Mannheim DHBW Duale Hochschule Baden-Württemberg

Bericht zu Modul Praxis II
3. Praxisphase: 04.01.10 - 26.03.10
4. Praxisphase: 22.06.10 - 30.09.10

Beschleunigung der Verarbeitungsgeschwindigkeit flugzeuggestützter SAR Daten durch Auslagerung rechenintensiver Verarbeitungsschritte auf eine Grafikkarte

Maren Künemund

20. August 2010

Matrikelnummer: 292597
Kurs: TIT08A

Institut für Hochfrequenztechnik und Radarsysteme
Abteilung SAR-Technologie
Fachgruppe Signalverarbeitung

Betreuer: Dr. Rolf Scheiber

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

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 20. August 2010

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Tabellenverzeichnis

1 Einleitung

Mit dem experimentellen Nachweis von Heinrich Hertz, dass elektromagnetische Wellen auch bei niedrigeren Frequenzen und nicht ausschließlich im Frequenzbereich des sichtbaren Lichtes existieren, startete die Entwicklung einer Technik zur Ortung von metallischen Objekten.
Schließlich zeigte auch das Militär Interesse an einer solchen Technologie, um so Schiffe, Flugzeuge und andere militärische Fahrzeuge frühzeitig orten zu können. Die Entwicklung schritt voran, doch mit Ende des Zweiten Weltkrieges konnte Radar (ursprünglich Radio Aircraft Detection and Ranging) auch andere Anwendungsgebiete erobern, z.B. in der Erdfernerkundung.
Radar ermöglicht es den Menschen Informationen über entfernte Ziele zu erhalten, die über seinen eigenen Sichtbereich hinausgehen. Die Orientierung geschieht durch reflektierte bzw. zurückgesendete elektromagnetische Wellen. Ein Radarbild entsteht nach dem Echoprinzip. Der Sender strahlt mit einer Frequenz zwischen 1 bis 100 GHz in das Beobachtungsgebiet. Die angestrahlten Ziele absorbieren und reflektieren teilweise die Strahlung. Die zurückreflektierten Signale werden durch die Empfangsantenne aufgenommen. Mit Hilfe dieser Signale kann man Ziele orten und vermessen, bzw. zu einem Radarbild verrechnen.
Vergleicht man nun ein durch Radar entstandenes Bild mit einem optischem Bild, kann man feststellen, dass die optische Feinauflösung in der Regel sehr viel besser ist, als bei einem Radarbild. Der Vorteil vom Radar liegt jedoch in der Unabhängigkeit von Wetter und Tageszeit. Zudem hat es eine größere Reichweite und es kann die Lage, Entfernung und Geschwindigkeit von Zielen bestimmen.
Im Institut für Hochfrequenztechnik und Radarsysteme des DLR in der Abteilung SAR-Technologie am DLR hat man es sich zur Aufgabe gemacht, Radarbilder der Erdoberfläche zu erstellen. Seit Ende der 1980'er Jahre ist das flugzeuggetragene Radarsystem mit synthetischer Apertur E-SAR im operationellen Betrieb. Dieses befindet sich an Bord eines Flugzeugs vom Typ Do-228 und fliegt Kampagnen für kleine und mittelgroße Unternehmen sowie Universitäten, Forschungsinstitute und europäische und internationale Agenturen.
Aufgrund der hohen Kundenanfrage wird derzeit an dem neuen und verbes-

serten F-SAR-System im Institut weiterentwickelt, welches seit 2008 das alte E-SAR sukzessiv ablöst.

2 Aufgabenstellung

Ereignisse wie Überschwemmungen oder Unfälle, aber auch Massenveranstaltungen mit Tausenden von Teilnehmern können die Infrastruktur des Verkehrswesen lahm legen. Um diese Probleme erfolgreich zu bewältigen, ist ein gutes Verkehrmanagment nötig, dass in den richtigen Momenten die richtigen Entscheidungen trifft. Doch worauf kann man seine Entscheidungen bauen? Genau mit dieser Problematik setzt sich das Projekt VABENE auseinander. VABENE steht für Verkehrsmanagement bei Großereignissen und Katastrophen. Viele Institute des DLRs arbeiten gemeinsam an Wegen zur besseren Bewältigung solcher Belastungen des Verkehrsnetzes. Unter ihnen ist auch das Institut für Hochfrequenztechnik und Radarsysteme. Die Abteilung SAR-Technologie leistet mit dem Einsatz des F-SAR Systems einen wichtigen Beitrag für VABENE.
Wetter- und Tageszeitenunabhängigkeit sowie Bewegungserkennung machen Radarbilder des F-SAR Systems so bedeutend für das Projekt. Anhand des Beispiels einer Überschwemmung kann dies klar herausgestellt werden. Während Fahrzeuge auf Verkehrsnetze angewiesen sind, die im Falle einer Katastrophe nur eingeschränkt genutzt werden können, sind Flugzeuge unabhängiger und bieten die notwendige Mobilität, Informationen über Krisengebiete so schnell wie möglich zu erhalten. Radarbilder warten darüber hinaus mit dem Vorteil auf, dass sie sowohl bei schlechtem Wetter als auch in der Nacht zuverlässige Bilder liefern.
Wichtiges Kriterium bei VABENE ist die Zeit, in der die Informationen zur Verfügung stehen. Um die Krisensituationen so schnell wie möglich zu bewältigen, sind verwertbare Daten notwendig, die unmittelbar vorliegen müssen. Eine Echtzeitprozessierung der Radarrohdaten an Bord des Flugzeugs ist bereits möglich. Jedoch weisen die Bilder dieser Prozessierung eine relativ geringe Auflösung auf, was lediglich für einen groben Überblick der Situation ausreicht. Hochauflösende Bilder hingegen werden derzeit offline am Boden gerechnet, sollen aber in Nahezu-Echtzeit verfügbar sein. Pro Kanal

fallen Daten in der Größenordnung von 2 bis 3 GB an. Die Rechenzeit eines solchen Bildes am Boden beträgt momentan 2,5 Stunden. Diese Zeiten sind natürlich für einen realen Einsatz inakzeptabel und sollen deshalb auf 5-10 Minuten verringert werden.
Um dies zu bewerkstelligen reicht es nicht aus, auf höhere Rechenleistung des Hauptprozessors (CPU) zu setzen. Zwar schreitet die CPU-Entwicklung immer mehr voran, kann aber nicht mit den anfallenden Prozessierungsaufgaben Schritt halten. Aus diesem Grund müssen in Sachen Hardware und Algorithmen neue Wege gegangen werden.
Da viele Aufgaben bei der Bildprozessierung gut parallelisierbar sind, liegt ein Umstieg auf einen Grafikprozessor nahe. Wie unter Quelle [1] berichtet wird, hat sich die Umstellung in anderen Anwendungsgebieten bereits gelohnt.

		Output	**Dauer in min**
		read	0:05
I		First order moco	0:01
		No Cuda enabled device found: proceed with CPU processing	0:00
	1	Range FFT	0:12
		First order MoCo	0:05
		Range IFFT	0:03
		first moco end	0:00
	2	Velocity compensation cubic	0:31
		Performing Doppler Centroid Correction ...	0:04
	3	Presumming	0:48
		RDVV ges.	1:44
II		ECS	0:04
	4	Azimuth FFT	0:17
		Chirp scaling	0:06
	5	Range FFT & RCMC & Range IFFT	0:45
	6	Phase Correction	0:06
	7	Azimuth IFFT	0:04

Weiter auf der nächsten Seite

		Output	**Dauer in min**
II	7	Second order motion compensation with resampling (reference height)	1:19
		Calibration correction	0:14
		Azimuth FFT	0:14
	8	Azimuth Compression	0:23
		Azimuth Antenna Pattern Correction	0:00
		Azimuth IFFT	0:08
		ECS ges.	3:40
		Write	0:06
		insgesamt	5:35

Tabelle 1: Auszug der Ausgaben des ECS-Prozesses und deren Laufzeit in min für eine Rohdatengröße von 2048×32768 komplexen Elementen

In diesem Semester lag meine Aufgabe darin, die Radardatenvorverarbeitung (RDVV) mit Hilfe einer Implementierung auf der Grafikkarte zu beschleunigen. Zunächst mussten dazu Zeiten aufgenommen werden, um zu erkennen, welche Teilschritte dieses Prozesses am längsten dauern und somit die meisten Performancevorteile bringen würden. Rechenintensive Schritte wie Fouriertransformationen, Interpolationen und Matrixmultipliaktionen konnten identifiziert werden. In Tabelle 1 sind einzelne Schritte der SAR Datenverarbeitung aufgeschlüsselt. Die Zeitangaben beziehen sich auf ein prozessiertes Bild der Größe 2048×32768 komplexer Datenelemente. Das ergibt ein Datenvolumen von rund 500 MB. Die vollständige Berechnugszeit des Prozesses beträgt 5:35 Minuten. Ziel ist es, diese Rechenzeit zu verringern, indem gut parallelisierbare und rechenintensive Schritte auf die Grafikkarte ausgelagert werden.
Die folgenden Kapitel beschreiben die Grundlagen bezüglich CUDA mit anschließender Darstellung der Umsetzung. Es wird dabei näher auf die Arbeitsumgebung eingegangen, sowie auf Hard- und Software und die Art und Weise der Implementierung bzw. der zu berücksichtigenden Besonderheiten bei der Realisierung der einzelnen Prozessierungsschritte auf der GPU.

3 Grundlagen

3.1 Grafikprozessor vs Hauptprozessor

Die Graphics Processing Unit (GPU) ist ein Grafikprozessor, welcher standardmäßig für Bildschirmausgaben und Graphic Rendering verantwortlich ist. Die große Nachfrage an neuen Spielen mit immer besserer Grafik und neuen Ansprüchen hat die Entwicklung der GPU schnell vorangetrieben. Durch dieses Anwendungsgebiet ist sie auf rechenintensive und stark parallelisierte Aufgaben spezialisiert. Somit liegt das Hauptaugenmerk ihrer Komponenten mehr auf Datenverarbeitung als auf Datencaching und Flusskontrolle.
Datencaching wird dadurch vermieden, dass Zwischenergebnisse nicht abgespeichert, sondern neu berechnet werden. Das führt weiterhin zur Verbergung von Speicherlatenzen. Stärken der GPU liegen in der hohen Parallelisierbarkeit, der Speicherbandbreite innerhalb der Grafikkarte und der Multithreading-Möglichkeit. Aufgrund ihrer Manycore-Prozessoren haben sie große Bedeutungen für parallelisierbare Algorithmen. Im Detail bedeutet das, dass gleiche Programme parallel auf vielen Datenelementen ausgeführt werden. Da hierbei immer die gleichen Verarbeitungsschritte aufgerufen werden, ist weniger Flusskontrolle notwendig. Aus diesen Gründen werden mehr Bauteile für Recheneinheiten verwendet als für Speicher, wie Abbildung 1 deutlich zeigt.

Abbildung 1: Aufbau von Grafik- und Hauptprozessor [10]

Im Gegensatz zur Central Processing Unit (CPU) besitzt die GPU einen relativ begrenzten Speicher. Zudem können Daten, die vorerst nicht gebraucht werden und nicht in den Hauptspeicher passen, nicht wie bei der CPU in

den Hintergrundspeicher ausgelagert werden. Es ist also stets der verfügbare Speicher im Auge zu behalten. Auch die Speicherzugriffszeiten können Bottlenecks einer Berechnung auf der GPU sein. Geschwindigkeitseinbußen hat man vor allem bei der Übertragung der Daten vom Hauptspeicher auf den Grafikkartenspeicher über den PCI-Express Bus. Für dieses Problem gibt es Abhilfe in Form von verschiedenen Speichermöglichkeiten wie Shared Memory, Constant Memory, Texture und noch einigen mehr. Für verschiedene Anforderungen gibt es spezialisierte Speicher, die die Speicherlatenzen optimal verbergen.

