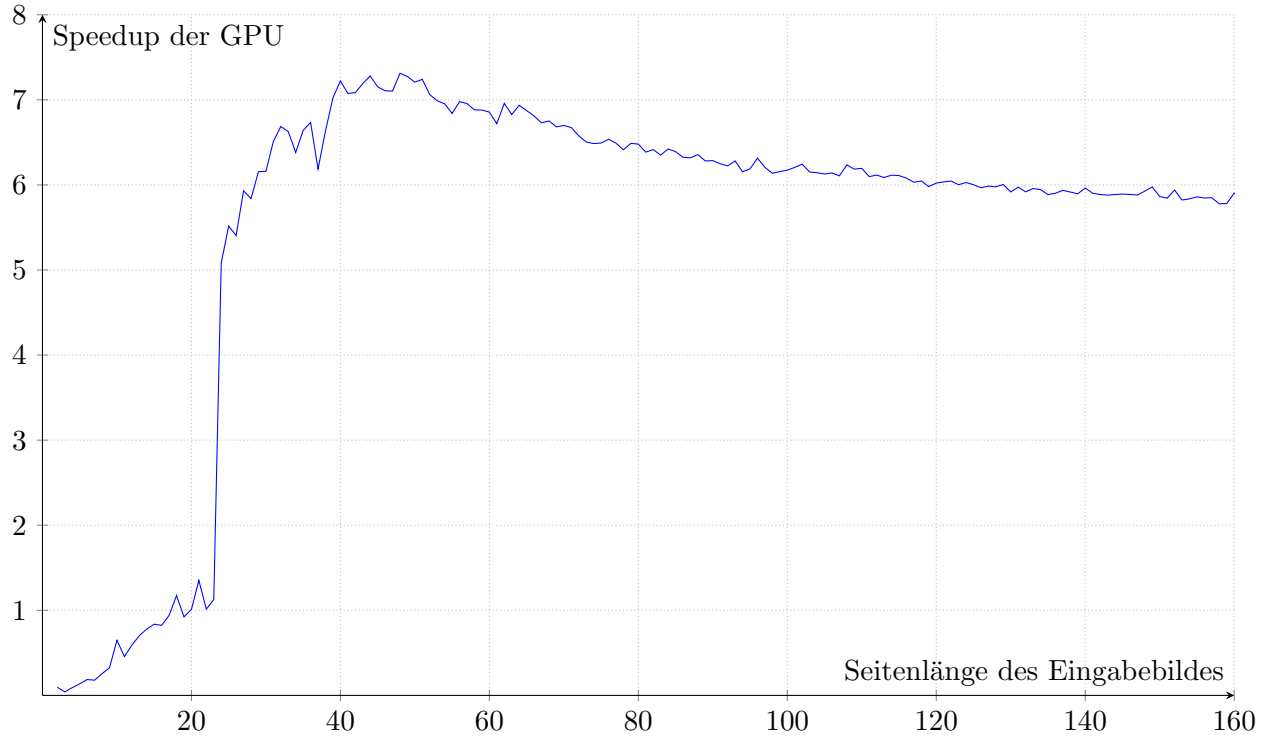


Abbildung 24: Speedup der CUDA-Version

Die Implementierung mittels CUDA erzielt eindeutig eine signifikante Beschleunigung gegenüber der Matlab-Variante (Abb. 23). Selbst für kleinere Bildgrößen ergibt sich keine deutlich verschlechterte Laufzeit.
Ab einer Bildseitenlänge von ungefähr 20 ist die GPU-Variante mindestens genauso schnell wie die CPU-Variante und bei einer Bildseitenlänge von über 26 ergibt sich eine eindeutige Beschleunigung, die für alle größeren Bilder bestehen bleibt (Abb. 24). Im Schnitt lässt sich für größere Bilder ungefähr eine Performance-Zunahme um den Faktor 6,5 bestimmen.

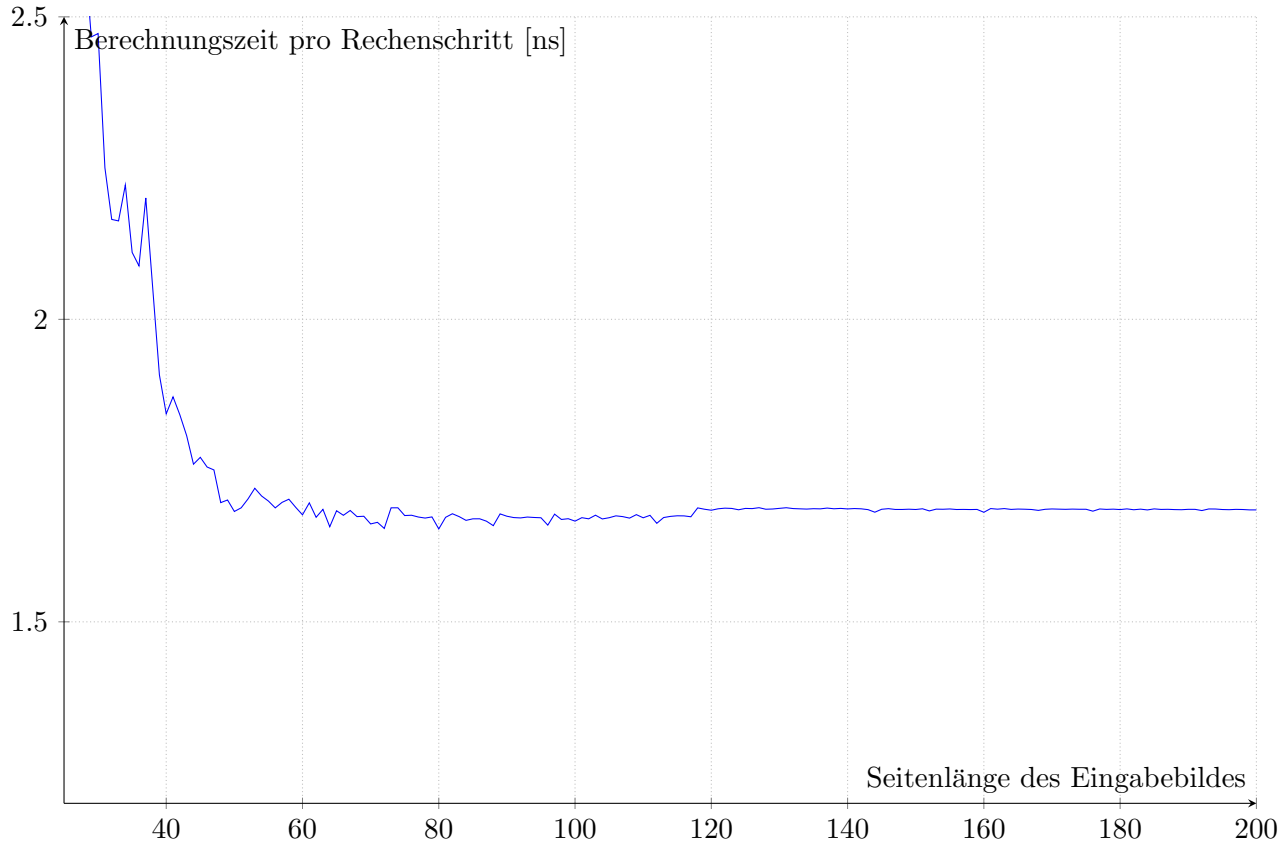


Abbildung 25: Zeit pro parallelisiertem Rechenschritt zur Bestimmung des Overheads

Zur Bestimmung des Overhead-Einfluss, wird außerdem ein Diagramm erstellt, in dem die Laufzeiten im Verhältnis zur Anzahl der Pixel quadratisch normalisiert dargestellt wird. Der Graph zeigt also die Zeit pro notwendigem Rechenschritt an. Dadurch lässt sich der Overhead ablesen, da Code mit niedrigem Overhead einen konstanten Verlauf produzieren sollte.
In diesem Fall gibt es am Anfang einen Hochpunkt, der sich dann bereits bei einer Bildseitenlänge von über 50 in einen konstanten Verlauf verändert (Abb. 25). Es kann davon ausgegangen werden, dass der Overhead bereits bei Bildern mit einer Seitenlänge von 100 Pixeln nicht mehr signifikant ist.

