

Diplomarbeit

**Effizientes Lösen linearer
Gleichungssysteme über $\text{GF}(2)$ mit
GPUs**

von

Denise Demirel

27. September 2010

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Theoretische Informatik
Hochschulstraße 10
64289 Darmstadt

Betreuer: Dr. rer. nat. Stanislav Bulygin

Prüfer: Prof. Dr. rer. nat. Dr. h. c. Johannes Buchmann

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 26. September 2010 Denise Demirel

Danksagung

In den letzten Monaten, die ich mich Vollzeit meiner Diplomarbeit gewidmet habe, hatte ich das Glück von vielen Menschen dabei unterstützt worden zu sein.

Zuerst möchte ich mich daher bei meinen vielen Nachbarn und Mitbewohnern bedanken, die sich alle bereit erklärt haben Teile meiner Arbeit zu lesen, zu korrigieren und zu kommentieren: Jan, Julia, Sebastian, Tati, Lena, Caspar, Nika, Jörg, Stephan, Lars, Sven und Attila. Auerdem gilt meinen Mitbewohnern Tobi, Jörg, Stephan, Nico und Miguel ein besonderer Dank, die in stressigen Zeiten mein schmutziges Geschirr weggeräumt oder toleriert und für Aufmunterung gesorgt haben.

Natürlich danke ich auch meinen Eltern, die mir nicht nur mein Studium finanziert und damit ermöglicht haben, sondern auch immer eine große seelisch Stütze waren.

Während meiner Diplomarbeit hatte ich die Möglichkeit 6 Wochen an der National Taiwan University in Taipei zu studieren. Das war für mich eine sehr bereichernde Erfahrung, die Prof. J. Buchmann, mein Betreuer Stanislav sowie Chen-Mou und Bo-Yin möglich gemacht haben.

Auerdem gilt mein Dank Mohamed und Sven für ihre fachliche Beratung und Michael und Marc Stöttinger dafür, dass sie mir Zugang zu den Grafikkarten gewährt haben, mit denen sie arbeiten.

Zum Schluß bedanke ich mich nochmal bei Stanislav für die Betreuung meiner Arbeit und bei allen anderen, die ich hier vergessen habe.

Denise Demirel, September 2010

Inhaltsverzeichnis

1	Einleitung	1
2	Parallele Programmierung mit CUDA	5
2.1	Parallele Architekturen	5
2.2	NVIDIA	7
2.3	Parallele Programmierung	8
2.3.1	Unterschiede zwischen Host und Device	8
2.3.2	Threads, Blöcke und Grids	9
2.3.3	Streams	11
2.4	CUDA	11
2.4.1	Die CPU Komponente der Laufzeitbibliothek	12
2.4.2	Erweiterungen der C-Programmiersprache	13
2.5	Die Architektur von NVIDIA Grafikkarten	16
2.5.1	Die <i>Compute Capability</i>	17
2.5.2	Die Architektur von Grafikkarten mit einer <i>Compute Capability</i> von 1.x	17
2.5.3	Die Architektur von Grafikkarten mit einer <i>Compute Capability</i> von 2.0	18
2.5.4	Der Grafikkartenspeicher	18
2.5.5	Der CPU-Speicher	28
2.6	Optimiertes Programmieren mit CUDA	29
2.6.1	Optimierung des Befehlsdurchsatzes	30
2.6.2	Optimierung von Speicherzugriffen	33
2.6.3	Optimierung des Datendurchsatzes	34
3	Die Umsetzung des M4RI in CUDA	37
3.1	Die “Method of Four Russians“ Inversion	38
3.2	Die Umsetzung von M4RI mit CUDA	42

3.2.1	Beschreibung der verwendeten Grafikkarten	42
3.2.2	Die verwendete API	43
3.2.3	Der umgesetzte Algorithmus	43
3.2.4	Übertragung der Daten	46
3.2.5	Speicherverwaltung	46
3.2.6	Parameter	50
3.2.7	Optimierungen	55
3.2.8	Der Programmaufbau	58
3.2.9	Analyse	63
3.2.10	Zeitvergleiche mit Berechnungen auf der CPU	66
3.3	Multiple GPUs	67
3.3.1	Theoretische Umsetzung	67
3.3.2	Umsetzung mit CUDA	70
4	Fazit und Ausblick	73
A	Programmcode	77
B	Technische Spezifikationen	87

Kapitel 1

Einleitung

Das Lösen von linearen Gleichungssystemen über $GF(2)$ ist Bestandteil vieler Methoden der Kryptoanalyse. Beispiele sind Attacken auf Stromverschlüsselungen wie der XL (eXtended Linearization) Algorithmus oder Matrix-basierte Gröbnerbasis Algorithmen wie F4 und F5. Das Lösen von linearen Gleichungssystemen ist auch Teil der Kryptoanalyse asymmetrischer Kryptosysteme wie HFE und SFLASH. Die Effizienz dieser Algorithmen ist demnach davon abhängig, wie schnell diese Systeme gelöst werden können.