Vorteile	Nachteile
-viele Rechenoperationen pro Zeiteinheit -hohe Speicherbandweite innerhalb GPU-Speicher -sehr hohe Parallelisierbarkeit -für verschiedene Anwendungsfälle stehen verschiedene Speicher zur Vergügung	-Datenübertragung von CPU- auf GPU-Speicher -kleinerer Speicher

Tabelle 2: Vor- und Nachteile einer GPU-Implementierung

Zusammenfassend kann man sagen, dass sich die Umstellung der Berechnung auf eine Grafikkarte erst lohnt, wenn man mit großen Datenmengen rechnet und die Operationen hoch parallisierbar sind. Ein typischer Ablauf solcher Berechnungen kann wie folgt umrissen werden: Zuerst wird der Datensatz auf die Grafikkarte übertragen. Nun folgen komplexe, parallelisierte Berechnungen. Abschließend werden die Daten zurück in den Hauptspeicher kopiert. Sind die Datenmengen zu groß, als dass sie in den Speicher der GPU passen, müssen diese in kleinere Teile aufgetrennt und nacheinander berechnet werden. Tabelle 2 zeigt in Kurzform die besprochenen Vor- und Nachteile einer GPU.
Im Falle der zu optimierenden RDVV fallen vor allem Aufgaben wie FFT und Matrixmultiplikationen an, die im hohen Maße parallelisierbar sind. Die zu berechnenden Datenmengen werden kaum weniger als 2000×8000 kom-

plexe Elemente beinhalten. Somit erscheint die Grafikkarte für diese Aufgabe wie geschaffen zu sein. Ob sich dieser Umstieg rentiert und welche Beschleunigungen diese Berechnungen erfahren, wird sich in Kapitel 5 zeigen.

3.2 Arbeitsumgebung

Hardware Um die Vorteile der CUDA-Architektur nutzen zu können, braucht man eine CUDA-aktivierte Grafikkarte. Das bedeutet, man ist in Bezug auf Hardware an Grafikkarten von Nvidia gebunden. Einzige Möglichkeit, diesen Nachteil zu überwinden, ist eine andere Architektur zu wählen. Warum aber CUDA gewählt wurde und welche Vorzüge es im Gegensatz zu den anderen hat, beschreibt das Kapitel 3.4.
Um den hardwaremäßigen Anforderungen zu entsprechen, wurde ein neuer Rechner mit einer NVidia Quadro FX 4800 angeschafft. Tabelle 3 zeigt die Spezifikationen sowohl des Hauptprozessors als auch der Grafikkarte. Jegliche zeitliche Verbesserungen müssen immer unter Berücksichtigung der Hardwareeigenschaften der CPU vorgenommen werden. Ein direkter Vergleich zwischen GPU-Implementierungen und normaler, eventuell parallelisierter Implementierung kann auf Grund der verschiedenen Architekturen irreführend sein.
Die NVidia Quadro Reihe ist vor allem für den profesionellen Einsatz entwickelt und deswegen für wissenschaftliche Arbeiten gut geeignet. Mit ihrem 1,5 GB globalen Speicher, den vielen Prozessoren und der hohen Rechenleistung ist sie für die Verarbeitung größerer Bilder optimal ausgelegt. Aber auch die Möglichkeit, Berechnungen in double-precision auszuführen, die für einige Rechenschritte sehr wichtig sind, spricht für die Nutzung einer Quadro-Grafikkarte.

Software Zur Entwicklung von Programmen, die auf einer NVidia Grafikkarte laufen sollen, existiert die Programmierschnittstelle CUDA (Compute Unified Device Architecture). Sie ist eine Erweiterung der Programmiersprache C und unterstützt mittlerweile einige Features wie Templates und Namespace, die man aus C++ kennt. Durch das Einführen neuer Syntax in C ist es möglich, eigene Funktionen zu schreiben, die auf der Grafikkarte ausgeführt werden können. Wie genau solche Funktionsaufrufe aussehen und was dabei

Name	**NVidia Quadro FX 4800**	**Intel(R) Xeon(R) CPU**
Globaler Speicher/ Hauptspeicher	1,5 GB	4 GB
Multiprozessoren	24	2
Anzahl aller Prozessoren	192	4
Taktrate	1,2 GHz	2,5 GHz

Tabelle 3: Hardwarespezifikationen der Entwicklungsumgebung

zu beachten ist, wird in Kapitel 3.3 erklärt.
Da der Quelltext des F-SAR C++ Prozessors sehr umfangreich und komplex ist, kann schnell die Übersicht über die vielen Dateien verloren gehen. Um dennoch schnell, bequem und effizient neuen Code zu entwickeln, ist eine gute Entwicklungsumgebung unabdingbar. Für meine Arbeit entschied ich mich, Eclipse mit C++ Erweiterung zu verwenden. Es unterstützt nicht nur Syntaxhighlighting für C/C++, auch Autovervollständigung und verschiedene Kompilierungstools sind einsetzbar. Mit verschiedenen Suchfunktionen können Klassen und Methoden schnell gefunden werden, ohne dass mühselig durch die Ordnerstruktur geklickt werden muss. Ein weiteres komfortables Tool ist der Vergleich einer Datei mit sich selbst zu verschiedenen Zeitpunkten oder im SVN-Repository.
Sobald ein Teil der Algorithmen implementiert ist, müssen die Ergebnisse überprüft werden. Im Falle von Radardaten liegen viele komplexe Datenelemente vor, die schwer überschaubar und vergleichbar sind. Aus diesem Grund ist es sehr hilfreich, sich diese Daten zu visualisieren. Hierfür existieren höhere Programmiersprachen (z.B. IDL, Matlab). IDL, bereits auf dem Entwicklungsrechner vorinstalliert, eignet sich nicht nur hervorragend zum Plotten verschiedenster Daten, sondern bietet auch eine gute Umgebung, um Daten schnell zu berechnen und zu manipulieren. Darüber hinaus kann man sich kleine Programme schreiben, um immer wiederkehrende Berechnungs- und Plottingaufgaben zu automatisieren. Im Rahmen dieser Praxisarbeit wurden nicht nur die Daten mit Hilfe dieses Tools überprüft, auch die Ergebnisse

wurden so festgehalten und analysiert.

3.3 Parallele Berechnungsarchitektur - CUDA

CUDA steht für Compute Unified Device Architecture und ist eine von NVidia entwickelte parallele Berechnungsarchitektur. Sie erlaubt es ohne größeres Vorwissen über Grafikkartenprogrammierung, Programme für diese Hardware zu entwickeln. Vor allem ist CUDA mit C-Erweiterung verbreitet. Aber auch JCUDA für Java gewinnt immer mehr an Bedeutung.
CUDA erweitert die Programmiersprache C um einige Syntaxelemente sowie einige Befehle, um zum Beispiel Speicher zu allokieren oder bestimmte Rechenoperationen vorzunehmen. Zur Kompilierung der in CUDA geschriebenen Programme gibt es einen hauseigenen Compiler: nvcc. Näheres findet sich dazu in Quelle [12].
Um eine klare Grenze zwischen Code, der auf dem Hauptprozessor läuft, und Code auf der Grafikkarte zu ziehen, werden zwei Begriffe eingeführt: Host und Device. Mit Host beschreibt man den Quellcode, der wie gewohnt auf dem Hauptprozessor ausgeführt wird. Dementsprechend umfasst Device den GPU-spezifischen Quelltext.

3.3.1 Funktionsaufruf

Wesentliches Merkmal von CUDA-Code ist der sogenannte Kernel - eine Funktion, die direkt auf der Grafikkarte ausgeführt wird. Um die hohe Anzahl von Prozessoren auszunutzen, werden jedem Kernel zwei Argumente übergeben. Sie spezifizieren, wieviele Threads und Blöcke für den jweiligen Kernel benutzt werden.
Ein Kernel wird stark parallelisiert aufgerufen. Jede Ausführung des Programms wird als Thread bezeichnet. Um nun besser auf Daten zugreifen zu können, werden Threads gleichmäßig zu Blöcken zusammen gefasst. Alle Blöcke zusammen ergeben wiederum ein Grid, wobei jeder Block durch die zwei eben erwähnten Argumente definiert und mit Hilfe von Indizes angesprochen wird. Bild 2 ist eine grafische Veranschaulichung dieser Thematik.
Blöcke können linear oder zweidimensional organisiert sein. Um sich diese Eigenschaft zunutze zu machen, existieren in CUDA die Variablen gridDim und blockIdx. In einem zweidimensionalen Grid werden die Koordinaten eines

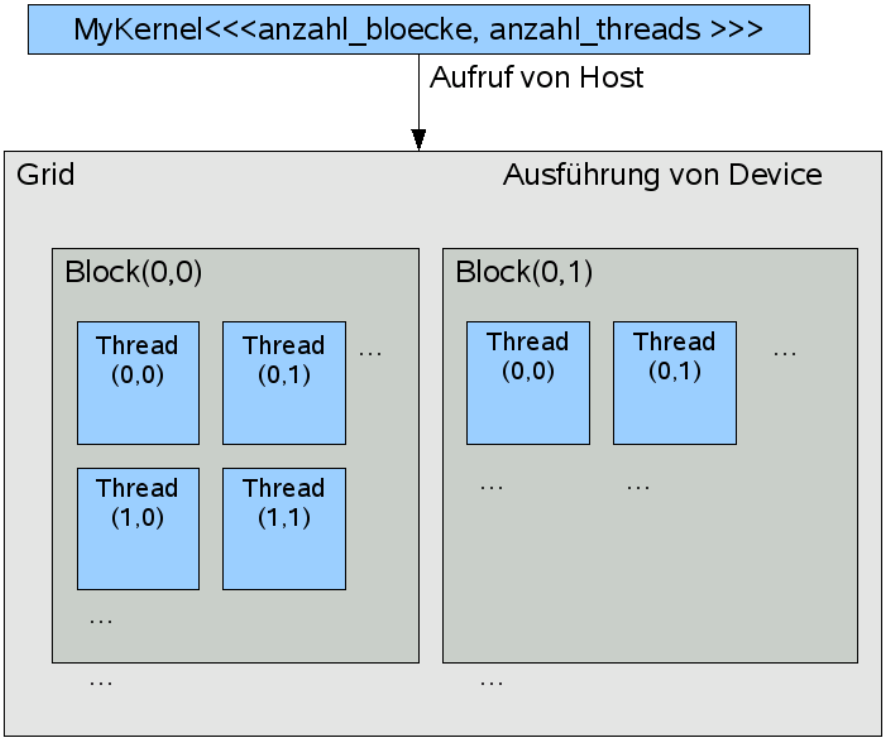


Abbildung 2: Kernelaufruf

Blockes mit blockIdx.x und blockIdx.y angesprochen. gridDim speichert die Anzahl der im Grid vorhanden Blöcke und hat analog zu blockIdx auch eine x und y-Variable. Für ein lineares Grid entfällt die y-Variable.
Genauso wie Blöcke werden auch Threads organisiert, jedoch können diese sogar dreidimensionale Ausmaße annehmen. Hierzu werden blockDim und threadIdx um die Variable z erweitert.
Die Aufteilung eines Grids in Blöcke und Threads rührt von der Hardwarearchitektur der Grafikkarten her. Jede Karte besteht aus mehreren Multiprozessoren. Und jeder Multiprozessor besitzt 8 Prozessoren mit eigenen Registern und einer gemeinsamen Instruction Unit. Übetragen auf einen Kernelaufruf wird jedem Multiprozessor ein Block zugeordnet und jedem Prozessor ein Thread.
Welche Anzahl von Blöcken und Threads optimal sind, hängt stark von der Anzahl der Multiprozessoren und dem benötigten Shared Memory ab. Eine Faustregel ist, 192 oder 256 Threads pro Block zu verwenden. Um die Re-

sourcen optimal auszunutzen, ist es auf jeden Fall notwendig, dass die Anzahl der Threads einem Vielfachen von 32 entsprechen. Blöcke sollten wenigstens genauso viele vorhanden sein wie Multiprozessoren. Bei der Wahl dieser Angaben muss man ein gutes Gleichgewicht zwischen Speicherauslastung und Belegung von Rechenzeiten auf den Prozessoren schaffen. Wenn zu wenig Blöcke und Threads definiert werden, kann es passieren, dass einige Prozessoren der Grafikkarte nichts zu rechnen haben. Wählt man die Anzahl der Blöcke und Threads zu hoch, kann der Speicher und die Anzahl der Register nicht ausreichen, um die Berechnungen zu parallelisieren.

