Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Marc-Ruben Lorbeer

Analyse der NVIDIA CUDA-Architektur für die Implementierung von Signal- und Bildverarbeitungsalgorithmen

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Marc-Ruben Lorbeer

Analyse der NVIDIA CUDA-Architektur für die Implementierung von Signal- und Bildverarbeitungsalgorithmen

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Ing. Hans Peter Kölzer
Zweitgutachter : Prof. Dr. Ing. Karl-Ragmar Riemschneider

Abgegeben am 16. Februar 2010

Marc-Ruben Lorbeer

Thema der Bachelorthesis

Analyse der NVIDIA CUDA-Architektur für die Implementierung von Signal- und Bildverarbeitungsalgorithmen

Stichworte

Signalverarbeitung, Bildverarbeitung, NVIDIA, CUDA, GPGPU

Kurzzusammenfassung

Es wird die parallele Rechnerarchitektur und Programmierumgebung CUDA-fähiger Graphikprozessoren untersucht. Besonderes Augenmerk wird auf die Darstellung der Hardware- und Programmierumgebung mit Erläuterung des Implementierungs- und Optimierungsverfahrens gelegt. Durch Vergleich von konventionellen, sequentiellen Implementierungen zu parallelen Implementierungen von Algorithmen in CUDA wird das Beschleunigungspotenzial durch CUDA aufgezeigt. Mit der Implementierung des FFT Algorithmus wird bis zum 70-fachen, mit der Implementierung des Faltungsalgorithmus für die Bildverarbeitung bis zum 130-fachen beschleunigt. Programmiersprache ist C/C++.

Marc-Ruben Lorbeer

Title of the paper

Analysis of the NVIDIA CUDA architecture for the implementation of signal and image processing algorithms

Keywords

signal processing, image processing, NVIDIA, CUDA, GPGPU

Abstract

The parallel computer architecture and programming environment of CUDA-enabled graphics processors is studied. Particular attention is paid to the illustration of the hardware and programming environment with explaining the implementation and optimization procedure. A comparison of conventional sequential implementations and parallel implementations of algorithms in CUDA shows the CUDA accelerated pace. The implementation of the FFT algorithm it is up to 70-times faster. With the implementation of the convolution algorithm for image processing it speeds up to 130-times. Programming language is C/C++.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungen und Bezeichnungen

3D-NOW ..... Equivalenter Instruktionssatz zu MMX (Hersteller: AMD)
API ..... Application Programming Interface
CPU ..... Central Processing Unit
CUDA ..... Compute Unified Device Architecture
DFT ..... Diskrete Fourier-Transformation
DRAM ..... Dynamic Random Access Memory
FFT ..... Fast Fourier Transformation
G80 ..... Grafikprozessor des Herstellers NVIDIA
GDDR3 ..... Graphics Double Data Rate - DDR-Arbeitsspeicher in der 3. Generation für den Einsatz auf einer Grafikkarte
GPGPU ..... General Purpose Computation on Graphics Processing Unit
GPU ..... Graphics Processing Unit
GT200 ..... Aktueller Grafikprozessor des Herstellers NVIDIA
MMX ..... Multimedia Instuction Set Extension (Hersteller: Intel)
PCIe ..... Peripheral Component Interconnect Express
PTX ..... Parallel Thread Execution
RAM ..... Random Access Memory
ROP ..... Raster Operation
SDK ..... Software Development Kit
SIMD ..... Single Instrunction Multiple Data
SIMT ..... Single Instruction Multiple Thread
SISD ..... Single Instrunction Single Data
SM ..... Streaming Multiprocessor
SSE ..... Streaming SIMD Extensions
Alignment ..... Anpassung, Ausrichtung von Speicherzugriffen
C for CUDA ..... Spracherweiterungen der CUDA RUNTIME API
Coalescing ..... Zusammenfassung, Verschmelzung von Speicherzugriffen
Compute Capability Spezifizierung der CUDA Grafikprozessoren
Constant Memory Speicherbereich auf globaler Ebene (Nur Lesen)
CUDA DRIVER API Application Programming Interface auf der Treiberebene
CUDA RUNTIME API High-Level Application Programming Interface
Device ..... Hardwareumgebung der Grafikprozessoreinheit
Device-Memory ... Speicher des Grafikprozessors
Global Memory ... Speicherbereich auf globaler Ebene (Lesen und Schreiben)
Grid ..... Zusammenhängende Organisation über die Menge der Threadblöcke bzw.

	Threads
Host	CPU/Hauptrechner
Host-Memory	Arbeitsspeicher des Hauptrechners
Kernel	Programmcode für den Grafikprozessor
Local Memory	Speicherbereich auf Threadebene
NVCC	CUDA Kompilierer-Treiber
Nvidia	Hersteller der CUDA Architektur
Register	ON-Chip Speicherbereich auf der lokalen Ebene (Threadebene)
Shared Memory	Speicherbereich auf Threadblockebene
Texture Memory	Speicherbereich auf globaler Ebene (Nur Lesen)
Threadblock	Die Zusammenfassung von parallelen Ausführungssträngen in einer Gruppe
Threads	Parallele Ausführungsstränge
Unified Shader Modell	Verallgemeinerte Grafikprozessorarchitektur
Warp	Bündelung von parallelen Ausführungssträngen in CUDA

Kapitel 1

Einleitung

Die ursprünglich nur für Grafikberechnung auf dem PC vorgesehenen Prozessoren der Grafikkarten haben auf Grund des immer weiter steigenden Leistungsbedarfs immer höhere Rechenleistung erzielt. Diese sind im Bezug auf ihre reine Rechenleistung den handelsüblichen Prozessoren deutlich überlegen. Somit stellt sich die Frage, ob sich diese enorme Rechenleistung auch in weiteren Segmenten für den allgemeinen, wissenschaftlichen Einsatz nutzen lässt!

Die Idee, eine Grafikkarte für neue Berechnungsaufgaben einzusetzen, ist nicht neu. Die Problematik ist allerdings, sich diese Leistung zu Nutze machen. Die üblichen Entwurfsmöglichkeiten für Softwareelemente für die Grafikkarte sind über Grafik-APIs (Application Programming Interfaces) realisierbar. Um diese jedoch für den wissenschaftlichen Einsatz nutzbar zu machen, sind die Berechnungsaufgaben so anzupassen, dass sie in diesen APIs umgesetzt werden können. Der Einsatz von Grafikkarten ist in dieser Form für allgemeine Berechnungsaufgaben viel zu abstrakt und für den „normalen" Anwender nicht zu gebrauchen.

Die Firma NVIDIA hat mit der Veröffentlichung der G80 Grafikprozessorgeneration im November 2006 zu einem entscheidenden Fortschritt für den allgemeinen und wissenschaftlichen Berechnungseinsatz von Grafikkarten geführt. Diese Grafikprozessoren beruhen auf einer verallgemeinerten Architekturstruktur mit einem dafür entwickelten parallelen Programmiermodell mit der Bezeichnung **CUDA - Compute Unified Device Architecture**, die unmittelbar eine Programmierung der Grafikkarte ermöglicht.

Ein Anwendungsgebiet, die Signal- und Bildverarbeitung, beinhaltet eine große Zahl an Berechnungsaufgaben und Algorithmen, die in einer parallel verarbeitenden Form enorm beschleunigt werden können. Daher ist es besonders wertvoll, eine Analyse der NVIDA CUDA-Architektur anhand von grundlegenden Algorithmen der Signal- und Bildverarbeitung durchzuführen, was Zweck dieser Arbeit ist. Hierzu wird diese Architektur studiert, um eine alternative Implementierungsmöglichkeit für die bisher üblichen Formen zu finden und Berechnungsabläufe entscheidend zu beschleunigen. Die Erörterung findet anhand von Implementierungen ein- und zweidimensionaler Fast Fourier Transformationen und des Faltungsalgorithmus (Bildverarbeitung) statt. Diese Implementierungen dienen der Darstellung der Umsetzungsformen und der Analyse der Geschwindigkeitsvorteile im Vergleich zu konventionellen Umsetzungen.

Kapitel 2

Nvidia CUDA

Die Bezeichnung CUDA steht für eine datenparallel verarbeitende Architektur zum allgemeinen Einsatz von Berechnungsaufgaben in der Softwareentwicklung. Diese wird in Grafikprozessoren der Firma NVIDIA seit der Grafikprozessorgeneration G80 implementiert und stetig weiterentwickelt. Der Hersteller NVIDIA beschreibt diesen Begriff, wie folgt:

> „CUDA™, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture"[1]

Diese Aussage beschreibt ein Gesamtkonzept eines/einer speziell entwickelten Hard- und Softwaremodells/-Umgebung für den Einsatz auf Grafikprozessoren des Herstellers NVIDIA. Der Begriff CUDA ist durch die Gesamtheit dieser Elemente bestimmt und definiert den allgemeinen Zusammenhang des Konzeptes.

Der Einsatz von CUDA erfolgt mittels einer aktuellen Grafikkarte der Firma NVIDIA in einem Windows basierenden PC System mit dem dafür nötigen Softwarepaket. Dieses Softwarepaket umfasst den Grafikkartentreiber, das CUDA Toolkit[2] und die CUDA SDK[3]. Eine detaillierte Dokumentation der eingesetzten Hard- und Software erfolgt in Abschnitt 4.1.

2.1 Grundbegriffe

In diesem Abschnitt sind die grundlegenden Strukturen und Einsatzmöglichkeiten von Rechnerarchitekturen sowie die Grundbegriffe von CUDA erläutert. Die darauf folgenden Abschnitte dienen der allgemeinen Veranschaulichung der Grafikprozessorarchitektur und deren Einsatzmöglichkeiten für die allgemeine Datenverarbeitung.

2.1.1 Rechnerarchitekturen im Vergleich

Zur sinnvollen Betrachtung des CUDA-Hardwaremodells, vorweg ein kurzer Ausblick über die bisher existierenden Rechnerarchitekturen:

[1] Zitat mit der Quelle: [NVI09a] S.3
[2] Enthält Application Programming Interfaces und Dokumentation [NVI09a]
[3] Enthält Beispielcode [NVI09f]

Die Von-Neuman- und Harvard-Architektur sind die bekanntesten Rechnerarchitekturen, welche in Mikroprozessoren implementiert werden. Diese basieren auf einem taktgesteuerten Ablauf und besitzen ein Rechen- und Steuerwerk sowie den dazugehörigen Steuer-, Adress- und Datenbus. Der wesentliche Unterschied besteht darin, dass der Von-Neuman-Rechner einen gemeinsamen Bus für Daten und Befehle besitzt, wodurch er Bus in seiner Größe beschränkt ist. Bei der Harvard-Architektur ist der Daten- und Befehlsbus getrennt und kann somit unabhängig dimensioniert und auf die jeweiligen Bedürfnisse angepasst werden.

Flynnsche Klassifikation

Durch die unterschiedliche Anordnung der Architekturen bieten sie verschiedene Anwendungsmöglichkeiten. Die prinzipielle Struktur ist als **SISD**-Architektur (***Single Instrunction, Single Data***) realisiert. Diese Prozessoren sind nur in der Lage eine Instruktion für ein Datenelement zur selben Zeit auszuführen. Jedoch besitzen die modernen Prozessoren unterschiedliche Modifikationen und Kombinationen der unterschiedlichen Architekturen. Dadurch wird das parallele Ausführen von Instruktionen auf mehrere Datenelemente zunehmend möglich. Die **SIMD**-Architektur (***Single Instrunction, Multiple Data***) ist eines dieser Konzepte. Diese Struktur ist in verschiedenen Formen in modernen Prozessoren implementiert. Die Instruktionssätze MMX, SSE sowie 3D-NOW, um einige zu nennen, sind in den heutigen Prozessoren im Personal Computer Bereich im Einsatz. Sie ermöglichen die simultane Ausführung einer Instruktion auf mehrere Datenelemente eines Datensatzes.

Dies sei an folgendem Beispiel demonstriert:

Ein 64-Bit Datenwort in acht 8-Bit Datenwörter, vier 16-Bit Datenwörter oder zwei 32-Bit Datenwörter aufgeteilt und zur selben Zeit mit derselben Instruktion verarbeitet.[4]

Das SIMD-Architekturprinzip findet Einsatz in sogenannten Vektorprozessoren und hat folgende wesentliche Merkmale:

- ▷ Die Größe der Datensätze ist fest und lässt somit keine Flexibilität in der Verarbeitung zu.
- ▷ Diese Recheneinheiten erreichen bedingt durch die parallele Verarbeitung von mehreren Datenelementen eines Datensatzes einen höheren Instruktionsdurchsatz.
- ▷ Die Bandbreite der Einsatzgebiete ist durch die starre Struktur der Vektorprozessoren begrenzt.
- ▷ Der Geschwindigkeitsvorteil ist meist nur in Problemstellungen der Parallelverarbeitung nutzbar, bei denen die gleiche Operation mit vielen Datenelementen durchzuführen ist

Spezielle Formen

Es existieren auch Konzepte einer Anordnung von mehreren unabhängigen Rechen- und. Instruktionseinheiten, die es ermöglichen mehrere Operationen zur gleichen Zeit durchzuführen. Die Implementierung einer Harvardarchitektur unter Verwendung von Instruktionspipelines und einer höheren Anzahl von Instruktionseinheiten ermöglicht diese Funktionalität. An dieser Stelle sind

[4] Quelle: [BJ05] S. 25, 26

als Beispiel die Signal Prozessoren des Herstellers Texas Instruments aus der Serie TMS320C6x[5] zu nennen.

Das CUDA-Hardwaremodell beinhaltet eine Form der SIMD-Architektur mit einer arrayförmigen Anordnung von skalaren Prozessoreinheiten. Diese Technologie trägt die Bezeichnung **SIMT** und steht für ***Single Instuction Multiple Threading***. Sie basiert auf einer Architektur mit einer hohen Anzahl von Ausführungssträngen, welche in Bündeln jeweils eine Instruktion simultan durchführt. Diese Bündel werden im Kontext von CUDA als Warps bezeichnet. Die ausführliche Erläuterung dieser Technologie erfolgt in Abschnitt 2.2.4.

2.1.2 Rechnerarchitektur GPU

Im Zusammenhang von Grafikprozessoren ist oftmals die Rede von dem Begriff GPU. GPU steht für die Bezeichnung Graphics Processing Unit. Um die Vor- und Nachteile eines Grafikprozessors erkennen zu können, muss man die Eigenschaften einer GPU verstehen. Bei genauer Betrachtung des Grafikchips fällt auf, dass der Großteil der Fläche des Chips für Recheneinheiten (ALUs) verwendet wird und nur ein kleiner Teil für Steuereinheiten und Datencache. Dies ist ein großer Gegensatz zum Aufbau einer konventionellen CPU aus dem PC Segment, die viel mehr Platz für Steuereinheiten und Datencache benötigt und somit ist der Platz für Recheneinheiten begrenzt. Daher sind diese in kleinerer Stückzahl in der CPU enthalten . Die Architektur einer GPU ist allerdings für rechenintensive, stark parallele Rechenoperationen konzipiert, welches dem typischen Anwendungsfall in 3D-Computergrafikberechnungen entspricht. Ein typischer Anwendungsfall der GPU ist die Umrechnung voneinander unabhängigen Polygonen (geometrische Körper) eines Polygonmodells in ein Pixelraster. Für die schnelle Umsetzung dieser Problemstellung ist eine parallele Rechnerarchitektur mit viel Rechenleistung am geeignetsten. Die Architektur einer GPU ist auf diese Form der Problemstellung angepasst, bei der nur wenige Operationen zur Flusskontrolle, Speicherzugriffsverwaltung und Datenpufferung (Cache) nötig sind.

Diese Struktur bedingt jedoch eine höhere Abstraktionsebene für den Anwender (Programmierer), da dieser selbst Kontroll- und Organisationsaufgaben übernehmen muss, um das volle Leistungspotenzial dieses Hardwaremodells zu entfalten.

Ein großes Problem bilden Latenzen (Verzögerungen) im Arbeitsspeicherzugriff (RAM-Zugriff), welche in CPUs durch großen Datencache verschleiert werden sollen. In dem GPU-Architekturmodell wird dieses Problem durch intelligentes Speichermanagement und durch einen hohen Grad an arithmetischen Operationen angegangen. Diese Architektur ist für einen hohen Grad an arithmetischen Operationen ausgelegt, die in einem viel größeren Verhältnis zu Speicheroperationen stehen. Somit ist auch ein relativ kleiner Cache von ein paar Kilobyte für die Rechenoperationen ausreichend. Des Weiteren steht ein Video-RAM mit einer sehr hohen Bandbreite zur Verfügung, wodurch Speichertransfervorgänge erheblich beschleunigt werden. Bei der Implementierung von Softwareelementen für einen Grafikprozessor ist daher immer zu beachten, dass arithmetische Operationen den Speicheroperationen vorzuziehen und komplizierte Speicherzugriffe zu vermeiden sind.[6]

[5] Für weitere Informationen siehe Literaturliste: [TIC64, TIC67]
[6] [NVI09a] S. 1 - 4

2.1.3 GPGPU - General Purpose Computation on GPU

Der Begriff GPGPU steht für den Einsatz von Grafikprozessoren für Berechnungen in unterschiedlichen, computergrafikberechnungsfremden Aufgabenbereichen. Dabei werden den üblichen Prozessoren Aufgaben abgenommen, um rechenintensive Vorgänge zu beschleunigen. Die modernen GPU-Architekturen eignen sich immer mehr für die Berechnung von allgemeinen Problemstellungen. Ihre theoretische Rechenleistung ist den in PCs eingesetzten, konventionellen Prozessoren weit voraus. Ihre Architektur ist für einen hohen Operations- und Datendurchsatz entwickelt worden. Herkömmliche CPUs sind hingegen für den allgemeinen Gebrauch in unterschiedlichsten Einsatzgebieten konzipiert.

Die heutigen Grafikprozessoren sind nicht für jede Einsatzmöglichkeit geeignet. Sie bringen jedoch einen großen Geschwindigkeitsvorteil bei der Implementierung von rechenintensiven, parallelen Algorithmen. Dahinter steht das Konzept einer parallelen Architektur und die darauf beschränkten Operationen. Daher werden im Bereich des GPGPU Grafikprozessoren als Coprozessoreinheit (über Zusatzkarte bzw. Grafikkarte) verwendet. Sie kann jedoch nicht ohne die Grundprozessoreinheit (CPU) eingesetzt werden, weil die CPU (Host) die Organisation übernimmt.[7]

2.1.4 Grundbegriffe in CUDA

In den Bezeichnungen des Systems im Zusammenspiel zwischen CPU und GPU wird von den Begriffen **Host** und **Device** gesprochen. Unter Host versteht man die CPU bzw. den Hauptrechner, analog dazu wird die Hardwareumgebung der Grafikprozessoreinheit als Device bezeichnet. Der im PC zur Verfügung stehende Arbeitsspeicher wird als Host-Memory und der Videospeicher (DRAM) des Grafikprozessors als Device-Memory bezeichnet.

Die parallele Berechnung und Datenverarbeitung ist in CUDA über **Threads** realisiert. Bei Threads handelt es sich um parallele Ausführungsstränge, welche in CUDA in hoher Stückzahl eingesetzt werden, um eine große Anzahl von Operationen zeitgleich auszuführen.

Diese Threads werden in Bündel von 32 Threads zusammengefasst und tragen die Bezeichnung **Warp**. Des Weiteren spielen, im Zusammenhang mit CUDA, **Threadblöcke** eine große Rolle. Dabei handelt es sich um ein Zusammenfassen von Threads bzw. mehreren Warps in Gruppen (Blöcken). Die Menge der Threadblöcke ist zusammenhängend in einem **Grid** organisiert, wobei dieses Grid wie ein Koordinatensystem zu verstehen ist, um eine genaue Zuordnung und Identifizierung der einzelnen Threadblöcke vornehmen zu können. Die Struktur des Grids, mit den enthaltenden Threadblöcken, dient zur Organisation der parallelen Ausführungsstränge. In den Threadblöcken erfolgt die einzelne Identifizierung der enthaltenen Ausführungsstränge(Threads).

Der für das Device vorgesehene Programmabschnitt muss in einer bestimmten Struktur entwickelt werden, damit der Grafikprozessor dieses Programm in der gewünschten Form parallel ausführen kann. Dieser Devicecode(Programmcode für die Hardware) wird als **Kernel** bezeichnet und beschreibt den kompletten Programmablauf auf dem Device. Bevor der Kernel durch den Host gestartet wird, ist das Maß des Grids (Anzahl Threads und Threadblöcke) festzulegen. Über diese Definition ist dem Kernel die tatsächliche Anzahl der Ausführungsstränge bekannt. Die Umsetzungsform dieses Programmcodes in dem CUDA-Programmiermodell ist im Abschnitt 3.1 zu

[7] [NVI09a] S. 1 - 4

finden. Die ideale Anzahl der Threads in einem Threadblock ist ein Vielfaches von 32. Für die Threadblöcke wird auf diesem Weg eine ganzzahlige Anzahl von Warps erzeugt. Die Wahl der Threadanzahl in dieser Form ist für den effizienten Einsatz von CUDA, auf Grund der Charakteristik der Hardware, verpflichtend. Die genaue Erläuterung dieser Charakteristik folgt in Abschnitt 2.2.4.

2.2 Hardware- und Verarbeitungsstruktur in CUDA

Das CUDA-Hardwaremodell steht in Verbindung mit dem Unified Shader Architekturmodell, welches erstmals in dem NVIDIA G80 Grafikprozessor implementiert wurde. Die entscheidende Charakteristik ist eine Verallgemeinerung der Architekturstruktur um höhere Flexibilität zu erreichen. Auch die aktuellen Grafikprozessoren des Herstellers NVIDIA sind an diesem Modell orientiert und unterscheiden sich lediglich in der Anzahl der Recheneinheiten, dem Speicher, der Taktraten und speziellen Modifikationen. Seit der Veröffentlichung des G80 GPU spezifiziert NVIDIA die Architektureigenschaften ihrer Grafikprozessoren nach den **Compute Capability** Standards. Diese Spezifikationen beschreiben die Grundelemente der Architektur, die Erweiterungen in den neueren Generationen, sowie die Spezifikationen für die Programmierung dieser Hardware. Eine detaillierte Ausführung dieser Spezifikationen folgt in Abschnitt 2.2.3 auf Seite 9.

2.2.1 CUDA-Hardwaremodell

Die früheren Grafikkartengenerationen sind für die pipelineförmige Verarbeitungsstruktur von Shaderelementen (Renderingelemente) konzipiert worden. Die in einer Pipeline angeordneten Shadereinheiten stellen die einzelnen Recheneinheiten dar, die für klassische 3D-Grafikberechnungen im Einsatz sind. Sie führen die einzelnen Bearbeitungsschritte durch, wobei jede einzelne Shadereinheit eine bestimmte Aufgabe übernimmt. Der Aufbau dieser Pipeline (siehe Abbildung 2.1) zeigt die Instruktionsabfolge der Shaderpipeline für die Abarbeitung der einzelnen Berechnungsschritte.

Nach diesem Schema wird die Berechnung der 3D-Daten im klassischen Sinne organisiert. Der in Abbildung dargestellte Zusammenhang ist eine schematische Darstellung einer Shaderpipeline und kann im Detail von der tatsächlichen Hardwareimplementierung abweichen.

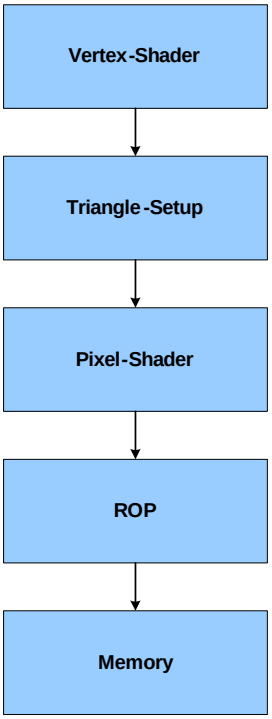


Abbildung 2.1: Klassisches Modell der Shaderpipeline (Grafik-API DirectX 9.0)

Verarbeitungsweise: Shaderpipeline[a]

1. Der Vertex-Shader führt die Berechnung von Geometrieelementen durch.
2. Das Triangle-Setup wandelt die Eckpunkte in primitive Modelle um (z.B. Linien, Dreiecke und Rechtecke)
3. Berechnung der Pixeldaten im Pixelshader
4. Der Raster Operator (ROP) führt Sichtbarkeitsprüfungen und eine mögliche Anti-Aliasing Operation durch.
5. Ablage der Daten in den Bildspeicherpuffer

[a] Quelle: [WA06]

Die Einführung des G80 Grafikprozessors bedeutet eine entscheidende Umstellung der Architektur, bei dem das ursprüngliche Hardwaremodell durch ein Unified Shader Modell ersetzt wird. Das Unified Shader Modell nimmt eine Verallgemeinerung der ursprünglichen Recheneinheiten mit festen Aufgaben in der Shaderpipeline vor. Skalare Recheneinheiten übernehmen in dem Unified Shader Modell die Aufgaben des Vertex- oder Pixelshaderrecheneinheit und sind somit flexible Recheneinheiten für den universellen Einsatz. Auf diesem Weg ist eine höhere Auslastung der Recheneinheiten möglich, da jede Recheneinheit für jede Aufgabe geeignet ist. Diese Eigenschaft benötigt eine neue Organisationsstruktur bei der Abarbeitung der Instruktionen, die im Abschnitt 2.2.4 im Detail erläutert ist.

Der Aufbau der universellen Architekturstruktur organisiert sich, wie folgt:

Der globale Speicher (DRAM), auch als Device-Memory bezeichnet, ist über ein GDDR3-Speicher-Interface mit einer größeren Anzahl von Multikernprozessoren verbunden. Die in der Hardware implementierten Multiprozessoren oder auch Streaming Multiprocessors (SM) dienen zur Verarbeitung der in Threadblöcken zusammengefassten Ausführungsstränge (Threads). Jeder Threadblock wird von einem Multiprozessor verarbeitet. Die Threadblöcke werden durch den globalen Block Scheduler (Regelung der zeitlichen Abfolge) auf die Multiprozessoren verteilt. Eine schematische Darstellung der Architektur ist der Abbildung 2.2 zu entnehmen.

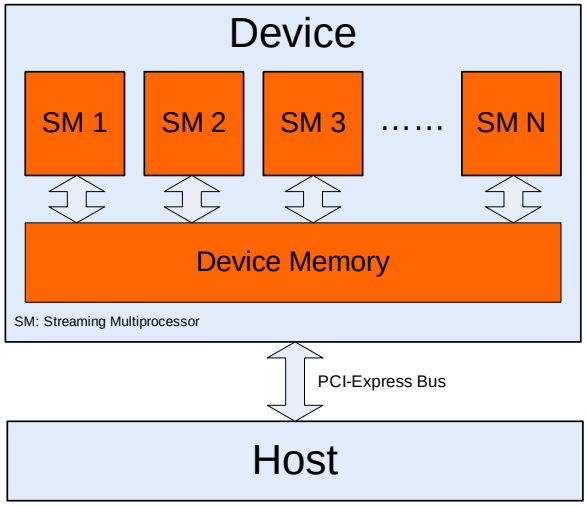
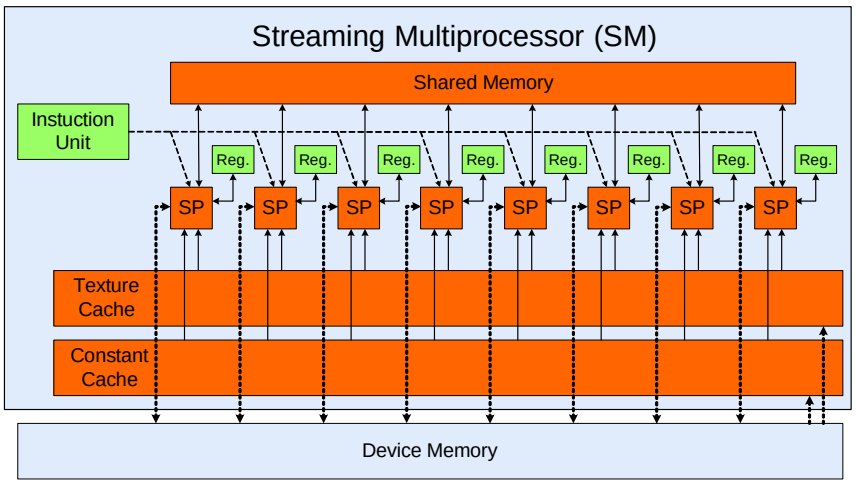


Abbildung 2.2: CUDA-Hardwaremodell

Abbildung orientiert an Quelle: [NVI09a] S.75

2.2.2 Multiprozessormodell

Die Multiprozessoren sind in ihre Grundstruktur seit der G80 bis zur GT200 Grafikprozessorgeneration nur geringfügig modifiziert worden. Die Abbildung 2.3 zeigt das Grundmodell eines Streaming Multiprocessors mit seinen Elementen. Die Grundelemente sind eine bestimmte Anzahl von mehreren skalaren Recheneinheiten (Streaming Processors), wobei die Anzahl von 8 bisher nie verändert worden ist und jeder Streaming Processor seinen eigenen Registersatz besitzt. Die Streaming Processors sind in der Lage, einen gemeinsamen Read/Write-Cache mit der Bezeichnung Shared Memory zu nutzen. Jeder Multiprozessor besitzt darüber hinaus noch eine Instruktionseinheit, die Steuerungsaufgaben mit multiplen Threads im Multiprozessor übernimmt.



- ▷ Streaming Multiprocessor (SM)
- ▷ Multithreaded Instruction Unit
- ▷ Streaming Processor (SP)
- ▷ Shared Memory
- ▷ Registers (Reg.)

Abbildung 2.3: Streaming Multiprocessor
Abbildung orientiert an Quelle: [NVI09a] S.75

Zusätzlich dient ein Constant Cache zur Beschleunigung der Zugriffe auf einen Read-Only Speicher-

bereich für den Konstantspeicher im Device-Memory (Constant Memory Space). Ebenfalls ist ein Texture Cache für die Beschleunigung der Speicherzugriffe auf einen weiteren Read-Only Speicherbereich für Texturenspeicher (Texture Memory Space) implementiert. Die Deklaration dieses Speichers erfolgt durch den Host in der Programmierung und kann zum Ausführungszeitpunkt des Kernels nur gelesen werden. Für Speicherzugriffe auf den lokalen und globalen Speicher ist kein Cache vorhanden, somit sind die Zugriffe auf diese Speicherbereiche mit Latenzen behaftet.

2.2.3 Compute Capability Specifications

Im CUDA Programming Guide[8] ist die genaue Erläuterung der Spezifikationsbezeichungen angegeben.

Die zur Zeit der Erstellung dieser Arbeit existierenden Spezifikationen sind über die Versionen 1.x spezifiziert und liegen bisher in der Version 1.0 bis 1.3 vor. Die Stelle vor dem Punkt bestimmt die Grundarchitektur des Multiprozessors. Die Stelle nach dem Punkt ist die Versionsnummer der Modifikationen für die unterschiedlichen Prozessorgenerationen. Diese Modifikationen basieren auf der Grundarchitektur, dargestellt in Abbildung 2.3.

In Abschnitt 2.2.2 ist die Grundarchitektur des Multiprozessors dargestellt. Durch die Compute Capability Specifications sind die genaueren Details der einzelnen Elemente festgelegt. Eine Liste der Spezifikationen ist im Anhang A angegeben und wird in den folgenden Abschnitten wiederholt als Bezugsquelle verwendet.

2.2.4 Parallele Berechnung mit SIMT

Durch die Verwendungen der Streaming Multiprocessors in ihrer allgemeinen Form können diese in unterschiedlicher Anzahl auf Grafikprozessoren eingesetzt werden. Der Verarbeitungsprozess der Ausführungsstränge in Threadblöcken wird in einem Multiprozessor durchgeführt. Jeder Multiprozessor ist dafür verantwortlich, einen oder mehrere Threadblöcke zu verarbeiten. Das im Abschnitt 2.1.4 erläuterte Grid trägt zur Generalisierung der Architektur durch die strukturelle Anordnung der Threadblöcke bei.

Der im Grafikprozessor integrierte Blockscheduler verteilt die einzelnen Threadblöcke im Grid auf die Multiprozessoren, um diese von den einzelnen Multiprozessoren verarbeiten zu lassen. Die tatsächliche Anzahl von gleichzeitig auf dem Multiprozessor aktiven und wartenden Threadblöcken ist über ihre Charakteristiken bestimmt. Die Ausführungscharakteristiken der Threadblöcke drücken sich über Speicherbedarf und Warpanzahl aus, die in Bezug zu den Hardwarespezifikationen der Multiprozessoren gesetzt werden. Jedoch sind alle Threadblöcke voneinander unabhängig. Diese Eigenschaft ermöglicht es ihnen, in unbestimmter Reihenfolge verarbeitet zu werden. Daraus resultiert eine Veranlassung der Verarbeitung sowohl in paralleler als auch sequenzieller Abfolge, und das zu beliebigen Zeitpunkten. Die Anzahl der integrierten Multiprozessoren ist entscheidend dafür, in welcher Anordnung die Blöcke (parallel und/oder sequenziell) auf den Multiprozessoren zu bearbeiten sind. Der Verarbeitungsprozess ist auf diesem Weg sehr flexibel und skalierbar gehalten, veranschaulicht in Abbildung 2.4. In dem CUDA Programming Guide [NVI09a] wird diese

[8] [NVI09a] S.14

Anordnung als „Automatic Scalability"[9] bezeichnet, die als automatische Skalierung der Threadblöcke verstanden wird. Diese automatische Aufteilung der Blöcke ermöglicht es jedem für CUDA entwickelten Programm, auf jeder CUDA-Hardware lauffähig zu sein, unabhängig von der Anzahl der eingesetzten Multiprozessoren. Die Hardware unterliegt der einzigen Auflage, nach den entsprechenden Standards (Abschnitt 2.2.3) spezifiziert zu sein, für das dieses Programm entwickelt wurde. Die entwickelten CUDA-Softwareelemente sind durch die Gewährleistung der Abwärtskompatibilität der Compute Capability Specifications auch auf zukünftiger Hardware lauffähig. Unterschiede der einzelnen Grafikprozessoren und Grafikprozessorgenerationen spiegeln sich in den Ausführungszeiten der Kernel wieder. Eine geringere Anzahl von Multiprozessoreinheiten bewirkt eine entsprechende Zunahme der sequenziellen Abarbeitung der Threadblöcke.

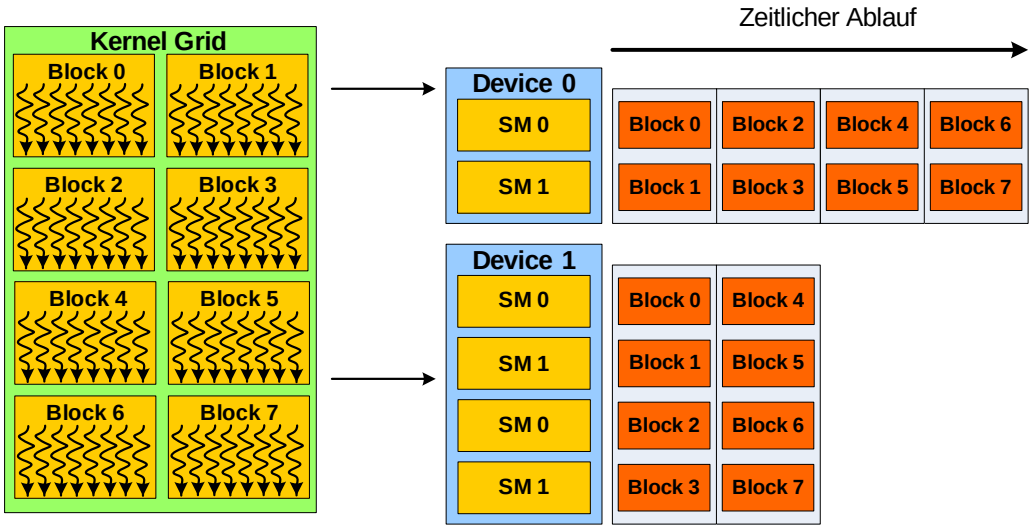


Abbildung 2.4: Skalierbarkeit des Programmablaufs

Quelle Abbildung: [NVI09a] S.72

Die Ausführungsform der Instruktionen im Multiprozessor ist im Folgenden anhand des SIMT-Prinzips erläutert.

Der Begriff SIMT[10] ist eine von NVIDIA neu eingeführte Innovation und steht für „Single Instrunction Multiple Thread". Dabei handelt es sich um eine speziell für CUDA entwickelte Rechnerstruktur für die Instruktionsverarbeitung mit parallelen Ausführungssträngen. Sie liegt der SIMD Technologie zu Grunde, die bereits in Abschnitt 2.1.1 angesprochenen wurde.

Ein bereits erwähntes Schlagwort in CUDA sind Warps. Warps stellen die Bündel von 32 Threads dar. Der Grund für diese Bündelung beruht auf der Instruktionsverarbeitung auf den Streaming Multiprocessors. Bei der Ausführung eines Threadblocks werden die Warp-Bündel eines Threadblocks nacheinander auf dem Multiprozessor abgearbeitet. Für jeden Warp wird die gleiche Operation simultan zur Ausführung gebracht, weshalb diese Verarbeitungsart als eine SIMD-Operation angesehen wird. Die skalaren Recheneinheiten (Streaming Processors) des Multiprozessors verarbeiten dabei jeweils einen Thread separat, jedoch führen alle Streaming Prozessors die gleiche

[9] [NVI09a] S.72
[10] [NVI09a] S.72 -74

Operation zur gleichen Zeit durch. Der im Multiprozessor integrierte Warp-Scheduler definiert die Abarbeitungsreihenfolge der einzelnen Warps in einem Threadblock.

Durch eine genauere Analyse des SIMD-Architekturmodells wird ein gravierender Unterschied deutlich. Die SIMD Technologie basiert auf einer starren Struktur. Die SIMD-Einheiten liegen oftmals als eine Vektorprozessoreinheit[11] vor, wogegen die CUDA-Multiprozessoren aus einer arrayförmigen Anordnung von einzelnen skalaren Recheneinheiten bestehen. Hinzu kommt, dass die einzelnen SIMD-Instruktionen in einem sequenziellen Programmiermodell organisiert sind. Infolgedessen müssen die Datensegmente der Datensätze durch den Anwender (Programmierer) selbst zusammengefasst, vorbereitet und organisiert werden. Diese Problematik wird in CUDA durch sein paralleles Programmiermodell mit einer hohen Anzahl von Ausführungssträngen in Angriff genommen. Die parallel orientierte Programmstruktur ermöglicht es, den in einem Warp zusammen gefassten Threads zur gleichen Zeit eine Operation auf einem Multiprozessor zur Ausführung zu bringen. Die Threads werden automatisch in Warps zusammengefasst und in der Hardware verarbeitet. Die Organisationsaufgaben werden folglich vom Anwender (Programmierer) fern gehalten. In dieser Struktur ist es jedem einzelnen Thread auch möglich, unterschiedliche Ausführungswege zu nehmen und infolgedessen unterschiedliche Operationen auszuführen, die theoretisch zum selben Zeitpunkt erfolgen würden. Unterschiedliche Programmpfade stellen jedoch eine diffizile Angelegenheit in der SIMT-Instruktionsverarbeitung dar. Diese Problematik bewältigt die SIMT Technologie, indem sie in solchen Fällen über die Option verfügt, Threads eines Warps vorübergehend abzuschalten. Es werden immer nur die Threads gleichzeitig ausgeführt, bei denen die Ausführung derselben Operation ansteht. Jeder Warp wird so oft erneut ausgeführt, bis alle Threads und die dafür benötigten Operationen abgearbeitet sind. Diese Organisationsstruktur gestattet unterschiedliche Ausführungswege der Ausführungsstränge durch eine Verlängerung der Ausführungszeit.