Die Erfahrung zeigt, dass viele Probleme unter Verwendung von Parallelrechnern schneller gelöst werden können als mittels eines Rechners mit nur einer CPU. Ein Beispiel hierfür ist die Multiplikation zweier Matrizen. Diese Operation ist sehr gut parallelisierbar, da die Daten zeilenweise auf die verfügbaren Prozessoren verteilt werden können, welche dann ihre Teilkomponenten des Lösungsvektors ohne weitere Kommunikation berechnen.

Der Trend bei CPUs geht schon seit einigen Jahren in Richtung der Multicore-Prozessoren, bei denen sich mehrere Hauptprozessoren auf einem Chip befinden. Diese ermöglichen vor allem eine vervielfachte Rechenleistung. Wie stark diese jedoch gesteigert werden kann hängt davon ab, ob und in welchem Maße sich das Programm parallelisieren und damit auf verschiedene Ausführungseinheiten verteilen lässt. Es bietet sich also an einen Algorithmus in mehrere voneinander unabhängig berechenbare Module zu unterteilen. Wird dieser Weg verfolgt, dann sollte auch eine Umsetzung mit Grafikkarten in Betracht gezogen werden, da diese bereits über eine parallele Architektur verfügen, weit verbreitet und vergleichsweise preiswert sind.

Grafikkarten für parallele Berechnungen zu verwenden ist sinnvoll, war aber lange Zeit schwer umsetzbar, weil deren Schnittstelle auf graphische Anwendungen ausgelegt ist. Durch zunehmenden Einsatz paralleler Architekturen, bot es sich an mittels einer Programmiersprache GPUs auch für andere Anwendungsgebiete nutzbar zu machen. Seit 2007 stellt der Grafikkartenhersteller NVIDIA eine parallele Berechnungsarchitektur namens "Compute Unified Device Architecture", kurz CUDA, bereit, welche es ermöglicht Grafikprozessoren als Co-Prozessoren zu verwenden. Teile der Programme, die parallel berechnet werden können, kann der Anwender an eine oder mehrere GPUs senden. Zu den Vorteilen gehört die frei erhältliche API und die an C erinnernde Syntax. Außerdem gibt es Wrapper für die Programmiersprachen Python, Java, Fortran und .NET sowie Anbindungen an Matlab. Seit 2009 werden Alternativen in Form von Streamprozessoren auch von dem Chiphersteller AMD angeboten und sind unter dem Namen FireStream bekannt. NVIDIA stellt jedoch länger parallele Berechnungsarchitekturen zur Verfügung wodurch ihre Systeme ausgereifter sind.

Eine weitere Alternative zu den parallelen Berechnungsarchitekturen CUDA und FireStream ist OpenCL. Diese Programmierplattform hat den Vorteil, dass die erzeugten Programme plattformunabhängig sind. Dadurch hat OpenCL jedoch den Nachteil, dass die Laufzeit in vielen Fällen länger als die der CUDA-Anwendungen ist.

Die vorliegende Diplomarbeit beschreibt eingehend den Aufbau von NVIDIA Grafikkarten und erforscht die Möglichkeit, mit Hilfe der von NVIDIA eingeführten parallelen Berechnungsarchitektur CUDA, diese als Co-Prozessoren beim Lösen eines linearen Gleichungssystems über $GF(2)$ einzusetzen.

Um Berechnungen zu beschleunigen wird die Verwendung von Grafikkarten immer mehr zu einer gängigen Praxis. Da diese normalerweise zum Rendern von Grafiken verwendet werden, handelt es sich bei GPUs um mächtige arithmetische Maschinen. Dort arbeiten Tausende von Threads zeitgleich auf dem selben Adressraum. Deshalb eignen sie sich besonders für Berechnungen, die von einer parallelen Bearbeitung positiv beeinflusst werden können.

Da für die Verwendung von Grafikkarten die Daten erst an diese übertragen werden müssen, ist es wichtig im Vorfeld zu prüfen, ob die Komplexität der Berechnungen die Latenz, welche durch den Datentransfer entsteht, ausgleichen kann. Ein wichtiger Wert ist in diesem Zusammenhang die Menge der ausgeführten Operationen pro Datenelement. Bei Gleichungssystemen hängt die Anzahl der Rechenschritte auf einem Feld von dessen Position in der $n \times m$ - Matrix ab und liegt zwischen einer und m Operationen. Die genaue Dimension der aus kryptoanalytischen Methoden hervorgehenden linearen Gleichungssysteme ist unklar, da diese z.B. bei Attacken auf Stromverschlüsselungen von der Umwandlung des polynominalen in das lineare System abhängen. Dennoch kann davon ausgegangen werden, dass die Matrizen sehr groß sind und die Anzahl ihrer Elemente nicht selten sechsstellig ist.

Im ersten Schritt wurde das Gaußsche Eliminationsverfahren umgesetzt, da es sich dabei um einen einfachen Algorithmus handelt, der schnell parallelisiert werden kann. Des Weiteren ist er Teil eines anderen, schnelleren Algorithmus, der "Method of Four Russians" Inversion. Dieser war das eigentliche Ziel der Arbeit.