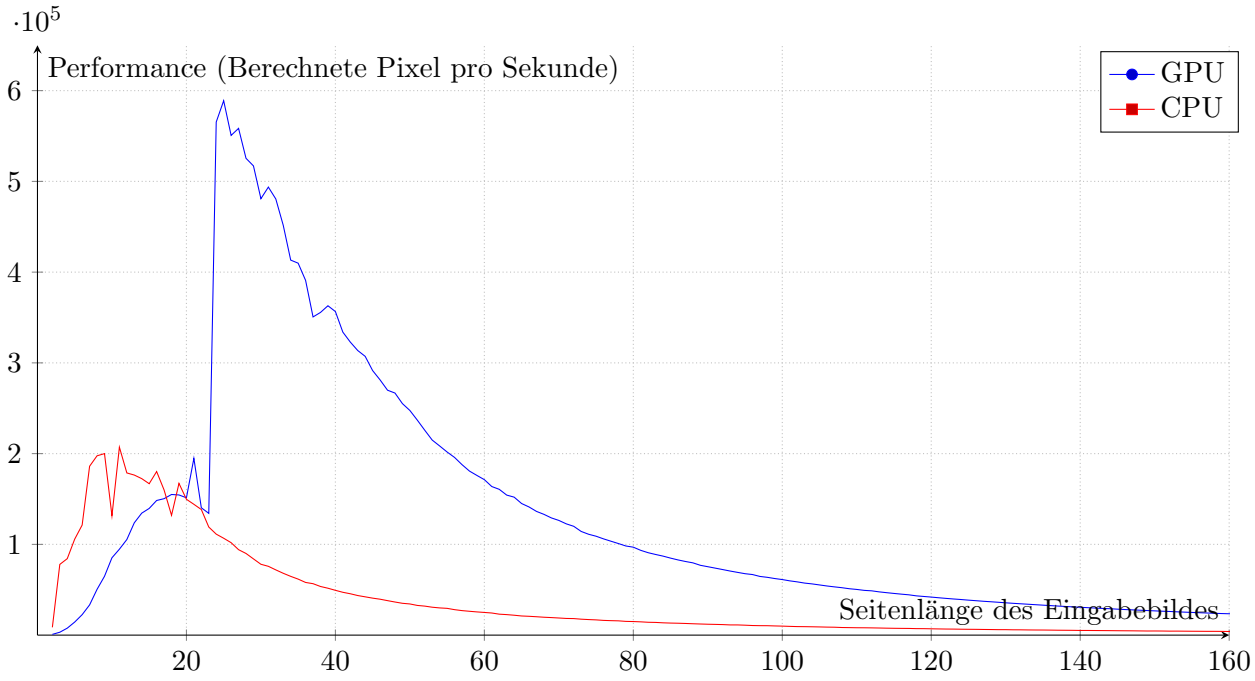


Abbildung 26: Vergleich der Performance von Matlab und CUDA

Der Performance-Graph (Abb. 26) zeigt unterstützend, dass die GPU-Version deutlich effizienter als die Matlab-Version ist. Es lässt sich außerdem erkennen, dass die maximale Performance mit einer Verarbeitung von $6 \cdot 10^5$ Pixeln pro Sekunde bei einem Bild mit der Seitenlänge 25 auftritt.
Bei einem kleineren Bild ist der Overhead noch zu groß und bei größeren Bildern setzt zunehmend der Effekt der quadratisch zunehmenden Laufzeit ein.

5.4.2 Profiling

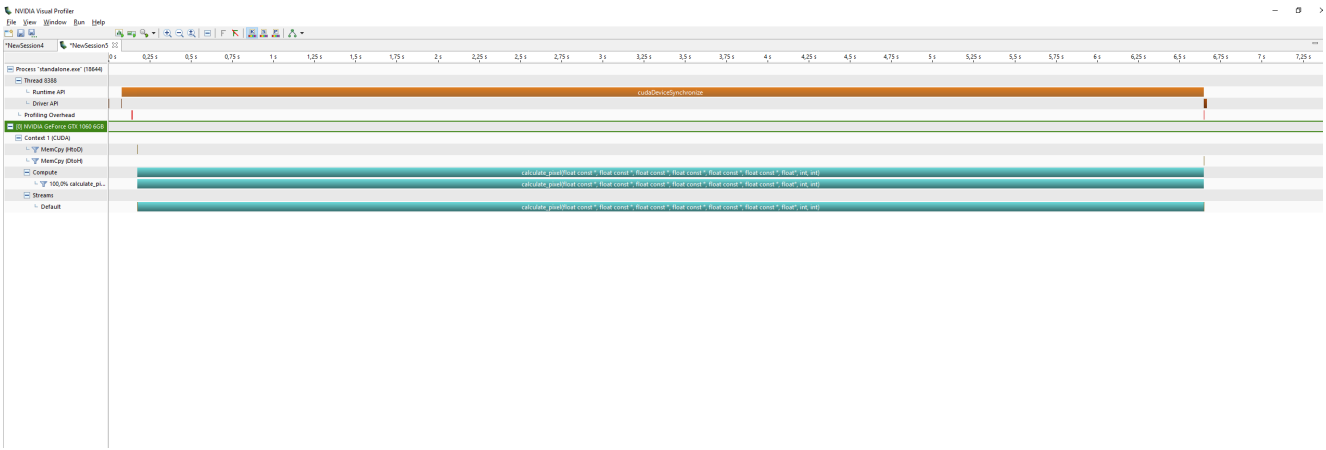


Abbildung 27: Profiling der CUDA-Version

Das Profiling (Abb. 27) zeigt, dass der direkt implementierte CUDA-Code die Laufzeit fast ausschließlich mit der Ausführung der Pixelberechnung verbringt. Das Problem mit zu vielen Speicher-Operationen, das beim GPU-Coder bestand, tritt hier nicht mehr auf.

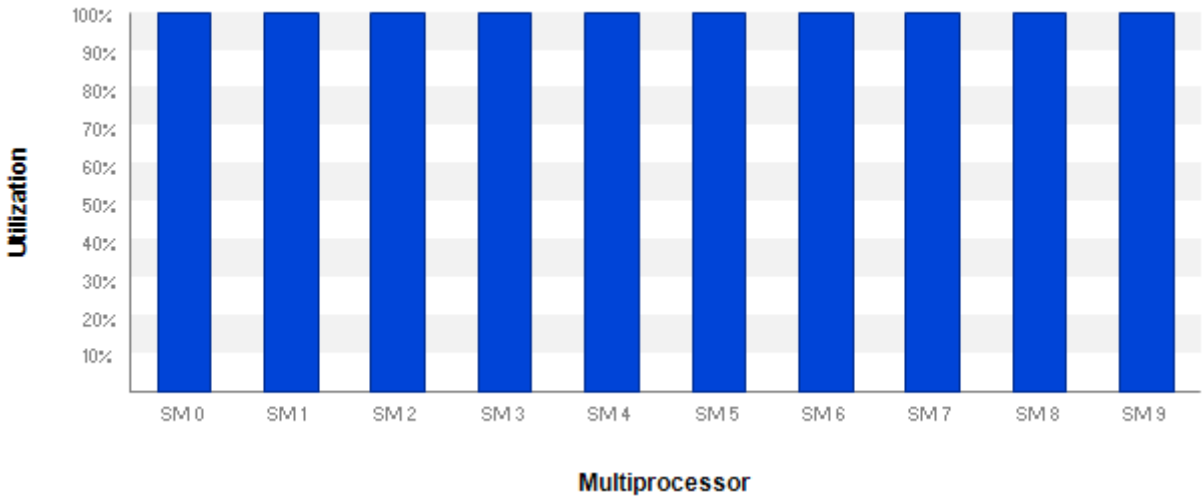


Abbildung 28: Genauere Profiling-Daten bezüglich der Multiprozessor-Nutzung

Das Profiling zeigt die volle Ausnutzung der Multiprozessoren bei diesem Ansatz im Gegensatz zum GPU-Coder-Ergebnis (Abb. 28). Das bedeutet, dass die einzelnen Prozessoren der GPU durchgängig verwendet werden. Daraus kann nicht direkt gefolgert werden, dass die Implementierung effizient ist. Es ist auch möglich, dass die Implementierung ineffizient ist und die Multiprozessoren viel Zeit mit unnötigen Rechenoperationen oder Warten verbringen.
Allerdings ist so sichergestellt, dass die GPU ihr volles Potenzial an Rechenleistung entfaltet, was eine notwendige Bedingung für eine effiziente Implementierung ist.

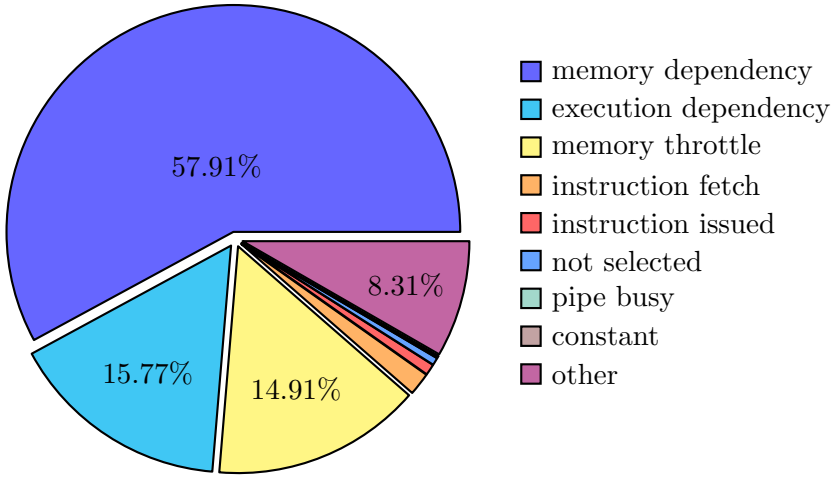


Abbildung 29: Profiling der Gründe für Latenz innerhalb des Kernels

Zur Untersuchung der Verbesserungsmöglichkeiten der Performance, wird ein Latenz-Profiling erstellt (Abb. 29).
Der größte Teil der Latenz tritt dadurch auf, dass ein Wert nicht schnell genug aus dem Speicher ausgelesen werden kann.
Auch das code-spezifische Profiling bezüglich der Latenz (Abb. 30) zeigt, dass vor allem in Code-Zeilen, in denen ein Wert aus dem globalen Speicher ausgelesen wird, Latenz auftritt.
Jeder Eingabearray wird in jedem Durchlauf nur einmal ausgelesen, was notwendig ist, um die Eingeabeinformationen zu erhalten. Es ist in diesem konkreten Fall also schwer die Speicher-Latenz zu optimieren.

```
49
50      FLOAT_DT value = two_pi * delay1[z] / wavel1_value;
51      if (max_y % 2 != 1 && max_y / 2 == z && x==0) {
52          value = 0;
53      }
54      FLOAT_DT ps1_value_real, ps1_value_imaginary;
55      sincosf(value, &ps1_value_imaginary, &ps1_value_real);
56
57      value = two_pi * delay2[z] / wavel2_value;
58      if (max_x % 2 != 1 && max_x / 2 == z && y == 0) {
59          value = 0;
60      }
61      FLOAT_DT ps2_value_real, ps2_value_imaginary;
62      sincosf(value, &ps2_value_imaginary, &ps2_value_real);
63
64      FLOAT_DT total_real = ps1_value_real * ps2_value_real + ps1_value_imaginary * ps2_value_imaginary;
65      FLOAT_DT total_imaginary = ps1_value_real * ps2_value_imaginary - ps1_value_imaginary * ps2_value_real;
66
67      atomicAdd(&output[z], (image_real[x + y * max_y] * total_real - image_imag[x+y*max_y]*total_imaginary) *
68  }
69
```

Abbildung 30: Auszug aus Profiling mit Zeilen mit hoher Latenz

5.4.3 Vergleich mit vollständiger Parallelisierung

Neben der ursprünglichen Implementierung wird auch noch eine zweite Implementierung erstellt (Kap. 8.5), die überhaupt keine Vorberechnungen enthält, sondern alle Rechnungen im Pixel-Berechnungs-Kernel durchführt.