[11] Prozessoreinheit zur Durchführung einer Instruktion auf mehrere Datenelemente eines Datensatzes (Datenvektor)

Kapitel 3

Programmierumgebung

In Abschnitt 2.2 wird die Struktur des parallelen NVIDIA CUDA-Hardwaremodells veranschaulicht. Um dieses Hardwaremodell in einer überschaubaren Form zu handhaben, bedarf es eines durchdachten Programmiermodells, um für diese Hardware hoch qualitative Softwareelemente zu entwickeln. Die von NVIDIA zur Verfügung gestellte Programmierumgebung wird, seit der Veröffentlichung der ersten Version für den G80 Grafikprozessor, stetig weiterentwickelt und ermöglicht eine professionelle Entwicklung von Software für dieses Hardwaremodell. Die folgenden Abschnitte präsentieren das Programmiermodell, die zugehörige Umgebung mit den Application Programming Interfaces, sowie die neuen Möglichkeiten und Problematiken dieses Modells. Die in diesem Abschnitt aufgeführten Programmierbeispiele sind in „C for CUDA" gehalten. Dabei handelt es sich um eine Erweiterung der Programmiersprache C für die Softwareentwicklung in CUDA.

3.1 Einführung Programmiermodell

Die Einführung in die Programmierung von CUDA-Softwarelementen beinhaltet eine ausführliche Erläuterung der Organisationsstruktur anhand von Quellcode-Beispielen in „C for CUDA", um eine Verdeutlichung der Architektureigenschaften und ihrer Verwendung in der Programmierung zu erreichen.

3.1.1 Die Kernel

Der erste Schritt in der parallelen Programmierung mit CUDA ist die Erstellung von Devicecode (Programmcode für den Grafikprozessor).

Bei den als Kernel bezeichneten Programmcodeelementen handelt es sich um C-Funktionen, die in N-facher Anzahl nebenläufig auf dem Grafikprozessor zur Ausführung kommen. Die Anzahl N ist hierbei die Zahl der parallelen Ausführungsstränge (Threads). Die Identifikation(ID) des aktuellen Threads in der Funktion ist über die im Kernel eingebundene Struktur (struct in C) möglich. Jeder ausgeführten Funktion ist die eigene Thread-ID bekannt. Die Kernel-Funktionen sind mit dem Schlüsselwort ___global___ gekennzeichnet und können anhand dieser Kennzeichnung identifiziert werden. Die Menge der verwendeten Threads wird über ein neu eingeführtes Syntaxkonstrukt mit den Symbolen <<<...>>> definiert, die im Standard C-Syntax bisher nicht zu finden sind.

Für das Verständnis ist an dieser Stelle ein erstes Programmcodebeispiel aufgeführt:

Beispielcode 3.1 Beispiel Kernel

```
__global__ void Mul(float* in1,float* in2,float* out)
{
   int index = threadIdx.x;
   out[index] = in1[index]*in2[index];
}
int main(void)
{
   ..... // Definitionen von in1, in2, out
   int nThreads = 512;
   // Kernel ausführen - nThreads = Anzahl der Threads
   Mul<<< 1 , nThreads >>>(in1, in2, out);
   .....
}
```

Dieser Kernel führt eine elementweise Multiplikation der einzelnen Fließkommawerte in einem Array der Länge N durch. Jeder einzelne Thread ist für eine Multiplikation verantwortlich. Die Erkennung und Handhabung der einzelnen Threads ist über die Indizierung (ThreadIdx.x - Strukturelement) ermöglicht.

3.1.2 Struktur der Threadhierarchie

Die Kernelausführung ist organisiert in der Definition des Grids und der Threadblöcke. In Hierarchiestufen betrachtet, gibt das Grid die Anzahl der Threadblöcke vor, die in einer zweidimensionalen Ebene organisiert werden.

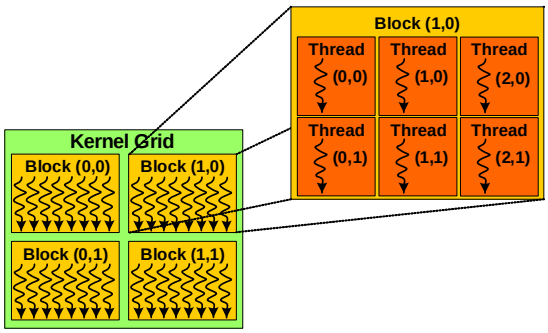


Abbildung 3.1: Threadhierarchie in zweidimensionaler Ebene

Abbildung orientiert an Quelle: [NVI09a] S.15

Die enthaltenen Threadblöcke lassen eine Organisation der Threads in einer ein-, zwei- oder dreidimensionalen Form zu. Diese Organisationsstrukturen ermöglichen die direkte Ansteuerung der einzelnen Threadblöcke und Ausführungsstränge über ihre Indizierung, dargestellt in Abbildung 3.1.

Die Organisationsstruktur der Threadblöcke in unterschiedlichen Dimensionen ermöglicht eine intuitive Ansteuerung von ein- oder mehrdimensionalen Arrays.

Diese Ansteuerung geschieht in den Threadblöcken über die Struktur **threadIdx** mit drei enthaltenden Elementen für die x-, y- und z-Ebene. Die Definition des Threadblocks (siehe Codebeispiel 3.2: dimBlock) erfolgt über einen, in der Programmierumgebung, vordefinierten Vektordatentyp. Dieser ist in Form einer Struktur mit Elementen für die drei Dimensionen implementiert.

Dieses Beispiel veranschaulicht die Organisation der Threadhierarchie mit einem einzigen Threadblock:

Beispielcode 3.2 Einfache Threadhierarchie

```
#define N 16 // Anzahl der Datenelemente in x- und y-Richtung
__global__ void Mul(float** in1,float** in2,float** out)
{
   int x = threadIdx.x;
   int y = threadIdx.y;
   out[x][y] = in1[x][y]*in2[x][y];
}
int main(void)
{
   ..... // Definitionen von in1, in2, out
   // Grid = 1 - Anzahl der Threads = 16*16 = 256
   dim3 dimBlock(N, N); // Definition des Threadblocks
   Mul<<< 1 , dimBlock >>>(in1, in2, out);
   .....
}
```

Bei der Dimensionierung des Threadblocks ist jedoch zu beachten, dass dieser in x- und y-Richtung auf den Maximalwert von 512, sowie in z-Richtung auf 64 begrenzt ist. Ebenso ist zu berücksichtigen, dass ein Threadblock eine Anzahl von 512 Threads niemals überschreiten darf (siehe Anhang A.1.1). Andernfalls erfolgt ein direkter Abbruch der Kernelausführung. Es ist daher zu empfehlen, bei der Dimensionierung immer den Spezifikationen der Architektur (Anhang A) Beachtung zu schenken, um Fehlerquellen dieser Art zu vermeiden.

Um eine komplette Ausnutzung der vorliegenden Hardware zu erreichen, ist die Organisation in einem Grid mit mehreren Threadblöcken zu empfehlen. Die Definition des Grids erfolgt über eine weitere Variable eines dreidimensionalen Vektordatentyps. Die Ansteuerung der einzelnen Threadblöcke im Grid ist über die Strukturen **blockIdx** und **blockDim** vorzunehmen. Hierbei handelt es sich ebenfalls um dreidimensionale Vektordaten. Die Variable **blockIdx** ist für die Indizierung der Threadblöcke und **blockDim** für die Anzahl der Threads in x-, y- und z-Richtung verantwortlich. Über die Variable **gridDim** besteht die Option, die Anzahl der Threadblöcke im Grid in x- und y-Richtung abzurufen. In der Grid-Dimensionierung ist die Limitierung des Grid auf eine zweidimensionale Form mit einer maximalen Größe von 65535×65535 zu berücksichtigen.

Die Übergabe der definierten Grid- und Block-Dimensionierungen an den Kernel erfolgt, wie bereits erwähnt, über das Syntaxkonstrukt `<<< GridDim , BlockDim >>>`.

Erweiterung des Codebeispiels 3.2:

Beispielcode 3.3 Komplexere Threadhierachie

```
#define N 512 // Anzahl der Datenelemente in x- und y-Richtung
__global__ void Mul(float** in1,float** in2,float** out)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < N && y < N) // Falls Threadanzahl > N*N
      out[x][y] = in1[x][y]*in2[x][y];
}
int main(void)
{
   ..... // Definitionen von in1, in2, out
   // Anzahl der Threads pro Block = 16*16
   dim3 dimBlock(16, 16);
   // Definition des Grids -- Mindestanzahl an Threads N*N = 512*512
   // Mindestanzahl erreicht durch Aufrundung ((N + 15)/16 >=N)
   dim3 dimGrid( (N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y - 1) /
       dimBlock.y );
   Mul<<< dimGrid , dimBlock >>>(in1, in2, out);
   .....
}
```

Die oben angegebene Form der Dimensionierung definiert mehrere Threadblöcke mit der einheitlichen Größe von 16×16. Über die Definition des Grids werden die elementweisen Multiplikationen der $N \times N$ großen Matrizen auf mehrere Blöcke verteilt, wobei sich die Anzahl der Threadblöcke über die Größe der Datenmenge festlegt. Jeder Block arbeitet einen Bereich der Matrix[1] (Größe 16×16) nebenläufig ab. Auf diesem Weg wird die parallele Abarbeitung in Threadblöcken organisiert, wobei die einzelnen Threadblöcke zu einem beliebigen Zeitpunkt komplett unabhängig ausgeführt werden können. Diese Eigenschaft ist eine im vorherigen Abschnitt 2.2.4 beschriebene Voraussetzung des Hardwaremodells für die automatische Skalierung der Threadblöcke. Durch die überschaubare Implementierung im Programmiermodell ist diese Problematik auf einfachstem Wege zu handhaben.

Für die Threads in einem Block besteht die Möglichkeit auf einfacher Ebene untereinander zu kooperieren. Diese Kontrolleigenschaft ist für die Koordinierung auf einem gemeinsamen Speicher im Block (der Shared Memory) zur Verfügung gestellt. Die Threads in einem Block sind in der Lage über den Shared Memory zu kommunizieren. Der Speicherzugriff kann über eine eingebundene Synchronisationsstufe organisiert werden. Der Befehl `__syncthreads()` stellt eine Barriere für die ablaufenden Threads dar. Die Überquerung dieser Barriere ist nur gemeinsam möglich (alle Threads in einem Block). Die weitere Ausführung der bereits eingetroffenen Threads wird unterbunden bis alle Threads diesen Punkt erreicht haben.

3.1.3 Speicherkonzept

Die Umsetzung des Programmiermodells auf der Hardware benötigt unterschiedliche Hierarchiestufen und Speicherbereiche in der Speicherorganisation. Die Speicherhierarchie des CUDA-Programmiermodells ist in drei markante Ebenen aufgeteilt. Der threadeigene Speicherbereich trägt die

[1] Bei einer Größe von $N \leq 16$ wird nur ein einziger Threadblock ausgeführt

Bezeichnung „Local Memory". In diesem Fall handelt es sich um den privaten Speicher für jeden einzelnen Thread. Die zweite Ebene drückt sich im „Shared Memory" aus. Dieser Speicher kann durch alle Threads in einem Block gemeinsam genutzt werden. Die dritte und letzte Stufe beschreibt die globale Ebene. Aus diesem Grund wird dieser Speicher als globaler Speicher bezeichnet und dient der gemeinsamen Nutzung. Die systematische Darstellung ist in Abbildung 3.2 veranschaulicht.

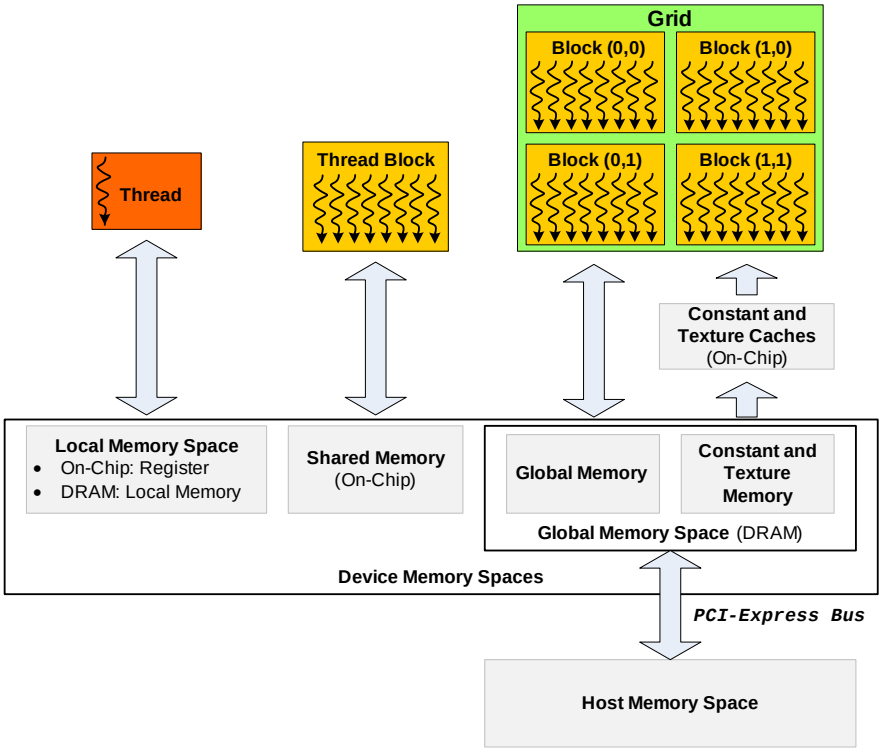


Abbildung 3.2: Speicherorganisation und Speicherhierarchie

Abbildung orientiert an Quelle: [NVI09a] S.11 / [NVI09b] S.19

Die globale Stufe der Speicherhierarchie ist jedoch noch in spezielle Speicherbereiche unterteilt, die jeweils für bestimmte Einsatzfälle optimiert sind. Außerdem repräsentiert der globale Speicher die Kommunikationsschnittstelle mit dem Host, wobei der tatsächliche Datentransfer über einen PCI-Express-Bus[2] realisiert ist. Dem Device ist ein direkter Speicherzugriff auf die Host-Speicherbereiche nicht möglich. Aus diesem Grund wird für den Programmablauf auf dem Device ein Datentransfer aus dem Host-Speicherbereich in den Speicherbereich des Devices vorausgesetzt. Um eine Realisierung des Datenaustausches zu ermöglichen, stellt die API Funktionen für den Datenaustausch zwischen dem Device und dem Host zur Verfügung. Dabei besitzt der Host Lese- und Schreibrechte auf die globalen Speicherbereiche (Global Memory, Texture Memory und Constant Memory) des Devices.

Die strukturelle Aufteilung der physikalischen Speicherbereiche ist hierbei über den DRAM und einem direkt im Chip implementierten Speicher realisiert. Durch die Wahl ihrer Implementierungsform weisen die einzelnen Speicherbereiche unterschiedliche Charakteristiken auf.

[2]PCIe: „Peripheral Component Interconnect Express"

Der lokale Speicherbereich beinhaltet zwei unterschiedliche Formen. Bei den On-Chip Registern handelt es sich um einen direkt im Chip ansteuerbaren Speicherbereich auf der lokalen Ebene. Dieser Speicher ist in seiner Größe begrenzt und deshalb über den DRAM erweiterbar. In der Umsetzung im Programmiermodell ist die Aufteilung der Speicherorganisation auf ON-Chip und DRAM nicht direkt beeinflussbar.

Der Shared Memory mit Zugriffsberechtigung auf Threadblockebene ist ein zusätzlicher Speicherbereich, der direkt im Chip implementiert ist. Dieser Speicher trägt den Geschwindigkeitsvorteil der direkten Zugriffsmöglichkeit eines On-Chip-Memory. Die Zugriffsgeschwindigkeit des Shared Memory ist mit den Registern vergleichbar, solange keine Kollisionen in den Speicherzugriffen zwischen den Threads vorliegen. Die Problematik der Speicherzugriffskonflikte ist im Abschnitt 3.4.3 genau diskutiert. Die Speicherorganisation auf Threadblockebene ist im Programmiermodell eingebunden. Die Nutzung und Organisation dieses Speicherbereichs muss der Anwender selbst übernehmen.

Die globale Speicherebene beinhaltet eine weitere Aufteilung der Elemente dieses Speicherbereichs. Dabei wird dieser in einen Speicherbereich mit Lese- und Schreibrechten im Device und in einen Speicherbereich nur mit Leserechten separiert. Der globale Read/Write-Speicherbereich unterliegt der intensivsten Nutzung der globalen Speicherformen. Dieser Speicherbereich verfügt dennoch über keinen Cache im Speicherzugriff. Hingegen ist in der Verwendung des Read-Only Texture- und Constant- Speicherbereichs ein Cache im Speicherzugriff eingebunden.

Die gravierenden Unterschiede der Speicherbereiche spiegeln sich in der Zeitverzögerung im Speicherzugriff wieder. Der Zugriff auf die ON-Chip-Speicherbereiche erfolgt prinzipiell ohne Zeitverzögerung. In diesem Zusammenhang ist allerdings zu beachten, dass durch Zugriffskonflikte auf den Shared Memory Zugriffsserialisierungen eintreten, die Verzögerungen zur Folge haben und daher zu Geschwindigkeitseinbußen führen. Prinzipiell ist der Zugriff auf den DRAM mit einem Delay von 400 bis 600 Takten behaftet, welches einen erheblichen Geschwindigkeitsunterschied zu den ON-Chip-Speicherbereichen bedeutet. Trotzdem ist eine Minimierung des Delayeinflusses oder sogar eine komplette Verschleierung dieses Delays über die in Abschnitt 3.4.1 dokumentierte Methode realisierbar.

Die für den Programmierer entscheidenden Informationen liegen in einer strukturierten Form in Tabelle 3.1 vor.

Speicherart	ON/OFF Chip	Cached	Zugriffsform	Lebensdauer	Speicherbegrenzung
Register	On	—	READ/WRITE	Thread	***RA*** * 32-bit / Multicore
Local	Off	No	READ/WRITE	Thread	16 KB / Thread
Shared	On	—	READ/WRITE	Block	16 KB / Multicore
Global	Off	No	READ/WRITE	Host-Allokation	–
Constant	Off	Yes	READ	Host-Allokation	64 KB
Texture	Off	Yes	READ	Host-Allokation	–

Tabelle 3.1: Detailinformationen des Device-Speicherbereichs

RA: Entspricht der Anzahl von 8192 bzw. 16384 32-Bit Registern[3] — (Tabelle orientiert an Quelle: [NVI09b] S. 20)

[3] Anzahl der Register ist abhängig von der Compute Capability Specification (siehe Anhang 2.2.3). Version 1.0: 8192 , Version 1.2: 16384

3.1.4 Programmablauf und Kompilierung

Ein effektiv durchstrukturierter Programmablauf ist für dessen Organisation von essenzieller Bedeutung. Die Organisationsstruktur des Applikationsablaufs ist in Abbildung 3.3 dargestellt. Die Aufgabe für die Organisation der Kontrollstrukturen in der Applikation unterliegt dem Host. Dieser führt die Initialisierungs-, Kommunikations- und Ausführungsprozesse durch. In der Abbildung 3.3 ist zu erkennen, dass die Ablauforganisation in einem sequentiell abgearbeiteten Thread im Host erfolgt.

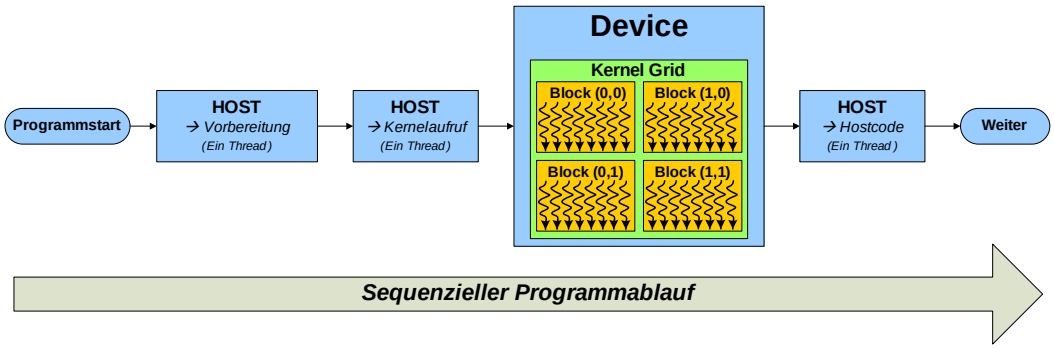


Abbildung 3.3: Programmablauf

Vor der Kernelausführung erfolgt eine Initialisierung des Devices und eine Vorbereitung des benötigten Speichers im globalen Speicherbereich des Devices. Auf Grund der getrennten Speicherbereiche findet ein Datenaustausch zwischen Host und Device statt, wobei diese Transaktion durch den Host kontrolliert wird. Nach Abschluss aller Vorbereitungsprozesse erfolgt der Aufruf des Kernels. Diese Ausführung erfolgt asynchron zum Host. Der Host führt lediglich die Initialisierung (den Start) durch. Diese Eigenschaft ist in der Programmierung zu berücksichtigen. Es besteht die Möglichkeit, mehrere Kernel direkt hintereinander zu starten. Die Ausführung auf dem Device erfolgt allerdings sequenziell. Nach den, zum Zeitpunkt der Erstellung dieser Arbeit existierenden, Compute Capability Specifications ist eine nebenläufige Ausführung von mehreren Kerneln nicht möglich. Nach Beendigung der Verarbeitungsprozesse auf dem Device erfolgt eine Übertragung der Ergebnisdaten zurück auf den Host-Speicherbereich, um mit diesen in der Applikation weiter zu verfahren. Der Datenaustausch ist in den üblichen Implementierungsformen synchron zwischen Host und Device organisiert. Alternativ steht allerdings auch eine asynchrone Organisationsform in mehreren Ausführungsströmen (In Cuda: Streams) zur Verfügung. Über diese Methode ist seit dem Compute Capability Standard in der Version 1.1 eine Überschneidung der Kopier- und Ausführungsvorgänge auf dem Device möglich. Die soeben beschriebene Implementierungsmethode ist in Abschnitt 3.4.5 dokumentiert.

Neben der Softwareentwicklung auf der Hochsprachenebene in C stehen für die Devicecode Entwicklung eigens von NVIDIA entwickelte Instruktionssätze auf der Assembly-Ebene zur Verfügung. Diese Instruktionssätze tragen die Bezeichnung **PTX**[4]**.** Die Programmcodeentwicklung auf der Assembly-Ebene führt zu einer höheren Effizienz des Codes. Sie ist jedoch mit einem wesentlich höheren Aufwand verbunden. Die direkte Programmierung des PTX-Codes ist in dieser Arbeit nicht weiter beschrieben.

[4]Parallel Thread Execution

Der Kompilierer-Treiber NVCC[5] übernimmt die Organisation für die Umsetzung des Devicecodes in ausführungsfähige Elemente. Die Softwareentwicklung für CUDA findet im Rahmen dieser Arbeit in der Programmiersprache C/C++ mit den im folgenden Abschnitt 3.2 erläuterten Erweiterungen statt. Der NVCC Kompilierer-Treiber realisiert eine Umsetzung des in C erstellten Devicecodes in die PTX-Assembly-Form, sowie in die direkte Binärcode-Form (binary files: cubin object). Die Wahl der Kompilierung in PTX-Code ermöglicht einen weiteren Optimierungsschritt und daher einen schnelleren Programmablauf. Des Weiteren ist dieser Code in PTX- oder Binärform Plattformunabhängig einsetzbar und die „Cuda Driver API"[6] ermöglicht eine direkte Ausführung des PTX-Codes.

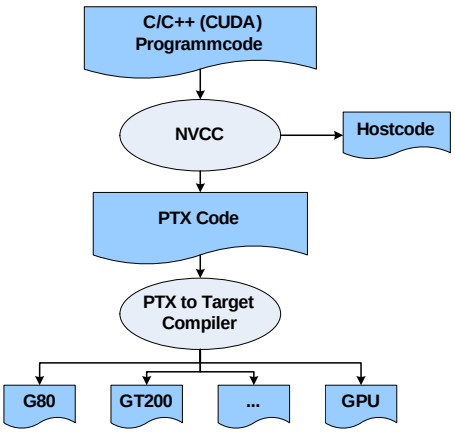


Abbildung 3.4: Kompilierungsvorgang

Abbildung Quelle: [NVI08] S.19

Der komplette Kompilierungsprozess ist in Abbildung 3.4 dargestellt. In diesem Prozess werden die Hostcodeelemente durch den NVCC abgespalten und der Devicecode in PTX- bzw. Binärcode umgesetzt. Für die Ausführung des PTX-Codes ist eine Umsetzung in einen hardwareausführbaren Code vorausgesetzt.

Die in dieser Arbeit eingesetzte Methode für die Erzeugung von Applikationen findet durch die Umsetzung des Programmcodes in Objektdateien[7] statt. Anschließend erfolgt eine Zusammenführung in einer ausführbaren Datei(Executable). Dies geschieht mittels des NVCC-Aufrufparameter „`-c`". Die Hostcodeelemente werden durch den C/C++ Kompilierer der verwendeten Entwicklungsumgebung in die benötigte Form umgesetzt. Im Kompilierungsprozess ist die korrekte Auswahl der Zielarchitektur (Compute Capability Mode) zu beachten, um die komplette Bandbreite der in der Hardware implementierten Funktionalitäten zu nutzen. Die Auswahl des Compute Capability Standards in der Version 1.0 bis 1.3 findet seine Umsetzung über den Parameter „`-arch [sm_10 .... sm_13]`". Die Definition der Anzahl der eingesetzten Register pro Thread ist von entscheidender Bedeutung und über den Parameter „`-maxrregcount=N`" realisiert. Auf diesem Weg wird eine Aufteilung der vorhanden Register im Multiprozessor auf die einzelnen Threads vorgenommen. Zusätzlich ist über den Parameter „`-use_fast_math`" eine forcierte Auswahl der beschleunigten, jedoch weniger akkuraten, mathematischen Funktionen[8] der CUDA-Laufzeitumgebung zu errei-

[5] NVIDA eigene Innovation für die Cuda-Programmcodeentwicklung , Quelle: [NVI09a] S.16
[6] Ist in dieser Arbeit nicht im Einsatz: siehe Abschnitt 3.2
[7] Unter dem Betriebssystem Microsoft Windows mit der Dateiendung „`.obj`" , ansonsten „`.o`"
[8] Funktionsbezeichnung: ___FUNCTION() – [NVI09a] Appendix C - S.119

chen.

Die für die CUDA-Softwareentwicklung wichtigen Parameter sind in diesem Abschnitt erläutert. Für die Vertiefung dieser Thematik ist die detailierte Dokumentation im CUDA Programming Guide ([NVI09a] S.16) und im NVCC User Manual ([NVI09c]) zu Hilfe zu nehmen.

3.2 CUDA Application Programming Interfaces

3.2.1 Überblick

Die Ausführung von Programmcodeelementen auf dem Device erfolgt über den auf dem Betriebssystem installierten Treiber der Coprozessor-Zusatzkarte (meist Grafikkarte). Die Firma NVIDIA stellt auf der Treiberebene eine Low-Level API(Application Programming Interface), die CUDA DRIVER API, in der Programmiersprache C zur Verfügung. Eine in der High-Level Ebene umgesetzte API beinhaltet eine Laufzeitumgebung, die unter Einsatz der in der Hochsprache C implementierten Spracherweiterungen „C for Cuda" eine Softwareimplementierung von CUDA-Programmelementen ermöglicht. Die CUDA RUNTIME API realisiert ihre Ausführung über die CUDA DRIVER API. Die Verwendung der „C for Cuda" Spracherweiterung beinhaltet den Einsatz der CUDA RUNTIME API. Aus diesem Grund wird der Begriff „C for Cuda" in dieser Arbeit gleichbedeutend mit der RUNTIME API eingesetzt.

Diese APIs sind für den Einsatz in der Hostcode Softwareimplementierung vorgesehen um folgende Aufgaben zu übernehmen:[9]

- ▷ Device Management
- ▷ Memory Management
- ▷ Stream Management
- ▷ Event Management
- ▷ Execution Control
- ▷ Texture Reference Management
- ▷ OpenGL/Direct3D Interoperability

In der Auswahl der geeigneten API für die zu entwickelnde Softwareumsetzung ist zu berücksichtigen, dass die beiden C APIs nicht miteinander kombiniert werden können. Aus diesem Grund ist die Auswahl der Programmierschnittstellen über die Betrachtung der beiden Bibliotheken und ihrer Charakteristiken zu treffen. Bei der Entscheidung ist zu berücksichtigen, dass die Wahl der API nur Einfluss auf die Realisierung des Hostcodes nimmt. Die Implementierung der Kernel bleibt in jedem Fall bestehen.

Die „C for Cuda" Bibliothek (Runtime API) eignet sich für eine strukturierte und überschaubare Softwareimplementierung in Form eines vereinfachten Programmcodes, wobei jedoch weniger Einfluss auf den Programmablauf besteht. Des Weiteren besitzt diese API die Unterstützung der

[9] Quelle: [NVI09b] S.6

Device-Emulation (siehe Abschnitt 3.3.1). Dabei hat die CUDA Runtime API einen erheblichen Vorteil im Debugging-Vorgang.

Die Entwicklung des Programmcodes unter der Driver API fällt im Verhältnis zur „C for Cuda" Bibliothek wesentlich anspruchsvoller aus. Dieses drückt sich durch den wesentlich höheren Aufwand in der Programmcodeentwicklung aus. Die Vorteile liegen in einer besseren Kontrollmöglichkeit über das Device und die Eigenschaft einer Just-In-Time Kompilierung des Devicecodes. Jedoch muss diese Bibliothek auf eine Device-Emulation verzichten. Die Kombination aus komplexerem Programmcode und dem Verzicht auf eine Device-Emulation erschwert den Debugging-Vorgang in der Softwareentwicklung.

Prinzipiell stellen beide APIs die gleichen Funktionalitäten bereit. Die High-Level Charakteristik der „C for Cuda" Bibliothek ermöglicht eine Softwareentwicklung unter einem geringeren Aufwand gegenüber der Driver API. Daher ist unter dem Aspekt der bestmöglichen Implementierungsform die „C for Cuda"-API der Driver API vorzuziehen, wenn die Charakteristiken dieser API nicht benötigt werden.

Deshalb ist für die Erstellung dieser Arbeit die Wahl auf die „C for Cuda" Bibliothek gefallen.

3.2.2 Allgemeine Spracherweiterung in C

Für die Programmcodeentwicklung in CUDA ist die Einführung einer allgemeinen Spracherweiterung von essenzieller Bedeutung. Die folgenden Erläuterungen gelten für beide zuvor genannten C Bibliotheken. Diese Erweiterung beinhaltet Elemente für die Implementierung des Devicecodes sowie für die Interaktion der Host- und Device-Softwareelemente.

Die Spracherweiterung ist als eine Dialektform der Programmiersprache C zu betrachten. In diesem Dialekt sind Erweiterungen für die Deklaration von Funktionen und Variablen vorgesehen, um eine Unterscheidung zwischen Funktionstypen und Speichertypen vorzunehmen. Außerdem stehen in der Entwicklung neue Vektordatentypen (implementiert als Strukturen in C) zur Verfügung, die zum einen für die Threadorganisation in den Kerneln und zum anderen für eine verbesserte Speicherorganisation dienen. Die Thematik der erweiterten Speicherorganisation ist in Abschnitt 3.4.3 dokumentiert.

Funktionsdeklarationen

Die Spracherweiterung der Funktionsdeklaration findet über die Einführung der Schlüsselwörter ___global___ , ___device___ und ___host___ statt. Die Bedeutung dieser Schlüsselwörter sind aus der Tabelle 3.2 zu entnehmen.

Funktionstyp	Aufruf	Ausführung
___host___	Host	Host
___device___	Device	Device
___global___	Host	Device

Tabelle 3.2: Funktionstypen

Die exemplarische Darstellung der möglichen Funktionsdeklarationen ist dargestellt im Beispielcode 3.4.

Beispielcode 3.4 Beispiele für die Funktionsdeklaration

```
// Deklaration eines Kernels -- Aufruf durch Host -- Ausgeführt durch Device
__global__ void kernel(float value);

// Deklaration einer Devicefunktion -- Aufruf in Kernel (Device) --
    Ausgeführt in Kernel (Device)
__device__ float funktion(float value);

// Hostfunktion -- Entspricht normaler Funktion für den Host (__host__
    könnte entfallen)
__host__ float funktion(float value);

// Deklaration einer Funktion -- Aufruf durch Host und Device möglich
__host__ __device__ float funktion(float value);
```

Die entwickelten Kernel sind als Funktionen mit dem Schlüsselwort __global__ gekennzeichnet. Bei diesen Funktionen ist darauf zu achten, dass sie keinen Rückgabewerte (void) besitzen, da diese in n-facher Anzahl (n = Anzahl der Threads) ausgeführt werden und keinen bestimmten Wert zurückgeben können. Das Schlüsselwort __device__ signalisiert für den Kernel vorgesehene Funktionen, die im Übrigen, der C üblichen Syntax entsprechen. Ausschließlich Funktionen mit dieser Kennzeichnung können in einem Kernel ausgeführt werden. Die Deklaration __host__ ist dafür zuständig Hostcode zu kennzeichnen, und damit gleichbedeutend mit der üblichen Funktionsdeklaration. Das augenscheinlich redundante Element der Spracherweiterung ist als Möglichkeit gedacht, Funktionen für den Einsatz in Host- und Devicecode zu deklarieren.

Variablendeklaration

Die Spracherweiterung der Variablendeklaration ist durch die Einführung der Schlüsselwörter __device__, __shared__ und __constant__ definiert. Die Bedeutung dieser Schlüsselwörter sind aus der Tabelle 3.3 zu entnehmen.

Variablentyp	Speicherbereich	Deklarationspunkt
–	Register/Local	Deklariert in Devicecode
__shared__	Shared Memory	Deklariert in Devicecode
__device__	Global Memory	Globale Deklaration
__constant__	Constant Memory	Globale Deklaration

Tabelle 3.3: Variablentypen

Die exemplarische Darstellung der möglichen Funktionsdeklarationen ist in Beispielcode 3.5 veranschaulicht.

Beispielcode 3.5 Beispiele für die Variablendeklaration

```
// Globale Deklarationen
__device__ float value; // Variable in Global Memory
__constant__ float value; // Variable in Constant Memory

__device__ float funktion(void)
{
   // Variablen befinden sich in den Registern oder im Local Memory(DRAM)
   float i;          // Durch geringe Datenmenge voraussichtlich in
       Registern
   float values[512];   // Durch hohe Datenmenge voraussichtlich im "Local
       Memory"

   // Statischer Shared Memory
   __shared__ float values[512];
   // Dynamischer Shared Memory (Größendefinition über Kernelaufruf)
   __shared__ extern float values[];
}
```

Die aufgeführten Deklarationsformen gelten ausschließlich für den Devicecode und sind somit auf den Speicherzugriff im Devicecode beschränkt. Allerdings gelten für global deklarierte Variablentypen bestimmte Ausnahmen. In der Deklaration und Verwendung der einzelnen Speicherformen ist stets auf die jeweilige Einsatzform und die Geschwindigkeitsunterschiede der Speicherbereiche zu achten. Die Differenzierung der einzelnen Speicherbereiche ist im vorherigen Abschnitt 3.1.3 dokumentiert.

Die Deklaration der lokalen Variablen erfolgt ohne bestimmte Kennzeichnung. Dabei ist jedoch zu beachten, dass nicht direkt beeinflusst werden kann, ob die Wahl des eingesetzten Speicherbereichs auf die Register oder den lokalen Speicherbereich im DRAM fällt. Hierbei werden die Register in der Deklaration präferiert verwendet. Erst im Fall ungenügenden Speichers in den Registern erfolgt eine Zuteilung von Speicher im lokalen Speicherbereich im DRAM. Aus diesem Grund sollte besser eine genaue Abwägung über den tatsächlichen Speicherbedarf pro Thread durchgeführt werden. Ein minimaler Einsatz von lokalen Variablen bedeutet, dass mit größerer Wahrscheinlichkeit ausschließlich Register bei der lokalen Deklarierung verwendet werden.

Bei der globalen Deklaration einer Variablen mit __device__ nutzt die Variable den globalen Speicherbereich im DRAM. Dieser Speicher ist in allen durch das Device ausgeführten Funktionen adressierbar. Die als __constant__ deklarierten Variablen beinhalten die gleichen Eigenschaften in der Deklaration. Der Unterschied dieser Speicherform drückt sich durch eine begrenzte Größe (siehe Tabelle 3.1) und einen gecacheten Zugriff auf den „Constant Memory”-Speicherbereich aus. Diese Eigenschaften sind in Abschnitt 3.4.3 genauer aufgeführt. Wie bereits erwähnt, ist der Zugriff auf __device__ oder __constant__ global deklarierter Variablen nur im Devicecode erlaubt. Allerdings besteht die Möglichkeit, dass ein Zugriff über bestimmte, in der API integrierte Funktionen im Hostcode, erfolgt.

Die Deklaration von Variablen im Shared Memory findet durch die Kennzeichnung mit dem Schlüsselwort __shared__ statt. In dieser Deklaration ist es möglich, den Shared Memory in statischer und dynamischer Form festzulegen. Die statische Form erfolgt über eine in C übliche Definitionsform, um Variable bzw. Arrays mit statischer Größe festzulegen. Die Definition des dynamischen Shared Memory erfolgt über das zusätzliche Schlüsselwort extern und einer unbestimmten Größe

der Variablen. Die Menge des verwendeten Shared Memory wird über den Kernelaufruf (in „C for CUDA") festgelegt.

Vektordatentypen

Die Spracherweiterung mit zusätzlichen Datentypen für die Einbindung im Threadmanagement und der Datenhaltung.

Beispielcode 3.6 Kernelaufruf im Detail

```
// Vektordatentyp mit 4 Elementen Initialisieren
uint4 var = make_uint4(1,2,3,4);
var.x = var.y = var.z = var.w = 5; // Alle Strukturelemente == 5

dim3 grid(5); // grid.x == 5 , y und z == 1
dim3 block(N,N); // grid.x == grid.y == N, z == 1
```

Die Einführung der bereits erwähnten Vektordatentypen in Form einer Struktur in der Programmiersprache C beinhaltet die Definition von Strukturen aller bekannten Datentypen in Signed- und Unsigned-Form mit 1 bis 4 Elementen (Elementenbezeichnung von 1 bis 4 ist x, y, z und w). Die prinzipielle Struktur der Datentypbezeichnungen lautet `[u]DATENTYP[1..4]`. Dabei ist [u] Zusatz für Unsigned-Datentypen und [1..4] ist die Vektorgröße (1 bis 4). Für das bessere Verständnis sind hier als Beispiel einige Datentypen aufgeführt: **float3**, **uint1** und **char4**. Die Initialisierung von Variablen mit diesen Datentypen erfolgt über die Funktion `make_DATENTYP(argument1,...)`. Die vorgegebenen Datentypen ermöglichen über ihre Umsetzungsform einen einheitlichen Aufbau der CUDA-Softwareelemente. Über die in Abschnitt 3.4.5 dargestellte Implementierungsform ist eine Signalisierung der Datenzusammengehörigkeit für einige dieser Datentypen realisiert, um einen Geschwindigkeitsvorteil in der Datenübertragung zu erreichen.