3.3.2 Speicherhierarchie

Um schnellstmöglichen Zugriff und Datenaustausch zwischen Threads zu gewährleisten, ist der Grafikkartenspeicher mit verschiedenen Speichermedien ausgestattet. Abbildung 3 zeigt die unter CUDA existierenden Speicherbereiche und wer darauf zugreifen kann.

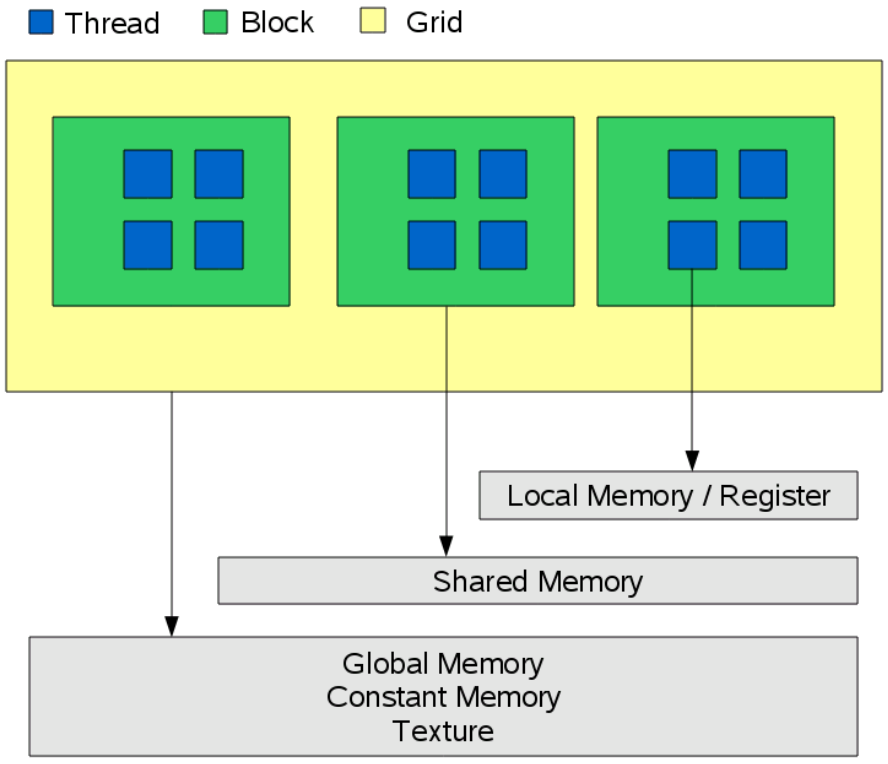


Abbildung 3: Speicherhierarchie

Register und Local Memory Der schnellste Zugriff kann über die Register der jeweiligen Prozessoren erfolgen. Da jeder Thread einem Prozessor zugeordnet wird, hat jeder Zugriff auf seine eigenen Register. Zudem existiert für jeden Thread ein Local Memory, der weder gecached wird noch über eine hohe Datenübertragungsrate verfügt. Einziger Vorteil dieses Speicher liegt im zusammenhängenden Ablegen der Daten. Das ermöglicht einen effektiven Zugriff ohne Konflikte.

Shared Memory Der Speicher pro Block wird als Shared Memory bezeichnet. Der Zugriff auf diesen Speicher kann genauso schnell erfolgen, wie der Zugriff auf Register, wenn:

1. beim Zugriff keine Bankkonflikte auftreten und
2. sich die entsprechenden Threads innerhalb eines Warps befinden.

Um schneller auf den Shared Memory zuzugreifen, wird dieser in gleich große Speicherbereiche aufgeteilt, sogenannte Bänke. Bankkonflikte entstehen, wenn gleichzeitig von mehreren Threads auf den selben Speicherbereich einer Bank zugegriffen wird.
Ein Warp ist eine Gruppe von Threads (in der Regel besteht sie aus 32 Threads), welche theoretisch simultan ausgeführt werden. Technisch wird die Hälfte, also ein Half-Warp, gleichzeit abgearbeitet. Auch der Shared Memory pro Block ist begrenzt, in der Regel auf 16kB.

Schließlich folgt der Speicherbereich, der vom ganzen Grid benutzt werden kann. Dazu gehören Constant Memory, Texturen, sowie Global Memory, der darüber hinaus Grid-übergreifend Daten speichert.

Constant Memory Constant Memory ist ein in der Regel ein 64kB großer read-only Speicher. Er eignet sich für Konstanten und Vektoren, dessen Größen bereits vor der Laufzeit feststehen. Constant Memory wird gecached und bietet damit gute Datenzugriffe. Zudem können Latenzen verborgen werden, indem Thread des gleichen Warps auf die gleichen Speicherbereiche des Constant Memorys zugreifen.

Textures Texturen sind wie der Constant Memory read-only, jedoch können die Größen von ein-, zwei- oder dreidimensionalen Feldern dynamisch sein. Texturen bieten neben einem Cache noch viele weitere nützliche Funktionen, auf die in Kapitel 4.3.1 näher eingegangen wird.

Global Memory Global Memory ist der größte Speicher von allen, besitzt aber die höchsten Zugriffslatenzen. Optimaler Zugriff auf den Global Memory erfolgt über ein bestimmtes Zugriffsschema. Zu diesem Zwecke müssen die im Global Memory abgelegten Daten zusammenhängend abgelegt werden. Da diese Aufgabe sehr mühselig sein und viel Zeit in Anspruch nehmen kann, gibt es in CUDA bereits Datentypen, die diesen Anforderungen entsprechen. Um den Restriktionen des Zugriffsschema aus den Weg zu gehen, kann außerdem auf Shared Memory ausgewichen werden.

3.3.3 Einschränkungen

Bei der Erstellung eines CUDA-Programms mit großen Datensätzen und Dimensionen gibt es einige Dinge, die zu beachten sind.

Texturen Texturen sind einfach zu handhabende Datenstrukturen, deren Daten im Global Memory gespeichert werden. Im Einsatz zeigt sich, dass diese nicht beliebig groß werden können.

- Eine eindimensionale Texturreferenz gebunden an ein CUDA Array hat eine maximale Länge von 2^{13}
- Eine eindimensionale Texturreferenz gebunden an linearen Speicher hat eine maximale Länge von 2^{27}
- Eine zweidimensionale Texturreferenz gebunden an linearen Speicher oder an ein CUDA Array hat die maximalen Ausmaße $2^{16} \times 2^{15}$
- Eine dreidimensionale Texturreferenz gebunden an ein CUDA Array hate die maximalen Ausmaße $2^{11} \times 2^{11} \times 2^{11}$

Für die Prozessierung von F-SAR Daten bedeutet dies, dass Bilder die Größe von 65536×32768 Elementen in den jeweiligen Dimensionen nicht überschreiten

dürfen und nicht mehr Speicher belegen können, als im Gobal Memory vorhanden. Weitere Besonderheit ist die Unterscheidung von linearen Speicher und einem CUDA Array. Tabelle 4 zeigt die jeweiligen Vor- und Nachteile der einzelnen Speichermöglichkeiten auf. Aus diesen geht hervor, wann sich welche Art von Speicherallokierung als nützlich erweist.

Speichertyp	**Erstellungsart**	**Textur-möglichkeit**	**Texturupdate**
Linear memory	cudaMalloc()	1. Fungiert als linearer Cache	Frei innerhalb von Threads in den Globalen Speicher zu schreiben, solange der Schreibvorgang threadsicher ist
CUDA array	cudaMallocArray() cudaMalloc3D()	1. Cache optimiert für zusammenhängende Daten im Speicher 2. Interpolation, Wrapping und Clamping (Verhalten beim Indizieren außerhalb der Texturgrenzen)	Das Schreiben innerhalb eines Kernels ist nicht erlaubt
Weiter auf der nächsten Seite			

Speichertyp	**Erstellungsart**	**Textur-möglichkeit**	**Texturupdate**
2D pitch linear memory	cudaMallocPitch()	1. Cache optimiert für zusammenhängende Daten im Speicher 2. Interpolation, Wrapping and Clamping (Verhalten beim Indizieren außerhalb der Texturgrenzen)	Frei innerhalb von Threads in den Globalen Speicher zu schreiben, solange der Schreibvorgang threadsicher ist

Tabelle 4: Gegenüberstellung der verschiedenen Speichertypen [2]

Kernel Bei der Auswahl der Kernelspezifikationen muss man nicht nur auf Ausgeglichenheit zwischen Register und Prozessoren achten, auch ist man in der maximalen Anzahl von Threads und Blöcken eingeschränkt.

- Die maximale Anzahl von Threads pro Block liegt bei 512.
- Die Anzahl an Threads darf in den entsprechenden Dimensionen $512 \times 512 \times 64$ nicht überschreiten.
- In jeder Dimension dürfen die Blöcke eine Anzahl von 65535 nicht überschreiten.
- Pro Multiprozesser ist ein Shared Memory von 16 KB verfügbar.