In Kapitel 2 soll ein grober Überblick über CUDA sowie Grafikkarten von NVIDIA gegeben und die geschichtliche Entwicklung dargestellt werden. Für eine optimale Programmierung ist es unerlässlich die genaue Architektur und Funktionsweise der GPUs zu kennen. Beispielsweise kann die volle Bandbreite bei einem Zugriff auf den globalen Speicher nur erreicht werden, wenn das Programm bestimmte Rahmenbedingungen einhält. Neben diesem Off-Chip-Speicher wird auch ein On-Chip-Speicher angeboten, welcher als Cache fungiert und von dem Anwender selbst verwaltet wird. Mit diesem können Verzögerungszeiten bei einem mehrmaligen Zugriff auf die selben Daten im RAM vermieden und so eine bessere Performanz erreicht werden.

Kapitel 3 beschreibt die "Method of Four Russians" Inversion und erläutert die genaue Umsetzung. Wird ein Algorithmus, welcher normalerweise auf der CPU arbeitet, auf der GPU ausgeführt, so ist es sinnvoll diesen zuvor anzupassen. Zum einen sollte das Verfahren so modifiziert werden, dass es aus verschiedenen autonomen Blöcken besteht, die parallel bearbeitet werden können, und zum anderen dürfen zwischen diesen Ausführungseinheiten keine Datenabhängigkeiten vorliegen, da es sonst zu falschen Ergebnissen

kommen kann.

Neben der Dokumentation des implementierten Algorithmus enthält die Arbeit auch Testergebnisse von verschiedenen Grafikkarten und eine Aufzeichnung der verwendeten Einstellungen. Als lineare Gleichungssysteme wurden von CUDA erzeugte Zufallsmatrizen verwendet.

Anschließend konnten die erreichten Zeiten mit denen von CPU-Berechnungen verglichen werden. Dabei wurden die Resultate des “SAGE/M4RI-Teams“ verwendet, welches sich seit der Veröffentlichung des Algorithmus durch Gregory Bard [2] mit diesem beschäftigt und in regelmäßigen Abständen seine aktuellen Ergebnisse veröffentlicht. Anhand dieser Bestzeiten zeigte sich, dass die Lösung linearer Gleichungssysteme auf Grafikkarten schneller durchführbar ist.

Zum Abschluss der Arbeit wird in Kapitel 4 ein Fazit gezogen und sinnvolle, zukünftige Erweiterungen des Programms vorgestellt.

Kapitel 2

Parallele Programmierung mit CUDA

Kapitel 2 enthält alle Informationen, die zum Verständnis des Programmcodes benötigt werden. Neben der bisherigen Entwicklung im Bezug auf parallele Architekturen und Daten zum Grafikkartenhersteller NVIDIA in den Unterkapiteln 2.1 und 2.2 enthält dieses Kapitel auch Informationen über die benutzte parallele Berechnungsarchitektur CUDA, um einen Einstieg in die Verwendung von Grafikkarten als Co-Prozessoren zu geben. Kapitel 2.3 und 2.4 beschreiben aus welchen Komponenten die APIs bestehen und gibt eine Einführung in die Programmierung von parallelen Anwendungen. Um die gegebenen Hardwareeigenschaften der Grafikkarte ausnutzen zu können wird in Kapitel 2.5 die Architektur erläutert und anhand dieser in Kapitel 2.6 erklärt, wie die Performanz von CUDA-Programmen maximiert werden kann.

2.1 Parallele Architekturen

Bei der parallelen Datenverarbeitung handelt es sich um eine Verarbeitung von Daten, bei der Ergebnisse gleichzeitig von mehreren Prozessoren berechnet werden. Massiv-Parallele Computer besitzen einige zehn bis einige Tausend unabhängige Ausführungseinheiten, die zeitgleich Operationen durchführen können und so ermöglichen regelmäßig aufgebaute Datensätze effizient zu bearbeiten. Dabei verfügt jeder Hauptprozessor über seinen eigenen Arbeitsspeicher (Vergleiche [8]). Computer, die eine parallele Berechnung

ermöglichen werden als Parallelrechner bezeichnet. Sie verteilen die Aufgaben auf verschiedene Prozesse, um die Arbeitsgeschwindigkeit zu verbessern. Da dies auf unterschiedliche Weise realisiert werden kann umfasst die Klasse der Parallelrechner ein sehr weites Feld von parallelen Architekturen (Siehe auch [9]).

Es gibt mehrere Möglichkeiten einzelne Arten von Parallelrechnern zu differenzieren. Unterscheiden kann man diese beispielsweise nach ihrem Programmablauf. Führen alle Prozessoren denselben Befehl auf mehreren Datensätzen gleichzeitig aus, so spricht man von einer Datenparallelität bzw. von einem *Single Instruktion Multiple Data*- kurz *SIMD*- Rechner. Bearbeitet jeder Prozessor sein eigenes individuelles Programm und werden somit verschiedene Operationen zeitgleich durchgeführt, so spricht man von einer Kontrollparallelität bzw. von einem *Multiple Instruction Stream, Multiple Data Stream*- kurz *MIMD*- Rechner.

Eine Softwarelösung für Parallelrechner ist die *Parallele Virtuelle Maschine*. Dieses an der Universität Tennessee entwickelte Public-Domain-Softwarepaket vernetzt PCs so, dass sie wie ein großer Parallelrechner verwendet werden können.