Die eingeführten Datentypen **dim3** und **uint3** stellen den Grundsatz für die im Devicecode eingebundene Threadorganisation dar. Hierbei ist anzumerken, dass der Datentyp **dim3** dem Datentyp **uint3** entspricht. Die Initialisierung erfolgt über die Methode `dim3 VAR_NAME(argument1,...)`. Die in diesem Typ implementiere Vorinitialisierung der Strukturelemente mit dem Wert 1 lässt eine verkürzte Argumentendefinition zu. Über die integrierten Variablen **threadIdx** und **blockIdx** (uint3) sowie **blockDim** und **gridDim** (dim3) ist eine genaue Identifizierung der einzelnen Threads im kompletten Grid realisierbar. Diese Eigenschaft dient als Basis für die effiziente und überschaubare Threadorganisation im Programmiermodell. Diese Variablen sind im Devicecode direkt integriert und der Inhalt ist über die Parametrisierung des Kernelaufrufs definiert. Der Zusammenhang und die Verwendung dieser Variablen ist in Abschnitt 3.1.2 festgehalten.

Die Spracherweiterung für den Kernelaufruf und dessen Konfigurationsweise unterscheidet sich zwischen der „C for CUDA" und Driver API, weshalb die Beschreibung des Kernelaufrufs mit dessen Eigenschafften in Abschnitt 3.2.3 für die „C for CUDA" Bibliothek im Detail dokumentiert ist.

Die Ergänzung der Laufzeitumgebung um mathematische Funktionen[10] für den Einsatz im Host-

[10] [NVI09a] Appendix C - S.119

und Devicecode ermöglichen einen vereinfachten Entwicklungsprozess von mathematischen Zusammenhängen im Devicecode.

In der Implementierung von Devicecode ist zu beachten, dass die von NVIDIA veröffentlichte Spracherweiterung bestimmten Limitierungen und Vorgaben unterliegt. In der Erstellung von Programmcode für das Device ist darauf zu achten, keine rekursive Form der Programmierung einzusetzen, sowie die Verwendung von statischen Variablen und Funktionspointern auf Devicefunktionen (Deklaration mit ___device___) zu unterlassen.

Abgesehen von diesen eben genannten Bedingungen sind alle sonstigen Sprachkonstrukte der Programmiersprache C erlaubt. Die Spracherweiterungen ermöglichen auch den zusätzlichen Einsatz von Sprachkonstrukten der Programmiersprache C++. Hierbei sind die Überladung von Funktionen und Operatoren ein interessantes Element. Außerdem ist der Einsatz von Namensräumen, generischer Programmierung (Templates) und die Verwendung von Referenzen in der Spracherweiterung implementiert.

3.2.3 C for CUDA

Die C-Bibliothek[11] „C for Cuda" (Runtime API) bildet den Grundsatz für die in dieser Arbeit umgesetzten Softwareelemente in CUDA. In der Entwicklung des Programmcodes stehen in der „C for Cuda" Bibliothek vereinfachte Entwicklungsmethoden für die Umsetzung der Initialiserungs-, Kommunikations- und Ausführungsprozesse auf der Hostseite in der Programmiersprache C zur Verfügung. Der überschaubare Initialisierungsprozess in der Auswahl des Devices für die Ausführung des Kernels ermöglicht, die Konzentration auf die Kommunikations (Datentransfer)- und Ausführungsaufgaben.

Um den Umfang dieser Arbeit überschaubar zu halten, findet in diesem Abschnitt eine verkürzte Darstellung der „C for Cuda" Bibliothek statt. Die einzelnen Problematiken für die effiziente Programmcodeentwicklung sind mittels Beispiele anhand von „C for Cuda" in Abschnitt 3.4 festgehalten. Eine intensive Betrachtung der „C for Cuda" Bibliothek ist in dem NVIDIA Programming Guide [NVI09a] nachzulesen.

Die Ausführung der Laufzeitumgebung erfolgt aus der **„cudart dynamic library"**, die Funktionen sind mit dem Präfix **„cuda"** gekennzeichnet.[12]. Die Funktionen mit den zugehörigen Übergabeparameter und Rückgabewerte sind im Cuda Referenz Manual [NVI09d] dokumentiert. Aus diesem Grund ist es zu empfehlen, für die Implementierung von Softwareelementen in CUDA auf den Programming Guide [NVI09a] und das Referenz Manual [NVI09d] zurück zugreifen, um detaillierte Informationen für die Programmcodeentwicklung zu erhalten. Diese Dokumente sind im CUDA Toolkit enthalten.

Device-Initialisierung

Auswahl des Devices für die Ausführung des Kernels

Aufruf Device-Auswahl : `cudaSetDevice();`

Die Funktion `cudaGetDeviceCount()` ermöglicht eine Anfrage, über die tatsächliche Anzahl der im System installierten CUDA-fähigen Devices. Um über diese nähere Informationen zu erhalten,

[11] Quelle der aufgeführten Funktionen in diesem Abschnitt: [NVI09d]
[12] [NVI09a] S.17

ist die Funktion `cudaGetDeviceProperties()` zu verwenden. Die endgültige Auswahl des Devices erfolgt über die Funktion `cudaSetDevice()`.

Error-Handling

Die Verarbeitung der auftretenden Fehler im Einsatz von „C for Cuda"

Fehler Abfrage: `cudaGetLastError()`

Die in der „C for Cuda" Bibliothek implementierten Funktionen besitzen den generellen Rückgabewert `cudaError_t` (Enumeration: `typedef enum cudaError`). Dieser Rückgabewert enthält den durch die Funktion zurückgegeben Fehlercode. Ist dieser Fehlercode verschieden von dem Wert `cudaSuccess`, ist bei der Ausführung ein Fehler aufgetreten. Die Umwandlung des Fehlercodes in eine entsprechende Fehlermeldung ist über die Funktion `cudaGetErrorString()` zu realisieren. Die Funktion `cudaGetLastError()` liefert den zuletzt aufgetretenen Fehler(letzter Errorcode). Die veranschaulichten Methoden ermöglichen eine unkomplizierte Fehlerbehandlung für eine rasche Lokalisierung von Fehlerquellen und Problemstellen im Programmcode. Die Liste der möglichen Fehlercodes der bekannten Fehlerquellen ist dem Cuda Referenz Manual ([NVI09d]) zu entnehmen.

Speicherallokation/Datentransfer

Umsetzung der Kommunikations- und Datentransferprozesse im Hostcode

Die Allokation des benötigten Speichers in den verschiedenen Speicherbereichen ist über die in der Bibliothek implementierten Funktionen umsetzbar. Für die Reservierung von Speicher im Host-Speicherbereich stehen, neben den C-Standardfunktionen, in der CUDA-API Methoden für die Speicherallokation zur Verfügung. Auf diesem Weg ist eine verbesserte Übertragungsmöglichkeit aus dem Host-Speicherbereich in den Device-Speicherbereich gegeben. Die so zu erreichende Optimierung ist im Detail in Abschnitt 3.4.3 dokumentiert.

Der Einsatz der Device-Speicherbereiche ist unter der Verwendung eines linearen Speichers mit den Funktionen `cudaMalloc()` , `cudaMemcpy()` und `cudaFree()` zu erzielen. Im Vorgang des Speichermanagements auf dem Device existieren Funktionen, um eine Speicherorganisation in verschiedenen Dimensionen zu realisieren. Die Funktion `cudaMemcpy()` ermöglicht eine „Host zu Device"-(Funktionsargument: `cudaMemcpyHostToDevice`), „Device zu Host"- (`cudaMemcpyDeviceToHost`), „Device zu Device" (`cudaMemcpyDeviceToDevice`) und „Host zu Host" (`cudaMemcpyHostToHost`) -Transaktion.

Die Nutzung des Constant- und Texture-Speicherbereiches ist über spezielle Funktionen umgesetzt. Die Methoden `cudaBindTexture()`, `cudaBindTexture2D()` und `cudaBindTextureToArray()` befähigen den allokierten Device-Speicher zu einer Zuordnung auf den Texture-Speicherbereich. Der Datentransfer in eine mit ___constant___ deklarierte Variable (Constant-Speicherbereich) ist über die Funktion `cudaMemcpyToSymbol()` umzusetzten. Die optimale Handhabung des READ-Only Speicherbereichs im Device ist in Abschnitt 3.4.3 dokumentiert.

Kernelaufruf

Die durch ___global___ gekennzeichneten Devicefunktionen werden in n-facher Anzahl für jeden vorhandenen Thread ausgeführt. Die Syntaxform <<< ... >> kennzeichnet den Kernelaufruf in der „C for Cuda" API.

Exemplarischer Aufruf: `kernel<<< Kernelargumente >>>(Funktionsargumente);`

Beispielcode 3.7 Kernelaufruf im Detail

```
// Defintion der Blöcke und des Grids
dim3 dimBlock(16, 16);  dim3 dimGrid( N, N);
// Kernelaufruf mit möglichen Konfigurationsargumenten
kernel<<< dimGrid , dimBlock, sharedMemSize, streamId >>>(data);

// Üblicher Kernelaufruf (sharedMemSize = 0,  streamId = 0)
kernel<<< dimGrid , dimBlock >>>(data);

// Kernelaufruf mit dynamischem Shared Memory (streamId = 0)
kernel<<< dimGrid , dimBlock,sharedMemSize >>>(data);

// Kernelaufruf mit dynamischem Shared Memory und Stream
kernel<<< dimGrid , dimBlock,sharedMemSize,streamId >>>(data);
```

Die in Abschnitt 3.1.1 vorgestellte Methode der Kernelausführung veranschaulicht die meist etablierte Umsetzungsform. Allerdings verfügt die Kernelausführung über eine erweiterte Argumentenliste. Die gesamte Bandbreite der Konfigurationsargumente lässt eine Definition des Grids, des Threadblocks, der benötigten Speichergröße im Shared Memory genauso wie eine Zuweisung des Verarbeitungsstreams zu. Die ersten beiden Argumente stehen, wie bereits bekannt, für die Dimensionierung des Grids und den darin enthaltenen Threadblöcke. Ein weiteres Argument definiert die Speichergröße des dynamischen Shared Memory in Bytes (Deklarationsform im Kernel: siehe Abschnitt 3.2.2). Schließlich ist es möglich, dem Kernel eine StreamId zu übergeben, die für das in Cuda implementierte Streamverarbeitungskonzept benötigt wird. Das Umsetzungskonzept in „C for Cuda" für den Programmablauf in Streams ist in der folgenden Beschreibung „Verarbeitungsströme" veranschaulicht.

Die Kernelausführung erfolgt asynchron (Standard: StreamId = 0). Der Host führt lediglich die Initialisierung und den Start des Kernels durch. Daraufhin wird die Ausführung des Hostcodes fortgesetzt. Diese Eigenschaft ist in der Programmierung des Hostcodes zu berücksichtigen.

Um im Kernel einen schnellen Zugriff auf die Konfigurations- und Funktionsargumente zu gewährleisten, werden diese automatisch im Shared Memory abgelegt. Die in den Funktionsargumenten enthaltenden Zeiger müssen auf Elemente im globalen Speicher des Devices verweisen. Die Speichermenge der Funktionsargumente ist auf 256 Bytes beschränkt.

Verarbeitungsströme

Die Organisation des Programmablaufs in Verarbeitungsströmen auf dem Device.

Stream erzeugen: `cudaStreamCreate()`

Die Steuerung der Abfolge von auszuführenden Elementen auf dem Device mit asynchroner Ablaufstruktur zum Host ist über Verarbeitungsströme realisiert. Die Ausführung von Programmelementen auf dem Device ist standardmäßig über einen Stream (StreamID=0) organisiert. Die Ausführung auf dem Standardstream ist jedoch vom Anwender fern gehalten. Erst mit dem Einsatz von asynchronen Datentransferfunktionen kommt der Anwender mit der Streamorganisation in Kontakt.

Die Ablaufstruktur der Kernel lässt, wie in Abschnitt 3.1.4 erläutert, nach den bisherigen Standards keine parallele Ausführung der Kernel zu. Die Möglichkeit der Überschneidung von Daten-

transfervorgängen und Kernelausführungen über die asynchrone Ausführung von Kopiervorgängen (`cudaMemcpyAsync()`) setzt eine Ablauforganisation in Streams voraus, um Speicherzugriffe des Kernels auf noch zu übertragende Elemente zu vermeiden. Darüber hinaus symbolisiert der Stream einen Ablaufplan der auszuführenden Programmelemente auf diesem Stream. Der Einsatz von mehreren Streams ist ohne weiteres möglich.

Die Erstellung des Ausführungsstroms ist über die Funktion `cudaStreamCreate()` realisiert. Nach Gebrauchsbeendigung dieses Streams kann dieser über die Methode `cudaStreamDestroy()` entfernt werden. Der Aufruf von asynchronen Datentransfermethoden erfolgt unter der Angabe der StreamId. Die detaillierte Implementierungsform mit den Vorteilen der Streamverarbeitung ist in Abschnitt 3.4.5 dargestellt.

3.3 Entwicklungswerkzeuge

In der Softwareentwicklung mit Compute Unified Device Architecture sind zusätzliche Hilfsmittel für den Debugging- und Optimierungsprozess von entscheidender Bedeutung. Der Hersteller dieser Architektur stellt als Analyse- und Entwicklungswerkzeug die Einsatzmöglichkeit einer Device-Emulation und einen visuellen Profiler für den Devicecode zur Verfügung.

3.3.1 Device-Emulation

Die Programmcodeentwicklung unter dem Einsatz von „C for Cuda" ermöglicht die Verwendung eines Device-Emulationsmodus. Die Aktivierung des Emulationsmodus erfolgt über den `„-deviceemu"` Parameter des NVCC Kompilierer-Treiber, dessen Aktivität wird durch das Präprozessormacro `__DEVICE_EMULATION__` signalisiert.

Die Device-Emulation ist eine überschaubare Methode für den Debugging Prozess von Devicecode. Die Emulation führt eine Umsetzung des parallel organsierten Programmcodes auf der CPU durch. In dieser Umsetzung erfolgt eine Erzeugung eines Threads auf der CPU für jeden definierten CUDA-Thread in einem Threadblock. Auf diesem Weg ist es möglich, das in der verwendeten Entwicklungsumgebung eingebundene Debugging-Werkzeug einzusetzen. Über diese Umwandlungsform ist ein Überprüfungsmodus der implementierten Algorithmen im Devicecode geschaffen, um die Korrektheit des Ablaufs und der Ergebnisse festzustellen. Zusätzlich bietet die Emulation eine Ausführungsvariante des Programmcodes ohne die Notwendigkeit einer vorhandenen CUDA-fähigen Hardware. Auf diesem Weg können Applikationen in CUDA entwickelt und debugged werden, ohne für diese Prozesse eine CUDA-fähige Hardware zu benötigen. Wichtig: Der Optimierungsprozess des Devicecodes kann nicht ohne die passende Hardware stattfinden.

Der Device-Emulationsmodus beinhaltet zusätzliche Methoden für den Einsatz im Debugging-Prozess. Die Einbindung von „`printf()`"-Funktionen ist für Debug-Zwecke im Emulationsmodus zugelassen. Dabei handelt es sich um eine Hostfunktion, die normalerweise im Devicecode nicht eingesetzt werden kann. Die zusätzliche Einbindung von Synchronisationsbarrieren mit der Funktion `__syncthreads()` ermöglicht im Debugging-Vorgang eine Analyse der einzelnen Threads über die augenscheinliche Abarbeitungsform in for-Schleifen. Auf diesem Weg ist eine schrittweise Analyse jedes Verarbeitungsschritts über die Abpassung der einzelnen Threads zu erzielen. Eine Überschrei-

tung der Synchronisationsbarrieren findet erst ab dem Zeitpunkt der kompletten Abarbeitung aller Ausführungsstränge statt.

Wichtig: Die Ergebnisse in den Berechnungen im Emulationsmodus und dem direkten Ablauf auf der CUDA-fähigen Hardware unterliegen einem Genauigkeitsunterschied. Diese Eigenschaft ist bedingt durch das unterschiedliche Verarbeitungsverhalten in der Fließkommaarithmetik durch den Host und das Device, sowie durch die Ausführung des Devicecodes durch die CPU im Emulationsmodus. Die CUDA-Hardware folgt zum großen Teil dem IEEE 754 Floting-Point[13] Standard, der sich allerdings im Detail von dem im Host eingesetzten IEEE 754 Standard unterscheiden kann. Dabei sind mögliche Einflussgrößen: Der Kompilierer mit den dazugehörigen Einstellungen, die unterschiedlichen Instruktionssätze und vor allen Dingen die andersartigen Architekturen[14]. Diese Eigenschaften sind im Einsatz des Emulationsmodus stets zu beachten.

3.3.2 Visual Profiler

Das CUDA Toolkit beinhaltet den CUDA Visual Profiler. Dieses Entwicklungswerkzeug ermöglicht eine visualisierte Veranschaulichung der Devicecode Charakteristiken. Hierbei besteht eine Analysemöglichkeit des strukturellen Aufbau des Grids mit Struktur und Anzahl der Threadblöcke, die eingesetzten Speichermengen der ON-Chip-Speicherbereiche, so wie das Datenübertragungsverhalten mit Mengenangaben und Transferraten. Die Analyse der divergenten Ausführungsstränge und serialisierte „Shared Memory“-Zugriffe stehen gleichermaßen zur Verfügung.

Der Visual Profiler bewerkstelligt eine detaillierte Analyse des gesamten Programmablaufs und dessen Charakteristiken im Daten- und Instruktionsdurchsatz, sowie der exakten Ausführungszeiten der Devicecode-Elemente. Dabei erfolgt eine tabellarische Darstellung der Messungen und Eckdaten der ausgeführten Kernel- und Datenübertragungsfunktionen. Der Profiler veranschaulicht zusätzlich in grafischer Form die zeitliche Abfolge der Devicecode-Elemente. Eine detaillierte Liste der veranschaulichten Daten ist in Anhang B aufgeführt.

Die Aufnahme dieser Statistiken wird anhand eines einzigen Multiprozessors durchgeführt und steht als Repräsentant für die gesamte GPU. Aus diesem Grund sind die Ergebnisse jedoch auch nur als Richtwert zu behandeln, um eine bessere Bewertungsmöglichkeit im Entwicklungsschritt der Optimierung zu besitzen. Die Ermittlung eines repräsentativen Ergebnisses setzt eine starke Auslastung der GPU vorraus, die über in eine hohe Anzahl von auszuführenden Threadblöcken zu erreichen ist. Diese Struktur ist die Voraussetzung, um über die eingesetzte Datenermittlungsform in einem einzigen Multiprozessor auf den vollen Umfang der Auswirkungen in der GPU schließen zu können.

Dieses Entwicklungswerkzeug ermöglicht eine detaillierte Analyse aller Programmelemente, um den Optimierungsvorgang dieser Softwareteiler effizienter durchführen zu können. Hinweise auf die aufgeführten Datenbezeichnungen und ihrer Bedeutung erfolgt in der Darstellung der einzelnen Optimierungsschritte, bei denen diese Daten für die Analyse von essenzieller Bedeutung sind (siehe Abschnitt 3.4).

[13] Detailinformationen: [NVI09a] Appendix A S. 103, 104
[14] Quelle: [NVI09a] S.45

3.4 Elemente der effizienten Implementierung

In der Implementierung von Softwareelementen ist die Orientierung anhand eines Entwicklungsleitfadens von entscheidender Bedeutung. Dieser Abschnitt bescheibt die allgemeinen Empfehlungen.

Die in diesem Abschnitt präsentierten Optimierungsschritte sind in folgender Abfolge in übersichtlich festgehalten:

Genereller Grundsatz:

Höchstmögliche Hardwareausnutzung mit maximaler Parallelisierung

1. **Implementierung des Algorithmus**

   Der grundsätzlich erste Entwicklungsschritt enthält die allgemeine Implementierung des Algorithmus in der parallelen Struktur des vorgestellten Programmiermodells.

2. **Optimierung auf Speicheroperationen**

   In diesem Schritt findet eine Analyse der geeigneten Speicherorganisation statt. Die Analyse beinhaltet die Wahl des eingesetzten Speicherbereichs mit der dazugehörigen Implementierungsform und eine Minimierung von Speicherzugriffen. In der effizienten Implementierung ist der Speicherorganisation in der CUDA Architektur aus zeitkritischen Aspekten besondere Aufmerksamkeit zu schenken.

3. **Arithmetische Optimierung**

   Die Zielsetzung der Speicherzugriffsoptimierung beinhaltet zusätzlich auch eine Erhöhung der arithmetischen Dichte (arithmetische Operationen als effektivere Alternative zu Speicherinstruktionen). In diesem Implementierungsschritt besteht die Absicht, über eine Wahl der effizientesten Operationen, die in diesem Algorithmus möglich sind, um einen höheren Instruktionsdurchsatz zu erzeugen.

4. **Vermeidung von divergenten Ausführungssträngen**

   In den aufgeführten Optimierungsschritten ist stets das SIMT-Konzept zu beachten. Die Vermeidung von divergenten Ausführungssträngen führt direkt zu einer Verminderung der Ausführungszeit.

Diese dargestellten Punkte dienen als grundlegender Leitfaden für die effiziente Implementierung von Softwareelement in CUDA.

3.4.1 Maximale Auslastung - Maximale Parallelisierung

Die Aufmerksamkeit in diesem Optimierungsschritt dient der Veranschaulichung von Gründen und Lösungen für eine größtmögliche Auslastung über maximale Parallelisierung der zur Verfügung stehenden Hardware. In der Theorie ist ein CUDA-Multiprozessor limitiert auf die gleichzeitige Aktivität von 8 Threadblöcken auf einem Multiprozessor. Die tatsächliche Anzahl von Threadblöcken ist jedoch limitiert durch die maximale Anzahl an aktiven Warps in einem Multiprozessor,

wie auch die Restriktionen im Speicherbedarf der einzelnen Threads (Register) und Threadblöcke (Shared Memory). Die Hardware bedingten Begrenzungen sind in den Compute Capability Specifications festgehalten. Diese Eigenschaften sind in der Dimensionierung des Grids, der Threadblöcke, der Zuweisung der Registeranzahl pro Thread und der Größe des eingesetzten Shared Memory zu berücksichtigen. Die CUDA SDK beinhaltet ein Excel-Sheet in dem eine genaue Bestimmung der Hardwareausnutzung vorgenommen werden kann. Das Dokument mit der Bezeichnung „CUDA GPU Occupancy Calculator"[15] bestimmt, unter der Angabe der benötigten Register, der Shared Memory Größe und der Anzahl der Threads in einem Block, die prozentuale Auslastung[16] der Multiprozessoren. In den Ergebnissen wird Aufgrund der unterschiedlichen Hardwarebegebenheiten unter den einzelnen Compute Capability Versionen unterschieden (siehe Anhang A). Der in Abschnitt 3.3.2 vorgestellte Visual Profiler ist in der Lage in der Analyse eines Softwareelements dieselben Ergebnisse zu liefern[17]. Anhand des Excel-Sheets ist jedoch eine Bewertung der Applikation im Bezug auf die möglichen Änderungen durchführbar, um eine Steigerung der Auslastung zu erreichen. Generell ist jedoch die Aussage zu treffen, dass eine Minimierung des Speicherbedarfs an Registern und Shared Memory direkt zu einer höheren Auslastung führt. Eine Wahl der Threadblockgröße von midestens 128 Threads[18] hat sich von diesem Aspekt her, als effektiv herausgestellt, wobei zu jedem Zeitpunkt darauf zu achten ist, die Threadanzahl als ein Vielfaches von 32 (SIMT-Prinzip) zu wählen.

Die maximale Auslastung hat nicht direkt eine Beschleunigung des Programmablaufs zur Folge, da der Verarbeitungsprozess prinzipiell auf der SIMT-Technologie (Instruktionsausführung in Warps) beruht. Jedoch ist dies die Möglichkeit, Latenzen von Speicherzugriffen zu verstecken. Die erhöhte Anzahl von gleichzeitig aktiven Warps im Multiprozessor ermöglicht die Überlagerung von Wartevorgängen im Speicherzugriff (Delays) mit arithmetischen Operationen. Dieses Verfahren bewirkt eine Sicherstellung des höchstmöglichen Instruktions- und Datendurchsatzes.

3.4.2 Threadstruktur

Um eine Optimierung in der Ausführungsgeschwindigkeit zu erreichen, ist die Instruktionsverarbeitung in Warps und die Auswirkung von divergenten Ausführungssträngen stets zu beachten (siehe Abschnitt 2.2.4). In diesem Abschnitt sind potenzielle Verursacher von divergenten Ausführungswegen[19] veranschaulicht.

In diesem Zusammenhang ist es zu empfehlen, als ersten Schritt in der Blockdimensionierung in jedem Fall ein Vielfaches von 32 als Threadanzahl zu wählen. Auf diesem Weg ist eine ganzzahlige Menge an Warps in jedem Block enthalten. An dieser Stelle erfolgt zusätzlich eine Veranschaulichung der Programmcodekonstrukte, die divergente Ausführungsstränge zur Folge haben können. Die Kontrollstrukturen `if, switch, do, for` und `while` stellen dabei einen potenziellen Ausgangspunkt von divergenten Ausführungswegen dar. Über diese Strukturen ist es möglich, Unterscheidungen im Ausführungsverhalten der einzelnen Threads zu bewirken. Dem Kompilierer obliegt die Möglichkeit Kontrollstrukturen im Optimierungsprozess zu entfernen und durch

[15] CUDA_Occupancy_calculator.xls im Verzeichnis: „tools" der SDK[NVI09f]
[16] Bezogen auf einen Multiprozessor: Verhältnis der aktiven Warps (Anzahl zu Maximaler Anzahl)
[17] Datenerläuerung Visual Profiler: siehe Anhang B- Occupancy, Register Count, Shared Memory Size
[18] Max. Aktive Blöcke pro Multiprozessor: $\frac{768}{128} = 6$, $\frac{768}{256} = 3$ bzw. $\frac{1024}{128} = 8$, $\frac{1024}{256} = 4$
[19] Im Profiler: divergent branch (siehe Anhang B)

„Pseudo"-Operationen ohne Auswirkungen auf die Datenelemente zu ersetzen (Branch Predication[20]), um divergente Ausführungsstränge in den Warps zu vermeiden.

Dieser Zusammenhang ist an einem kurzen Beispiel verdeutlicht.

Beispielcode 3.8 Divergente Ausführungsstränge

```
__global__ void Kernel(float* in1,float* in2,float* out)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   // Divergente Ausführungsstränge in den Warps
   if((x % WARPSIZE) == 0) // Wählt den ersten Thread jedes Warps aus
      out[x] = in1[x]*in2[x];
   else
      out[x] = in2[x] + 50;
}
```

Im Codebeispiel 3.8 erfolgt die Auswahl jedes ersten Threads eines Warps. In diesem Fall realisieren diese Threads eine Multiplikation und die verbleibenden Threads in den Warps einen Addition. Diese Verarbeitungsform hat mögliche divergente Threads in einem Warp zur Folge.

Beispielcode 3.9 Nicht divergente Ausführungsstränge

```
__global__ void Kernel(float* in1,float* in2,float* out)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   // Gemeinsame Ausführungswege in den Warps
   if((x / WARPSIZE) == 0) // Jeder Thread des ersten Warps
      out[x] = in1[x]*in2[x];
   else
      out[x] = in2[x] + 50;
}
```

Der im Codebeispiel 3.9 veranschaulichte Zusammenhang führt zu einer Auswahl jedes Threads im ersten Warp. Dieser Warp führt in jedem Ausführungsstrang eine Multiplikation durch. Eine Addition findet ihre Ausführung in den verbleibenden Warps statt. Diese Kontrollstrukturen können nicht zu divergenten Ausführungssträngen innerhalb eines Warps führen.

Die oben aufgeführten Beispiele veranschaulichen potenzielle Verursacher von divergenten Ausführungssträngen. In Folge dessen stellen divergente Ausführungsstränge nur eine Problematik innerhalb eines Warps dar. Die divergenten Ausführungswege von unterschiedlichen Warps (innerhalb des Warps gemeinsame Ausführungswege) stellen in diesem Architektur- und Programmiermodell keine problembehaftete Situation dar.

3.4.3 Speicherorganisation

Die beiden bereits vorgestellten Methoden sind bei der Optimierungsintensivierung von Devicecodeelementen weiterhin zu jeder Zeit zu berücksichtigen. Die positiven Auswirkungen einer starken Parallelisierung für die Verschleierung von Speicherzugriffsverzögerungen sind in der Speicherorganisation von relevanter Bedeutung. Die geschwindigkeits- und zugriffsbedingten Unterschiede der

[20] [NVI09b] S. 47, 48

Device-Speicherbereiche sind in Abschnitt 3.1.3 deutlich veranschaulicht. Die in diesem Abschnitt dokumentierten programmiertechnischen Organisationsstrukturen dienen der effizienten Ausnutzung der vorgestellten Speicherbereiche.

Die prinzipielle Entwicklungsstrategie im Speichermanagement stellt die Vermeidung von Speichertransfers und Speicherzugriffen dar. Die Ausführung von arithmetischen Operationen sind den Speicheroperationen stets vorzuziehen. Die Begründung liegt hierbei in der Charakteristik der GPU-Architektur[21].

Host-Memory

Die Kommunikationsvorgänge (Datentransfer) bilden allgemein eine zeitverzögernde Charakteristik im Gesamtablauf der Berechnungsvorgänge. Diese lassen sich jedoch nicht vermeiden. Den größten Flaschenhals in diesem System bildet hierbei die Kommunikation zwischen dem Host und Device, die in diesem Fall über eine PCI-Express-Schnittstelle realisiert ist.

Die interne Speicherbandbreite auf der Coprozessorkarte (Grafikkarte) steht in einem großen Verhältnis zu der Übertragungsgeschwindigkeit in die Host-Speicherbereiche. Der derzeitig[22] aktuellste Grafikprozessor GT200 der Firma NVIDA beinhaltet bis zu acht implementierte 64-Bit GDDR3-Speicherinferfaces (Zusammen 512 Bit). Bei einer effektiven Taktrate von 2000 MHz ist folglich eine theoretische Übertragungsrate von 128 GB/s zu erwarten. Der PCI-Expressbus x16 2.0, der für die GT200-Zusatzkarten eingesetzt wird, erlaubt eine theoretische maximale Übertragungsrate von 8 GB/s. In diesem Zusammenhang bilden sich die Problematiken der Host->Device bzw. Device->Host-Kommunikationen aus.

Auf Grund dieser Problematik sind in der C Bibliothek „C for Cuda" Methoden implementiert, um die Datentransaktionen unter effizienter Ausnutzung der Bandbreite zu realisieren. Die in den Standardbibliotheken eingesetzten Funktionen für die Allokations- und Kopiervorgänge beruhen auf den als „Pageable Host Memory"[23] bezeichneten Speicher. In dieser Form der Speicherorganisation ist es allerdings nicht möglich, das volle Potenzial des PCI-Expressbusses auszunutzen. Die effizientere Ausnutzung der Busbandbreite wird durch die CUDA-APIs unter dem Einsatz von „Page-Locked Host Memory" oder auch bekannt als „Pinned Memory" ermöglicht. Diese Speicherorganisationsform erzielt höhere Übertragungsraten im Datentransfer. Die in der „C for Cuda" Bibliothek implementierten Funktionen `cudaMallocHost()` und `cudaHostAlloc()`[24] ermöglichen eine Speicherallokation in dieser Form. Die Freigabe erfolgt über die Funktion `cudaFreeHost()`. Der Einsatz dieser Speicherorganisationsform ist die Voraussetzung für die Überlagerung von Kopiervorgängen und Kernelausführungen in einer Ausführungsstruktur von Verarbeitungsströmen (Streams). Unter Betrachtung dieses Hintergrundes, ist der Einsatz von „Page-Locked Memory" als eine besonders effizienzsteigernde Maßnahme zu betrachten.

Global Memory

Die prinzipiellen Eigenschaften der globalen Speicherbereiche sind in Abschnitt 3.1.3 dargestellt. Dieser Teil des Abschnitts dokumentiert die Zusammenhänge im Speicherzugriff auf den unge-

[21] Allgemeiner Aufbau GPU in Abschnitt 2.1.2
[22] Zum Erstellungszeitpunkt dieser Arbeit
[23] Hintergrundinformationen für diesen Abschnitt: CUDA Programming Guide[NVI09a] S.32
[24] Wie cudaMallocHost mit speziellen Funktionalitäten

cachten „Global Memory". Der Optimierungsprozess beinhaltet eine Minimierung der Latenzbehafteten Speicherzugriffe auf diesen Speicherbereich.

Alignment

Die Speicheroperationen sind prinzipiell für den Global und Shared Memory in halben Warps (16 Threads) organisiert. Dabei führt jede Thread des „Half Warps" eine 32-Bit, 64-Bit oder 128-Bit Speicherinstruktion aus. Die Wahl der Speicherinstruktion ist hierbei abhängig von der auszulesenden Datenmenge des Datentyps. Um eine Minimierung der Speicherinstruktionen zu erreichen, besteht die Aufgabe, die Datenanordnung und Kennzeichnung in der Form zu konstruieren, dass die größtmögliche Datenmenge mit einer einzigen Instruktion übertragen wird. Diese Problematik ist anhand von 32-Bit Integerdatentypen zu veranschaulichen. Diese Auslese- und Schreiboperationen dieser Datenelemente sind im Normalfall ohne weitere Kennzeichnung durch vier 32-Bit Speicherinstruktionen realisiert. Die Zusammenfassung dieser Daten, in eine einzige gemeinsame 128-Bit Speicherinstruktion, ist durch den Einsatz einer Struktur in C mit einer Kennzeichnung über das Schlüsselwort `__align__()` zu realisieren. Diese Kennzeichnung ist für die Speicherinstruktionen in den Größen von 4, 8 und 16 Bytes umsetzbar. Der „Alignment Specifier"[25] veranlasst den Kompilierer die Instruktionen in dieser Form durchzusetzen. Für diese Speicherorganisation ist es erforderlich, dass die Speicheradresse einem Vielfachen der Speichereinheit entspricht.

Beispielcode 3.10 Beispiel Alignment

```
// Alignment für einen 16 und 4 Byte Speicherzugriff
struct __align__(16) bsp1 { int a; int b; int c; int d;}; // Beispiel 1
struct __align__(4) bsp2 { BYTE a; BYTE b; BYTE c; BYTE d;}; // Beispiel 2

// Alignment für einen 16 Byte Speicherzugriff mit 12 Byte Daten
struct __align__(16) bsp3 { int a; int b; int c;}; // Beispiel 3
// Alignment für zwei 16 Byte Speicherzugriffe mit 20 Byte Daten
struct __align__(16) bsp4 { int a; int b; int c; int d;int d;}; // Beispiel
    4
```

Die im Beispielcode 3.10 aufgeführten Varianten sind auf Grund verschiedener Prozesse in der dargestellten Form umgesetzt. Das erste Beispiel stellt eine Definition für die bereits beschriebene Problematik dar.

Die kleinste Form des Speicherzugriffs erfolgt über eine 32-Bit Instruktion. Diese hat jedoch zur Folge, dass einzelne 8-Bit Werte mittels einer 32-Bit Instruktion ausgelesen, und in diesem Fall, für den Übertragungsvorgang von vier Byte in vier Instruktionen umgesetzt wird. Für die vier auszuführenden Instruktionen bei dem Datentyp BYTE (z.B. char) ist über ein Alignment eine Minimierung auf eine einzige Instruktion vornehmbar.

Die in der Programmierumgebung zur Verfügung gestellten Vektordatentypen sind an den soeben vorgestellten Prinzipien orientiert.

Die Variationen der Alignmentkonstrukte erlauben weiterhin die Transformation von mehreren Zugriffsformen (32-Bit oder 64-Bit) in zusammengefassten Instruktionen, auch wenn diese nicht exakt der genauen Größe entsprechen. Die beiden letzten aufgeführten Beispiele veranschaulichen die Umsetzung von drei 4 Byte Instruktionen in eine 16 Byte Instruktion (Beispiel 3) sowie fünf 4

[25] [NVI09a] S.81

Byte Instruktionen in zwei 16 Byte Instruktionen. Im Einsatz dieser Konstruktionen ist allerdings zu beachten, dass der Speicherbedarf dieser Daten anhand der Alignement-Größe definiert ist und nicht an den Größen der Strukturelemente. Dies drückt sich in Beispiel 3 in der Größe von 16 Byte und in Beispiel 4 in einer Speichermenge von 32 Bytes aus (siehe [NVI09a] S. 81).

Coalescing

Die Transaktionsorganisation in halben Warps ermöglicht obendrein die Reduzierung von Speicherzugriffen auf einer nächst höheren Ebene. Auf diesem Weg ist eine Zusammenfassung der Speicherzugriffe des „Half Warp" in einer einzigen Transaktion von 32, 64 oder 128 Bytes erreichbar. Diese Verarbeitungsweise ist als eine verschmolzene Speichertransaktion mit der Bezeichnung „Coalesced Memory Access"[26] zu betrachten und ist in Abbildung 3.5 und 3.6 in verschiedenen Szenarien dargestellt und folgend erläutert.