3.4 CUDA im Vergleich mit anderen Architekturen

Mit der Veröffentlichung von CUDA für C im Juni 2007 war dies die erste Architektur, die es ermöglichte, die Grafikkarte als Recheneinheit zu benutzen. Erst im Dezember 2008 zog AMD nach und entwickelte seine sogenannte ATI

Stream Technology. Mit einem Jahr Entwicklungsvorsprung hatte NVidia die Nase vorne.
Im August 2009 entwickelte Apple zusammen mit AMD, IBM, Intel und Nvidia die Programmierplattform OpenCL. Sie ist im Gegensatz zu Stream und CUDA eine hardwareunabhängige Plattform für Berechnungen auf einer GPU.
Bei dieser Auswahl an Möglichkeiten würde die Wahl schnell auf OpenCL fallen, weil dieses frei von Hardwareeinschränkungen ist. Nachteil einer solchen Plattform ist die Abstraktion. Damit ein Programm auf jeder Hardware laufen kann, ist eine gewissen Abstraktion nötig, die mit Performanceverlust einher geht. Das heißt, ein Programm in OpenCL würde zwar auf allen Plattformen funktionieren, aber die Hardwaremöglichkeiten zum Beispiel einer NVidia-Karte nicht voll ausreizen können. Ein CUDA-Programm hingegen schon. Dafür würde dieses aber nicht auf eine ATI-Karte ausgeführt werden können.
Was CUDA nun so besonders macht, ist die frühere Einführung gegenüber seinen Konkurrenten. Während Stream und OpenCL noch recht „junge“ Sprachen sind, hat CUDA eine längere Entwicklungshistorie hinter sich. Auch die Nachfrage im wissenschaftlichen Bereich hat eine große Gemeinschaft entstehen lassen, die aktiv programmiert und NVidia bei der Weiterentwicklung durch Bug-Reports, Feature-Requests und Optimierung des Codes hilft. Somit stellt CUDA bereits stabile Bibliotheken zur Verfügung, bietet eine umfangreiche Dokumentation und hat eine große Gemeinschaft, die gute Unterstützung für Neueinsteiger und erfahrene Programmierer bietet.
Alles in Allem kann man sagen, dass CUDA die erste Wahl ist, da es umfangreiche Möglichkeiten hat und weiter entwickelt ist, als die anderen beiden Plattformen. Mit der Zeit wird sich zeigen, ob CUDA diesen Vorsprung weiter ausbauen kann oder ob OpenCL durch seinen großen Vorteil der Hardwareunabhängigkeit diesen Rang ablaufen kann. Doch selbst in dem Falle hat CUDA aus heutiger Sicht einen weiteren Vorteil: Die Portierung von CUDA nach OpenCL stellt laut Hersteller kein Problem dar.
Im wissenschaftlichen Bereich könnte OpenCL seinen Vorteil verlieren. Hier werden Programme entwickelt, die nicht zwingend auf vielen verschieden Arbeitsplätzen mit unterschiedlichen Grafikkarten funktionsfähig sein müssen.

Es wird vor allem auf Geschwindigkeit Wert gelegt und hauptsächlich nur auf einer speziellen Hardwarekonfiguration gerechnet. Aus diesem Grund steht die Wahl nur noch zwischen Stream und CUDA. Diese beiden können die Rechenleistung der jeweiligen Hardware optimal ausnutzen und zur besten Performance führen. Da die erwähnten Vorteile von CUDA dominieren, fällt die Wahl für viele wissenschaftliche Bereich und für meine Aufgabe auf NVidias Berechnungsarchitektur.

4 Radardatenvorverarbeitung

Die Radardatenvorverarbeitung ist in dem Prozess ECS (Extended Chirp Scaling) eingegliedert. Aus Tabelle 1 geht hervor, dass der ECS-Prozess grundsätzlich in zwei Schritte gegliedert werden kann: RDVV und dem eigentlichen Algorithmus zum Berechnen des Bildes. Die Aufgabe des Prozesses ist es, aus zweidimensionalen komplexen Radarrohdaten ein fertiges schwarzweißes Bild der Radarreflektivität zu berechnen.
Weiterhin ist die RDVV in drei Teilschritte aufgeteilt: der Bewegungskompensation 1. Ordnung, Neuabtastung und Presumming. Diese Schritte entsprechen in Tabelle 1 den Punkten 1 bis 3 und wurden im Rahmen dieser Praxisarbeit für die Grafikkarte umgeschrieben.
Die Prozessierung der Bilddaten mit Hilfe des ECS-Algorithmus findet unter Punkt II in Tabelle 1 statt. Die Unterpunkte 4 bis 8 sind Teilschritte, die zur Erstellung des Bildes notwendig sind.

4.1 CUDA-Eingliederung in den F-SAR C++ Prozessor

Kompilierung Zur Erstellung von Programmen, die auf der Grafikkarte ausgeführt werden sollen, müssen CU-Dateien erstellt werden. Das sind Dateien mit .cu-Erweiterung, die mit dem nvcc-Compiler kompiliert werden. Wenn diese Dateien in Verbindung mit gewöhnlichen C/C++ - Dateien stehen, werden beide Dateiarten seperat kompiliert und im Anschluss mit dem C/C++ Compiler verlinkt. Dieser Vorgang kann der Einfachheit halber in einem Makefile zusammengefasst werden, um sich die langen Eingaben auf

der Kommandozeile zu sparen. Für diesen Zweck gibt es ein vorgefertigtes Makefile in den CUDA SDK Beispielen, in dem bereits alle möglichen Verlinkungsarten abgedeckt sind. Über ein Template-Makefile werden nur noch die entsprechenden Dateien und Bibliotheken angegeben.
Zur Kompilierung der FSAR-Software mit Einbindung von CUDA-Code kann dieses Makefile-Template nicht genutzt, da der Kompilierungsprozess durch ein Autotool automatisch konfiguriert und dann durchgeführt wird. Hierfür existiert eine Datei im Rootverzeichnis der Software mit den Namen configure.ac. Diese ist für das Finden von notwendigen Bibliotheken und Verzeichnissen verantwortlich. Pro Ordner gibt es weiterhin ein Makefile.am, welches das Einbinden von Bibliotheken und weiteren Unterordnern regelt. Die Dateien für das Autotool müssen also so angepasst werden, dass sie auch die Kompilierung des CUDA-Codes übernehmen.
Für gewöhnlich werden nur die Makefile.am verändert, um zum Beispiel für eine Klasse eine neue Bibliothek einzubinden. Da aber für CUDA neue Bibliotheken dem Autotool erst bekannt gemacht werden müssen, muss auch configure.ac verändert werden. Dazu werden, wie in Quelltext 1 zu sehen, entsprechende Zeilen hinzugefügt.
Diese Zeilen sind notwendig, um dem Autotool mitzuteilen, wo sich die CUDA-Bibliotheken und nvcc befinden. Wichtig hierbei ist die Systemvariable LDFLAGS. Sie wird anhand der neuen Information aktualisiert, damit der CUDA-Code später mit nvcc kompiliert werden kann. Sollte diese nicht korrekt belegt werden, kann der Pfad manuell auf der Komandozeile mit export (unter Linux) nachträglich gesetzt werden. Typische Fehlermeldung hierfür ist das Nichtfinden einer bestimmten Library, obwohl diese beim Kompilieren angegeben wird.
Weiterhin muss das Makefile.am im Rootverzeichnis der Software verändert werden. Welche Zeilen notwendig sind, damit die Verlinkung korrekt funktioniert und die CUDA-Kompilierung fehlerfrei durchläuft, zeigt Quelltext 2. Cudalt.py ist ein Python-Script, welches aus Quelle [5] stammt und sicherstellt, dass verschiedene CU-Files richtig miteinander kompiliert werden.
Jeder Prozess, der mit Hilfe der Grafikkarte beschleunigt werden soll, braucht ein angepasstes Makefile.am, welches sich im entsprechenden Unterordner befindet. Quelltext 3 zeigt den wesentlichen Bestandteil der Erweiterung,

```
AC_ARG_WITH([cuda],
   [  --with-cuda=PATH
if test -n "$with_cuda"
then
   CUDA_CFLAGS="-I$with_cuda/include"
   CUDA_LIBS="-L$with_cuda/lib$SUFFIX"
   CUDA_LDFLAGS="-L$with_cuda/lib$SUFFIX"
   NVCC="$with_cuda/bin/nvcc"
else
   CUDA_CFLAGS="-I/usr/local/cuda/include"
   CUDA_LIBS="-L/usr/local/cuda/lib$SUFFIX"
   CUDA_LDFLAGS="-L/usr/local/cuda/lib$SUFFIX"
   NVCC="nvcc"
fi
AC_SUBST(CUDA_CFLAGS)
AC_SUBST(CUDA_LIBS)
AC_SUBST(NVCC)

#Check for CUDA libraries
save_LDFLAGS="$LDFLAGS"
LDFLAGS="$LDFLAGS␣$CUDA_LDFLAGS"
AC_CHECK_LIB([cudart], [cudaMalloc])
LDFLAGS="$save_LDFLAGS"
```

Quelltext 1: Erweiterung von Configure.ac [5]

die ein Makefile.am erfährt. Dies stellt sicher, dass für CU-Files der richtige Compiler verwendet wird und entsprechende Bibliotheken gesetzt werden.
Schließlich kann der Quellcode mit Hilfe des Autotools kompiliert werden. Im ersten Schritt wird ein .reconf durchgeführt. Das führt ein automake durch und erstellt in den entsprechenden Ordnern aus den Makefile.am ein jeweils passendes Makefile.in. Anschließend folgt ein .configure. Es überprüft, ob alle notwendigen Bibliotheken installiert sind, und erstellt aus Makefile.in schließlich die Makefiles. Mit diesen kann jetzt aus jedem Ordner der Software heraus ein make und make install durchgeführt werden. Ein make in einem Prozessordner hätte zur Folge, dass nur dieser Prozess kompiliert wird. Das Kompilieren der gesamten Software kann durch ein make im Rootverzeichnis veranlasst werden.

```
dist_data_DATA=cudalt.py autogen.sh
CCLD = $(CC)
LINK = $(CCLD) $(AM_CFLAGS) $(CFLAGS) $(AM_LDFLAGS) $(LDFLAGS
    ) -o $@
```

Quelltext 2: Erweiterung des Makefile.am im Rootverzeichnis der Software [5]

```
.cu.o:
    $(NVCC) -o $@ -c $< $(NVCCFLAGS)
.cu.lo:
    $(top_srcdir)/cudalt.py $@ $(NVCC) $(NVCC_CFLAGS) --
        compiler-options=\"$(CFLAGS) $(DEFAULT_INCLUDES) $(
        INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)\" -c $<
```

Quelltext 3: Erweiterung des Makefile.am im jeweiligen Unterordner [5]

Eingliederung der Kernels Ein Prozess, der Berechnungen auf der Grafikkarte auslagern soll, besteht aus vier Dateien. Diese vier Dateien sind in Abbildung 4 dargestellt. Der FSAR CPP - Quellcode beinhaltet die Algorithmen zum Prozessieren der Daten. Bisher werden diese Berechnungen nur von der CPU übernommen. Beim Starten des Prozesses kann ein Parameter übergeben werden, der dafür sorgt, dass bereits implementierte Algorithmen auf der Grafikkarte gerechnet werden. Hierzu wird an passender Stelle im Quelltext eine Funktion aufgerufen, wie zum Beispiel cudaFomoco(...). Doch bevor dies geschehen kann, muss sie im Header-File, wie in Quelltext 4 dargestellt, deklariert werden. Diese Funktionen werden im Kernelwrapper implementiert.
Im Kernelwrapper wird cuda.h inkludiert, sodass hier CUDA-Syntax und spezielle CUDA-Funktionen verwendet werden können. Der Quellcode in dieser Datei übernimmt Aufgaben wie Speicherallokierungen und -übertragungen auf die Grafikkarte, kann CUFFT-Bibliothek nutzen und zum Beispiel Eigenschaften der Grafikkarte abfragen. Die wichtigste Aufgabe dieses Quelltextes ist jedoch das Aufrufen von Kernels.
Kernels wiederum werden in der Kernel-Datei implementiert. Die Trennung von CUDA-Code und den Kernels geschieht, damit die Dateien übersichtlich

