Deutsches Zentrum
für Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

DHBW
Duale Hochschule
Baden-Württemberg

Hochauflösende Prozessierung flugzeuggestützter SAR Daten auf GPU

BACHELORARBEIT

für die Prüfung zum

Bachelor of Engineering

des Studienganges Informationstechnik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Maren Künemund

27.06.2011 – 19.09.2011

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	292597, TIT08AIN
Ausbildungsfirma	Deutsches Zentrum für Luft- und Raumfahrt Oberpfaffenhofen
Betreuer der Ausbildungsfirma	Dr.-Ing. Rolf Scheiber
Gutachter der Dualen Hochschule	Dr. Harald Kornmayer

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig angefertigt wurde und ich keine weiteren als die angegebenen Quellen und Hilfsmittel verwendet habe.

Maren Künemund

Oberpfaffenhofen, den 16. September 2011

Hochauflösende Prozessierung flugzeuggestützter SAR Daten auf GPU

Zu Zeiten von Katastrophen und Großereignissen ist die Bevölkerung auf Hilfskräfte angewiesen. Vor allem die Infrastruktur des Verkehrssystem ist unter diesen Bedingungen starken Belastungen ausgesetzt und kann somit schnell zum Erliegen kommen. Damit Hilfskräfte trotz dieser erschwerten Situation rechtzeitig zum Krisenherd gelangen können, ist ein gut organisiertes Verkehrsmanagement essentiell. Wichtige Voraussetzungen sind hierbei gute Zusammenarbeit der zuständigen Behörden und ein Netzwerk aus Informationen.

Da vor allem die nötigen Informationen schwierig zu beschaffen sind, wurde das Projekt VABENE ins Leben gerufen. VABENE steht für **V**erkehrsm**a**nagement **be**i Großei**n**sätz**e**n und Katastrophen. In diesem Projekt arbeiten viele Institute des Deutschen Zentrums für Luft- und Raumfahrt zusammen, um bei Krisen zeitnah die nötigen Informationen bereitzustellen, damit zuständige Behörden verantwortungsbewusst und auf Grundlage aktueller Daten handeln können.

Das Institut für Hochfrequenztechnik und Radarsysteme beteiligt sich mit dem flugzeuggestützten Radarsystem F-SAR an diesem Projekt. Mit Hilfe des Radarsystems ist es möglich, Bilder der Erdoberfläche unabhängig von Tageszeit und Wetterlage aufzunehmen. Anders als bei optischen Messungen liegen während der Datenaufnahme noch keine vom Menschen interpretierbare Bilder vor. Erst durch Anwendung komplexer Verarbeitungsalgorithmen entsteht aus den empfangenen Radarsignalen ein visuell interpretierbares Bild.

Da im Rahmen des Projektes VABENE eine zeitnahe Informationsgewinnung im Vordergrund steht, müssen Wege gefunden werden, trotz rechenintensiver Verarbeitungsschritte, eine unmittelbare Bereitstellung der Radarbilder zu gewährleisten. Eine Hardwaretechnologie ist nötig, die es sowohl ermöglicht, hochauflösende Radarbilder in Echtzeit zu prozessieren, als auch an Bord des Flugzeuges verfügbar zu sein. Somit kommt nur eine Technologie – die Grafikkarte – in Frage.

Die Grafikkarte wurde in erster Linie für massiv parallelisierbare Aufgaben entwickelt. Vor allem in der Computergrafik fand sie daher Anwendung. In den letzten Jahren erweiterte sich jedoch die Einsatzmöglichkeit der rein auf Computergrafik spezialisierten Grafikkarte. Die Programmierarchitektur CUDA ermöglicht es, auch komplexe Prozessierungsschritte der Radarverarbeitung auf die Grafikkarte auszulagern und zu beschleunigen.

Aufgabe im Rahmen dieser Bachelorarbeit ist es, diese Verarbeitungsschritte für die Grafikkarte zu portieren und einen optimalen Weg zu finden, die Portierung einzelner Algorithmenschritte für die beiden im Institut genutzten F-SAR-Prozessierungssoftwares

Maren Künemund, 292597, TIT08AIN, DLR Oberpfaffenhofen

verfügbar zu machen.

Die Tests haben ergeben, dass eine Integrierung in die bestehende Software durch Shared Objects vorteilhaft ist. Damit bleibt die Portabilität und Flexibilität hinsichtlich kleineren Änderungen der Algorithmik gesichert.

Die Benchmarks zu den einzelnen im Rahmen dieser Bachelorarbeit realisierten Implementierungen der Radarverarbeitungsschritte zeigen auf Modulebene eine Performancesteigerung um den Faktor 10 bis 70 im Vergleich zu einer Implementierung für einen einzigen Hauptprozessor. Sie verdeutlichen somit das Potenzial dieser Hardwarearchitektur. Dennoch sind weitere Optimierungen nötig und möglich, um den Anforderungen der Echtzeitprozessierung vollends gerecht zu werden.

Ein Einsatz in dem bestehenden Onboard-System im Flugzeug ist kurzfristig nicht umsetzbar, da keine geeigente Grafikkarte gefunden wurde. Es ist zwar möglich, mehrere leistungsstarke Grafikkarten zu spezifizieren, diese gehen aber nicht mit den physischen Beschränkungen konform und können nicht unter den derzeitigen Umgebungsbedingungen im Flugzeug betrieben werden.

Da die Algorithmen zur Radarverarbeitung parallelisierbar sind, kann die Rechenleistung der Grafikkarte voll ausgeschöpft werden. Ein Portieren der gesamten Radarverarbeitung auf die Grafikkarte ermöglicht eine Echtzeitprozessierung in voller Auflösung trotz aufwändiger Algorithmen, ist aber für das F-SAR Radarsystem derzeit nicht nur offline in einem Rechnerraum oder Büro möglich.

High Resolution Processing of Airborne SAR Data on GPU

The traffic infrastructure is a sensitive system which can be disrupted easily by catastrophies and large scale events. Especially in this case citizen's security depends on timely arrival of rescue forces. But because of lacking information they often can't reach the trouble spot on time. This shows clearly how important it is for the public authorities to work together and get necessary information promptly. To gather all important information in a timely manner proves very difficult, thus the project VABENE was established. VABENE stands for Traffic Management in time of Catastrophies an Large Scale Events. Many DLR's institutes (German Aerospace Center) are cooperating to improve the information flow, so that the appropriate public authorities can take action with discrement and based most up-to-date information.

The airborne based radar system F-SAR built and operated by the Microwave and Radar Systems Institute is a major contribution to the project VABENE. With its help it is possible to take pictures of the earth independent from weather and daylight. In contrast to optical pictures it is necessary to transform the incoming signals using computationally expensive algorithms in the first place, so that humans can interprete them.

Within the work of the VABENE project there is the need to acquire high resolution data as fast as possible. Neither the possibility to compute radar images on board nor offline on the ground meets the requirements. The onboard processor only has low resolution and full resolution can't be provided in real-time.

That's the reason why CPU-based hardware architectures will not allow solving this problem. New technologies are needed and that's why general purpose graphics processing units (GPGPU) are playing an important role. Normal GPUs are specialized to work with computergraphics. With the help of the programming architecture CUDA they become a computing machine capable of solving computationally demanding tasks in short time.

GPUs are normally used for computer graphics because they are designed for massiv parallelized computations. The development of the last years shows that, because of the programming architecture CUDA, it is now possible to port radar data processing on GPU to speed up processing time.

The topic of this bachelor thesis is to implement the radar data proccessing steps for GPU usage and to find a way to integrate the software into the existing software systems of the F-SAR processor.

Tests show that integrating CUDA code in the existing software architectures is best done by implementing dynamic libraries – shared objects. With these, CUDA code is

Maren Künemund, 292597, TIT08AIN, DLR Oberpfaffenhofen

portable and can be used by every C or IDL program. It is also important to be flexible as algorithmic steps in the radar data processing chain may change.

Benchmark tests performed for every module have shown that the computing time of individual modules compared to single-core CPU processing is 10 to 70 times faster. That shows clearly the potential of the GPU architecture. But there are more possibilities left to further optimize the proccessing for full real-time processing.

An upgrade of the existing F-SAR on-board processor for GPU usage is presently not feasible. Although there are existing graphic cards satisfying the computational requirements. GPUs suitable for the existing on board processor interfaces and space are not available. Furthermore it has to be taken account of the operability in unfavourable environment.

Because the algorithms for radar data processing can be massively parallelized, the capacity of the GPU computation force can be used efficently. Therefore porting the whole radar processing to the GPU enables the radar system to calculate a full resolution radar image in real time despite of the complex algorithms used. However, because of present F-SAR hardware restrictions this performance is presently limited to offline environments.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Tabellenverzeichnis

1 Einleitung

Großveranstaltungen und Katastrophen sind eine starke Belastung für das Verkehrssystem. Ohne Koordinierung und das richtige Management kommt der Verkehr schnell zum Erliegen – Einsatz- und Rettungskräfte können bei Katastrophen nicht zum Krisenherd, die Mobilität der Bevölkerung ist erheblich eingeschränkt. Um diese Herausforderung erfolgreich zu bewältigen, gibt es verantwortliche Akteure. Diese müssen zeitnah mit entsprechenden Maßnahmen auf solche Situationen reagieren. Jedoch gibt es kein übergreifendes Verkehrsmanagement, welches nötige Informationen sofort verfügbar hat und die erforderlichen Maßnahmen ergreifen kann. Aus diesem Grund wurde vom Deutschen Zentrum für Luft- und Raumfahrt (DLR) das Projekt VABENE ins Leben gerufen. VABENE steht für Verkehrsmanagement bei Großeinsätzen und Katastrophen und bietet den zuständigen Behörden die notwendige Unterstützung in Form von Informationen zur erfolgreichen Bewältigung der Krisen.

Viele Institute des DLR arbeiten gemeinsam an diesem Projekt. Auch das Institut für Hochfrequenztechnik und Radarsysteme des DLRs in Oberpfaffenhofen leistet seinen Beitrag durch das flugzeuggestützte Radarsystem F-SAR. Ein Radarsystem hat den großen Vorteil, vom Wetter und der Tageszeit unabhängig zu sein. Zudem ist es mit dem Flugzeug möglich, schnell und ohne vom Verkehrsnetz abhängig zu sein, in das entsprechende Einsatzgebiet zu fliegen. Dabei ermöglicht das Radarsystem nicht nur, Bilder zu erzeugen, die einem optischen Bild sehr ähnlich sind, sondern auch Informationen über Geschwindigkeit, Form und Beschaffenheit von verschiedenen Objekten in Erfahrung zu bringen. Damit liefert das F-SAR einen wesentlichen Beitrag zur Informationsgewinnung für das Projekt VABENE.

Ein Nachteil dieser Technologie ist jedoch die Auswertung der Daten. Im Gegensatz zu optischen Bildern, müssen die Radardaten mit aufwändigen Algorithmen verarbeitet werden. Um schnell Maßnahmen ergreifen zu können, ist deshalb eine Echtzeitprozessierung notwendig, die an Bord des Flugzeuges geschieht. Da jedoch bei hochaufgelösten Bildern Daten im Gigabyte-Bereich anfallen, hat dies spezielle Anforderungen an den Rechner im Flugzeug. Zum einen kann aus physikalischen Gründen kein großer Supercomputer im Flugzeug installiert werden und zum anderen müssen die Daten so hochaufgelöst und zeitnah wie möglich verarbeitet werden. Eine Brücke zwischen diesen beiden Extrema schafft die Grafikkarte – ein kleiner Superrechner, der für massiv parallelisierbare Aufgaben ausgelegt ist.

In den letzten Jahren hat sich die Grafikkarte von einer Bildverarbeitungseinheit immer

mehr zu einer hochperformanten Allzweckrecheneinheit entwickelt. Sie ist im Gegensatz zu Supercomputern wesentlich günstiger und für den beschriebenen mobilen Einsatz verwendbar.

Im Rahmen dieser Bachelorarbeit wurden Algorithmen zur Radarverarbeitung für die Grafikkarte implementiert und für die im Institut verwendeten Prozessierungssoftware angepasst. Dazu wurden über verschiedene Möglichkeiten zur Integration in diese untersucht und Betrachtungen zur Performance angestellt. Abschließend wurde eine Auswertung durch, um die Qualität der Daten zu prüfen und ein Benchmarking aufzustellen. Damit kann überprüft werden, ob die gesetzten Anforderungen erfüllt werden können.

Kapitel 2 beschreibt knapp die theoretischen Grundlagen der Algorithmen. Die Umsetzung auf die Grafikkarte und die Auswertung verschiedener Tests wird in Kapitel 3 und 4 ausführlich behandelt. Abschließend untersucht Kapitel 5 Fragen der Spezifikation einer Grafikkarte, die für den mobilen Einsatz im Flugzeug geeignet ist.

2 Stand der Technik

Um die Radarverarbeitung auf die Grafikkarte auszulagern, ist neben den Kenntnissen über Hardware und Architektur der GPU auch Kenntnis über den SAR-Prozessor von Nöten.

In den folgenden Kapiteln sollen gängige Algorithmen zur Radarsignalverarbeitung kurz beschrieben werden und eine Einführung in die Prozessierung von SAR-Daten gegeben werden (SAR = Synthetic Aperture Radar, siehe Quelle [KH00]). Dieses Wissen bildet die Grundlage zum besseren Verständnis der Algorithmen, die auf die Grafikkarte übertragen werden sollen.

Neben Grundlagen der SAR-Verarbeitung soll auch ein aktueller Stand zur eingesetzten Hardware und deren Nutzung gegeben werden. Gemeint ist hier die Grafikkarte. Da Algorithmenumsetzungen auf der Grafikkarte momentan sehr gefragt sind, wird viel hinsichtlich der Schnittstellen getan. Neue Programmierumgebungen werden veröffentlicht und alte weiterentwickelt. Aus diesem Grund soll der aktuelle Stand zum Zeitpunkt der Erstellung dieser Arbeit festgehalten und beschrieben werden.

Eine kurze Erörterung, warum sich ein Umstieg auf die Grafikkarte lohnt und welche Versuche bereits angestellt wurden, beendet dieses Kapitel.

2.1 Die Radarverarbeitung

Ein Radarsystem sendet hochfrequente, elektromagnetische Wellen aus, sogenannte Mikrowellen. Diese elektromagnetischen Wellen werden in Abhängigkeit von der Beschaffenheit der bestrahlten Objekte reflektiert. Dabei reagieren Mikrowellen auf die dielektrischen und geometrischen Eigenschaften der Zielobjekte. Das zurückgestreute Signal enthält somit verschiedene Informationen über die Beschaffenheit.

Ein auf eine mobile Plattform angebrachtes Radarsystem macht sich die Eigenschaften von Mikrowellen zunutze. Es sendet in regelmäßigen Abständen ein sehr kurzen Impuls Mikrowellen aus. Diese treffen im Zielgebiet auf Objekte, die das Signal unterschiedlich zurückstreuen. Anhand der Laufzeit, Phasendifferenz und der Stärke des Empfangssignals können die Daten zu einem Bild prozessiert werden, welches dem eines optischen Bildes sehr ähnlich sein kann.

Um eine verhältnismäßig hohe Auflösung zu erzielen, können zwei Wege genommen werden. Einerseits wird die Pulsdauer so kurz wie möglich gehalten, andererseits wird eine hohe Bandbreite für das Signal verwendet. Mit dem Verkürzen der Pulsdauer, sinkt jedoch die ausgestrahlte Leistung des Signals. Zur Vermeidung dieses Konfliktes verwendet wird ein spezielles linear frequenzmoduliertes Signal verwendet, den sogenannten Chirp (siehe Abbildung 2.1).

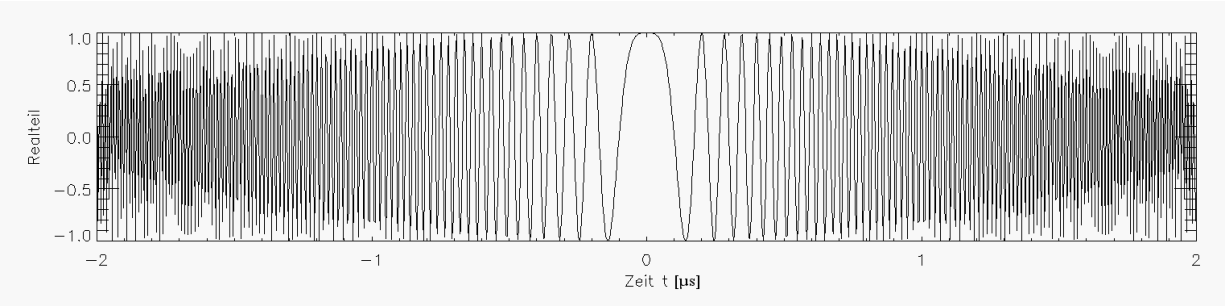


Abbildung 2.1: Realteil eines Chirpsignales

Ein Chirp ist eine komplexe Funktion und lässt sich folgendermaßen mathematisch ausdrücken (entnommen aus [Mar06], Seite 7):

$$\tilde{f}(t) = \exp\left[j\left(\omega t + \frac{\alpha t^2}{2}\right)\right] rect\left[\frac{t}{\tau}\right] \tag{1}$$

Wird also ein moduliertes Signal gesendet, kann die Auflösung erheblich erhöht werden. Auf der Empfangsseite bestehen die zurückgestreuten Radarimpulse dementsprechend auch

aus modulierten Signalen. Die Aufgabe der Auswertungssoftware liegt nun darin, das Signal eines Punktzieles auf einen Punkt zu fokussieren. In der Prozessierung wird dies als Kompression bezeichnet. Abbildung 2.2 zeigt das empfangene Signal vor und nach der Kompression.

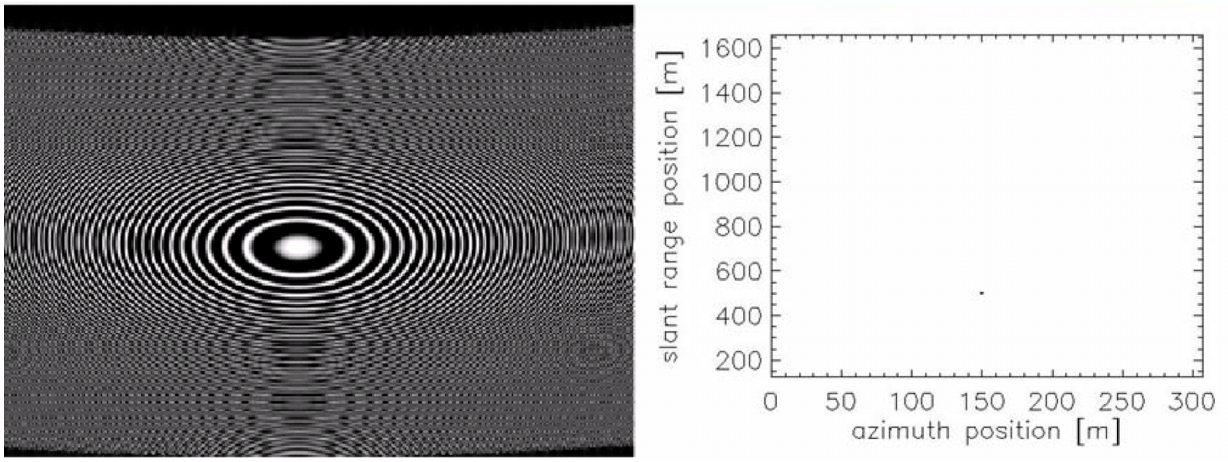


Abbildung 2.2: Empfangssignal eines Zielpunktes, links vor und rechts nach der Komprimierung

Im Institut für Hochfrequenztechnik und Radarsysteme wird an einem auf einem Flugzeug angebrachten Radarsystem gearbeitet, dem sognenannte F-SAR. Abbildung 2.3 zeigt die idealisierte Geometrie dieser Aufnahmetechnik. Zur Vereinfachung des Modells werden eine geradlinige Flugbahn, sowie eine konstante Geschwindigkeit während der Aufnahme der Daten angenommen.

In der Grafik 2.3 sind verschiedene Kenngrößen angegeben. Darunter die Impulsdauer τ, mit der die Antenne Radarimpulse senkrecht zur Flugrichtung und schräg nach unten sendet. Die Frequenz mit der die Signale wiederholt werden wird als PRF (Pulse Repetition Frequency) bezeichnet. Der Blickwinkel ϑ und Öffnungswinkel θ beeinflussen die Antennenbeleuchtungsfläche, das durch den Radarimpuls beleuchtete Gebiet. Durch die Bewegung der Radarplattform wird ein Abbildungsstreifen nach und nach vermessen. Eine weitere wichtige Kennzahl ist die Länge der Antenne l_a. Bei einem synthetischen Radar wird diese künstlich verlängert, um so unabhängig von der Entfernung r zum Zielgebiet eine hohe Auflösung zu erzielen (maximal $\frac{l_a}{2}$).

Eine ausführlichere Einleitung in die Grundlagen der SAR-Theorie bieten die Quellen [Fis02, Mar06, Aul05]. Für eine tiefere Auseinandersetzung mit der Thematik, sind die Quellen [KH00, rad91] zu bevorzugen.

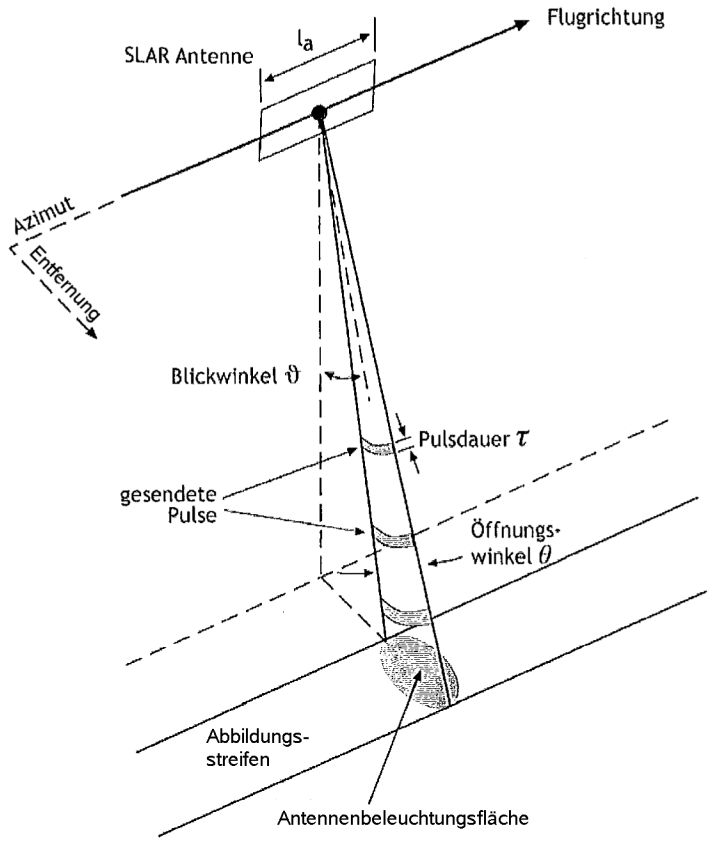


Abbildung 2.3: Aufnahmegeometrie eines abbildenden Radarsystems (entnommen aus Quelle [Aul05], Seite 29)

Um die nötigen Bildinformationen aus den empfangenen Signalen zu gewinnen, ist eine Verarbeitung der Daten zwingend notwendig. Im Gegensatz zur Verarbeitung von optischen Daten, ist die Radarprozessierung sehr aufwändig. Aus diesem Grund existieren verschiedene Varianten und Algorithmen, um die Berechnungen zu optimieren. Einer dieser Algorithmen, der im Rahmen dieser Arbeit für die Grafikkarte umgesetzt wurde, ist der Extended Chirp Scaling Algorithmus [MMS96]. Dieser macht Gebrauch von mehreren Fouriertransformationen, um den Aufwand der Berechnung zu verringern. Da eine sehr effiziente Methode zur Umwandlung von Daten in den Frequenzraum existiert, ist diese Art der Prozessierung wesentlich performanter als eine direkte Berechnung im Zeitbereich.

Hierbei erfolgt eine Prozessierung der Daten abwechselnd in Azimut (Flugrichtung) und in Entfernungsrichtung.

Einen generellen Ablauf des Extended Chirp Scaling Algorithmus' liefert die Abbildung

2.4. Bis auf die erste Bewegungskompensation wurden alle Schritte im Rahmen dieser Bachelorarbeit für die Grafikkarte implementiert. Eine zugehörige Radardatenvorverarbeitung wurde bereits in der Praktikumsarbeit [Kü10] implementiert.

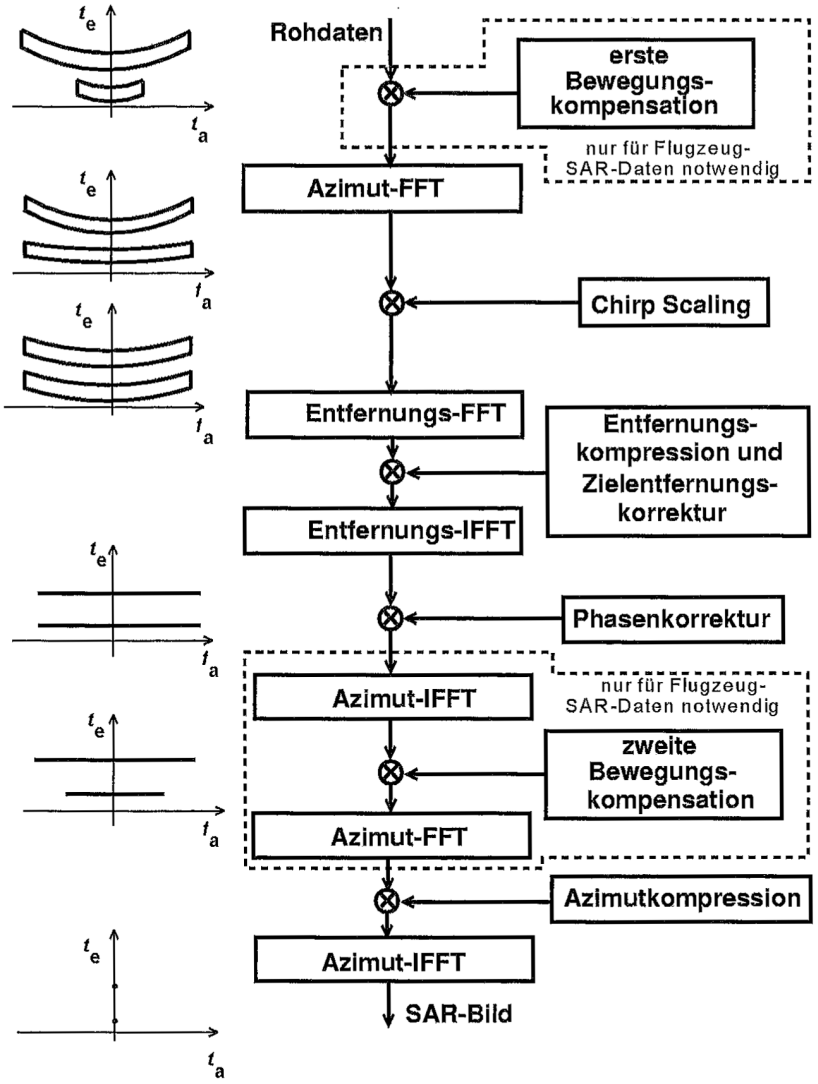


Abbildung 2.4: Blockdiagramm zur Datenverarbeitung mit Hilfe des Extended Chirp Scaling Verfahrens. Links ist die Konturdarstellung der SAR-Daten für zwei Punktziele in den verschiedenen Verarbeitungsschritte dargestellt. (entnommen aus Quelle [KH00], Seite 250)

Die Radardatenvorverarbeitung beinhaltet bereits eine Bewegungskompensation erster Ordnung. Hierbei werden Abweichungen von der eigentlichen Flugbahn herausgerechnet,

um so die Qualität des Endbildes erheblich zu verbessern. Danach folgt das Resampling. Dieser Schritt beinhaltet eine Interpolation (Neuabtastung) der Daten auf eine neue Achse. Das Presumming, welches die Daten in der Azimut-Richtung um den Presumming-Faktor verkleinert, schließt die Radardatenvorverarbeitung ab.

Im ersten Schritt der Prozessierung findet das Chirp Scaling statt. In diesem werden die Phasenzentren der Punktziele verschoben. Danach ist es möglich, eine Entfernungskompression durchzuführen. Bevor eine Azimutkompression stattfinden kann, muss zunächst eine Zielentfernungs- und Phasenkorrektur vorgenommen werden.

Durch die Vorwärtsbewegung der Plattform kommt eine Änderung der Entfernung zum Punktziel zustande. Diese Entfernungen der Plattform zum Punktziel r in Abhängigkeit von der Zeit t bilden einen hyperbolischer Funktionsverlauf, welcher in der Zielentfernungskorrektur kompensiert wird. Weiterhin entsteht durch die vorhergehenden Prozessierungsschritte eine entfernungsabhängige Phasenverschiebung. Noch vor der Azimutkompression muss dies in der Phasenkorrektur herausgerechnet werden.

Da sich das Radarsystem auf einer mobilen Plattform befindet, kann es zu kleinen Abweichungen kommen. Zwar wurde in der Radardatenvorverarbeitung die erste Bewegungskompensation durchgeführt. Der Restfehler wird jedoch erst jetzt in der zweiten Bewegungskompensation herausgerechnet. Zu guter Letzt erfolgt die Azimutkompression.

Nach diesen Schritten erhält man einen komplexwertigen Datensatz, dessen Betrag ein monochromatisches Bild ergibt. Abbildung 2.5 zeigt beispielhaft ein Radarbild.

Für die komplette theoretische Herleitung und Beschreibung der Grundlagen des Extended Chirp Scaling sei hier auf die Quelle [MMS96] verwiesen.

2.2 Programmierung auf der Grafikkarte

Die Geschichte der Grafikkarte reicht bis in die frühen 1980er Jahre zurück. Damals war die Grafikkarte nur ein einfaches Bindeglied zwischen dem Hauptprozessor und der Bildschirmausgabe. Sie übernahm noch keine Rechenaufgaben. Als sich die Spiele- und Videoindustrie weiterentwickelte und bei den Kunden die Ansprüche wuchsen, entwickelte sich die Graphics Processing Unit (GPU) zu der Recheneinheit, wie man sie heute kennt. Da sie vor allem in der Computergrafik eingesetzt wurde, übernimmt sie massiv parallelisierbare Aufgaben.

Unlängst wurde von der Herstellerfirma NVidia erkannt, dass sich die Recheneinheit nicht nur zur Verarbeitung von Bild- und Videodaten zu Darstellungszwecken eignet. Mit dieser Erkenntnis war die General Purpose GPU (GPGPU) geboren, die das Aufgabenge-

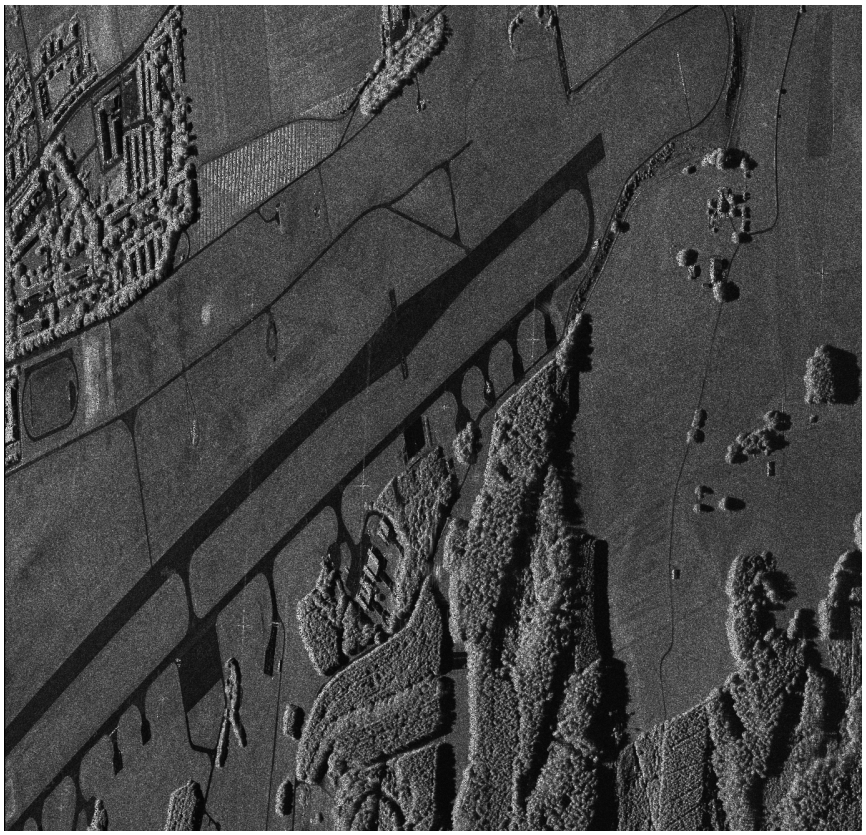


Abbildung 2.5: Mit der GPU prozessiertes Radarbild des Testgebiets Kaufbeuren aufgenommen mit F-SAR, DLR

biet einer gewöhnlichen GPU um ein Vielfaches erweitert.