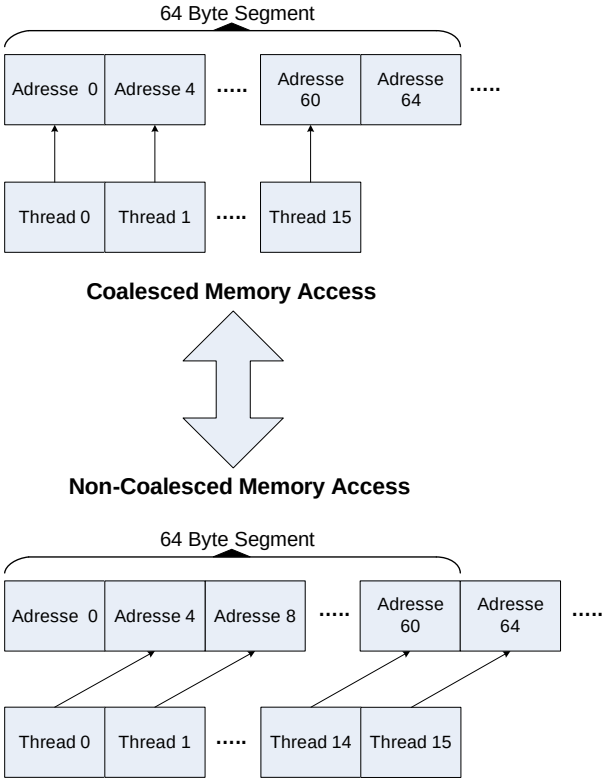


Abbildung 3.5: Coalesced / Non-Coalesced Memory Access - Compute Capability 1.0/1.1

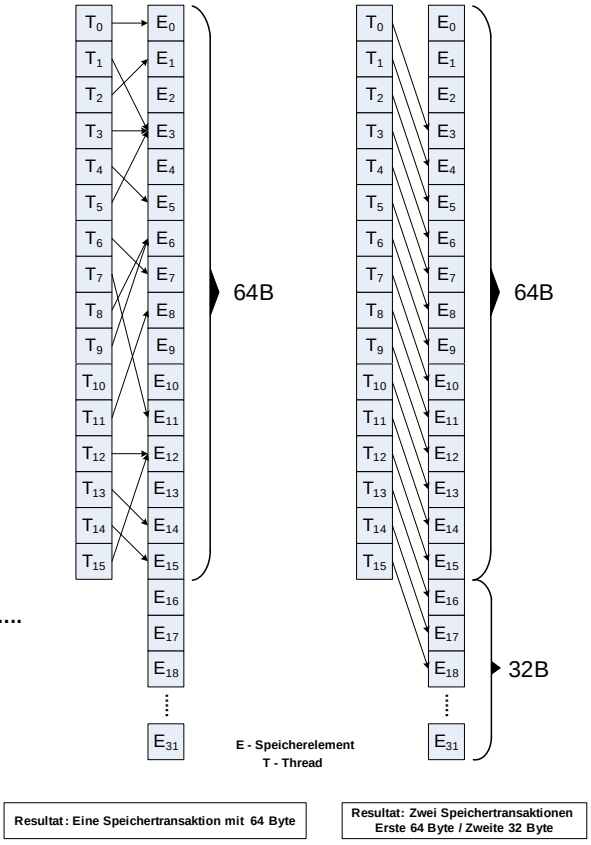


Abbildung 3.6: Coalesced Memory Access - Compute Capability 1.2

Das Coalescing von Speicherinstruktionen ist grundsätzlich über das Alignment in 4, 8 und 16 Byte Speichertransaktionen[27] und der exakten Adressierung in 32, 64 oder 128 Byte (von Alignment abhängig) großen Speichersegmenten im „Global Memory" zu erzielen. Diese Eigenschaft ist eine prinzipielle Voraussetzung für die nach den Compute Capability Specifications 1.0 und 1.1 standardisierte Hardware. Geschieht dieses jedoch nicht, findet kein verschmolzener Speicherzugriff in diesen Versionen statt und es werden 16 einzelne Speicherzugriffe durchgeführt. Nach der neueren

[26] Analyse im Visual Profiler: gld und gst (siehe Anhang B)
[27] Alignment von Strukturen in vorweg erläuteter Form

Standardisierung in den Versionen ab 1.2 ist eine willkürliche Adressierung gestattet. Die Speicherzugriffe eines halben Warps innerhalb eines bestimmten Segments werden in einer einzigen Speichertransaktion zusammengefasst. Demzufolge erfolgt in dieser Version eine Verschmelzung der Zugriffe für jedes adressierte Segment, wobei n Speichertransaktionen für n Segmente erfolgen. Die Wahl der Speichertransaktion erfolgt auf die jeweils kleinstmögliche Größe. Der detaillierte Zusammenhang dieser Vorgänge ist im CUDA Programming Guide [NVI09a] dokumentiert.

Shared Memory

Der Shared Memory dient als gemeinsam im Threadblock eingesetzter On-Chip-Memory, dargestellt in Abschnitt 3.1.3. Dieser Speicherbereich verfügt über die Geschwindigkeitsvorzüge seiner Implementierungsform, um einen zweitverzögerungsfreien Pufferspeicher für die Entwicklung von Algorithmen in der CUDA-Programmierumgebung zu Verfügung zu stellen.

Die generelle Struktur des Shared Memorys ist über 32-Bit große, hintereinander abfolgende Speicherbänke realisiert. Der Speicherzugriff eines Warps findet in einem Zeitraum von 2 Taktzyklen statt, bei dem dieser Warp in zwei separate Speicherzugriffe eines halben Warps aufgeteilt wird. Die Speicherzugriffe auf den Shared Memory werden in einem halben Warp umgesetzt. Die beiden Speicherzugriffe können durch die separate Behandlung (separate Zugriffe) nicht in Konflikt geraten. Das „Shared Memory"-Zugriffsverhalten beinhaltet eine gleichzeitige Verarbeitungsform von 16 aufeinanderfolgende 32-Bit Speicherbänke in einem halben Warp.

Der Zugriff auf die einzelnen Speicherbänke des On-Chip Memory erfolgt unmittelbar. Der Speicher ist nicht mit Latenzen im Zugriff belastet, wie es für den globalen Speicherbereich vorliegt. Eine Zeitverzögerung der Datentransaktionen wird nur über mehrere Speicherzugriffe auf dieselbe Speicherbank verursacht. Die in diesem Fall auftretenden Zugriffskonflikte werden in hintereinander abfolgende Speicherzugriffe umgewandelt. Die Serialisierung dieser Operationen besitzt folglich eine erhöhte Verarbeitungszeit der einzelnen Warps im Speicherzugriff. Diese Problematik ist auch nicht mit einer Überlagerung von arithmetischen Operationen zu beseitigen, da es sich in diesem Fall nicht um Verzögerungen in der Abarbeitung einer einzigen Speicherinstruktion, sondern um eine Mehrfachausführung von Speicheroperationen handelt.

Die grundlegende Struktur von konfliktfreien „Shared Memory"-Zugriffen erfolgt über die einmalige Ansteuerung der einzelnen Speicherbänke durch jeweils einen Thread eines halben Warps. Zugriffskonflikte können nur innerhalb des halben Warps auftreten. Die konfliktfreien und konfliktbehafteten Zugriffsmuster[28] sind in Abbildung 3.7 veranschaulicht.

[28] Analyse im Visual Profiler: warp serialize (siehe Anhang B)

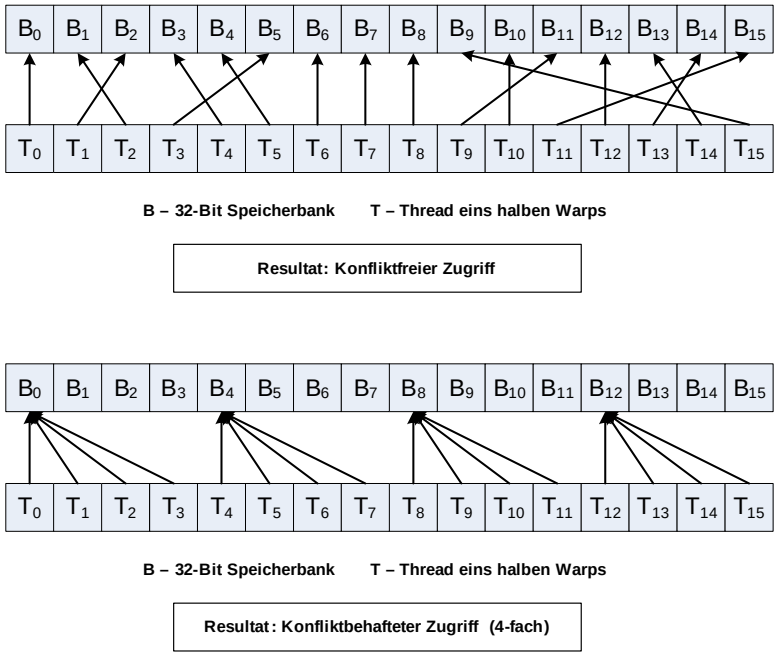


Abbildung 3.7: Zugriffsmuster im „Shared Memory"-Zugriff

Die korrekte Ansteuerung von Shared Memory Datenelementen (Größe 32-Bit) in Form eines Arrays ist wie folgt möglich:

```
__shared__ float data[32];
float var = data[threadIdx.x];
```

Prinzipiell ist der Mehrfachzugriff auf die einzelnen Speicherbänke durch Threads eines halben Warps stets zu vermeiden, da n-Zugriffe auf eine Speicherbank in n-Speicheroperationen in serialisierter Form umgesetzt werden. Allerdings ist für die Leseoperationen eine Funktionalität implementiert, die das gleichzeitige Auslesen durch mehrere Threads erlaubt, ohne Speicherkonflikte zu verursachen. Die Funktionalität wird im CUDA Programming Guide [NVI09a]als Broadcast bezeichnet, bei der mehreren Threads simultan dasselbe 32-Bit Datenelement zugesandt wird. Hierbei ist jedoch zu beachten, dass diese Methode nur gleichzeitig für eine Speicherbank innerhalb des halben Warps ausgeführt werden kann. Diese Methode ist somit nur im Zusammenhang mit einem Auslesevorgang durch eine große Anzahl von Threads einzusetzen, veranschaulicht in Abbildung 3.8.

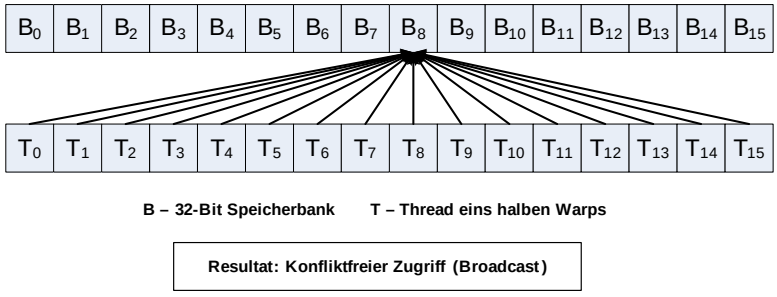


Abbildung 3.8: Konfliktfreier Lesezugriff auf den „Shared Memory" über Broadcast

Im Einsatz des Shared Memory für die Datenhaltung von Datentypen, die nicht der Größe von 32-Bit entsprechen, ist zu beachten, dass über die übliche Indizierung von Datenelementen in einem Array, Zugriffskonflikte entstehen würden. Um diese Problematik an einem Beispiel fest zu machen, sei eine Betrachtung der Abspeicherung von vier Datenelementen des Typs char (Größe 8-Bit) in einer 32-Bit Speicherbank vorzunehmen. Die Speicherzugriffe auf diese vier Elemente erfolgen über vier Speichzugriffe auf eine Speicherbank, welches folglich zu Konflikten führt. Diese Problematik ist beispielsweise über die folgende Indizierung eines char-Arrays im Shared Memory zu beseitigen:

```
__shared__ char data[128];
char var = data[threadIdx.x*4];
```

Des Weiteren besteht eine Lösungsmöglichkeit dieser Problematik über die Zusammenfassung in eine einzige 32-Bit Speichertransaktion (Vektordatentyp char4).

Die Behandlung von 64-Bit oder 128-Bit Speichertransaktionen ist in der Dokumentation nicht ganz eindeutig dargestellt. Diese Zugriffe werden augenscheinlich in einzelne 32-Bit Operationen mit versetzter Adressierung der Speicherbänke[29] (64-Bit Zugriff -> Versatz von einer Bank) umgesetzt. Diese Problematik gilt ebenfalls für einzeln organisierte 32-Bit Speicherzugriffe mit Versatz in der Adressierung. Dieses Zugriffsmuster lässt aus dem Konstrukt der Speicherorganisation in 16 Speicherbänken folgern, dass in einer Adressierung in dieser Form die Speicherbänke zum Teil wiederverwendet (Mehrfachzugriffe) oder ausgelassen werden, veranschaulicht in Abbildung 3.9.

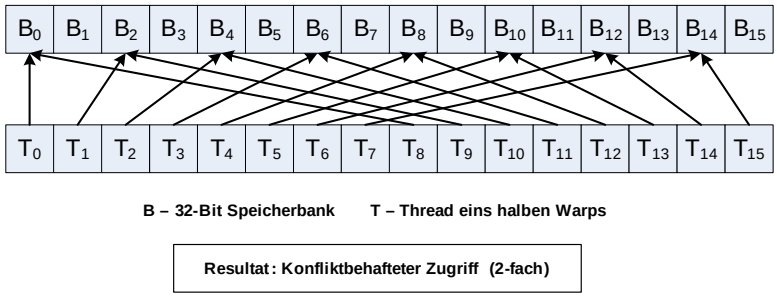


Abbildung 3.9: Versetzter Zugriff /„Stride-Access" um eine 32-Bit Speicherbank

Durch die Erfahrung der Implementierung von Algorithmen in CUDA-Programmelementen hat sich der Einsatz von Shared Memory in dieser Form als problematisch herausgestellt. Die Lösung dieser Problemstellung ist über die Aufteilung der Speicherelemente in separate Shared Memory Arrays mit 32-Bit großen Datenelementen oder die Indizierung des Arrays mit Offset(für die Separierung) zu erreichen.

```
// Separate Arrays
__shared__ float data1[32];
__shared__ float data2[32];
float2 var = input[threadIdx.x];
data1[threadIdx.x] = var.x;
data2[threadIdx.x] = var.y;
```

```
// Datenablage mit Offset
__shared__ float data[64];
float2 var = input[threadIdx.x];
data[threadIdx.x] = var.x;
data[threadIdx.x + 32] = var.y;
```

Grundlegend ist zu beobachten, dass die Datentransaktionen mit Datentypen mit einer Größe ungleich 32-Bit zu Problematiken im „Shared Memory"-Zugriff führen können und wenn möglich zu

[29] Im CUDA Programming Guide[NVI09a] als „stride" bezeichnet

vermeiden sind. Anderenfalls sind dieses in aufeinander folgende 32-Bit Speicheroperationen zusammenzufassen bzw. mit Versatz zu indizieren (wenn kleiner als 32-Bit) oder in der oben präsentierten Form zu separieren (wenn größer als 32-Bit).

Die Reflexion dieser dargestellten Vorteile und Problemstellungen im Einsatz des Shared Memory erfordern eine grundsätzliche Aussage, die im Verhältnis zu dem Einsatz des globalen Speicherbereiches steht. Der erhebliche Geschwindigkeitsvorteil des konfliktfreien Zugriffs auf den Shared Memory spricht grundlegend für den Einsatz dieses Speicherbereichs. In dem Falle von unvermeidbaren schwerwiegenden Zugriffskonflikten ist der globale Speicherbereich, in der bereits dargestellten optimierten Form, vorzuziehen. Die Latenzen des globalen Speichers können verschleiert und die Zugriffe minimiert werden, die Zeitverzögerung, resultierend aus den Mehrfachzugriffen auf den Shared Memory, allerdings nicht. Eine allgemeine Empfehlung ist nicht fest definierbar. Es ist im Entwicklungsvorgang der Algorithmusimplementierung jeweils der effektive Einsatz der einzelnen Speicherbereiche zu untersuchen.

Texture and Constant Memory

Die Texture und Constant Memory READ-Only-Speicherform im globalen Speicherbereich verfügen über den Einsatz von gecachten Speicherzugriffen mit Geschwindigkeitsvorteilen für ihre vorgesehenen speziellen Einsatzziele. Der Zugriff auf die im Cache abgelegt Datenelemente erfolgt in einer mit den Registern vergleichbaren Zugriffsgeschwindigkeit.

Der Constant-Speicherbereich ist in dem Fall zu Hilfe zu ziehen, wenn eine große Anzahl von Threads auf den gleichen Wert zurückgreifen muss. Der Constant-Speicherbereich ist als Hilfsmittel zu verstehen, um die Argumentenliste der Funktionsparameter im Kernel zu minimieren. Erfolgt ein Speicherzugriff eines halben Warps auf dieselbe Adresse im Constant Memory, ist die Zugriffsgeschwindigkeit mit der eines Zugriffs auf ein Register zu vergleichen. Die Umsetzungsform im Programmcode ist in dem vorangegangen Abschnitt 3.2 veranschaulicht.

Der Texture-Speicherbereich ist für den Auslesevorgang von Daten mit zweidimensionaler Anordnung im globalen Speicher optimiert. Dieser Speicher ist für den Speicherzugriff eines halben Warps auf nahe beieinander liegenden Adressen vorgesehen. Außerdem ist eine Normalisierung von 8- und 16-Bit Festkommawerte auf 32-Bit Fließkommawerte (Normalisierung auf Werte [von, bis]: [0.0, 1.0] oder [-1.0, 1.0])[30] ermöglicht.

Der übliche Einsatz erfolgt in zweidimensionalen Texture-Elementen, als eine 2D-Texture-Referenz.

Eine globale Definition der Texture-Referenz ist hierbei Voraussetzung:

```
texture<Type, Dim, ReadMode> texRef;
```

In der Deklaration erfolgt die Definition des Datentyps, sowie die Angabe der Dimensionsanzahl und des Lesemodus.

In diesem Prozess erfolgt eine Farbkanaldefinition und eine Einbindung eines des allokierten Speichers im Global Memory in die Texture-Referenz. Der Abschnitt 3.2.4 im CUDA Programming Guide[31] beinhaltet eine intensive Erläuterung der Implementierungsvarianten unter Einsatz des Texture-Speicherbereichs.

[30] Quelle: [NVI09a]S.89
[31] [NVI09a]S.27 ff

In Abschnitt 5.2.3 ist der Einsatz des Texture Memorys an einem praktischen Beispiel veranschaulicht.

3.4.4 Arithmetische Optimierung

Die Architektur eines Grafikprozessors ist generell für Fließkommaoperationen ausgelegt. Die zum Zeitpunkt der Erstellung dieser Arbeit existierende CUDA-Hardware ist dementsprechend für Fließkommaoperationen konzipiert und besitzt somit den größten Instruktionsdurchsatz in „Single-Precision-Floatingpoint-Operations", die in der Programmiersprache C über den Einsatz des Datentyps float bekannt sind. Der Einsatz von Festkommaoperationen ist in dieser Hardware erlaubt, jedoch bestehen bestimmte Einschränkungen im Instruktionsdurchsatz. Die nach dem Compute Capability Standard 1.3 spezifizierten Grafikprozessoren verfügen über „Double-Precision-Floatingpoint"-Recheneinheiten in ihren Multiprozessoren. Der Einsatz dieser Operationen ist allerdings möglichst zu vermeiden. Das Verhältnis der Anzahl von Einfach-Genauen-Fließkommarecheneinheiten zu den Doppelt-Genauen-Fließkommarecheneinheiten ist 8:1, mit anderen Worten „Single- Precision -Floatingpoint-Operations" erfolgen in achtfach höherer Geschwindigkeit als „Double-Precision-Floating-Point-Operations". Die generelle Empfehlung besteht darin, sich auf den Einsatz der Fließkommaarithmetik mit einfacher Genauigkeit zu beschränken.

Der Instruktionsdurchsatz wird gemäß Dokumentation[32] über die Anzahl der Operationen pro Taktzyklus ($\frac{n\,OPs}{Taktzyklus}$) in einem Multiprozessor beschrieben. Eine Warp-Instruktion (SIMT) besitzt eine Anzahl von 32 Operationen. Aus diesem Grund drückt sich der tatsächliche Instruktionsdurchsatz eines Warps in $\frac{32}{n}\,Taktzyklen$ aus. Diese Bewertungsform drückt den Instruktionsdurchsatz eines CUDA-Multiprozessors aus. Im Einsatz des Visual Profiler ist hierbei zu beachten, dass dieser den Instruktionsdurchsatz in einem Verhältnis der ausgeführten Operationen zur maximalen Anzahl von 32-Bit Fließkommaoperationen($MAX\,FP32$) in einer bestimmten Zeitspanne($\frac{n\,OPs}{MAX\,FP32\,OPs}$) darstellt.

Die in Fließkommaoperationen (einfache Genauigkeit) umgesetzten Additionen, Multiplikationen und Multiply-Add-Operationen besitzen einen Durchsatz von $\frac{8\,OPs}{Taktzyklus}$ (4 Taktzyklen pro Warp), die der Division lediglich $\frac{0.88\,OPs}{Taktzyklus}$ (Implementierte Funktion: `__fdividef(x, y)` mit $\frac{1.6\,OPs}{Taktzyklus}$). Der Einsatz der Division ist auf Grund des geringeren Instruktionsdurchsatzes logischerweise zu meiden.

Die Spracherweiterung in CUDA stellt eine Reihe von mathematischen Funktionen für die Fließkommaarithmetik zur Verfügung, die im Detail unterschiedliche Instruktionsdurchsätze besitzen, siehe CUDA Programming GUIDE[33]. Bei der Auswahl der Funktionen ist zu beachten, dass die in der Form „`__MATHFunktion()`" definierten mathematischen Funktionen (automatisch eingesetzt über den NVCC Parameter „`-use_fast_math`") eine geringere Ausführungszeit als die „`MATHFunktion()`" Methoden benötigen. Allerdings liefern sie weniger akkurate Ergebnisse.

Beim Einsatz der Festkommaarithmetik ist zu beachten, dass eine Integer-Addition ebenfalls mit einem Durchsatz von $\frac{8\,OPs}{Taktzyklus}$ und die 32-Bit Integer-Multiplikation nur mit einem $\frac{1}{4}$ der Verarbeitungsgeschwindigkeit ($\frac{2\,OPs}{Taktzyklus}$) ausgeführt wird. Diese Problemstellung ist auch nicht mit 8-Bit oder 16-Bit Integerdaten zu lösen, da diese Daten prinzipiell für die Berechnungen in einen

[32] [NVI09b] S.43
[33] [NVI09a] S. 77 ff zu entnehmen sind

32-Bit Integerwert transferiert werden. Gemäß der aktuellen CUDA-Hardwarespezifikationen ist es möglich, eine 24-Bit Integer-Arithmetik über die implementierte Funktion `__[u]mul24` einzusetzen, die einen Instruktionsdurchsatz von $\frac{8\,OPs}{Taktzyklus}$ besitzt. Des Weiteren haben Integer-Divisionen und Modulo-Operationen einen geringen Instruktionsdurchsatz. Diese sollten möglichst vermieden oder durch bitweise Operationen ersetzt werden. Die bitweisen Operationen besitzen den maximalen Instruktionsdurchsatz von $\frac{8\,OPs}{Taktzyklus}$. Divisions- und Modulooperationen zur Basis 2 sind ohne Probleme in bitweise Operationen umzusetzen. In diesen Fällen kann prinzipiell die Operation `x/n` in `i>>ld(n)` und die Operation `x%n` in `x&(n-1)` transformiert[34] werden (n ist Vielfaches von 2) um einen erhöhten Instruktionsdurchsatz zu erzielen.

Im Verhätnis zu den arithmetischen Operationen ergibt sich ein Instruktionsdurchsatz für Speicheroperationen von $\frac{8\,OPs}{Taktzyklus}$, allerdings sind hierbei die Latenzen des globalen Speichers von 400 bis 600 Takten zu beachten.

3.4.5 Streamorganisation

Die Verarbeitungskoordinierung der Devicecodeelemente in Streams ermöglicht eine überschaubare Ablaufsteuerung in komplexen Softwarekonzepten unter dem Einsatz von CUDA. Die Streams sind für die asynchrone Ablauforganisation auf der CUDA-Hardware verantwortlich. Über die Streams ist eine Festlegung der Ablaufreihenfolge der Datentransfer- und Kernelausführungen auf der CUDA-Hardware realisiert. Die Streamorganisation ermöglicht in der aktuellen Hardware keine nebenläufige Ausführung von Kerneln. Allerdings ist eine Einsparung von Leerlaufzeit in der GPU über diese Ausführungsstruktur realisierbar. Die Initialisierung(der Start) der Kernel obliegt dem Host, die Kontrolleigenschaften in der Ausführung dem Device, das die Ablauforganisation über die definierten Streams vornimmt.

Der Einsatz von „Page-Locked-Memory"(siehe Abschnitt 3.4.5 - Host-Memory) und die asynchronen Datentransfermethoden der API bilden die Grundlage für eine nebenläufige Datentransfer- und Kernelausführung in verschiedenen Streams (Voraussetzung: Mindestens Compute Capability 1.1). Dies dient der Optimierung von Softwareelementen für die CUDA-Architektur mit hohen Transfermengen von unabhängigen Datensätzen, die in mehreren Kerneln verarbeitet werden können bzw. müssen. Die Ausführungsorganisation ist schematisch in Tabelle 3.4 dargestellt.

Kombinationen	Stream 1: Kopiervorgang	Stream 1: Kernelausführung
Stream 2: Kopiervorgang	Nur Sequentiell	Nebenläufig oder Sequentiell
Stream 2: Kernelausführung	Nebenläufig oder Sequentiell	Nur Sequentiell

Tabelle 3.4: Mögliche Ablaufreihenfolge auf meheren Streams

Der Ablauf in der CUDA-Hardware erfolgt über das folgende, grundlegende Schema:

Die Funktion, die als erstes aufgerufen wird, wird auch als erstes im Device abgearbeitet. Ein Programmelement wird erst ausgeführt, wenn die vorherige Funktion auf demselben Stream beendet ist. Aus diesem Grund ist es möglich, dass hintereinander aufgerufene Kernel auf unterschiedlichen Streams trotzdem in einer anderen Reihenfolge ausgeführt werden können. Dies geschieht beispielsweise dann, wenn der Kopiervorgang eines Streams immer noch aktiv ist. Ist ein Kernel

[34] [NVI09b] S.43,44

auf einem anderen Stream schon ausführungsbereit, wird dieser ausgeführt. Die Abbildung 3.10 veranschaulicht mehrere Szenarien, die im Einsatz von Streams auftreten können.

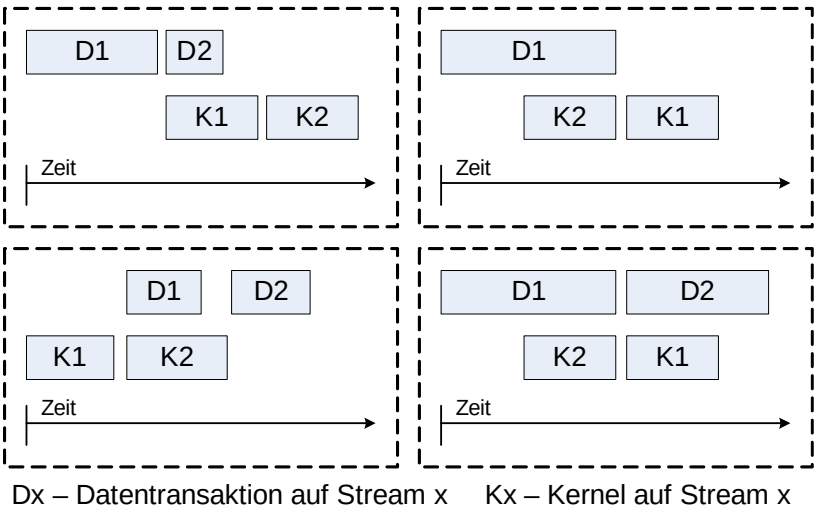


Abbildung 3.10: Ablaufstruktur in Streams

Unglücklicherweise ist eine visuelle Darstellung der nebenläufigen Ausführungen der Kopier- und Kernelausführungen im Visual Profiler nicht möglich, da augenscheinlich eine komplette Serialisierung der Streams durchgeführt wird, um die exakte Ausführungszeit der einzelnen Elemente zu bestimmen. Die Resultate der erläuterten Implementierungsform sind jedoch in den Ausführungszeiten der verschiedenen Implementierungsformen, mit und ohne separate Streams, zu beobachten. Die Veranschaulichung der Geschwindigkeitsvorteile in der Streamkoordinierung anhand von entwickeltem Programmcode erfolgt in Abschnitt 5.1.

Kapitel 4

Versuchsumgebung

4.1 Eingesetzte Hard- und Software

Die Versuchsumgebung besteht aus einer Hard- und Softwareumgebung auf Basis eines Personal Computers. In diesem Abschnitt sind die entscheidenden Elemente der Versuchsumgebung aufgeführt.

Hardwareplattform

Die für CUDA eingesetzte Grafikkarte ist die NVIDIA Geforce GTX 260 (Basis: GT200 Grafikprozessor). Diese ist nach dem Compute Capability Standard 1.3 spezifiziert und besitzt eine Anzahl von 27 Multiprozessoren, sowie ein 448 Bit-GDD3-Speicherinterface mit 896 MB DRAM. Als Schnittstelle zum Host ist ein PCI-Express-Bus x16 2.0 im Einsatz. Diese Grafikkarte ist als Coprozessoreinheit im Einsatz ohne Aufgaben für die Grafikanzeige. Der gesamte Grafikspeicher steht für die Entwicklung in CUDA zur Verfügung.

Die technischen Daten sind in einer Übersicht in Tabelle 4.1 aufgeführt.

Bezeichnung	GPU	Speicher	Speicherbus	Speichertyp	Compute Capability	SMs
Geforce GTX 260	GT200	896 MB	448 Bits	GDDR3	1.3	27

	Grafikprozessor	Recheneinheiten	Speicher
Taktraten in MHz	625	1348	1100 (Effektiv 2200)

Tabelle 4.1: Device: Technische Daten Grafikkarte

Der Host verfügt über einen AMD Phenom II X2 550 CPU und 4 GB DDR2-800 Arbeitsspeicher. Die technischen Daten sind in folgender Tabelle aufgeführt.

Bezeichnung	Anzahl Kerne	Cache	Speicherbandbreite	Speichertyp	Speicher
AMD Phenom II X2 550	2	ca. 7 MB	128 Bits	DDR2	4 GB

	CPU Kerne	RAM
Taktraten in MHz	3100	400 (Effektiv 800)

Tabelle 4.2: Host: Technische Daten

Die theoretisch mögliche Übertragungsraten zwischen Host und Device über den PCIe-Bus x16 2.0 sind in folgender Tabelle aufgeführt.

	Host zu Device	Device zu Device
Transferrate in GB/s	8 (RAM: 12,8)	123200

Tabelle 4.3: Theoretische Werte

Die Ermittlung der praktisch möglichen Übertragungsraten zwischen Host und Device[1] über den PCIe-Bus x16 2.0 (max. 8 GB/s) sind in folgender Tabelle aufgeführt.

	Host zu Device	Device zu Host	Device zu Device
Pageable Memory	2460 GB/s	1700 GB/s	102074 GB/s
Page-Lockt Memory	5744 GB/s	5059 GB/s	101652 GB/s

Tabelle 4.4: Übtragungsraten der verschiedenen Host-Memory-Typen

Softwareplattform

Die Hardwareplattform wird mit dem Betriebssystem Microfsoft Windows XP x64 mit SP2 betrieben (64-Bit Betriebssystem). Für die Softwareentwicklung ist die Entwicklungsumgebung Microsoft Visual Studio 2008 im Einsatz.

Für die CUDA-Entwicklung sind folgende Softwareelemente im Einsatz:

- ▷ Der NVIDIA Geforce Grafikkartentreiber in der Version 190.38
- ▷ Das NVIDIA CUDA Toolkit 2.3
  - – Enthält API, zusätzliche Bibliotheken, die Dokumentation und den Visual Profiler
- ▷ Das NVIDIA CUDA SDK2.3
  - – Enthält Beispielcode und zusätzliche Tools ([NVI09f])

Die aufgeführten Elemente sind in der 64-Bit Version für das 64-Bit Betriebssystem im Einsatz. Der erstelle Programmcode ist passend für Hard- und Softwareplattform (64-Bit System) kompiliert.

4.2 Programmierumgebung

Die Programmcodeentwicklung für diese Arbeit findet grundsätzlich in der Visual Studio 2008 Entwicklungsumgebung statt, unter dem Einsatz der Programmiersprache C/C++ mit den allgemeinen und speziellen Erweiterungen in „C for Cuda". Als Grundlage für die Implementierung diente die CUDA-Dokumentation ([NVI09a], [NVI09b] und [NVI09f]).

Diese Arbeit umfasst mehrere Programmierprojekte (in Visual Studio als Solutions organisiert). Dabei handelt es sich um zwei Projektvorlagen, die als Grundlage für die in den folgenden Abschnitten dokumentierten Implementierungen dienen.

[1] Ermittelt über bandwidthtest.exe in CUDA SDK [NVI09f]

Die erste Vorlage veranschaulicht ein allgemeines Projekt. Es enthält vorimplementierte Funktionen für die Device-Initialisierung, Fehlerbehandlung und für die grundlegende Struktur[2] in CUDA-Softwareelementen. Die grundlegende Struktur umfasst die dazugehörenden Verarbeitungs- und Aufrufschritte, die in Abschnitt 3.1.4(Allgemeiner Programmablauf in CUDA) und 3.2.3(Funktionen in „C for Cuda") aufgeführt sind. Dieses Projekt dient als Grundlage für den in Abschnitt 5.1 präsentierten FFT-Algorithmus.

Die zweite Projektvorlage basiert ebenfalls auf der ersten. Sie stellt somit eine Erweiterung für die Bild- und Bildstromverarbeitung (Imagestream) dar. Dieses Projekt enthält eine zusätzliche komplexe Struktur, um wiederum vereinfacht neue Dateiformate und Datenquellen mit einzubinden. Diese Funktionalität ist über ein spezielles Entwurfsmuster realisiert, das eine allgemeingültige Überführung in den Verarbeitungsprozess vornimmt. Die in der Programmiersprache C++ ermöglichte objektorientierte Struktur lässt, über den Einsatz von Klassenvererbung, eine Verarbeitung von Bilddaten in einer allgemeinen Form zu. Die in einer Grundklasse (Abstrakte Klasse) implementierten Funktionen stellen die verallgemeinerten Funktionen zur Verfügung, um den Programmablauf mit verschiedenen Dateiformaten und Datenquellen über die gleichen Funktionen zu steuern. Die Funktionen, Datei-Öffnen/Daten-Laden, Algorithmus-Ausführen und Speiern/Schließen, sind in dieser Klasse implementiert. Diese können über eine Vererbung für jede Datenquelle spezifiziert werden, ohne dass diese Vorgänge nach außen sichtbar werden müssen. Zusätzlich existiert eine weitere Klassenhierarchie, um einen beliebigen Austausch von Verarbeitungsalgorithmen vorzunehmen zu können.

Die Vorteile dieser Struktur drücken sich über eine Erweiterbarkeit des Programmcodes ohne intensiver Veränderung bzw. Überarbeitung der existierenden Programmelemente aus. Die hinzukommenden Elemente bauen auf dem Grundkonzept auf und verändern dieses nicht. Das gesamte Konzept lässt einen beliebigen Austausch von Datenquellen und Verarbeitungsalgorithmen zu. Es erlaubt durch die aufeinander angepasste Struktur jegliche Kombination von Datenquellen und Algorithmen auf Basis der geschaffenen Klassenstruktur.

Der Grund für die verallgemeinerte Klassenstruktur des vorgestellte Vorlageprojekts ist wie folgt. In der Entwicklung von CUDA-Softwareelementen für die Bildverarbeitung ist es ermöglicht, die als „OpenGL[3] Interoperability" bezeichnete Methode einzusetzen. Hierüber ist es in CUDA realisierbar, ein OpenGL-Bufferobject in den globalen Speicher des Devices einzubinden. Anhand diese Bufferobjects sind die Ergebnisse der Bilddatenverarbeitung direkt aus dem globalen Speicher darzustellen. In dieser Implementierungsform ist die kostenlos einsetzbare C Bibliothek „GLUT - The OpenGL Utility Toolkit"[4] für die Darstellung der Berechnungsergebnisse in einem Grafikfenster zuständig. Die Überführung dieser C Bibliothek in eine C++ Klasse ist anhand des Entwurfsmusters „Singleton" (siehe [EG05]) realisiert, um eine Mehrfachinstanziierung dieser Klasse zu vermeiden. Die Struktur der C Bibliothek lässt eine Mehrfachinstanziierung nicht zu (Objekt wird maximal einmal erstellt). Um für jede neue Implementierung von neuen Datenquellen oder Algorithmen keine Änderungen in der OpenGL-Klasse vornehmen zu müssen, ist die vorweg präsentierte Klassenstruktur implementiert. Als Quelle für die Erstellung dieser Klasse diente die GLUT API [MJK96], die CUDA SDK [NVI09f] und [GL05]. Die gesamte Klassenstruktur ist in Abbildung 4.1 dargestellt.

[2] siehe Anhang C.1
[3] Eine API für Computergrafik in zwei und drei Dimensionen
[4] Quelle: GLUT API [MJK96] und CUDA SDK [NVI09f] enthalten (Eingesetzt für die Entwicklung der Klasse)

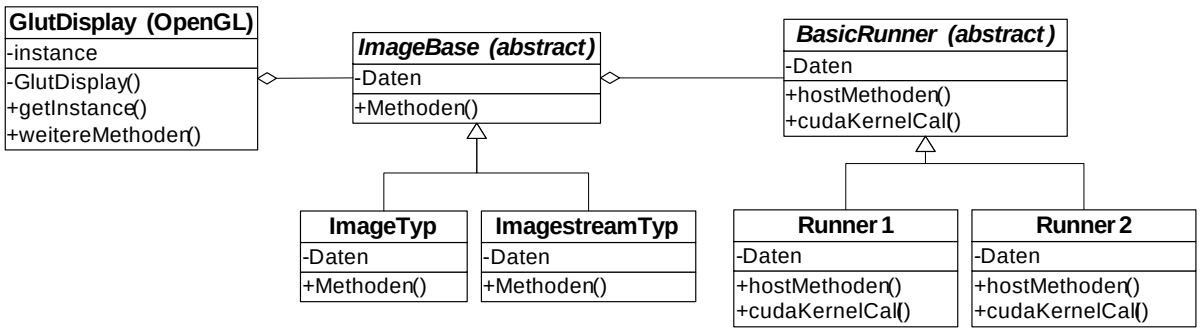


Abbildung 4.1: Klassenstruktur

Die beiden Vorlageprojekte ermöglichen, ohne großen Aufwand, eine direkte Implementierung von Algorithmen mit den dazugehörigen Randbedingungen. Die grundlegenden Probleme des Datenaustausches, der Fehlerbehandlung, sowie das Auslesen und Verarbeiten von Bilddaten mit der direkten Ergebnisdarstellung in einem Fester (mit OpenGL/Glut), ermöglichen die Konzentration auf die wesentlichen Elemente der Devicecode-Entwicklung. Die Wahl der Entwurfsmuster im Vorlageprojekt ermöglicht eine vereinfachte Funktionalität für die Erweiterung mit neuen Dateiformaten und neuen Algorithmen. Dieses Erweiterungsverfahren bietet einen einfachen Weg, um Vergleiche von Algorithmen in Host- und Device-Implementierungsform durchzuführen.

Für die Implementierung von Bildverarbeitungsalgorithmen ist das PNM-Bilddateiformat („Portable Any Map"[5]), für die Verarbeitung von unkomprimierten Graustufen- und RGB-Bilddaten, im Projekt eingebunden. Für die Verarbeitung von Bilderströmen ist ein Bilderstrom für unkomprimierte Bilddaten (Formatbezeichnung: CLM), basierend auf dem Bilddateiformat BMP[6] (Windows Bitmap), implementiert. Die Datei enthält einen einfach strukturierten Datei-Header, der die Informationen über die Bildgröße (Width and Height), die Anzahl der Farbkanäle, die Bilderanzahl und die Datenmenge der Bilder enthält.

Die Dateistruktur ist in der Projektorganisation für die C++-Klassen und CUDA-Programmelemente getrennt, um eine eindeutige Übersicht und Funktionalität zu gewährleisten. Die Trennung findet über *.cpp- und *.cu-Dateien statt. Dabei enthalten die *.cpp-Dateien den C++ spezifischen Programmcode mit den bereits beschriebenen Klassen, die wiederum, die entwickelten C-Funktionen der CUDA-Programmelemente aus den *.cu-Dateien aufrufen. Der Kompilierungsvorgang erfolgt über den in Abschnitt 3.1.4 veranschaulichten Prozess. Hierbei werden die *.cpp-Dateien über den Visual C++ - Kompilierer in Objektdateien umgesetzt. Die Koordinierung des Kompilierungsprozesses für die *.cu-Dateien übernimmt der NVCC Kompilierer-Treiber.