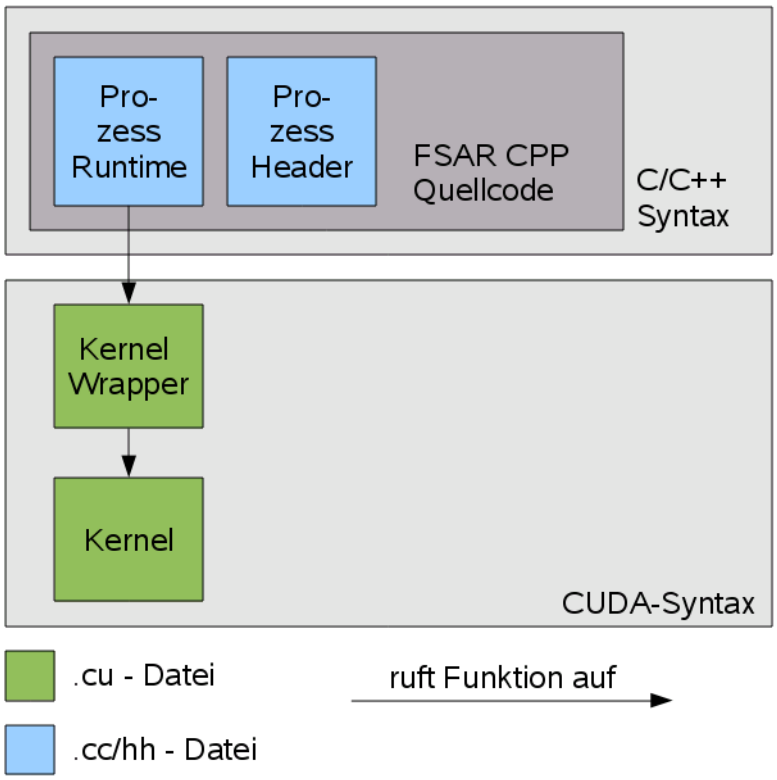


Abbildung 4: Dateien für einen Prozess mit Grafikkartenunterstützung

```
1 extern "C" void cudaFomoco(args);
```

Quelltext 4: Deklaration von CUDA-Funktionen im Header-File

bleiben und eine klare Grenze zwischen Host- und Device-Code gezogen wird. Weiterer Vorteil ist, dass diese Funktionen unabhängig voneinander aufgerufen werden können, da sie modular aufgebaut sind. Somit können gerechnete Daten unabhängig vom Prozessierungsschritt auf Richtigkeit geprüft werden. Einzelne Funktionen können auskommentiert werden, da sie nicht von vorher auf der GPU berechneten Daten abhängen. Das verhilft Bottlenecks zu finden, einzelne Algorithmen zu optimieren und GPU- mit den CPU-Daten abzugleichen.
Ein Nachteil hingegen ist der hohe Datentransfer zwischen Host und Device. Vor und nach jedem Modul werden die Daten vom Host zum Device und umgekehrt transferiert. Optimal wäre eine komplette Berechnung des RDVV auf

der Grafikkarte. Da aber während dieser Praxisarbeit vorerst Wert auf Korrektheit der Daten und Algorithmen gelegt wurde, wird sich das Optimieren auf einen späteren Zeitpunkt verschieben.

4.2 Bewegungskompensation 1. Ordnung

4.2.1 Ablauf

Die Bewegungskompensation 1. Ordnung (Fomoco) besteht im Wesentlichen aus drei Schritten.

1. Fast Fouriertransformation (FFT) zeilenweise über den gesamten Datensatz.
2. Zeilenweise Multiplikation mit einem berechneten Vektor.
3. inverse FFT (iFFT) zeilenweise über den modifizierten Datensatz.

Für die Fouriertransformationen gibt es bereits fertige Bibliotheken, die im FSAR-Quellcode genutzt werden. Im Falle des Devicecodes wird auf die NVidia Bibliothek CUFFT (siehe Quelle [6]) zurückgegriffen. Ähnlich der FFT West (siehe dazu Quelle [3], wird zuerst ein Plan erstellt und der eigentlichen FFT-Funktion übergeben. Quelltext 5 gibt diese Schritte wieder. Besonderheit der CUDA FFT beim eindimensionalen Transformieren

```
1 cufftPlan1d(&plan, width, CUFFT_C2C, height ));
2 cufftExecC2C(plan, idevData, idevData, CUFFT_FORWARD));
```

Quelltext 5: Gebrauch von CUFFT

ist, dass ein Batch angegeben werden kann. Dieser bewirkt eine parallele Ausführung von batch-Anzahl eindimensionalen FFTs. Weiterhin ist es wichtig zu beachten, dass ein Aufruf von *cufftExec* stets Daten zurückliefert, die nicht normalisiert sind. Das heißt, die Eingangsdaten entsprechen nicht den Ausgangsdaten, wenn über den Daten eine FFT und anschließend eine iFFT gerechnet wird. Jedes Element muss erst mit dem Reziproken der Anzahl der Elemente verrechnet werden.
Zur Berechnung der Phase – das ist der Vektor unter Punkt 2 – werden verschiedene Eingangsvektoren und -faktoren benötigt. Dazu gehören:

- First Order Step (Vektor mit der Länge Höhe der Daten)
- mFr - Range Frequency (Vektor mit der Länge Breite der Daten)
- Ideal filter (Vektor mit der Länge Breite der Daten)
- Wellenlänge (λ), Lichtgeschwindigkeit (c_0) und aux3 (φ_3) (Skalare)

Die Phase wird nun wie folgt berechnet:

$$\varphi_1 = \frac{-4\pi}{\lambda} \cdot FirstOrderStep$$
$$\varphi_2 = \frac{-4\pi}{c_0} \cdot FirstOrderStep$$
$$Phase = idealFilter \cdot \exp(-j \cdot ((\varphi_2 + \varphi_3) \cdot mFr + \varphi_1))$$

Die in den Rechnungen vorkommenden Multiplikationen mit Vektoren sind jeweils elementarweise Multiplikationen. Werden zwei Vektoren multipliziert, so werden die ersten Elemente miteinander multipliziert, dann die zweiten usw. Wird ein Vektor mit einem Skalar multipliziert, so dient der Skalar als Faktor für jedes Element des Vektors.

4.2.2 Implementierung

Wie bereits in Kapitel 4.2.1 beschrieben, so besteht auch der Devicecode nur aus drei wesentlichen Schritte. Die Fouriertransformation wird hierbei von der CUFFT-Bibliothek übernommen. Die Berechnungen der Phase, *aux1* und *aux2*, sowie das zeilenweise Multiplizieren mit den Radardaten erfolgen in einem Kernel.
Dieser Kernel besteht aus 256 linear organisierten Threads pro Blocks. Die Anzahl der ebenso linear organisierten Blöcke richtet sich nach der Höhe der Radardaten. Der Kernel ist so implementiert, dass jeder Block von Threads eine Zeile der Matrix berechnet. Jeder Thread muss sequentiell $\frac{Höhe}{256}$ Elemente berechnen. Nur ein Thread jedes Blocks berechnet *aux1* und *aux2* und speichert diese im Shared Memory ab. Auf diese Weise können jetzt alle Threads innerhalb des Blocks relativ schnell auf die Werte zugreifen. Da kein Thread diese Variablen lesen darf, bevor sie berechnet wurden, steht ein *__syncthreads()* in Zeile 10 des Quelltextes 6.

```
__global__ void vecXmat(Complex *A, double* mFr) {
   unsigned int rowIdx = matrixWidth * blockIdx.x;
   __shared__ float aux1, aux2;

   if(threadIdx.x == 0) {
      aux1 = -4.0 * PI/cArgs[0] * tex1Dfetch(cFoStep,
         blockIdx.x);
      aux2 = -4.0 * PI/cArgs[1] * tex1Dfetch(cFoStep,
         blockIdx.x);
   }

   __syncthreads();

   Complex phase;
   for(unsigned int idx = threadIdx.x; idx < matrixWidth; idx
      += blockDim.x) {
      phase.x = 0.0;
      phase.y = (aux2 + cArgs[2]) * mFr[idx] + aux1;
      phase = ComplexExp(phase);
      phase = ComplexMul(phase, tex1Dfetch(cVec, idx));
      A[rowIdx + idx] = ComplexMul(A[rowIdx + idx], phase);
      A[rowIdx + idx] = ComplexScale(A[rowIdx + idx], 1.0/
         matrixWidth);
   }
}
```

Quelltext 6: Kernel: fomoco

Da alle in Kapitel 4.2.1 erwähnten Variablen vom Kernel nicht verändert werden, brauchen sie nicht im Globalen Speicher abgelegt zu werden. Für die Vektoren werden Texturen angelegt, die weder interpolationsfähig noch beschreibbar sein müssen. Sie hätten ebenfalls im Constant Memory gespeichert werden können. Jedoch müsste dazu die Länge dieser bereits vor der Laufzeit feststehen. Im Gegensatz dazu ist die Anzahl der Skalare fest und ist vor der Laufzeit bekannt. Um nun die Datenübertragung auf die Grafikkarte zu vereinfachen, werden alle Skalare in einem Array abgelegt und dieses in den Constant Memory kopiert.
Die Berechnung des Zeilenindex erfolgt über $matrixWidth \times blockIdx.x$ wie

in Quelltext 6 dargestellt. Man kann sagen, dass blockIdx.x die y-Koordinate wiederspiegelt. Um nun den Index zu berechnen, betrachtet man ein zweidimensionales Array als ein eindimensionales. Ein Index ergibt sich aus der Rechnung $Zeile \times Breite + Spalte$. Da jeder Thread mehrere Elemente einer Zeile berechnet, wird zu jedem Zeilenindex ein Spaltenindex addiert, der ein Vielfaches von 256 ist und ein Offset von threadIdx.x besitzt. Dies zeigt sich im Inkrementieren der Laufvariablen *idx* innerhalb der *for*-Schleife.
Wie man an den übergebenen Parametern des Kernels erkennen kann, ist die Range Frequency als double-precision gespeichert. Aber nicht nur diese Variable hat solch hohe Genauigkeit, auch die Konstanten innerhalb des Arrays *cArgs*, aux1 und aux2 weisen die gleiche Eigenschaft auf. Das ist notwendig, damit die Vorabberechnungen mit höchster Genauigkeit ausgeführt werden. Während sich Frequenzen im Gigahertzbereich befinden, sind Wellenlängen nur im Meterbereich. Würde mit single-Precision gerechnet werden, wären die Ergebnisse zu ungenau und würden zu keinem guten Endbild führen.

4.3 Interpolation

Die Interpolation geschieht in einem Teilschritt der RDVV, in dem nicht nur die Daten neu abgetastet werden, sondern auch eine Phasenverschiebung auf Doppler Null und zurück geschieht.
Die Verarbeitung der Daten auf der Grafikkarte erfolgt folgendermaßen. Zuerst werden die Daten nach folgender Berechnung modifiziert:

$$Data = Data \cdot \exp(j(Phase))$$

Data ist ein zweidimensionales Array, in dem die komplexen Elemente der Radardaten gespeichert sind. Da es sich bei den Elementen des Phasenvektors um reelle Zahlen handelt, werden sie mit der Exponentialfunktion in komplexe umgerechnet und anschließend elementweise mit den Radardaten multipliziert.
Nun folgt die Neuabtastung der Daten mit Hilfe eines weiteren Vektors. Dieser sogenannte *axis*-Vektor beinhaltet jeden Punkt, an dem interpoliert wird. Diese Interpolation findet in y-Richtung statt. Das heißt, die x-Koordinaten das Datenarrays werden nicht interpoliert. Hier werden die ursprünglichen Koordinaten als Indizierung herangezogen.