Zu den physikalischen Strukturen gehören homogene Multiprozessorsysteme mit gleichartigen Prozessoren, eng gekoppelte Multiprozessorsysteme, welche über ein schnelles internes Netzwerk oder einen gemeinsamen Speicher verfügen und lose gekoppelte Multiprozessorsysteme bei denen es sich um Rechner handelt, die über eine größere Distanz miteinander verbunden sind und somit eine geringe Bandbreite und eine hohe Latenzzeit aufweisen.

Des Weiteren kann man physikalische Strukturen nach ihrer Speicherarchitektur unterscheiden. *Symmetrische Multiprozessorsysteme* kurz *SMP* kennzeichnen sich dadurch, dass sie über einen gemeinsamen globalen physikalischen Adressraum verfügen. Beispiele hierfür sind die *IBM POWER Chips*. Eine andere Gruppe sind die *Distributed/Virtual Shared Memory Systeme*. Hier verfügt jeder Prozessor über einen eigenen lokalen Speicherbereich, die alle zusammen einen gemeinsamen logischen Adressraum bilden. Ein Beispiel für *Distributed/Virtual Shared Memory Systeme* ist die *SGI Altix 3700*. Die *Blue Gene/L JUBL* von IBM gehört zu der Kategorie der Cluster. Hierbei handelt es sich um vernetzte *SMP*-Knoten oder einzelne Rechner. Sie sind nachrichtengekoppelt und haben eine einfache Struktur. Genauere Informationen zu den einzelnen Arten von Parallelrechnern kann man [7] und [8] entnehmen.

Seit ca. 2000 werden auch Grafikprozessoren mit ihren Rechenleistungen für parallelisierbare Operationen verwendet. Dabei handelt es sich oft um Berechnungen, die traditionell auf der CPU durchgeführt werden. Das Lösen von allgemeinen Aufgaben mit Grafikkarten wird als *General Purpose Computation on Graphics Processing Unit* oder kurz GPGPU bezeichnet (Siehe [17]). Grafikkarten verfügen über eine SIMD-Architektur. Neuere GPUs von NVIDIA ermöglichen die zeitgleiche Ausführung mehrerer unterschiedlicher Programme auf einer Grafikkarte wodurch auch die MIMD Architektur umgesetzt ist. Das Multiprozessorsystem ist homogen und alle Prozessoren befinden sich auf der selben Karte. Jeder verfügt über einen eigenen Adressraum und hat Zugriff auf den globalen Speicherbereich, welchen sich alle Prozessoren teilen.

2.2 NVIDIA

NVIDIA wurde 1993 von Jen-Hsun Huang, Curtis Priem und Chris Malachowsky gegründet und gehört heute zu einer der größten Firmen für die Entwicklung von Grafikprozessoren und Chipsätzen für Personal Computer und Spielkonsolen (Siehe auch [11]).

Im Februar 2007 wurde die *Compute Unified Device Architecture*, kurz CUDA entwickelt. Diese API ermöglicht es Grafikprozessoren als Co-Prozessoren für die CPU einzusetzen um parallelisierbare Berechnungen zu beschleunigen. Bis dahin war die Schnittstelle nur auf graphische Anwendungen ausgelegt, was eine Verwendung in anderen Anwendungsbereichen erschwerte. Dennoch wurde der Einsatz von GPUs für parallele Berechnungen immer mehr zu einer gängigen Praxis. Grafikkarten verfügen über eine parallele Many-Core Architektur, bei der jeder Grafikern tausende von Threads gleichzeitig ausführen kann (Siehe auch Kapitel 2.5). Mit der Entwicklung von CUDA konnte NVIDIA den Markt für seine Produkte erweitern. Anwendbar ist die API auf viele Karten der GeForce und Quadro Serie, sowie für alle Tesla Karten. Tesla Karten sind Streamprozessoren, welche für den Einsatz als Co-Prozessor entwickelt wurden und nicht über Monitoranschlüsse verfügen (Siehe [13])

CUDA ermöglicht mit C for CUDA einen leichten Einstieg in die Programm-entwicklung, da es sich um die relativ bekannte Programmiersprache C mit

NVIDIA Erweiterungen handelt.

Außerdem existieren Wrapper für die Programmiersprachen Python, Java, Fortran und .NET, es wird DirectX sowie OpenGL und PhysX unterstützt und es ist möglich Anwendungen an Matlab anzubinden. Seit 2009 bietet auch der Chiphersteller AMD eine Alternative unter dem Namen FireStream an.

Jede CUDA-fähige Grafikkarte von NVIDIA verfügt über eine Revisionsnummer, bestehend aus 2 Zahlen, der so genannten *Compute Capability* (Siehe auch Kapitel 2.5.1). Devices, deren erste Ziffer übereinstimmen, verfügen über die selbe Architektur. Die 2010 auf dem Markt erschienenen Fermi-Grafikchips sind mit der Zahl 2 gekennzeichnet, alle älteren mit der 1. Die zweite Ziffer beschreibt die stufenweise Optimierung der Bauweise und das Hinzufügen von neuen Features (Siehe [4]). Eine Auflistung aller Grafikkarten mit ihren Revisionsnummern und den dazugehörigen Features findet man im Anhang Kapitel B.