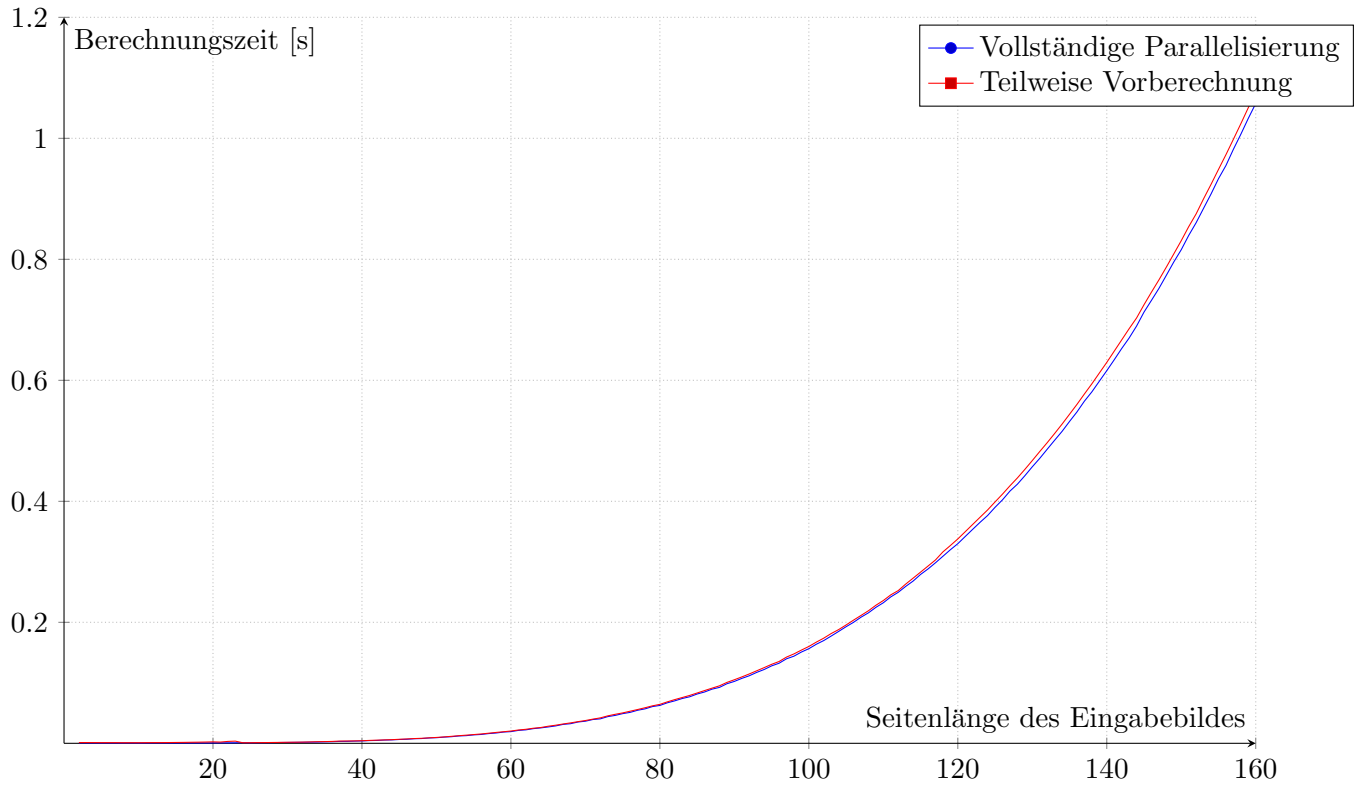


Abbildung 31: Vergleich der beiden Versionen

Beim Vergleich der Laufzeiten der beiden Implementierungen ergibt sich keine signifikante Beschleunigung durch einen der beiden Ansätze.

5.4.4 Genauigkeit

Zur Überprüfung der Genauigkeit der Berechnungen wird die absolute Differenz zwischen der Ausgabe der Matlab und der CUDA-Variante für unterschiedlich große Bilder bestimmt.
Dabei wird mit realistischen Daten (Abb. 32) und mit zufällig generierten Bildern (Abb. 33) experimentiert. In beiden Fällen liegt die maximale Abweichung unter 10^{-5}. Es lässt sich erkennen, dass die Stärke der Abweichung steigt, je größer das Bild ist. Das ist dadurch erklärbar, dass bei mehr Pixeln auch die Wahrscheinlichkeit höher wird, dass einer davon ungenau berechnet wird.