4.3 Allgemeine Problemstellung und Vergleichsverfahren der Softwareelemente

Die allgemeine Problemstellung in der Datenverarbeitung mit High-Performance-Computing Hardware ist die Datenübertragung. Diese Problemstellung bildet sich auch sehr in der CUDA Programmcodeentwicklung aus, die sich am Beispiel der Host zu Device Datentransaktionen festmacht.

[5] Quelle: [PNM03]
[6] Quelle: siehe [WB] und [MSBS09]

Die in den nächsten Abschnitten dokumentierten Implementierungen von Devicecode beinhalten eine Analyse der Ausführungszeiten mit einem Vergleich zu Algorithmusimplementierungen für den Host (CPU). Die Programmcodeelemente für den Host werden in konventioneller, sequenzieller Programmcodeform auf einem einzigen Thread (ein Kern in der CPU) ausgeführt. In der Analyse findet eine Bestimmung der reinen Rechenzeit und der Ausführungszeit inklusive der Datentransaktionen zwischen Device und Host statt. Auf diesem Weg ist ein eindeutiges Bestimmungsverfahren der Berechnungszeiten möglich. Für die Zeitmessung sind die in „C for Cuda" implementierten „Events" und der CUDA Visual Profiler im Einsatz. Der Hostcode wird für die Messungen für ein 64-Bit System kompiliert (siehe Betriebssystem).

Der Prozess der Datenbeschaffung wird in der Analyse nicht mit einbezogen, da dieser sich von Fall zu Fall unterscheiden kann und keinen direkten Einfluss auf den Algorithmus nimmt. In den in dieser Arbeit dokumentierten Implementierungen sind Daten eingesetzt, die direkt von der Festplatte des Hostrechners gelesen oder vor der Berechnung im Programm erzeugt werden. Dieses Verfahren ist für die Veranschaulichung der Algorithmen völlig ausreichend, jedoch ist im endgültigen Einsatz von CUDA für eine ausreichende Datenübertragung in der Datenbeschaffung zu sorgen, um das Potenzial der CUDA-Hardware ganz entfalten zu können.

Kapitel 5

Realisierung und Analyse von Algorithmen

Die folgenden Abschnitte präsentieren die praktische Anwendung von CUDA über die Implementierung von Signal- und Bildverarbeitungsalgorithmen.

5.1 Implementation eines Radix-2 FFT-Algorithmus

Die schnelle Implementierungsform einer diskreten Fourier-Transformation (Fast Fourier Transformation) ist in ihrem mathematischen Zusammenhang über folgende Formel beschrieben:

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} \left(x_e \cdot w_{\frac{N}{2}}^{kn} \right) + w_N^k \cdot \sum_{n=0}^{\frac{N}{2}-1} \left(x_o \cdot w_{\frac{N}{2}}^{kn} \right) \tag{5.1}$$

Dieser mathematische Zusammenhang ermöglicht den Einsatz der Radix-2 FFT Implementierungsform. Dieser Algorithmus führt eine Berechnung der N-Punkte FFT ($N = 2^d$, $d \epsilon N$) mit $\frac{N}{2} \times ld(N)$ komplexe Multiplikationen und $N \times ld(N)$ komplexe Additionen, anstatt der Anzahl $N \times N$ komplexen Multiply-Add-Operationen in einer nicht optimierten diskreten Fourier-Transformation (DFT) durch.

5.1.1 Beschreibung des Algorithmus für die parallele Implementierung

Der interessante Aspekt dieses Algorithmus liegt in seiner Fähigkeit, in paralleler Form ausgeführt zu werden. Die schematische Veranschaulichung des Algorithmus anhand einer 8 Punkte FFT ist in Abbildung 5.1 dargestellt.

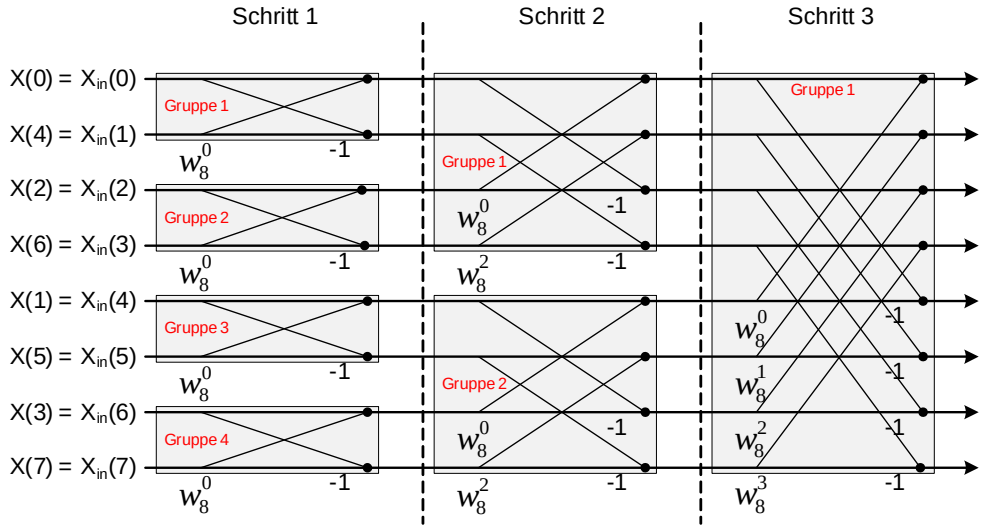


Abbildung 5.1: Radix-2 FFT mit 8 Punkten

Diese Abbildung veranschaulicht das Parallelisierungspotenzial des Algorithmus und das daraus folgende Beschleunigungspotenzial einer parallelen Implementierung unter CUDA. Dabei stellt der sogenannte Butterfly die Grundoperation in der FFT dar. Die FFT-Berechnung findet in mehreren Schritten($ld(N)$) mit jeweils $\frac{N}{2}$-Butterfly-Operationen pro Schritt statt. Es gilt hierbei, jeden einzelnen Schritt zu parallelisieren.

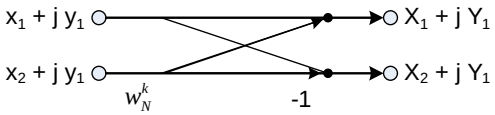


Abbildung 5.2: Butterfly

Die in Abbildung 5.2 veranschaulichte Butterfly-Operation drückt sich in folgendem mathematischen Zusammenhang aus:

$$Twiddlefactor: \; w_N^k = e^{-j\frac{2\pi k}{N}} = \cos\left(\frac{2\pi k}{N}\right) - j \cdot \sin\left(\frac{2\pi k}{N}\right) \tag{5.2}$$

$$X_1 + j \cdot Y_1 = (x_1 + j \cdot y_1) + w_N^k \, (x_2 + j \cdot y_2) \; und \; X_2 + j \cdot Y_2 = (x_1 + j \cdot y_1) - w_N^k \, (x_2 + j \cdot y_2) \tag{5.3}$$

$$C_x = \cos\left(\frac{2\pi k}{N}\right) \cdot x_2 + \sin\left(\frac{2\pi k}{N}\right) \cdot y_2 \; und \; C_y = \cos\left(\frac{2\pi k}{N}\right) \cdot y_2 - \sin\left(\frac{2\pi k}{N}\right) \cdot x_2 \tag{5.4}$$

$$X_1 = x_1 + C_x, Y_1 = y_1 + C_y \; und \; X_2 = x_1 - C_x, Y_2 = y_1 - C_y \tag{5.5}$$

Die Schlussfolgerung der präsentierten Zusammenhänge lässt eine Verallgemeinerung des Radix-2 Algorithmus über die Struktur der veranschaulichten 8-Punkte FFT zu. In Abbildung 5.1 ist die Bit-reversible Adressierung des Eingangsvektors veranschaulicht. Über diese Adressierung findet im Vorfeld eine Umstrukturierung der Eingangsdaten statt.

Die Adressierung bzw. Indizierung der Eingangswerte in der Butterfly-Operation erfolgt in der folgend dargestellten allgemeinen Form:

Bezeichnung	**Wert**	Berechnung in CUDA
Anzahl der Werte	N	N
Anzahl der Stufen	$stages = ld(N)$	$stages = \frac{log(N)}{log(2)}$(FP-Operation)
Aktueller Schritt	$stage$	$stage$
Anzahl der Butterflies pro Stufe	$\frac{N}{2}$	N >> 1
Anzahl der Gruppen pro Stufe (ANZ_G)	$2^{stages-(stage+1)}$	1 << (stages - (stage + 1))
Anzahl der Butterflys pro Gruppe(ANZ_{BF})	2^{stage}	1 << stage
Schrittweite (stepsize)	2^{stage}	1 << stage
Aktueller Butterfly in der Gruppe	bf	ThreadID % stepsize (Int-Operation)
Aktuelle Gruppe	group	ThreadID / stepsize (Int-Operation)

Tabelle 5.1: Auflistung der relevanten Parameter

Anmerkung: Die Indizierung beginnt in der C üblichen Form mit 0

Mittels der in Abbildung 5.3 bestimmten Daten wird die Butterfly-Operation durchgeführt.

Butterfly(x_0 , x_1 , w_N^k)

$$x_1 = x_{in}\,(bf + 2 \cdot stepsize \cdot group)$$

$$x_2 = x_{in}\,(bf + 2 \cdot stepsize \cdot group + stepsize)$$

$$w_N^k = e^{(-j\frac{2\pi \cdot ANZ_G \cdot bf}{N})} = cos(\tfrac{2\pi \cdot ANZ_G \cdot bf}{N}) - j \cdot sin(\tfrac{2\pi \cdot ANZ_G \cdot bf}{N})$$

Abbildung 5.3: Daten Butterlfy

Die Ergebnisdaten der FFT Berechnungen liegen in einer linearen Adressierung von 0 bis $N-1$ vor.

5.1.2 Umsetzung und Analyse in CUDA

Der vorweg beschriebene Zusammenhang dient als Grundlage für die Implementierung in CUDA.

In der Implementierung sind folgende Grundelemente definiert:

- ▷ **Jede Butterfly-Operation** wird von **einem Thread** in jedem Schritt übernommen
- ▷ Die Abarbeitung der einzelnen Schritte erfolgt über eine for-Schleife
- ▷ Jeder Threadblock berechnet eine separate FFT von N = 2 bis 1024 Punkte
- ▷ Vor der Verarbeitung erfolgt eine Umstrukturierung der Daten (Bit-reversible Adressierung)
- ▷ Pufferspeicher: Shared Memory - Separierung der Real- und Imaginärteile und der Eingangswerte der Butterflies[1] (x1 und x2 sind separiert)

[1] In Optimierungsprozess entfernt

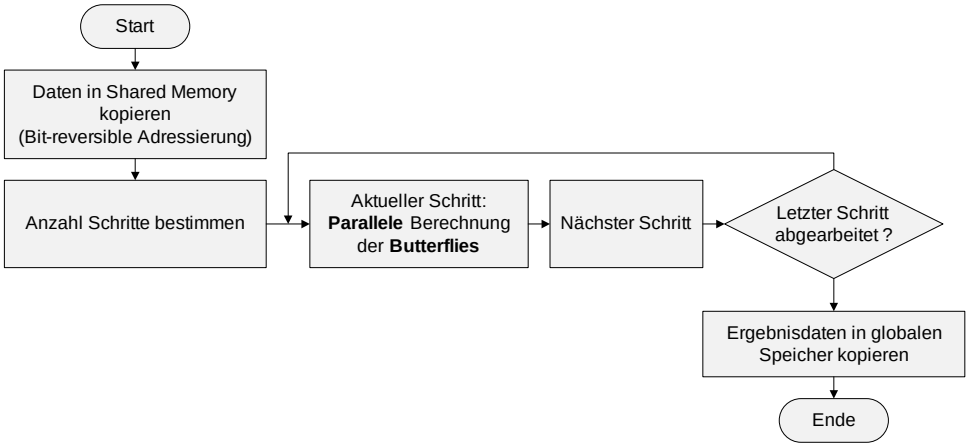


Abbildung 5.4: Schematischer Ablauf

Die Implementierung ist an dem, in Abschnitt 3.4 präsentierten, Entwicklungsleitfaden orientiert. Die Punkte des Leitfadens werden für den FFT-Algorithmus in diesem Abschnitt intensiv diskutiert.

Auf Grund der Struktur des FFT-Algorithmus ist die Threadstruktur **eindimensional** organisiert.

Erste Implementierung

Dieser Implementierungsschritt beschreibt die erste funktionstüchtige Umsetzung mit den ersten bereits integrierten Optimierungen. Der Aufbau ist der so eben vorgestellten Struktur angelehnt. Der Quellcode ist in Anhang D.1 und die Ergebnisse des Visual Profilers in Anhang D.2 dargestellt.

Eingangsdaten für die Veranschaulichung: N=1024 , Threads pro Block = N/2 = 512

Entscheidende Parameter der eindimensionalen Threadstruktur: threadIdx.x, blockIdx.x und blockDim.x

1. **Umstrukturierung durch Bit-reversible Adressierung**

   Die im globalen Speicher liegenden komplexen Eingangswerte werden über eine Bit-reversible Adressierung in den Shared Memory umkopiert.

2. **Struktur des Shared Memory**

   Die in Abschnitt 3.4.3 präsentierten Problematiken im Einsatz des Shared Memory beinhalten den Problemfall mit versetzter Adressierung von Speicherbänken. Aus diesem Grund findet eine Separierung der komplexen Eingangswerte nach Real- und Imaginärteil statt. Ebenfalls erfolgt eine Trennung der Eingangswerte (x(0) / x(1) usw.) für den ersten Schritt. Diese Eingangswerte unterliegen Aufgrund ihrer Zugriffsanordnung der gleichen Problematik.

   Der in Anhang D.1 dokumentierte Quellcode veranschaulicht die Umstrukturierung und Übertragung in den Shared Memory. Die offensichtlich komplexe Adressierung resultiert aus den noch nicht optimal gewählten „Alignments" für die Eingangsdaten, sowie des direkt Bit-reversibl adressierten globalen Speichers. Die Ergebnisse dieser Adressierung sind, laut Visual Profiler, zusammengefasste 32-Byte Speichertransaktionen. Im Idealfall würden jedoch 128-Byte Speichertransaktionen ausgeführt werden.

3. **Abarbeitung der einzelnen Schritte über eine for-Schleife**

   In jedem Schleifendurchlauf wird ein Butterfly von einem Thread (N/2 pro Schritt) berechnet. Der Twiddle-Faktor für den Butterfly ist jedes mal neu zu berechnen, da Arithmetische Operationen den Speicheroperationen vorzuziehen sind. Das in Abschnitt 5.1.1 präsentierte Berechnungsverfahren für die Adressierung der Eingangswerte im Butterfly ist um die Bedingungen der aufgeteilten Daten im Shared Memory zu ergänzen. Diese sind in der ersten Implementierung über Kontrollstrukturen (if-Abfragen) umgesetzt, die leider zu divergenten Ausführungssträngen führen, und wenn möglich, noch zu entfernen sind (Austausch durch Arithmetische Operationen).

   Die Separierung der Eingangswerte für die Butterflies führt leider nicht zu eine kompletten Vermeidung von serialisierten Speicherzugriffen, jedoch fallen diese augenscheinlich nicht schwerwiegend aus. Durch die mehrmalige Wiederholung von Speicherzugriffen in den Schleifendurchläufen bietet der Shared Memory eine effiziente Alternative zum globalen Speicherbereich. Der globale Speicherbereich wäre in diesem Algorithmus durch die mehrmaligen Speicherzugriffe mit wechselnder, versetzter Adressierung nicht optimal einsetzbar.

4. **Rückkopierung der Daten in den globalen Speicher**

   Die Rückkopierung erfolgt mit einer linearen Adressierung und kann somit in 128-Byte Speichertransaktionen ausgeführt werden.

Der Algorithmus birgt in dieser Umsetzungsform noch einiges Optimierungspotenzial. Die entscheidenden Aspekte stellen hierbei der Speicherzugriff, die Eliminierung von divergenten Ausführungssträngen, die arithmetische Optimierung und die Ausrichtung auf maximale Auslastung der Multiprozessoren dar.

Kernelausführungszeit der ersten Implementierung:

	N	**Anzahl FFTs** (Eine FFT pro Block)	**Ausführungszeit**
Wert	1024	1	107µs

Komplette Optimierung

Die Optimierung umfasst die bereits angesprochenen Problemstellungen in der ersten Implementierung. Der optimierte Quellcode ist in Anhang D.3 und die Profiler Ergebnisse sind in Anhang D.4 dokumentiert. Für den Vergleich bestehen die gleichen Voraussetzungen.

Eingangsdaten für die Veranschaulichung: N=1024 , Threads pro Block = N/2 = 512

1. **Arithmetische Optimierung**

   Die implementierten Kontrollstrukturen (if-Abfragen) dienen lediglich für eine Fallunterscheidung und sind durch Modulo-2 Operationen zu ersetzten. Des Weiteren finden im FFT-Algorithmus einige mathematische Operationen zur Basis 2 statt, die ohne Probleme durch bitweise Operationen ersetzt werden können. In diesem Schritt findet eine Optimierung anhand der in Abschnitt 3.4.4 beschriebenen Möglichkeiten statt, inklusive des Einsatzes des NVCC Parameters „`-use_fast_math`". Der komplette Umfang ist dem Quellcode in Anhang

D.3 zu entnehmen. Die Ausführungszeit des Kernels hat sich durch diesen Entwicklungsschritt auf 50 µs minimiert (Eine 1024-Punkte FFT). Jedoch benötigt der Algorithmus auf der eingesetzten Hardware (27 Multiprozessoren) eine Zeit von 71 µs bei 27 parallel verarbeiteten 1024-Punkte FFTs. Dieses Ergebnis deutet auf ein weiteres Optimierungspotenzial in der Speicherorganisation hin, denn die Kernel-Ausführungszeiten sollten im Idealfall nahezu identisch sein.

2. **Maximale Auslastung der CUDA-Hardware**

   Die limitierenden Elemente in der Auslastung der Hardware sind in Abschnitt 3.4.1 veranschaulicht. Die Minimierung des On-Chip-Memory-Einsatzes ist bei der maximalen Auslastung der Multiprozessoren von Bedeutung. Für die vorliegende Hardware ist ein maximaler Einsatz von 16 Registern pro Thread der ideale Zustand ($\frac{16384\,Register}{1024\,Threads} = 16\frac{Register}{Thread}$). Die Minimierung der Anzahl der benötigten Register ist durch die Eliminierung von überflüssigen Variablen und Zusammenfassungen und/oder Minimierung von Rechenoperationen zu erreichen. Um die Flexibilität des Algorithmus für die höchstmögliche Auslastung für jede Anzahl von Punkten zu gewährleisten, ist der Einsatz von dynamischen Shared Memory zwingend. Auf diesem Weg findet eine Zuordnung des benötigten Speichers im Threadblock zum Aufrufzeitpunkt statt, ohne feste Definition in der Kernelfunktion. Die 1024 Punkte FFT benötigt 8192 Byte Shared Memory (Size = 1024*2* sizeof(float)) bei einer Anzahl von 512 Threads (16 Warps) pro Threadblock. Der zusätzlich benötigt Shared Memory für die Aufruf- und Funktionsargumente des Kernels lassen in diesem Fall eine maximale Auslastung[2] von 50 % zu. Der Einsatz von dynamischem Shared Memory für eine 512 Punkte FFT (Für 256 Threads sind 4096 Byte benötigt) erreicht hingegen eine maximale Auslastung der Multiprozessoren von 75 %. Mit statischen Shared Memory von 8192 Byte[3] wären es nur 25%, da durch die fixe Menge von Shared Memory immer nur ein Threadblock auf einem Multiprozessor aktiv sein kann.

3. **Optimierung globaler Speicherzugriffe**

   Die im Quellcode aufgeführte Methode für die nachträgliche Bit-reversible Adressierung ermöglicht eine vereinfachte Adressierung des globalen Speichers in einem 16 Byte Alignment (float4). Das Resultat ist eine Minimierung auf 128 Byte Speichertransaktionen (siehe Anhang D.4).

   Die nachträgliche Bit-reversible Adressierung beinhaltet eine Umsortierung der Eingangswerte im Umspeicherungsvorgang in den Shared Memory. Die Separierung der Eingangswerte für die Butterflies des ersten Schrittes ist mit diesem Optimierungsschritt hinfällig. Die dafür nötige komplexe Struktur der Adressierung des Shared Memory eliminiert den Vorteil dieser Separierung (Führt zu erhöhter Serialisierung von Speicherzugriffen). Eine Minimierung der serialisierten Speicherzugriffe ist über die ausschließliche Trennung der Real- und Imaginärteile der Eingangswerte zu erreichen. Die **Trennung der Butterfly-Eingangswerte** wurde **entfernt**.

Der Optimierungsprozess beinhaltet die Eliminierung der divergenten Ausführungsstränge, die Minimierung von Instruktionen und serialisierten Shared Memory Zugriffen, sowie die Optimierung der globalen Speicherzugriffe, die zu folgenden Kernelausführungszeiten führen:

[2] Maximale Menge Shared Memory pro Multiprozessor: 16384 Byte
[3] Definition von 8192 Byte für maximal mögliche 1024-Punkte FFT

N	**Anzahl FFTs** (Eine FFT pro Block)	**Ausführungszeit**
1024	1	40 µs
1024	27	42,8 µs

Das Optimierungsverfahren beinhaltet mehrere einzelne Optimierungsschritte, die jeweils mit dem Visual Profiler zu untersuchen sind. Die Methode der schrittweisen Optimierung hat sich als sehr effektiv erwiesen, um die Effekte der einzelnen Optimierungsschritte genau zu untersuchen und deren Potenzial zu erfassen.

Die Empfehlung für eigene Devicecode-Implementierungen liegt in einer schrittweisen Optimierung, wie sie in diesem Abschnitt und im Optimierungsleitfaden (Abschnitt 3.4) präsentiert ist. Erfahrung in der CUDA-Entwicklung bewirkt zusätzlich einen Blick auf die in der Optimierung wesentlichen „Brennpunkte" im Devicecode.

5.1.3 Vergleich mit sequenzieller Implementierung auf dem PC

In der Untersuchung von geeigneten Algorithmen für die parallele Implementierung in CUDA ist es von großem Interesse, einen Vergleich mit konventionellem, sequentiellem Programmcode anzustellen, um den Geschwindigkeitsvorteil von CUDA zu ermitteln.

In diesem Abschnitt findet eine Untersuchung der Ausführungszeiten auf der CPU (Hostcode) mit der Implementierung in CUDA statt. Die CUDA Implementierung ist unter dem Aspekt der Kernelausführungszeit und mit Einbezug der Transferzeiten zwischen Host und Device zu untersuchen.

Unterschiede von synchronem und asynchronem Datentransfer

Die erste Untersuchung gilt der Analyse des Einsatzes eines einzigen Verarbeitungsstroms unter der Verwendung von synchronen und asynchronen Datentransferfunktionen (Host zu Device). Die Ergebnisse der Untersuchung sind dem Visual Profiler entnommen.

Testvorrausetzungen:

Anzahl der FFTs (Eine pro Threadblock)	N (Anzahl der Punkte pro FFT)
108	1024

Die synchrone Ausführung führt zu dem in Abbildung 5.5 veranschaulichten Ergebnis und ist folgendermaßen realisiert:

```
// Memcopy von Host zu Device
cudaMemcpy(d_idata,h_data,datasize,cudaMemcpyHostToDevice);
// Kernelausführung (streamId = 0 : Standardstream)
cl_run_kernel(d_idata,d_odata,N,anzahl,0);
// Memcopy von Device zu Host
cudaMemcpy(h_data,d_odata,datasize,cudaMemcpyDeviceToHost);
```

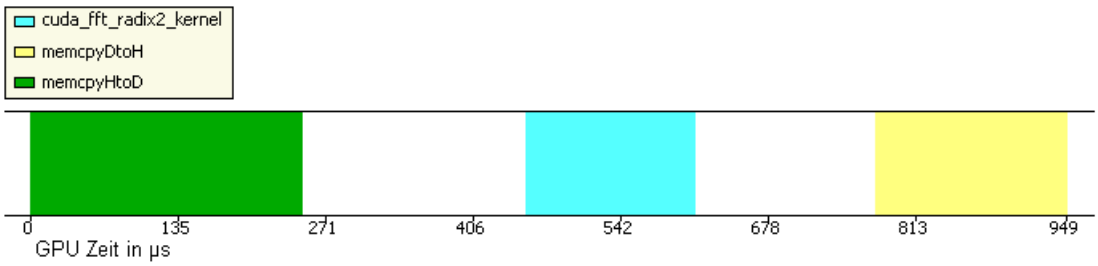


Abbildung 5.5: Ablaufstruktur mit synchronem Datentransfer

Die asynchrone Ausführung führt zu dem in Abbildung 5.6 veranschaulichten Ergebnis und ist folgendermaßen realisiert:

```
// Memcopy von Host zu Device
cudaMemcpyAsync(d_idata,h_data,datasize,cudaMemcpyHostToDevice,0);
// Kernelausführung (streamId = 0 : Standardstream)
cl_run_kernel(d_idata,d_odata,N,anzahl,0);
// Memcopy von Device zu Host
cudaMemcpyAsync(h_data,d_odata,datasize,cudaMemcpyDeviceToHost,0);
```

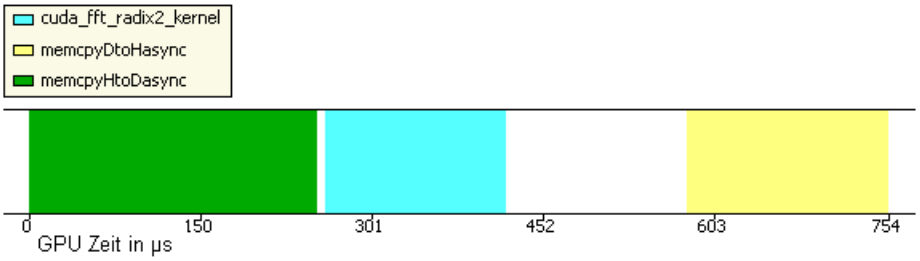


Abbildung 5.6: Ablaufstruktur mit asynchronem Datentransfer

Der Einsatz von asynchron ausgeführten Methoden veranschaulicht die Einsparungsmöglichkeiten von GPU Leerlaufzeiten, die sich auf die komplette Ausführungszeit auswirken.

Der Aufruf von synchronen Datentransferfunktionen durch den Host verursacht Leerlaufzeiten zwischen dem Datenübertragungsabschluss und der Kernelausführung auf dem Device. Erst nach Abschluss der Übertragung erfolgt die Initialisierung des Kernels. Die Kontrolle über die Ausführung obliegt somit, in diesem Fall, dem Host und drückt sich über Wartepausen der GPU aus.

Der asynchron organisierte Hostcode übergibt die Kontrolle an die GPU, der die bestmögliche Abfolge der Programmelemente auf dem Device vornimmt und somit Zeit einspart.

Die Ergebnisse lassen auf den grundsätzlichen Einsatz von asynchronen Datentransferfunktionen schließen. Aus diesem Grund werden diese Funktionen in den weiteren präsentierten Ergebnissen eingesetzt. Für die Ermittlung der Ausführungszeiten sind die „Timer-Subroutinen” im Einsatz, aufgeführt in Anhang C.1.

Untersuchung der Ausführungszeiten mit unterschiedlicher Anzahl von FFTs

Diese Untersuchung dient als Veranschaulichung der Verarbeitungszeiten des Grafikprozessors im Verhältnis zu den Ausführungszeiten der sequentiellen Host-Implementierung. Die Messung beinhaltet einen Vergleich der Ausführungszeiten der GPU mit und ohne Einbezug der Datentransferzeiten zwischen Host und Device. Der Vergleich findet anhand verschiedener Anzahl von FFTs statt.

Testvorrausetzungen:

Anzahl FFTs	N
1 bis 108	1024

Die Abbildung 5.7 veranschaulicht das Verhältnis der Ausführungszeiten von CPU zu GPU.

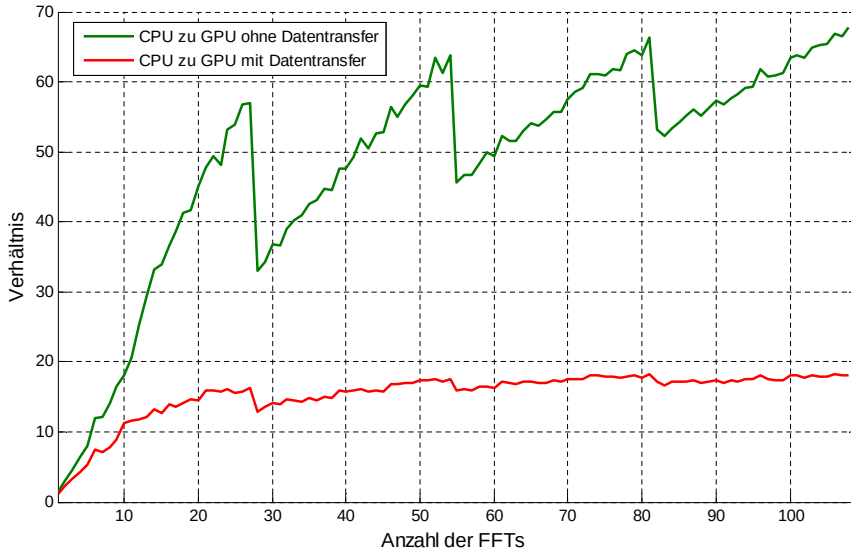


Abbildung 5.7: N FFTs: Verhältnis CPU Zeit zu GPU Zeit

Die Ausführungszeit einer einzelnen FFT bzw. wenigen FFTs zeigen keine signifikanten Unterschiede zwischen der GPU- und der CPU-Implementierung. Die Ausführung von mehreren parallel verarbeiteten FFTs veranschaulicht die enorme parallele Rechenleistung der CUDA-Hardware, die sich in Berechnungsaufgaben für die Berechnung von mehreren unabhängigen FFTs auszahlt. Im Verhältnis der Ausführungszeit ist bei steigender Anzahl von berechneten FFTs eine Konvergenz gegen ein bestimmtes Verhältnis zu beobachten (ca. zwischen 15:1 und 18:1 in den Ausführungszeiten mit Datentransfer). Der Einbezug der Datentransaktionen zwischen Host und Device hat einen erheblichen Einfluss auf die Ausführungszeit. Sie ist somit in jeder Implementierung stets zu berücksichtigen, sogar, wenn immer möglich, zu optimieren, um eine effizientere Ausnutzung des Geschwindigkeitspotenzials der CUDA-Hardware zu realisieren.

Der sägezahnförmige Anstieg der Verhältniskurve von CPU- zu GPU-Zeit ohne Datentransfer (reine Kernelausführungszeit) ist durch die Struktur der eingesetzten CUDA-Hardware zu erklären. Die gemessenen Ausführungszeiten auf der GPU sind in Abbildung 5.8 dargestellt. Jeweils bei der Überschreitung der Anzahl von FFTs von Vielfachen von 27 erfolgt eine sprunghafte Zunahme der Kernel-Ausführungszeit um ca. 30 µs (Ohne Offset: Ausführungszeit ca. verdoppelt). Ansonsten ist bei der Erhöhung der Anzahl der FFTs die Ausführungszeit nahezu konstant. Dies ist bedingt

durch die automatische Skalierung der Threadblöcke (eine FFT pro Threadblock) auf die Multiprozessoren (siehe Abschnitt 2.2.4) und die dafür zur Verfügung stehenden 27 Multiprozessoren. Dieser Effekt verdeutlicht die parallele Verarbeitungsweise der CUDA-Hardware.

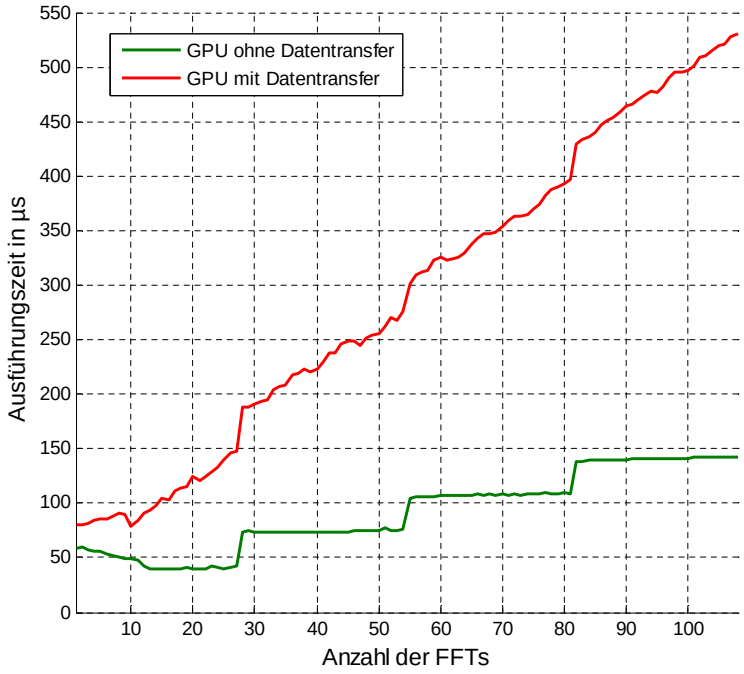


Abbildung 5.8: N FFTs: GPU-Ausführungszeiten in µs

Die Abweichungen zum idealen Kurvenverlauf kann man durch die Messungenauigkeiten und Toleranzen in den Timer-Subroutinen erklären, besonders im Bereich von sehr geringen Ausführungszeiten.

Ein ideales Anwendungsfeld für den implementierten Algorithmus ergibt sich in der parallelen Verarbeitung einer höheren Anzahl von Eingangssignalen oder einer zweidimensionalen FFT in der Bildverarbeitung. Der Einfluss des Host/Device Datentransfers ist immer unter dem Hintergrund zu betrachten, dass für eine Weiterverarbeitung der berechneten FFTs auf der Grafikkarte keine erneute Transaktion nötig ist, da die Daten sich bereits auf der Grafikkarte befinden. In Kombination von verschiedenen Algorithmen fallen die Transferzeiten im Bezug auf die gesamte Verarbeitungszeit weniger ins Gewicht.

Untersuchung der Ausführungszeiten für unterschiedliche N Punkte FFTs

Im Folgenden ist ein Vergleich der einzelnen Ausführungszeiten bei unterschiedlichen N Punkte FFTs (N Punkte zur Basis 2) aufgeführt. Hierbei stellt sich die Frage, ab welchem Zeitpunkt der Einsatz von CUDA für die FFT Berechnung sinnvoll ist. Für diese Messung ist eine hohe Anzahl von FFTs zu wählen, um Messungenauigkeiten auszuschließen. Das Ergebnis, dargestellt in Abbildung 5.9, dient zur Veranschaulichung der Tendenzen, wobei die tatsächliche Anzahl der FFTs nicht entscheidend ist.

Testvorrausetzungen:

Anzahl der FFTs (Eine pro Threadblock)	N (Anzahl der Punkte pro FFT)
5400	2 bis 1024

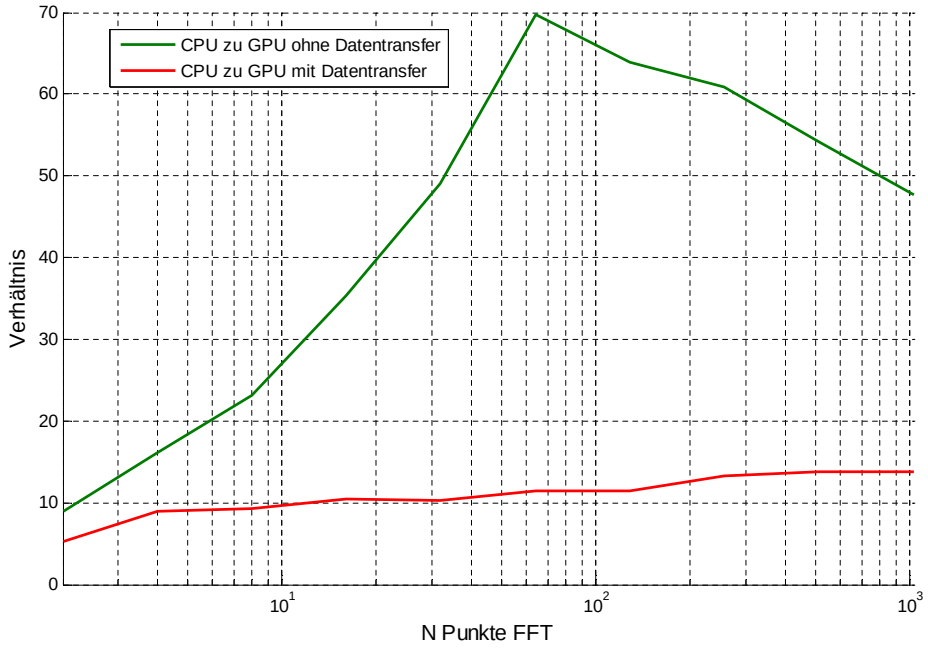


Abbildung 5.9: N Punkte: Verhältnis CPU Zeit zu GPU Zeit

Tendenziell besteht ein erheblich großes Verhältnis zwischen der Ausführungszeit auf der CPU zu der GPU, siehe Abbildung 5.9. Die reine Kernelausführungszeit ohne Datentransfer (grün) veranschaulicht zusätzlich einen rapiden Anstieg bei zunehmender Anzahl von Punkten für die Berechnung der FFT, die auf eine zunehmende Effizienz in der Berechnung schließen lässt. Hierbei ist ein Maximum bei $N = 64$ zu beobachten. Das Verhalten für $2 \leq N \leq 64$ ist über die Zusammenhänge für den effizienten Einsatz der CUDA-Hardware in der N Punkte FFT Berechnung zu erklären. Es ist ein Abfall für $N > 64$ zu beobachten. Dieser Effekt ist über die Verarbeitungscharakteristik der CPU zu erklären. Dabei kann eine unterschiedliche Anzahl von FFTs in diesem Bereich zu anderen Ergebnissen führen. Aus diesem Grund sind zusätzlich die Kernel-Ausführungszeiten in Tabelle 5.2 aufgeführt.

Eine Betrachtung der Multiprozessorauslastung veranschaulicht die Effizienz des implementierten Algorithmus für unterschiedliche N Punkte FFTs.

Die Auslastung der CUDA-Multiprozessoren (SMs) für den implementierten FFT-Algorithmus prägt sich in folgender Form aus:

N	2	4	8	16	32	64	128	256	512	1024
Auslastung in %	**25**	**25**	**25**	**25**	**25**	**25**	**50**	**75**	**75**	**50**
Dynamic Shared Memory	16	32	64	128	256	512	1024	2048	4096	8192
Max. Aktive Blöcke/SM	**8**	**8**	**8**	**8**	**8**	**8**	**8**	**6**	**3**	**1**
Anzahl Threads	1	2	4	8	16	32	64	128	256	512
Threads pro Warp	**1**	**2**	**4**	**8**	**16**	**32**	32	32	32	32
Warps pro Block	1	1	1	1	1	1	2	4	8	16
Aktive Warps pro SM	**8**	**8**	**8**	**8**	**8**	**8**	**16**	**24**	**24**	**16**
Max. Aktive Warps/SM	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**	**32**
GPU-Zeit in ms (5400 FFTs)	0,063	0,086	0,102	0,127	0,162	0,24	0,501	1,164	2,753	6,565

Tabelle 5.2: Auslastung der Multiprozessoren

Die Ergebnisse veranschaulichen die Auslastung bei unterschiedlichen N-Punkte FFTs. Die Anzahl der zu berechnenden FFTs ist in diesem Fall nicht ausschlaggebend, da eine FFT pro Multiprozessor (Threadblock) berechnet wird. Prinzipiell ist eine Auslastung von 50% bis 75% eine übliche Auslastung, im Einsatz von CUDA. Diese sollte jedoch nur akzeptiert werden, wenn die Begrenzung durch den ON-Chip Memory auftritt. Dieses trifft in diesem Fall für die 256, 512 und 1024 Punkte FFT zu. Eine größere Problematik ergibt sich jedoch in der Situation, wenn Warps (SIMT-Instruktionen) nicht voll ausgenutzt werden. Dieser Fall tritt bei der 2, 4, 8, 16 und 32 Punkte FFT ein. In dieser Zusammensetzung ergibt sich nach Definition ($\frac{Anzahl\,Aktive\,Warps}{Max\,Anzahl\,Aktive\,Warps}$ auf einem Multiprozessor) zwar eine Auslastung von 25 %, jedoch unter einem beschränkten Einsatz der SIMT-Technologie (Warp mit weniger als 32 Threads) und der daraus folgenden Minimierung des Instruktionsdurchsatzes.