Dafür muss natürlich eine Schnittstelle zwischen Programmierer und Hardware geschaffen werden. Hier hat NVidia eine Vorläuferstellung eingenommen, in dem sie die Compute Unified Device Architecture (CUDA) entwickelte. Auch AMD unternahm einige Versuche, sich in diesem Bereich mit ATI Stream zu etablieren. Durchsetzen konnte sich bis heute nur die von NVidia entwickelte Architektur CUDA und die Bibliothek OpenCL, bei der auch NVidia einen großen Anteil an der Entwicklung trägt.

Abbildung 2.6 zeigt die Entwicklung der Rechenkapazitäten der Grafikkarten im Vergleich zu herkömmlichen Prozessoren. Diese Bilder zeigen das hohe Rechenpotential der Grafikkarten.Im Gegensatz zu Supercomputer und Cluster, die mit der Rechenkapazität einer Grafikkarte vergleichbar sind, sind sie wesentlich günstiger. In Bereichen wie Wissenschaft, Videokodierung vielem mehr kann die Grafikkarte Anwendung finden. Bekannte Produkte wie Adobe Photoshop und PhysX machen von dieser Technologie gebrauch.

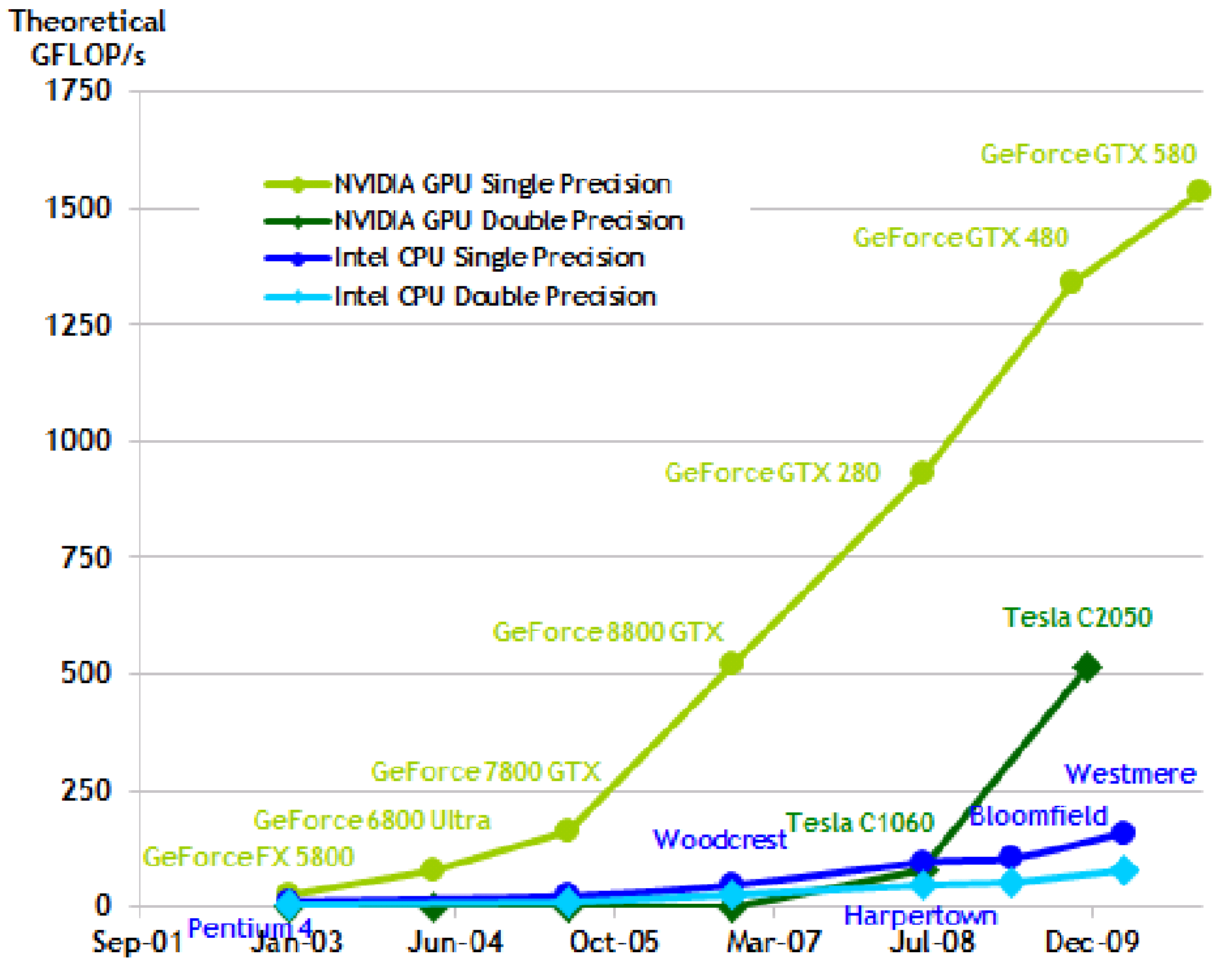


Abbildung 2.6: Floating-Point Operations per Seconds (FLOP/s) von GPUs und CPUs entnommen aus [NVi11b], Seite 2

Wie in Quelle [Kü10] in Kapitel 3.4 dargestellt, bietet OpenCL eine Kompatibilität zu anderen Hardwaresystemen wie AMDs Radeons. Da jedoch für den wissenschaftlichen Bereich hauptsächlich NVidias Teslas zum Einsatz kommen, lohnt eine Programmierung in OpenCL nicht. Zumal diese auch wegen ihrer hohen Kompatibilität nicht an die Effizienz von CUDA herankommen. Hierzu wurde in Quelle [PW10] ein Benchmark durchgeführt, welches die Implementierung von SAR Algorithmen mit CUDA und OpenCL vergleicht. Die Ergebnisse zeigen deutlich, dass auf Grund der hardwarespezifischen Implementierung von CUDA eine höhrere Performance erzielt werden kann. Dies geht jedoch mit der Inkompatibilität zu anderen Grafikkarten einher.

Weiterhin ist OpenCL eine vergleichsweise junge Entwicklung, während CUDA bereits

über mehrere Jahre entwickelt wurde. CUDA ist demnach sehr stabil, hat sehr viele Bibliotheken für mathematische Funktionen und eine große Gemeinschaft im Internet.

Bereits viele andere wissenschaftliche Berichte setzen sich ausführlich mit der Radarprozessierung auf der Grafikkarte auseinander, beispielsweise [HFB+09, PW10]. Alle kamen zu dem Ergebnis, dass eine Auslagerung der Prozessierung auf die GPU erhebliche Geschwindigkeitsvorteile bringt. Sowohl eine Implementierung in CUDA als auch in OpenCL ist bei zeitkritischen Anwendungen sinnvoll. In den Quellen [HFB+09, LWLY09, BF05, RSB, LWY09] gibt es einige Berichte, die sich auch mit der Implementierung der Radarprozessierung auf die Grafikkarte beschäftigt haben. Da die aufwändigsten Rechenoperationen wie Fourier-Transformationen und Matrixoperationen sehr gut parallelisierbar sind, haben sich bereits verschiedene Forschungseinrichtungen intensiv damit auseinander gesetzt.

Dies zeigt deutlich, dass Grafikkarten geeignet sind, um hochaufgelöste SAR-Bilder in Echtzeit zu prozessieren.

Wesentliche Grundlagen

Zu diesem Zweck soll ein grober Umriss der Architektur von CUDA erfolgen. Da bereits im Bericht des 4. Semesters (Quelle [Kü10]) eine ausführliche Einführung in diese Thematik erfolgte, sei hier nur auf allgemeine Schlagworte hingewiesen. Für eine ausführliche Beschreibung kann auch der Programming Guide (Quelle [NVi10a]) zu Rate gezogen werden.

Die Architektur der Grafikkarte ist so ausgelegt, dass sie sehr viele Prozessoren besitzt, die alle parallel laufen. Statt, wie bei der Central Processing Unit (CPU) auf einen hohen Kontrollfluss wert zu legen, läuft in der Regel nur ein einzelnes Programm – der Kernel – auf der GPU. Die aktuelle Entwicklung zeigt jedoch, dass bis zu 16 Programme gleichzeitig auf der Grafikkarte laufen können (Quelle [NVi10a] Seite 38). Die Berechnungen werden auf die einzelnen Prozessoren verteilt. Dies ermöglicht massive Parallelisierung.

Der Kernel durchläuft auf jedem Prozessor den gleichen Programmcode, jedoch sind die Daten, auf denen er arbeitet, unterschiedlich. Ein idealer Anwendungsfall wäre beispielsweise eine Multiplikation einer sehr großen Matrix mit einem Faktor. Jeder Prozessor bekommt die Aufgabe, ein bestimmtes Element der Matrix mit dem Faktor zu multiplizieren. Da dies alles gleichzeitig geschieht, können hochperformante Anwendungen geschrieben werden.

Bottlenecks jeder Grafikkarten-Anwendung sind jedoch die Übertragungs- und Zugriffszeiten auf den Speicher. Bevor von der Grafikkarte auf die Daten zugegriffen werden kann, müssen diese in den Speicher der GPU übertragen werden. Auf dieser gibt es viele verschiedene Speichertypen, die je nach Anwendungsfall und Größe der Daten benutzt werden

können. Ein gutes Beispiel hierfür bietet die Quelle [LWLY10], die den Texture Memory nutzt, um einen Schritt der Radarverarbeitung auf der Grafikkarte zu beschleunigen.

3 Radarverarbeitung auf der Grafikkarte

Als das Projekt F-SAR startete, wurde eine Softwarearchitektur in der Programmiersprache C++ entwickelt. Diese bot die Möglichkeit, Prozesse modular einzubinden. Ein Ausführen eines einzelnen Prozesses ist möglich, aber auch Abhängigkeiten zu Ausgaben von anderen Prozessen können hergestellt werden. Somit entstand eine Prozesskette, die zu einem Workflow führt. Dieser Workflow kann zum Beispiel eine komplette Radarprozessierung beschreiben.

Zum Debuggen und Auswerten der Daten wurde häufig auf die Programmiersprache IDL zurückgegriffen. Mit der Zeit entschied man sich, eine Umsetzung der Algorithmen zur Radarverarbeitung auch in IDL vorzunehmen. IDL besticht vor allem durch seine einfache und intuitive Nutzung mit der Möglichkeit, einzelne Befehle im Interpretationsmodus auszuführen. Ein falscher Programmcode kann schnell debuggt und Daten auf dem Bildschirm visualisiert werden.

Diese Entwicklung hatte zur Folge, dass in der Abteilung SAR-Technologien zwei Implementierungen vorhanden sind, die gewartet werden müssen. Soll nun für beide Implementierungen die Möglichkeit bestehen, rechenintensive Schritte auf die Grafikkarte auszulagern, so muss ein Weg gefunden werden, wie diese am effektivsten für beide umgesetzt werden kann.

Das folgende Kapitel wird sich mit dieser Problematik auseinander setzen, die einzelnen Vor- und Nachteile diskutieren und darstellen, auf welche Art und Weise die GPU-Implementierung letztendlich umgesetzt wurde. Anschließend folgt eine Beschreibung der Tests zur Untersuchung der Qualität und Laufzeit der einzelnen Module bzw. der gesamten Radarverarbeitung. Darauf folgt die Ausführung zu den einzelnen Modulen mit einer abschließenden Auswertung der Tests.

3.1 Wahl der Programmierschnittstelle zur Grafikkarte

Bevor im einzelnen auf die Implementierung im Speziellen eingegangen wird, soll zunächst betrachtet werden, welche Möglichkeiten vorliegen. Um möglichst effektiv die Grafikkarte anzusprechen, ist eine Programmierung in C mit CUDA Erweiterungen die optimale Lösung. Für IDL gibt es keine direkte Syntaxerweiterung, die ein Programmieren auf der

Grafikkarte erlaubt. Stattdessen ist die kostenpflichtige GPUlib [Mes09] vorhanden, die für IDL einige Funktionalitäten bereithält.

Somit wäre es möglich, zwei Implementierungen umzusetzen. Auf der einen Seite C mit CUDA-Erweiterungen und auf der anderen Seite die einfach zu benutzende Bibliothek für IDL. Dies hat natürlich den großen Nachteil, dass zwei Implementierung gewartet und parallel geändert werden müssen. Auch das Entwickeln auf diese Art gestaltet sich schwierig, da ein Portieren zwischen IDL und C nur sehr schwer möglich ist. Im Grunde genommen würde diese Art ein Entwickeln von zwei verschiedenen Programmen bedeuten.

Aus diesem Grund soll neben der GPUlib auch eine dynamische Bibliothek näher untersucht werden. Mit dynamischen Bibliotheken ist es nicht nur möglich, den CUDA-Code für die in C entwickelte Software zu kapseln, sondern ihn weiterhin auch in IDL einzusetzen. Dies wird über die Schnittstelle von dynamisch ladbaren Modulen (DLM) von IDL ermöglicht. Quelle [ITT09] gibt eine ausführliche Dokumentation über dieses Thema. Aus den Eigenschaften einer dynamischen Bibliothek ergeben sich verschiedene Vorteile:

- Die Implementierung des CUDA-Codes erfolgt nur einmal.
- Die Nutzer der dynamischen Bibliothek müssen bei Änderungen der Bibliothek kein Neukompilieren vornehmen.
- Die dynamische Bibliothek kann universell eingesetzt werden. Das heißt, sie wird mit CUDA-spezifischen Code kompiliert. Der nutzende Quellcode braucht davon nichts zu wissen und keine speziellen Bibliotheken einbinden. Die einzige Abhängigkeit besteht zur dynamischen Bibliothek selbst.
- Die Schnittstellen werden über eine Header-Datei definiert und gelten für alle Nutzer dieser Bibliothek.
- Die Bibliothek wird erst zur Laufzeit dynamisch zum Quellcode hinzugefügt.

Eine Implementierung dieser Art kann nach dem Schema in Abbildung 3.1 umgesetzt werden. Zum einen wird ein Shared Object, also eine dynamische Bibliothek für den CUDA-Code erstellt. Für den C++ Code des F-SAR Prozessors reicht es aus, den Header der Bibliothek zu inkludieren. Zu beachten ist jedoch, dass die Konventionen bei einem Header in C anders sind als in C++. Aus diesem Grund wird der Header der dynamischen Bibliothek mit Makros umgeben, die ein Verbinden von C und C++-Headern ermöglicht. Beim Kompilieren muss weiterhin ein Flag angegeben werden, um den Kompiler über den Namen der dynamischen Bibliothek zu informieren.

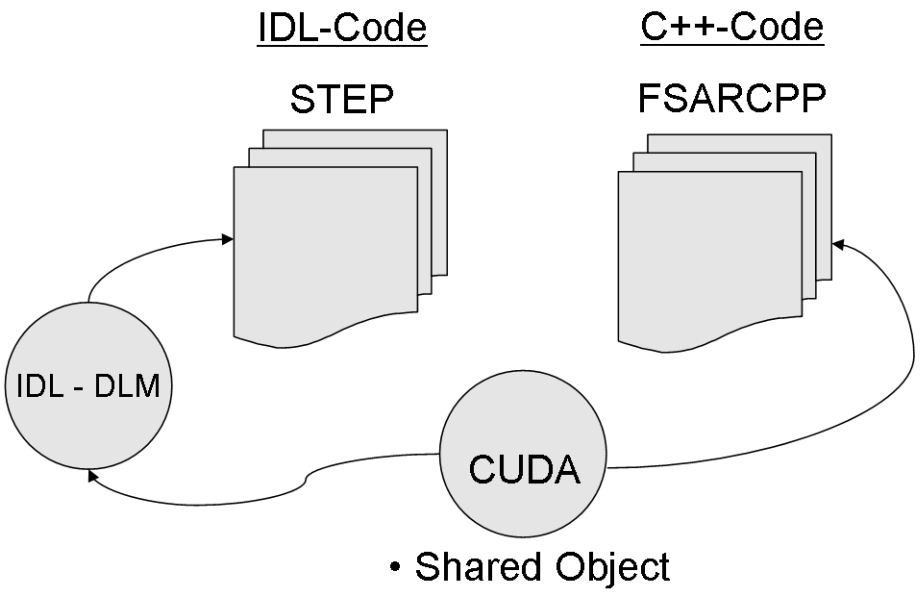


Abbildung 3.1: Schema zur Verwendung der dynamischen Bibliothek

Auf der IDL-Seite sieht es etwas komplizierter aus. Hier hängt ein Shared Object dazwischen, welches die Aufgaben der Schnittstelle von IDL zu C übernimmt. Dieses Shared Object definiert die Funktions- und Prozedurnamen wie sie später in IDL benutzt werden. Auch die Anzahl der Parameter werden hier festgelegt. Anschließend erfolgt ein Konvertieren der IDL-Datentypen in C-Datentypen. Da IDL eine schwach typisierte Sprache ist, muss vor dem Aufruf der Bibliotheksfunktion auf Richtigkeit der Datentypen geprüft werden.

Im Folgendem soll untersucht werden, wie sich die GPUlib im Verhältnis zu der Shared Object Variante verhält und ob sich ein doppeltes Implementieren der Algorithmen lohnt. Tabelle 1 zeigt die Spezifikation der für die Tests benutzten Grafikkarte.

Vergleich der dynamischen Bibliotheken mit GPUlib

In diesem Abschnitt werden die einzelnen Rechenzeiten für die verschiedenen Implementierungen untersucht. Hierbei wurden die Shared Objects für IDL (in den Legenden als IDL abgekürzt) und C++ (abgekürzt mit C) mit Beispielimplementierungen mit der GPUlib verglichen. Die Programme in den Beispielimplementierungen sehen dabei wie folgt aus:

Ein Testdurchlauf für C, IDL bzw. GPUlib besteht aus 100 Programmaufrufen. Das heißt, ein Durchlauf startet und beendet das Programm vollständig. Abbildung 3.2 zeigt die Rechenzeiten der drei Implementierungen für eine Interpolation mit einem zweidimen-

Name	NVidia Tesla C2070
Anzahl Tesla-Grafikprozessoren	1
Höhe der 64-Bit Gleitkommagenauigkeit	515 GigaFLOPS
Höhe der 32-Bit Gleitkommagenauigkeit	1.03 TFLOPS
Gesamter eigener Speicher	6 GB GDDR5
Speichertyp	1.5 GHz
Speicherbandbreite	144 GB/s

Tabelle 1: Spezifikation der genutzten Grafikkarte, entnommen aus [Nvi]

```
1 reading permanent data
2 cast data
3 copy data to gpu
4 launch kernel
5 back copy data to ram
6 write data permanently
```

Quelltext 1: Programmstruktur

sionalen 512×1024 Pixel großen Float-Arrays. Durch die Interpolation wird das Array auf eine Größe von 2048×4096 Pixeln skaliert. In der eigenen Implementierung wurden die Daten mit Hilfe von Texturen bilinear interpoliert. Eine Ausführung hierzu steht zum einen in Quelle [NVi10a, Far09] und zum anderen in Quelle [Kü10] zur Verfügung. Da die GPUlib bereits einige Funktionen zur Interpolation bereitstellt, wurden diese für den Testlauf herangezogen.

Wie im Bild zu erkennen ist, liegt der Aufruf aus dem C-Code mit rund zwei Sekunden weit hinten. Obwohl der Aufruf aus IDL den gleichen CUDA-Code benutzt, benötigt er weniger als eine halbe Sekunde. Tabelle 2 zeigt statistische Kennzahlen der 100 Durchläufe. Das Maximum liegt bei jedem zwischen zwei und drei Sekunden. Im Gegensatz dazu haben sich IDL und die GPULib auf eine durchschnittliche Rechenzeit von weniger als eine Sekunde eingepegelt. Dies kann auf die dynamische Bibliothek zurückgeführt werden, auf die später noch ausführlicher eingegangen wird.

Zu erkennen ist, dass die GPULib eine gute Performance hat. Positive Merkmale sind vor allem das einfache Implementieren der Interpolation. Mit wenigen Zeilen Code konnte die Interpolation durchgeführt werden, während es sich im CUDA Quellcode anders her-

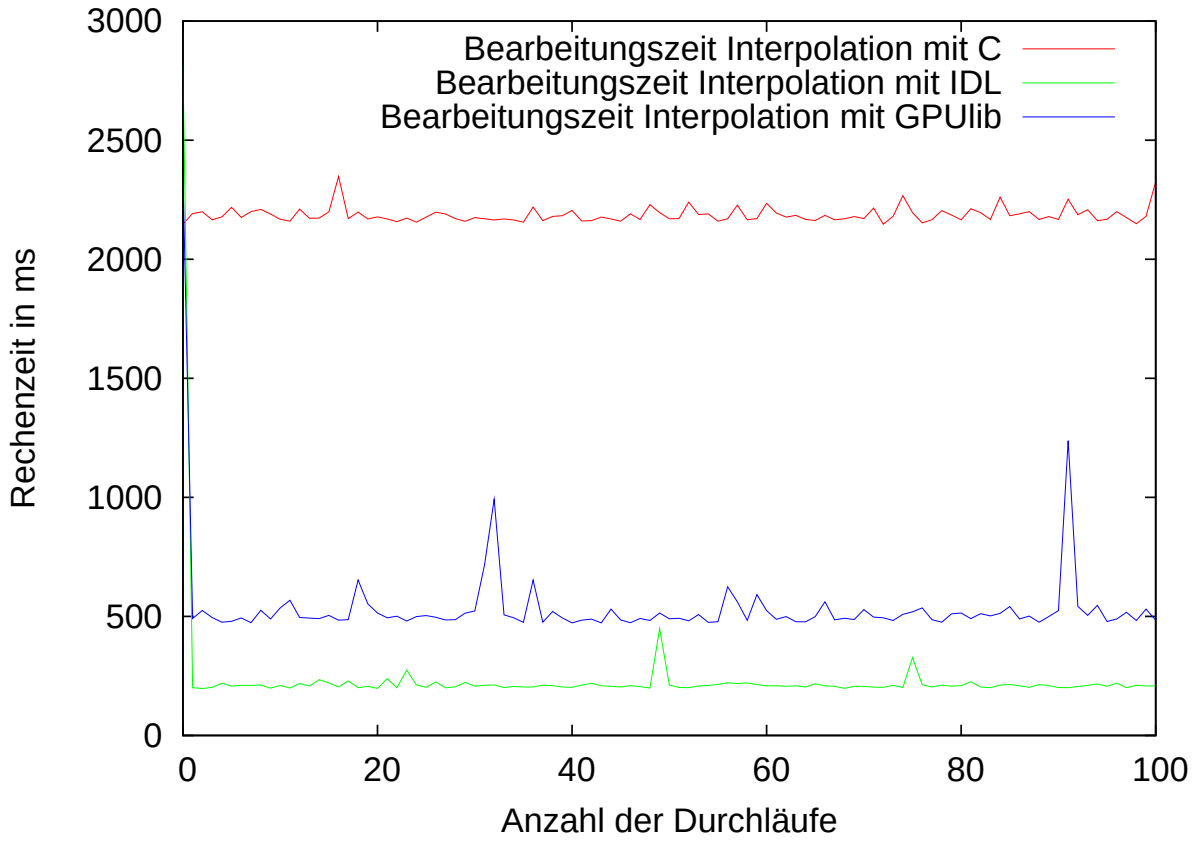


Abbildung 3.2: Vergleich der Rechenzeit für eine Interpolation in C, IDL und GPUlib

ausstellt. Eine Gegenüberstellung dieser Quellcodes kann im Anhang unter Quelltext 5 und 6 nachvollzogen werden. Warum die GPUlib trotz dieser Eigenschaften nicht für die Entwicklung geeignet ist, zeigt die Abbildung 3.3.

Auch hier hat die GPUlib geringere Rechenzeiten im Vergleich zur C-Implementierung. Jedoch erfolgen nach 40 Durchläufen keine Berechnungen mehr. Dieser zweite Testdurchlauf bestand wieder aus 100 Programmaufrufen. Diesmal wurde eine Fast Fourier Transformation (FFT) mit 2048×8192 komplexwertigen Pixeln durchgeführt. Das ergibt eine unkomprimierte Datenmenge von $128MB$. Für eine Fouriertransformation werden also $256MB$ benötigt, um die Operation durchzuführen. Der Grund des Abbruchs der Berechnung durch die GPUlib kommt durch ein Speicherleck in der Programmierung zu Stande. Wie auch schon bei der Interpolation, werden die Variablen auf der Grafikkarte angelegt, aber beim Beenden des Programms nicht wieder freigegeben. Da bei der Interpolation nur mit kleinen Daten gearbeitet wurde, konnten die 100 Durchläufe reibungslos durchgeführt

	Minimale Rechenzeit in ms	**Maximale Rechenzeit in ms**	**Durchschnittliche Rechenzeit in ms**
IDL	197,238	2816,330	238,937
GPULib	472,747	2358,150	538,460
C	2147,000	2348,000	2185,910

Tabelle 2: Kennzahlen der Durchläufe für eine Interpolation auf der Grafikkarte

werden. Bei der FFT ist der Grafikkartenspeicher schließlich nach 40 Durchläufen voll. Da die Implementierung der FFT und Interpolation für GPUlib frei zugänglich sind, könnte man dieses Speicherleck suchen. Weiterhin gäbe es die Möglichkeit, die neuste Version der Bibliothek herunterzuladen. Doch leider ist diese Version mittlerweile nicht mehr kostenfrei und ein Debuggen für diese Aufgabe zu zeitaufwändig.

Aus diesen Ergebnissen zeichnet sich deutlich ab, dass sich GPUlib für diese Arbeit nicht eignet. Nicht nur, dass sie inkompatibel zur C-Implementierung ist, sie weist auch viele Fehler auf und ist kostenpflichtig. Ein einfaches Programmieren eines individuellen Kernels ist nicht möglich. Die Möglichkiten schränken sich durch die bereitgestellten Kernelfunktionen der Bibliothek ein.

Zusammenhang zur Anzahl der genutzten Shared Objects

Im Folgenden soll herausgefunden werden, ob und wie die Nutzung zweier Shared Objects sich auf die Rechenzeit auswirken. Hierzu wurden wieder 100 Durchläufe durchgeführt und die entsprechenden Zeiten aufgenommen. Zuerst soll eine Betrachtung folgen, die die IDL- und C-Implementierung miteinander vergleicht. Der Ablauf hat sich dabei nicht geändert, es wird lediglich das Programm zur Berechnung nicht beendet. Das heißt, das Programm wird gestartet, die Berechnung erfolgen 100-mal aufs Neue und erst danach beendet sich das Programm. Abbildung 3.4 zeigt den Vergleich der Rechenzeiten zwischen IDL und C.

Wie man sehr gut erkennen kann, haben beide beim ersten Aufruf eine Rechenzeit von etwas mehr als zwei Sekunden. Dies wird durch die genauen Zahlen in der Tabelle 3 unterstützt.

Zu erklären ist dies durch das erstmalige Suchen des Shared Objects beim Starten des Programms. Dazu wird auf dem Rechner nach der aktuellen Version gesucht und in den Hauptspeicher geladen. In allen nachfolgenden Aufrufen wird mit der geladenen Implemen-

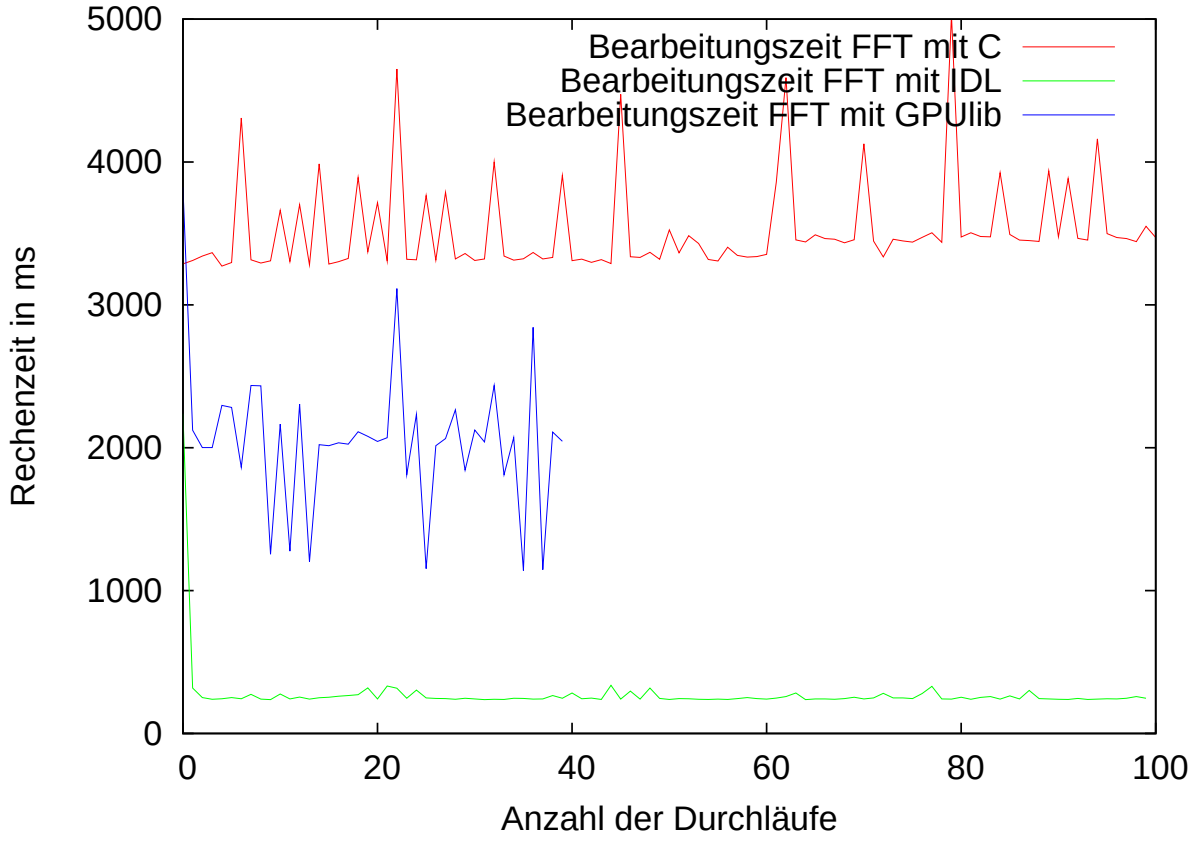


Abbildung 3.3: Vergleich der Rechenzeit für eine FFT in C, IDL und GPUlib

tierung gearbeitet.

Beide Shared Objects sind in etwa gleich schnell. Das heißt, dass der Umweg über ein zweites Shared Object auf die Rechenzeit keinen nennenswerten Einfluss nimmt. Dies erklärt auch, wie die kurze Rechenzeit nach dem ersten Aufruf in der IDL-Variante zu Stande kommt. Pro IDL-Session wird das DLM einmal geladen. Wird es im Anschluss wieder genutzt, liegt es bereits im Hauptspeicher, wodurch sehr kurze Zugriffszeiten die Folge sind. Auch wenn das IDL-Programm 100-mal beendet wurde, solange man die IDL-Session nicht beendet, bleibt das DLM weiterhin geladen.