2.3 Parallele Programmierung

Bei der Programmierung mit CUDA wird der Code auf zwei verschiedenen Plattformen ausgeführt. Das Host System besteht aus einer oder mehreren CPUs, welche mit einem oder mehreren Devices arbeiten, bei denen es sich um CUDA-fähige NVIDIA Grafikkarten handelt. Der Anwender muss dafür das Problem in Teilprobleme zerlegen. Die sequentiellen Berechnungen bearbeitet die CPU, während die Parallelen als so genannte Kernel auf die GPU geladen werden. Das Device verwendet eine Single Instruction Multiple Thread Architektur, die es ermöglicht eine Anweisung zeitgleich von mehreren Threads auf verschiedenen Datensätzen auszuführen. Die Grafikkarte kommt vor allem dann als Co-Prozessor zum Einsatz, wenn große Mengen von Daten mit den selben Operationen bearbeitet werden sollen.

2.3.1 Unterschiede zwischen Host und Device

CPUs können nur mit einer limitierten Anzahl von parallelen Threads arbeiten. Bei einem Server mit 4 Quad-Core Prozessoren sind es somit 16 parallele Prozesse und 32, wenn Hyper-Threading unterstützt wird. Bei letzterem werden die Lücken in der Pipeline mit Befehlen eines anderen Threads aufgefüllt, um die Rechenwerke des Prozessors besser auszulasten (Siehe auch [12]).

Bei einer Grafikkarte werden immer 32 Threads zusammengefasst und bilden die kleinste ausführbare Einheit namens Warp. Je nach GPU können auf einem Multiprozessor bis zu 32 dieser Warps, also insgesamt 1024 Prozesse, aktiv sein. Devices, wie beispielsweise die verwendete GeForce GTX 295, verfügen über 30 Multiprozessoren, was zu mehr als 30000 Threads führt. Im Gegensatz zur CPU ist die Erzeugung eines Threads auf der Grafikkarte kein deutlicher Mehraufwand, da es sich um leichtgewichtige Objekte handelt. Es muss sogar eine Mindestmenge verwendet werden, damit die GPU effizient arbeiten kann. Die Warps werden vom Warp Scheduler verwaltet (Siehe auch 2.5.2 und 2.5.3). Wenn ein Warp warten muss, weil dieser beispielsweise Daten aus dem globalen Speicher angefordert hat, dann beginnt der Prozessor solange die Arbeit an einem anderen, dessen Prozesse bereit sind ihre Anweisungen abzuarbeiten. Auf diese Weise ist es möglich die Verzögerungszeiten, welche beim Zugriff auf den Speicher entstehen, zu verbergen (Siehe auch Kapitel 2.6). Jeder Thread verfügt über seine eigenen Register, was es nicht, wie bei der CPU, nötig macht den Kontext auszutauschen. Die Ressourcen werden ihm so lange zur Verfügung gestellt, bis seine Arbeit beendet ist. Ein weiterer Unterschied zwischen Host und Device besteht im Zugriff auf den Speicher. Bei der CPU werden viele Transistoren für Steuerungsaufgaben und Caching verwendet, während auf der GPU die Verzögerungszeit, welche beim Zugriff auf den Speicher entsteht, durch Berechnungen ausgeglichen wird, deren Durchführung in der Wartezeit möglich ist. Des Weiteren unterscheiden sich Host und Device in ihrer Architektur. Genauere Informationen darüber werden in Kapitel 2.5 gegeben.

2.3.2 Threads, Blöcke und Grids

Den parallelen Teil der Berechnungen bearbeiten so genannten Kernel. Diese werden n -mal von n verschiedenen Threads ausgeführt. Jeder Thread befindet sich in genau einem Block und jeder Block enthält mehrere Threads. Ein Block ist Element eines Grids, welches wiederum über mehrere Blöcke verfügt. Ein Kernel wird von allen Threads eines Grids bearbeitet und es ist möglich die Größe sowie die Dimension des Grids und der Blöcke beim Kernel-Aufruf zu setzen. Jeder Multiprozessor führt anschließend so viele Blöcke wie möglich gleichzeitig aus. Diese Hierarchie soll die Organisation des parallelen Arbeitens auf Datenelementen wie Vektoren, Matrizen oder Räumen vereinfachen. Daher ist es möglich über eine Variable in jedem Thread bzw. Block

seine eindeutige ein-, zwei oder dreidimensionale ID und damit seine Position im Block bzw. Grid abzufragen. Die beschriebene Unterteilung ist nicht