Diese Ergebnisse haben zur Folge, dass für niedrige N eine Berechnung von mehreren FFTs in einem Threadblock zusammenzufassen wären, um eine effizientere Nutzung der CUDA-Multiprozessoren zu realisieren. Die Entwicklung dieses Algorithmus ist jedoch an dem Einsatz für 256, 512 und 1024 Punkte FFTs orientiert und ist deshalb in vorliegender flexibler Form realisiert.

Untersuchung der Ausführungszeiten unter Einsatz von Verarbeitungsströmen

Die folgende Untersuchung veranschaulicht die Vorteile von Verarbeitungsströmen und dessen Fähigkeit zur Überlagerung von Datentransfers und Kernelausführungen. Die Untersuchung dieser Methode findet unter dem Aspekt der Optimierung bzw. Beschleunigung der Datentransaktionen zwischen Host und Device statt.

Die Implementierung ermöglicht die Unterteilung in unabhängige Datensätze ($\frac{Anzahl\,Werte}{N} = Anzahl\,unabhängige\,Datensätze$), die parallel über diesen Algorithmus verarbeitet werden müssen. Für diesen Test werden mehrere 1024-Punkte FFTs auf mehrere Streams verteilt, die somit in paralleler (ein Kernel auf einem Stream) und sequentieller Form (mehrere Kernel auf jeweils einem Stream) verarbeitet werden. Die Umsetzung der Berechnungsseparierung in mehreren Streams ist dem Quellcode in Anhang D.5 zu entnehmen. Die eingebundene Verarbeitungsstruktur ist in Abbildung 5.10 veranschaulicht.

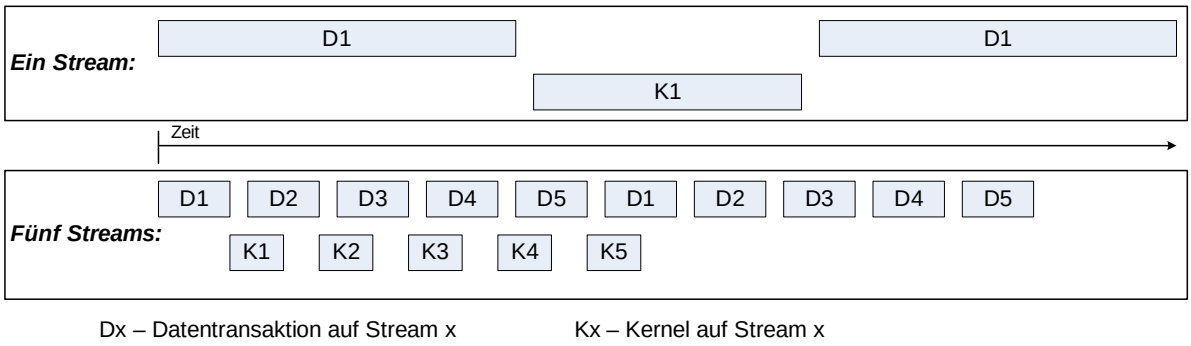


Abbildung 5.10: Verarbeitungsstruktur

In dieser Messung findet eine Bestimmung der Ausführungszeiten mit unterschiedlicher Anzahl von FFTs, inklusive der Datentransfers, statt. Es findet ein Vergleich des Einsatzes eines einzigen

Streams für die gesamte Verarbeitung mit der Separierung auf 5 Streams statt. Die Abbildung 5.11 veranschaulicht die Messergebnisse.

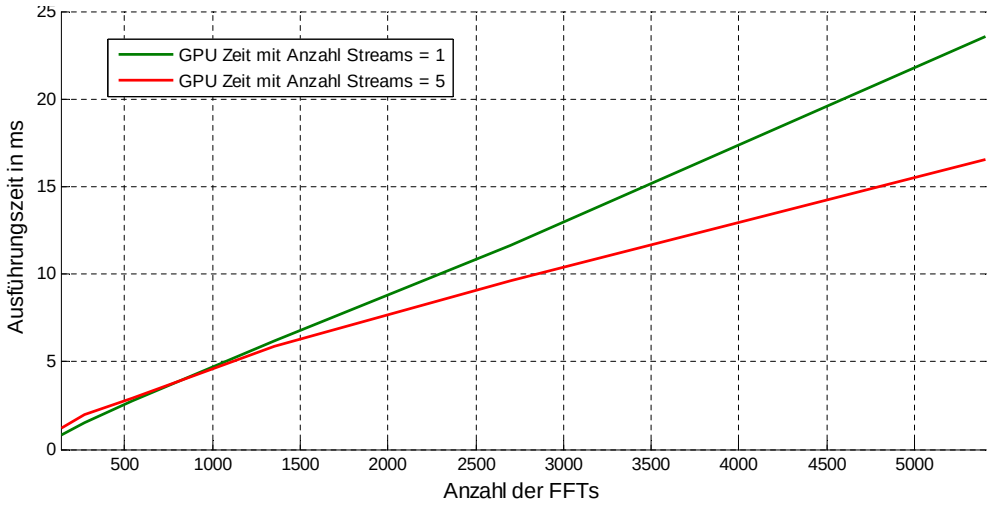


Abbildung 5.11: N FFTs mit Streams: Ausführungszeit in ms

Die Vorteile dieser Ausführungsstruktur ergeben sich im Fall einer hohen Anzahl von unabhängigen Datensätzen, die für die Verarbeitung mit dem FFT-Algorithmus vorgesehen sind. Ein hoher Bedarf an Übertragungszeit im Verhältnis zu der geringeren Ausführungszeit des Kernels stellt ein Indiz für die Optimierung über die Separierung in Streams dar. Als problematisch erweist sich die Methode bei geringen Ausführungszeiten von Datentransferfunktionen und Kernelausführung. Die Mehrfachausführung von aufeinanderfolgenden Datenübertragungsfunktionen verursachen im Initialisierungszeitpunkt für die gesamte Ausführungszeit entscheidende Verzögerungen. In diesen Fällen ist von einer Separierung abzuraten.

Für jeden Anwendungsfall ist eine Untersuchung der sinnvollen Separierung in mehrere Ausführungsstränge durchzuführen.

Des Weiteren ist diese Methode eine effiziente Variante für sequentiell zu verarbeitende Datensätze. Diese Situation tritt oft in einer stromförmigen Datenübertragung auf, wie sie in der Abtastung von Signalen mit der zugehörigen Pufferung von Daten für die Berechnung oder in der Videostreamverarbeitung zu finden ist[4].

5.1.4 Algorithmus Fazit

Die Ergebnisse zeigen weitere Optierungsmöglichkeiten für die Effizienzsteigerung des Algorithmus auf. Die in diesem Abschnitt präsentierte FFT Implementierung dient zur Veranschaulichung der Implementierungs- und Optimierungsschritte in CUDA und ist hiermit für diese Arbeit abgeschlossen.

In diesem Zusammenhang ist jedoch eine Darstellung der Verbesserungsmöglichkeiten angebracht. Die Problematik der versetzten Adressierung im Shared Memory Zugriff ist im Radix-2 Algorithmus möglicherweise über eine komplexe Umsortierung nach der Beendigung jedes Schrittes zu beheben.

[4]Es müssen z.B. erst 1024 Werte abgetastet werden bis sie verarbeitet werden können

Jedoch wäre dieses immer mit „Querzugriffen" in der Lese- oder Schreiboperation verbunden und würde wahrscheinlich nicht zu der gewünschten Optimierung führen.

Die Wahl eines Algorithmus, der in seiner Umsetzungsform weniger Speicherzugriffe und weniger Schritte benötigt, wäre in diesem Fall die effizientere Wahl, wobei an dieser Stelle als Beispiel ein Radix-4 oder Radix-8 Algorithmus zu nennen wäre.

Die im CUDA-Toolkit enthaltene FFT Bibliothek „CuFFT" behandelt diese Problemstellung in speziellen Implementierungsformen für jede bestimmte Anzahl von Eingangswerten, unter dem Einsatz verschiedener Algorithmen, um für jeden Fall die effizienteste Methode zur Verfügung zu stellen.

Diese Vorgehensweise würde den Umfang dieser Arbeit überschreiten. Daher ist für die Veranschaulichung des CUDA-Implementierungsverfahrens die Wahl auf den in diesem Abschnitt präsentierten Algorithmus gefallen.

Der Implementierungsprozesses unter CUDA ist in Abschnitt 5.1 mit den veranschaulichten Implementierungs-, Optimierungs- und Analyseschritten des FFT Algorithmus präsentiert, mit den dazugehörigen Optimierungs- und Analysemöglichkeiten für eine effiziente Programmcodeentwicklung.

5.2 Implementation eines Faltungsalgorithmus in der Bildverarbeitung

Die Implementierung eines Faltungsalgorithmus in der Bildverarbeitung in CUDA findet unter Einsatz des in Abschnitt 4.2 präsentierten Vorlageprojektes für die Bildverarbeitung statt. Die Bilddaten liegen in 256 Stufen (ein Byte) pro Farbkanal pro Bildpunkt vor.

5.2.1 Grundlagen der Faltungsoperation

Die Faltungsoperation in der Bildverarbeitung gilt als eine lineare, verschiebungsinvariante Operation. Dies bedeutet, dass die Durchführung der Operation unabhängig von der Position im Bild geschieht. Diese Methode ist in dieser Arbeit für bildverbessernde Vorgänge im Einsatz, um eine digitale Bildfilterung zu realisieren.

Die Faltung ist prinzipiell über die Operation $H = f * g$ umgesetzt. Hierbei repräsentiert f die zu verarbeitenden Bilddaten und g die Faltungsfunktion, in Form einer Matrix.

Allgemeine Form der Operation:

$$h(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} g(i,j) \cdot f(x+i, y+j) \tag{5.6}$$

In der praktischen Anwendung von bildverbessernden Maßnahmen in der digitalen Bildverarbeitung ist die Faltungsfunktion auf einen bestimmten Bereich beschränkt und drückt sich in der speziellen Form über folgende Formel aus:

$$h(x,y) = \sum_{i=-k}^{k} \sum_{j=-l}^{l} g(i,j) \cdot f(x+i, y+j) \tag{5.7}$$

Die einzelnen Elemente der Faltungsmatrix stellen Gewichtungsfaktoren für den Einbezug der in diesem Bereich vorhandenen Bildpunkte für die Bestimmung des neuen Bildpunktwertes an der aktuellen Position dar. Diese Operation ist für jeden Bildpunkt im Bild durch zuführen und dieses jeweils mit den unveränderten Bilddaten. Das grundlegende Berechnungsverfahren der Faltungsoperation ist in Abbildung 5.12 veranschaulicht.

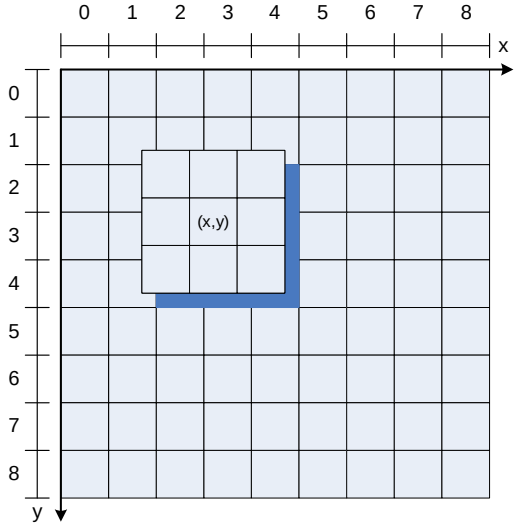


Abbildung 5.12: Faltungsoperation

Die Umsetzung von Bildfilterung über diese Operation ist in den Randgebieten der Bilddaten problematisch. Daher existieren verschiedene Verfahren für die Beschaffung von Eingangsdaten in Regionen des Bildes für die keine Daten existieren ($x + i < 0$ oder $y + j < 0$ oder $x + i \geqq Breite$ oder $y + j \geqq Höhe$). Die Aussparung der Randregionen stellt dabei die einfachste Methode dar. Die üblichen Methoden für die Faltungsoperation sind das Zero-Padding(Auffüllen mit Nullen) und die zyklische Verarbeitung der Bilddaten (Wiederholung der Daten). Die schematische Darstellung dieser Methoden ist in Abbildung 5.13 dargestellt.

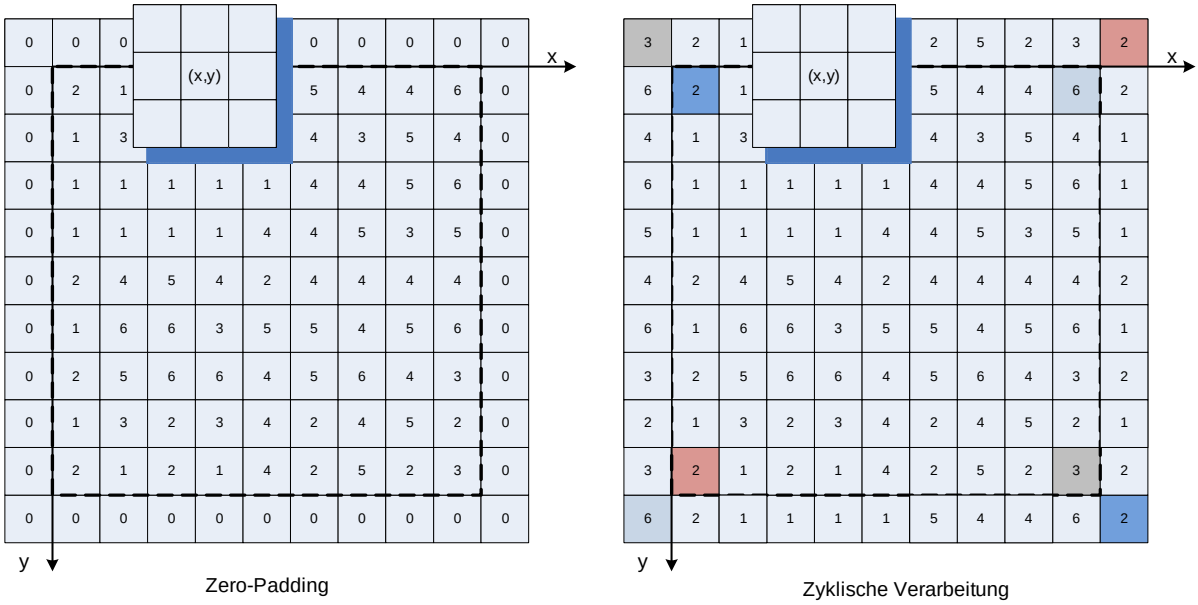


Abbildung 5.13: Zero-Padding und zyklische Verarbeitung

5.2.2 Implementierungsstrategie der Parallelverarbeitung in CUDA

Das im vorherigen Abschnitt erläuterte Verfahren beschreibt die für diesen Abschnitt eingesetzte Verarbeitungsform von Bilddaten mit einem Farbkanal mit 256 Stufen (Grauwertbilddaten).

Die grundlegende Struktur drückt sich in der Verarbeitung jedes einzelnen Bildpunktes über einen einzelnen Thread aus, wobei die Threadstruktur in einer zweidimensionalen Form organisiert ist.

Grundstruktur

- ▷ Ein Thread pro Bildpunkt[5]
- ▷ Threadblock: $16 \times 16 = 256$ Threads
- ▷ Dynamische Anpassung des Grids auf die Größe des Bildes
- ▷ Faltungsmatrix in Constant Memory abgespeichert
- ▷ Optionale Speicherorganisation für die Analyse[6]

Die Implementierung ist für verschiedene Größen von Faltungsmatrizen (3×3, 5×5 , 7×7 usw.) vorgesehen.

5.2.3 Implementierung und Optimierung des Algorithmus

Die im Folgenden erläuterte Implementierung ist zunächst für die Verarbeitung von Grauwertbilddaten konzipiert. Die aufgeführten Kernelausführungszeiten sind in diesem Abschnitt dem Visual

[5] Mögliches Verfahren mit mehreren Farbkanälen: Ein Thread pro Farbkanal pro Bildpunkt
[6] Speicherorganisation im Global oder Texture Memory, sowie der möglichen Datenpufferung im Shared Memory

Profiler entnommen. Der Quellcode in Anhang E.1 enthält die zusammengefasste Implementierungsform für die Speicherorganisation der Eingangsdaten im Global Memory oder Texture Memory, sowie die mögliche Datenpufferung im Shared Memory. Die Implementierung erfolgt an dem Beispiel der Mittelwertfilterung (Tiefpassfilterung) in einer 3×3 Matrix für die Faltungsoperation (siehe Gleichung 5.7).

$$Matrix: \begin{matrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{matrix} = \frac{1}{9} \cdot \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} \tag{5.8}$$

In der Implementierung ist stets auf die Optimierungsrichtlinien (siehe Abschnitt 3.4) zu achten. Auf Grund der arithmetischen Optimierung in der Umsetzung sind, wenn möglich, Divisionsoperationen zu vermeiden und Floatingpoint-Operationen den Integer-Operationen vorzuziehen. Die Ganzzahlmultiplikationen sind in 24 Bit Genauigkeit umzusetzen (`__[u]mul24()`). Der entwickelte Quellcode für den Kernelaufruf ist in Anhang E.2 dokumentiert und die Ergebnisse des Profilers anhand eines 640×472 Grauwertbildes in Anhang E.3 präsentiert.

Die Untersuchung der Implementierungsform des Faltungsalgorithmus, unter Einsatz des globalen Speichers für die Eingangsdaten, ergibt einen hoch parallelisierten Ablauf mit einer Multiprozessorauslastung von 100%. Diese Tatsache ist über den niedrigen On-Chip Speicherbedarf(Register: 10, Shared Memory: 40) der 16×16 Threadblöcke (256 Threads) begründet, die die Bilddaten für jeden Bildpunkt parallel verarbeiten.

Die Ausführung des Kernels beinhaltet eine geringe Anzahl von divergenten Ausführungssträngen[7], die wahrscheinlich auf Grund der speziellen Behandlung der Randgebiete des Bildes und den überflüssigen Threads aus der dynamischen Grid-Dimensionierung entstehen. In der Grid-Dimensionierung findet eine Aufrundung[8] (feste Threadblockgröße 16×16) statt. Die nicht benötigten Threads sind über eine Kontrollstruktur (if-Abfrage) abzufangen, um fehlerhafte Speicherzugriffe und Operationen zu vermeiden. Eine korrekte dynamische Dimensionierung des Grids ist andernfalls nicht möglich.

Die Implementierung beinhaltet zwei Varianten für die Behandlung des Randbereichs. Dazu zählt die Aussparung des Randbereiches und Zero-Padding in der Faltungsoperation. Die resultierenden Kernelausführungszeiten unterscheiden sich erheblich. Dies ist mit der erhöhten Komplexität durch die Kontrollstrukturen im Zero-Padding zu begründen. Dieses Kriterium ist in der Wahl des geeigneten Speicherbereiches zu berücksichtigen.

Kernelausführungszeiten:

	Randaussparung	Zero-Padding
Zeit in µs	106,7	194,7

[7] Möglicherweise Messfehler des Visual Profilers und es sind keine divergenten Ausführungsstränge vorhanden („Branch Predication" - siehe Abschnitt 3.4.2 auf Seite 31)

[8] Siehe Beispielcode 3.3

Texture Memory

Die optimalen Einsatzgebiete für den Texture Speicherbereich ergeben sich in der Bildverarbeitung. In diesem Sinne ist eine Prüfung für den effektiven Einsatz von Texture Memory für die Eingangsdaten des Algorithmus vorzunehmen. Der modifizierte Kernelaufruf ist in Anhang E.4 und die Ergebnisse des Profilers anhand eines 640×472 Grauwertbildes in Anhang E.5 aufgeführt.

Der Einsatz von Texture Memory ist in diesem Zusammenhang zwar sinnvoll, jedoch hat die erläuterte Modifikation in der bisherigen Implementierungsform keinen effektiven Geschwindigkeitsvorteil geliefert und führt durch die erhöhten Vorbereitungsmaßnahmen im Kernelaufruf zusätzlich zu Verzögerungen im Ablauf. Die Ergebnisse des Visual Profilers zeigen eine wesentliche Erhöhung der Anzahl der Instruktionen an, die Einfluss auf die Ausführungszeit des Kernels nehmen. Eine Untersuchung für den effektiven Einsatz von Texture Memory ist über Fallunterscheidungen in Abschnitt 5.2.4 dokumentiert.

Shared Memory

Der Einsatz des Shared Memory Speicherbereichs für die Pufferung der Eingangsdaten ist für diesen Algorithmus genau zu analysieren. Die Bildpunkte im Bereich der Faltungsmatrix müssen aus dem Speicher gelesen werden, um die Faltungsoperation durchzuführen.

Die Verwendung des Shared Memory führt zu einer Minimierung der globalen Speicherzugriffe. Hierbei ist jedoch zu beachten, dass die Daten vorweg in den Shared Memory kopiert werden müssen. Die Threadstruktur lässt keine weitere Optimierung der Zugriffe auf den globalen Speicher zu. Die eingesetzte CUDA-Hardware (Compute Capability 1.3) realisiert die globalen Speicherzugriffe über zusammengefasste 32 Byte Transaktionen (siehe Anhang E.3). Daher ist zu untersuchen, ob ein effektiver Einsatz von Shared Memory in diesem Algorithmus möglich ist.

Die Problematik im Einsatz von Shared Memory ergibt sich über die Bildregionen, die für die Faltung in den Shared Memory kopiert werden müssen. Die Verarbeitung ist auf 16×16 große Threadblöcke definiert, die die Faltungsoperation für die Bildpunkte in dem zugeteilten Bereich durchführen. Aus der Faltungsoperation ergeben sich jedoch Speicherzugriffe (Zugriffe auf Bildpunkte) außerhalb dieser Regionen (Faltung im Randgebiet). Die gesamte Menge der benötigten Datenelemente sind im Übertragungsvorgang vom Global Memory in den Shared Memory zu berücksichtigen. Diese Vorgehensweise ist in Abbildung 5.14 veranschaulicht.

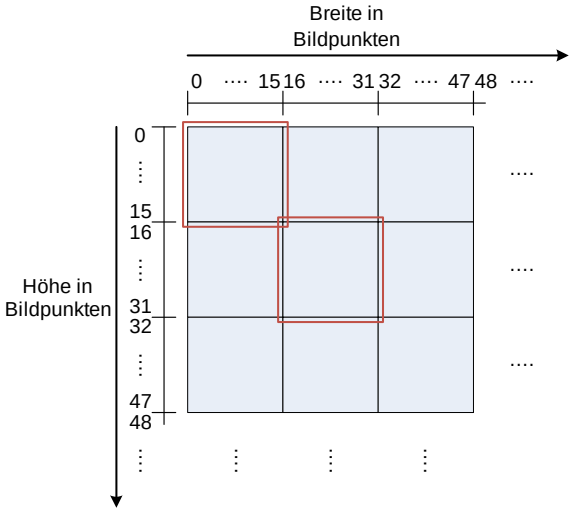


Abbildung 5.14: Faltung mit Shared Memory

Die im Einsatz von Shared Memory zu kopierenden Regionen sind rot gekennzeichnet. Der Randabstand (d) zu dem zu verarbeitenden Bereich (hier: 16×16) entspricht der halben abgerundeten Höhe/Breite der Faltungsmatrix($3 \times 3 \triangleq d = 1$). Die benötigte Datenmenge im Shared Memory ergibt sich aus diesem Zusammenhang (Anzahl Datenelemente: $(16 + 2 \cdot d) \times (16 + 2 \cdot d)$). Der Kopiervorgang ist in vier Schritten organisiert, veranschaulicht in Abbildung 5.15. Dieses Verfahren ist bedingt durch die eingesetzte Threadstruktur. Bei Kopierzugriffen auf nicht existierende Bilddaten findet über eine Kontrollstruktur ein automatisches Zero-Padding statt.

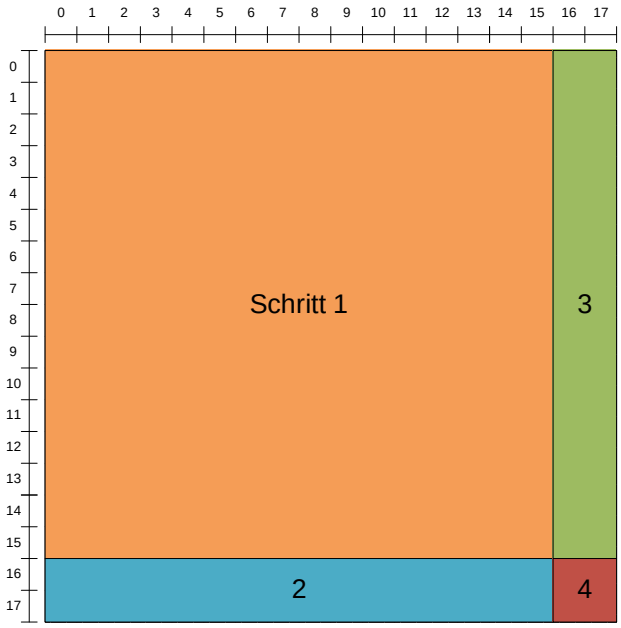


Abbildung 5.15: Shared Memory Kopiervorgang mit $d = 1$ und 16×16 Threadblock

Die Ergebnisse des Visual Profilers anhand eines 640×472 Grauwertbildes sind in Anhang E.6 dokumentiert.

Die Kernelausführungszeit unter Einsatz von Shared Memory für die Datenpufferung liegt bei

162,4 µs. Dieses Ergebnis verdeutlicht eine Erhöhung der Ausführungszeit im Einsatz von Shared Memory im Vergleich zum Einsatz von Global Memory (ohne Shared Memory) mit Randaussparung. Die Ergebnisse des Profilers veranschaulichen eine Minimierung der globalen Speicherzugriffe (von 24795 auf 6946). Auf Grund der komplexen Struktur des Datentransferprozesses ist jedoch ein erheblicher Anstieg von Instruktionen und divergenten Ausführungssträngen zu beobachten. Der Zugriff auf den Shared Memory in der Faltungsoperation stellt hierbei keine Problematik dar, denn es sind keine serialisierten Speicherzugriffe auf den Shared Memory festzustellen. Aus der Messung mit dem Visual Profiler resultiert eine 100%-Auslastung der CUDA-Multiprozessoren.

Die tatsächliche Eignung der Datenpufferung im Shared Memory ist in einer Fallunterscheidung mit unterschiedlichen Datenquellen (Global und Texture Memory) in Abschnitt 5.2.4 dokumentiert. Schlussfolgerung: Besteht ein definitiver Bedarf von Zero-Padding in der Behandlung der Randgebiete ist der Shared Memory einzusetzten.

Finale Implementierungsform

Die Implementierungsform des Kernels für Grauwertbilddaten ist für den Faltungsalgorithmus speziell anzupassen, um die Übertragung der Implementierung auf Bilddaten mit mehreren Farbkanälen zu realisieren. Die Realisierung erfolgt über eine separate Parallelverarbeitung jedes Farbkanals (RGB-Daten), mit einem Thread pro Farbkanal pro Bildpunkt. Dies ist über eine versetzte Adressierung(stride = Anzahl der Kanäle) und einer dementsprechenden Erhöhung der Threadanzahl ($Threads \cdot (Anzahl\,Farbkanäle)$) zu erreichen.

Die veränderte Implementierungsform, unter dem Einsatz von globalen Speicher für die Eingangsdaten, ist in Anhang E.7 aufgeführt. Die Kernelausführungszeit beträgt bei 640×472 RGB-Daten (drei Farbkanäle) **290,7 µs**. Dieses Ergebnis entspricht den Erwartungen (ca. dreifache Ausführungszeit gegenüber den Grauwertdaten). Die Profiler Ergebnisse sind in Anhang E.8 aufgeführt.

Der Shared Memory ist in der soeben erläuterten Implementierungsform **nicht effektiv** einsetzbar. Aus diesem Grund steht alternativ eine spezielle Implementierung für RGB-Daten (24-Bit Daten) unter Einsatz des Global oder Texture Memory für die Eingangsdaten, sowie der möglichen Datenpufferung im Shared Memory zur Verfügung. Diese spezielle Implementierung basiert auf der vorweg präsentierten Umsetzungsform für Grauwertbilddaten. Für diese spezielle Implementierung wird **ein Thread pro Bildpunkt** eingesetzt. **Alle Farbkanäle** eines Bildpunktes werden durch **denselben Thread** verarbeitet.

5.2.4 Analyse und Präsentation der Ergebnisse

Die Analyse der erstellten Implementierungen der Faltungsoperation unter CUDA findet anhand der erläuterten Implementierungssformen statt. Dabei ist die Umsetzung für die Grauwertdatenverarbeitung und die spezielle Implementierung für die RGB-Datenverarbeitung (drei Farbkanäle) im Einsatz. Ein Zero-Padding wird in allen Implementierungen verwendet. Die Bestimmung der Ausführungszeiten findet über die Timer-Subroutinen, aufgeführt in Anhang C.1, statt. Die folgende Untersuchung beinhaltet eine Tiefpassfilterung mit unterschiedlichen Größen der Faltungsmatrix und unterschiedlichen Bildgrößen.

Untersuchungsbedingungen

Die im Folgenden aufgeführten Daten werden für die Messung der Ausführungszeiten jeweils miteinander kombiniert.

Matrixgrößen	3×3	5×5	7×7	9×9

Bildgrößen	512×512	1024×1024	2048×2048

Bestimmung der idealen Speicherorganisation

Die folgende Untersuchung veranschaulicht eine Fallunterscheidung für die unterschiedlichen Speicherorganisationsformen im Kernel.

Die Abbildung 5.16 zeigt die Differenzen der Kernelausführungszeiten für **Grauwertdaten**. Die Differenz der Ausführungszeiten für den Einsatz von Global Memory für die Eingangsdaten bildet sich aus der Ausführungszeit ohne Datenpufferung im Shared Memory und mit Datenpufferung im Shared Memory. Das Gleiche gilt für den Einsatz von Texture Memory für die Eingangsdaten.

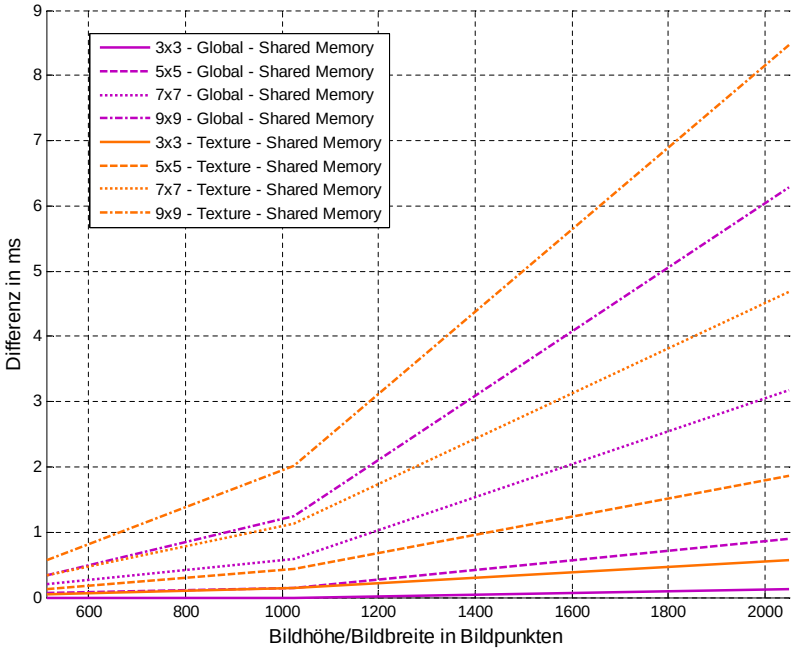


Abbildung 5.16: Differenzkurve der Kernelausführungszeiten mit Grauwertdaten

In dieser Abbildung ist zu beobachten, dass sich in der Ausführungszeit bei einer 3×3 Faltungsmatrix kein signifikanter Unterschied mit und ohne Pufferung im Shared Memory ergibt. Der Shared Memory ist in diesem Fall tendenziell eher langsamer. Allerdings mit Vergrößerung der Faltungsmatrix verdeutlichen sich die Unterschiede. Diese Ergebnisse sprechen für den prinzipiellen Einsatz von Shared Memory. Die Auswertung veranschaulicht zusätzlich, dass im Falle der Grauwertdatenverarbeitung in der eingesetzten Implementierungsform sich der Einsatz von Texture Memory nicht als effektiv erweist, im Einsatz von Shared Memory ist diese Tatsache jedoch unerheblich. Die

in Anhang E.9 aufgeführten Ausführungszeiten veranschaulichen keinen signifikanten Unterschied für die Eingangsdaten im Global oder Texture Memory, wenn dieses Daten für die Berechnung im Shared Memory gepuffert werden.

Die Abbildung 5.17 zeigt die Differenzen der Kernelausführungszeiten für **RGB-Daten**. Die Differenz der Ausführungszeiten für den Einsatz von Global Memory für die Eingangsdaten bildet sich aus der Ausführungszeit ohne Datenpufferung im Shared Memory und mit Datenpufferung im Shared Memory. Das Gleiche gilt für den Einsatz von Texture Memory für die Eingangsdaten.

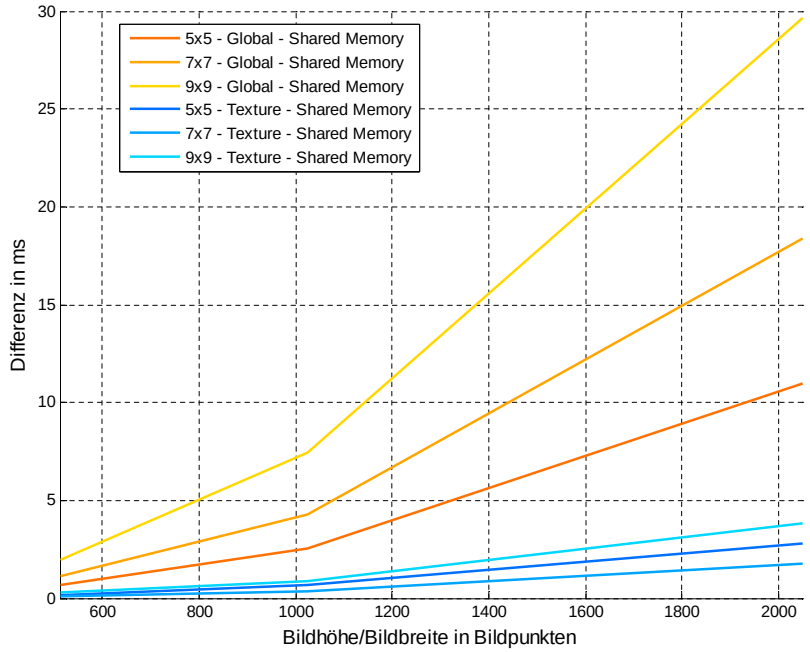


Abbildung 5.17: Kernelausführungszeiten RGB-Daten: Global Memory und Shared Memory Pufferung

Die in der Abbildung dargestellten Ergebnisse bestätigen den prinzipiellen Einsatz von Shared Memory. Aus der Anordnung der Differenzengraphen verdeutlicht sich eine besondere Effizienz des Shared Memory für die 5×5 Faltungsmatrix. Bei diesen Ergebnissen ist jedoch zu beobachten, dass für die spezielle Implementierung mit RGB-Daten der Texture Memory wesentlich effektiver einzusetzen ist als der Global Memory.

Die gesamten Ergebnisse der Messungen der Kernelausführungszeiten sind in Anhang E.10 für RGB-Daten aufgeführt.

Aus der Analyse ergibt sich, dass für den Faltungsalgorithmus sich der **Einsatz von Shared Memory als effektivste Implementierung** erweist.

Vergleich mit Implementierung für den Host (CPU)

Für den Vergleich der Ausführungszeiten zwischen CPU und GPU erfolgt eine Verhältnisbetrachtung der CPU Ausführungszeit zu der GPU Ausführungszeit. Die Ergebnisse dieses Vergleichs sind

in Tabelle 5.3 mit allen Kombinationen, sowie mit und ohne Einbezug der Datentransferzeiten zwischen Host und Device, dargestellt. Die CPU-Implementierung liegt in einer konventionellen, sequentiellen Form für die Analyse vor, die im Vergleich zur GPU **Implementierung mit Shared Memory** steht.

	Grauwerte / Ohne Datentransfer			RGB / Ohne Datentransfer		
Faltungsmatrix	512 × 512	1024 × 1024	2048 × 2048	512 × 512	1024 × 1024	2048 × 2048
3 × 3	89,53	104,90	108,71	**82,88**	92,20	91,47
5 × 5	91,98	106,13	100,81	102,24	116,55	118,97
7 × 7	126,96	130,18	129,14	87,40	90,11	91,07
9 × 9	126,60	131,21	**133,00**	89,41	91,53	91,95
	Grauwerte / Mit Datentransfer			RGB / Mit Datentransfer		
Faltungsmatrix	512 × 512	1024 × 1024	2048 × 2048	512 × 512	1024 × 1024	2048 × 2048
3 × 3	36,32	50,70	59,79	36,11	40,87	40,88
5 × 5	68,37	76,82	73,91	55,87	67,27	70,70
7 × 7	74,24	98,96	104,93	57,28	68,91	70,93
9 × 9	93,06	109,32	114,48	69,90	76,04	77,69

Tabelle 5.3: Verhältnis der Ausführungszeiten: CPU zu GPU

Das Verhältnis der CPU- zur Kernelausführungszeit liegt nicht unter dem Wert 80, dass eine minimale ca. **80-fache Beschleunigung** durch den Grafikprozessor bedeutet. Die Betrachtung des Einflusses der Datentransferzeiten zeigt eine dementsprechende Minimierung des Verhältnisses.

Die Transferzeiten fallen durch die erhöhten Rechenzeiten bei größeren Faltungsmatrizen weniger ins Gewicht. Dieser Effekt ist durch die gleichbleibenden Transferzeiten zwischen Host und Device (Hin- und Rückweg) für das jeweilige Bild zu erklären. Die durchschnittlichen Transferzeiten sind für die einzelnen Bildgrößen in Abbildung 5.18 dargestellt. Die Darstellung beinhaltet die zusätzliche Messung für die Bildgrößen 128 × 128 und 256 × 256.