Abbildung 3.5 zeigt den Zusammenhang zur GPU-Rechenzeit und zum Gesamtaufruf. Dem Diagramm kann die reine Rechenzeit der GPU entnommen werden, die Länge der Kopiervorgänge zwischen Device und Host und die Länge des gesamten Aufrufs. Die Differenz zwischen der Bearbeitungszeit gesamt (cyan gekennzeichnet) und der aufsummierten Bearbeitungszeit der GPU (magenta gekennzeichnet) gibt die Latenz des Shared Objects

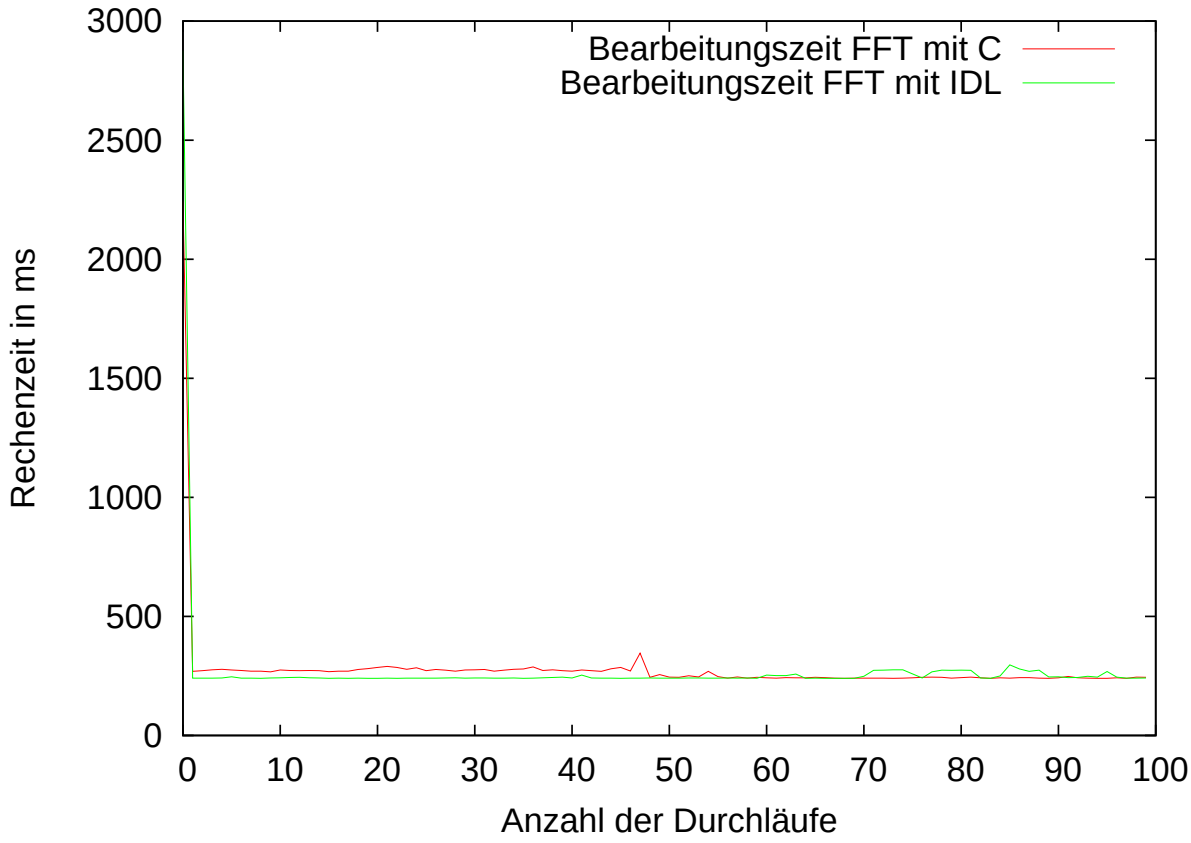


Abbildung 3.4: Vergleich der Rechenzeit zwischen IDL und C ohne Programmneustart

an. Zu Beginn beträgt sie mehrere Sekunden und stellt sich dann auf etwa $200ms$ ein.

Damit zeigt sich, dass ein Erarbeiten von zwei Implementierungen nicht sinnvoll ist. Zum einen auf Grund des aufwändigen Wartungsaufwandes und zum anderen, weil dies durch die GPUlib keinen Vorteil bringt. Zusammenfassend bedeutet das, dass beim Nutzen der dynamischen Bibliothek bei jedem Funktionsaufruf mit einer Latenz von etwa $200ms$ gerechnet werden muss. Dabei ist es unwesentlich, ob der Aufruf innerhalb IDL oder C geschieht.

3.2 Beschreibung der Tests zur Qualitäts- und Performanceanalyse

Zur Auswertung der GPU-Module, die für den Extended Chirp Scaling Algorithmus programmiert wurden, sind verschiedene Tests definiert worden. Hierbei wurde ein Datensatz

	Minimale Rechenzeit in ms	**Maximale Rechenzeit in ms**	**Durchschnittliche Rechenzeit in ms**
IDL	239,656	2876,280	273,440
C	240,000	2243,000	279,050

Tabelle 3: Kennzahlen der Durchläufe für eine FFT auf der Grafikkarte

von 8192×16384 Pixeln verwendet. Das macht eine Datenmenge von 1 GB. Hinzu kommen Daten wie Vektoren oder andere Arrays, die je nach Modul weiteren Speicherbedarf beanspruchen. Die Spezifikation der für die Tests benutzten Grafikkarte ist durch Tabelle 1 gegeben.

Die Tests für das Benchmarking können in zwei größere Kategorien eingeteilt werden. Zum einen die Kategorie, in der die Module einzeln getestet werden. Sie werden unabhängig von den anderen Modulen gestartet. Zu jedem Modul besteht somit ein auf CPU gerechneter Referenzdatensatz, mit dem die auf GPU berechneten Daten verglichen werden können. Diese Tests werden mit der Kennung F0 im Falle des F-SAR-C-Prozessors und S0 im Falle des IDL-STEP-Prozessors versehen. Danach folgt die Kennung für das entsprechende Modul, welche in Tabelle 4 angegeben ist. Beispielsweise ist der Testfall für das Modul der zweiten Bewegungskompensation im C-Prozessor F0-4.

Auswertungen zu diesen Tests erfolgen mit der Referenz, mit deren Hilfe entsprechende Plots und Bilder erzeugt werden können. Um Phase und Kohärenz zu berechnen, werden folgende Berechnungen durchgeführt:

$$\begin{aligned} \gamma_{\phi} &= \arctan\left(data \cdot ref^{-1}\right) \\ \gamma_{abs} &= \frac{\sum data \cdot ref^{-1}}{\sqrt{\sum(data \cdot ref^{-1}) \cdot \sum(ref \cdot ref^{-1})}} \end{aligned} \tag{2}$$

Wobei $data$ der errechnete und ref der Referenzdatensatz ist . Die Phase wird dabei auf die Werte $-\frac{\pi}{10}$ bis $\frac{\pi}{10}$ skaliert und die Kohärenz auf 0,999 bis 1.

Weiterhin existieren Plots, die die Phasendifferenz

$$\Delta\phi = \arctan\left(data \cdot ref^{-1}\right) \tag{3}$$

einer Spalte und einer Zeile des Datensatzes zeigen. Darüber hinaus wird die Differenz und das Verhältnis des Betrages einer Zeile und einer Spalte geplottet. Diese 6 Bilder

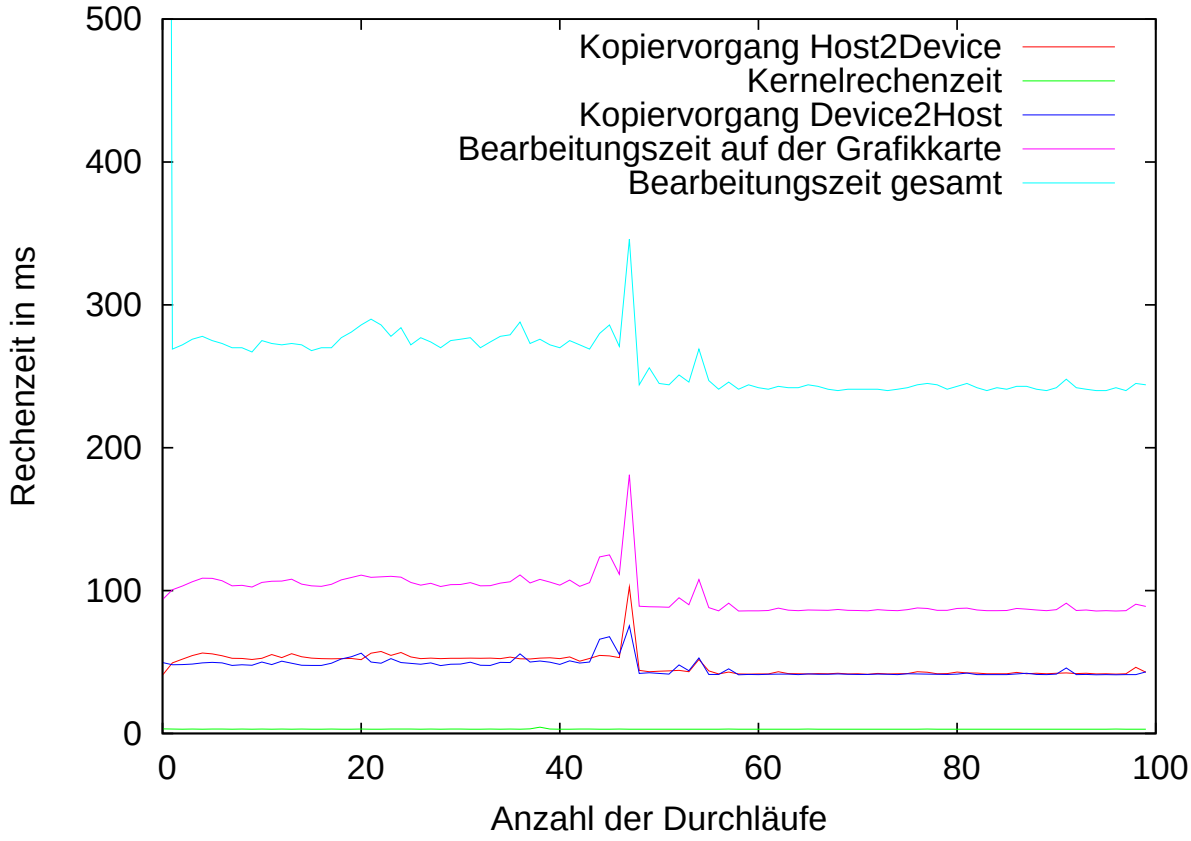


Abbildung 3.5: Vergleich der Gesamtrechenzeit zur Rechenzeit auf der GPU

ergeben die Auswertungsergebnisse zu jedem implementierten Modul, sowohl in C als auch in IDL.

Die zweite Testkategorie testet nicht die einzelnen Module, sondern das Endergebnis und besitzt die Kennung der Form F1 bzw. S1. Für die Kohärenzberechnung werden auch hier die in C und IDL berechnete Referenzbilder herangezogen.

Neben der Prüfung der Qualität erfolgt auch eine Zeitvergleich. Dieser existiert für jedes einzelne Modul. Der Timer startet unmittelbar vor Beginn der jeweiligen Funktion und endet unmittelbar danach. Das schließt Kopiervorgänge, Konvertierungen und den Kernelaufruf an sich für jedes Modul mit ein.

Ein Test besteht für beide Kategorien aus 100 Durchläufen. Auf Grund der leichten Schwankungen, die bei der Zeitmessung entstehen können, sollen etwaige Ausreißer ausgeglichen werden. In den jeweiligen Tabellen zur Performance werden dabei Minimal- und Maximalwert angegeben, sowie über die 100 Durchläufe der Durchschnitt gebildet. Für die

Modulname	Kennung
Chirp Scaling	-1
RC-RCMC	-2
PhaseCorrection	-3
2. Motion Compensation	-4
Azimuth Compression	-5

Tabelle 4: Übersicht der Kennungen für die Modultests

Qualitätsauswertung wird nur 1 Durchlauf benötigt, da alle Schritte der Radarverarbeitung deterministisch sind und dementsprechend bei gleichen Eingangsdaten keine Abweichungen entstehen.

3.3 Implementierung des Extended Chirp Scaling

Die Funktionsweise des Extended Chirp Scaling wurde bereites in Kapitel 2.1 kurz umrissen. In den nachfolgenden Kapiteln sollen die Grundlagen der einzelnen Schritte näher erläutert werden. Die dabei verwendeten Formeln benutzen Variablennamen entsprechend den Namen in der verwendeten Software. Da sich die in der Software benutzten Berechnungen nicht wesentlich von der Theorie unterscheiden, kann neben diesen Ausführungen die Quelle [MMS96] genutzt werden. Weiterhin soll auf die bereitgestellten Schnittstellen bezüglich dem C- und IDL-Code eingegangen werden und die Implementierung auf der Grafikkarte kurz erklärt werden. Abschließend folgt eine Auswertung zu der erreichten Performance und der Qualität der Ergebnisse im Vergleich zur Referenzimplementierung in C und IDL.

3.3.1 Chirp Scaling

Das Chirp Scaling ist der erste Schritt nach der Radardatenvorverarbeitung. Mit der Anwendung dieses Schrittes wird die Zielentfernungsänderung für den gesamten Entfernungsbereich angeglichen. Hierzu werden folgende Operationen durchgeführt. Zunächst werden die Radardaten in den Range-Doppler-Bereich überführt. Dies geschieht durch eine Fourier-Transformation über die Azimutdimension. Danach werden die Daten mit einer Phasenfunktion multipliziert. Um eine effektive Transformation in den Freuquenzbereich zu vollziehen, wird der Fast Fourier Transformation Algorithmus verwendet.

Die besagte Phasenfunktion ist eine komplexe Funktion mit quadratischer Phasenmodulation und kann wie folgt berechnet werden:

$$\begin{aligned} phase &= \exp(j \cdot aux1 \cdot aux2) \\ aux1 &= -\pi \cdot ecr \cdot azScaleCurvature \\ aux2 &= \left(\tau + \frac{-2 \cdot rfa}{c_0}\right)^2 \end{aligned} \tag{4}$$

Der quadratische Verlauf in Abhängigkeit von der Entfernung verschiebt das Phasenzentrum der Radardaten. In Bild 3.6 wird diese Verschiebung bildlich dargestellt. Weiterhin ist im Bild 3.7 die grafische Veranschaulichung von zwei Punktzielen der SAR-Daten in Konturdarstellung dargestellt. Innerhalb dieser Darstellung wird klar, wie sich das Chirp Scaling auf die Position der einzelnen Punktziele in Abhängigkeit zur Entfernung auswirkt.

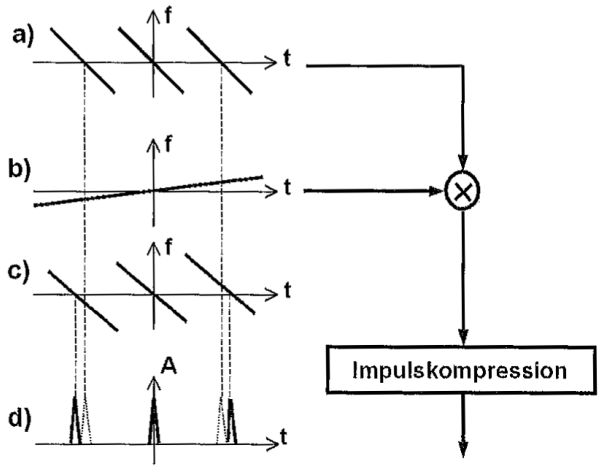


Abbildung 3.6: Veranschaulichung des Chirp Scaling Prinzips (Quelle [KH00], Seite 249)

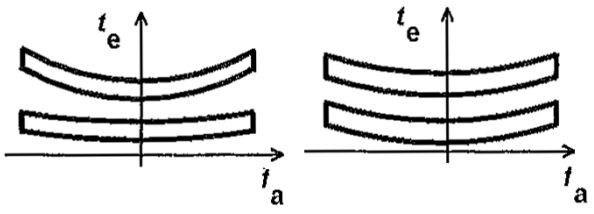


Abbildung 3.7: Auswirkung des Chirp Scalings auf SAR-Daten, Konturdarstellung zweier Punktziele (links vor dem Chirp Scaling, rechts danach)(Quelle [KH00], angepasst)

Variablenname	**Typ**	**Größe**	**Beschreibung**
data	complex	entfernung×azimut	SAR-Datenmatrix
τ	double	entfernung	Signallaufzeit
ecr	double	azimut	effektive Chirp Rate
azScaleCurvature	double	azimut	Chirp Scaling Faktor (skaliert)
rfa	double	azimut	Entfernungsvariation im Azimutfrequenzbereich

Tabelle 5: Schnittstellen für den Chirp Scaling-Algorithmus

Eine genaue Auseinandersetung mit der Thematik des Chirp Scaling findet sich in folgender Literatur: [MMS96]

Schnittstellen zu C und IDL

Tabelle 5 zeigt die erforderlichen Eingabeparameter für den Algorithmus. Da sich beide Implementierung nicht wesentlich unterscheiden, gilt die Tabelle sowohl für IDL als auch für C. Für diesen Schritt müssen keine Anpassungen am Shared Object Wrapper für die IDL-Anbindung vorgenommen werden.

Umsetzung auf GPU

Da keine Anpassungen nötig sind, wird beim Shared Object lediglich eine Überprüfungen der Variablen vorgenommen. Dazu gehört eine Überprüfung auf Typ und Größe, sowie Anzahl der Dimensionen und eine Mindestgröße. Da auch die Variablen bereits standardmäßig im richtigen Datentyp übergeben werden, ist keine Konvertierung der Variablen nötig.

Im ersten Schritt des Algorithmus' auf der GPU, müssen die Daten in den Range-Doppler-Bereich überführt werden. Eine eindimensionale Fouriertransformation über die Azimut-Dimension wird durchgeführt. Hierzu müssen die Daten wegen ihrer Lage im Speicher transponiert werden. Für diesen Schritt ist bereits ein Kernel für die Grafikkarte implementiert, der von NVidia frei zur Verfügung steht. Die Implementierung und den Aspekt der Speicherverwaltung kann man in Quelle [RM09] nachschlagen.

Anschließend folgt die Durchführung des Chirp Scalings mit folgenden Einstellungen auf der Grafikkarte. Die einzige Variable τ, die die Größe der Range-Dimension hat, wird als Textur auf die Grafikkarte kopiert. Da auf τ nur lesend zugegriffen wird, eignet sich eine Textur hervorragend für diese Aufgabe. Sie besitzt einen Cache, der Zugriffe wesentlich beschleunigt. Auch können keine Zugriffskonflikte während des Lesens auftreten, welche die Performance der Rechenoperation verschlechtern würden. Eine sehr genaue Auseinandersetzung mit Texturen in CUDA bietet [Far09].

Der Kernel für die Berechnung auf der Grafikkarte wird als eindimensionales Grid angelegt. Die Anzahl der Blöcke eines Grids entspricht der Azimut-Dimension. Jeder Block besitzt 512 Threads. Dadurch wird pro Block eine vollständige Multiplikation des Phasenvektors mit einer Zeile des Daten-Arrays durchgeführt. Gibt es mehr Elemente in einer Zeile als Threads, werden weitere elementweise Berechnungen serialisiert. Hat eine Zeile 2048 Elemente führt jeder Thread eines Blockes 4 elementweise Operationen nacheinander durch. Dies geschieht mit der in Listing 2 gegebenen For-Schleife.

```
1 for(unsigned int i= threadIdx.x; i<matrixWidth;i+=blockDim.x) {
2    //Berechnungen
3    ...
4    //Multiplikation der Daten mit der Phasenfunktion
5    data[blockIdx.x*matrixWidth + i] =
6          ComplexMul( data[blockIdx.x*matrixWidth +i],
7                   phase );
8 }
```

Quelltext 2: For-Schleife für elementweise Berechnungen

Hierbei wird sichergestellt, dass der Zugriff auf das Daten-Array optimal erfolgt. Dies ist wichtig, da es sich im globalen Speicher der Grafikkarte befindet. Bei ungünstigen Zugriffen auf diesen, kann die Rechenkapazität der Grafikkarte nicht voll ausgeschöpft werden. Konkurrierende Lese- und Schreibvorgänge würden serialisiert werden. Daher wird berücksichtigt, dass eine theoretische, parallele Ausführung von 32 Threads pro Prozessor erfolgt, dem sogenannten Warp. Dies und die 512 Threads stellen sicher, dass ein Warp nebeneinander liegende Elemente des Arrays liest und schreibt. Dadurch werden Bankkonflikte und ein Hin- und Herspringen des Pointers im Speicher vermieden.

Neben der Textur und dem ausgerichteten Zugriff auf das Daten-Array wird weiterhin der Zugriff auf den Speicher durch den Shared Memory begünstigt. Dies kann durch die

beschriebene Kernelspezifikation erreicht werden. Der Parameter $aux1$ zur Berechnung der Phasenfunktion braucht pro Block nur einmal berechnet werden. Weiterhin ist die Eingangskomponente rfa pro Block für jedes Element gleich. Aus diesem Grund werden diese beiden Variablen wie in Listing 3 berechnet und in den Shared Memory gespeichert.

```
1 __shared__ double aux;
2 __shared__ double srfa;
3
4 if(threadIdx.x == 0) {
5    aux = -1.0*PI*ecr[blockIdx.x]*sclcurve[blockIdx.x];
6    srfa = -2.0*rfa[blockIdx.x]/C0;
7 }
8
9 __syncthreads();
```

Quelltext 3: Belegen des Shared Memorys für den Chirp Scaling Algorithmus

Nach jedem Schreiben in den Shared Memory muss stets sichergestellt werden, dass die Threads synchronisiert werden. Ansonsten führt dies zu unvorhersehbaren Werten beim nächsten Lesezugriff.

Innerhalb der for-Schleife erfolgt nun die elementweise Berechnung der Phase und dem anschließenden Multiplizieren mit den Daten. Bei dieser Gelegenheit werden die Daten zudem normalisiert. Dies ist nötig, da die Fast Fourier-Transformation für CUDA keine Normalisierung enthält.

Komplexe Operationen wie die Exponentialfunktion, Multiplikation und Skalierung sind nicht in CUDA enthalten und müssen deswegen selbst implementiert werden. Diese können dem Anhang entnommen werden.

Auswertung der C-Implementierung

In diesem Teil erfolgt die Auswertung des Tests F0-1. Anhand dieser Tests ist es möglich, Fehler in der Implementierung herauszufinden und die ersten Benchmarks anzuführen. Die Tests zeigen jedoch noch nicht auf, ob sich das einzelne Modul bereits fehlerfrei im Gesamtcode verhält. Zu diesem Zwecke folgt ein Gesamttest.

Im Bild 3.8 ist die Kohärenz zwischen dem Referenzdatensatz und dem Grafikkartendatensatz zu sehen. Der besseren Sichtbarkeit halber wurde das Bild mit einem schwarzen Rahmen versehen. Weiße Flächen stellen eine Kohärenz von 1 dar und schwarze eine von

0,99. Im Bild 3.8 wurde die Kohärenz der Beträge beider Bilder verglichen. Deutlich erkennbar ist die sehr gute Übereinstimmung beider Datensätze.

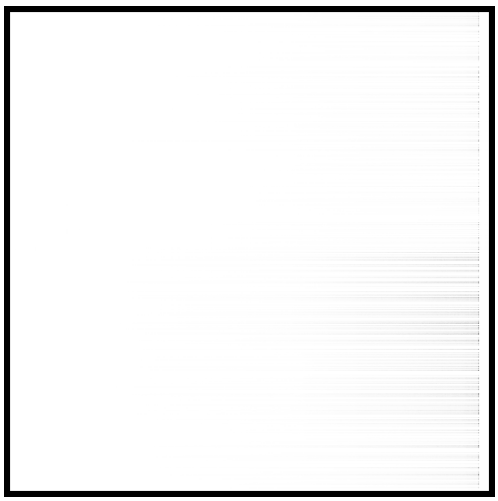


Abbildung 3.8: Kohärenz nach dem Chrip Scaling, skaliert auf weiß = 1, schwarz = 0,999

Bild 3.9 zeigt die Übereinstimmung der Phase. Da bei der SAR-Prozessierung geringe Abweichungen von der Phase großen Einfluss auf die Endqualität haben, ist es hier besonders wichtig, gute Ergebnisse zu erzielen.

Wie Abbildung 3.9 zeigt, stimmt auch die Phase sehr gut mit dem Referenzbild überein. Hierbei wurde das Bild auf Winkel von $-\frac{\pi}{10}$ bis $\frac{\pi}{10}$ rad skaliert. Eine graue Fläche stellt demnach geringe Phasenabweichungen dar.

Diese Bilder zeigen, dass nur sehr geringe Abweichungen zu Stande kommen, die auf Ungenauigkeiten einiger auf der Grafikkarte implementierten Funktionen zurückführbar sind (beispielsweise Kosinus oder Sinus). Die vollständigen Daten zur Auswertung können dem Anhang entnommen werden.

Tabelle 6 zeigt die Rechenzeiten für das Chirp Scaling.

Trotz der Kopiervorgänge zwischen Host und Device, ist die Grafikkartenimplementierung um einen Faktor von durchschnittlich 52 schneller als die CPU-Variante des F-SAR-C-Prozessors.

Auswertung der IDL-Implementierung

Die Auswertungen zur IDL-Implementierung erfolgt durch den Test I0-1. Wie die Bilder 3.10b und 3.10a zeigen, konnte der Algorithmus fehlerfrei für die Grafikkarte umgesetzt

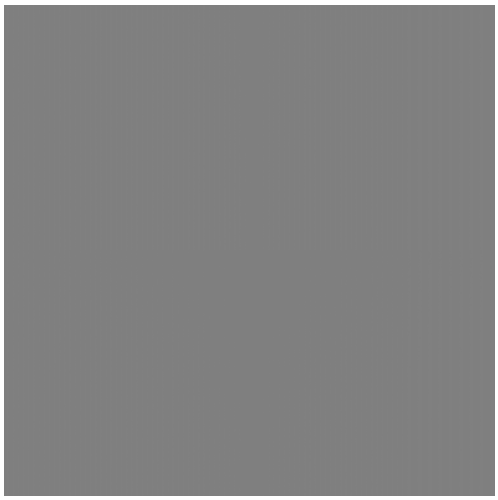


Abbildung 3.9: Phasedifferenz nach dem Chirp Scaling zum Referenzbild, skaliert auf weiß $= -\frac{\pi}{10}$, schwarz $= \frac{\pi}{10}$

	min [ms]	max [ms]	mean [ms]
CPU	30120	32824	30371.2
GPU	570	751	583.27
Faktor	52,84	43,71	52,07

Tabelle 6: Rechenzeiten in ms für das Chirp Scaling auf der Grafikkarte und auf der CPU

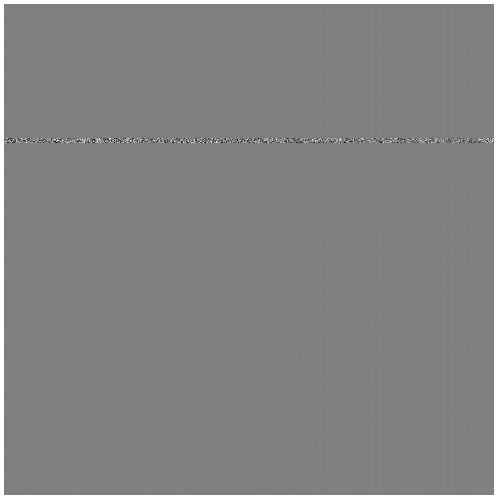
werden. Jedoch fällt sofort ein Streifen ins Auge, der auf Unstimmigkeiten schließen lässt.

Dieser Streifen, in dem die Daten nicht übereinstimmen, wird durch Nullen verursacht. Da der Datensatz an dieser Stelle mit Nullen belegt ist, kommt es somit an dieser Stelle zu einer fehlerhaften Berechnung der Kohärenz. Wird der Nenner in der Kohärenzberechnung Null, wird das Ergebnis an dieser Stelle auf `NaN` gesetzt, in Bild 3.10b erscheint dies schwarz.

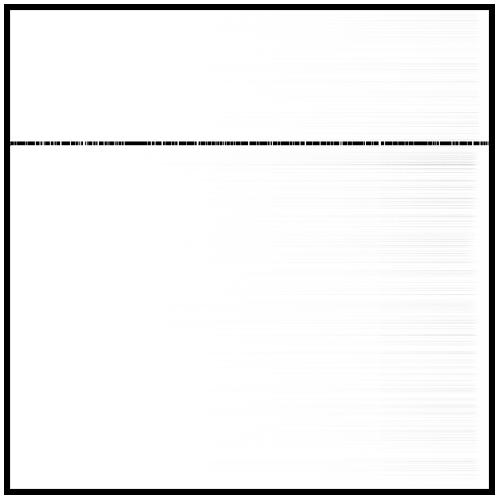
Wie Tabelle 7 zeigt, ist die Implementierung für den IDL-Prozessor wesentlich schneller als die für den C-Prozessor, wohingegen keine größeren Unterschiede in der Performance bei der Grafikkartenvariante liegen. Aus diesem Grund konnte lediglich ein Beschleunigungsfaktor von 13 erzielt werden.

3.3.2 Zielentfernungskorrektur und Entfernungskompression

Nachdem das Chirp Scaling die Krümmung der Zielentfernungsänderung angeglichen hat, folgt, bildlich beschrieben, ein „Geradebiegen“ und „Komprimieren“ der Pulse. Abbildung



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz(weiß = 1, schwarz = 0,999)

Abbildung 3.10: Phasendifferenz und Kohärenz nach Chirp Scaling in der IDL-Implementierung

	min [ms]	max [ms]	mean [ms]
CPU	7521,73	8900,69	7608,23
GPU	569	809	585,3
Faktor	13,22	11,00	13,00

Tabelle 7: Rechenzeiten in ms für das Chirp Scaling auf der Grafikkarte und auf der CPU in der IDL-Implementierung

3.11 zeigt die grafisch.

Die Entfernungskompression und Zielentfernungskorrektur wird im Entfernungsfrequenzbereich durchgeführt. Das heißt, dass eine weitere Fouriertransformation durchgeführt wird, diesmal über die Entfernungsdimension. Damit befinden sich die SAR-Daten nun im zweidimensionalen Frequenzraum, dem sogenannten Wavenumber-Raum.

Um die SAR-Daten in Entfernungsrichtung komprimieren zu können, werden sie mit einer quadratischen Phase multipliziert. Dies wird mit Hilfe des Optimalfilters[1] durchgeführt, der sich wie ein entgegengesetzter Entfernungschirp folgendermaßen errechnet (siehe dazu

[1]Unter Kenntnis der Struktur des gesendeten Signals, kann das Signal-Rausch-Verhältnis des empfangenen Signals optimiert werden.

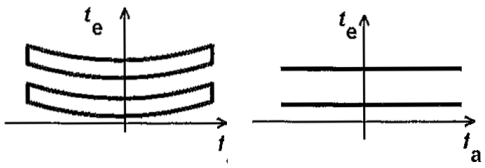


Abbildung 3.11: Auswirkung der Zielentfernungskorrektur und Entfernungskomprimierung auf SAR-Daten, Konturdarstellung zweier Punktziele (links vor, rechts nach der Anwendung des Schrittes)(Quelle [KH00], angepasst)

Kapitel 2.1):

$$\exp\left(-j\pi k_r t_r^2\right) \circ\!\!-\!\!\bullet \exp\left(j\pi \frac{f_r^2}{k_r}\right) \tag{5}$$

Hierbei stellt $\circ\!\!-\!\!\bullet$ die Fouriertransformation und k_r die Entfernungsmodulationsrate, mit dem das Signal in der Entfernung moduliert wird, dar. Auf Grundlage der Theorie des Optimalfilters, siehe hierzu [KH00] Kapitel 8.2.3, kann nun die Entfernungskompression durchgeführt werden.

Weiterhin wird in diesem Schritt eine Zielentfernungskorrektur vorgenommen. Zur Veranschaulichung der Problematik dient Abbildung 3.12. In Schwarz ist dabei der anfängliche Signalverlauf zu sehen, der durch die Geometrie des Signalempfangs zu Stande kommt (siehe Kapitel 2.1).