In der CPU-Variante der Implementierung wird eine kubische-Spline Interpolation gerechnet, die den Imaginär- und Realteil der komplexen Zahlen seperat betrachtet. Die Berechnungen erfolgen über mehrere verschachtelte *for*-Schleifen, was den Algorithmus durch die sequentielle Abarbeitung sehr langsam macht. In der GPU-Variante werden diese Berechnungen durch einen sogenannten Texture-Fetch stark parallel ausgeführt. Eine ausführlichere Beschreibung folgt im anschließenden Kapitel.
Auf die gleiche Art und Weise wie die Interpolation der Daten erfolgt das Interpolieren der Phase. Zu guter letzt werden mit der gleichen Rechnung wie oben dargestellt die interpolierten Werte der Phase und der Radardaten miteinander multipliziert. Damit wird die anfängliche Phasenverschiebung auf Doppler Null zurückgerechnet und die kubische Geschwindigkeitskompensation abgeschlossen.

4.3.1 Textureninterpolation

Die Interpolation von Daten ist in CUDA bereits über Texturen geregelt. Diese können per Konfiguration und sogenannter Channel Descriptions eingestellt und definiert werden. Da in CUDA lediglich eine lineare Interpolation zur Verfügung steht, habe ich mich zu Beginn der Aufgabe über mögliche kubische Interpolationen informiert. Herausgestellt hat sich, dass ein Mitglied des NVidia Forums die Texturen so angepasst hat, dass sie eine kubische B-spline Interpolation anstelle einer linearen durchführen. Unter Quelle [14] stellt er seinen Quellcode in Form einer kleinen Bibliothek zur Verfügung. Da die Handhabung der Texturen für kubische und lineare Interpolation gleich ist, werden hier die Beispiele anhand einer linearen ausgeführt.
Zur Konfiguration einer Textur wird ein Channel Descriptor benötigt. Quelltext 7 zeigt eine Konfiguration, bei der linear interpoliert wird. *texData* ist eine zweidimensionale Textur, deren Elemente komplexwertig sind. *cudaReadModeElementType* gibt bei der Deklaration an, wie die Elemente der Textur zurückgegeben werden. In diesem Falle werden sie nicht konvertiert. Die Textur kann über verschiedene Funktionen konfiguriert werden. Der *addressMode* bestimmt, was zurückgegeben wird, wenn Elemente adressiert werden, die außerhalb der Dimensionen der Textur liegen. Wenn die Daten nicht normalisiert sind, ist *cudaAdressModeClamp* die einzige Möglichkeit. In diesem Falle

```
texture<Complex, 2, cudaReadModeElementType> texData;
//Anlegen der Textur
cudaChannelFormatDesc cd = cudaCreateChannelDesc<Complex>();
//passender Channel Descriptor
texData.addressMode[0] = cudaAddressModeClamp;
//Adressierungsmodus in x-Richtung
texData.addressMode[1] = cudaAddressModeClamp;
//Adressierungsmodus in y-Richtung
texData.filterMode = cudaFilterModeLinear;
//Festlegung, dass interpoliert wird
texData.normalized = false;
//Adressierung erfolgt nicht mit normalisierten Koordinaten
cudaBindTextureToArray( texData, caData, cd);
//Bindung der Daten an Textur
```

Quelltext 7: Channel Descriptor

werden Adressen kleiner als 0 wie 0 behandelt. Adressen größer als N sind gleichbedeutend wie N-1, wobei N die Anzahl der Elemente in einer Dimension ist. Diese Einstellungen können für jede Dimension seperat durchgeführt werden.
Desweiteren kann angegeben werden, ob die Koordinaten normalisiert sind. Da die Elemente der Achsen nicht verändert werden, würde eine Normalisierung nur unnötigen Rechenaufwand bedeuten. Mit *filterMode* wird angegeben, dass bei einer Adressierung zwischen zwei gespeicherten Elementen linear interpoliert wird. Abbildung 5 stellt diesen Sachverhalt bildlich dar. Eine weitere Variante, die in der Bewegungskompensation für die konstanten Vektoren benutzt wird, ist *nearest Point*. Dies bewirkt, wie der Name bereits sagt, dass der Adresse am nächsten liegende Punkt zurückgegeben wird. Dadurch findet keine Interpolation statt, die bei dieser Anwendung auch gar nicht benötigt wird. Trotzdem können Vorteile wie Datencaching genutzt werden.
Der Zugriff der Daten erfolgt über Funktionen dargestellt in Quelltext 8. *tex1Dfetch* ist eine spezielles Fetching, welches nur mit einer nicht interpolierenden Textur funktioniert, die zusätzlich an linearen Speicher gebunden ist.

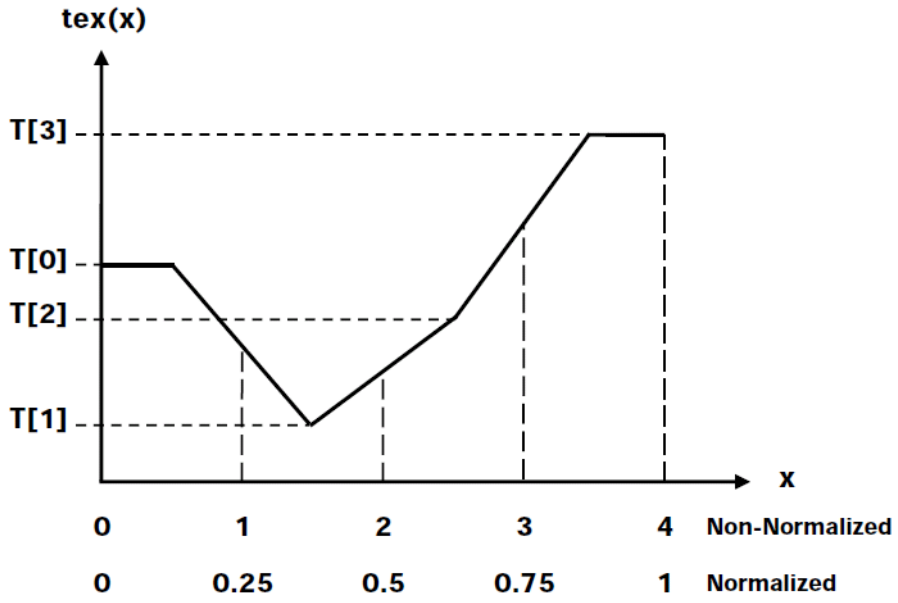


Abbildung 5: Lineare Interpolation beim Texture Fetching [10]

```
1 tex1D(tex, float x);
2 tex1Dfetch(tex, float x);
3 tex2D(tex, float x, float y);
4 tex3D(tex, float x, float y, float z);
```

Quelltext 8: Texture Fetching

4.3.2 Implementierung

Der Kernel hat $\frac{Width}{256} \times Height$ Blöcke, die zweidimensional organisiert sind. Jeder Block besteht aus 256 linear organisierten Threads. Für die Interpolation bedeutet das, dass ein Block 256 Elemente einer Zeile interpoliert. Da in y-Richtung interpoliert wird, können Daten wie zum Beispiel die y-Koordinate, an der interpoliert wird, im Shared Memory abgespeichert werden.
Das erste Verrechnen der Phase mit den Radardaten erfolgt über einen seperaten Kernel. Die Ursache dafür liegt darin, dass die Radardaten in einer Textur gespeichert sind. Sie können entweder nur über ein *cudaMallocPitch()* allokiertes Array verändert werden, oder über linearen Speicher. Da die zweite Variante keine Interpolation unterstützt, fällt diese ganz weg. Bei der ersten Variante muss man auf Konsistenz der Daten mit dem Cache Acht geben.

Aus diesem Grund wurde über eine weitere Möglichkeit nachgedacht.

```
__global__ void preData(Complex* data) {
   unsigned int row = blockIdx.y;
   unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;

   __shared__ Complex phase;
   if(threadIdx.x == 0) {
      phase.x = 0.0;
      phase.y = tex1D(texPhase, row + 0.5);
      phase = ComplexExp(phase);
   }
   __syncthreads();

   data[row*matrixWidth + col] = ComplexMul(data[row*
      matrixWidth + col], phase);
}
```

Quelltext 9: Kernel: preData

Die Daten werden wie gewohnt im globalen Speicher der Grafikkarte abgelegt und über den im Quelltext 9 dargestellten Kernel verändert. Danach wird mittels *cudaMemcpyToArray()* ein Datentransfer zum texturgebundenen *cudaArray* innerhalb der Grafikkarte veranlasst. Jetzt ist es möglich, auf die veränderten Radardaten mittels Texture Fetching und einem zweiten Kernel zuzugreifen.
Quelltext 10 stellt den Kernel zur Texturinterpolation dar. Wie auch bei der Fomoco werden Daten, die in den Shared Memory gespeichert werden, mit __syncthreads() kohärent gehalten. Da jeder Thread eines Blocks die gleiche y-Koordinate ($texY$) zum Interpolieren benötigt, kann der Zugriff durch den Shared Memory optimiert werden. Nicht nur die Koordinate, auch das Element des interpolierten Phasen-Vektors werden in diesen Speicher gelegt. Dieser Vorgang wird nur einmal pro Block ausgeführt. Nachdem Daten und Phase interpoliert wurden, werden sie miteinander multipliziert und zurück in den globalen Speicher geschrieben.

```
__global__ void textureInterpol(Complex *data, float* axis) {
   unsigned int row = blockIdx.y;
   unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;

   __shared__ float texY;
   __shared__ Complex phase;
   Complex element;
   if(threadIdx.x == 0) {
      texY= axis[row] + 0.5f;
      phase.x = 0.0;
      phase.y = 0 - tex1D(texPhase, texY);
      phase = ComplexExp(phase);
   }
   __syncthreads();

   element =  tex2D(texData, col + 0.5f, texY);
   data[row*matrixWidth + col] = ComplexMul(element, phase);
}
```

Quelltext 10: Kernel: textureInterpol

4.4 Presumming

4.4.1 Allgemeiner Ablauf

Abbildung 6 zeigt die Start- und Endprodukte der Presumming-Berechnung. Im Detail verläuft das Presumming folgendermaßen:
Zuerst wird über den gesamten Datensatz eine Fourier Transformation in Azimutrichtung ausgeführt. Da unter C/C++ ein zweidimensionales Array zeilenweise abgespeichert wird, wird eine FFT also über die Spalten des Arrays durchgeführt. Zu diesem Zwecke muss das Array vor der Transformation transponiert werden.
Im nächsten Schritt wird mit Hilfe eines Presumming-Skalierungsfaktors die neue Höhe des Arrays (beim transponierten Array ist das die Breite) festgelegt. Dieser Skalierungsfaktor bestimmt, welche Daten für die inverse Transformation in Betracht gezogen werden.
Schließlich wird über diesen Daten eine inverse FFT gerechnet und anschließend in ein kleineres Array zurück transponiert. So erhält man den in Abbil-

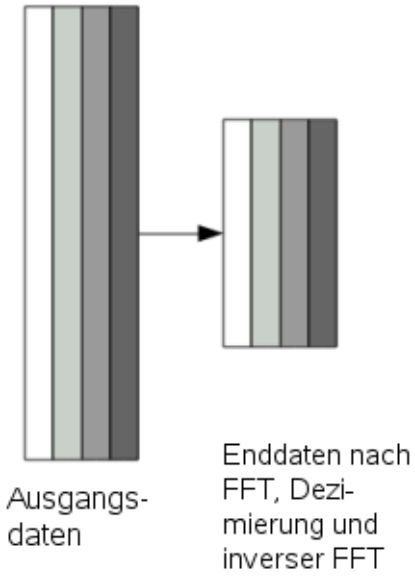


Abbildung 6: Größe der Daten beim Presumming

dung 6 auf der rechten Seite dargestellten Datensatz.
Quelltext 11 zeigt die Implementierung dieser Vorgehensweise auf der CPU und verdeutlicht nocheinmal, wie die Daten transformiert werden.