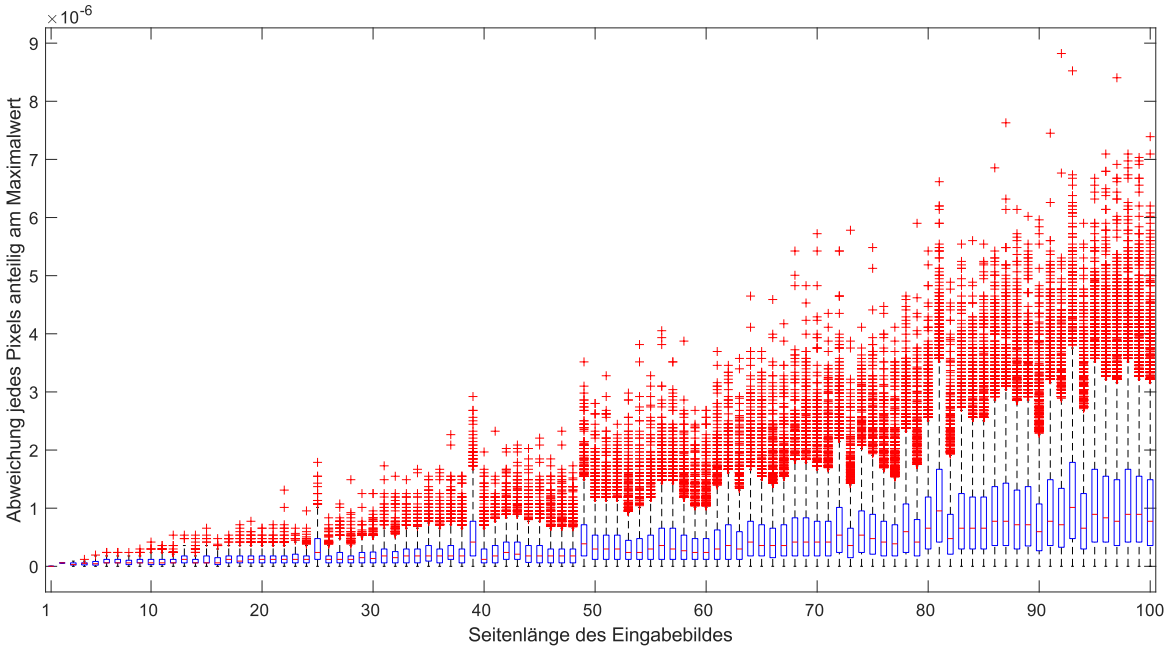


Abbildung 32: Genauigkeitsmessung mit realistischen Daten

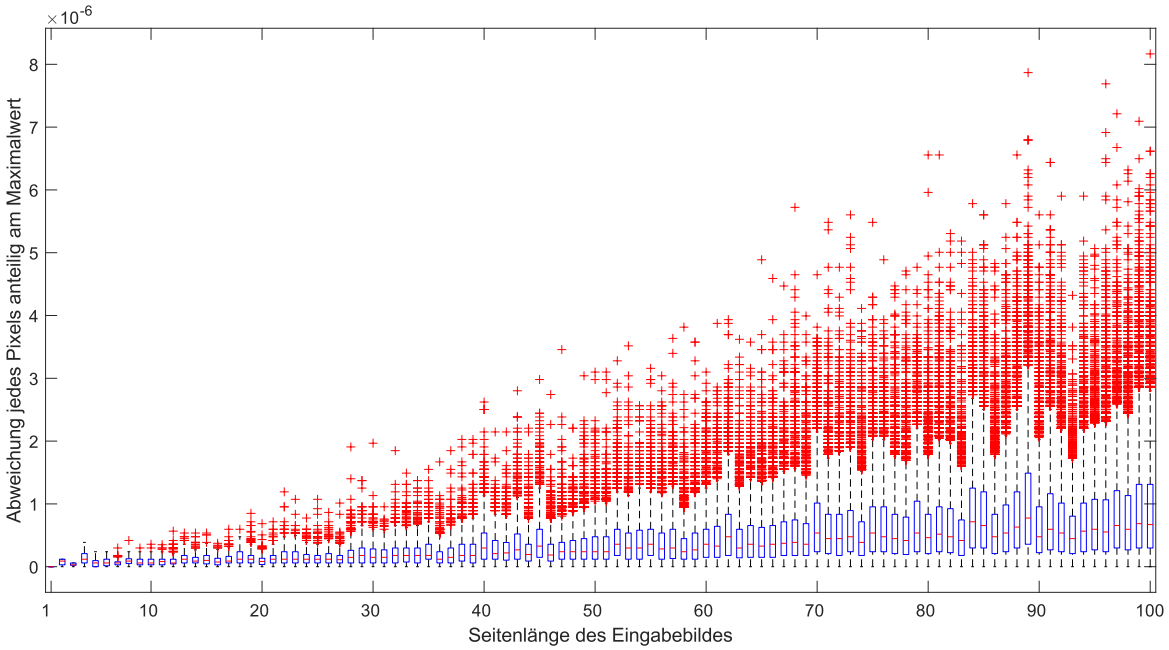


Abbildung 33: Genauigkeitsmessung mit zufällig generiertem Rauschen

5.5 Branch-Vermeidung

Für GPUs können if-Abfragen je nach Implementierungsart einen großen Laufzeit-Faktor ausmachen. Im CUDA-Kernel-Code befinden sich zwei if-Abfragen, die zwar nur für einen speziellen Sonderfall wichtig sind, aber trotzdem bei jeder Berechnung durchlaufen werden müssen.
Aus diesem Grund wurde ein Konzept erarbeitet, wie diese if-Abfragen vermieden werden können. Diese setzen unter einer bestimmten Bedingung einen Wert auf null. Es ist allerdings auch möglich, diesen Filter nicht mit einer if-Abfrage, sondern mit einer Multiplikation umzusetzen. So kann die Variable, die auf null gesetzt werden soll, mit der Bedingung der if-Abfrage multipliziert werden. Die Bedingung wird dabei von einem Boolean zu einem Integer konvertiert und ist somit entweder null oder eins. Dabei muss die Bedingung noch logisch invertiert werden, damit die Variable auf null gesetzt wird, wenn die Bedingung erfüllt ist. Wenn die Bedingung nicht erfüllt ist, wird die Variable mit eins multipliziert.
Im Code werden dann die if-Abfragen (Zeile 21-23, 28-30, Kap. 8.4) durch folgende Multiplikationen ersetzt:

```
1 ...
2 value *= !(max_y % 2 != 1 && x == max_y / 2);
3 value2 *= !(max_x % 2 != 1 && y == max_x / 2);
4 ...
```

Bei Laufzeitmessungen mit dieser Verbesserungen ergab sich eine Beschleunigung um ungefähr 0.5%.

5.6 Brute-Force-Testung unterschiedlicher Compilerflags

Neben der Optimierung von Hand gibt es auch die Möglichkeit, dem CUDA-Compiler durch sogenannte Compilerflags anzuzeigen, an welcher Stelle möglichst viel automatisch optimiert werden soll.
Der Compiler optimiert teilweise schon selbstständig, aber dabei können noch Bereiche angegeben werden, die möglichst stark optimiert werden sollen. Solche Optimierungen haben auch fast immer einen Nachteil. Das können zum Beispiel eine längere Compile-Zeit, Genauigkeitsverlust oder mehr Bedarf an nicht-optimierten Ressourcen sein.
Zur Testung der unterschiedlichen Compilerflag-Kombinationen wurde eine Liste an Compilerflags erstellt, die möglicherweise eine Optimierung hervorrufen könnten [17]:

- *-O0*, *-O1*, ..., *-O5*: Generelle Optimierungsflags
- *-maxrregcount=16*: Begrenzt die Register auf 16 pro Thread. Kann dazu führen, dass mehr Threads parallel laufen können, weil weniger Register verbraucht werden.
- *-use_fast_math*: Aktiviert die Verwendung von Approximation bei mathematischen Funktionen. Kann dazu führen, dass die Ergebnisse ungenauer werden.
- *–prec-div=true*: Aktiviert das Runden von Gleitkommazahlen.
- *–prec-sqrt=true*: Aktiviert die Approximation bei Wurzeloperationen.
- *-extra-device-vectorization*: Aktiviert aggressivere Code-Vektorisierung.

Anschließend wurde ein Matlab-Skript geschrieben, dass alle Kombinationen dieser Compilerflags ausprobiert und einen Performance-Test erstellt.
Dabei kam heraus, dass die Verwendung von
-O0 -maxrregcount=16 -use_fast_math –prec-sqrt=true -extra-device-vectorization
die Performance im Schnitt am meisten erhöht. Die Optimierung dieser Kombination kommt allerdings fast ausschließlich von *-use_fast_math*, was alleine eine ähnliche Performance erzielt.

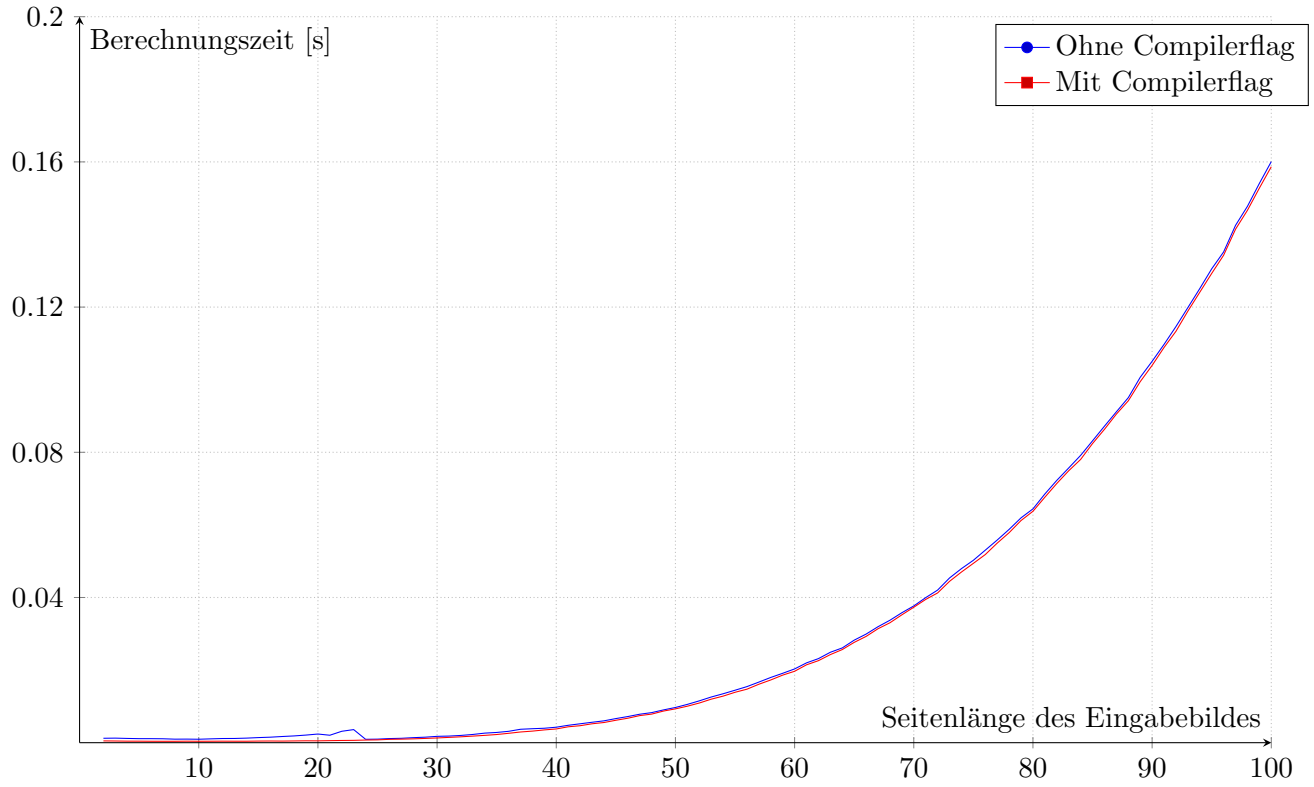


Abbildung 34: Zeitmessung bei Verwendung von approximativer Funktionen

Die Verwendung von approximativen mathematischen Funktionen beschleunigt den Code ein wenig (Abb. 34). Außerdem wurde bei einem Profiling der Genauigkeit festgestellt, dass die Verwendung dieser Compilerflag nicht zu einer signifikanten Verschlechterung der Genauigkeit führt.

5.7 Umstrukturierung des Codes

Die große Latenz, kann unter anderem dadurch behoben werden, dass die Code-Reihenfolge geändert wird. So können bestimmte Rechenoperationen nach hinten verlagert werden. Dadurch werden die benötigten Memory-Zugriffe schon getätigt, bevor die Rechenoperation ausgeführt werden soll.
Aus diesem Grund wurden unterschiedliche Versionen des Codes erstellt und getestet. Dabei wurde vor allem versucht, latenzerzeugende Operationen weiter nach hinten zu verschieben. Beim Profiling der unterschiedlichen Varianten war zwar zu erkennen, dass sich die Latenz-Gründe verändert (Abb. 35), aber es konnte keine Verbesserung der Laufzeit festgestellt werden.