Diese schwarze Kurve, sowie ihre Fouriertransformierte können mit folgenden Formeln berechnet werden:

$$\begin{aligned} r(t_a) &\circ\!\!-\!\!\bullet r(f_a) \\ \sqrt{R_{rref}^2 + v_p^2 \cdot t_a^2} &\circ\!\!-\!\!\bullet R_{ref} \cdot \frac{1}{\sqrt{1 - \left(\frac{\lambda f_a}{2V_p}\right)^2}} = R(f_a) \end{aligned} \tag{6}$$

Zur Vereinfachung wird die Kurve nur mit einer Referenzentfernung R_{ref} korrigiert. Weiterhin wird die Geschwindigkeit v_p der Sendeplattform benötigt.

Um nun die Kurve, in Rot dargestellt, zu „begradigen“, muss für jeden Punkt der Kurve ein $\Delta\tau$ berechnet und der Funktionswert um diesen verschoben werden. $\Delta\tau$ wird wie folgt berechnet:

$$\Delta\tau = -R(f_a) \cdot \frac{2}{c_0} \tag{7}$$

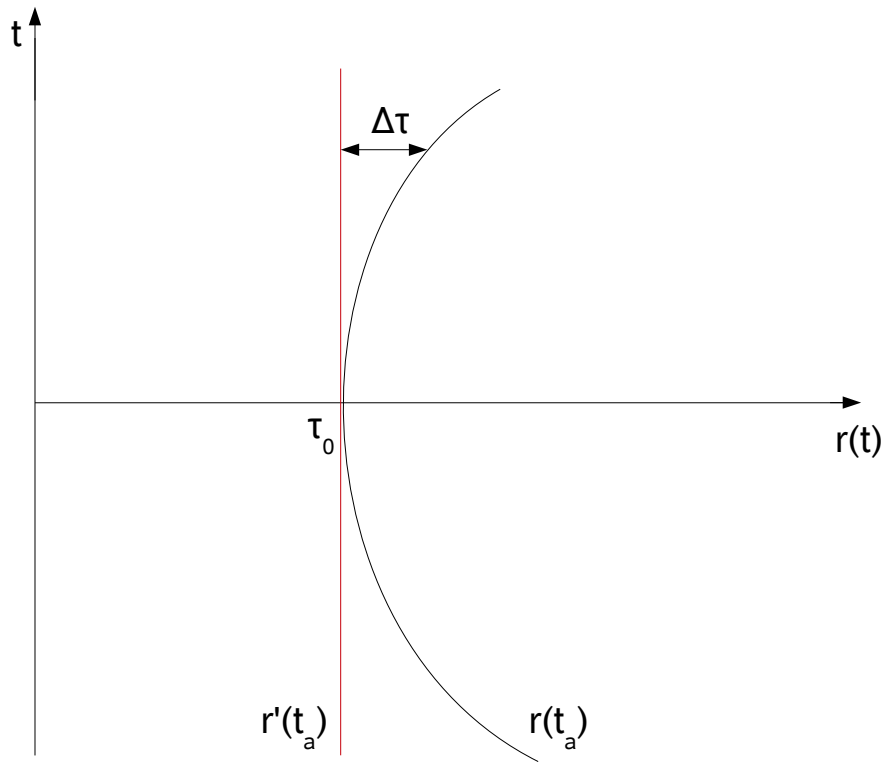


Abbildung 3.12: grafische Veranschaulichung der Zielentfernungskorrektur

Da die Daten bereits im Frequenzbereich vorliegen, kann die Verschiebung nach dem Fouriertheorem wie folgt verschoben werden:

$$S(\tau - \Delta\tau) \circ\!\!-\!\!\bullet S(R(f_a)) \cdot \exp(-j2\pi \cdot \Delta\tau \cdot f_r) \tag{8}$$

Abschließend wird neben der Rücktransformation in den Range-Doppler-Bereich zuvor eine Gewichtung der Daten, für gewöhnliche eine Hamming-Gewichtung, über die Entfernungsdimension durchgeführt. Diese dient dazu, die Nebenkeulen der $\frac{\sin x}{x}$ Impulsantwort zu unterdrücken.

Schnittstellen

Um kompatibel für beide Implementierungen zu sein, musste die Definition dieser Schnittstelle angepasst werden. Zwar übernimmt der Wrapper für das IDL-Shared Object neben

Variablenname	**Typ**	**Größe**	**Anmerkung und Beschreibung**
data	complex	entfernung×azimut	SAR-Datenmatrix
ecr	double	azimut	effektive Chirp Rate
azCurvature	double	azimut	Chirp Scaling Faktor
azScaleCurvature	double	azimut	Chirp Scaling Faktor (skaliert)
window	float	entfernung	spektrale Gewichtung
sampleFrequency	double	skalar	Abtastfrequenz
referenceRange	float	skalar	Referenzentfernung
offset	unsigned int	skalar	Offset,nicht in IDL vorhanden, deswegen mit dem Wert 0 übergeben
rgSize	unsigned int	skalar	neue Größe der Entfernungsdimension, nicht in IDL vorhanden, deswegen mit range übergeben

Tabelle 8: Schnittstellen für die Zielentfernungskorrektur und Entfernungskompression

der Aufgabe der Parameterüberprüfung keine weiteren Aufgaben, dennoch gibt es bei der Paremeterübergabe eine kleine Besonderheit.

Die Schnittstelle der Bibliothek bietet neben den benötigten Parametern noch zwei weitere. Diese werden jedoch nur von der C-Implementierung genutzt. Das Shared Object für die IDL-Implementierung stellt diese nicht bereit. Stattdessen werden im Wrapper die Variablen mit entsprechenden Standardwerten belegt, die die weitere Verarbeitung nicht beeinflussen. Tabelle 8 zeigt die Definition der Schnittstelle.

`offset` und `rgSize` sorgen im C-Code dafür, dass eine Zeile über die Entfernungsdimension auf `rgSize` vergrößert wird. Die eigentlichen Daten beginnen dabei ab dem Index `offset`. Das Fenster, also die Gewichtungsfunktion, wird über die ganze Breite `rgSize` multipliziert.

Zur Berechnung der Entfernungskompression werden `effectiveChirpRate` und `azScaleCurvature` benötigt. Wie bereits zuvor dargestellt, wird ein Chirp-Signal verwendet und mit den SAR-Daten entsprechend dem Faltungstheorem als Multiplikation im Frquenzbereich gefaltet. Für dieses Signal wird die Entfernungsmodulationsrate benötigt,

die sich wie folgt berechnen lässt:

$$k_r = ecr \cdot (1 + azScaleCurvature) \tag{9}$$

Ausgedrückt als `aux1` wird der Faktor für die Entfernungskompression wie folgt berechnet:

$$aux1 = \frac{-\pi}{ecr \cdot (1 + azScaleCurvature)} \cdot f_r^2 \tag{10}$$

`azCurvature` aus dem Code stellt die Funktion $r(f_a)$ dar, wobei R_{ref} seperat in den Faktor für die Zielentfernungskorrektur mit einberechnet wird. Setzt man alle Formeln und den Verschiebungsfaktor -2π für `aux2` entsprechend ein, gelangt man zu folgender Gleichung:

$$aux2 = \frac{4\pi \cdot rref \cdot azCurvature}{c_0} \tag{11}$$

Die Berechnung der Gewichtungsfunktion erfolgt nicht auf der GPU, sondern wird als Parameter an die Bibliothek übergeben. Dies geschieht, da sich je nach Einstellungen des Prozessors, die Art der Gewichtungsfunktion ändern kann. Zudem gibt es zu viele Parameter, die aus den Parameterstrukturen gelesen werden müssen. Da dies nicht sehr gut auf die Grafikkarte umsetzbar ist und zudem kaum Geschwindigkeitsvorteile bringen würde, wurde dies als Input für die Bibliothek definiert.

Umsetzung auf GPU

Auch die Zielentfernungskorrektur und die Entfernungskompression sind nach dem gleichen Schema aufgebaut wie das Chirp Scaling. Aus diesem Grund ist der Kernel wie der Kernel des vorigen Schrittes konfiguriert.

`aux1`, der Faktor für die Entfernungskompression, und `aux2`, der Faktor für die Zielentfernungskorrektur, werden zuvor von einem Thread im Block berechnet und in den Shared Memory gespeichert. Danach werden die Threads synchronisiert und in einer for-Schleife jedes einzelne Element der Daten berechnet.

Bei diesen Berechnungen werden die beiden Teilschritte als Addition zusammengefasst, denn es gilt:

$$\exp(a) \cdot \exp(b) = \exp(a + b) \tag{12}$$

Anschließend wird die eben errechnete Phase mit dem entsprechenden Element der Gewichtungsfunktion multipliziert. Da auf die Daten der Gewichtungsfunktion lediglich lesend zugegriffen wird, können diese in eine Textur kopiert werden und bieten somit höchste Performance für diese Anforderungen. Auch in diesem Schritt wird eine Fouriertransformation durchgeführt. Deshalb müssen die Daten neben der Phase auch mit dem Reziproken der Anzahl ihrer Elemente in Entfernungsrichtung normalisiert werden.

Da pro Zeile ein Offset von `rgOffset` besteht, werden die Daten nicht wie gewohnt im linearen Speicher abgelegt, sondern mit `cudaMallocPitch` allokiert. Das legt die zweidimensionalen Daten für die jeweilige Hardwarearchitektur optimiert in den Speicher. Ein Pitch wird errechnet, der für weitere Zugriffe auf die Daten benötigt wird. Dies bedeutet zwar einen gewissen Verwaltungsaufwand für die Daten, doch vereinfacht ein Allokieren und Kopieren der Daten mit einem Offset.

Das Arbeiten mit diesem Pitch erschwert jedoch die eindimensionale Fouriertransformation. Anstatt wie bisher einen Batch von *Höhe* anzugeben, wird nur eine einzige Fouriertransformation für den Plan[2] angegeben. In einer for-Schleife wird der Pointer für die Daten um Pitch-Speicherplätze verschoben, die Fouriertransformation erstreckt sich jedoch nur über alle *Range* - Elemente. Dies hat den Nachteil, dass die vielen Fouriertransformationen serialisiert sind und nicht parallel ausgeführt werden. Abhilfe könnte hier die Funktion `planMany` schaffen, die für jede eindimensionale Transformation ein Offset annehmen kann und in einem Batch ausführt. Diese Funktionalität steht zwar bereits zur Verfügung, ist jedoch in der CUDA Version 3.2 noch nicht implementiert.

Auswertung der C-Implementierung

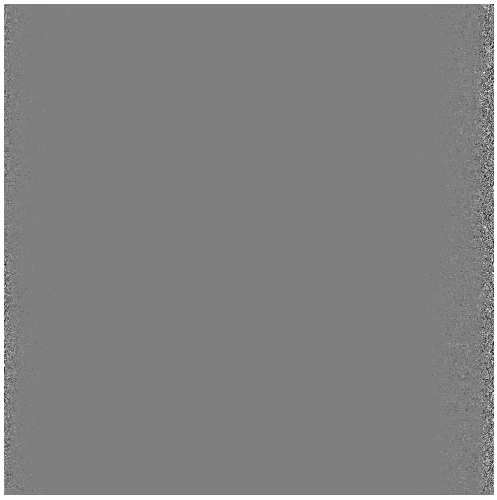
Zur Auswertung für dieses Modul soll der Test F0-2 herangezogen werden. Wie bereits in Kapitel 3.3.1 dienen die beiden Bilder 3.13a und 3.13b zur Kontrolle der der Datenqualität.

Im Gegensatz zum Chirp Scaling ist die Kohärenz nicht überall nahezu 1. Vor allem an den Rändern kommt es zu größeren Abweichungen. Da diese Bereiche durch die Software im Nachhinein auf ungültig gesetzt werden, fallen diese Abweichungen nicht ins Gewicht.

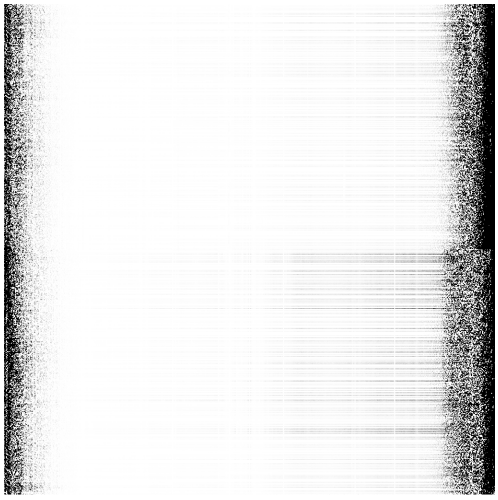
Tabelle 9 zeigt die Rechenzeit und den Faktor der Beschleunigung. Auch hier können große Rechenzeiteinsparungen verzeichnet werden.

Wie deutlich am Faktor zu erkennen ist, konnte in diesem Modul nicht an die Perfor-

[2]Damit die Fouriertransformation so performant wie möglich ausgeführt werden kann, wird vor deren Ausführung die Höhe und Breite der Dimensionen angegeben. Ein Wiederverwenden des Plans bei gleichen Dimensionen der Transformation ist möglich und wird empfohlen.



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.13: Phasedifferenz und Kohärenz nach der Zielentfernungskorrektur und Entfernungskompression

	min [ms]	max [ms]	mean [ms]
CPU	41620	44809	42064,1
GPU	1048	1941	1089,06
Faktor	39,71	23,09	38,62

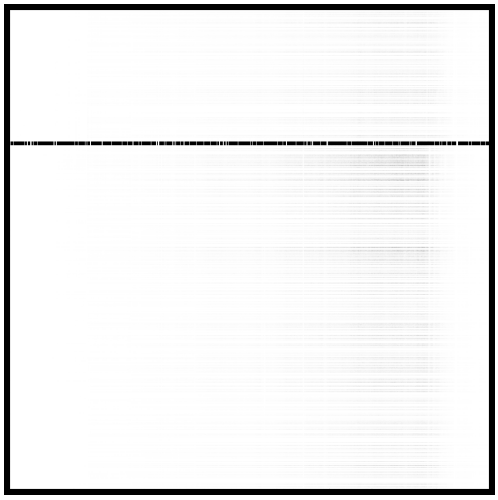
Tabelle 9: Rechenzeiten in ms für das Modul der Zielentfernungskorrektur und Entfernungskompression auf CPU und GPU

mance der vorangegangenen Schritte angeknüpft werden. Dies kommt durch das höhere Datenaufkommen, welches durch `rgSize` definiert wird, und die serielle Ausführung der FFT. In diesem Fall ist `rgSize` doppelt so groß wie die eigentlichen Daten in Entfernungsrichtung.

Auswertung der IDL-Implementierung

Die Auswertung für die Zielentfernungskorrektur und Entfernungskompression erfolgt durch den Test I0-2. Neben den Streifen in der Kohärenz gibt es keine größeren Abweichungen zum Referenzbild. Wie bereits in Kapitel 3.3.1 geklärt wurde, entsteht der Streifen durch Nullen, die bereits vor der Verarbeitung im Datensatz vorhanden sind.

(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.14: Phasedifferenz und Kohärenz nach der Zielentfernungskorrektur und Entfernungskompression in IDL

	min [ms]	max [ms]	mean [ms]
CPU	18318,11	21068,32	18568,27
GPU	971	1023	987,38
Faktor	18,87	20,59	18,81

Tabelle 10: Rechenzeiten in ms für das Modul der Zielentfernungskorrektur und Entfernungskompression auf CPU und GPU in IDL

Ein Blick auf die Tabelle zeigt, dass durch den IDL-Wrapper keine weiteren Latenzen entstehen und somit die Rechenzeiten für den IDL- und C-Prozessor annähernd gleich sind. Wie bereits beim Chirp Scaling festgestellt wurde, ist die IDL-Implementierung des Algorithmus' wesentlich schneller als die des C-Prozessors. Obwohl in der C-Implementierung die Entfernungsdimension um den Faktor 2 größer ist (durch `rgOffset` und `rgSize`), entstehen auf der GPU lediglich Zeitdifferenzen von annähernd 100ms. Dies zeigt deutlich, dass eine Verdopplung der Entfernungsdimension für die Rechenzeit auf der Grafikkarte wenig Auswirkung hat. Dies lässt den Schluss zu, dass bei der Ausführung des Kernels mit kleinerer Entfernungsdimension die Ressourcen der Grafikkarte nicht optimal ausgeschöpft werden.

Variablenname	**Typ**	**Größe**	**Beschreibung**
data	complex	entfernung×azimut	SAR-Datenmatrix
τ	double	entfernung	Signallaufzeit
ecr	double	azimut	effektive Chirp Rate
azCurvature	double	azimut	Chirp Scaling Faktor
azScaleCurvature	double	azimut	Chirp Scaling Faktor (skaliert)
window	float	entfernung	spektrale Gewichtung
α	float	skalar	Skalierungsfaktor
referenceRange	float	skalar	Referenzentfernung

Tabelle 11: Schnittstellen für die Phasenkorrektur

3.3.3 Phasenkorrektur

Dieser Schritt entspricht einer algorithmenbedingten Phasenkorrektur und wird näher in [MMS96] erläutert.

Da hierbei die Daten im Range-Doppler-Bereich vorliegen, erfolgt die Korrektur wieder durch eine Multiplikation mit einer komplexen Phasenfunktion. Die Phase berechnet sich wie in den vorherigen Schritten aus zwei Faktoren, `aux1` und `aux2`:

$$\begin{aligned} aux1 = & \\ & 4\pi \cdot ecr \cdot azScaleCurvature \cdot \frac{(1 + azCurvature)^2}{c_0^2\,(1 + azScaleCurvature)} \\ aux2 = & \left(\frac{\tau \cdot \frac{c_0}{2} - R_{ref}}{\alpha} \right)^2 \end{aligned} \tag{13}$$

Schnittstellen

Da die beiden Implementierungen zur Phasenkorrektur die gleichen Berechnungen anstellen, benötigt die Schnittstelle lediglich die zuvor erwähnten Werte. Das Shared Object in IDL übernimmt ausschließlich Typumwandlungsaufgaben. Die Funktion für die Phasenkorrektur wird aus diesem Grund in IDL wie im C-Prozessor gleich aufgerufen.

Die hierfür definierten Parameter können der Tabelle 11 entnommen werden.

	min [ms]	max [ms]	mean [ms]
CPU	20205	22329	20407,8
GPU	573	631	586,29
Faktor	35,26	35,39	34,81

Tabelle 12: Rechenzeiten in ms für das Modul der Phasenkorrektur auf CPU und GPU

Implementierung auf der GPU

Zum Starten des Kernels wird das Grid eindimensional festgelegt. Die Anzahl der Blöcke entspricht der Azimut-Dimension, die Anzahl der Threads wird auf 512 festgelegt. Durch diese Konfiguration übernimmt ein Block die Berechnung der Multiplikation der Phase mit einer Zeile der SAR-Daten. Jeder Thread berechnet dabei mindestens ein Element des Phasenvektors.

Um die Zugriffe auf den Global Memory so gering wie möglich zu halten, wird der erste Skalierungsfaktor `aux1` einmal pro Block berechnet und in den Shared Memory gespeichert. Auch die Elemente für $azCurvature$ und $azScaleCurvature$ werden in den Shared Memory gelegt, da innerhalb eines Blockes auf ein und dasselbe Elemente der jeweiligen Variable zugegriffen wird. Alle Elemente von τ hingegen werden pro Block einmal gelesen. Aus diesem Grund wird τ, wie zuvor auch, als Textur in den Speicher kopiert.

Neben den Vektoren existieren α und R_{ref} als konstante Skalare. Um auch hier den Zugriff auf diese zu optimieren, werden sie in den Constant Memory kopiert.

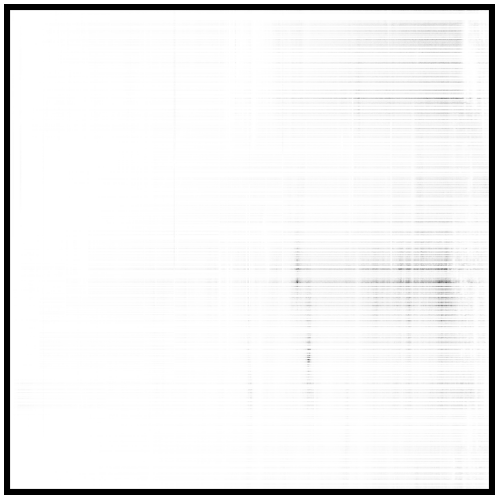
Obwohl in der Theorie keine inverse Fourier-Transformation über die Azimut-Dimension durchgeführt wird, schließt die Implementierung auf der Grafikkarte mit Hin- und Rücktransponierung und FFT ab. Somit entfällt die Fouriertransformation für den nächsten Schritt.

Auswertung der C-Implementierung

Die Kohärenzbilder zeigen deutlich, dass die Ausführung des Moduls auf der Grafikkarte nur sehr wenig Auswirkung auf die Qualität der Daten hat, sowohl im Betrag als auch in der Phase. Der besseren Sichtbarkeit wegen wurde wieder um Bild 3.15b ein Rahmen gelegt.

Wie bereits in den anderen Kapiteln, sind die zum Chirp Scaling Schritt vergleichbaren Ergebnisse des Tests F0-2 in Tabelle 12 zusammengefasst.

(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.15: Phasendifferenz und Kohärenz nach der Phasenkorrektur

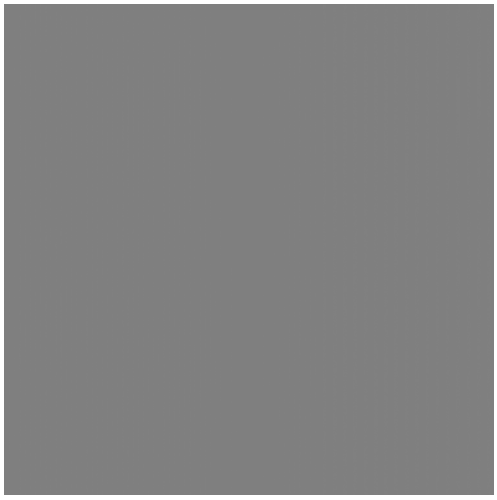
	min [ms]	max [ms]	mean [ms]
CPU	10370,23	11280,35	10513,08
GPU	584	781	600,74
Faktor	17,76	14,44	17,50

Tabelle 13: Rechenzeiten in ms für das Modul der Phasenkorrektur auf CPU und GPU in IDL

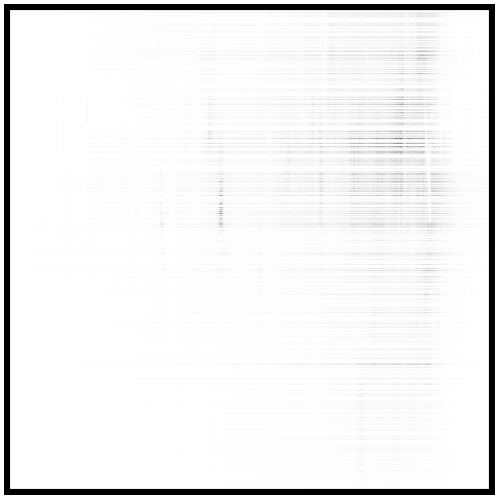
Auswertung der IDL-Implementierung

Die Ergebnisse des Tests I0-3 werden durch die Bilder 3.16a und 3.16b veranschaulicht. Sie zeigen deutlich, dass die Implementierung auch für den IDL-Prozessor fehlerfrei funktioniert.

Wie vorher auch ist die IDL-Implementierung des IDL-Prozessors effizienter als die des FSAR-C-Prozessors. Da im Wrapper neben der Überprüfung der Variablen nichts berechnet werden muss, kommt es bei der GPU-Variante der beiden Implementierungen nur zu geringen Abweichungen in der Rechenzeit. Grund für die im Durchschnitt 10ms Abweichung können Ausnutzung der Grafikkarte durch andere Nutzer oder Programme sein, da keine Unterschiede im GPU-Ablauf beider Varianten vorhanden sind.



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.16: Phasendifferenz und Kohärenz nach der Phasenkorrektur in IDL

3.3.4 Bewegungskompensation zweiter Ordnung

Beim F-SAR befindet sich das Radarsystem auf einer beweglichen Plattform. Im Gegensatz zu feststehenden Radarsystemen oder Satelliten, kann die Flugbahn des Flugzeuges nicht genau genug angepasst werden. Bei dieser Art der SAR-Prozessierung wirken sich bereits kleine Phasenfehler erheblich auf das Endergebnis aus. Daher müssen die Abweichungen von der Flugbahn aufgezeichnet und in die Berechnung einbezogen werden. Dies geschieht über zwei Schritte.

Zunächst erfolgt in der Radardatenvorverarbeitung eine Bewegungskompensation erster Ordnung. Hierbei wird der Fehler für eine Referenzentfernung herausgerechnet. In diesem Schritt, der Bewegungskompensation zweiter Ordnung, wird nun auch der Restfehler kompensiert. Abbildung 3.17 zeigt die Geometrie dieser Thematik.

Die gestrichelte Linie im Bid stellt die nominelle Flugbahn dar. Mit A und A' gekennzeichnet ist die nominelle und tatsächliche Position des Phasenzentrums der Antenne (Quelle [KH00]).

Um den Phasenfehler zu berechnen, wird Δr benötigt.

$$\Delta r = r'(t) - r(t), \tag{14}$$

wobei $r(t)$ die nominelle und $r'(t)$ die tatsächliche Schrängentfernung ist. Mit Hilfe dieser Differenz kann eine Phasenfunktion berechnet werden, die multipliziert mit den Daten den

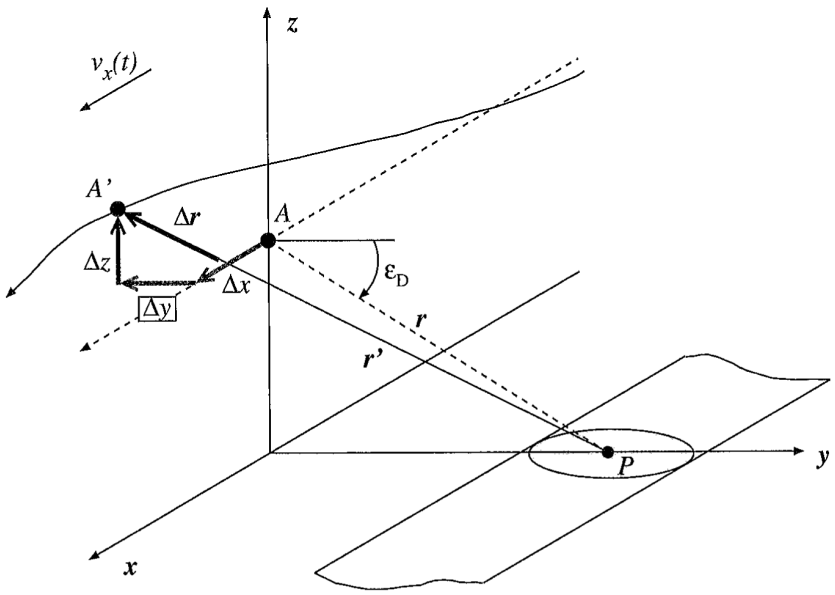


Abbildung 3.17: Grafische Veranschaulichung der Abweichung der tatsächlichen Flugbahn von der nominellen, entnommen aus [KH00], Seite 294

Phasenfehler korrigiert.

Somit ist der Phasenfehler nach [KH00]

$$\phi_{err}(t) = \phi(t) - \phi'(t) = -\frac{4\pi}{\lambda}\left[r'(t) - r(t)\right] \tag{15}$$

Die Daten können nun mit $e^{j\phi_{err}(t)}$ korrigiert werden.

Um nun die jeweiligen Schrägentfernungen zu berechnen, wird für r die Schrägentfernung aus der ersten Bewegungskompensation $rgScale$ herangezogen. Für die tatsächliche Schrägentfernung wird auf die Abweichung Δx, Δy und Δz zurückgegegriffen.

Aus der geometrischen Herleitung ergeben sich folgende Formeln:

$$\begin{aligned} gr &= \sqrt{rgScale^2 - (h_0 + \Delta z)^2} \pm \Delta y \\ r'(t) = real &= \sqrt{gr^2 + h_0^2 + (rgScale \cdot \tan\phi)^2} \\ \phi_{err}(t) = real &- \frac{rgScale}{\cos\phi} \end{aligned} \tag{16}$$

Die Abweichung Δx wurde bereits im Resampling-Schritt der Radardatenvorverarbeitung korrigiert.

Variablenname	**Typ**	**Größe**	**Beschreibung**
data	complex	entfernung×azimut	SAR-Datenmatrix
rgScale	double	entfernung	skalierter Entfernungsvektor
moco1	double	azimut	Moco1 Phase
h_0	double	entfernung×azimut	Flughöhe über Grund
Δy	double	azimut	horizontale Abweichung vom Referenzpfad
Δz	double	azimut	vertikale Abweichung vom Referenzpfad
λ	float	skalar	Wellenlänge
ϕ	float	skalar	Squintwinkel
lookDir	int	skalar	Blickrichtung

Tabelle 14: Schnittstellen für die Bewegungskompensation zweiter Ordnung in C

Schnittstellen

In den Implementierungen der Bewegungskompensation zweiter Ordnung der beiden Prozessoren gibt es einige Unterschiede, die vor allem im IDL-Wrapper zu beachten sind. Bevor auf dies eingegangen wird, folgt eine nährere Erläuterung der Schnittstelle zum C-Prozessor. Eine Übersicht darüber gibt Tabelle 14

lookDir bestimmt die Richtung der Antenne und somit, ob Δy in Formel 16 addiert oder subtrahiert werden soll. Weiterhin stellt *moco1* die Werte für $r(t)$ bereit, welche bereits in der ersten Bewegungkompensation berechnet und kompensiert wurden. Alle weiteren Variablen sind entsprechend denen, die im vorherigen Unterkapitel für die Formeln verwendet wurden.

Im IDL-Code sind die Variablen *rgScale*, h_0, y und z nicht ohne weiteres verfügbar und müssen deswegen innerhalb des IDL-Wrappers berechnet werden. Die hierfür vorgesehene Schnittstelle kann der Tabelle 15 entnommen werden.

Die Variable *topo* ist optional. Bei Angabe dieser, wird der Wert in *terrain* ignoriert. Diese beiden Parameter sind für die Berechnug von h_0 verantwortlich. Wird mit *terrain* gerechnet, wird lediglich dieser konstante Wert von einem Vektor abgezogen. Wird hingegen mit *topo* gerechnet, welches ein zweidimensionales Array ist, variiert der Subtrahent für

Variablenname	**Typ**	**Größe**	**Beschreibung**
data	complex	entfernung×azimut	SAR-Datenmatrix
pos	double	4×azimut	reale Positionen
ref	double	4×azimut	Referenzpositionen
moco1	double	azimut	Moco1 Phase
range delay	double	skalar	Signallaufzeit des ersten Samples
rs	double	skalar	Abtastung in Entfernung
λ	float	skalar	Wellenlänge
ϕ	float	skalar	Squintwinkel
lookDir	int	skalar	Blickrichtung
terrain	double	skalar	Topografie
topo	double	entfernung×azimut	Topografie (2D)

Tabelle 15: Schnittstellen für die Bewegungskompensation zweiter Ordnung in IDL

jedes Element. Nachfolgende Formel zeigt die Berechnung von h_0. Hierbei steht der Index von *ref* für die Spalte des 4-Spaltenvektors.

$$h_0 = ref_3 - topo \tag{17}$$

Für die Berechnung der von der GPU-Bibliothek benötigten Variablen y und z werden folgende Formeln verwendet.

$$\begin{aligned} y &= pos_2 - ref_2 \\ z &= pos_3 - ref_3 \end{aligned} \tag{18}$$

Auch *pos* ist ein 4-Spaltenvektor. Die verwendete Spalte wird wieder mit einem Index angegeben.

Schließlich wird *rgScal* berechnet. Hierfür werden die Paramter *range delay* und *rs* wie folgt verwendet:

$$\begin{aligned} rgScale &= range\_delay \cdot \frac{c_0}{2} + i \cdot rps \\ rps &= \frac{c_0}{2 \cdot rs} \end{aligned} \tag{19}$$

i ist eine Laufvariable von 0 bis *range* und c_0 die Lichtgeschwindigkeit.