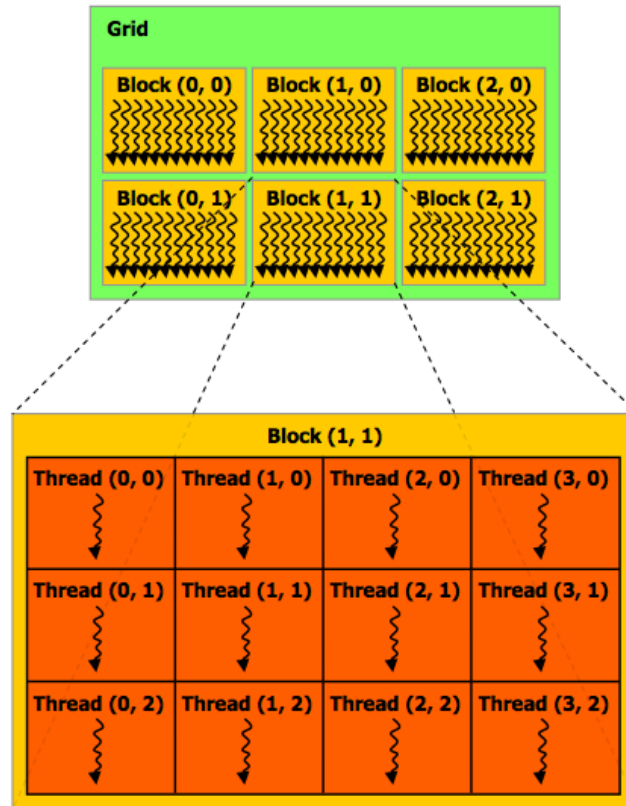


Abbildung 2.1: Grid mit Threadblöcken [4].

nur durch die zugrunde liegende Datenstruktur sinnvoll, sondern auch durch die Tatsache, dass alle Threads eines Blocks in den selben Prozessor geladen werden und sich dort die bereit gestellte Menge an Ressourcen wie beispielsweise RAM und Register teilen müssen.

Alle Prozesse eines Warps starten an der selben Programmadresse, haben aber einen eigenen Adresszähler und Register, wodurch sie unabhängig voneinander ablaufen und verzweigen können. Da immer nur ein gemeinsamer Befehl parallel verarbeitet werden kann, wird die beste Performanz erreicht, wenn die 32 Threads eines Warps den selben Programmablauf haben. Sobald eine Verzweigung enthalten ist, die von Werten der jeweiligen Prozesse

abhängt, kann es sogar dazu kommen, dass dieser Teil seriell bearbeitet werden muss.

Alle Threads eines Blocks befinden sich auf dem selben Prozessor und können Informationen und Daten über einen Shared Memory austauschen. Außerdem gibt es Befehle um sie zu synchronisieren. Diese funktionieren wie Barrieren. Es darf erst mit der Ausführung fortgefahren werden, wenn alle Threads des Blocks eine bestimmte Stelle im Code erreicht haben. Threadblöcke müssen voneinander unabhängig sein, da sie in einer zufälligen Reihenfolge auf alle vorhandenen Prozessorkerne verteilt und bearbeitet werden. Threads unterschiedlicher Blöcke können daher nur über den globalen Speicher kommunizieren (Vergleiche [4]).

2.3.3 Streams

Ein Stream ist eine Sequenz von Operationen, welche in einer vorgegebenen Reihenfolge auf dem Device ausgeführt werden. Anweisungen in verschiedenen Streams können ineinander überlaufen und in manchen Fällen auch überlappen, da sie ihre Befehle völlig unabhängig voneinander bearbeiten. Mit ihnen wird der zeitgleiche Zugriff verschiedener Prozesse auf die Grafikkarte gemanaged. Es ist beispielsweise möglich, dass Daten mit Hilfe von Streams stückweise kopiert und von verschiedenen Kernel bearbeitet werden. Dadurch können Berechnungen mit dem Datentransfer überlappen.

Die Ausführung eines Kernels wird als ein Stream mit mehreren Tasks betrachtet. Im Gegensatz zu allen anderen arbeitet der Defaultstream synchron. Das heißt, dass eine Operation erst beginnen kann, nachdem alle Aufrufe in allen anderen Streams beendet wurden und kein weiterer anfangen kann zu arbeiten, bevor der Defaultstream seine Arbeit nicht abgeschlossen hat (Vergleiche [4]).

2.4 CUDA

Die parallele Berechnungsarchitektur ermöglicht eine Unterstützung der CPU durch die GPU, wodurch die Performanz vieler Algorithmen verbessern werden kann. Eine vollständige Beschreibung aller durch CUDA gegebenen Programmierungsmöglichkeiten soll hier nicht gegeben werden. Nur die für das Projekt interessanten Methoden werden erläutert. Weitere Informationen kann man der offiziellen CUDA-Dokumentation entnehmen [4].

CUDA bietet eine Programmierschnittstelle für Anwender, die es ihnen ermöglicht die Grafikkarte als Co-Prozessor zu verwenden. Im wesentlichen besteht die API aus speziellen Laufzeit-Bibliotheken und Erweiterungen der Programmiersprache C. Die Bibliotheken unterteilen sich in drei Bereiche: Zum Einen eine CPU-Komponente, welche die GPU steuert und den Zugriff auf diese ermöglicht, eine GPU-Komponente, die der Grafikkarte spezielle Funktionen bereit stellt und eine Komponente, welche von beiden genutzt wird.