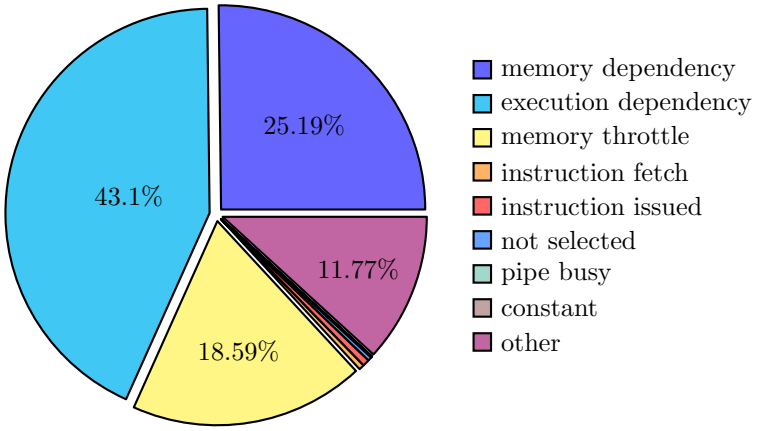


Abbildung 35: Profiling der Gründe für Latenz bei Änderung der Code-Reihenfolge

5.8 Doubles

Zusätzlich zur float-Variante wurde noch eine alternative Implementierung mit Doubles erstellt.

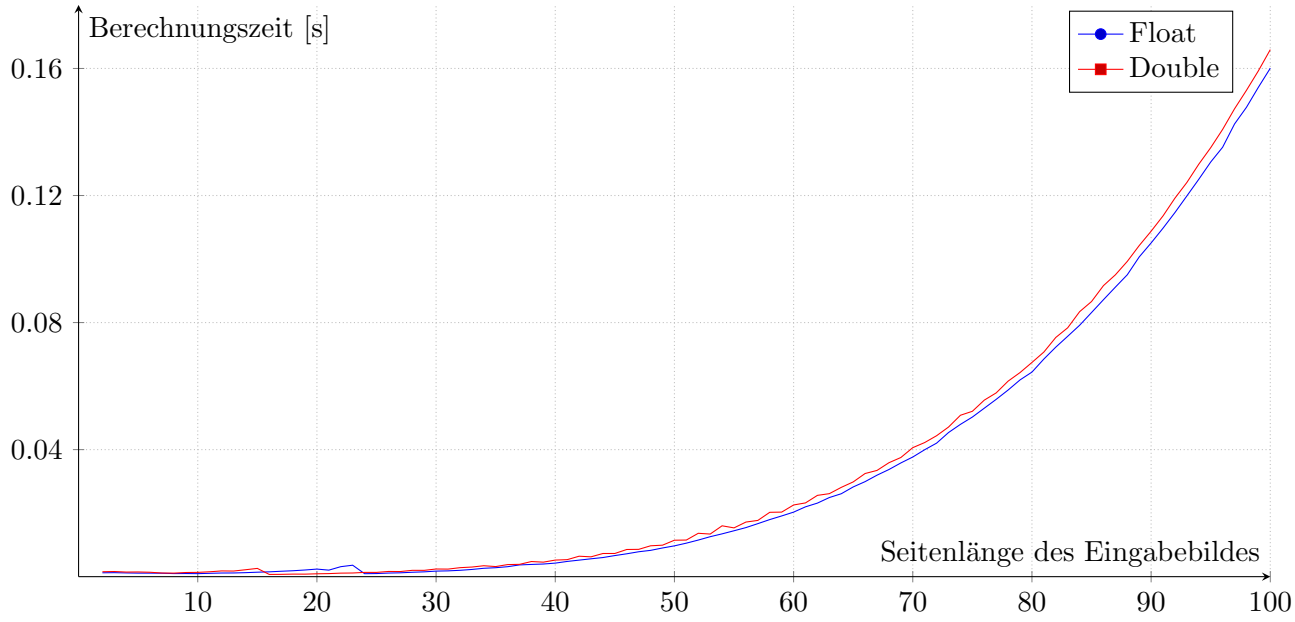


Abbildung 36: Laufzeitvergleich von Float und Double

Die Laufzeit mit Doubles ist trotz der erwarteten schlechteren Performance (Kap. 3.3) nicht deutlich geringer. Das verstärkt die Vermutung, dass die Laufzeit vor allem durch Memory-Operationen und Latenz begrenzt ist.
Außerdem erzielen Doubles wie erwartet eine deutlich bessere Genauigkeit als Floats (Abb. 37).

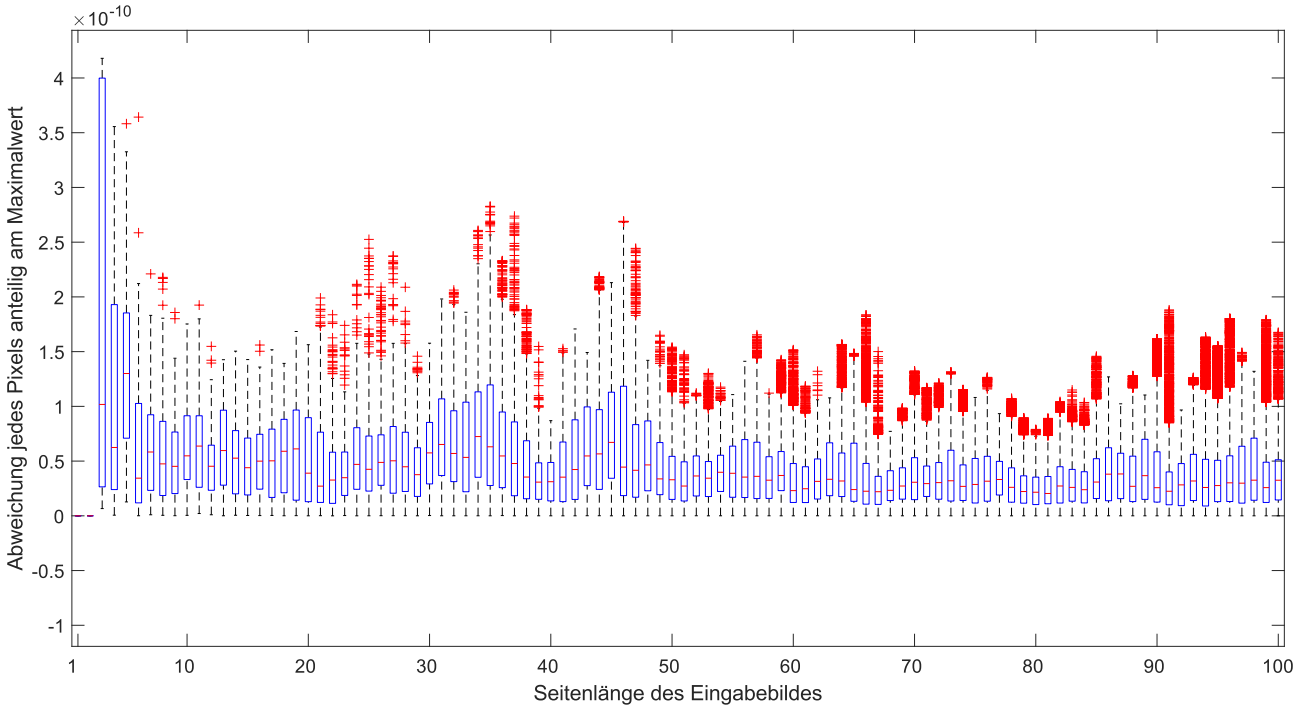


Abbildung 37: Genauigkeitsmessung bei Doubles

6 Fazit und Ausblick

Das in CUDA implementierte Konzept der Parallelisierung über die Pixel funktioniert und erzielt eine Beschleunigung gegenüber der CPU-Variante. Dadurch wurde das grundsätzliche Ziel des Projekts erfüllt. Der Laufzeitaufwand für den Anwendungsfall von einem dreidimensionalen Bild mit einer Seitenlänge von 250 Pixeln konnte auf ungefähr 28 Minuten reduziert werden. Für große Bilder ist die Laufzeit allerdings trotzdem noch sehr hoch, da der Laufzeitverbrauch quadratisch ansteigt.
Für die effizientere Gestaltung der 3D-Bildrotation, wäre die Umsetzung eines Multi-GPU-Ansatzes, der nicht nur über Pixelberechnungen, sondern auch über Ebenenberechnungen parallelisiert, möglich. Neben einem anderen Parallelisierungsansatz wäre auch ein anderer algorithmischer Ansatz eine Idee. Falls es möglich wäre, Die CUDA-Implementierung kann durch noch genaueres Profiling wahrscheinlich auch noch weiter verbessert werden. So wäre es zum Beispiel vielversprechend, stärker auf die Speicher-Latenz-Probleme einzugehen. Diese könnten möglicherweise mit einer anderen Architektur reduziert werden, indem zum Beispiel *shared Memory* oder eine andere Registerstruktur verwendet werden.
Für eine realistische Aussage über die Nutzbarkeit des Verfahrens, wäre es notwendig, dieses für unterschiedliche GPU-Modelle zu testen. Dabei soll verglichen werden, welche Modellreihe sich am besten für diese Art von Berechnungen eignet.

7 Danksagung

Mein großer Dank geht an Michael Zapf vom IPE (Institut für Prozessdatenverarbeitung und Elektrotechnik) am Karlsruher Institut für Technologie, der diese Arbeit betreut und ermöglicht hat.
Außerdem möchte ich mich bei den Kursleiterinnen und Kursleitern des Hector-Seminars Paul Bischof, Dietmar Gruber, Norbert Krieg, Anke Richert, Thomas Hermann sowie Thomas Knecht für die große Unterstützung während meiner Jahre beim Hector-Seminar bedanken.
Überdies danke ich Josephine und Dr. Hans-Werner Hector und ihrer Stiftung für die großzügige Unterstützung des Hector-Seminars.