Damit sind nun alle Unterschiede zwischen der IDL- und der C-Implementierung angepasst. Da jedoch viele Berechnungen vorgenommen wurden, stellt sich die Frage, wie sich dies auf die Performance der IDL-Implementierung auswirkt und ob Unterschiede zwischen den Rechenzeiten der beiden erkennbar sind. Bevor dies jedoch im Auswertungsabschnitt analysiert wird, folgt eine nähere Untersuchung zur Implementierung des Moduls auf der Grafikkarte.

Implementierung auf der GPU

Wie in den anderen Schritten, wurde der Kernel mit der gleichen Konfiguration festgelegt: Ein Block führt die Berechnungen für eine Zeile aus und besteht aus 512 Threads. Jeder Thread übernimmt die Aufgabe, mehrere Elemente der Zeile zu berechnen.

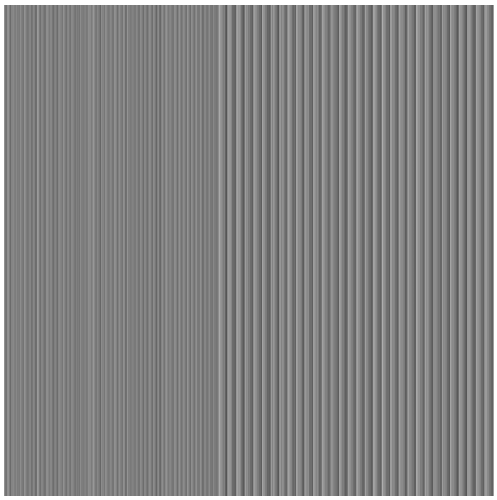
Dementsprechend werden die Texturen und Shared Memory Variablen wie folgt festgelegt. h_0 wird als Textur angelegt. Da dies ein zweidimensionales Array ist, muss es als CUDA-Array in den globalen Speicher kopiert werden. Linearer Speicher kann hierfür nicht verwendet werden. Weiterhin ist es wichtig anzumerken, dass Texturen nur mit einfacher Genauigkeit arbeiten. Doppelte Genauigkeit wird in der verwendeten CUDA-Version nicht unterstützt.

Das ist auch der Grund, warum *rgScale* nicht als Textur angelegt wird, so wie es in vorherigen Schritten mit gleich aufgebauten Vektoren geschehen ist. Mit einer einfachen Genauigkeit wären die Berechnungen für die Phase zu ungenau. Eine gründlichere Auseinandersetzung mit dieser Thematik folgt im anschließenden Paragraphen zur Auswertung. Statt der Textur wird *rgScale* also der Einfachheit wegen in den globalen Speicher der Grafikkarte transferiert und von dort aus abgerufen. Eine bessere Lösung wäre es, einen Teil des Vektor von *rgScale* in den Shared Memory zu speichern. Dazu müsste der Kernel so umkonfiguriert werden, dass die Blöcke in Abhängigkeit von der Größe des Vektors kleiner werden. Tests haben jedoch gezeigt, dass der Aufwand im Rahmen dieser Arbeit nicht in Verhältnis zum Nutzen steht. Darum sei hier nur auf diese Möglichkeit der Verbesserung der Performance hingewiesen.

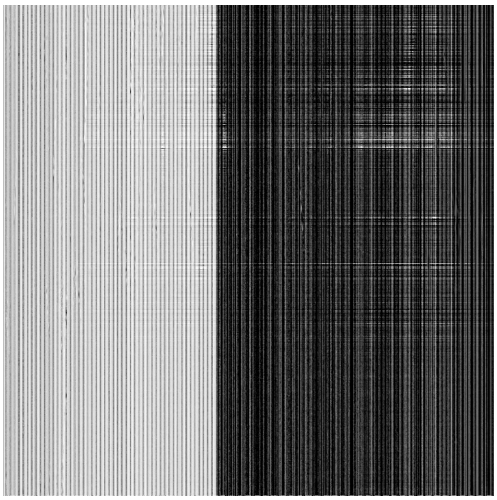
Sowohl Elemente von *moco1*, als auch y und z werden in den Shared Memory gespeichert. Die Berechnungen von *real* und *ground* erfolgt wie bereits beschrieben und werden als Makros im GPU-Kernel festgelegt. Dadurch soll Speicher in den Registern des Multiprozessors gespart werden.

Auswertung zur einfachen Genauigkeit von Gleitkommazahlen

Wie bereits erwähnt, kam es zu Fehlern in der Berechnung, wenn für *rgScale* eine einfache Gleitkommagenauigkeit verwendet wurde. Dieser Fehler konnte durch den Modultest F0-4 aufgedeckt werden. Die Bilder 3.18a und 3.18b zeigen deutlich, dass es innerhalb des Moduls zu Unstimmigkeiten kommt. Durch die Hilfe einzelner Plots konnte der Fehler identifiziert werden.



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.18: Phasendifferenz und Kohärenz nach der zweiten Bewegungskompensation mit einfacher Gleitkommagenauigkeit

Vor allem die Phase gibt Aufschluss über die Art, wie sich die Ungenauigkeit auf das Endergebnis auswirkt. Weiterhin soll die Abbildung 3.19 herangezogen werden, um die Thematik noch deutlicher zu visualisieren.

In den Bildern ist zu erkennen, dass der Fehler mit steigender X-Koordinate, der Entfernungsdimension, größer wird. Ab Spalte 4000 steigt dieser sprungartig an (die Linien werden dichter und befinden sich außerhalb der Skala). Bei näherem Betrachten, würde man feststellen, dass der Fehler der Phasendifferenz auf einen Fehlerwert anwächst und danach auf 0 zurückfällt. Dies ist zurückzuführen auf die Auflösungsgenauigkeit einer Floatzahl.

Innerhalb der Grauzone in Bild 3.20 reicht der Speicher nicht mehr aus, um die Zahlen 3124,4322 und 3124,4327 voneinander zu unterscheiden. Stattdessen werden $2 \cdot 10^{-7}$ und $7 \cdot 10^{-7}$ abgeschnitten. Fälschlicherweise werden somit alle Werte innerhalb des grauen Bereichs als Wert 3124,432 intepretiert. Es entsteht ein Fehler von bis zu $9{,}\tilde{9} \cdot 10^{-7}$. Erst

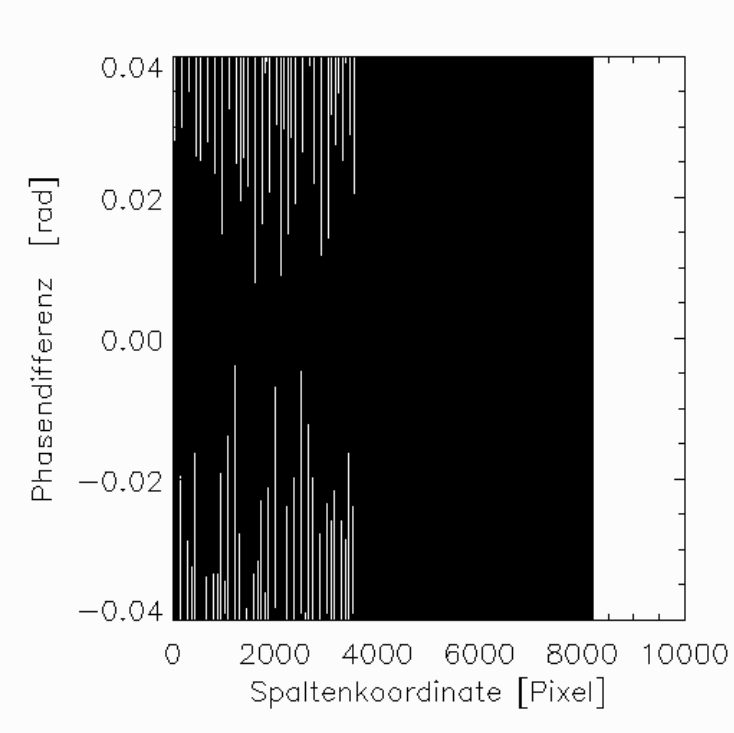


Abbildung 3.19: Plot einer Zeile der Phasendifferenz in Rad nach der zweiten Bewegungskompensation

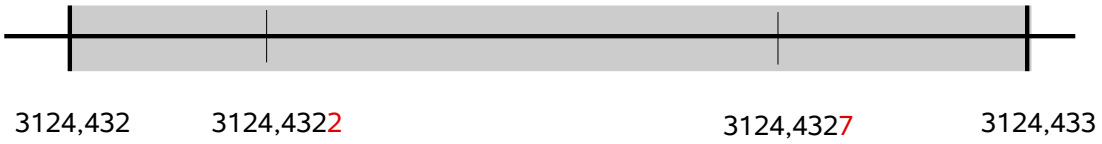


Abbildung 3.20: Veranschaulichung der Auflösung einer Gleitkommzahl mit einfacher Genauigkeit

wenn ein Wert auf 3124,433 ansteigt, wird der Fehler wieder geringer. Dies erklärt die ständigen Phasensprünge, die man in Abbildung 3.18a als viele Gradienten sehen kann. Dieser Fehler liegt bei Gleitkommazahlen mit einfacher Genauigkeit bei ungefähr 10^{-7} und kann je nach Größenordnung der Zahlen abweichen. Er wird als Unit in the Last Place (ULP) bezeichnet.

Der Sprung bei 4000 liegt daran, dass die Werte des *rgScale* Vektors ansteigend sind

und ab dem Wert 4000 in eine andere Größenordnung fallen. Somit wird der nicht unterscheidbare Bereich größer.

Um dieser Problematik aus dem Weg zu gehen, wird der *rgScaleCurvature*-Vektor als Gleitkommavektor mit doppelter Genauigkeit angelegt. Erst nach der Verrechnung von anderen Werten, deren Werte in einem wesentlich kleineren Bereich liegen, wird er auf einfache Genauigkeit konvertiert. Somit steht während der Rechnung die Genauigkeit zur Verfügung, die benötigt wird, um die relative hohen Werte mit den relative kleinen Werten zu verrechnen.

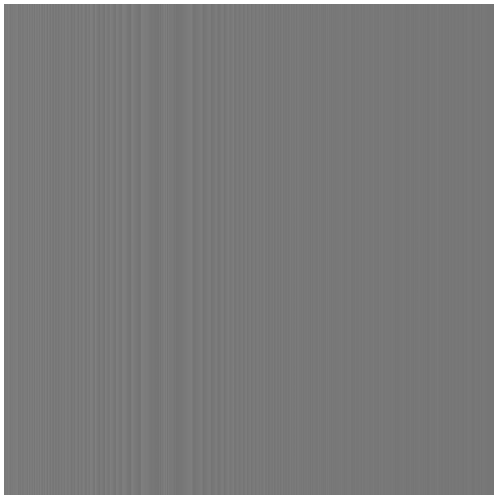
Daraus lässt sich generell schlussfolgern, dass stets eine höherere Genauigkeit nötig ist, sobald Berechnungen mit Werten stattfinden, die sich sehr in ihrer Größenordnung unterscheiden.

Auswertung der C-Implementierung

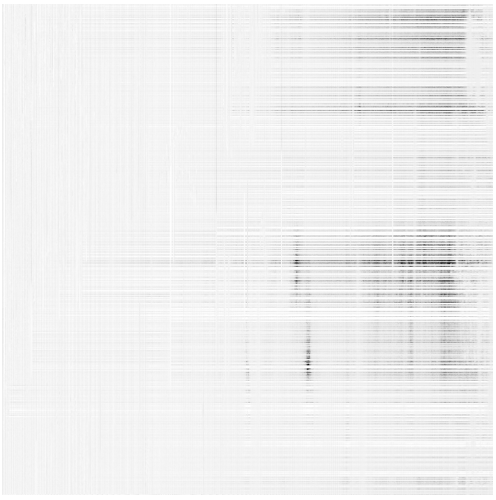
Die Auswertung des Tests F0-4 mit doppelter Genauigkeit des Vektors *rgScale* zeigt, dass der Fehler soweit behoben werden konnte. Eine vollständige Übereinstimmung zum Referenzbild konnte bisher jedoch nicht erreicht werden, wie die Bilder 3.21a und 3.21b zeigen. Dennoch liegen die Ergebnisse innerhalb der Toleranz. Für spätere Versuche kann auch das zweidimensionale Array h_0 zur Verbesserung des Ergebnisses auf doppelte Genauigkeit angepasst werden.

Wie Tabelle 16 sehr klar darstellt, ist die Verbesserung der Verarbeitungsgeschwindigkeit nicht sehr stark ausgeprägt. Hierbei wird die Verarbeitung nur um einen Faktor von 3,4 beschleunigt. Dies liegt vor allem an den großen Datenmengen, die übertragen werden müssen. Dazu gehören zum einen die SAR-Daten selbst, als auch das zweidimensionale Array h_0. Weiterhin müssen viele Parameter übergeben werden. Dementsprechend häufig wird aus dem Speicher der Grafikkarte gelesen. Hierbei können die Stärken der Grafikkarte nicht so intensiv genutzt werden wie in den anderen Modulen. Wie weitere Test ergeben haben, spielt auch die Konvertierung von h_0 nach float eine wesentliche Rolle. Wird die Zeitmessung nach der Konvertierung gestartet, entsteht eine Zeitdifferenz von über 2 Sekunden. Im Zuge einer weiteren Optimierung sollte h_0 deswegen als `float` in der Shared Object Datei angelegt werden.

Schließlich kommt hinzu, dass die Implementierung auf der CPU relativ performant implementiert ist. Der rechenintensive Verarbeitungsschritt des Resamplings innerhalb der zweiten Bewegungskompensation wurde in dieser Version noch nicht implementiert. Mit Hinzukommen dieses Features könnte der Beschleunigungsfaktor um einiges höher ausfal-



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.21: Phasendifferenz und Kohärenz nach der Bewegungskompensation zweiter Ordnung

	min [ms]	max [ms]	mean [ms]
CPU	9582	9969	9716,73
GPU	2786	3351	2852,23
Faktor	3,44	2,97	3,41

Tabelle 16: Rechenzeiten in ms für das Modul der Bewegungskompensation zweiter Ordnung auf CPU und GPU

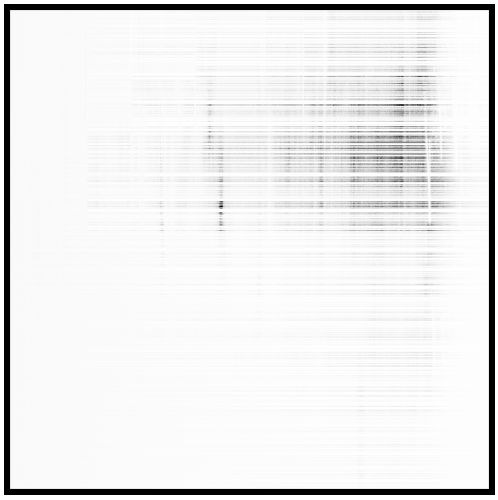
len.

Auswertung der IDL-Implementierung

Wie die Auswertungsbilder 3.22a und 3.22b zeigen, stimmt die Implementierung mit den Referenzdaten überein. Ein Blick auf einen einzelnen Plot B.15c zeigt jedoch, das es bei der Phase zu einem Offset kommt. In Plot B.16c kann man wiederum deutlich erkennen, dass die Abweichung sich quadratisch verhält. Demzufolge ist noch ein kleiner Fehler vorhanden, der durch die Verwendung von Floats kommen kann. Mit Hilfe von weiteren Untersuchungen soll auch diese letzte Unstimmigkeit getilgt werden.

Wie vorher angemerkt, wurden innerhalb des IDL-Wrappers einige Berechnungen ange-

(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.22: Phasendifferenz und Kohärenz nach der Bewegungskompensation zweiter Ordnung in IDL

	min [ms]	max [ms]	mean [ms]
CPU	13325,04	13479,47	13351,75
GPU	3037,93	3161,22	3093,17
Faktor	4,37	4,26	4,32

Tabelle 17: Rechenzeiten in ms für das Modul der Bewegungskompensation zweiter Ordnung auf CPU und GPU in IDL

stellt, um den Schnittstellen der Bibliothek zu genügen. Tabelle 17 zeigt, dass die Berechnungen von im Durchschnitt 200ms nicht stark ins Gewicht fallen. Viel zeitaufwändiger gestaltet sich das Konvertieren von Double nach Float innerhalb der Bibliothek. Weiterhin kann man beim Vergleichen mit der Tabelle 16 feststellen, dass der IDL-Prozessor 1,4 mal langsamer ist als der C-Prozessor.

3.3.5 Azimutkompression

Nachdem nun die Zielentfernungsänderung eliminiert und die Bewegungskompensation abgeschlossen ist, kann die Azimutkompression stattfinden. Durch die vorherigen Schritte ist es möglich, diese auf eine Multiplikation der Daten mit einer Azimutreferenzfunktion zu

vereinfachen. Diese Referenzfunktion ist eine eindimensionale Funktion mit hyperbolischen Verlauf und berechnet sich wie ein komplex-konjugierter Azimut-Chirp:

$$\begin{aligned} azimutfilter(t_a, r_0) &= \exp\left[j\frac{4\pi}{\lambda}r(-t_a, r_0)\right] \\ azimutfilter(t_a, r_0) &\circ\!\!-\!\!\bullet\, azimutfilter(f_a, r_0) \\ azimutfilter(f_a, r_0) &= \exp\left[-\frac{4\pi}{\lambda}\cdot r_0 \cdot \left(1 - \sqrt{1 - \left(\frac{\lambda f_a}{2v}\right)^2}\right)\right] \end{aligned} \tag{20}$$

Schnittstellen

Im Gegensatz zur Implementierung in IDL, berechnet der C-Prozessor neben der Azimutreferenzfunktion eine Antennencharakteristik-Korrektur. Diese geschieht auf Grundlage der Eigenschaften der Antenne und beinhaltet lediglich eine weitere komplexe Skalierung der Daten mit einem Vektor.

Aus diesem Grund bietet die GPU-Bibliothek einen weiteren Parameter für einen Vektor, der jedoch von der IDL-Schnittstelle nicht unterstützt wird. Wird an die Bibliothek eine 0 anstatt eines Vektors übergeben, wird keine Pattern-Korrektur durchgeführt.

Tabelle 18 zeigt die Definition der Schnittstelle mit Anmerkung der Unterschiede zwischen IDL und C.

Implementierung auf der GPU

Wie bereits in Kaptiel 3.3.3 erwähnt, wurden die entsprechenden Fouriertransformationen nicht im Schritt der Zweiten Bewegungskompensation durchgeführt. Stattdessen wird in diesem Schritt neben der inversen FFT über die Azimut-Dimension am Ende auch eine Vorwärtstransformation durchgeführt am Anfang. Für die Implementierung bedeutet das, dass zuerst die Daten transponiert werden müssen. Erst im Darauffolgenden kann die FFT angewendet werden. Um Rechenzeit zu sparen, werden die Daten jedoch nicht zurücktransponiert. Stattdessen muss der Kernel für die Azimutkompression die Indizes für die Daten anders berechnen.

Statt einer Berechnung wie bisher

$$index = y \cdot dim_x + x, \tag{21}$$

wobei dim_x die Größe der Entfernungsdimension darstellt und x und y die jeweiligen Indizes für die Entfernungs- und Azimutpositionen sind, erfolgt sie bei transponierten Daten

Variablenname	**Typ**	**Größe**	**Anmerkung und Beschreibung**
data	complex	entfernung×azimut	SAR-Datenmatrix
f_a	double	azimut	Azimutfrequenzvektor
β	double	azimut	β-Faktor [MMS96]
rgScale	double	entfernung	skalierter Entfernungsvektor
window	float	entfernung	spektrale Gewichtung
ϕ	float	skalar	Squintwinkel
v	float	skalar	Vorwärtsgeschwindigkeit
λ	float	skalar	Wellenlänge
patternCorrection	double	azimut	Azimut-Antennendiagramm, optional, jedoch nicht für IDL verfügbar

Tabelle 18: Schnittstellen für die Azimutkomprimierung

folgendermaßen:

$$index^T = x \cdot dim_y + y \tag{22}$$

Zur Komprimierung werden, wie bereits üblich, die zwei Faktoren `aux1` und `aux2` definiert. Die Berechnung der Faktoren sieht wie folgt aus:

$$\begin{aligned} aux1 &= 4\pi \frac{\beta - 1}{\lambda} \\ aux2 &= 2\pi \cdot f_a \frac{\tan\phi}{v} \end{aligned} \tag{23}$$

Damit lässt sich die Phase wie folgt berechnen:

$$phase = \exp\left[aux1 \cdot rgScale + aux2 \cdot rgScale\right] \tag{24}$$

und wird abschließend noch mit der übergebenen Gewichtungsfunktion `window` skaliert.

Obwohl die Daten in transponierter Form vorliegen, wird der Kernel wie bereits in den anderen Schritten definiert. Hierbei sind `aux1` und `aux2` Variablen im Shared Memory, sie werden pro Block nur einmal berechnet. Auch die Gewichtung wird in dieser Konfiguration

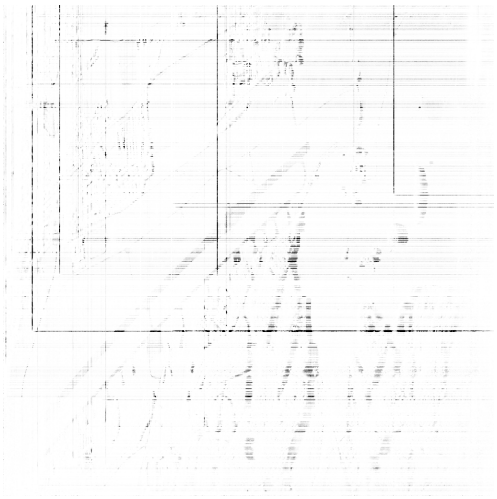
pro Block nur einmal gelesen. Aus diesem Grund wird auch sie in den Shared Memory geladen.

Die elementweise Verarbeitung erfolgt dieses Mal ohne Textur, da kein größerer Vektor vorliegt, der gecached werden müsste. Einzig $rgScale$ wird pro Thread an unterschiedlichen Stellen gelesen. Da aber jeder Thread nur einmalig auf diese Daten zugreift, lohnt eine Textur in diesem Fall nicht. Stattdessen wird das gelesene Datum in einem Register der Prozessoren der Grafikkarte geladen. Diese Register sind sehr kleine, aber schnelle Speicher, die einzelne Skalare aufnehmen können. Reicht der Speicher für einen Kernelaufruf nicht aus, werden sie in dem Global Memory gespeichert. In solchem Fall können die Zugriffszeiten auf diese Variablen erheblich ansteigen. Aus diesem Grund ist es wichtig, stets auf den verfügbaren Speicher zu achten auch während der Kompilierung. Um zu prüfen, ob der sogenannte Local Memory genutzt wird, steht zum einen das Überprüfen des Assembler-Codes zur Auswahl und zum anderen kann man sich mit Hilfe eines Compiler-Flags den genutzten Speicher pro Multiprozessor anzeigen lassen. Näheres dazu gibt es in Quelle [NVi10a] Kapitel 5.3.2.2.

Zum Abschluss folgt die inverse FFT gefolgt von einer Transposition.

Auswertung der C-Implementierung

(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.23: Phasendifferenz und Kohärenz nach der Azimutkomprimierung

Die Implementierung der Azimutkompression auf die Grafikkarte hat nur eine geringe

	min [ms]	max [ms]	mean [ms]
CPU	58934	64895	59437.5
GPU	852	1004	870,65
Faktor	69,17	64,64	68,27

Tabelle 19: Rechenzeiten in ms für das Modul der Azimutkompression auf CPU und GPU

	min [ms]	max [ms]	mean [ms]
CPU	17441,52	19980,95	18081,20
GPU	670	763	687,31
Faktor	26,03	26,19	26,31

Tabelle 20: Rechenzeiten in ms für das Modul der Azimutkompression auf CPU und GPU in IDL

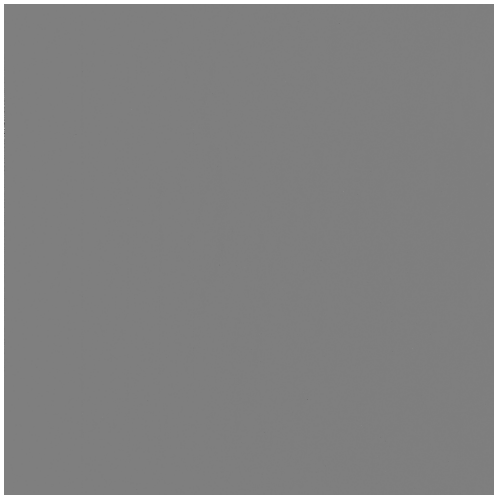
Beeinflussung auf die Qualität. Die Abbildungen 3.23a und 3.23b zeigen die Ergebnisse des Testdurchlaufs F0-5. Anhand dieser Bilder wird deutlich, dass es zu einigen Unstimmigkeiten mit dem Referenzbild kommt. Da bei der Azimutkompression mit Funktionen wie Kosinus und Sinus gearbeitet wird, können diese Abweichungen auf Ungenauigkeiten der genannten Funktionen zu Gunsten der Performance zurückgeführt werden. Diese Ungenauigkeiten drücken sich vor allem in der Kohärenz aus. Dort sind einige Streifen erkennbar, in denen die Kohärenz geringer ist. Trotz dieser schwarzen Streifen liegt die Übereinstimmung hoch genug, um damit weiterarbeiten zu können.

In diesem Modul konnte eine gute Performance erzielt werden, wie der Faktor in Tabelle 19 zeigt.

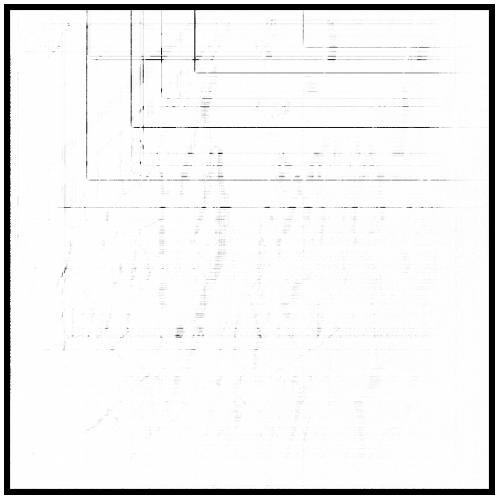
Auswertung der IDL-Implementierung

Da bis auf die Pattern-Korrektur kaum Unterschiede in den Implementierungen liegen, liefern die Bilder zu Qualitätsauswertung des Tests I0-5 ein ähnliches Ergebnis.

Die Tabelle 20 zeigt einen Unterschied in den beiden Implementierungen. In der IDL-Variante sind kürzere Rechenzeiten zu verzeichnen, da in dieser Version keine Pattern-Korrektur durchgeführt wird. Dies schlägt sich mit 170ms auf die Ergebnisse nieder. Demnach sind die Faktoren nicht genauso hoch oder gar höher wie in der C-Implementierung, da auch hier der IDL-Prozessor parallelisiert rechnet.



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.24: Phasendifferenz und Kohärenz nach Azimutkomprimierung in IDL

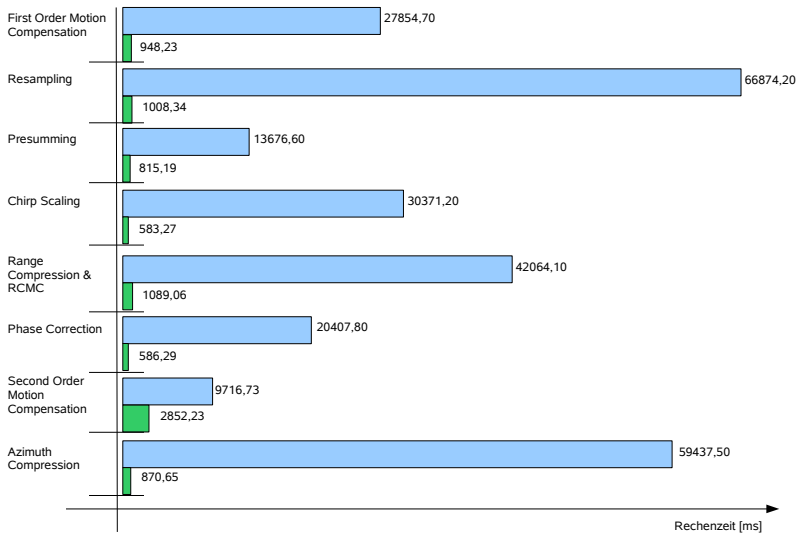
3.4 Auswertung der Tests

Die Abbildungen 3.27 zeigen eine Gesamtübersicht über den Modultest. Wie daran deutlich zu erkennen ist, konnte durch die Portierung auf die Grafikkarte sehr viel Rechenzeit gespart werden. Da für den IDL-Prozessor das Resampling noch nicht fehlerfrei funktionierte, wurde dies für die folgenden Auswertungen durch die CPU-Implementierung ersetzt.

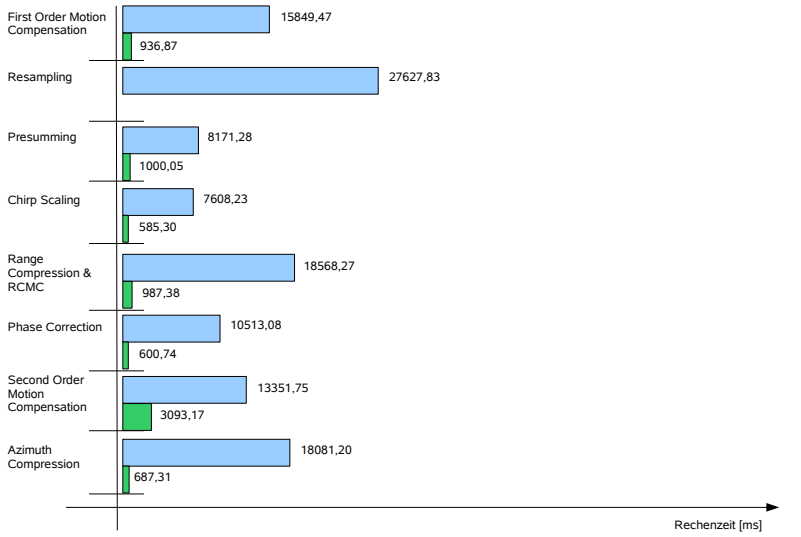
Nun, da für jeden Algorithmenschritt ein Modul geschrieben wurde, muss auch geprüft werden, ob diese in Zusammenarbeit genauso einwandfrei funktionieren und wie groß die Performancesteigerung für diesen Fall ist. Hierzu sollen wieder die Betrachtungen in C und IDL getrennt erfolgen.

3.4.1 Auswertung des Extended Chirp Scalings im C-Prozessor

Zur Auswertung des Gesamtergebnisses wurde für den C-Prozessor der Test F1 durchgeführt. Die Ergebnisse sind unter anderem in den Bildern 3.26a und 3.26b zu sehen. Hieran wird deutlich, dass vor allem die Berechnungen der Phase auf der Grafikkarte mit dem Referenzbild übereinstimmt. Die Kohärenz weist jedoch weniger gute Ergebnisse auf. Im Allgemeinen liegt die Übereinstimmung bei 0,98. Bei näheren Untersuchungen hat sich herausgestellt, dass diese Unstimmigkeiten vor allem durch den Resampling-Schritt hervorgerufen werden.