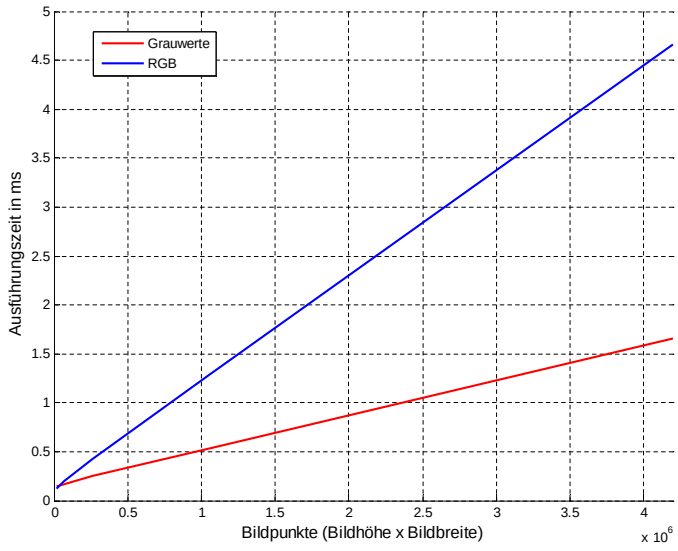


Abbildung 5.18: Durchschnittliche Transferzeiten zwischen Host und Device

Die Abbildung veranschaulicht eine Verzögerungszeit (Totzeit) in der Datenübertragung von ca. $0,15\,ms$, die über Initialisierungs- und Speicherzugriffsverzögerungen zu erklären sind. Dieser Effekt ist bereits bei der Betrachtung einer Separierung von unabhängigen Berechnungsaufgaben auf mehrere Streams in Abschnitt 5.1.3 zu beobachten gewesen, die bei geringen Datenmengen durch die wiederholten Kopiervorgänge zu Verzögerungen geführt haben.

Der Zeitunterschied zwischen Grauwert- und RGB-Daten begründet sich in der dreifachen Datenmenge der RGB-Daten (Verhältnis in Abbildung ca. 3:1). Die effektive Datenübertragungsrate von ca. **2,8 GB/s** ist über die Geradengleichung (lineare Interpolation) der Graphen zu bestimmen.

Allgemein ist der Einfluss der Datenübertragung für jeden Anwendungsfall genau zu untersuchen. Hierbei ist die Echtzeitverarbeitung von Videostreams zu nennen, die in bestimmten Intervallen Bilddaten für die Verarbeitung liefern. In der Regel werden die Ergebnisse der Berechnung direkt dargestellt. Über die „OpenGL Interoperability" in CUDA ist es ermöglicht, die Resultate direkt aus dem Grafikspeicher in einem Fenster auszugeben. Diese Methode vermeidet die Datentransaktion von Host- zum Device. Des Weiteren steht über die bereits vorgestellten Streamverarbeitungsmethoden (Überlagerung von Datenübertragung und Kernelausführung) ein Verfahren zur Verfügung, um die Datentransferzeiten in den Hintergrund geraten zu lassen. Aus diesem Grund ist eine reine Betrachtung der Kernelausführungszeiten von größerer Bedeutung.

Dieser Abschnitt verdeutlicht, das enorme Leistungsvermögen der CUDA-Hardware, die, je nach Fall (Faltungsmatrix und Bildgröße), eine ca. **80-fache bis 130-Fache Leistungssteigerung** zur CPU bewirkt. Dieses Ergebnis bestätigt das hohe Leistungspotenzial für den Einsatz in verschiedensten Berechnungsvorgänge und Algorithmen der Bildverarbeitung.

5.3 Implementation einer 2D-Fourier-Transformation

Die Implementierung der 2D-Fourier-Transformation in der Bildverarbeitung in CUDA findet unter Einsatz des in Abschnitt 4.2 präsentierten Vorlageprojektes für die Bildverarbeitung statt. Die Bilddaten liegen in 256 Stufen (ein Byte) pro Farbkanal pro Bildpunkt vor.

Die Implementierung der 2D-Fourier-Transformation bedient sich der CUDA Bibliothek „CuFFT". Dieses Verfahren dient der Veranschaulichung, dass CUDA für die Beschleunigung von Berechnungsvorgängen eingesetzt werden kann, ohne die kompletten, komplexen Implementierungsmethoden von CUDA zu beherrschen. Die im CUDA Toolkit eingebundenen Bibliotheken „CuFFT" für Fourier-Transformationen und „CuBLAS" für lineare Algebra ermöglichen den vereinfachten Einsatz von CUDA.

5.3.1 Grundlagen der zweidimensionalen Fourier-Transformation

Dieser Algorithmus ermöglicht die Bestimmung der Frequenzen in einem Bild. Diese Frequenzen drücken sich über die Farbwechsel im Bild aus. Die übliche Darstellungsform erfolgt in einem zweidimensionalen Koordinatensystem.

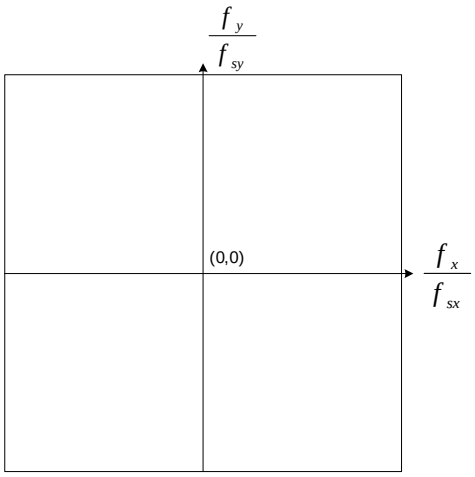


Abbildung 5.19: Darstellung im Frequenzbereich

Quelle Abbildung: [TH05] S.151

Hierbei werden die Frequenzen in x- und y-Richtung mit der zentrieren Null-Frequenz in der Mitte über eine Verschiebung der Berechnungsergebnisse dargestellt, siehe Abbildung 5.19. Bei den dargestellten Frequenzen handelt es sich um normierte Frequenzen mit dem Maximalwert $\frac{f_{max}}{f_s} = 0,5$. Die Transformation findet anhand realer Eingangswerte statt, dass sich über ein punktsymmetrisches Spektrum zur Null-Frequenz ausdrückt.

Die Berechnung einer diskreten Fourier-Transformation in zweidimensionaler Form ist über zwei bekannte Methoden zu realisieren. In der Verarbeitung der Transformationsergebnisse ist zu beachten, dass diese nicht wie in der Abbildung 5.19 dargestellten Form vorliegen. Die Null-Frequenz ist in den jeweiligen Ecken des Ergebnisbildes zu finden. Die präsentierte Darstellungsform dient zur besseren Veranschaulichung der Frequenzen.

1. Diskrete Fourier-Transformation in zweidimensionaler Form

$$DFT:\ S(f_x, f_y) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left(b(x,y) \cdot e^{-j2\pi \left(\frac{x \cdot f_x}{M} + \frac{y \cdot f_y}{N} \right)} \right) \tag{5.9}$$

$$Inverse\,DFT:\ b(x,y) = \frac{1}{M \cdot N} \sum_{f_x=0}^{M-1} \sum_{f_y=0}^{N-1} \left(S(f_x, f_y) \cdot e^{j2\pi \left(\frac{x \cdot f_x}{M} + \frac{y \cdot f_y}{N} \right)} \right) \tag{5.10}$$

2. Separate Berechnung über einzelne 1D-Transformationen für Zeilen und Spalten

Die 2D-Fourier-Transformation ist über eine Berechnung einer 1D-Transformation für jede Zeile, und darauf folgend, für jede Spalte zu realisieren, dargestellt in Abbildung 5.20.

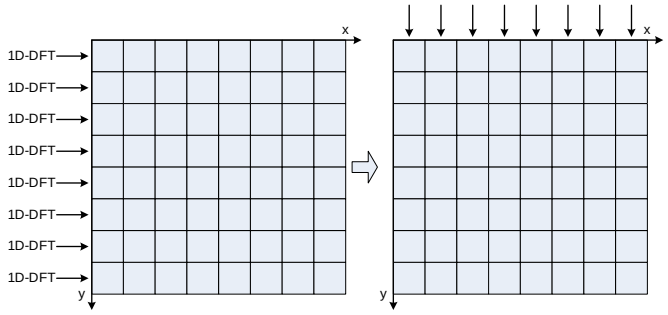


Abbildung 5.20: Separierbare 2D-Fourier-Transformation

Mit diesen Methoden können Bilddaten aus dem Ortsbereich in den Ortsfrequenzbereich transformiert werden. Über diesen Zusammenhang ist es möglich, bildverbessernde Maßnahmen im Ortsfrequenzbereich über vereinfachte Operationen zu realisieren. Die Faltung im Ortsbereich entspricht einer Multiplikation im Frequenzbereich.

$$Faltungsoperation: \; b_{out}(x,y) = F^{-1}\{S(f_x,f_y) \cdot H(f_x,f_y)\} \tag{5.11}$$

5.3.2 Implementierungsverfahren unter CUDA

Der Einsatz der „CuFFT" Bibliothek beinhaltet mehrere für CUDA optimierte Fast-Fourier-Transformationsfunktionen für die jeweilig benötigte Einsatzform (ein-, zwei-, und dreidimensional), dokumentiert in [NVI09g].

Die veranschaulichte Methode umfasst eine 2D-FFT Hin- und Rücktransformation mit einer Multiplikation mit der Filterfunktion im Ortsfrequenzbereich (Schnelle Faltung), um eine digitale Bildfilterung zu realisieren. Die Transformationen sind mit RealToComplex- bzw- ComplexToReal-Transformationen realisiert, die die FFT-Berechnungen mit realen Eingangswerten (bzw. Ausgangswerte) durchführen. Aus der Transformation mit $M \cdot N$ realen Eingangswerten resultieren, auf Grund des symmetrischen Spektrums, $(\frac{M}{2}+1) \cdot N$ komplexe Ausgangswerte. Diese Beschaffenheit ist bei der Implementierung von Filterungsoperationen zu berücksichtigen. Der Programmablauf mit den benötigten Funktionen ist in Abbildung 5.21 dargestellt. Der Quellcode für dieses Implementierungsverfahren ist in Anhang F.1 aufgeführt. Die Einsatzziele der einzelnen Funktionen sind als Kommentare im Quellcode festgehalten.

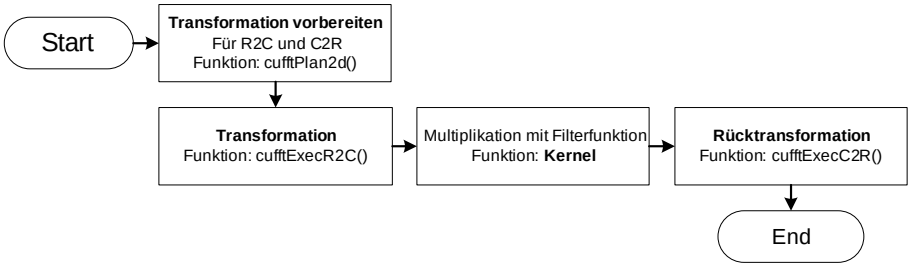


Abbildung 5.21: Ablaufstruktur

Kapitel 6

Zusammenfassung und Ausblick

Ziel war die ***Analyse der NVIDIA CUDA-Architektur für die Implementierung von Signal- und Bildverarbeitungsalgorithmen***. Zudem wurde aufgezeigt, wie effektiv die Möglichkeiten von CUDA tatsächlich sind. Dieses wurde durch intensive Betrachtung und Veranschaulichung an praktischen Beispielen erreicht.

Diese Arbeit dokumentiert die Grundbegriffe der **Compute Unified Device Architecture** und stellt einen Vergleich des CUDA-Hardwaremodells zu bisher existierenden Rechnerarchitekturen auf. Über das verallgemeinerte CUDA-Architekturmodell ist ein flexibler, paralleler Programmablauf für CUDA-Softwareelemente zu erreichen. Dieser besitzt eine hohe Anzahl von parallelen Ausführungssträngen, die in Threadblöcken automatisch auf die CUDA-Multiprozessoren skaliert und abgearbeitet werden. Die nicht festgelegte Anzahl der Multiprozessoren für die allgemeine CUDA-Softwareentwicklung im Einsatz der CUDA-Hardware bewirkt eine besondere Flexibilität. Die Abarbeitung der Threadblöcke erfolgt jeweils durch einen Streaming Multiprocessor(SM), der die Instruktionsverarbeitung in der SIMD-förmigen **SIMT**-Technologie (*Singel Instruktion Multiple Thread*) realisiert. Die verallgemeinerte und standardisierte CUDA-Hardware ermöglicht einen hardwareunabhängigen Einsatz von entwickelten Softwareelementen. Die Hardwarespezifizierung nach den **Compute Capability Specifications** ermöglicht diese Eigenschaft. Die für ältere Standards vorgesehene Software ist auf neuerer Hardware mit neueren Spezifikationen lauffähig (abwärtskompatibel). Somit ist garantiert, dass die entwickelte Software auch auf zukünftiger Hardware lauffähig bleiben wird.

Hier ist die Programmierung der Hardware in der Programmiersprache C/C++ durchgeführt worden. Die gesamte Programmcodeentwicklung erfolgt über den Einsatz der CUDA-APIs und des speziell entwickelten Programmcodes (Kernel - Devicecode). Diese Entwicklung wird durch zusätzliche Bibliotheken („CuFFT" und „CuBlas")[1], den vorgestellten Entwicklungstools (Visual Profiler und Device-Emulation) und der ausführlichen CUDA-Dokumentation mit SDK (Beispielcode) unterstützt. Die CUDA-Programmierumgebung schließt das speziell für CUDA entwickelte Programmiermodell, dessen Organisationsstruktur auf parallelen Ausführungssträngen in Threadblöcken in einem Grid beruht, ein. Die Ausführung durch den Grafikprozessor erfolgt über Kernel-Funktionen, die für jeden Thread jeweils einmal ausgeführt werden. Die Threadorganisation und Identifizierung geschieht über die in diesen Funktionen eingebundenen Indizes. Diese überschaubare Organisati-

[1] CuFFT- Bibliothek für FFTs, CuBlas - Bibliothek für lineare Algebra

onsform für die direkte, parallele Programmierung der CUDA-Hardware ermöglicht den effektiven Einsatz für allgemeine und wissenschaftliche Berechnungsaufgaben. Darüberhinaus wurde ein Entwicklungsleitfaden aufgestellt, in dem die benötigten Elemente für die effiziente Implementierung in CUDA aufgezeigt werden und jederzeit für die Entwicklung zur Hilfe genommen werden können. Die in dieser Arbeit ausgearbeiteten Projektvorlagen sind sehr hilfreich bei der Programmcodeentwicklung.

Am praktischen Beispiel der Implementierung des **FFT Algorithmus** werden die Problemstellungen und Methoden in der Programmcodeentwicklung unter CUDA veranschaulicht. Das dazu entwickelte Softwareelement bewirkt eine bis zu ca.

70-fache Beschleunigung der Ausführungszeit (Kernelausführungszeit)

gegenüber einer konventionellen, sequentiellen Implementierung für den PC. Mit diesem Algorithmus wird das Potenzial der parallelen Rechenleistung der CUDA-Hardware verdeutlicht.

Die Implementierung des **Faltungsalgorithmus für die Bildverarbeitung** unterstreicht diese Ergebnisse. Er bewirkt eine bis zu ca.

130-fache Beschleunigung der Ausführungszeit (Kernelausführungszeit)

im Vergleich zur konventionellen Implementierung.

Die hohe Rechenleistung der Grafikprozessoren wird durch das Programmiermodell und die Programmierumgebung für den allgemeinen und wissenschaftlichen Einsatz so unterstützt, dass es in verschiedensten Anwendungsbereichen effektiv eingesetzt werden kann. Das wird in dieser Arbeit durch die Implementierung von Signal- und Bildverarbeitungalgorithmen deutlich. Die Analyse der CUDA-Architektur zeigt auch, dass das gesamte Konzept von CUDA mit dem parallelen Hardware- und Programmiermodell eine besonders interessante Alternative zu den üblichen Implementierungsverfahren bietet. Es wurde aufgezeigt, dass CUDA-fähige Grafikprozessoren auf der PC Plattform je nach Anwendungsgebiet durchaus konkurrenzfähig zu leistungsstarken Signalprozessoren sind.

Die CUDA-Hardware ist somit in der Signalverarbeitung in vielen Bereich einsetzbar. Sie kann als Unterstützung für aufwendige Simulationen dienen, die einen hohen Rechen- und Zeitbedarf in Anspruch nehmen, sowie für praktische Anwendungen, die eine hohe Rechenleistung benötigen, um in Echtzeit realisiert zu werden. Solche Problemstellungen sind oft in der Signal- und Bildverarbeitung zu finden.

Die Ergebnisse dieser Arbeit sprechen für eine Lösung über das präsentierte CUDA Hard- und Softwarekonzept. Basierend darauf wäre ein nächster Schritt, diese Hardware auch in einer echtzeitkritischen Anwendung einzusetzen.

Literaturverzeichnis

[CUOP] NVIDIA - CUDA Optimization `www.prace-project.eu/hpc-training/training_pdfs/2653.pdf` (Aufrufdatum: 13.01.2010)

[BJ05] Bernd Jähne: Digitale Bildverarbeitung, Springer-Verlag, Heidelberg, 6. Aufl., 2005 , ISBN 3-540-24999-0

[DK08] David Kanter: NVIDIA's GT200: Inside a Parallel Processor , 08.09.2008 `http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1` (Aufrufdatum: 16.12.2009)

[EG05] Carsten Eichholz und Jens Günther: Entwurfsmuster - Übersicht, `http://wwwswt.informatik.uni-rostock.de/deutsch/Infothek/Entwurfsmuster/patterns/index.html` (Aufrufdatum: 24.01.2010)

[GL05] Jackie Neider, Tom Davis, und Mason Woo: OpenGL Programming Guide - The Official Guide to Learning OpenGL, Addision-Wesley Publishing Company, Silicon Grafics Inc., Realease 1, 1993, ISBN 0-201-63274-8

[HR08] Hanno Rabe: Ray-Tracing mit CUDA - Diplomarbeit, Koblenz, September 2008, `http://www.slideshare.net/guest485d9/da-hanno-rabe` (Aufrufdatum: 17.12.2009)

[JB02] Josef Börcsök: Rechnerarchitekturen - Struktur und Konzepte, VDE Verlag GmbH, Berlin, 2002, ISBN 3-8007-2629-7

[KDT05] Klaus D. Tönnies: Grundlagen der Bildverarbeitung, Pearson Studium, München, 2005, ISBN 3-8273-7155-4

[MSBS09] Microsoft Developer Network - Bitmap Structures `http://msdn.microsoft.com/de-de/library/dd183392%28en-us,VS.85%29.aspx` (Aufrufdatum: 24.01.2010)

[MJK96] Mark J. Kilgard: The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3, Silicon Graphics, Inc., 1996 `http://www.opengl.org/documentation/specs/glut/spec3/spec3.html` (Aufrufdatum: 24.01.2010)

[NVI08] Greg Ruetsch, Brent Oster: NVISION 08 - Getting Started with CUDA, `http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf` (Aufrufdatum: 15.12.2009)

[NVI09a] NVIDIA: NVIDIA CUDA - Programming Guide, Version 2.3, Juli 2009 (`www.nvidia.com/cuda`, `http://forums.nvidia.com/index.php?showtopic=102548` Aufrufdatum 25.07.2009)

[NVI09b] NVIDIA: CUDA C Programming Best Practices Guide - CUDA Toolkit, Version 2.3, Juli 2009 (`www.nvidia.com/cuda`, `http://forums.nvidia.com/index.php?showtopic=102548` Aufrufdatum 25.07.2009)

[NVI09c] NVIDIA: The CUDA Compiler Driver NVCC - CUDA Toolkit, Version 2.3, 29.07.2009 (`www.nvidia.com/cuda`, `http://forums.nvidia.com/index.php?showtopic=102548` Aufrufdatum 25.07.2009)

[NVI09d] NVIDIA: CUDA Reference Manual , Version 2.3, Juli 2009 (`www.nvidia.com/cuda`, `http://forums.nvidia.com/index.php?showtopic=102548` Aufrufdatum 25.07.2009)

[NVI09e] NVIDIA: CUDA Visual Profiler Help File, Version 2.3, 2009 (`www.nvidia.com/cuda`, `http://forums.nvidia.com/index.php?showtopic=102548` Aufrufdatum 25.07.2009) – enthalten in CUDA Visual Profiler Version 2.3 `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/cudaprof_2.3_readme.txt` (Aufrufdatum: 15.01.2010)

[NVI09f] NVIDIA: NVIDIA CUDA SDK - Code Samples, Version 2.3, Juli 2009 (`www.nvidia.com/cuda`, `http://forums.nvidia.com/index.php?showtopic=102548` Aufrufdatum 25.07.2009)

[NVI09g] NVIDIA: CUDA - CUFFT Library, Version 2.3, Juni 2009 (`www.nvidia.com/cuda`, `http://forums.nvidia.com/index.php?showtopic=102548` Aufrufdatum 25.07.2009)

[PNM03] PNM: Portable Any Map - Bilddateiformat `http://netpbm.sourceforge.net/doc/pnm.html` (Aufrufdatum: 24.01.2010)

[RDX2] Fast Fourier Transform (FFT) `http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html` (Aufrufdatum: 12.11.2009)

[SDS87] Samuel D. Stearns: Digitale Verarbeitung analoger Signale - Digital Signal Analysis, R. Oldenbourg Verlag GmbH, München, 3. Aufl., 1987, ISBN 3-486-20329-0

[TH05] Thorsten Hermes: Digitale Bildverarbeitung - Eine praktische Einführung, Carl Hanser Verlag, München, 2005, ISBN 3-446-22969-8

[TIC64] Dokumentation: TMS320C6416 - Fixed-Point Digital Signal Processor `http://focus.ti.com/docs/prod/folders/print/tms320c6416.html` (Aufrufdatum: 04.01.2010)

[TIC67] Dokumentation: TMS320C6713B - Floating-Point Digital Signal Processors `http://focus.ti.com/docs/prod/folders/print/tms320c6713b.html` (Aufrufdatum: 04.01.2010)

[TRH08] Tom R. Halfhill: Parallel Processing With Cuda , 28.01.2008, `http://www.nvidia.de/docs/IO/55972/220401_Reprint.pdf` (Aufrufdatum: 10.12.2009)

[WA06] Wolfgang Andermahr: Test - nVidia GeForce 8800 GTX, S.3, 8.11.2006 `http://www.computerbase.de/artikel/hardware/grafikkarten/2006/test_nvidia_geforce_8800_gtx/3/#abschnitt_technik_im_detail_part_1` (Aufrufdatum: 8.12.2009)

[WB] Windows Bitmap: BMP - Bilddateiformat `http://de.wikipedia.org/wiki/Windows_Bitmap` (Aufrufdatum: 24.01.2010)

Anhang A

Compute Capability Specifications

Spezifikationen bezogen aus dem Cuda Prgramming Guide (Quelle: [NVI09a] Appendix A.1.1 - A.1.3, S. 102, 103)

A.1 Compute Capability 1.0

A.1.1 Thread, Warp and Block Specifications

- ▷ The maximum number of threads per block is 512
- ▷ The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively
- ▷ The maximum size of each dimension of a grid of thread blocks is 65535
- ▷ The warp size is 32 threads
- ▷ The maximum number of active blocks per multiprocessor is 8
- ▷ The maximum number of active warps per multiprocessor is 24
- ▷ The maximum number of active threads per multiprocessor is 768

A.1.2 Memory Specifications

- ▷ The number of registers per multiprocessor is 8192
- ▷ The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks (see Section 5.1.2.5 [NVI09a])
- ▷ The total amount of constant memory is 64 KB
- ▷ The total amount of local memory per thread is 16 KB
- ▷ The cache working set for constant memory is 8 KB per multiprocessor
- ▷ The cache working set for texture memory varies between 6 and 8 KB per multiprocessor

A.1.3 Programming Specifications

- ▷ For a one-dimensional texture reference bound to a CUDA array, the maximum width is 213
- ▷ For a one-dimensional texture reference bound to linear memory, the maximum width is 227
- ▷ For a two-dimensional texture reference bound to linear memory or a CUDA array, the maximum width is 216 and the maximum height is 215
- ▷ For a three-dimensional texture reference bound to a CUDA array, the maximum width is 211, the maximum height is 211, and the maximum depth is 211
- ▷ The limit on kernel size is 2 million PTX instructions

A.2 Compute Capability 1.1

- ▷ Support for atomic functions operating on 32-bit words in global memory

A.3 Compute Capability 1.2

A.3.1 Thread, Warp and Block Specifications

- ▷ The maximum number of active warps per multiprocessor is 32
- ▷ The maximum number of active threads per multiprocessor is 1024.

A.3.2 Memory Specifications

- ▷ The number of registers per multiprocessor is 16384

A.3.3 Programming Specifications

- ▷ Support for atomic functions operating in shared memory and atomic functions operating on 64-bit words in global memory (see Section B.10 [NVI09a])
- ▷ Support for warp vote functions (see Section B.11 [NVI09a])

A.4 Compute Capability 1.3

- ▷ Support for double-precision floating-point numbers.

Anhang B

NVIDIA CUDA Visual Profiler Version 2.3

List of supported features [1]

Execute a CUDA program with profiling enabled and view the profiler output as a table. The table has the following columns for each GPU method:

- **GPU Timestamp**: Start time stamp
- **Method**: GPU method name. This is either "memcpy*" for memory copies or the name of a GPU kernel. Memory copies have a suffix that describes the type of a memory transfer, e.g. "memcpyDToHasync" means an asynchronous transfer from Device memory to Host memory.
- **GPU Time**
- **CPU Time**
- **Stream Id** : Identification number for the stream
- Columns only for kernel methods:
  - **Occupancy** : Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of active warps.
  - **Profiler counters**:
    - **gld uncoalesced** : Number of non-coalesced global memory loads
    - **gld coalesced** : Number of coalesced global memory loads
    - **gld request** : Number of global memory load requests (available only for GPUs with compute capability 1.2 or higher)
    - **gld_32/64/128b** : Number of 32 byte, 64 byte and 128 byte global memory load transactions (available only for GPUs with compute capability 1.2 or higher).

[1] Spezifikationen entnommen aus der CUDA Visual Profiler Helpfile (Quelle: [NVI09e] Am Anfang des Dokuments)

        * **gst uncoalesced** : Number of non-coalesced global memory stores
        * **gst coalesced** : Number of coalesced global memory stores
        * **gst request** : Number of global memory store requests (available only for GPUs with compute capability 1.2 or higher)
        * **gst_32/64/128b** : Number of 32 byte, 64 byte and 128 byte global memory store transactions (available only for GPUs with compute capability 1.2 or higher).
        * **local load** : Number of local memory loads
        * **local store** : Number of local memory stores
        * **tlb hit** : Number of instruction or constant memory cache hits
        * **tlb miss** : Number of instruction or constant memory cache misses
        * **sm cta launched** : Number of threads blocks launched on a multiprocessor
        * **branch** : Number of branches taken by threads executing a kernel
        * **divergent branch** : Number of divergent branches within a warp
        * **instructions** : Number of instructions executed
        * **warp serialize** : Number of thread warps that serialize on address conflicts to either shared or constant memory
        * **cta launched** : Number of threads blocks executed
    - **grid size X** : Number of blocks in the grid along dimension X
    - **grid size Y** : Number of blocks in the grid along dimension Y
    - **block size X** : Number of threads in a block along dimension X
    - **block size Y** : Number of threads in a block along dimension Y
    - **block size Z** : Number of threads in a block along dimension Z
    - **dyn smem per block** : Dynamic shared memory size per block in bytes
    - **sta smem per block**: Static shared memory size per block in bytes
    - **reg per thread**: Number of registers per thread

▷ **Columns only for memcopy methods:**

    - **mem transfer size**: Memory transfer size in bytes

Please refer the "Interpreting Profiler Counters" section below for more information on profiler counters. Note that profiler counters are also referred to as profiler signals.

Display the summary profiler table. It has the following columns for each GPU method:

▷ **Method**: Method name

▷ **#calls**: Number of calls

▷ GPU usec: Total GPU time in micro seconds

▷ **CPU usec**: Total CPU time in micro seconds

▷ **%GPU time**: Percentage GPU time

- ▷ **Total counts for each profiler counter**
- ▷ **glob mem read throughput(GB/s)**: Global memory read throughput in giga-bytes per second.
- ▷ **glob mem write throughput(GB/s)**: Global memory write throughput in giga-bytes per second.
- ▷ **glob mem overall throughput(GB/s)**: Global memory write throughput in giga-bytes per second.
- ▷ **instruction throughput** : instruction throughput ratio for each kernel

Display various kinds of plots:

- ▷ Summary profiling data bar plot
- ▷ GPU Time Height plot
- ▷ GPU Time Width plot
- ▷ Profiler counter bar plot
- ▷ Profiler output table column bar plot
- ▷ Comparison Summary plot

Analysis of profiler output lists out method with high number of:

- ▷ Uncoalesced stores
- ▷ Uncoalesced loads
- ▷ Warp serializations

Compare profiler output for multiple program runs of the same program or for different programs.

Each program run is referred to as a session.

Save profiling data for multiple sessions. A group of sessions is referred to as a project.

Import/Export CUDA Profiler CSV format data.

Anhang C

Anhang Projektvorlagen

C.1 Quellcode: Eingesetzte CUDA-Hostfunktionen

```
////////// File: cuda_code.cu //////////////
#include "cuda_kernel.cu" // Devicecode-File
// Header for Hostcode (contains #include "cuda_runtime.h" - Runtime API)
#include "cuda_code.h"

//***************** Error Handling ********************//
__host__ const char* cl_callsafe(cudaError_t error)
{
   if(error!=cudaSuccess)
      return cudaGetErrorString(error);
   else
      return NULL;
}

__host__ bool cl_checkerror(cudaError_t error)
{
   if(error!=cudaSuccess)
   {
      printf("Error: %s \n",cl_callsafe(error));
      return false;
   }
   else
      return true;
}
//*****************************************************//

//***************** Device Init ***********************//
__host__ cudaError_t cl_init_device(char * info)
{
   int cores=0;
   int device=0,count;
```

```
    struct cudaDeviceProp prop;

    // Get Number of Devices
    if(!cl_checkerror(cudaGetDeviceCount(&count)))
        return cudaGetLastError();

    // Choose one Device
    for(int i=0;i<count;i++)
    {
        // Get Device properties
        if(!cl_checkerror(cudaGetDeviceProperties(&prop,i)))
            return cudaGetLastError();

        // Choose Device with Max. Multicores
        if(prop.multiProcessorCount > cores)
        {
            cores=prop.multiProcessorCount;
            device=i;
            strcpy(info,prop.name);
        }
    }
    // Set selected Device
    if(!cl_checkerror(cudaSetDevice(device)))
        return cudaGetLastError();

    return cudaSuccess;
}
// Calls Device Init Funktion
__host__ cudaError_t cl_open_device(void)
{
    char info[256];
    if(!cl_checkerror(cl_init_device(info)))
        return cudaGetLastError();
    else
    {
        printf("Cuda Device: %s\n",info);
        return cudaSuccess;
    }
}
//*****************************************************//

//***************** Timer Functions ******************//
// Start Timer
__host__ cudaError_t cl_start_timer(cudaEvent_t* start,cudaEvent_t* stop,
    cudaStream_t stream)
{
    // Create Start and Stop Event
    cudaEventCreate(start);
    cudaEventCreate(stop);
```

```
    // Start event revcord
    cudaEventRecord(*start, stream ); ;

    return cudaSuccess;
}

// Stop Timer
__host__ cudaError_t cl_stop_timer(cudaEvent_t* start,cudaEvent_t* stop,
    float* time,cudaStream_t stream)
{
    // Stop event revcord
    cudaEventRecord( *stop, stream );
    cudaEventSynchronize( *stop );  // Sync to Stop Event

    // Get Time
    cudaEventElapsedTime( time, *start, *stop );
    // Destroy Events
    cudaEventDestroy( *start );
    cudaEventDestroy( *stop );

    return cudaSuccess;
}

// Start Timer on streamId = 0
__host__ cudaError_t cl_start_timer(cudaEvent_t* start,cudaEvent_t* stop)
{
    return cl_start_timer(start,stop,0);
}

// Stop Timer on streamId = 0
__host__ cudaError_t cl_stop_timer(cudaEvent_t* start,cudaEvent_t* stop,
    float* time)
{
    return cl_stop_timer(start,stop,time,0);
}

//*****************************************************//

//** kernel call parts**//

// Alloc Device Memory
__host__ cudaError_t cl_alloc_d_mem(void** d_idata,void** d_odata,size_t
    datasize)
{
    cudaError_t error = cudaSuccess;

    // MEMALLOC Device
    if(d_idata!=NULL) // Indata of global Memory
```

```
    {
        if((error=cudaMalloc(d_idata,datasize))!=cudaSuccess) return error;
    }

    if(d_odata!=NULL)  // Outdata of global Memory
    {
        if((error=cudaMalloc(d_odata,datasize))!=cudaSuccess)
        {
            error=cudaFree(*d_idata);
            return error;
        }
    }

    return error;
}

// Free Device Memory
__host__ cudaError_t cl_free_d_mem(void* d_idata,void* d_odata)
{
    cudaError_t error = cudaSuccess;

    // FREE Device Mem
    if((error=cudaFree(d_odata))!=cudaSuccess)
    {
        error=cudaFree(d_idata);
        return error;
    }

    if((error=cudaFree(d_idata))!=cudaSuccess) return error;

    return error;
}

// Device Memcpy
__host__ cudaError_t cl_d_memcpy(void* des_data,void* src_data,size_t
    datasize,enum cudaMemcpyKind kind,cudaStream_t stream)
{
    return cudaMemcpyAsync(des_data,src_data,datasize,kind,stream);
}

// Kernel Call -- Example
__host__ cudaError_t cl_run_kernel(float* d_idata,float* d_odata,int N,
    cudaStream_t stream)
{
        dim3 dimBlock(16,16);
        dim3 dimGrid(N,N);
        size_t sharedMem = N*sizeof(float);
```

```
    #ifndef __DEVICE_EMULATION__ // Time is not correct on Device
        Emulation
        // Start Timer
        cudaEvent_t start, stop; float time;
        if(!cl_checkerror(cl_start_timer(&start,&stop,stream)))
            return cudaGetLastError();
    #endif
    cuda_kernel_exp<<< dimGrid,dimBlock,sharedMem,stream >>>(d_idata,
        d_odata,N);

    #ifndef __DEVICE_EMULATION__ // Time is not correct on Device
        Emulation
        // Stop Timer
        if(!cl_checkerror(cl_stop_timer(&start,&stop,&time,stream)))
            return cudaGetLastError();
        printf("Processing time kernel: %f ms\n",time);
    #endif

  return cudaSuccess;
}
```

C.2 Quellcode: Projektvorlage 1 - Allgemein

Dieser Anhang ist in elektronischer Form auf der CD abgelegt und beim Prüfer Prof. Dr. Ing. Hans Peter Kölzer einzusehen.

C.3 Quellcode: Projektvorlage 2 - Bildverarbeitung

Dieser Anhang ist in elektronischer Form auf der CD abgelegt und beim Prüfer Prof. Dr. Ing. Hans Peter Kölzer einzusehen.

C.4 Quellcode: Projekt - CLM Image Streams

Dieser Anhang ist in elektronischer Form auf der CD abgelegt und beim Prüfer Prof. Dr. Ing. Hans Peter Kölzer einzusehen.