4.4.2 Ablauf auf der GPU

Während die Daten auf der CPU auf einfache Weise transformiert werden können, müssen mehr Überlegungen auf der GPU angestellt werden. Zum einen können nicht mehrere Arrays für die Daten angelegt werden, da einfach nicht genügend Speicher vorhanden ist. Desweiteren wird für eine Fourier Transformation mit CUFFT stets weiterer Speicher benötigt, sodass auch hier ein gewisser Teil für Verwaltung wegfällt. Die optimale Lösung wäre, nur ein Array für die Daten auf der Grafikkarte anzulegen. Das wäre möglich, indem im globalen Speicher vereinzelt Daten zwischengespeichert werden. Die Nutzung des Shared Memory bietet hierbei keinen Lösungsweg. Aus diesem Grund werden wenigstens zwei Arrays benötigt.
Eine Veranschaulichung der Vorgehensweise bietet Abbildung 7. Im ersten Schritt werden die Daten mit einem Kernel transponiert. Genaueres zum Verfahren der Transposition findet sich unter Kapitel 4.4.3. Damit die Daten beim parallelen Ausführen dieses Kopiervorgangs konsistent bleiben, wird ein

```
int sx, sy, sy_2;
sx = data.Width();
sy = data.Height();
sy_2 = sy / presum_fact;

Array2D<complex<float> > data2(sx, sy_2);
Array1D<complex<float> > tmp(sy);
Array1D<complex<float> > tmp2(sy_2);

for (int i = 0; i < sx; i++) {
   for (int j = 0; j < sy; j++) {
      tmp[j] = data[j][i]; //Transposition
   }
   tmp = fft(tmp, Forward);   //Fouriertransformation
   copy(tmp.begin(), tmp.begin() + sy_2 / 2, tmp2.begin());
   //erste Haelfte der relevanten Daten kopieren

   copy(tmp.begin() + (sy - sy_2 / 2), tmp.begin() + sy, tmp2
      .begin() + sy_2 / 2);
   //letzte Haelfte

   tmp2 = fft(tmp2, Inverse); //iFFT

   for (int j = 0; j < sy_2; j++) {
      data2[j][i] = tmp2[j];  //zurueck transponieren
   }
}
```

Quelltext 11: Presumming auf der CPU

zweites Array benötigt, in dem die Daten zwischengespeichert werden.
Anschließend wird parallel über alle Zeilen eine eindimensionale FFT gerechnet. In Abbildung 7 entspricht das der schwarz schraffierten Fläche. Der Presumming-Skalierungsfaktor bestimmt die neue Breite des transponierten Arrays. Die relevanten Daten entsprechen den $\frac{neueHöhe}{2}$ ersten und letzten Datenelementen. Somit wird ein Datensatz der Größe $alteHöhe - neueHöhe$ mittig ausgelassen. In Abbildung 7 ist dies das dritte Array. Die irrelevanten Daten sind etwas heller hervorgehoben.
Da über solch einem Datensatz keine performante FFT durchgeführt werden

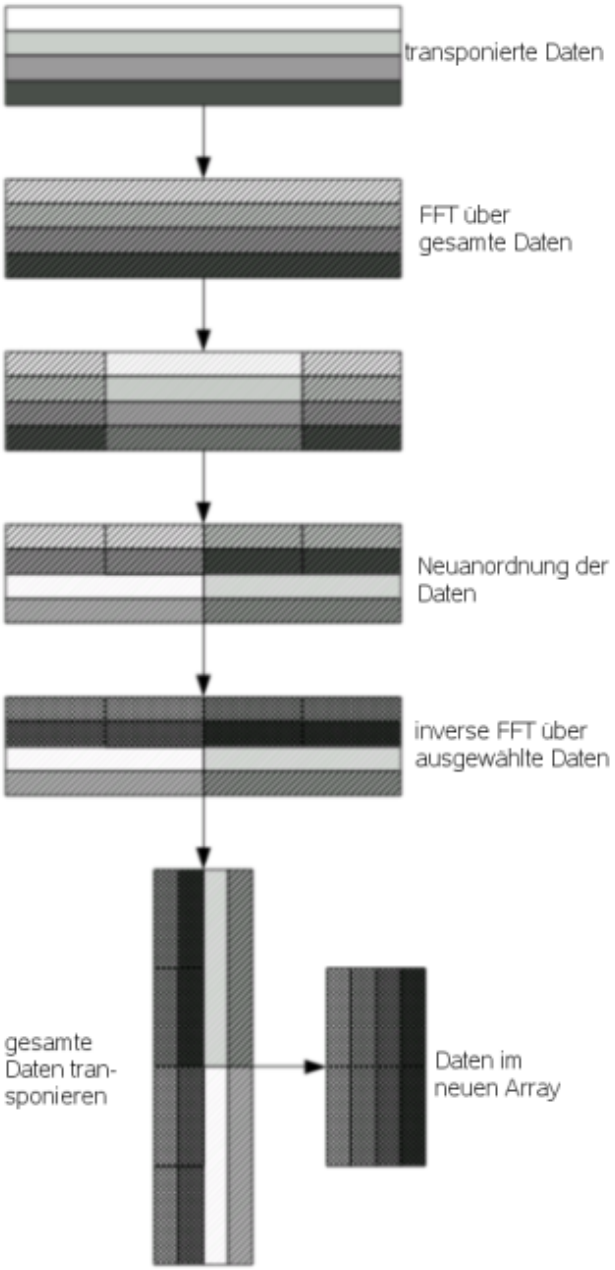


Abbildung 7: Transformieren der Daten auf der GPU beim Presumming

kann, müssen die Daten neu angeordnet werden. Die Neuanordnung geschieht über einen weiteren Kernel, der in Kaptiel 4.4.3 noch näher erklärt wird. Das vierte Array im Bild stellt das Ergebnis des Kernels dar. Da Arrays im Speicher linear abgelegt werden, das heißt, die Zeilen liegen hintereinander im Speicher, kann über das anscheinend falsch ausgerichtete Array eine inverse Fourier Transformation ausgeführt werden. Hierbei werden der Funktion lediglich neue Parameter für die Breite übergeben. Das hat zur Folge, dass das Array, wie im letzten Schritt dargestellt, interpretiert wird(natürlich als transponiertes).
Schließlich kann das Array über den gleichen Kernel wie anfangs transponiert und dann zurück in den Hauptspeicher kopiert werden.

4.4.3 Implementierung

Matrixtransposition Dieser Kernel wurde nicht selbst erstellt, sondern von den CUDA SDK Beispielen übernommen. Er bietet optimalen Zugriff auf den globalen Speicher, bankkonfliktfreien Shared Memory Zugriff, sowie Vermeidung von Partition Camping mit Hilfe von diagonaler Indizierung. Eine ausführliche Dokumentation und entsprechenden Benchmarking findet sich unter Quelle [13].
Im letzten Schritt werden die Daten wieder zurücktransponiert. In Abbildung 7 wird eine Transposition über die gesamten Daten dargestellt. Da aber lediglich eine Transposition über die relevanten Daten notwendig ist, kann bei den Funktionsparametern als Höhe die neue Höhe angegeben werden. Somit spart man sich unnötige Kopiervorgänge und dementsprechend Rechenzeit.

alignData AlignData ist der Kernel zum Neuanordnen der Daten im Speicher. Hierbei werden die relevanten Daten zusammenhängend im Speicher abgelegt und die irrelevanten verworfen.
Beim ersten Transponieren der Daten wird ein zweites Array benötigt. Daraufhin wird eine in-place bzw. in situ FFT durchgeführt. Nun besitzt man zwei Arrays: in einem ruhen die gerechneten Daten, im anderen die Eingangsdaten.
Da alignData ein out-of-place Kernel ist, wird ein zweites Array benötigt. Um nicht noch mehr Speicher zu allokieren, werden die Ausgangsdaten des Ker-

```
__global__ void alignData(Complex *odata, Complex *idata,
    unsigned int height) {

    const unsigned int height2 = gridDim.x * TILE_DIM;
    const unsigned int offset  = height - height2;
    unsigned int col = blockIdx.x * TILE_DIM + threadIdx.x;
    unsigned int row = blockIdx.y * TILE_DIM + threadIdx.y;
    unsigned int index_out = col + row * height2;
    unsigned int index_in  = col + row * height;

    float norm = 1.0/height;

    __shared__ Complex tile[TILE_DIM][TILE_DIM+1];

    if(col >= height2/2)
       index_in += offset;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] = ComplexScale(idata[
          index_in+i*height], norm);
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height2] = tile[threadIdx.y+i][
          threadIdx.x];
    }
}
```

Quelltext 12: Kernel: alignData

nels einfach mit dem Inputarray überschrieben. Das bedeutet, es werden die ersten $neueHöhe \times Breite$ Datenelemente im linearen Speicher überschrieben. Die restlichen bleiben unverändert und werden im weiteren Verlauf des Presumming-Verfahrens nicht betrachtet. Zur Veranschauliching der Speicherplatzausnutzung kann Abbildung 8 herangezogen werden.
Jeweils ein Thread des Kernels speichert in einer im Shared Memory liegende Untermatrix der Größe 32×32 ($TILE\_DIM \times TILE\_DIM$) die

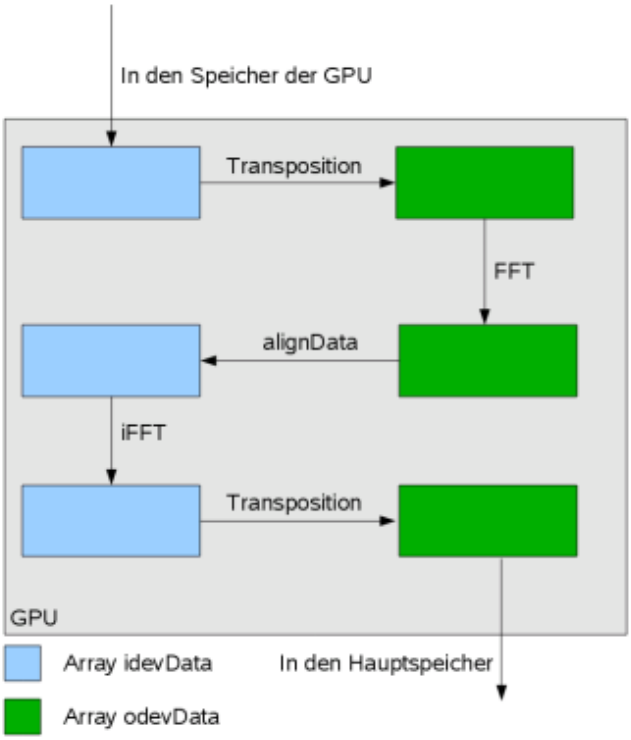
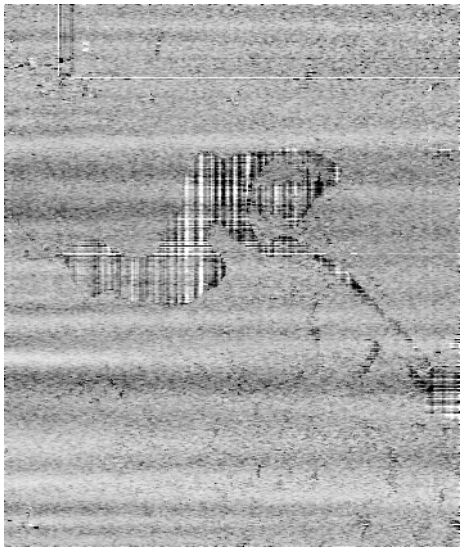