8 Anhang

8.1 Matlab-Prototyp

```
function [out]=AmplitudeExtractionFourier2D(CE_in,delay1_in,delay2_in,fs_in)
persistent recursivecount;
if isempty(recursivecount)
    recursivecount=0;
end

persistent solvedsize;
if isempty(solvedsize)
    solvedsize=0;
end

persistent problemsize;
if isempty(problemsize) || recursivecount == 0
    problemsize=numel(CE_in);
    solvedsize=0;
end

try
    % Immediatly produce out of memory error if problem is too big
    ceft=zeros([size(CE_in,1),size(CE_in,2),numel(delay1_in),2],'double');
    ps=zeros([size(CE_in,1),size(CE_in,2),numel(delay1_in),2],'double');
    val2=zeros([size(CE_in,1),size(CE_in,2),numel(delay1_in),2],'double');

    fs=fs_in-fs_in/length(CE_in);

    delay1=-delay1_in;
    delay2=-delay2_in;
    delay1=delay1(:)';
    delay2=delay2(:)';
    CE=CE_in;

    % Transpose matrix to avoid problems with squeeze at the end
    if size(CE,1)==1
        CE=CE';
        trans=1;
    else
        trans=0;
    end

    if mod(size(CE,2),2)==1 %odd
        f2=0:fs./(size(CE,2)-1):fs;
        f2=circshift(fftshift(f2-fs/2),[0 1]);
    else %even
       f2=0:fs./(size(CE,2)-1):fs;
       f2(end:-1:size(CE,2)/2+2)=-f2(2:size(CE,2)/2);
    end

    if mod(size(CE,1),2)==1 %odd
        f1=0:fs./(size(CE,1)-1):fs;
        f1=circshift(fftshift(f1-fs/2),[0 1]);
    else %even
       f1=0:fs./(size(CE,1)-1):fs;
       f1(end:-1:size(CE,1)/2+2)=-f1(2:size(CE,1)/2);
    end

    wavel1=1./f1;
    wavel2=1./f2;

    if length(delay1)>1
        d1=2*pi.*delay1'./repmat(wavel1,[size(delay1,2) 1]);
    else
```

```
        d1=2*pi.*delay1./wavel1;
    end
    if length(delay2)>1
        d2=2*pi.*delay2'./repmat(wavel2,[size(delay2,2) 1]);
    else
        d2=2*pi.*delay2./wavel2;
    end

    ps1=complex(cos(d1),sin(d1));
    ps2=complex(cos(d2),sin(d2));

    if mod(size(CE,1),2)~=1 %fs/2 fix
       ps1(:, size(CE,1)/2+1)=1;
    end
    if mod(size(CE,2),2)~=1 %fs/2 fix
       ps2(:, size(CE,2)/2+1)=1;
    end

    ps1_2d=(shiftdim(shiftdim(ps1,-1),2));
    ps2_2d=(shiftdim(ps2',-1));
    ps=ps2_2d.*ps1_2d;

    ceft=ifft2(CE);

    if length(delay1)>1
        ceft=repmat(ceft,[1 1 length(delay1)]);
    end

    val2=real(ceft.*ps);
    out=squeeze(sum(sum(val2.*abs(ps),1),2));

    if trans==1
        out=out';
    end

    solvedsize=solvedsize+numel(CE_in);

catch % out of memory -> split problem
    recursivecount=recursivecount+1;
    disp(['Recursion count+1 ' num2str(recursivecount) ' with problemsize ' ...
        num2str(numel(delay2_in)) ' (' num2str(numel(CE_in)*(numel(delay2_in))*8/1024^3) 'GB)']);

    clear ceft ps val2;
    delay1_in=delay1_in(:)';
    delay2_in=delay2_in(:)';
    out1 = AmplitudeExtractionFourier2D(CE_in,delay1_in(1:floor(length(delay1_in)/2)), ...
        delay2_in(1:floor(length(delay2_in)/2)),fs_in);
    out2 = AmplitudeExtractionFourier2D(CE_in,delay1_in(1+floor(length(delay1_in)/2):end), ...
        delay2_in(1+floor(length(delay2_in)/2):end),fs_in);
    out=cat(1, out1, out2);
    out=cat(1, out1, out2);

    recursivecount=recursivecount-1;
    disp(['Recursion count-1, resolved ' num2str(100*solvedsize/problemsize) '%' ]);

    if (solvedsize>=problemsize) && (recursivecount == 0)
        solvedsize = [];
        problemsize = [];
    end
end
```

8.2 Matlab-Prototyp mit Bearbeitung für GPU-Coder

```
function [out]=AmplitudeExtractionFourier2D(CE,delay1_in,delay2_in,fs_in)
fs=fs_in-fs_in/length(CE);

delay1=-delay1_in(:)';
delay2=-delay2_in(:)';

if size(CE,2)==1
    CE=CE';
    trans=true;
else
    trans=false;
end

f2 = zeros(1,size(CE, 2));
index = 1;

for count_up=0:(fs./(size(CE,2)-1)):fs
    f2(index) = double(count_up);
    index = index + 1;
end

if mod(size(CE,2),2)==1 %odd
    f2=double(circshift(fftshift(f2-fs/2),[0 1]));
else %even
    f2(end:-1:size(CE,2)/2+2)=-f2(2:size(CE,2)/2);
end

f1 = zeros(1,size(CE, 1));
index = 1;

for count_up=0:(fs./(size(CE,1)-1)):fs
    f1(index) = double(count_up);
    index = index + 1;
end

if mod(size(CE,1),2)==1 %odd
    f1=double(circshift(fftshift(f1-fs/2),[0 1]));
else %even
   f1(end:-1:size(CE,1)/2+2)=-f1(2:size(CE,1)/2);
end

wavel1=double(1./f1);
wavel2=double(1./f2);

if length(delay1)>1
    % hat selbe größe wie produkt
    d1 = repmat(wavel1,[size(delay1,2) 1]);

    for i=1:size(d1,2)
        d1(:,i) = 2*pi.*delay1'./d1(:,i);
    end
else
    d1=2*pi.*delay1./wavel1;
end

if length(delay2)>1
    % hat selbe größe wie produkt
    d2=repmat(wavel2,[size(delay2,2) 1]);

    for i=1:size(d2,2)
        d2(:,i) = 2*pi.*delay2'./d2(:,i);
    end
else
```

```
    d2=2*pi.*delay2./wavel2;
end

ps1=complex(cos(d1),sin(d1));
ps2=complex(cos(d2),sin(d2));

if mod(size(CE,1),2)~=1 %fs/2 fix
   ps1(:, size(CE,1)/2+1)=1;
end
if mod(size(CE,2),2)~=1 %fs/2 fix
   ps2(:, size(CE,2)/2+1)=1;
end

% size: 1 X Y
ps2_2d=(shiftdim(ps2',-1));
% size: X 1 Y
ps1_2d=(shiftdim(shiftdim(ps1,-1),2));

% Making compatible arrays same dimensions
% repmat für Speicherallokation unerheblich, weil später diese Variable
% auch verwendet wird, um die Ergebnisse zu speichern
ps1_2d = repmat(ps1_2d, 1, size(ps2_2d,2), 1);

% Durchloopen statt Matrizenmultiplikation
for i=1:size(ps1_2d,1)
    ps1_2d(i,:,:) = ps1_2d(i,:,:).*ps2_2d;
end

ceft=ifft2(CE);

if length(delay1)>1
    for i=1:length(delay1)
        ps1_2d(:,:,i) = real(ps1_2d(:,:,i).*ceft).*abs(ps1_2d(:,:,i));
    end

    out=squeeze(sum(sum(ps1_2d,1),2));
else
    val2=real(ceft.*ps1_2d);
    out=squeeze(sum(sum(val2.*abs(ps1_2d),1),2));
end
```

8.3 Interface-Code

```
#include "mex.h"
#include "matrix.h"
#include "cuda_kernel.h"
#include <string.h>
#define FLOAT_DT float

void mexFunction(int output_n, mxArray *output[], int input_n, const mxArray *input[]) {
    // Input validation
    if (input_n != 5){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","Four inputs required.");
    }
    if (mxGetNumberOfDimensions(input[0]) > 2){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","First input is not a 2-dimensional-array");
    }
    if (!mxIsSingle(input[0])){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","First input should be singles");
    }
    if (mxGetNumberOfDimensions(input[1]) > 2){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","Second input is not a 2-dimensional-array");
    }
    if (!mxIsSingle(input[1])){
```

```
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","Second input should be singles");
    }
    if (mxGetNumberOfDimensions(input[2]) > 2){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","Third input is not 2-dimensional-array");
    }
    if (!mxIsSingle(input[2])){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","Third input should be singles");
    }
    // Read dimension
    int* dimensions1 = (int*) mxGetDimensions(input[0]);
    int* dimensions2 = (int*) mxGetDimensions(input[1]);
    int* dimensions3 = (int*) mxGetDimensions(input[2]);
    int* dimensions4 = (int*) mxGetDimensions(input[3]);
    bool different_dimensions = false;

    if (dimensions1[0] != dimensions2[0] || dimensions1[2] != dimensions2[2]){
        different_dimensions = true;
    }
    if (dimensions2[0] != dimensions3[0] || dimensions2[2] != dimensions3[2]){
        different_dimensions = true;
    }
    if (different_dimensions){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","The dimensions of the input arrays are different");
    }
    if (mxGetNumberOfDimensions(input[3]) > 2 || dimensions4[0] != 1 || dimensions4[2] != 1){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","The fourth input should be a scalar");
    }
    if (!mxIsSingle(input[3])){
        mexErrMsgIdAndTxt("AmplitudeExtractionFourier2D:nrhs","The fourth input should be a single");
    }
    int max_y = dimensions1[0];
    int max_x = dimensions1[2];
    int total_length = max_x * max_y;

    // Read inputs
    FLOAT_DT *input1, *input2, *input3, *input5;
    FLOAT_DT input4;

    FLOAT_DT *output_local;
    double *output_double;
    output_local = (FLOAT_DT*)malloc(total_length*sizeof(FLOAT_DT));
    output_double = (double*)malloc(total_length*sizeof(double));

    input1 = (FLOAT_DT *)mxGetData(input[0]);
    input2 = (FLOAT_DT *)mxGetData(input[1]);
    input3 = (FLOAT_DT *)mxGetData(input[2]);
    input4 = (FLOAT_DT) mxGetScalar(input[3]);
    input5 = (FLOAT_DT *)mxGetData(input[4]);

    AmplitudeExtractionFourier2D(input1, input5, input2, input3, input4, output_local, max_x, max_y);

    for (int i = 0; i < total_length; i++) {
        output_double[i] = (double) output_local[i];
    }

    output[0] = mxCreateDoubleMatrix(total_length, 1, mxREAL);
    memcpy(mxGetPr(output[0]), output_double, total_length * sizeof(double));
    free(output_local);
    free(output_double);

    return;
}
```