Anhang D

Anhang FFT Implementierung

D.1 Device-Quellcode: Erste Implementierung

```
////////// File: cuda_kernel.cu //////////////

// Bit-reverse index function
// in: Index -- bits: Number of bits to reverse
__device__ unsigned int fft_reversed(unsigned int in,int bits)
{
   unsigned int out = 0;
   for(int i=0;i<bits;i++)
   {
      out = (out << 1) | ( in & 0x1);
      in>>=1;
   }
   return out;
}
////////////////////////////////////////////////////////////////
///   Shared-Memory devided into 2 by 2 parts
/// 1. Part1 first Index(0....N-1) -- > 2 Parts 1. Real(0....N/2-1) -- 2.
    Imag (N/2...N-1)
/// 2. Part2 second  Index(N...2N-1) -- > 2 Parts 1. Real(N....3N/2-1) -- 2.
     Imag (2N/2...2N-1)
/// REASON: avoid bank conflicts
/// Size of Shared-Memory: (N/2)*2*2 --> 2 Parts and Real/Imag for each part
////////////////////////////////////////////////////////////////
__global__ void cuda_fft_radix2_kernel(float* d_idata,float* d_odata,int N,
    int stages)
{
   // static shared memory
   __shared__ float i_buffer [2048];
   float2 first,second;

   // Index for global memory
```

```
int x =(blockIdx.x * blockDim.x) << 2 ;
int x_thread = threadIdx.x;

//// --- Reorder Inputdata /////
int index = x + 2*fft_reversed(x_thread*2,stages); // First
i_buffer [x_thread] = d_idata[index];
i_buffer [x_thread + blockDim.x] = d_idata[index + 1];
index = x + 2*fft_reversed(x_thread*2+1,stages); // Second
i_buffer [x_thread + 2 * blockDim.x] = d_idata[index];
i_buffer [x_thread + 3 * blockDim.x] = d_idata[index + 1];
__syncthreads();

// Run calculation
for(int stage = 0; stage < stages; stage++ )
{
   int stepsize = 1 << stage;
   int gruppe = (int)(x_thread / stepsize);
   int BF = (x_thread % stepsize);
   // Calc wk
   int wk_index = BF * (1 <<(stages - (stage + 1)));
   float wkpi  = __fdividef(6.283185307f,(float)N) * (float)wk_index;
   float co = cos(wkpi);
   float si = sin(wkpi);

   /////// Set first and second Index //////////////////
   int indexfirst = (BF + (stepsize << 1) * gruppe);
   int indexsecond = indexfirst + stepsize;

   // Set first and second position in shared memory
   if(indexfirst & 1)
   {
      indexfirst = (indexfirst-1) >> 1;
      indexfirst += 2 * blockDim.x;
   }
   else
      indexfirst >>= 1;

   if(indexsecond & 1)
   {
      indexsecond = (indexsecond-1) >> 1;
      indexsecond += 2 * blockDim.x;
   }
   else
      indexsecond >>= 1;

   ////////////////////////////////////////////
   float cx,cy;
   // read first
   first.x = i_buffer [indexfirst];
```

```
        first.y = i_buffer [indexfirst + blockDim.x];
        // read second
        second.x = i_buffer [indexsecond];
        second.y = i_buffer [indexsecond + blockDim.x];

        cx = co * second.x + si * second.y;
        cy = co * second.y - si * second.x;
        // write first
        i_buffer [indexsecond] = first.x - cx;
        i_buffer [indexsecond + blockDim.x] = first.y - cy;
        // write second
        i_buffer [indexfirst] = first.x + cx;
        i_buffer [indexfirst + blockDim.x] = first.y + cy;

        //////////////
        __syncthreads();
    }

    // Copy back to global Memory
    index = x + (x_thread << 2);
    d_odata[index] = i_buffer [x_thread];
    d_odata[index + 1] = i_buffer [x_thread + blockDim.x];

    index = x + ((x_thread*2+1) << 1);
    d_odata[index] = i_buffer [x_thread + 2 * blockDim.x];
    d_odata[index + 1] = i_buffer [x_thread + 3 * blockDim.x];
}
```

D.2 Visual Profiler: Erste Implementierung

Method	**Gputime in µs**	Cputime in µs	**Occupancy**	Threads
cuda_fft_radix2_kernel	107,456	128,508	0,5	512

divergent_branch	instructions	**warp_serialize**
96	33649	4545

SharedMem/Block		Register/Thread	**gld_32b**	**gst_128b**	gld_request	gst_request
static	**dynamic**	23	2048	256	64	64
8232	0					

Tabelle D.1: Erste Version: Profiler Ergebnisse FFT - 1024 Punkte

D.3 Device-Quellcode: Optimierte Implementierung

```
////////// File: cuda_kernel.cu //////////////
```

```
__global__ void cuda_fft_radix2_kernel(float4* d_idata,float4* d_odata,int n
    )
{
   // shared memory
   extern __shared__ float i_buffer [];
   // Index for global memory
   int x = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;

   float4 calc; // Buffer
   // Calc Stages (Steps)
   int stages = (int)(logf((float)n)/logf(2.f));
   float PI2_N = (6.283185307f /(float)n);

   // Read data from global memory
   calc = d_idata[x];

   // Save Even-Values
   int index = fft_reversed((threadIdx.x << 1),stages);
   i_buffer [index] = calc.x;                  // even real
   i_buffer [index + (blockDim.x << 1)] = calc.y;  // even  imag
   __syncthreads();
   // Save Odd-Values
   index = index + blockDim.x;
   i_buffer [index ] = calc.z;                     // odd real
   i_buffer [index + (blockDim.x << 1)] = calc.w;  // odd imag
   __syncthreads();

   // Run calculation
   for(int stage = 0; stage < stages; stage++ )
   {
      // stepsize = 2^stage
      int stepsize = 1 << stage;
      // BF = (x_thread % stepsize)
      int BF = (threadIdx.x & (stepsize-1));
      // groupe = (x_thread / stepsize)
       // (x_thread >> log2(stepsize)) ---> stage = log2(stepsize)
      int groupe = (threadIdx.x >> stage);

      /// Calc first and second index of Butterfly ///
      // BF + 2*stepsize*groupe
      int indexfirst = (BF + __mul24((stepsize << 1), groupe));
      // Stride = stepsize --- Stride = 2^stage
      int indexsecond = indexfirst + stepsize;

      ////////////////// BUTTERFLY ////////////////////////////
      // Read Second
      calc.z = i_buffer [indexsecond];                // Real
      calc.w = i_buffer [indexsecond + (blockDim.x << 1)];  // Imag
      __syncthreads();
```

```
        // wk_index = BF*2^(stages - (stage + 1)) // BF = (x_thread % stepsize
            )
        // wkpi = ((6.283185307f /(float)n)*(float)wk_index);
        float wkpi = (PI2_N*((float)__mul24(BF ,(1 <<(stages - (stage + 1)))))
            );
        float2 c = make_float2(cosf(wkpi) * calc.z + sinf(wkpi) * calc.w,
                               cosf(wkpi) * calc.w - sinf(wkpi) * calc.z);

        // Read First
        calc.x = i_buffer [indexfirst];                    // Real
        calc.y = i_buffer [indexfirst + (blockDim.x << 1)];      // Imag

        __syncthreads(); // faster  -- Less Warp Serialized
        // Write First
        i_buffer [indexfirst] = calc.x + c.x;
        i_buffer [indexfirst + (blockDim.x << 1)] = calc.y + c.y;
        // Write Second
        i_buffer [indexsecond] = calc.x - c.x;
        i_buffer [indexsecond + (blockDim.x << 1)] = calc.y - c.y;
        ////////////
        __syncthreads();

        ////////// END BUTTERFLY ////////////////////////

    }
    // Copy back to global Memory
    index = (threadIdx.x << 1);
    d_odata[x] = make_float4(i_buffer [index],i_buffer [index + (blockDim.x
        <<1)],
                i_buffer [index + 1],i_buffer [index + (blockDim.x<<1) + 1]);
}
```

D.4 Visual Profiler: Optimierte Implementierung

Method	**Gputime in µs**	Cputime in µs	**Occupancy**	Threads
cuda_fft_radix2_kernel	39,968	60,063	0,5	512

divergent_branch	instructions	**warp_serialize**
1	9218	3302

SharedMem/Block		Register/Thread	**gld_128b**	**gst_128b**	gld_request	gst_request
static	**dynamic**	14	64	64	16	16
48	8192					

Tabelle D.2: Optimierte Version: Profiler Ergebnisse FFT - 1024 Punkte

D.5 Host-Quellcode: FFT Algorithmus

```
////////// File: cuda_code.cu ////////////

// Invoke Kernel
__host__ cudaError_t cl_run_kernel(float* d_idata,
                                   float* d_odata,
                                          int N,int anzFFT,
                                          cudaStream_t stream)
{
   // N Punkte FFT ---> N/2 Threads pro Block -- Eindimensional
   dim3 dimBlock(N>>1,1);
   // anzFFT --- Anzahl der FFTs = Anzahl der Blöcke -- Eindimensional
   dim3 dimGrid(anzFFT,1);
   // N-Punkte * sizeof(float2) für Real und Imag
   size_t shared_mem = N*sizeof(float2);
   cuda_fft_radix2_kernel<<< dimGrid,dimBlock,shared_mem,stream >>>
                       ((float4*)d_idata,(float4*)d_odata,N);
   return cudaSuccess;
}
```

```
////////// File: main.cpp ////////////

#define N_CNT 1024
#define ANZ 27
#define STREAMS 5

int exitRun(int ret);
cudaError_t destroyStreams(cudaStream_t* streams, int anz);

int main()
{
   // Device Init
   if(!cl_checkerror(cl_open_device())) return 1;

   // Datasize
   int pitch = (ANZ * (N_CNT << 1));
   size_t datasize = pitch * sizeof(float);

   //// Create Signal on host ///
   float* h_data=NULL;
   if(!cl_checkerror(cudaHostAlloc((void**)&h_data,datasize*STREAMS,
       cudaHostAllocPortable)))
      return 1;

   // OutData for the host
   float* h_out=NULL;
   if(!cl_checkerror(cudaHostAlloc((void**)&h_out,datasize,
       cudaHostAllocPortable))) return 1;
```

```
////// Test Values ///////////
for(int j = 0;j<ANZ*STREAMS;j++)
{
    for(int i=0; i<N_CNT; i++)
    {
        h_data[2*(i+j*N_CNT)]= 256.f*sinf((2.f*PI_F*5.5f*i)/((float)N_CNT))
            ;
        h_data[2*(i+j*N_CNT)+1]= 0.f;
    }
}

// Create Cuda Streams //
cudaStream_t streams[STREAMS];
for(int i=0;i<STREAMS;i++)
    if(!cl_checkerror(cudaStreamCreate(&(streams[i]))))
    {
        for(int j=0;j<i;j++)
            cudaStreamDestroy(streams[j]);
        return exitRun(1);
    }
//////////////////////

///// Kernel alloc, copy and run /////
float *d_idata=NULL;
float *d_odata=NULL;

// Alloc Device Mem
if(!cl_checkerror(cl_alloc_d_mem((void**)&d_idata,(void**)&d_odata,
    datasize*STREAMS)))
{
    printf("\nError: Allocmem\n");
    cudaFreeHost(h_data);
    destroyStreams(streams,STREAMS);
    return exitRun(1);
}

// Start Timer
cudaEvent_t start, stop; float time;
if(!cl_checkerror(cl_start_timer(&start,&stop)))
    return cudaGetLastError();

/*cudaMemcpyAsync(d_idata,h_data,datasize,cudaMemcpyHostToDevice,0);
cl_run_kernel(d_idata,d_odata,N_CNT,ANZ,0);
cudaMemcpyAsync(h_out,d_odata,datasize,cudaMemcpyDeviceToHost,0);*/

// Copy Mem to Device
for(int i=0;i<STREAMS;i++)
{
```

```
    if(!cl_checkerror(cl_d_memcpy(d_idata + i*pitch,
                    h_data +i*pitch,datasize,
                    cudaMemcpyHostToDevice,streams[i])))
    {
        printf("\nError: Copy Mem H->D\n");
        cl_free_d_mem(d_idata,d_odata);
        cudaFreeHost(h_data);
        destroyStreams(streams,STREAMS);
        return exitRun(1);
    }
}

// RUN KERNEL //
for(int i=0;i<STREAMS;i++)
    if(!cl_checkerror(cl_run_kernel(d_idata + i*pitch,
                            d_odata +i*pitch,
                            N_CNT,ANZ,streams[i])))
    {
        printf("\nError: Run Kernel\n");
        cl_free_d_mem(d_idata,d_odata);
        cudaFreeHost(h_data);
        destroyStreams(streams,STREAMS);
        return exitRun(1);
    }

// Copy Mem from Device
for(int i=0;i<STREAMS;i++)
    if(!cl_checkerror(cl_d_memcpy(h_data + i*pitch,
                            d_odata +i*pitch,datasize,
                            cudaMemcpyDeviceToHost,streams[i])))
    {
        printf("\nError: Copy Mem D->H\n");
        cl_free_d_mem(d_idata,d_odata);
        cudaFreeHost(h_data);
        destroyStreams(streams,STREAMS);
        return exitRun(1);
    }

cudaThreadSynchronize();

// Stop Timer
if(!cl_checkerror(cl_stop_timer(&start,&stop,&time)))
    return cudaGetLastError();
printf("Processing time GPU: %f ms\n",time);

/*for(int i = 0 ; i < ANZ*N_CNT*2 ; i+=2)
    printf("Dataset: %d Real: %f , Imag: %f\n",(i/(N_CNT*2))+1,h_out[i],
        h_out[i+1]);*/
```

```
    //// Free All Mem ///
    if(!cl_checkerror(cl_free_d_mem(d_idata,d_odata)))
    {
        cudaFreeHost(h_data);
        destroyStreams(streams,STREAMS);
        return exitRun(1);
    }

    if(!cl_checkerror(cudaFreeHost(h_data))) return exitRun(1);
    if(!cl_checkerror(cudaFreeHost(h_out))) return exitRun(1);

    ////////////////////

    // Destroy Cuda Stream //
    if(!cl_checkerror(destroyStreams(streams,STREAMS))) return exitRun(1);
    ////////////////////////

    // Close Device --- Very Importend !!!! (see exitRun-Function)
    ////////////////////////////
    return 0;
}

int exitRun(int ret)
{
    cudaThreadExit();
    printf("\nPress␣Return␣to␣exit!\n");
    getchar();
    return ret;
}

cudaError_t destroyStreams(cudaStream_t* streams, int anz)
{
    for(int j=0;j<anz;j++)
        cudaStreamDestroy(streams[j]);

    return cudaGetLastError();
}
```

D.6 Quellcode: Gesamtes FFT-Projekt

Dieser Anhang ist in elektronischer Form auf der CD abgelegt und beim Prüfer Prof. Dr. Ing. Hans Peter Kölzer einzusehen.

Anhang E

Anhang Faltungsoperation

E.1 Quellcode: Implementierung ein Farbkanal

```
#define LINE 3
#define BLOCK LINE*LINE

// Choose Memory Types
#define SHARED 0
#define TEX 0

// Zero Padding for globale memory
#define ZEROPAD 0

#define SH_SIZE (THREADBLOCK+(LINE/2)*2)*(THREADBLOCK+(LINE/2)*2)
#define SH_OFFSET (LINE/2)
// Convolution-Matrix
__constant__ float matrix[BLOCK];
// texture referenz
texture<BYTE, 2, cudaReadModeElementType> tex_byte;

#if TEX // If Texture Memory
__global__ void cuda_Convolution_OneChannel(BYTE* d_odata,int width,int
    height)
#else
__global__ void cuda_Convolution_OneChannel(BYTE* d_idata,BYTE* d_odata,int
    width,int height)
#endif
{
   int row = __mul24(blockIdx.y,blockDim.y) + threadIdx.y ; // row
   int col = __mul24(blockIdx.x,blockDim.x) + threadIdx.x ; // column

   // Start Index for Convolution (row and col)
   int start = __mul24(-1,(LINE>>1));
   // End Index for Convolution (row and col)
```

```
    int end =  (LINE>>1) +1;

    // Sum-Var
    float value = 0.f;
    int i = 0,j=0;

#if SHARED // If Shared Memory
    __shared__ float buffer[SH_SIZE];
    if( (row < (height+(SH_OFFSET<<1))) &&  (col < width+(SH_OFFSET<<1)))
    {
        // Get top left corner of thread block
        int top_block = __mul24(blockIdx.y,blockDim.y) - SH_OFFSET ;
        int left_block = __mul24(blockIdx.x,blockDim.x) - SH_OFFSET ;

        // Copy elements into shared memory
        for(int x_index = threadIdx.x ; x_index < (THREADBLOCK + (SH_OFFSET <<
             1)); x_index+=THREADBLOCK)
            for(int y_index = threadIdx.y ; y_index < (THREADBLOCK + (SH_OFFSET
                 << 1)); y_index+=THREADBLOCK)
            {
                // Access Shared Memory
                // Buffer Index = pitch*index_y + index_x -- pitch = (blockDim.x
                     + OFFSET*2)
                // Data Index = pitch*y + x -- pitch = width, y = top + index_y,
                     x = left + index_x
                int row_data = top_block + y_index;
                int col_data = left_block  + x_index;

                if(( row_data >= 0) && ( row_data < height) && ( col_data >= 0)
                    && ( col_data < width))
                {
                #if TEX
                    buffer [__mul24((blockDim.x + (SH_OFFSET << 1)),y_index) +
                        x_index]
                          = (float)tex2D(tex_byte, col_data, row_data);
                #else
                    buffer [__mul24((blockDim.x + (SH_OFFSET << 1)),y_index) +
                        x_index]
                          = (float)d_idata[__mul24(width,row_data) + col_data];
                #endif
                }
                else
                    buffer [__mul24((blockDim.x + (SH_OFFSET << 1)),y_index) +
                        x_index] =  0.f; // Zero-Padding

            }
        __syncthreads();

        if( (row < height) &&  (col < width))
```

```
      {
         for(i=start;i<end;i++) // columns
            for(j=start;j<end;j++) // rows
            {
               // Access Shared Memory
               // pitch = (blockDim.x + OFFSET*2)
               // Buffer Index = pitch*(thread_y + j + OFFSET) + (thread_y +
                    i + OFFSET)
               // Matrix Index = ((LINE/2) + j)*LINE + i - START
               value += matrix[i + (LINE>>1) + __mul24((j+(LINE>>1)),LINE)]
                     *buffer [__mul24((blockDim.x + (SH_OFFSET << 1)),(
                         threadIdx.y + j + SH_OFFSET)) + threadIdx.x + i +
                         SH_OFFSET];
               __syncthreads();
            }
         // Copy back data
         d_odata[__mul24(width,row) + col] = fmaxf(0.f,fminf(value,255.f));
      }
   }
#else // Global Memory
   if((row >= abs(start)) && (row < (height + start)) && (col >= abs(start))
       && (col < (width + start)))// Without border
   {
      for(i=start;i<end;i++) // columns
         for(j=start;j<end;j++) // rows
         {
            // Data Index = pitch*y + x  --  pitch = width , y = row + j , x
                 = (col + i) //
            // Matrix Index = pitch*y + x -- pitch = LINE, y = (LINE/2) + j,
                 x = i + (LINE/2)
            value += matrix[i + (LINE>>1) + __mul24((j+(LINE>>1)),LINE)]
            #if TEX
                  *((float)tex2D(tex_byte, col + i, row + j));
            #else
                  *((float)d_idata[__mul24(width, row + j) + col + i]);
            #endif
         }

      // Write Result
      d_odata[__mul24(width,row) + col] = (BYTE) fmaxf(0.f,fminf(value,255.f
          ));
   }
  #if ZEROPAD // Zero-Padding with Global Memory
   else if( (row < height) &&  (col < width))
   {
      for(i=start;i<end;i++) // columns
         for(j=start;j<end;j++) // rows
         {
            int row_calc = row + j;
```

```
            int col_calc = col + i;
            if ((row_calc >= 0) && (row_calc < height) && (col_calc >= 0)
                && (col_calc < width))
            {
               // Data Index = pitch*y + x //
               // pitch = w_channel , y = row + j , x = (col + i)
               // Matrix Index = pitch*y + x //
               // pitch = LINE, y = (LINE/2) + j, x = i + (LINE/2)
               value += matrix[i + (LINE>>1) + __mul24((j+(LINE>>1)),LINE)]
               #if TEX
                     *((float)tex2D(tex_byte, col_calc, row_calc));
               #else
                     *((float)d_idata[__mul24(width, row_calc) + col_calc]);
               #endif
            }
         }
      // Write Result
      d_odata[__mul24(width,row) + col] = (BYTE) fmaxf(0.f,fminf(value,255.f
          ));
   }
  #endif // End Zero-Pad
#endif // End Shared
}
```

E.2 Quellcode: Grundstruktur Kernelaufruf

```
////////// File: cuda_code.cu ////////////
__host__ cudaError_t cl_run_kernel(
                           BYTE* d_idata,
                           BYTE* d_odata,
                           imagedata image,
                           cudaStream_t stream)
{
   ////// KERNEL SETTINGS
   dim3 dimBlock(THREADBLOCK,THREADBLOCK); // Threadblocksize
   // Gridsize: min. dimGrid.x = width , min. dimGrid.y = height
   dim3 dimGrid((image.width + dimBlock.x - 1) / dimBlock.x,
            (image.height + dimBlock.y - 1) / dimBlock.y);

   float matrix_data [BLOCK] = {0.1111111f,0.1111111f,0.1111111f,
                        0.1111111f,0.1111111f,0.1111111f,
                           0.1111111f,0.1111111f,0.1111111f};
   // Copy convolution matrix in constant memory
   cudaMemcpyToSymbol(matrix,matrix_data,
                  BLOCK*sizeof(float),0,cudaMemcpyHostToDevice);

   cuda_convolution_kernel<<<dimGrid,dimBlock,0,stream >>>
                  (d_idata,d_odata,image.width,image.height);
```

```
    return cudaSuccess;
}
```

E.3 Visual Profiler: Global Memory

Method	**Gputime in µs**	Cputime in µs	**Occupancy**
cuda_convolution_kernel	106,624	128,787	1

gridSizeX	gridSizeY	blockSizeX	blockSizeY
40	30	16	16

divergent_branch	instructions	**Threadblocks Launched**
73	25986	134

SharedMem/Block		Register/Thread	**gld_32b**	**gst_32b**	gld_request	gst_request
static	**dynamic**					
40	0	10	24795	2094	3168	352

Tabelle E.1: Grundstruktur Faltung: Mittelwertfilter mit 640×472 Grauwertdaten

Method	**Gputime in µs**	Cputime in µs	**Occupancy**
cuda_convolution_kernel	194,688	217,346	1

gridSizeX	gridSizeY	blockSizeX	blockSizeY
40	30	16	16

divergent_branch	instructions	**Threadblocks Launched**
78	32660	134

SharedMem/Block		Register/Thread	**gld_32b**	**gst_32b**	gld_request	gst_request
static	**dynamic**					
40	0	10	25539	2215	3332	379

Tabelle E.2: Grundstruktur Faltung: Mittelwertfilter mit 640×472 Grauwertdaten mit Zeropadding

E.4 Quellcode: Texture Memory

```
////////// File: cuda_code.cu ////////////

__host__ cudaError_t cl_tex_kernel_byte(BYTE* d_idata,
                                        BYTE* d_odata,
                                        imagedata image,
                                        cudaStream_t stream)
{
    // Create ChannelDesc
    cudaChannelFormatDesc description = cudaCreateChannelDesc(8,0,0,0,
        cudaChannelFormatKindUnsigned);
```

```
    // Bind Texture Reference
    textureReference* texPtr;
    cudaGetTextureReference((const textureReference**)&texPtr, "tex_byte");
    cudaBindTexture2D(0, texPtr, d_idata, &description,
                image.width, image.height, sizeof(BYTE)*image.width);

    // Run Kernel
    dim3 dimBlock(THREADBLOCK,THREADBLOCK);
    dim3 dimGrid((image.width + dimBlock.x - 1) / dimBlock.x,
             (image.height + dimBlock.y - 1) / dimBlock.y);

    float matrix_data [BLOCK] = {0.1111111f,0.1111111f,0.1111111f,
                         0.1111111f,0.1111111f,0.1111111f,
                            0.1111111f,0.1111111f,0.1111111f};
    // Copy convolution matrix in constant memory
    cudaMemcpyToSymbol(matrix,matrix_data,
                    BLOCK*sizeof(float),0,cudaMemcpyHostToDevice);

    cuda_convolution_kernel<<< dimGrid, dimBlock, 0 ,stream>>>
                        (d_odata, image.width, image.height);

    // Unbind Texture
    cudaUnbindTexture(tex_byte);

    return cudaSuccess;
}
```

E.5 Visual Profiler: Texture Memory

Method	**Gputime in µs**	Cputime in µs	**Occupancy**
cuda_convolution_kernel	167,104	190,527	1

gridSizeX	gridSizeY	blockSizeX	blockSizeY
40	30	16	16

divergent_branch	instructions	**Threadblocks Launched**
73	55222	134

SharedMem/Block		Register/Thread	**gst_32b**	gst_request
static	**dynamic**	11	2094	352
32	0			

Tabelle E.3: Faltung mit Texture Memory: Mittelwertfilter mit 640 × 472 Grauwertdaten

Method	**Gputime in µs**	Cputime in µs	**Occupancy**
cuda_convolution_kernel	245,92	268,19	1

gridSizeX	gridSizeY	blockSizeX	blockSizeY
40	30	16	16

divergent_branch	instructions	**Threadblocks Launched**
78	60485	134

SharedMem/Block		Register/Thread	**gst_32b**	gst_request
static	**dynamic**	9	2215	379
32	0			

Tabelle E.4: Faltung mit Texture Memory: Mittelwertfilter mit 640 × 472 Grauwertdaten mit Zeropadding

E.6 Visual Profiler: Datenpuffer im Shared Memory

Method	**Gputime in µs**	Cputime in µs	**Occupancy**
cuda_convolution_kernel	162,4	186,057	1

gridSizeX	gridSizeY	blockSizeX	blockSizeY
40	30	16	16

divergent_branch	instructions	**Threadblocks Launched**
434	48876	134

SharedMem/Block		Register/Thread	**gld_32b**	**gst_32b**	gld_request	gst_request
static	**dynamic**	14	6961	2104	794	352
1336	0					

Tabelle E.5: Faltung mit Shared Memory: Mittelwertfilter mit 640 × 472 Grauwertdaten

E.7 Quellcode: Flexible Implementierung - Unabhängige Anzahl Farbkanäle

```
////////// File: cuda_kernel.cu //////////////

#define LINE_FINAL 9
#define BLOCK_FINAL LINE_FINAL*LINE_FINAL
#define THREADBLOCK 16
#define ZEROPAD 0

// Convolution-Matrix
__constant__ float matrix[BLOCK_FINAL];
```

```
__global__ void cuda_convolution_kernel(BYTE* d_idata,BYTE* d_odata,
                                int width,int height,int channels)
{
   int row = __mul24(blockIdx.y,blockDim.y) + threadIdx.y ; // row
   int col = __mul24(blockIdx.x,blockDim.x) + threadIdx.x ; // column
   // Start Index for Convolution (row and col)
   int start = __mul24(-1,(LINE_FINAL>>1));
   // End Index for Convolution (row and col)
   int end =  (LINE_FINAL>>1) +1;

   int w_channel = __mul24(width,channels); // pitch = width*channels ,
       stride = channels
   float value = 0.f; // Sum-Var
   int i = 0,j=0;

   if((row >= abs(start))
      && (row < (height + start))
      && (col >= __mul24(abs(start),channels))
      && (col < __mul24((width + start),channels)))// Without border
   {
      for(i=start;i<end;i++) // columns
         for(j=start;j<end;j++) // rows
         {
            // Data Index = pitch*y + x //
            // pitch = w_channel , y = row + j , x = (col + i*channels)
            // Matrix Index = pitch*y + x //
            // pitch = LINE, y = (LINE/2) + j, x = i + (LINE/2)
            value += (float)d_idata[__mul24(w_channel,(row + j)) + col +
                __mul24(i,channels)]
                   * matrix[i + (LINE_FINAL>>1) + __mul24((j+(LINE_FINAL>>1))
                       ,LINE_FINAL)];
         }

      // Write Result
      d_odata[__mul24(w_channel,row) + col] = (BYTE) fmaxf(0.f,fminf(value
          ,255.f));
   }
#if ZEROPAD
   else if( (row < height) &&  (col < w_channel))
   {
      for(i=start;i<end;i++) // columns
         for(j=start;j<end;j++) // rows
         {
            int row_calc = row + j;
            int col_calc = col + __mul24(i,channels);

            if (( row_calc >= 0) && ( row_calc < height) && ( col_calc >= 0)
                && ( col_calc < w_channel))
            {
```

```
                // Data Index = pitch*y + x //
                // pitch = w_channel , y = row + j , x = (col + i*channels)
                // Matrix Index = pitch*y + x //
                // pitch = LINE, y = (LINE/2) + j, x = i + (LINE/2)
                value += (float)d_idata[__mul24(w_channel,row_calc) +
                    col_calc]
                      * matrix[i + (LINE_FINAL>>1) + __mul24((j+(LINE_FINAL
                          >>1)),LINE_FINAL)];
            }
        }
    // Write Result
    d_odata[channels*width*row + col] = (BYTE) fmaxf(0.f,fminf(value,255.f
        ));
  }
#endif
}
```

```
////////// File: cuda_code.cu ////////////
__host__ cudaError_t cl_run_kernel(
                          BYTE* d_idata,
                          BYTE* d_odata,
                          imagedata image,
                          cudaStream_t stream)
{
  ////// KERNEL SETTINGS
  dim3 dimBlock(THREADBLOCK,THREADBLOCK); // Threadblocksize
  // Gridsize: min. dimGrid.x = width*channels , min. dimGrid.y = height
  dim3 dimGrid((image.width*image.channels + dimBlock.x - 1) / dimBlock.x,
          (image.height + dimBlock.y - 1) / dimBlock.y);

  float matrix_data [BLOCK_FINAL] = {0.1111111f,0.1111111f,0.1111111f,
                    0.1111111f,0.1111111f,0.1111111f,
                       0.1111111f,0.1111111f,0.1111111f};
  // Copy convolution matrix in constant memory
  cudaMemcpyToSymbol(matrix,matrix_data,
                  BLOCK_FINAL*sizeof(float),0,cudaMemcpyHostToDevice);
  // Invoke Kernel
  cuda_convolution_kernel<<<dimGrid,dimBlock,0,stream >>>
                  (d_idata,d_odata,image.width,image.height,image.channels);

  return cudaSuccess;
}
```

E.8 Visual Profiler: Flexible Implementierung - Unabhängige Anzahl Farbkanäle

Method	**Gputime in µs**	Cputime in µs	**Occupancy**
cuda_convolution_kernel	290,688	324,343	1

gridSizeX	gridSizeY	blockSizeX	blockSizeY
40	30	16	16

divergent_branch	instructions	**Threadblocks Launched**
176	96956	400

SharedMem/Block		Register/Thread	**gld_32b**	**gst_32b**	gld_request	gst_request
static	**dynamic**					
44	0	9	74751	6269	9468	1052

Tabelle E.6: Faltung: Mittelwertfilter mit 640 × 472 RGB-Daten

E.9 GPU-Ausführungszeiten mit Grauwertbilddaten

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,165336 ms	0,532524 ms	2,157789 ms
5 × 5	0,384863 ms	1,328406 ms	5,417633 ms
7 × 7	0,684358 ms	2,424107 ms	10,343882 ms
9 × 9	1,064003 ms	3,934374 ms	16,868229 ms

Tabelle E.7: Kernelausführungszeiten für Grauwertdaten: Eingangsdaten im Global Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,398233 ms	1,041974 ms	3,772649 ms
5 × 5	0,733231 ms	1,94009 ms	7,110899 ms
7 × 7	0,929045 ms	2,949728 ms	12,026702 ms
9 × 9	1,317943 ms	4,480198 ms	18,534584 ms

Tabelle E.8: Ausführungszeiten mit Datentransfer für Grauwertdaten: Eingangsdaten im Global Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,169408 ms	0,540621 ms	2,029902 ms
5 × 5	0,32643 ms	1,176342 ms	4,518983 ms
7 × 7	0,488355 ms	1,833458 ms	7,168323 ms
9 × 9	0,718106 ms	2,69347 ms	10,580062 ms

Tabelle E.9: Kernelausführungszeiten für Grauwertdaten: Eingangsdaten im Global Memory mit Puffer im Shared Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,397567 ms	1,084902 ms	3,672069 ms
5 × 5	0,519902 ms	1,646237 ms	6,142283 ms
7 × 7	0,752965 ms	2,388777 ms	8,837447 ms
9 × 9	0,979248 ms	3,17976 ms	12,216979 ms

Tabelle E.10: Ausführungszeiten mit Datentransfer für Grauwertdaten: Eingangsdaten im Global Memory mit Puffer im Shared Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,215485 ms	0,680166 ms	2,595403 ms
5 × 5	0,455487 ms	1,603956 ms	6,374675 ms
7 × 7	0,833453 ms	2,954436 ms	11,845298 ms
9 × 9	1,287418 ms	4,720388 ms	19,044971 ms

Tabelle E.11: Kernelausführungszeiten für Grauwertdaten: Eingangsdaten im Texture Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,446371 ms	1,230669 ms	4,252139 ms
5 × 5	0,669559 ms	2,13786 ms	8,036151 ms
7 × 7	1,07369 ms	3,496344 ms	13,505063 ms
9 × 9	1,538158 ms	5,276934 ms	20,703491 ms

Tabelle E.12: Ausführungszeiten mit Datentransfer für Grauwertdaten: Eingangsdaten im Texture Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,173984 ms	0,533316 ms	1,967506 ms
5 × 5	0,376937 ms	1,172134 ms	4,434342 ms
7 × 7	0,491753 ms	1,807275 ms	7,062792 ms
9 × 9	0,70784 ms	2,669763 ms	10,465441 ms

Tabelle E.13: Kernelausführungszeiten für Grauwertdaten: Eingangsdaten im Texture Memory mit Puffer im Shared Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,42883 ms	1,103491 ms	3,577497 ms
5 × 5	0,507138 ms	1,619246 ms	6,04837 ms
7 × 7	0,84094 ms	2,377465 ms	8,69237 ms
9 × 9	0,962984 ms	3,204422 ms	12,15849 ms

Tabelle E.14: Ausführungszeiten mit Datentransfer für Grauwertdaten: Eingangsdaten im Texture Memory mit Puffer im Shared Memory

E.10 GPU-Ausführungszeiten mit RGB-Bilddaten

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,458315 ms	1,633484 ms	6,665187 ms
5 × 5	1,178379 ms	4,338151 ms	17,849934 ms
7 × 7	2,216887 ms	8,399857 ms	34,643532 ms
9 × 9	3,601113 ms	13,792097 ms	54,960117 ms

Tabelle E.15: Kernelausführungszeiten für RGB-Daten: Eingangsdaten im Global Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,920972 ms	2,932808 ms	11,301513 ms
5 × 5	1,603161 ms	5,617679 ms	22,506418 ms
7 × 7	2,637844 ms	9,673422 ms	39,298424 ms
9 × 9	4,03698 ms	15,082446 ms	59,621967 ms

Tabelle E.16: Ausführungszeiten mit Datentransfer für RGB-Daten: Eingangsdaten im Global Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,291393 ms	0,979932 ms	3,734288 ms
5 × 5	0,496802 ms	1,771553 ms	6,87825 ms
7 × 7	1,121789 ms	4,134589 ms	16,278711 ms
9 × 9	1,651351 ms	6,387987 ms	25,318043 ms

Tabelle E.17: Kernelausführungszeiten für RGB-Daten: Eingangsdaten im Global Memory mit Puffer im Shared Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,673635 ms	2,198858 ms	8,319579 ms
5 × 5	0,901469 ms	3,082663 ms	11,511715 ms
7 × 7	1,529055 ms	5,424075 ms	20,948975 ms
9 × 9	2,08621 ms	7,686144 ms	29,985376 ms

Tabelle E.18: Ausführungszeiten mit Datentransfer für RGB-Daten: Eingangsdaten im Global Memory mit Puffer im Shared Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,297817 ms	1,032458 ms	4,049915 ms
5 × 5	0,667571 ms	2,431178 ms	9,678102 ms
7 × 7	1,210406 ms	4,501299 ms	18,05353 ms
9 × 9	1,921188 ms	7,235723 ms	29,155251 ms

Tabelle E.19: Kernelausführungszeiten für RGB-Daten: Eingangsdaten im Texture Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,751064 ms	2,329225 ms	8,712332 ms
5 × 5	1,108504 ms	3,745884 ms	14,326803 ms
7 × 7	1,660874 ms	5,80833 ms	22,705906 ms
9 × 9	2,357799 ms	8,53721 ms	33,79504 ms

Tabelle E.20: Ausführungszeiten mit Datentransfer für RGB-Daten: Eingangsdaten im Texture Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,288675 ms	0,968276 ms	3,69582 ms
5 × 5	0,49938 ms	1,748624 ms	6,806279 ms
7 × 7	1,066468 ms	4,123463 ms	16,251198 ms
9 × 9	1,624793 ms	6,372109 ms	25,31011 ms

Tabelle E.21: Kernelausführungszeiten für RGB-Daten: Eingangsdaten im Texture Memory mit Puffer im Shared Memory

Faltungsmatrix	Bildgröße: 512 × 512	Bildgröße: 1024 × 1024	Bildgröße:2048 × 2048
3 × 3	0,662512 ms	2,18425 ms	8,269606 ms
5 × 5	0,913894 ms	3,029789 ms	11,453706 ms
7 × 7	1,627166 ms	5,392085 ms	20,865721 ms
9 × 9	2,07837 ms	7,67067 ms	29,956795 ms

Tabelle E.22: Ausführungszeiten mit Datentransfer für RGB-Daten: Eingangsdaten im Texture Memory mit Puffer im Shared Memory

E.11 CPU-Ausführungszeiten mit Grauwert-und RGB-Bilddaten

	Bildgrößen für Grauwertbilddaten			Bildgrößen für RGB-Bilddaten		
Faltungsmatrix	512	1024	2048	512	1024	2048
3 × 3	15,58 ms	55,95 ms	213,89 ms	23,92 ms	89,28 ms	338,06 ms
5 × 5	34,67 ms	124,34 ms	447,01 ms	51,06 ms	203,81 ms	809,74 ms
7 × 7	62,43 ms	235,27 ms	912,10 ms	93,20 ms	371,56 ms	1480,06 ms
9 × 9	89,62 ms	350,31 ms	1391,95 ms	145,27 ms	583,25 ms	2327,26 ms

Tabelle E.23: Ausführungszeiten auf der CPU

E.12 Quellcode: Gesamtes Projekt Faltung

Dieser Anhang ist in elektronischer Form auf der CD abgelegt und beim Prüfer Prof. Dr. Ing. Hans Peter Kölzer einzusehen.

Anhang F

Zweidimensionale Fourier-Transformation

F.1 Quellcode: Zweidimensionale Fourier-Transformation

```
////////// File: cuda_code.cu //////////////

// Kernel
__host__ cudaError_t cl_run_fft(BYTE* d_idata,
                        BYTE* d_odata,
                        imagedata image,
                        cudaStream_t stream)
{
   ////////////////////////////////////////////////////
   /////// Indata:      Real:  M = width  , N = height
   ///////
   /////// Frequencies: Complex: Mf = (M/2)+1 = (width/2)+1 ,  Nf = height
   ///////
   /////// Outdata:     Real: M = width  , N = height
   ///////     The outdata has to be normalize with (1/(M*N))
   ////////////////////////////////////////////////////

   cudaMemcpyAsync(d_odata,d_idata,image.datasize,cudaMemcpyDeviceToDevice,
       stream);
   // RUN KERNEL
   cudaEvent_t start, stop;
   float time;
   cl_start_timer(&start,&stop,stream);

   int M = image.width;
   int N = image.height;
   int Mf = (M/2+1);
   int Nf = N;
```

```
float norm =1.f/(float(M*N)) ;

// Threadblock
dim3 dimBlock(THREADBLOCK,THREADBLOCK);
dim3 dimGrid(((image.width + dimBlock.x - 1 ) / dimBlock.x ),((image.
    height + dimBlock.y -1) / dimBlock.y ));

/////// THE FFT RealToComplex <-> ComplexToReal /////
dim3 dimCpxGrid((( Mf + dimBlock.x - 1 ) / dimBlock.x ),((Nf + dimBlock.y
     -1) / dimBlock.y ));
// FFT DATA - COMPLEX-Type
cufftComplex *i_cpxdata; // frequencies
cudaMalloc((void**) &i_cpxdata,(Mf*Nf)*sizeof(cufftComplex));
////////////////////////////////////////////////////////

////////////// Transform BYTE-Indata into float data /////
float* i_fdata,float* o_fdata;
cudaMalloc((void**) &i_fdata,((Mf*2)*Nf)*sizeof(float)); /// Indata float
cudaMalloc((void**) &o_fdata,((Mf*2)*Nf)*sizeof(float)); /// Outdata
    float
cuda_kernel_byteToFloatData<<< dimGrid,dimBlock,0,stream >>>(d_idata,
    i_fdata,image);
////////////////////////////////////////////////////////

////////////// Create Cufft Plans ////////////////////////
// Create FFTPlan RealToComplex - FFT
cufftHandle plan_forward;
// Number of Rows = image.height = N -- Number of Cols = image.width = M
    //
cufftPlan2d(&plan_forward, N, M , CUFFT_R2C);
cufftSetStream(plan_forward, stream);

// ComplexToReal - FFT
cufftHandle plan_backward;
// Number of Rows = image.height = N -- Number of Cols = image.width = M
    //
cufftPlan2d(&plan_backward, N, M, CUFFT_C2R);
cufftSetStream(plan_backward, stream);
////////////////////////////////////////////////////////

////////////////////////////////////////////////
/////// Invoke RealToComplex-Transformation ///////
cufftExecR2C(plan_forward, i_fdata, i_cpxdata);
////////////////// Invoke Kernel ///////////////
cuda_kernel_blur<<< dimCpxGrid,dimBlock,0,stream >>>(i_cpxdata,i_cpxdata,
    image,1.f);
////////////////////////////////////////////////
/////// Invoke ComplexToReal-Transformation ///////
cufftExecC2R(plan_backward,i_cpxdata, o_fdata);
```

```
    ////////////////////////////////////////////////

    ////////////// Transform float-Indata into byte data //////
    cuda_kernel_floatToByteData<<< dimGrid,dimBlock,0,stream >>>(o_fdata,
        d_odata,image,norm);
    ////////////////////////////////////////////////////////////

    // Synch
    cudaThreadSynchronize();
    // stop timer //
    cl_stop_timer(&start,&stop,&time,stream);
    printf("Processing time kernel: %f ms\n",time);

    ////////////////////////////////////////////////////////
    // Destroy Cufft-Plans
    cufftDestroy(plan_forward);
    cufftDestroy(plan_backward);

    // Free Data //
    cudaFree(i_fdata);
    cudaFree(o_fdata);
    cudaFree(i_cpxdata);

    return cudaSuccess;
}
```

F.2 Quellcode: Gesamtes Projekt 2D-FFT

Dieser Anhang ist in elektronischer Form auf der CD abgelegt und beim Prüfer Prof. Dr. Ing. Hans Peter Kölzer einzusehen.

Versicherung über die Selbständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Ort, Datum	Unterschrift