2.4.1 Die CPU Komponente der Laufzeitbibliothek

Die Host Runtime Komponente der CUDA Softwareumgebung kann nur von Hostfunktionen verwendet werden und unterstützt Device Management, Kontext Management, Speichermanagement, Code Modul Management, Ausführungskontrollen, Texture Reference Management und Interoperabilität mit OpenGL und Direct3D. Sie enthält zwei verschiedene Programmierschnittstellen. Die Low-Level API namens CUDA Driver API und eine High-Level API genannt C Runtime for CUDA , welche auf der ersten implementiert wurde. Obwohl es inzwischen möglich ist unter Einhaltung bestimmter Rahmenbedingungen in einem Programm beide APIs zu verwenden, empfiehlt es sich nur eine der Schnittstellen zu benutzen. Einen Unterschied zwischen den APIs lässt sich am Host-Code erkennen. Die Programmierung der verwendeten Kernel bleibt bei beiden Schnittstellen gleich.

C Runtime for CUDA

Bei der C Runtime for CUDA API handelt es sich hauptsächlich um C-Code, der durch einen kleinen Satz von NVIDIA typischen Befehlen erweitert wurde. Die API ermöglicht so ein einfacheres Codemanagement, unterstützt eine implizite Initialisierung sowie das Kontextmanagement und Modulmanagement. Des Weiteren regelt sie das Laden der Kernel, das Einrichten ihrer Parameter und der notwendigen Konfigurationen bevor diese ausgeführt werden können. Der Anwender hat dadurch die Möglichkeit einen Thread als C-Funktion zu schreiben und auf einfache Weise Block- und Grid- Dimensionen anzugeben. Die C Runtime for CUDA Programmierschnittstelle besteht aus zwei Teilen. Zum einen enthält sie eine Low-Level Funktion mit

einem C-Style Interface und zum anderen eine High-Level Funktion mit einem C++-Style Interface, welches auf der ersten aufbaut. Code dieser API muss mit dem von NVIDIA mitgelieferten nvcc-Compiler kompiliert werden.

CUDA Driver API

Die CUDA Driver API benötigt im Gegensatz zu C Runtime for CUDA mehr Code und lässt sich schwieriger programmieren und debuggen. Im Besonderen ist es aufwendiger Kernel zu konfigurieren und einzusetzen. Da diese Schnittstelle keine C-Erweiterungen enthält ist es nicht möglich eine Syntax wie bei der High-Level API zu verwenden. Statt dessen werden Kernel als Module von CUDA in Binär- oder Assemblercode geladen und die Ausführungskonfigurationen und Parameter müssen mit expliziten Funktionsaufrufen spezifiziert werden. Allerdings besteht dadurch die Möglichkeit sprachenunabhängig zu programmieren und andere Compiler zu verwenden. Des Weiteren ist die API unabhängig von der Laufzeitbibliothek, wodurch der Anwender mehr Kontrolle über das Device hat. Die meisten Funktionen sind in beiden APIs enthalten. Dennoch verfügt die Driver API über zusätzliche Anwendungen, mit denen beispielsweise der Kontext von einem Thread einem anderen übergeben werden kann.

Die CUDA Driver API ist nur rückwärtskompatibel. Das heißt, dass Anwendungssoftware, Plug-ins und Bibliotheken nicht auf Systemen mit einer älteren Version laufen können und alle Plug-ins und Bibliotheken untereinander die selbe Laufzeitbibliothek benutzen müssen.

Verwendung von Texturen

Um den Texturspeicher und das damit verbundene Cachen von Daten verwenden zu können, muss während der Laufzeit ein Bereich im globalen Speicher als Textur bestimmt werden. Die Texturen können sich überlappen und es ist möglich mehrere verschiedene über dem selben Bereich zu definieren. Weitere Informationen zu diesem Speicher folgen in Kapitel 2.5.4.

2.4.2 Erweiterungen der C-Programmiersprache

Wie bereits in Kapitel 2.4.1 erwähnt verwendet die C Runtime for CUDA API C-Erweiterungen. Hier soll nur ein grober Überblick gegeben werden und

lediglich die verwendeten Methoden sind ausführlicher beschrieben. Mehr Informationen kann dem Best Practices Guide von NVIDIA entnommen werden (Siehe [5]).

Jede Funktion verfügt über einen Funktionskennzeichner, welcher angibt, wo diese ausgeführt wird und von wo aus sie aufgerufen werden kann. Auch die Variablentypen besitzen ein zusätzliches Attribut, das kennzeichnet in welchem Speicher auf welcher Plattform sich die Daten befinden. Außerdem wurde der C-Code um Built-In Variablen sowie Vektor- und Datentypen erweitert, mit denen beispielsweise die Block- und Gridgrößen und deren Indizes angegeben und abgefragt werden können. Neben den schon erwähnten Funktionen zum Laden und Initialisieren eines Kernels und seiner Threadblöcke gehören zu den Erweiterungen auch verschiedene Funktionen für mathematische Berechnungen, für die Verwendung von Texturen, zum Messen der Zeit, eine Profiler Counter Function und solche, mit denen der Anwender Threads synchronisieren und den Zugriff auf Speicherstellen regeln kann. Für bestimmte Funktionen ist es möglich so genannte *Atomic Functions* zu verwenden, die keine Unterbrechung zwischen dem Lesen, Bearbeiten und Schreiben eines Wertes erlauben.

Bestimmte Teile der Standard C-Bibliothek werden auch von der GPU unterstützt.