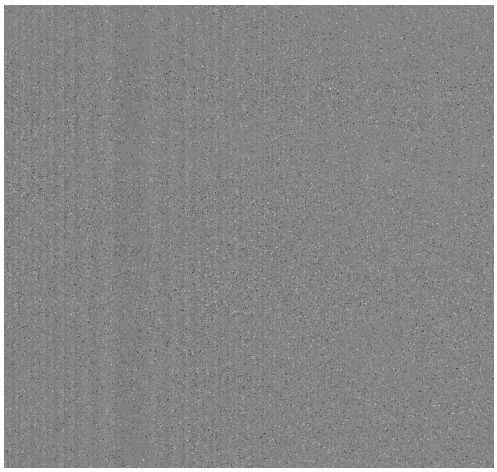


(a) Rechenzeiten für FSARCPP

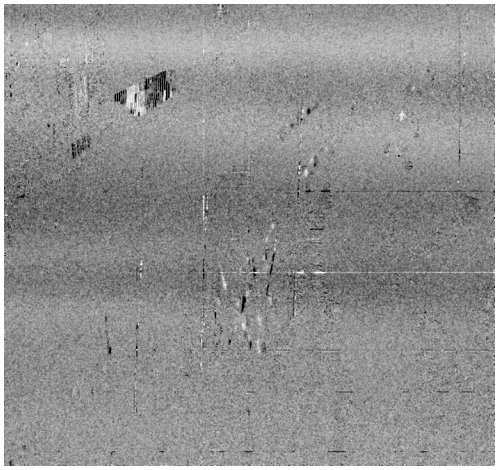


(b) Rechenzeiten für STEP

Abbildung 3.25: Rechenzeiten der einzelnen Module in ms, grün auf GPU, blau auf CPU



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,98)

Abbildung 3.26: Phasendifferenz und Kohärenz des Endbildes

In diesem Modul wird eine Interpolation durchgeführt. In der Referenzimplementierung wird mit Splines gearbeitet, während auf der Grafikkarte die lineare Interpolation von Texturen genutzt wird. Eine Bibliothek, die eine kubische Interpolation mit Texturen erlaubt, wurde ebenfalls getestet. Da die Ergebnisse keine Verbesserungen zeigten, soll als dritte Möglichkeit der Knab-Interpolator[3] getestet werden. Da dieser nicht mit Texturen funktioniert und die Implementierung einer größeren Anpassung bedarf, wurde dies im Rahmen dieser Arbeit nicht weiter untersucht.

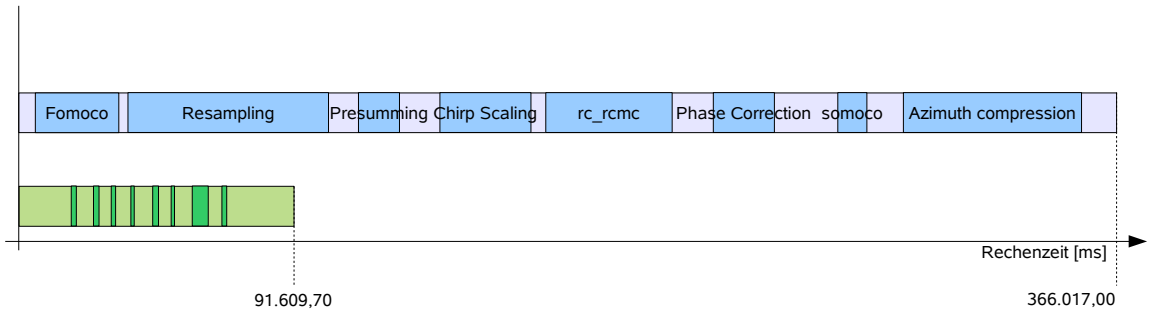


Abbildung 3.27: Darstellung der Gesamtrechenzeiten in ms des C-Prozessors, grün auf GPU, blau auf CPU

[3]Interpolationsverfahren mit Hilfe eines Sinus cardinalis (siehe Quelle [MNBC07])

CPU:	$\sum$ Module:	270.402,83	ms
	Gesamt:	366.017,00	ms
	Modulanteil:	73,88	%
	Δ:	95.614,17	ms
GPU:	$\sum$ Module:	12.394,23	ms
	Gesamt:	91.609,70	ms
	Modulanteil:	13,53	%
	Δ:	79.215,47	ms
Faktor:	Module:	21,82	
	Gesamt:	4,00	

Tabelle 21: Gegenüberstellung der Gesamtrechenzeiten der Radarprozessierung auf der CPU und GPU und Anteil der Rechenzeiten für die Module

Neben den Betrachtungen der Qualität ist auch die Performanceanalyse wichtig. Um die aufgenommen Zeiten bildlich zu veranschaulichen, dient die Abbildung 3.27. Aus Platzgründen wurden nicht die vollständigen Bezeichnungen der Modulnamen angegeben und bei dem Balken für die GPU-Rechenzeiten gänzlich weggelassen. Die Reihenfolge der Module ist: Bewegungskompensation erster Ordnung (Fomoco), Resampling, Presumming, Chirp Scaling, Zielentfernungskorrektur und Entfernungskompression (rc_rcmc), Phasenkorrektur, Bewegungskompensation zweiter Ordnung (Somoco), Azimutkompression.

Die Abbildung zeigt deutlich, dass durch die Verwendung der Grafikkarte Zeit eingespart werden konnte. Jedoch erscheinen die Rechenzeiten auf der GPU etwas nüchtern. Vor allem ist darauf deutlich zu erkennen, dass der größte Teil der Verarbeitung in der C-Implementierung durch die Module eingenommen wird. In der GPU-Implementierung nehmen die Modulberechnungen jedoch nur einen sehr geringen Teil ein. Was durch Abbildung 3.27 bildlich veranschaulicht wird, soll in Tabelle 21 mit Zahlen untermauert werden.

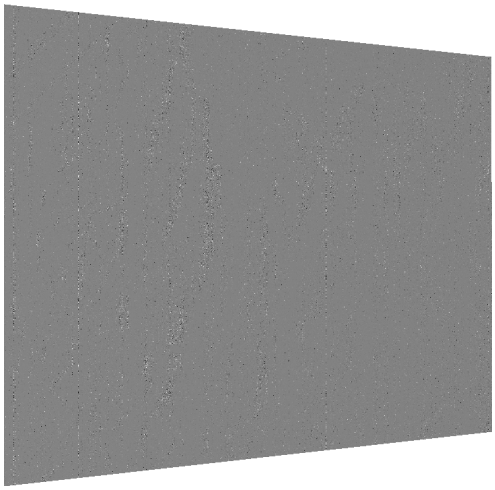
Während in der CPU-Implementierung die Rechenzeit für die Module zu $73,88\%$ eingeht, nehmen sie in der GPU-Implementierung nur einen Anteil von $13,53\%$ ein. Dies liegt an drei Dingen: Zum einen kommt es beim Einlesen und Schreiben der Daten zu Zugriffen auf der Harddisk. Diese sind wesentlich langsamer, als die Zugriffszeiten auf den Hauptspeicher und tragen damit stark zur Gesamtrechenzeit bei. Weiterhin wird eine Quadraturmodulation vorgenommen, die nicht für die Grafikkarte portiert ist. Beide Gründe fallen für die Onboard-Prozessierung weg und können deswegen vernachlässigt werden.

Der dritte Grund hingegen, das Fehlen der radiometrischen Kalibrierung, muss für den späteren Echtzeiteinsatz für die Grafikkarte behoben werden, das heißt ebenfalls auf die GPU portiert werden.

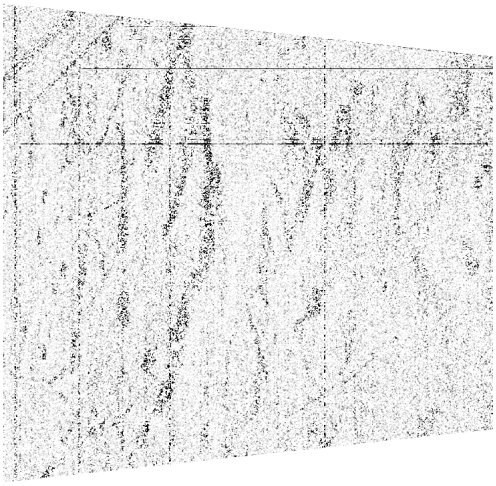
All diese Gründe führen dazu, dass lediglich ein Beschleunigungsfaktor von 4 erreicht wird, wohingegen der Faktor der reinen Modulrechenzeit 21, 82 beträgt. Somit wird deutlich, dass noch großes Potential für diese Prozessierung besteht. Vor allem ist zu beachten, dass für jedes Modul ungefähr die Hälfte der Zeit an Kopiervorgänge und Shared Object Latenz anfällt. Somit können durch weitere Optimierungen höhere Faktoren erreicht werden, um so den Anforderungen an die Echtzeitprozessierung gerecht zu werden.

3.4.2 Auswertung des Extended Chirp Scalings im IDL-Prozessor

Die Auswertung zum Test I1 haben gezeigt, dass das mit der Grafikkarte berechnete Bild besser mit dem Referenzbild übereinstimmt als in der C-Implementierung. Eine visuelle Veranschaulichung bieten Abbildungen 3.28a und 3.28b. Hierbei ist jedoch zu beachten, dass kein Resampling-Schritt für IDL zur Verfügung steht und deswegen auf die IDL-Implementierung zurückgegriffen wird. Aus diesem Grund ist auch eine höhere Kohärenz zu erkennen.



(a) Phasendifferenz (weiß = $\frac{\pi}{10}$ rad, schwarz = $-\frac{\pi}{10}$ rad)



(b) Kohärenz (weiß = 1, schwarz = 0,999)

Abbildung 3.28: Phasendifferenz und Kohärenz des Endbildes

Dementsprechend kann aber keine hohe Performancesteigerung wie im C-Prozessor erwartet werden. Nicht nur, dass der wohl rechenintesivste Schritt nicht auf der Grafikkarte

läuft, auch der höher parallelisierte Code im IDL-Prozessor sorgt dafür, dass ein geringerer Beschleunigungsfaktor erzielt werden konnte. Die bildliche Veranschaulichung folgt in Abbildung 3.29. Aus Platzgründen wurden die Bezeichnungen der einzelnen Schritte abgekürzt und über die entsprechenden Balken geschrieben. In den GPU-Balken wurden sie gänzlich weggelassen.

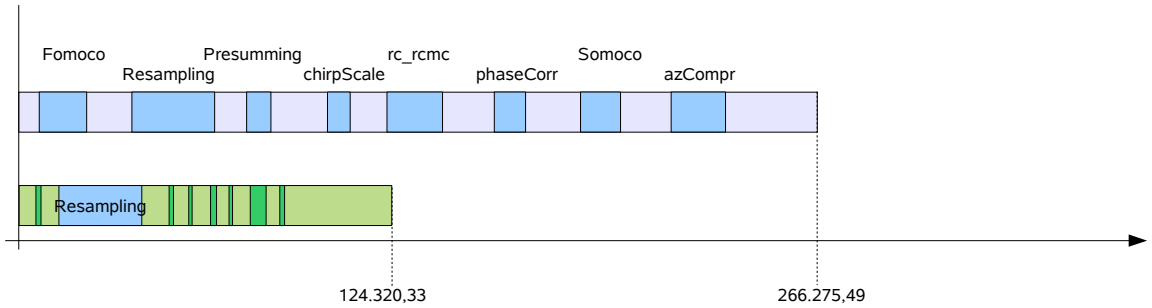


Abbildung 3.29: Darstellung der Gesamtrechenzeiten in ms des IDL-Prozessors, grün auf GPU, blau auf CPU

Auch im IDL-Prozessor bilden die Module einen wesentlichen Anteil an der Gesamtrechenzeit der Radarverarbeitung. Durch die Auslagerung der Berechnungen auf die Grafikkarte hat sich deren Anteil auf ein Minimum verringert. Mit einem erfolgreichen Bereitstellen des Resampling-Schrittes könnte die Gesamtrechenzeit ein weiteres Mal stark verringert werden. Wie Abbildung 3.25b zeigt, beträgt die Laufzeit des Resamplings 27 Sekunden. Ein Einbeziehen davon in die Modulrechenzeiten ergibt einen Anteil von 45%. Unter Annahme der gleichen Rechenzeit für das Resampling auf der GPU wie im C-Prozessor ergäbe dies eine Beschleunigung um den Faktor 13,46 für die Modulrechenzeit.

In Tabelle 22 ist die Gegenüberstellung der Rechenzeit für IDL dargestellt. Da der Modulanteil von 34,60% verhältnismäßig gering ist, wirkt sich selbst eine Beschleunigung um den Faktor 11,68 nur gering auf die Gesamtrechenzeit aus. Grund für den geringen Anteil ist zum einen das Ausschließen des Resamplings, aber auch die Schreib- und Lesevorgänge, die Kalibrierung des Prozessors und der nicht auf die Grafikkarte portierte Schritt der radiometrischen Korrektur.

Neben diesem Vergleich zwischen CPU- und GPU-Implementierung ist auch ein Vergleich zwischen IDL- und C-Prozessor möglich. Die Tabellen 21 und 22 zeigen, dass der STEP-Prozessor 102s schneller zu einem Ergebnis kommt als der C-Prozessor, obwohl beide

CPU:	$\sum$ Module:	92.143,28	ms
	Gesamt:	266.275,49	ms
	Modulanteil:	34,60	%
	Δ:	174.132,21	ms
GPU:	$\sum$ Module:	7.890,83	ms
	Gesamt:	124.320,33	ms
	Modulanteil:	6,00	%
	Δ:	116.429,50	ms
Faktor:	Module:	11,68	
	Gesamt:	2,29	

Tabelle 22: Gegenüberstellung der Gesamtrechenzeiten der Radarprozessierung auf der CPU und GPU und Anteil der Rechenzeiten für die Module der IDL-Implementierung

mit den gleichen Daten und mit der gleichen Datenmengen arbeiten. Die Gründe dafür zu finden lag nicht im Aufgabenbereich dieser Bachelorarbeit, dennoch konnte ein wesentlicher Faktor identifiziert werden: Die IDL-Prozessierung arbeitet häufiger parallel auf mehreren Prozessoren.

4 Zusammenführung der einzelnen Verarbeitungsschritte

Wie die Auswertungen der Tests gezeigt haben, konnten zwar für die einzelnen Module hohe Verbesserungen vermerkt werden. Jedoch erkennt man an den Test I1 und F1, dass in der Gesamtprozessierung die Module noch nicht effizient genug arbeiten. Wie bereits schon angesprochen, sollte die Radarverarbeitung gänzlich auf die GPU ausgelagert werden, ohne dabei mehrfach in den RAM der CPU zu kopieren. Erst dann können wesentlich kürzere Rechenzeiten vermerkt werden.

Anhand der Abbildung 4.1 wird deutlich, dass die reine Rechenzeit, der Kernel, nur einen Bruchteil von dem ausmacht, was während der Tests gemessen wurde. Neben den Kernelzeiten kommen die Übertragungszeiten zwischen Host und Device hinzu, die in der Regel mehr als doppelt so lange brauchen. Im Falle der Zielentfernungskorrektur und Entfernungskompression (rcrcmc) ist genau das Gegenteil der Fall. Dies kann zum einen an der

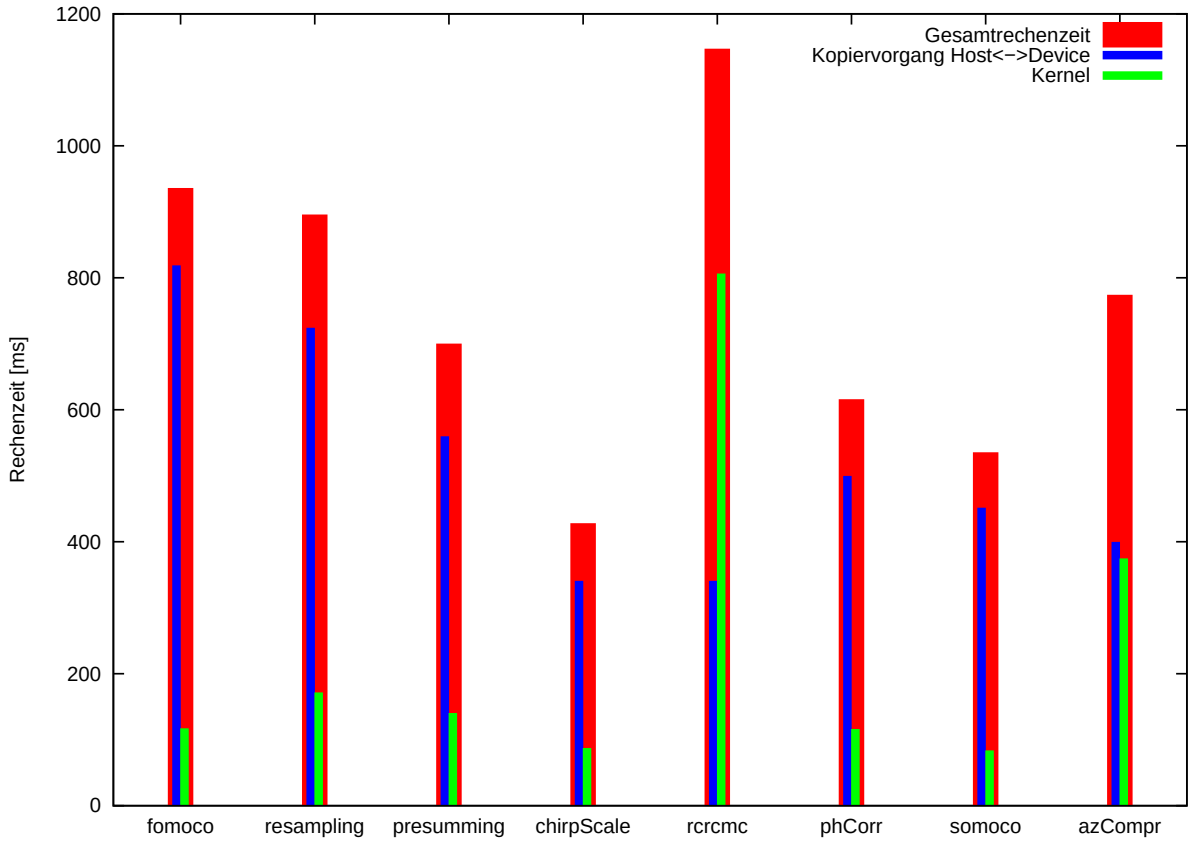


Abbildung 4.1: Darstellung der Rechenzeiten der einzelnen Schritte, aufgeschlüsselt in Kopiervorgänge und Kernelausführung für die Integration im C-Prozessor

höheren Auslastung der Grafikkarte liegen, aber auch daran, dass eine geringere Datenmenge kopiert, als letztendlich im Kernel verrechnet wurden (siehe hierzu Kapitel 3.3.2). Auch bei der Azimutkompression sind die Rechenzeiten des Kernels länger als bei anderen, da hier die Pattern-Korrektur hinzukommt (siehe Kapitel 3.3.5).

Alles in allem wird durch diese Grafik sehr deutlich, dass trotz der guten Modulergebnisse viel Latenzzeit durch die Kopiervorgänge zu Stande kommt. Dies bedeutet vor allem verschenkte Rechenzeit durch zum Teil mehrfaches Kopieren gleicher Daten. Aus diesem Grund sollen alle Rechenschritte der Radarverarbeitung gemeinsam in einem Kernel ausgelagert werden. Im Folgendem soll zuerst das Konzept des Programmes erläutert werden. Da im Rahmen dieser Bachelorarbeit noch keine Ergebnisse vorliegen, wird in dem Kapitel 4.2 lediglich eine Abschätzung gegeben.

4.1 Konzept

Wie in Kapitel 3.1 dargestellt wurde, wird die Implementierung der Grafikkarte als dynamische Bibliothek umgesetzt. Der Name dieser Bibliothek lautet GPU accelerated SAR processing (kurz GASP) und besteht aus 4 Dateien: Dem Shared Object der Bibliothek und seiner Header-Datei, sowie für die Integrierung in IDL einem zweiten Shared Object und die zugehörige Datei zur Beschreibung der Funktionen.

Damit der Code sowohl für den C- als auch IDL-Prozessor und nicht nur als kompletter Code, sondern auch schrittweise einsetzbar ist, wurde eine spezielle Struktur für die Implementierung der Bibliothek entwickelt. Diese Struktur ist als Schema in Abbildung 4.2 dargestellt.

Zunächst erfolgt eine Definition der Schnittstellen in `gasp.h`. Diese Datei wird von allen Nutzern der Bibliothek inkludiert und informiert sie über die bereitgestellten Funktionen. In `gasp.c` erfolgt die Implementierung dieser Funktionen. Da ein einfaches Mischen von CUDA-Code und gewöhnlichen C-Code nicht möglich ist, werden die CUDA-Funktionen wiederum in zwei andere Dateien ausgelagert. Dies ermöglicht eine Benutzung von GASP ohne dabei auf CUDA-spezifische Einstellungen achten zu müssen. In den beiden gasp-Dateien ist reiner C-Code implementiert, weswegen diese Daten in der Abbildung 4.2 gelb kodiert sind.

In den erwähnten anderen beiden Dateien `gpu.cuh` (Header-Datei) und `gpu.cu` erfolgt die eigentliche Implementierung. In der Header-Datei werden wieder die Prototypen der Funktionen festgelegt und einige Makros definiert, um zum Beispiel Fehler leichter zu handhaben oder das Benchmarking bequemer einzustellen.

Die Implementierung des GPU-Codes ist in `gpu.cu` wiederum zweigeteilt. Einmal existieren Funktionen mit dem Präfix `gpu_`, die die Speicherverwaltung übernehmen und die Funktionen mit dem Präfix `dev_` aufrufen. In `dev_`-Funktionen wird ausschließlich mit den Daten der Grafikkarte gearbeitet und die entsprechenden Kernels aufgerufen. Diese Zweiteilung vereinfacht die Umsetzung der einzelnen Module als zusammenhängenden Schritt. Für die Implementierung der gesamten Radarverarbeitung in einem Kernel muss im Idealfall nur eine weitere Funktion geschaffen werden, die alle nötigen Daten kopiert und nacheinander die einzelnen Schritte aufruft. Ohne diese Funktion werden die Daten für jeden Schritt in der jeweiligen `gpu_`-Funktion allokiert, kopiert und nach Ausführen des Algorithmus' wieder zurück in den Speicher des Hosts kopiert. Die Mischung des Host- und Device-Codes wird in Abbildung 4.2 farblich als Mischung von gelb und rot dargestellt.

Zu guter Letzt existieren zwei weitere Dateien. Zum einen `kernel.cu`, in dem die Ker-

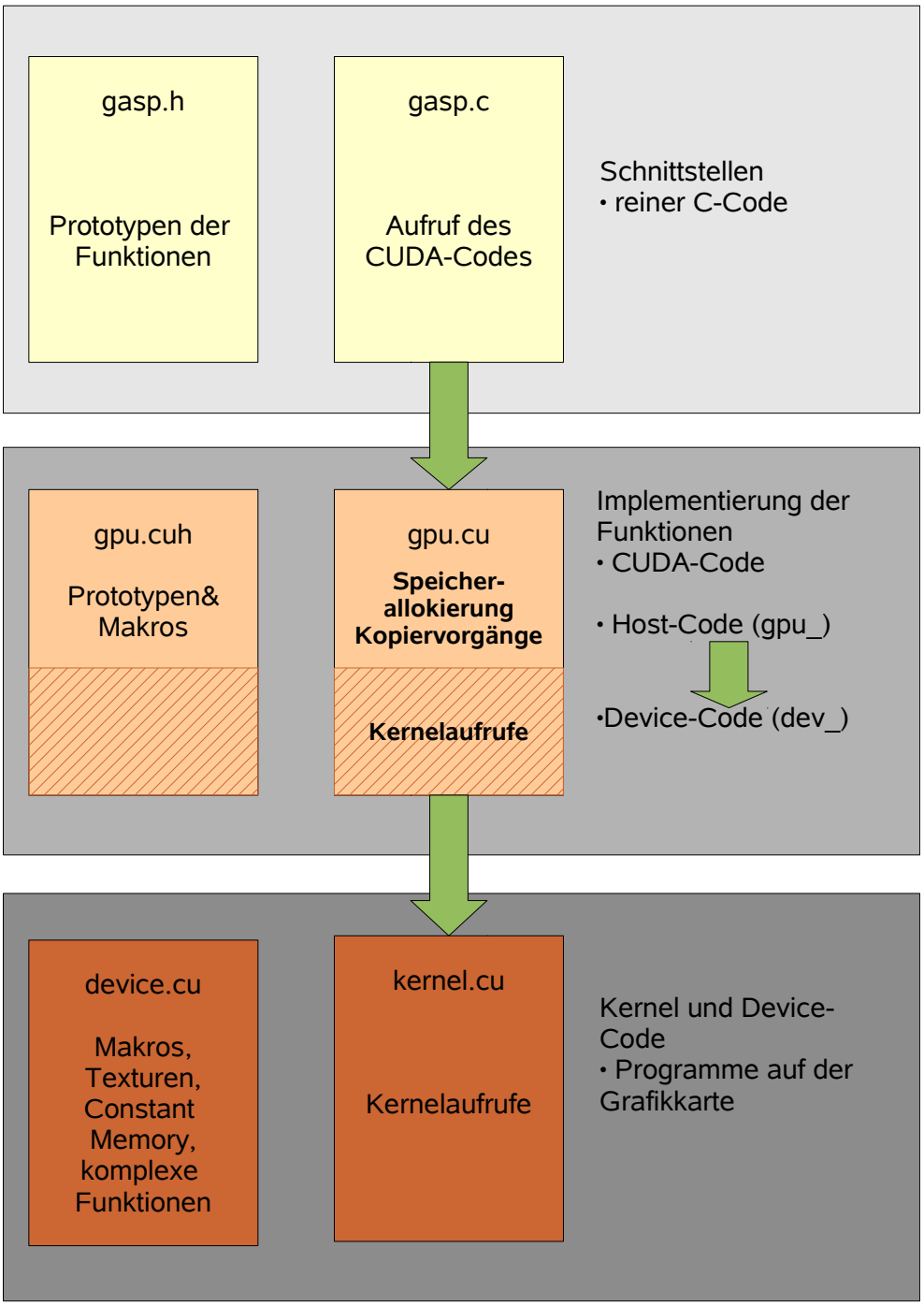


Abbildung 4.2: Darstellung der Programmstruktur

nel der Grafikkarte implementiert werden. Diese Funktionen müssen mit dem reservierten Wort `__global__` eingeleitet werden. Zum anderen `device.cu`, welches Device-Funktionen

enthält, wie beispielsweise Funktionen für komplexe Rechenoperationen, Variablen im Constant Memory und Texturen. Da hier nur Device-Code definiert wird, sind diese Dateien mit der Farbe Rot versehen.

Um die Bibliothek als Shared Object zu kompilieren, wird auch hier eine Zweiteilung vorgenommen. Zum einen wird der C-Code mit Hilfe des gcc-Compilers als Objekt-Datei kompiliert. Anschließend wird der CUDA-Code mit dem von NVidia bereitgestellten Compiler NVCC in eine Objektdatei umgewandelt. Im letzten Schritt werden beide Objekte mit dem gcc-Compiler zu einem Shared Object zusammengeführt. Das hierfür verwendete Makefile kann dem Anhang unter Quelltext 7 entnommen werden.

Zur Validierung der Richtigkeit der Implementierung wurde zunächst modular vorgegangen. Das heißt, jeder der Schritte wurde nach dem beschrieben Aufbau implementiert: `gpu_Schritt` ist die Funktion, die für die Speicherallokierung und die Kopiervorgänge sorgt, `dev_Schritt` übernimmt den eigentlichen Algorithmus für die Grafikkarte und ruft die nötigen Kernel auf. Nach außen wird die entsprechende Funktionen `gasp_Schritt` zur Verfügung gestellt.

Um nun die Verarbeitung der Radarverarbeitung effizienter zu gestalten, sollen die einzelnen Schritte ohne Umwege nacheinander auf der GPU bearbeitet werden. Mit Hilfe der eben erklärten Struktur ist es sehr einfach möglich, dies umzusetzen. In den Funktionen `dev_` wurde der Algorithmus für jedes Modul implementiert und bietet sozusagen den Kern des Algorithmus'. Die Funktionsparameter nehmen nur Parameter auf, die sich bereits auf der Grafikkarte befinden. Somit muss lediglich eine `gpu_`-Funktion implementiert werden, die die nötigen Aufrufe und Allokierungen tätigt. Quelltext 4 gibt die grobe Struktur, wie diese Funktion aufgebaut sein kann.

Damit also alle Funktionen nacheinander aufgerufen und eventuelle Zwischenrechnungen eingefügt werden können, wird diese gemeinsame Funktion sozusagen als Verwaltungsmodul verwendet. Um den Nutzern der Bibliothek nach außen hin eine Schnittstelle zu bieten, wird nach dem gleichen Schema wie bisher eine C-Funktion mit dem Präfix `gasp_` zur Verfügung gestellt. Auch das Shared Object für den IDL-Wrapper muss dementsprechend um diese Funktion erweitert werden.

4.2 Abschätzung der Performance

Auf Grundlage der in Abbildung 4.1 gegebenen Zeit für die Kernelausführung folgt eine Abschätzung, wie sich die Zeiten der Radarverarbeitung verhalten werden, wenn sie komplett und ohne Umwege auf der GPU durchgeführt werden würde.

```
1  void gpu_ecs(Complex* data, float *Parameter) {
2     Complex* devData;
3     ...
4
5     //Speicherallokierung
6     cudaMalloc( (void **) &devData,sizeof(Complex)*width*height);
7     ...
8
9     //Kopieren
10    cudaMemcpy( devData, (Complex*)data,
11                sizeof(Complex)*width*height,
12                cudaMemcpyHostToDevice);
13    ...
14
15    //Aufrufe
16    dev_fomoco(devData, ...);
17    dev_resample(devData, ...);
18    ...
19    dev_azCompr(devData, ...);
20
21    //Zurueckkopieren
22    cudaMemcpy( data, (char*)devData,
23             sizeof(Complex)*width*height,
24             cudaMemcpyDeviceToHost);
25    //Speicher wieder freigeben
26    cudaFree(devData);
27    ...
28
29    cudaThreadExit();
30 }
```

Quelltext 4: Struktur zur Zusammenführung der einzelnen Schritte auf die Grafikkarte

Bei einer Bildgröße von 8192×16384 (Datenmenge 1GB) und einschließlich der notwendigen Vektoren für die einzelnen Schritte fällt eine Datenmenge von 1,5GB an. Diese zu übertragen wird nach Abschätzung 500ms dauern. Durch Aufsummation der einzelnen Kernelzeiten wird eine Rechenezeit von 1898ms erreicht werden. Weiterhin muss mit einer einmaligen Latenz von 200ms bei jedem Funktionsaufruf und beim erstmaligen Starten des Shared Objects von etwa 2000ms gerechnet werden (siehe Kapitel 3.1). Das macht zusam-

men eine reine Rechenzeit von 4400ms. Hinzu können Datentypkonvertierungen kommen, die, wie in Kapitel 3.3.4 gezeigt, mehrere Sekunden kosten können.

Alles in allem ist bei der genannten Datenmenge mit der in den Tests benutzten Grafikkarte mit einer Gesamtrechenzeit auf der Grafikkarte von mindestens 5 Sekunden zu rechnen. Dem steht eine Aufzeichnungsdauer von 16 Sekunden gegenüber. Eine Echtzeitfähigkeit der hochauflösenden F-SAR Prozessierung ist damit gewährleistet.

5 Spezifikation einer geeigneten Grafikkarte für die F-SAR Echtzeitverarbeitung

An Bord des Flugzeuges soll während der Datenaufnahme eine Echtzeitprozessierung stattfinden. Damit dies in voller Auflösung möglich ist, soll die Hardware des Onboard-Prozessors durch eine Grafikkartenlösung erweitert bzw. ersetzt werden. Hierzu gibt es neben den physikalischen Beschränkungen auch die Anforderung an genügend Rechenkraft, um innerhalb der Aufnahmezeit die Daten zu verarbeiten.

Bevor auf eine Spezifikation eingegangen werden kann, soll zunächst beschrieben werden, wie die Echtzeitprozessierung während einer Überfliegung für einen Kanal stattfinden soll. Der Datenstrom von einem Kanal wird in einen Puffer abgelegt. Dieser Puffer hält die Daten für 10 bzw. 20s und leitet sie anschließend an die Grafikkarte zur Verarbeitung weiter. Durch den freigewordenen Speicher kann der Puffer mit den neu ankommenden Daten gefüllt werden. Währenddessen muss die Grafikkarte das Datenpaket vollständig verarbeiten, bevor der Puffer die Daten erneut an die GPU überträgt.