8.4 CUDA-Code

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <math_constants.h>
#include "cuda_kernel.h"

#define FLOAT_DT float
#define two_pi 6.283185308f
#define MAX(a, b) ((a > b) ? (a) : (b))

__global__ void calculate_pixel(const FLOAT_DT *delay1, const FLOAT_DT *delay2,
                                const FLOAT_DT *image_real, const FLOAT_DT *image_imag,
                                FLOAT_DT *output, FLOAT_DT fs, int max_x, int max_y) {
    // Coordinates
    long long threadID = threadIdx.x+blockIdx.x*blockDim.x;
    int x = threadID % max_y;
    int y = (threadID/max_y)%max_x;
    int z = threadID/(max_x*max_y);

    FLOAT_DT value = two_pi * delay1[z] / wavel1[x];
    if (max_y % 2 != 1 && max_y / 2 == x) {
        value = 0;
    }
    FLOAT_DT ps1_value_real, ps1_value_imaginary;
    sincosf(value, &ps1_value_imaginary, &ps1_value_real);

    value = two_pi * delay2[z] / wavel2[y];
    if (max_x % 2 != 1 && max_x / 2 == y) {
        value = 0;
    }
    FLOAT_DT ps2_value_real, ps2_value_imaginary;
    sincosf(value, &ps2_value_imaginary, &ps2_value_real);

    FLOAT_DT total_real = ps1_value_real * ps2_value_real + ps1_value_imaginary * ps2_value_imaginary;
    FLOAT_DT total_imaginary = ps1_value_real * ps2_value_imaginary - ps1_value_imaginary * ps2_value_real;

    atomicAdd(&output[z], (image_real[x + y * max_y] * total_real - image_imag[x+y*max_y]*total_imaginary)
                  * sqrtf(total_real * total_real + total_imaginary * total_imaginary));
}

// image: 2d, delay1_in: 1d, delay2_in: 1d
// delay1 und delay2 werden implizit transponiert
void AmplitudeExtractionFourier2D(FLOAT_DT *image_real, FLOAT_DT *image_imag, FLOAT_DT *delay1,
                                  FLOAT_DT *delay2, FLOAT_DT fs_in, FLOAT_DT *output,
                                  int max_x, int max_y) {
    int longest_dimension;
    FLOAT_DT fs;

    longest_dimension = MAX(max_x, max_y);
    fs = fs_in - fs_in / (FLOAT_DT) longest_dimension;

    FLOAT_DT *wavel1, *wavel2;
    wavel1 = (FLOAT_DT*)malloc(max_y*sizeof(FLOAT_DT));
    wavel2 = (FLOAT_DT*)malloc(max_x*sizeof(FLOAT_DT));

    // Create wavel1
    if (max_y % 2 == 1) {
        for (int i = 0; i < max_y; i++) {
            // circshift + fftshift
            int shifted_i = (i + ((int) max_y / 2)) % max_y;
            wavel1[i] = 1 / ((FLOAT_DT) shifted_i * (fs / (FLOAT_DT) (max_y - 1)) - fs / 2);
        }
    } else {
```

```
        for (int i = 0; i < max_y; i++) {
            // Shift (f1(end:-1:size(CE,1)/2+2)=-f1(2:size(CE,1)/2);)
            int backwards_i = max_y - i - 1;

            if (backwards_i < max_y / 2 - 1) {
                wavel1[i] = -wavel1[backwards_i + 1];
            } else {
                wavel1[i] = 1 / ((FLOAT_DT) i * (fs / (FLOAT_DT) (max_y - 1)));
            }
        }
    }

    // Create wavel2
    if (max_x % 2 == 1) {
        for (int i = 0; i < max_x; i++) {
            // circshift + fftshift
            int shifted_i = (i + ((int) max_x / 2)) % max_x;
            wavel2[i] = 1 / ((FLOAT_DT) shifted_i * (fs / (FLOAT_DT) (max_x - 1)) - fs / 2);
        }
    } else {
        for (int i = 0; i < max_x; i++) {
            // Shift (f2(end:-1:size(CE,2)/2+2)=-f2(2:size(CE,2)/2);)
            int backwards_i = max_x - i - 1;

            if (backwards_i < max_x / 2 - 1) {
                wavel2[i] = -wavel2[backwards_i + 1];
            } else {
                wavel2[i] = 1 / ((FLOAT_DT) i * (fs / (FLOAT_DT) (max_x - 1)));
            }
        }
    }

    for (int i = 0; i < max_x * max_y; i++) {
        output[i] = 0.0f;
    }

    // Allocate space on GPU
    FLOAT_DT *gpu_delay1, *gpu_delay2, *gpu_wavel1, *gpu_wavel2, *gpu_image_real, *gpu_image_imag, *gpu_output;
    cudaMalloc(&gpu_delay1, max_x*max_y*sizeof(FLOAT_DT));
    cudaMalloc(&gpu_delay2, max_x*max_y*sizeof(FLOAT_DT));
    cudaMalloc(&gpu_wavel1, max_y*sizeof(FLOAT_DT));
    cudaMalloc(&gpu_wavel2, max_x*sizeof(FLOAT_DT));
    cudaMalloc(&gpu_image_real, max_x*max_y*sizeof(FLOAT_DT));
    cudaMalloc(&gpu_image_imag, max_x*max_y*sizeof(FLOAT_DT));
    cudaMalloc(&gpu_output, max_x*max_y*sizeof(FLOAT_DT));

    // Copy data to GPU
    cudaMemcpy(gpu_delay1, delay1, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_delay2, delay2, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_wavel1, wavel1, max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_wavel2, wavel2, max_x*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_image_real, image_real, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_image_imag, image_imag, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);

    cudaError_t errSync  = cudaGetLastError();
    cudaError_t errAsync = cudaDeviceSynchronize();
    if (errSync != cudaSuccess)
        printf("Sync kernel error1: %s\n", cudaGetErrorString(errSync));
    if (errAsync != cudaSuccess)
        printf("Async kernel error1: %s\n", cudaGetErrorString(errAsync));

    // 1024: max -> 1024
    int count_threads = 1024;
    long long needed_blocks = ((long long) max_x*max_y*max_x*max_y/count_threads)+1;
    calculate_pixel<<<needed_blocks, count_threads>>>(gpu_delay1,gpu_delay2,gpu_image_real,
                                                      gpu_image_imag, gpu_output, fs,max_x,max_y);
```

```
130
131     errSync  = cudaGetLastError();
132     errAsync = cudaDeviceSynchronize();
133     if (errSync != cudaSuccess)
134         printf("Sync kernel error2: %s\n", cudaGetErrorString(errSync));
135     if (errAsync != cudaSuccess)
136         printf("Async kernel error2: %s\n", cudaGetErrorString(errAsync));
137
138     // Copy result from GPU
139     cudaMemcpy(output, gpu_output, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyDeviceToHost);
140
141
142     errSync  = cudaGetLastError();
143     errAsync = cudaDeviceSynchronize();
144     if (errSync != cudaSuccess)
145         printf("Sync kernel error3: %s\n", cudaGetErrorString(errSync));
146     if (errAsync != cudaSuccess)
147         printf("Async kernel error3: %s\n", cudaGetErrorString(errAsync));
148
149     cudaFree(gpu_delay1);
150     cudaFree(gpu_delay2);
151     cudaFree(gpu_wavel1);
152     cudaFree(gpu_wavel2);
153     cudaFree(gpu_image_real);
154     cudaFree(gpu_image_imag);
155     cudaFree(gpu_output);
156     free(wavel1);
157     free(wavel2);
158 }
```

8.5 CUDA-Code mit vollständiger Parallelisierung

```
#include <stdio.h>
#include <math.h>
#include <complex.h>
#include <math_constants.h>
#include "cuda_kernel.h"

#define FLOAT_DT float
#define two_pi 6.283185307f
#define MAX(a, b) ((a > b) ? (a) : (b))

__global__ void calculate_pixel(const FLOAT_DT *delay1, const FLOAT_DT *delay2,
                                const FLOAT_DT *image_real, const FLOAT_DT *image_imag,
                                FLOAT_DT *output, FLOAT_DT fs, int max_x, int max_y) {
    // Coordinates
    long long threadID = threadIdx.x+blockIdx.x*blockDim.x;
    int x = threadID % max_y;
    int y = (threadID/max_y)%max_x;
    int z = threadID/(max_x*max_y);