Abbildung 8: Speichernutzung beim Presumming

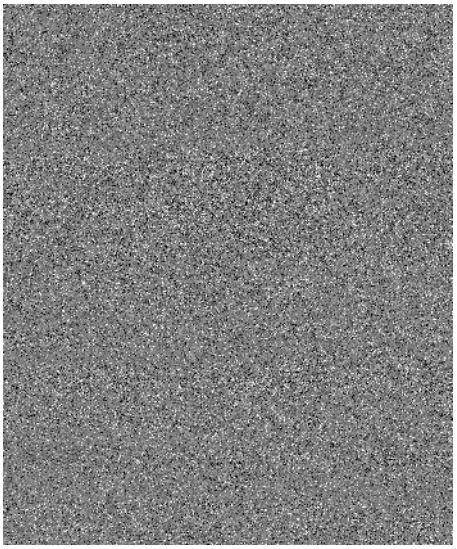
normalisierten Inputdaten ab. Nachdem alle Threads synchronisiert wurden, kann aus dem Shared Memory gelesen und die Outputdaten aktualisiert werden. Ein Block von Threads besteht jedoch nur aus 32×8 ($TILE\_DIM \times BLOCK\_ROWS$) Threads. Somit muss das Schreiben und Lesen in den Shared Memory sequentiell über eine *for*-Schleife erfolgen.
Die Berechnung der Indizes erfolgt im obersten Abschnitt des Quelltext 12. *Height2* ist die neue Höhe, *col* und *row* sind jeweils die Koordinaten in x- und y-Richtung. Da die transponierten Arrays jeweils eine andere Breite haben, müssen die Indizes verschieden berechnet werden. Beim Ausgangsarray kommt hinzu, dass ab der Hälfte der zu kopierenden Daten ein Offset existiert, der mit der *if*-Abfrage berücksichtigt wird. Anschließend erfolgt das Kopieren der Daten wie vorher beschrieben.

(a) von der CPU prozessiertes Bild

(b) von der GPU prozessiertes Bild



(c) Kohärenz beider Bilder (weiß =1, schwarz =0,999)



(d) Phasendifferenz beider Bilder ($wei\ss = 5°$, $schwarz = -5°$)

Abbildung 9: prozessierte Bilder: a) auf der CPU, b) auf der GPU, c) Kohärenz der Bilder, d) Phasendifferenz der Bilder
Bildgröße: 32768×2048 komplexe Elemente

5 Zusammenfassung

In diesem Praxissemester konnten die in Kapitel 2 gestellten Aufgaben in so weit abgeschlossen werden, dass sie einen Datensatz mit begrenzter Größe rechnen können. Diese Größenbegrenzung wird durch Textureinschränkungen, Speicherplatz und CUFFT-Begrenzungen definiert. Diese Begrenzung sollen durch Implementierung sequentieller Blockberechnungen umgangen werden. Momentan können Bilder berechnet werden, die auf der Grafikkarte ca 500 MB Speicherplatz belegen. Der Verarbeitungsschritt Fomoco könnte theoretisch bis knappe 1,5 GB unterstützen. Da aber die restlichen RDVV-Schritte noch nicht optimiert sind, entfällt eine vollständige Prozessierung dieser Größenordnung.

Berechnung	**Rechenzeit auf der CPU**	**Rechenzeit auf der GPU**	**Beschleunigung**
Fomoco	25 s	0,345 s	72,46
Interpolation	31 s	0,283 s	109,54
Presumming	48 s	0,285 s	168,42
RDVV ges.	104 s	0,814 s	127,76
ECS ges.	220 s	–	–
insgesamt	335 s	247 s	1,36

Tabelle 5: Gegenüberstellung der Rechenzeiten

Tabelle 5 zeigt die Zeiten, die für die Berechnungen auf der CPU und auf der GPU benötigt wurden. Die vierte Spalte enthält die Faktoren, um wieviel schneller die Berechnungen auf der GPU laufen. Aus dieser Tabelle ist deutlich zu erkennen, dass sich ein Parallelisieren der Algorithmen gelohnt hat. Jedoch darf mit diesen Faktoren nicht zu optimistisch umgegangen werden. Zum einen ist der Quelltext für die Berechnungen auf der CPU nicht parallelisiert. Zum anderen ist die RDVV nur ein Teil des Gesamtprozesses und macht ca. ein Drittel aller Berechnungen aus. Ohne den ECS-Algorithmus zu optimieren, könnte man also bestenfalls auf eine Geschwindigkeitsverbesserung von 30% kommen.
Neben der Performance muss auch die Qualität der Daten überprüft werden.

Abbildungen 9a und 9b stellen zwei Endergebnisse einer Berechnung dar. Mit dem bloßen Auge ist kaum ein Unterschied zu entdecken. Abbildungen 9c und 9d geben einen besseren Einblick in die Qualität der Daten. Die Kohärenz zeigt, dass wirklich nur sehr geringe und kaum merkliche Abweichungen in Bezug auf die CPU-Daten vorhanden sind. Auch die Phasendifferenz ist sehr gering mit einer Standardabweichung zwischen 1 bis 3 °.
Diese Auswertungen zeigen, dass nicht nur der Performancevorteil die Entwicklung in diese Richtung weiter attraktiv macht, sondern CUDA-Code auch mit guter Genauigkeit aufwarten kann.

6 Ausblick

Die nächsten Pläne zur Verbesserung der implementierten Algorithmen sind das Abfangen von zu wenig Speicherplatz oder zu großen Bildern. Weiterhin soll die RDVV als gesamter Rechenschritt auf die Grafikkarte ausgelagert werden, um so unnötigen Speichertransfer zu vermeiden. Es sollen neue Möglichkeiten gesucht werden, etwaige Bottlenecks zu beseitigen. Eine bisherige Überlegung ist, den Presumming-Schritt als Pipeline ausführen zu lassen.
Diese Überlegung geht mit einem vorherigen Update des Betriebssystem einher, da für neue CUDA-Releases eine neuere Linux-Version benötigt wird. Der Grund auf eine neue CUDA-Umgebung umzusteigen, liegt in der fehlenden Stream-Möglichkeit von CUFFT. Erst ab Version 3.0 wird dieses Software-Feature unterstützt und konnte deswegen in der momentan Praxisphase nicht getestet werden.
Um den gesamten Prozess zu beschleunigen, reicht es nicht aus, nur die RDVV zu beschleunigen. Weitere Aufgaben liegen also in der Anpassung des Extended Chirp Scaling Algorithmus'.
Ob weitere Prozesse mit CUDA beschleunigt werden sollen, steht noch nicht fest. Aber die guten Ergebnisse dieser Praxisarbeit zeigen, dass viel Potenzial im Parallelisieren der Prozessierung von Radardaten auf der GPU steckt.
Letztlich muss über die Möglichkeiten nachgedacht werden, wie der CUDA-Code für alle nutzbar gemacht werden kann. Momentan läuft dieser nur auf einer lokalen Entwicklungsumgebung. Natürlich sollen auch die Vorteile für

einen operationellen Einsatz nutzbar sein. Bisher stehen die Möglichkeit eines Job-Systems zur Verfügung, bei dem einzelne Aufgaben, die eine Grafikkarte benötigen, an den entsprechenden Rechner verteilt werden. Eine andere Möglichkeit wäre es, das derzeitige Cluster-System mit einer speziell für wissenschaftliche Aufgaben ausgerichtete Grafikkarte auszustatten.

Abkürzungsverzeichnis

API	Application Programming Interface
CPU	Central Processing Unit
CU	Abkürzung für CUDA, die als Dateierweiterung dient
CUDA	Compute Unified Device Architecture
CUFFT	CUDA FFT
DLR	Deutsches Zentrum für Luft- und Raumfahrt
ECS	Extended Chirp Scaling
E-SAR	Experimentelles Flugzeug-SAR
FFT	Fast Fourier Transformation
fomoco	First Order Motion Compensation (Bewegungskompensation 1. Ordnung)
F-SAR	Flugzeug-SAR
GB	Gigabyte
GPU	Graphics Processing Unit
IDL	Interactive Data Language
iFFT	inverse FFT
nvcc	NVidia CUDA Compiler
OpenCL	Open Computing Language
PCI-Express	Peripheral Component Interconnect Express
Radar	Radio Aircraft Detection an Ranging
RDVV	Radardatenvorverarbeitung

SAR	Synthetic Aperture Radar
SDK	Software Development Kit
SVN	Subversion
VABENE	Verkehrsmanagement bei Großereignissen und Katastrophen

Literatur

[1] Liu Bin, Wang Kaizhi, Liu Xingzhao, and Yu Wenxian. An Efficient Signal Processor of Synthetic Aperture Radar Based on GPU. In *Proceedings der EUSAR Konferenz 2010, Aachen*, 7. - 10. Juni 2010.

[2] Rob Farber. CUDA, Supercomputing for the Masses: Part 13. `http://www.drdobbs.com/cpp/218100902`, 2009. [Online; Stand 09. August 2010].

[3] Matteo Frigo and Steven G. Johnson. FFTW Home Page. `http://www.fftw.org/`, 1997-2008. [Online; Stand 20. August 2010].

[4] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Microsoft Corporation, 2008.

[5] Mateus Zarnick Interciso. Forumsthema: CUDA and autotools, configure.ac, makefile.am, ... `http://forums.nvidia.com/index.php?showtopic=53006`, 2007-2010. [Online; Stand 11. August 2010].

[6] NVidia. *CUDA, CUFFT Library*. NVidia Corporation, 1.1 edition, Oktober 2007.

[7] NVidia. *CUDA, CUBLAS Library*. NVidia Corporation, 2.0 edition, März 2008.

[8] NVidia. *NVidia CUDA C Programming, Best Practice Guide*. NVidia Corporation, 2.3 edition, Juli 2009.

[9] NVidia. *NVidia CUDA Development Tools 2.3, Installation and Verification on Linux*. NVidia Corporation, 2.3 edition, Juli 2009.

[10] NVidia. *NVIDIA CUDA, Programming Guide*. NVidia Corporation, 2.3.1 edition, August 2009.

[11] NVidia. *NVidia CUDA, Reference Manual*. NVidia Corporation, 2.2 edition, April 2009.

[12] NVidia. *The CUDA Compiler Drive NVCC*. NVidia Corporation, 2.3 edition, Juli 2009.

[13] Greg Ruetsch and Paulius Micikevicius. *Optimizing Matrix Transpose in CUDA*. NVidia Corporation, Januar 2009.

[14] Danny Ruijters. CUDA Cubic B-Spline Interpolation. `http://dannyruijters.nl/cubicinterpolation/`, 2008. [Online; Stand 11. August 2010].