Um das Datenaufkommen für einen Kanal pro Sekunde zu berechnen, ergibt sich folgende Gleichung:

$$N_{Byte/s} = 16384 Samples_{rg} \cdot 8 \text{ Byte } (\text{ 1 Complex Float Sample}) \cdot PRF \text{ [Hz]} \tag{25}$$

Hierbei wird eine Auflösung von 0,5m in der Schrägentfernung angenommen. Mit einer Pulswiederholungsfrequenz von 1008Hz ergeben sich in 10s 1,23GB Daten und in 20s 2,46GB Daten. Zuzüglich Vektoren und anderen auf der Grafikkarte gespeicherten Variablen wird für diese Art der Verarbeitung insgesamt ca. 2GB bzw. 4GB benötigt

Damit die Prozessierung stattfinden kann, muss eine entsprechende Umgebung geschaffen werden. Dazu gehören neben einem installierten Betriebssystem wie Windows oder Linux auch die entsprechenden Treiber der Grafikkarte.

Speicherkapazität	>4GB
Compute Capability	>1.3
Betriebssystem	Windows oder Linux
Hersteller	NVidia
Programmierarchitektur	CUDA
Umgebungstemperatur	$\leq 45°C$

Tabelle 23: Mindestanforderungen an die Grafikkarte

Speicher	DDR5
Speicherbandbreite	$\geq$120GB/s
Architektur	Fermi
Anzahl Prozessore	≥ 300

Tabelle 24: optimale Anforderungen an die Grafikkarte

Da die entwickelte Software zur Beschleunigung der Radarverarbeitung in CUDA Version 3.2 geschrieben ist, schränkt sich der Bereich des Hardwareherstellers auf NVidia ein. Weiterhin ist es wichtig, dass einige Operationen mit doppelter Gleitkommagenauigkeit durchgeführt werden. Das erfordert eine *Compute Capability* von wenigstens 1.3. Hierbei wäre die neue Fermi-Architektur von NVidia die optimale Wahl.

Tabelle 23 fasst alle notwendigen Anforderungen an die Grafikkarte zusammen, während Tabelle 24 optionale Anforderungen aufführt.

Die Echtzeitprozessierung an Bord des Flugzeuges erfordert neben der doppelten Gleitkommagenauigkeit auch einen für Grafikkarten großen Speicher. Somit schränkt sich das breite Angebot der NVidia Grafikkarten erheblich ein, da die gängigen Consumer Produkte diesen Anforderungen nicht gerecht werden können. Stattdessen fallen zwei größere Produktfamilien in die engere Auswahl: Tesla und Quadro.

Quadro-Grafikkarten schlagen die Brücke zwischen anspruchsvollen Arbeitsplatzanforderungen und rechenintensiven, komplexen Anwendungen. Es existiert die ältere FX Quadro-Reihe, die dementsprechend nicht mit der Fermi-Architektur ausgestattet ist. Die leistungsfähigste Grafikkarte dieser Reihe besitzt einen DDR3 Speicher von 4GB und würde den minimalen Anforderungen entsprechen. Die Nachfolgemodelle *Quadro 5000* und *Quadro 6000* besitzen bereits eine Fermi-Architektur. Die genauen Werte können der Tabelle 25 entnommen werden.

Grafikkarten der Quadro-Familie entsprechen den Standards der PCI Express Karte. Sie beanspruchen zwei Slots und werden vor allem in Desktop-Rechnern eingesetzt. Die Grafikkarten der Tesla-Reihe hingegen gibt es in drei verschiedenen Varianten. Zum einen wie die Quadro als Desktop-Lösung mit Belegung von zwei PCI Express Slots. Weiterhin als eingebautes Modul in einem vorkonfigurierten Computer-System. Dies kann ein Cluster, Server oder standardisiertes Rack sein. Die Grafikkarte übernimmt dabei nur Re-

Name	**Leistung [W]**	**#Prozessoren**	**SP/DP [GFLOP/s]**	**Speicher [GB]**	**Speicherbandbreite [GB/s]**
Quadro 5000	152	-	352	2,5	120
Quadro 6000	204	448	1030/515	6	144
Tesla 2050	238	448	1030/515	3	144
Tesla 2070	238	448	1030/515	6	144

Tabelle 25: Spezifikationen der in Frage kommenden Grafikkarten (SP: Single Precision, DP: Double Precision)

chenaufgaben und hat dementsprechend keinen Videoausgang. Und zu guter letzt gibt es die Möglichkeit, 4 oder mehr Teslas zu einem 19"1U System zusammenzufassen, welches in standardisierte Racks passt. Da alle Varianten die gleichen Tesla-Prozessoren besitzen, sei hier nur auf die Spezifikation der Teslas eingegangen und nicht auf die Einbindung in das System.

Teslas der vorangegangen Generation haben nur geringe Rechenkraft in Bezug auf doppelte Gleitkommagenauigkeit und werden aus diesem Grund nicht näher betrachtet. Teslas mit der neuen Fermi-Architektur gibt es in 3GB und 6GB DDR5 Speicher Ausführung. Die näheren Spezifikationen sind der Tabelle 25 zu entnehmen.

Anhand der Tabelle 25 wird deutlich, dass vor allem die Teslas und Quadro 6000 für die Echtzeitprozessierung verwendet werden können. Im Falle der Tesla 2050 sollte jedoch auf einen Zusammenschluss von wenigsten zwei dieser Modelle zurückgegriffen werden, um auch bei Änderungen des Puffers nicht zu schnell an die Grenzen zu stoßen.

Damit diese in das bestehende System an Bord des Flugzeuges integriert werden können, müssen sie weiteren Begrenzungen und Anforderungen entsprechen:

1. Schnittstelle zu Compact PCI[4]
2. starke Belastungen (Erschütterungen, Temperatur)
3. Platzbegrenzung im Flugzeugeinbau: derzeit $233{,}35mm \times 160mm$ (Compact PCI konform)
4. Flughöhe von 6000m

[4]Weitere Standards zu Compact PCI können unter Quelle [PCI95] gefunden werden

Da NVidia Grafikkarten hauptsächlich für den Consumer Bereich, sowie für wissenschaftliche Anwendungen und Rechenzentren entwickelt hat, gibt es noch keine Compact PCI Lösung. Bei einer steigenden Bedeutung von Grafikkarten im industriellen Bereich, könnte in absehbarer Zeit ein Compact PCI Express kompatibles Grafikkartensystem entwickelt werden.

Zur Umgehung des Problems wäre es möglich, ein Compact PCI Modul zu installieren, welches einen Adapter zwischen einer PCI Express Karte und der Compact PCI Schnittstelle bietet. Jedoch beginnen hier die weitaus kritischeren Inkompatibilitäten.

Wie der Quelle [EKF] zu entnehmen ist, können PCI Express Karten mit den Maßen $139{,}5mm \times 71mm$ auf ein entsprechendes Modul gesteckt werden. PCI Express konforme Grafikkarten von NVidia besitzen jedoch die Ausmaße $111{,}2mm \times 247{,}7mm$ und überschreiten damit weit den zur Verfügung stehenden Platz. Weiterhin ist zu erkennen, dass eine Grafikkarte mit diesen Formfaktoren nicht als Ersatz für ein Compact PCI-Modul passt. Und da diese Maße sowohl für den derzeitigen Onboard Prozessor feststehen, als auch für die vorhandenen Grafikkarten, kann die Integrierung der Hardware auf diese Weise nicht erfolgen.

Selbst unter der Bedingung, dass eine Lösung für dieses Problem gefunden wird, können die zur Verfügung stehenden Grafikkarten nicht in einer Höhe von 6000m betrieben werden. In den Spezifikationen zur Tesla S2070 in Quelle [NVi10b] konnten die entsprechenden Angaben gefunden werden. Mit einer maximalen Höhe von 1500m während des Betriebs sind die Anforderungen nicht ohne weitere Maßnahmen erfüllbar.

Damit wird klar, dass die derzeitigen Produkte von NVidia nicht für die Onboard-Prozessierung geeignet sind. Ohne Anpassungen an das bestehende System und weiteren Vorkehrungen kann keine geeignete Grafikkarte gefunden werden. Zwar sind GPUs mit ausreichender Rechenkraft vorhanden, sie können jedoch kurzfristig nicht in der Einsatzumgebung verwendet werden, sondern lediglich in der Offlineverarbeitung in einer Serverumgebung. Die Entwicklung auf den Grafikkartenmarkt sollte demnach weiter verfolgt werden, da es in 1 bis 2 Jahren voraussichtlich geeignetere Lösungsansätze geben wird.

6 Fazit

Alle geforderten Aufgabenstellungen der Bachelorarbeit wurden erfolgreich bearbeitet. Die wichtigen Details zur SAR-Prozessierung wurden eingangs schnell erlernt. Zur Prozessierung von Radardaten stehen am Institut für Hochfrequenztechnik und Radarsystem des

DLRs zwei Implementierungen für das F-SAR System zur Verfügung: ein IDL- und ein C-Prozessor. Für beide sollte der Extended Chirp Scaling Algorithmus für die Berechnung auf einer Grafikkarte implementiert werden. Hierzu wurden verschiedene Methoden zur Integration in die bestehende Software analysiert und schließlich eine ausgewählt. Zur Vermeidung von doppeltem Wartungsaufwand ist die Implementierung einer dynamischen Bibliothek mit Hilfe von Shared Objects die bevorzugte Variante. Shared Objects erlauben eine Nutzung des gleichen GPU-Quellcodes sowohl im IDL- als auch im C-Prozessor.

Im Anschluss daran erfolgte die Integration der Bibliothek in die beiden Prozessierungssoftwares. Damit war es möglich, jeden Prozessierungsschritt einzeln zu implementieren und hinsichtlich Berechnungsgenauigkeit und -laufzeit zu testen.

Die Tests zur Berechnungsgenauigkeit haben gezeigt, dass alle zu implementierenden Schritte mit einer geringen Fehlertoleranz von 0,1 bis 1% funktionieren. Jedoch hat sich auch herausgestellt, dass es beispielsweise bei der zweiten Bewegungskompensation zu einem Phasenoffset kommt, der noch nicht korrigiert werden konnte. Außerdem stellte sich heraus, dass der in Quelle [Kü10] implementierte Resampling-Schritt nicht die optimale Genauigkeit liefert und darüber hinaus für den IDL-Prozessor noch integriert werden muss.

Die Tests zur Laufzeit zeigen eine bedeutende Beschleunigung der Rechenzeit einzelner Module um den Faktor 10 bis 70. Dieser Trend konnte jedoch nicht in gleichem Maße auf die Gesamtrechenzeit übertragen werden (Faktor $\leq$ 4). Deshalb ist die Anpassung des Codes für die gesamte Radarverarbeitung auf der Grafikkarte empfehlenswert. Durch die ausführlichen Zeitaufnahmen, war es aber möglich, eine Vorabschätzung der zu erwartenden Rechenzeit auf der Grafikkarte anzustellen. Es konnte gezeigt werden, dass eine Verarbeitung in Echtzeit mit derzeitigen GPU-Karten durchaus möglich ist.

Jede durchgeführte Zeitaufnahme wurde durch höhere Schwankungen der Auslastung beeinflusst. Dadurch kam es des öfteren zu Problemen, da durch andere Benutzer die Grafikkarte blockiert wurde oder es durch vollen Arbeitsspeicher zu erheblichen Latenzen kam. Trotz Durchschnittsbildung von 100 Durchläufen kam es bei den Benchmarks je nach Auslastung zu größeren Abweichungen.

Zur Findung von Bottlenecks und Identifizierung von Opimierungsbedarf verhalfen die vielen durchgeführten Benchmarks und Tests. Ein Großteil der Optimierungen konnte bereits umgesetzt werden, weitere sind in anknüpfenden Arbeiten möglich. Insbesondere sind folgende Punkte zu vervollständigen:

- Laufzeitoptimierung der zweiten Bewegungskompensation.

- Identifizierung und Korrektur des Phasenoffsets.
- Bereitstellung des Resamplings für den IDL-Prozessor (Erweiterung des Shared Objects der IDL-Schnittstelle).
- Untersuchungen hinsichtlich Qualität und Rechenzeit zu verschiedenen Methoden der Interpolation. Darunter fallen lineare, kubische und Knab-Interpolation
- Darauf aufbauend, Implementierung des Resamplings für eine verbesserte zweite Bewegungskompensation.
- Zusammenführung aller Algorithmenschritte in einem Kernel zur Verkürzung der Gesamtrechenzeit auf der Grafikkarte.
- Untersuchungen zur generellen Perfo.rmancesteigerung

Die Echtzeitprozessierung von F-SAR Daten stellt spezielle Anforderungen an die GPU, die ebenfalls im Rahmen dieser Arbeit ermittelt und evaluiert wurden. Dabei stellte sich heraus, dass es mit den Einschränkungen der aktuellen F-SAR Hardware an Bord des Flugzeuges nicht möglich ist, eine passende und leistungsstarke Grafikkarte zu spezifizieren. Einerseits besitzen die in Frage kommenden Grafikkarten nicht die nötigen Schnittstellen und Formfaktoren (in der Länge sind sie 1cm zu lang) und andererseits sind sie lediglich bis zu einer Flughöhe von 1,5km spezifiziert. Somit beschränkt sich die Nutzung der Grafikkarte derzeit auf die Offlineprozessierung auf einem Server oder Desktop. Langfristig gesehen können jedoch Lösungsansätze gefunden werden, die auf Weiterentwicklungen von Grafikkarte speziell für den industriellen Bereich bauen.

Alles in allem wurden alle Schwerpunkte der Bachelorarbeit erfolgreich abgearbeitet. Eine vollständige Zusammenstellung der Testergebnisse ist im Anhang dieser Arbeit zu finden. Auf Grundlage dieser Tests kann die Vervollständigung und Optimierung der F-SAR Radarverarbeitung auf GPU in kürzester Zeit erfolgen.

Abkürzungsverzeichnis

az	Azimut
azCompr	Azimutkomprimierung
Chirp	Coherent Integration of Radar Pulses
CPU	Central Processing Unit, eng. Prozessor
CUDA	Compute Unified Device Architecture
DLM	dynamic loadable module, eng. dynamisch ladbares Modul
DLR	Deutsches Zentrum für Luft- und Raumfahrt
FFT	Fast Fourier Transformation
FLOPS	floating point operations, eng. Operationen mit Gleitkommazahlen
fomoco	First Order Motion Compensation (Bewegungskompensation erster Ordnung
F-SAR	Flugzeug-SAR
GASP	GPU accelerated SAR processing, eng. hecheln
GCC	GNU C Compiler
GNU	GNU is not Unix, eng. GNU ist nicht Unix
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit, eng. Grafikkarte
GPUlib	GPU Library, Bibliothek für IDL
IDL	Interactive Data Language
iFFT	inverse Fast Fourier Transformation

NVCC	NVidia CUDA Compiler
PCI	Peripheral Component Interconnect
phCorr	Phasenkorrektur
PRF	Pulse Repetition Frequency
Radar	Radio Aircraft Detection and Ranging
RAM	Random Access Memory, eng. Hauptspeicher
rc_rcmc	Range Compression and Range Cell Migration Correction (Entfernungskomprimierung und Zielentfernungskorrektur)
rg	Range
SAR	Synthetic Aperture Radar, eng. Radar mit synthetischer Apertur
somoco	Second Order Motion Compensation (Bewegungskompensation zweiter Ordnung
ULP	unit in last place
Vabene	Verkehrmanagement bei Großeinsätzen und Katastrophen

Literatur

[Aul05] Thomas Aulinger. Vergleich von Höhenmodellen aus InSAR- und Lidar-Daten über einem Naturwald im Nationalpark Bayerischer Wald. Master's thesis, Fachhochschule München, 2004/2005.

[BF05] M. Blom and P. Follo. VHF SAR image formation implemented on a GPU. In *Geoscience and Remote Sensing Symposium, 2005. IGARSS'05. Proceedings. 2005 IEEE International*, volume 5, page 3352–3356. IEEE, 2005.

[EKF] EKF. *SA1-FUSION Product Specification*.

[Far09] Rob Farber. CUDA, Supercomputing for the Masses: Part 13, Using Textures memory in CUDA. `http://drdobbs.com/article/print?articleId=218100902&siteSectionName=`, Juni 2009. [Online; Stand 10. September 2011].

[Fis02] Christian Fischer. Studie zur Volumen- und Oberflächenstreuung mit interferometrischen Multi-Baseline SAR-Daten. Master's thesis, Technische Universität München, 2002.

[Gal10] Michael Galloy. GPULib with IDL 8.0 slide presentation. on VISualize 2010, 2010. `http://www.txcorp.com/pdf/GPULib/documentation/gpulib-with-idl80.pdf`.

[HFB+09] Timothy D. R. Hartley, Ahmed Fasih, Charles A. Berdanier, Füsun Özgüner, and Ümit V. Çatalyürek. Investigating the use of GPU-accelerated nodes for SAR image formation. In *CLUSTER*, pages 1–8, 2009.

[ITT09] ITT Visual Informaiton Solutions. External Development Guide. Technical report, ITT Visual Information Solutions, 2009.

[KH00] Helmut Klausing and Wolfgang Hollp. *Radar mit realer und synthetischer Apertur*. Oldenburg Verlag, 2000.

[Khr11] Khronos Group. OpenCL Overview. `http://www.khronos.org/opencl/`, 2011. [Online; Stand 8. Juli 2011].

[Kü10] Maren Künemund. Beschleunigung der Verarbeitungsgeschwindigkeit flugzeuggestützter SAR Daten durch Auslagerung rechenintensiver Verarbeitungsschritte

auf eine Grafikkarte. Technical report, Institut für Hochfrequenztechnik und Radarsysteme, SAR-Technologie, August 2010.

[LKNK07] Martin Lambers, Andreas Kolb, Holger Nies, and Marc Kalkuhl. GPU-based framework for interactive visualization of SAR data. In *Geoscience and Remote Sensing Symposium, 2007. IGARSS 2007. IEEE International* , pages 4076–4079. IEEE, 2007.

[LWLY09] Bin Liu, Kaizhi Wang, Xingzhao Liu, and Wenxian Yu. An Efficient SAR Processor Based on GPU via CUDA. *2009 2nd International Congress on Image and Signal Processing*, pages 1–5, 2009.

[LWLY10] Bin Lio, Kaizhi Wang, Xingzhao Liu, and Wenxian Yu. Range Cell Migration Correction using texture mapping on GPU. In *Signal Processing (ICSP), 2010 IEEE 10th International Conference on* , pages 2172–2175. Dept. of Electron. Eng, Oktober 2010.

[LWY09] Yuan Lu, K. Wang, and Wenxian Yu. A GPU based real-time SAR simulation for complex scenes. In *Radar Conference - Suveillance for a Safer World. RADAR. International*, pages 1–4, Oktober 2009.

[Mar06] Luca Marotti. *Detection and Characterization of coherent Scatterers*. PhD thesis, Universita' Degli Studi di Napoli "Federico II"

, 2005/2006.

[Mes09] Peter Messmer. GPULib: GPU Computing in IDL/ENVI. on VISualize 2009, 2009. `http://www.txcorp.com/pdf/GPULib/documentation/VISualize09_GPULib.pdf`.

[MM] A. Moreira and J. Mittermayer. *Synthetic Aperture Radar - Basic Principles and Advanced Techniques*. Deutsches Zentrum für Luft- und Raumfahrt, Oberpfaffenhofen.

[MMS96] A. Moreira, J. Mittermayer, and R. Scheiber. Extended Chirp Scaling Algorithm for Air- and Spaceborne SAR Data Processing in Stripmap and ScanSAR Imaging Modes. *IEEE Transactions on Geoscience and Remote Sensing*, 34(5):1123–1136, September 1996.

[MNBC07] M. Migliaccio, F. Nunziata, F. Bruno, and F. Casu. Knab Sampling Window for InSAR Data Interpolation. *Geoscience and Remote Sensing Letters, IEEE*, 4(3):397 –400, july 2007.

[Nvi] Nvidia Corporation. Grafikprozessor Tesla C2050/C2070. `http://www.nvidia.de/object/product_tesla_C2050_C2070_de.html`. [Online; Stand 11. Juli 2011].

[NVi10a] NVidia Corporation. *NVIDIA CUDA C Programming Guide*, 3.2 edition, November 2010.

[NVi10b] NVidia Corporation. *TESLA 1U GPU COMPUTING SYSTEM, Product Specification*, Juni 2010.

[Nvi11a] Nvidia Corporation. Developer Zone — OpenCL. `http://www.nvidia.de/object/cuda_opencl_new_de.html`, 2011. [Online; Stand 8. Juli 2011].

[NVi11b] NVidia Corporation. *NVIDIA CUDA C Programming Guide*, 4.0 edition, Mai 2011.

[PCI95] PCI Industrial Computers Manufacturers Group. *Compact PCI Short Form Specification*, November 1995.

[PW10] Jimmy Pettersson and Ian Wainwright. Radar Signal Processing with Graphics Processors (GPUS), 2010.

[rad91] *Synthetic Aperture Radar: Systems and Signal Processing*. John Wiley & Sons, 1991.

[RM09] Greg Ruetsch and Paulios Micikevicius. Optimizing Matrix Transpose in CUDA. Technical report, Nvidia Corporation, Januar 2009.

[RSB] Gary Rubin, Earl V. Sager, and David H. Berge. GPU Acceleration of SAR/ISAR Imaging Algorithms. `http://www.accelereyes.com/content/collateral/GPUAcceleratedISAR_ver2.pdf`. [Online; Stand 13. September 2011].

A Quelltext

```
1  float eT, *idevData, *odevData;
2  size_t pitch;
3  gpuAssert(cudaMallocPitch((void**) &idevData, &pitch, width*sizeof(float)
       , height));
4  gpuAssert(cudaMalloc((void**) &odevData, 16*width*sizeof(float)*height));
5  gpuAssert(cudaMemcpy2D((void*)idevData, pitch,(void*)data, sizeof(float)*
       width, sizeof(float)*width, height, cudaMemcpyHostToDevice));
6
7  //set textures
8  cudaChannelFormatDesc cd = cudaCreateChannelDesc<float>();
9  tex.normalized           = false;
10 tex.filterMode           = cudaFilterModeLinear;
11 tex.addressMode[0]       = cudaAddressModeClamp;
12 tex.addressMode[1]       = cudaAddressModeClamp;
13 gpuAssert(cudaBindTexture2D(0, &tex, (void*) idevData, &cd, width, height
       , pitch));
14
15 //kernellaunch
16 dim3 block = dim3(16,16,1);
17 dim3 grid  = dim3(width/16, height/16, 1);
18 interpol4x <<< grid, block >>> (odevData);
19 gpuAssert( cudaThreadSynchronize() );
20
21 gpuAssert(cudaMemcpy( out, odevData, sizeof(float)*width*16*height,
       cudaMemcpyDeviceToHost));
22
23 cudaFree(idevData);
24 cudaFree(odevData);
25 cudaUnbindTexture(tex);
26
27 cudaThreadSynchronize();
28 cudaThreadExit();
```

Quelltext 5: Interpolation in CUDA

```
1  gpuinit;
2  data_gpu =  gpuFltarr(width, height)
3  out_gpu =   gpuFltarr(width*4, height*4)
4  gpuPutArr, data, data_gpu
5
6  out_gpu = gpuCongrid(data_gpu,width*4, height*4, INTERP=1)
7
8  gpuGetArr, out_gpu, out
9
10 gpuFree, data_gpu
11 gpuFree, out_gpu
```

Quelltext 6: Interpolation in GPULib

```
LIBDIR = -L/usr/local/cuda/lib64
LIB = -lcufft -lcuda
INCDIR = -I/home/kuen_ma/NVIDIA_GPU_Computing_SDK/C/common/inc -I/home/
    kuen_ma/Programming/CI/code
INC =
FLAGS = -shared -Xcompiler -fPIC -arch sm_20
#-arch sm_13 for older GPU with compute-capability 2
#arch sm_20 for new Fermi-Architecture

NVCC = nvcc

all:    libgasp.so

gasp.o: gasp.c
    gcc -c $< -o $@ -shared -fPIC
#add Flag -D__BENCH__ for benchmarking outputs

gpu.o: gpu.cu
    $(NVCC) -c $< -o $@ $(INCDIR) $(LIB) $(FLAGS)

libgasp.so: gasp.o gpu.o
    gcc $+ -o $@ $(INCDIR) $(LIBDIR) $(LIB) -shared -fPIC

clean:
    rm -f *.o *~ core a.out libgasp.so *.s
```

Quelltext 7: Makefile zur Erstellung der Gasp-Bibliothek

```
__device__ Complex ComplexMul(Complex a, Complex b) {
    Complex c;
    c.x = a.x * b.x - a.y * b.y;
    c.y = a.x * b.y + a.y * b.x;
    return c;
}
```

Quelltext 8: Device-Code: komplexe Multiplikation

```
1 __device__ Complex ComplexScale(Complex a, float s)
2 {
3     Complex c;
4     c.x = s * a.x;
5     c.y = s * a.y;
6     return c;
7 }
```

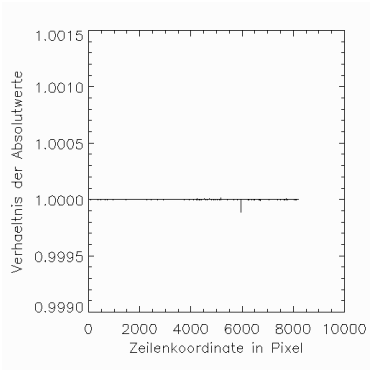
Quelltext 9: Device-Code: komplexe Skalierung

```
1 __device__ Complex ComplexExp(Complex a) {
2    float value = expf(a.x);
3    Complex result;
4    result.x = value * cosf(a.y);
5    result.y = value * sinf(a.y);
6    return result;
7 }
```

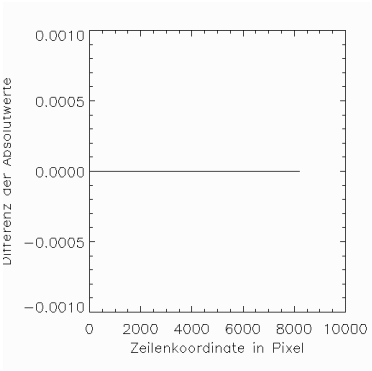
Quelltext 10: Device-Code: komplexe Exponentialfunktion

B Testergebnisse

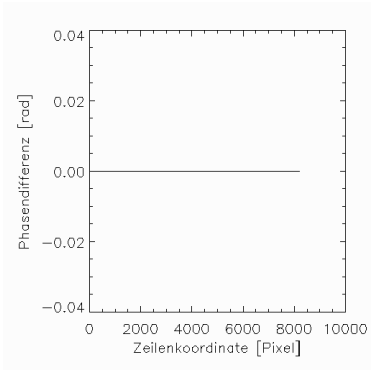
B.1 T0-1: Chirp Scaling



(a) Verhältnis der Absolutwerte

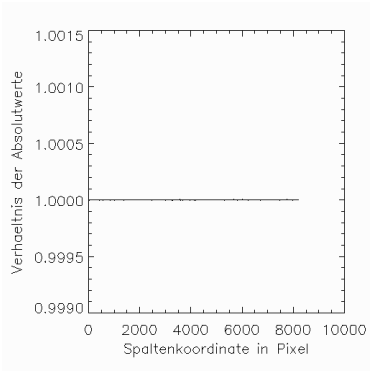


(b) Differenz der Absolutwerte

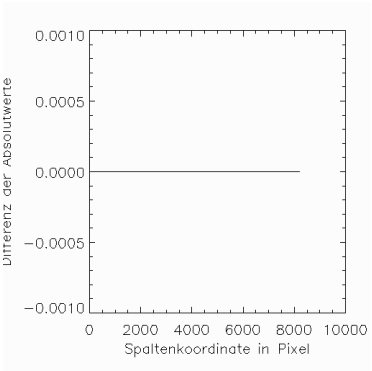


(c) Phasendifferenz

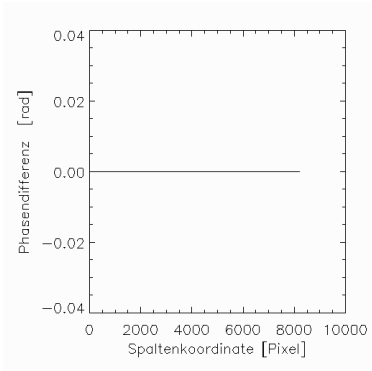
Abbildung B.1: F0-1, Spalten



(a) Verhältnis der Absolutwerte

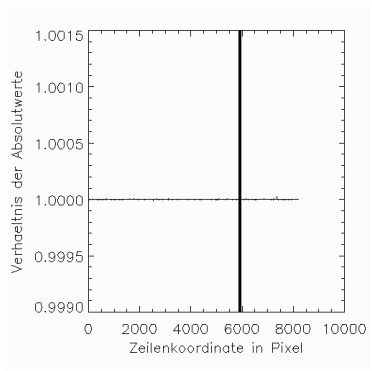


(b) Differenz der Absolutwerte

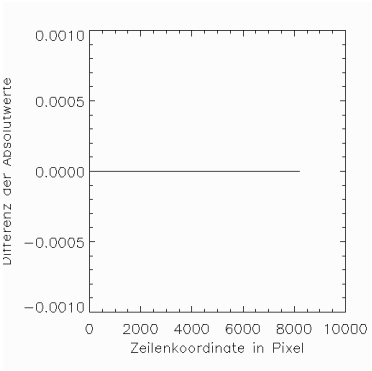


(c) Phasendifferenz

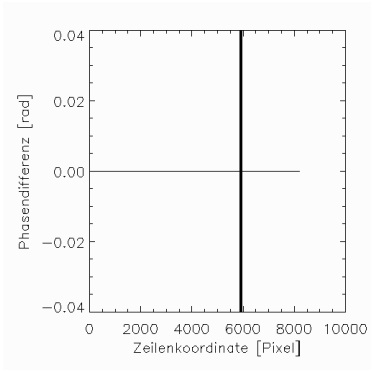
Abbildung B.2: F0-1, Zeilen



(a) Verhältnis der Absolutwerte

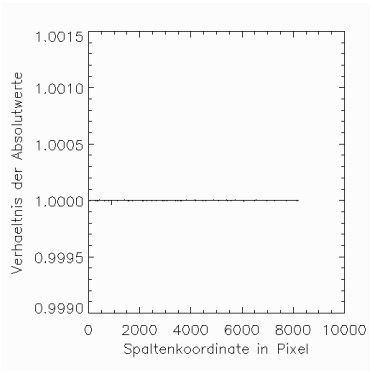


(b) Differenz der Absolutwerte

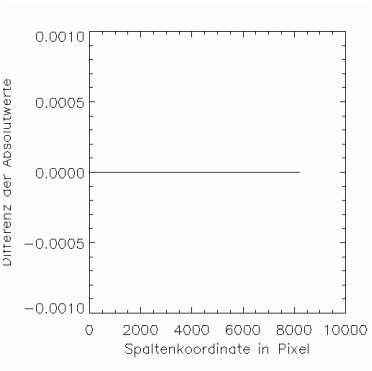


(c) Phasendifferenz

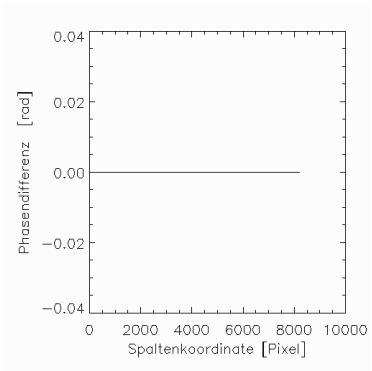
Abbildung B.3: I0-1, Spalten



(a) Verhältnis der Absolutwerte



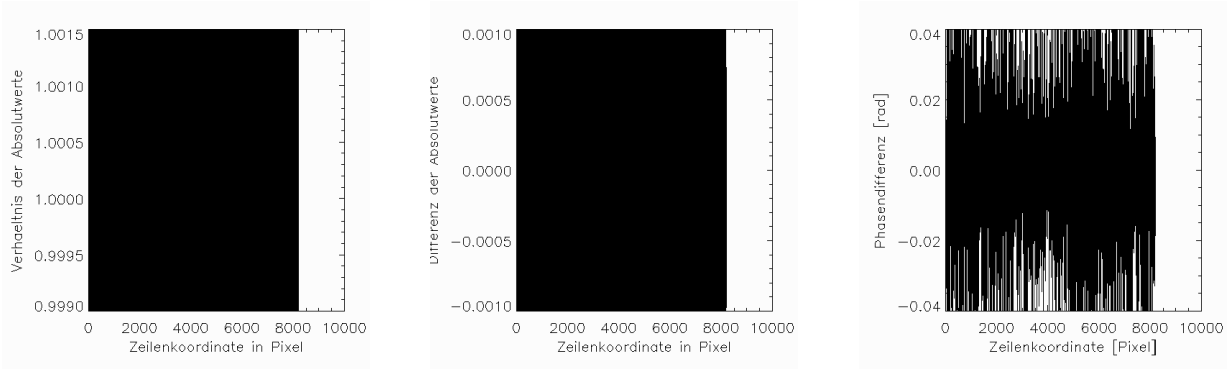
(b) Differenz der Absolutwerte



(c) Phasendifferenz

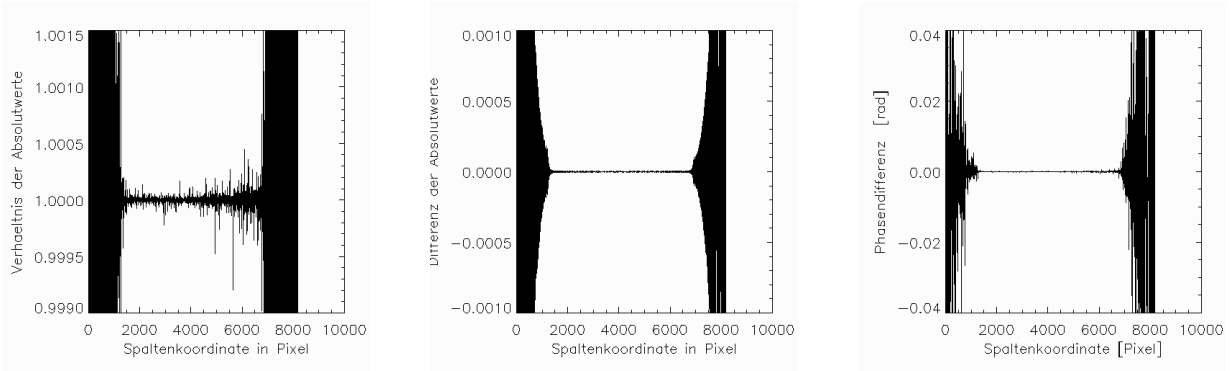
Abbildung B.4: I0-1, Zeilen

B.2 T0-2: Zielentfernungskorrektur und Entfernungskompression



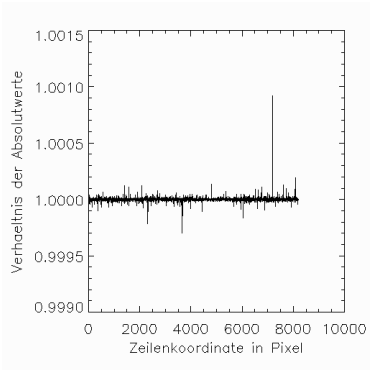
(a) Verhältnis der Absolutwerte (b) Differenz der Absolutwerte (c) Phasendifferenz