    FLOAT_DT wavel1_value;

    if (max_y % 2 == 1) {
        int shifted_x = (x + ((int) max_y / 2)) % max_y;
        wavel1_value = 1 / ((FLOAT_DT) shifted_x * (fs / (FLOAT_DT) (max_y - 1)) - fs / 2);
    } else {
        int backwards_x = max_y - x - 1;

        if (backwards_x < max_y / 2 - 1) {
            wavel1_value = -(1 / ((FLOAT_DT) (backwards_x +1) * (fs / (FLOAT_DT) (max_y - 1))));
        } else {
            wavel1_value = 1 / ((FLOAT_DT) x * (fs / (FLOAT_DT) (max_y - 1)));
        }
    }

    FLOAT_DT wavel2_value;

    if (max_x % 2 == 1) {
        int shifted_y = (y + ((int) max_x / 2)) % max_x;
        wavel2_value = 1 / ((FLOAT_DT) shifted_y * (fs / (FLOAT_DT) (max_x - 1)) - fs / 2);
    } else {
        int backwards_y = max_x - y - 1;

        if (backwards_y < max_x / 2 - 1) {
            wavel2_value = -(1 / ((FLOAT_DT) (backwards_y +1) * (fs / (FLOAT_DT) (max_x - 1))));
        } else {
            wavel2_value = 1 / ((FLOAT_DT) y * (fs / (FLOAT_DT) (max_x - 1)));
        }
    }

    FLOAT_DT value = two_pi * delay1[z] / wavel1_value;
    FLOAT_DT ps1_value_real, ps1_value_imaginary;
    sincosf(value, &ps1_value_imaginary, &ps1_value_real);

    value = two_pi * delay2[z] / wavel2_value;
    FLOAT_DT ps2_value_real, ps2_value_imaginary;
    sincosf(value, &ps2_value_imaginary, &ps2_value_real);

    FLOAT_DT total_real = ps1_value_real * ps2_value_real + ps1_value_imaginary * ps2_value_imaginary;
    FLOAT_DT total_imaginary = ps1_value_real * ps2_value_imaginary - ps1_value_imaginary * ps2_value_real;

    atomicAdd(&output[z], (image_real[x + y * max_y] * total_real - image_imag[x+y*max_y]*total_imaginary)
                      * sqrtf(total_real * total_real + total_imaginary * total_imaginary));
}
```

```
64
65  // image: 2d, delay1_in: 1d, delay2_in: 1d
66  // delay1 und delay2 werden implizit transponiert
67  void AmplitudeExtractionFourier2D(FLOAT_DT *image_real, FLOAT_DT *image_imag, FLOAT_DT *delay1,
68                                    FLOAT_DT *delay2, FLOAT_DT fs_in, FLOAT_DT *output,
69                                    int max_x, int max_y) {
70      int longest_dimension;
71      FLOAT_DT fs;
72
73      longest_dimension = MAX(max_x, max_y);
74      fs = fs_in - fs_in / (FLOAT_DT) longest_dimension;
75
76      for (int i = 0; i < max_x * max_y; i++) {
77          output[i] = 0.0f;
78      }
79
80      // Allocate space on GPU
81      FLOAT_DT *gpu_delay1, *gpu_delay2, *gpu_image_real, *gpu_image_imag, *gpu_output;
82      cudaMalloc(&gpu_delay1, max_x*max_y*sizeof(FLOAT_DT));
83      cudaMalloc(&gpu_delay2, max_x*max_y*sizeof(FLOAT_DT));
84      cudaMalloc(&gpu_image_real, max_x*max_y*sizeof(FLOAT_DT));
85      cudaMalloc(&gpu_image_imag, max_x*max_y*sizeof(FLOAT_DT));
86      cudaMalloc(&gpu_output, max_x*max_y*sizeof(FLOAT_DT));
87
88      // Copy data to GPU
89      cudaMemcpy(gpu_delay1, delay1, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
90      cudaMemcpy(gpu_delay2, delay2, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
91      cudaMemcpy(gpu_image_real, image_real, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
92      cudaMemcpy(gpu_image_imag, image_imag, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyHostToDevice);
93
94      cudaError_t errSync  = cudaGetLastError();
95      cudaError_t errAsync = cudaDeviceSynchronize();
96      if (errSync != cudaSuccess)
97          printf("Sync kernel error1: %s\n", cudaGetErrorString(errSync));
98      if (errAsync != cudaSuccess)
99          printf("Async kernel error1: %s\n", cudaGetErrorString(errAsync));
100
101     // 1024: max -> 1024
102     int count_threads = 1024;
103     long long needed_blocks = ((long long) max_x*max_y*max_x*max_y/count_threads)+1;
104     calculate_pixel<<<needed_blocks, count_threads>>>(gpu_delay1,gpu_delay2,gpu_image_real,
105                                                       gpu_image_imag, gpu_output, fs,max_x,max_y);
106
107     errSync  = cudaGetLastError();
108     errAsync = cudaDeviceSynchronize();
109     if (errSync != cudaSuccess)
110         printf("Sync kernel error2: %s\n", cudaGetErrorString(errSync));
111     if (errAsync != cudaSuccess)
112         printf("Async kernel error2: %s\n", cudaGetErrorString(errAsync));
113
114     // Copy result from GPU
115     cudaMemcpy(output, gpu_output, max_x*max_y*sizeof(FLOAT_DT), cudaMemcpyDeviceToHost);
116
117     errSync  = cudaGetLastError();
118     errAsync = cudaDeviceSynchronize();
119     if (errSync != cudaSuccess)
120         printf("Sync kernel error3: %s\n", cudaGetErrorString(errSync));
121     if (errAsync != cudaSuccess)
122         printf("Async kernel error3: %s\n", cudaGetErrorString(errAsync));
123
124     cudaFree(gpu_delay1);
125     cudaFree(gpu_delay2);
126     cudaFree(gpu_image_real);
127     cudaFree(gpu_image_imag);
128     cudaFree(gpu_output);
129 }
```

8.6 Header-Datei

```
#ifndef CUDA_KERNEL_H
#define CUDA_KERNEL_H
#define FLOAT_DT float

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#ifdef __cplusplus
extern "C" {
#endif

void __declspec(dllexport) AmplitudeExtractionFourier2D(FLOAT_DT *image_real, FLOAT_DT *image_imag,
                                    FLOAT_DT *delay1, FLOAT_DT *delay2, FLOAT_DT fs_in,
                                    FLOAT_DT *output, int max_x, int max_y);

#ifdef __cplusplus
}
#endif

#endif
```

9 Abkürzungsverzeichnis

GPU Graphics Processor Unit

CPU Central Processing Unit

MEX Matlab Executable

DLL Dynamic Link Library

CUDA Compute Unified Device Architecture

RAM Random-Access Memory

USCT Ultrasound Computer Tomography

10 Quellen

Literatur

[1] Robert Koch Institut. Brustkrebs. https://www.krebsdaten.de/Krebs/DE/Content/Krebsarten/Brustkrebs/brustkrebs_node.html, letzter Zugriff: 16. Juli 2022

[2] Robert Koch Institut. Krebs in Deutschland. https://www.krebsdaten.de/Krebs/DE/Home/homepage_node.html, letzter Zugriff: 16. Juli 2022

[3] Krebsgesellschaft. Mammographie-Screening als Früherkennungsmethode. https://www.krebsgesellschaft.de/onko-internetportal/basis-informationen-krebs/krebsarten/brustkrebs/mammographie-screening.html, letzter Zugriff: 16. Juli 2022

[4] Wikipedia. Nyquist-Shannon-Abtasttheorem. https://de.wikipedia.org/wiki/Nyquist-Shannon-Abtasttheorem

[5] Wikipedia. Fourier-Transformation. https://de.wikipedia.org/wiki/Fourier-Transformation

[6] Mathworks. GPU-Coder. https://de.mathworks.com/products/gpu-coder.html, letzer Zugriff: 26. Juli 2022

[7] Mathworks. Get started with GPU-Coder. https://de.mathworks.com/help/gpucoder/getting-started-with-gpu-coder.html, letzer Zugriff: 26. Juli 2022

[8] Mathworks. Mexcuda. https://de.mathworks.com/help/parallel-computing/run-mex-functions-containing-cuda-code.html, letzter Zugriff: 25. Juli 2022

[9] NVIDIA. Cuda-C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compilation-with-nvcc, letzter Zugriff: 25. Juli 2022

[10] NVIDIA. Nvprof. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof, letzter Zugriff: 25. Juli 2022

[11] NVIDIA. NVIDIA-Visual-Profiler. https://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual-profiler, letzter Zugriff: 25. Juli 2022

[12] NVIDIA. Difference between a cpu and a gpu. https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu, letzter Zugriff: 26. Juli 2022

[13] NVIDIA. CUDA Toolkit Documentation - Atomic Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions, letzter Zugriff: 25. August 2022

[14] David Goldberg, 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic.

[15] NVIDIA. Arithmetic instructions for float and double. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions, letzter Zugriff: 26. Juli 2022

[16] NVIDIA. cuFFT. https://docs.nvidia.com/cuda/cufft/index.html, letzer Zugriff: 26. Juli 2022

[17] NVIDIA. NVCC Command Options. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#nvcc-command-options, letzer Zugriff: 08. August 2022

11 Abbildungsverzeichnis

12 Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit unter der Beratung durch Michael Zapf am Karlsruher Institut für Technologie sowie Dietmar Gruber und Norbert Krieg im Zuge des Hector-Seminars, selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden, sowie Zitate kenntlich gemacht habe.

Karlsruhe, den

.....
Rouven Kiefer