Abbildung B.5: F0-2, Spalten

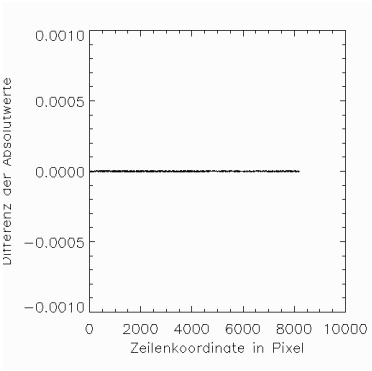


(a) Verhältnis der Absolutwerte (b) Differenz der Absolutwerte (c) Phasendifferenz

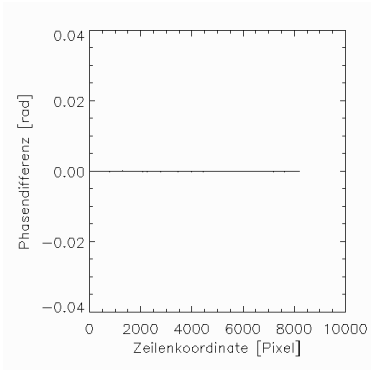
Abbildung B.6: F0-2, Zeilen



(a) Verhältnis der Absolutwerte

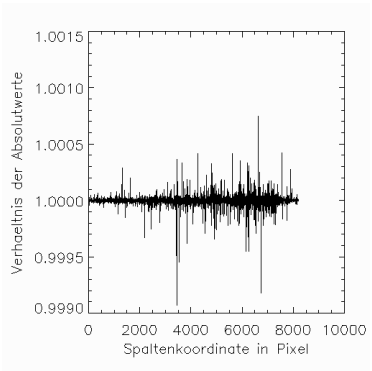


(b) Differenz der Absolutwerte

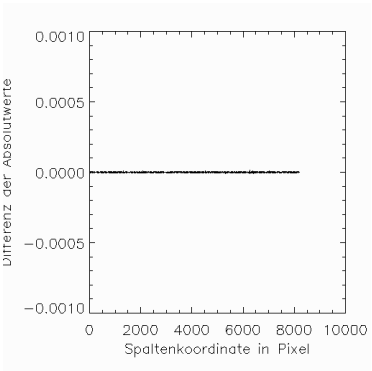


(c) Phasendifferenz

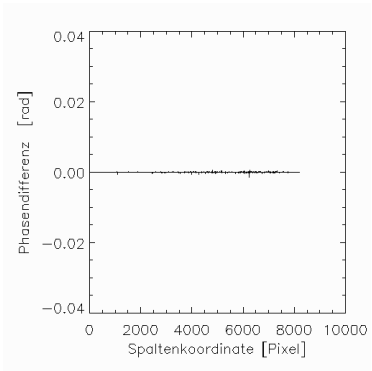
Abbildung B.7: I0-2, Spalten



(a) Verhältnis der Absolutwerte



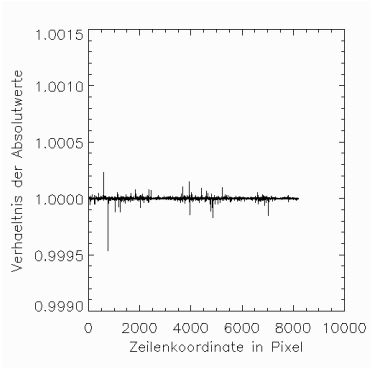
(b) Differenz der Absolutwerte



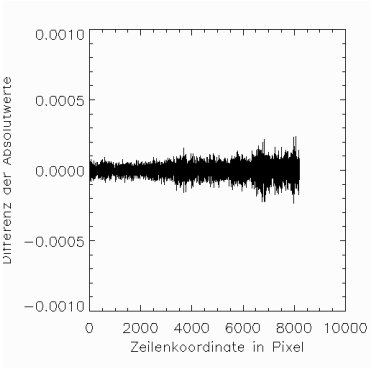
(c) Phasendifferenz

Abbildung B.8: I0-2, Zeilen

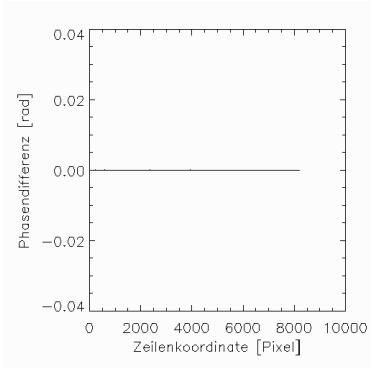
B.3 T0-3: Phasenkorrektur



(a) Verhältnis der Absolutwerte

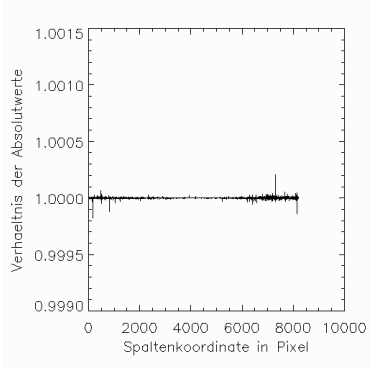


(b) Differenz der Absolutwerte

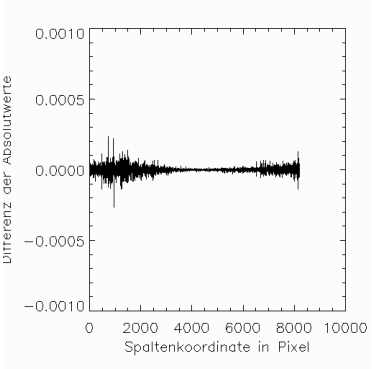


(c) Phasendifferenz

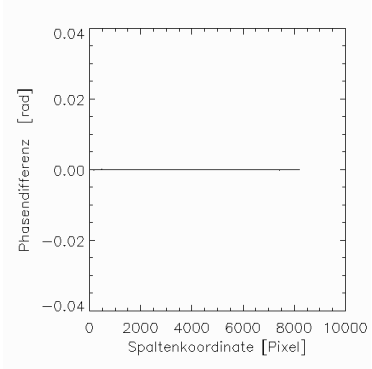
Abbildung B.9: F0-3, Spalten



(a) Verhältnis der Absolutwerte

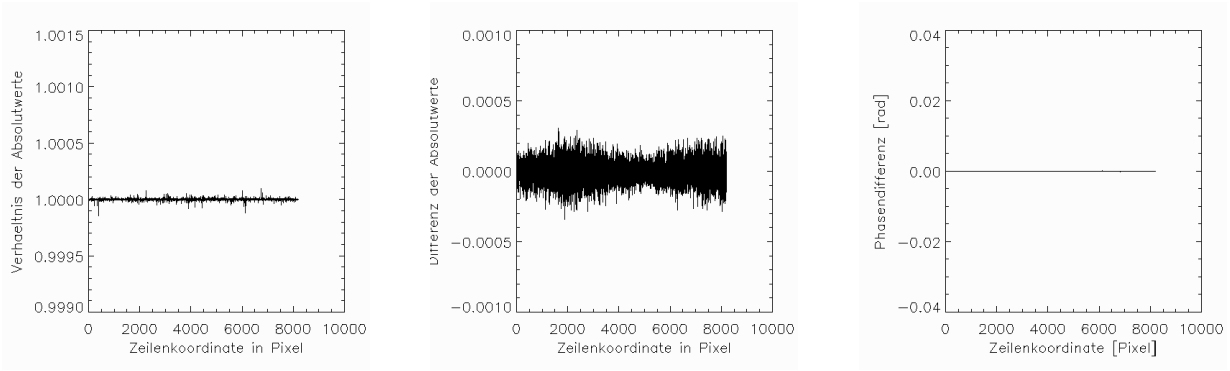


(b) Differenz der Absolutwerte



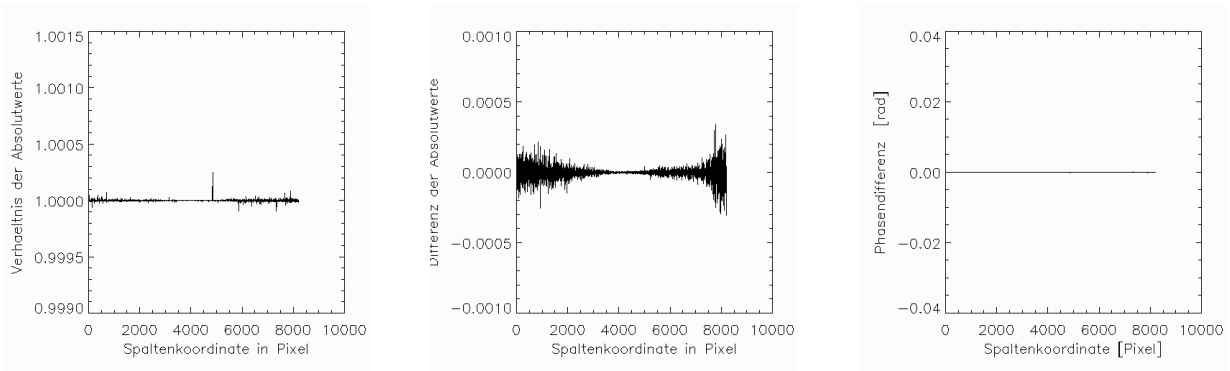
(c) Phasendifferenz

Abbildung B.10: F0-3, Zeilen



(a) Verhältnis der Absolutwerte (b) Differenz der Absolutwerte (c) Phasendifferenz

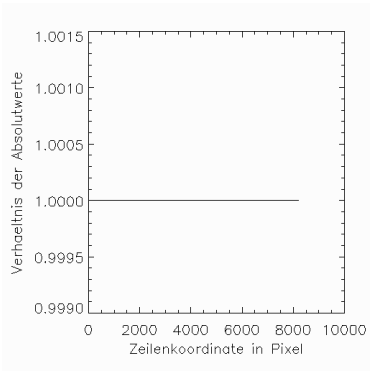
Abbildung B.11: I0-3, Spalten



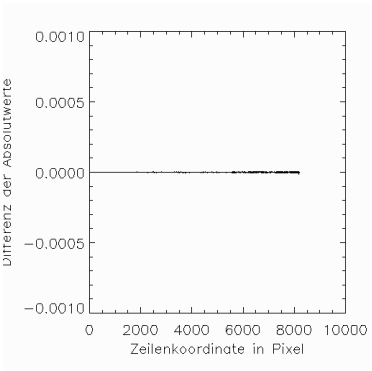
(a) Verhältnis der Absolutwerte (b) Differenz der Absolutwerte (c) Phasendifferenz

Abbildung B.12: I0-3, Zeilen

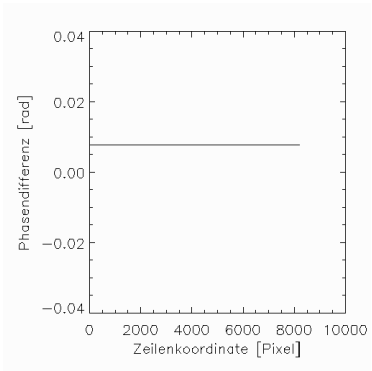
B.4 T0-4: Bewegungskompensation zweiter Ordnung



(a) Verhältnis der Absolutwerte

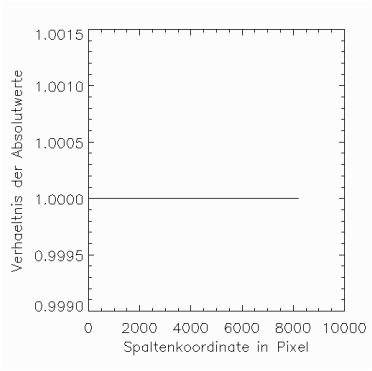


(b) Differenz der Absolutwerte

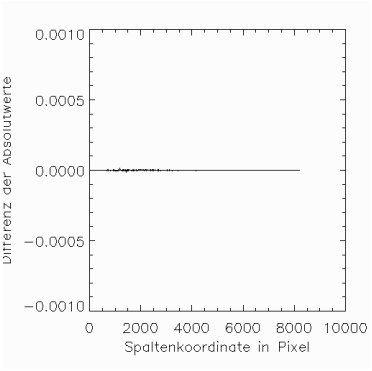


(c) Phasendifferenz

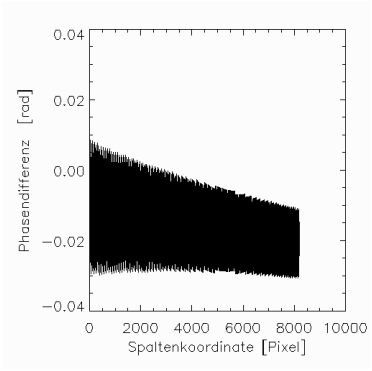
Abbildung B.13: F0-4, Spalten



(a) Verhältnis der Absolutwerte

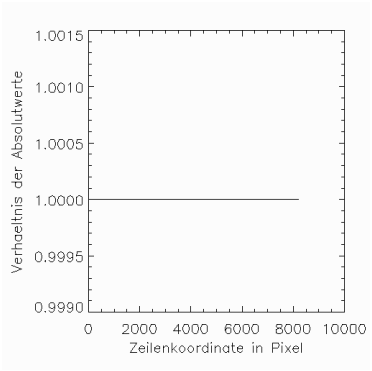


(b) Differenz der Absolutwerte

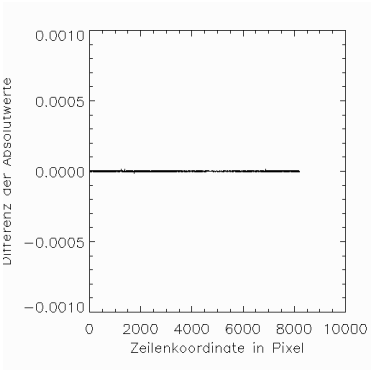


(c) Phasendifferenz

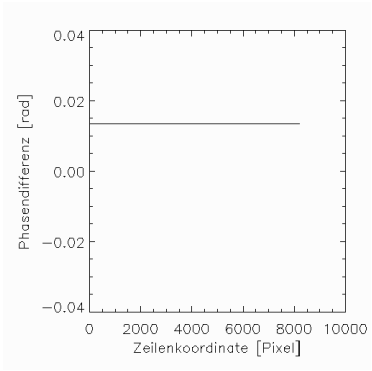
Abbildung B.14: F0-4, Zeilen



(a) Verhältnis der Absolutwerte

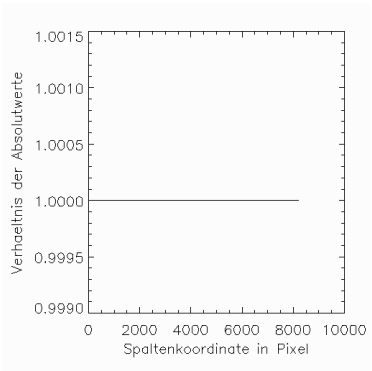


(b) Differenz der Absolutwerte

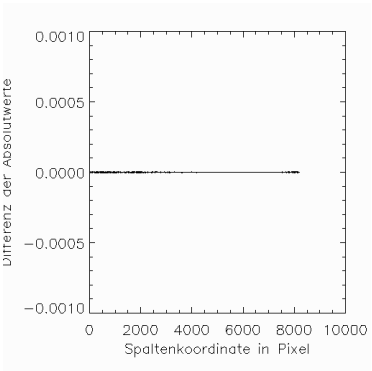


(c) Phasendifferenz

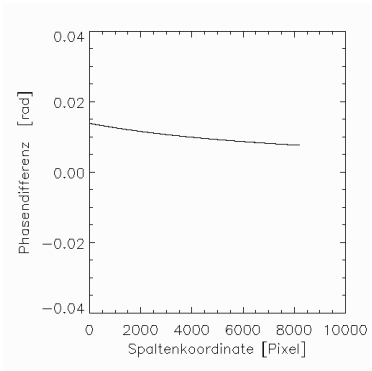
Abbildung B.15: I0-4, Spalten



(a) Verhältnis der Absolutwerte



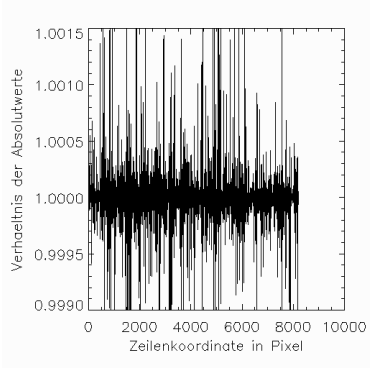
(b) Differenz der Absolutwerte



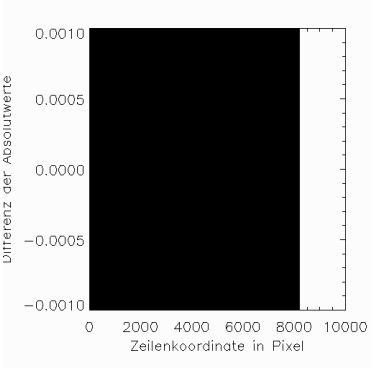
(c) Phasendifferenz

Abbildung B.16: I0-4, Zeilen

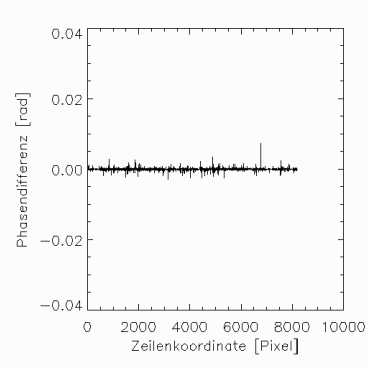
B.5 T0-5: Azimutkompression



(a) Verhältnis der Absolutwerte

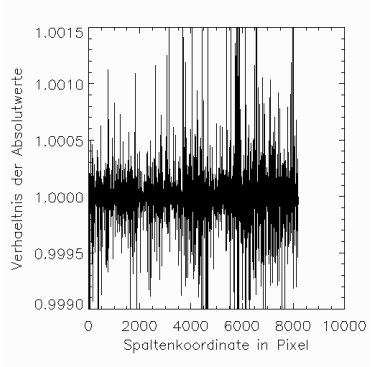


(b) Differenz der Absolutwerte

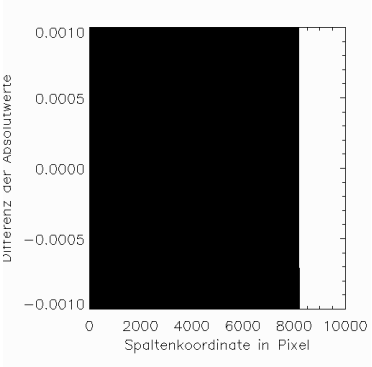


(c) Phasendifferenz

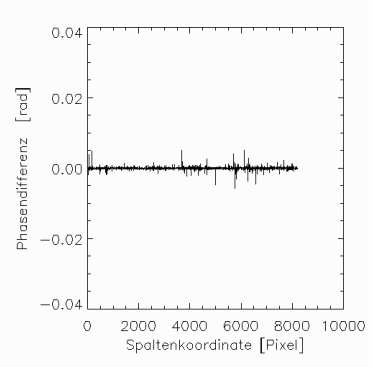
Abbildung B.17: F0-5, Spalten



(a) Verhältnis der Absolutwerte

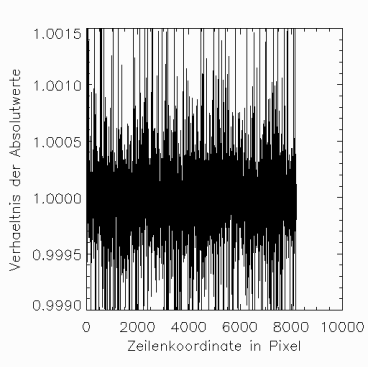


(b) Differenz der Absolutwerte

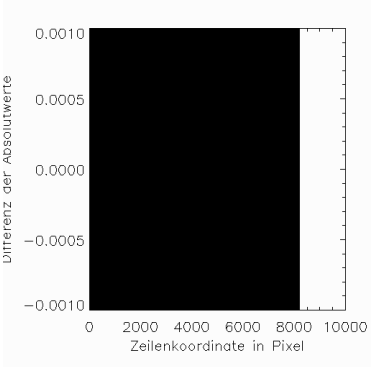


(c) Phasendifferenz

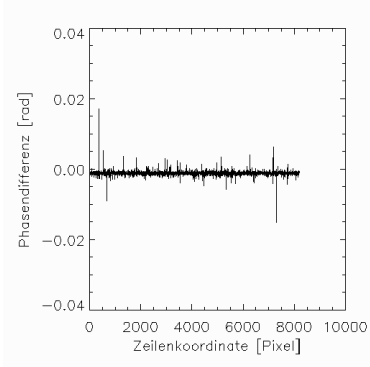
Abbildung B.18: F0-5, Zeilen



(a) Verhältnis der Absolutwerte

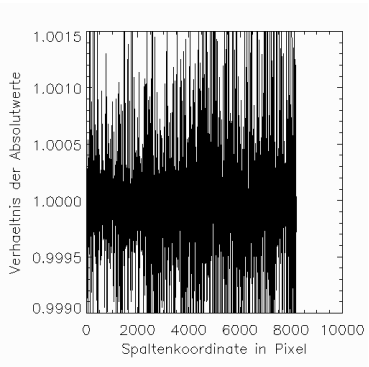


(b) Differenz der Absolutwerte

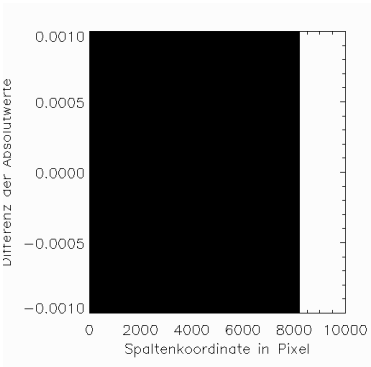


(c) Phasendifferenz

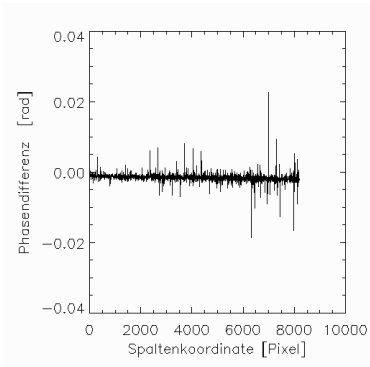
Abbildung B.19: I0-5, Spalten



(a) Verhältnis der Absolutwerte



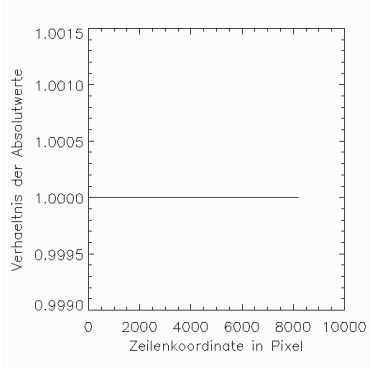
(b) Differenz der Absolutwerte



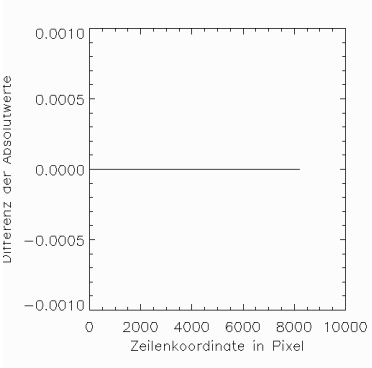
(c) Phasendifferenz

Abbildung B.20: I0-6, Zeilen

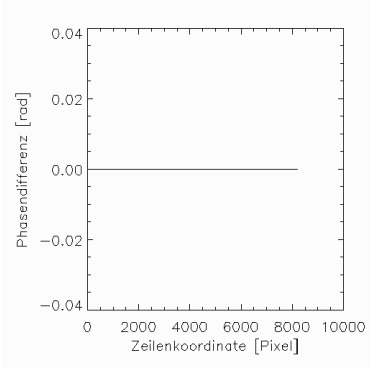
B.6 T1: Endergebnisse



(a) Verhältnis der Absolutwerte

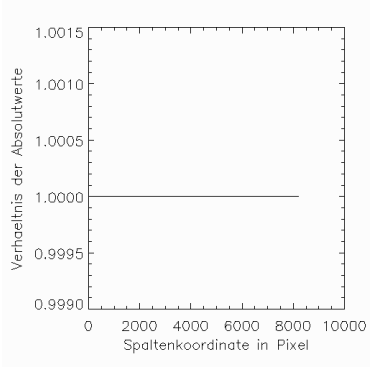


(b) Differenz der Absolutwerte

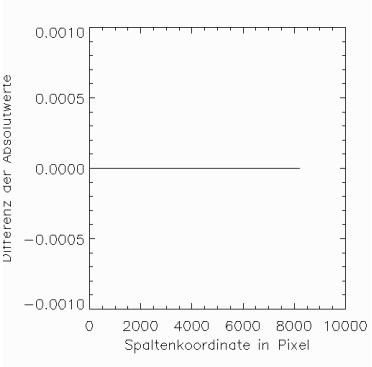


(c) Phasendifferenz

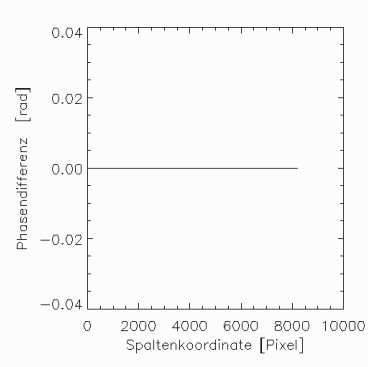
Abbildung B.21: F1, Spalten



(a) Verhältnis der Absolutwerte

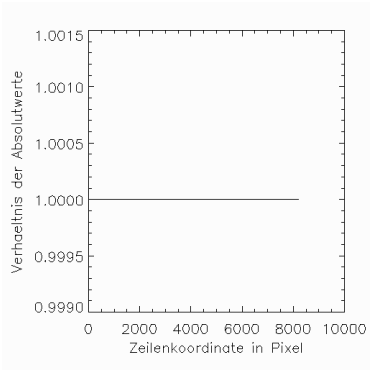


(b) Differenz der Absolutwerte

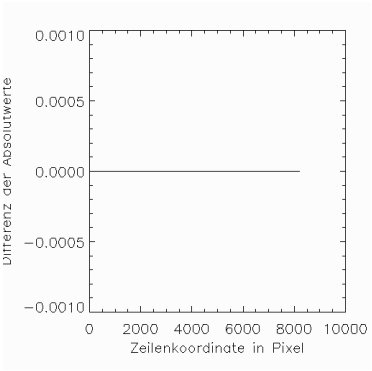


(c) Phasendifferenz

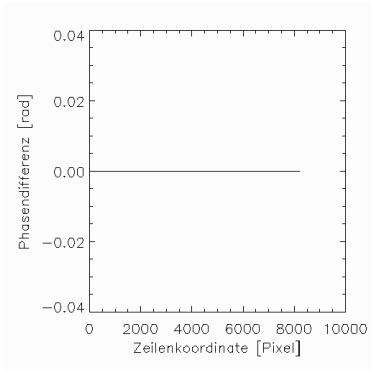
Abbildung B.22: F1, Zeilen



(a) Verhältnis der Absolutwerte

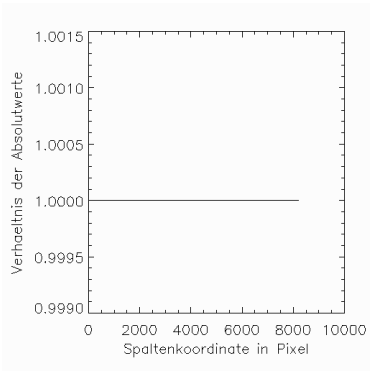


(b) Differenz der Absolutwerte

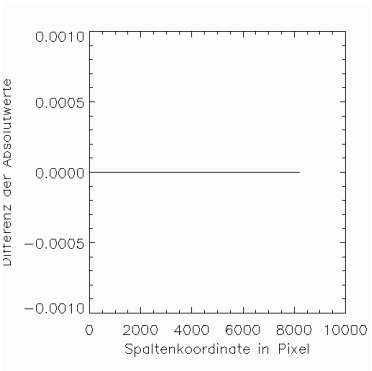


(c) Phasendifferenz

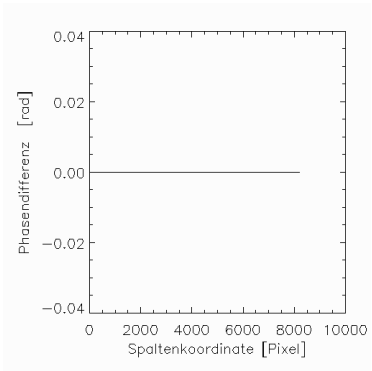
Abbildung B.23: I1, Spalten



(a) Verhältnis der Absolutwerte



(b) Differenz der Absolutwerte



(c) Phasendifferenz

Abbildung B.24: I1, Zeilen