Verwendete Built-In Variablen

Wie bereits in Kapitel 2.3.2 erwähnt ist es möglich zu jedem Thread bzw. Block die genaue ID abzufragen. Dazu enthält CUDA die Built-In Variablen `threadIdx` und `blockIdx`. Beide bestehen aus 3 Komponenten, welche die genaue Position in dem maximal 3 dimensional übergeordneten Element angeben. Eine weitere wichtige Variable ist `blockDim`, welche ebenfalls aus 3 Komponenten besteht und die Dimension der Blöcke enthält. Möchte der Anwender wissen, wie viele Threads sich in einem Warp befinden, so kann er dies mit der Variable `warpSize` überprüfen.

C-Funktionen zum Messen der Zeit

Um die Performanz eines Programms zu bestimmen, ist es hilfreich Werte, wie die erreichte Bandbreite, berechnen zu können. Notwendig dafür ist, die Zeit zu ermitteln, die ein Kernel zum Bearbeiten einer bestimmten Menge von Daten braucht. Es ist möglich dies auf zwei verschiedene Weisen zu

realisieren, da eine Zeitmessung mit Hilfe von CPU- oder GPU- Timern statt finden kann. Die Vor- und Nachteile beider Möglichkeiten sollen im Folgenden dargestellt werden.

Soll mit Hilfe des CPU-Timers ermittelt werden, wie lange ein Kernel braucht, um seine Berechnungen durchzuführen, so kommt es vor, dass dieser lediglich die Zeit misst, die für den Aufruf der CUDA API benötigt wird. Dies liegt darin begründet, dass es sich bei Funktionen, wie Kernelaufrufe und Kopiervorgänge zwischen Speichern, um asynchrone Prozesse handeln kann. In diesem Fall wird die Kontrolle an den CPU Thread zurückgegeben bevor die Prozesse ihre Berechnungen beendet haben. Wenn die korrekte Zeit für die komplette Abarbeitung ermittelt werden soll muss zuerst der CPU Thread mit dem GPU Thread synchronisiert werden. In diesem Fall wird der CPU Thread bis zur Beendigung der Berechnungen blockiert. Bei den resultierenden Werten sollte jedoch bedacht werden, dass die Synchronisation von CPU und GPU ein Drosseln in der GPU-Pipeline hervorrufen und so die Performanz negativ beeinflussen kann.

Um die benötigte Zeit eines Streams zu ermitteln eignet sich nur der synchrone Defaultstream (Siehe auch Kapitel 2.3.3). Bei allen Anderen können Zeiten für den Aufruf von weiteren Streams in den Messungen enthalten sein. Im Defaultstream wird das Event erst ausgelöst, wenn alle Tasks und Kommandos von allen vorangegangenen Streams beendet wurden.

Die CUDA Event API unterstützt Aufrufe, die Events erzeugen und zerstören können. Diese ermöglichen es, eine genaue Zeitanalyse der Kernel mit Hilfe von GPU-Timern durchzuführen. Spezielle Funktionen messen dafür die Millisekunden zwischen dem Eintreten der Events. Ein weiterer Vorteil der GPU Timer ist, dass sie mit der GPU-Clock arbeiten und somit betriebssystemunabhängig sind.

Synchronisationsfunktionen

Es gibt verschiedene Funktionen mit deren Hilfe Threads synchronisiert werden können. Mit `threadfence()` ist es möglich die weitere Abarbeitung zu blockieren, bis alle globalen Speicherzugriffe für alle Threads auf einem Device und alle Speicherzugriffe von allen Threads des selben Blocks auf den Shared Memory abgeschlossen und die Ergebnisse sichtbar sind. Notwendig wird dies, wenn Threads Daten lesen sollen, die von anderen erst kurz zuvor

geschrieben wurden.

Bei der Verwendung von `syncthreads()` werden nur die Threads innerhalb eines Blocks synchronisiert. Devices mit einer *Compute Capability* von mindestens 2.0 unterstützen 3 weitere Versionen dieses Aufrufs, welche hier jedoch nicht genauer beschrieben werden.

2.5 Die Architektur von NVIDIA Grafikkarten

Grafikkarten werden im Personal Computer für die Bildschirmanzeige verwendet. Sie wandeln Daten um, damit diese auf dem Monitor ausgegeben werden können. Da dies in der Regel mit einem erheblichen Rechenaufwand verbunden ist, wird der Großteil der Transistoren für Rechenaufgaben verwendet anstatt wie bei der CPU für Steuerungsaufgaben und Datencaching (siehe Abbildung 2.2). Die dadurch auf der Grafikkarte entstehenden Verzögerungszeiten beim Zugriff auf den Speicher werden durch Berechnungen ausgeglichen, welche die Prozessoren in der Wartezeit durchführen. Um ein Programm mit einer guten Performanz zu schreiben ist es unablässig die Hardware genau zu kennen. Aus diesem Grund soll in den folgenden Kapiteln die Architektur der Grafikkarten genauer beschrieben werden (Siehe auch [4]). Dabei wird zwischen Devices mit einer *Compute Capability* von 1.x und 2.x unterschieden, da diese diverse Unterschiede aufweisen.

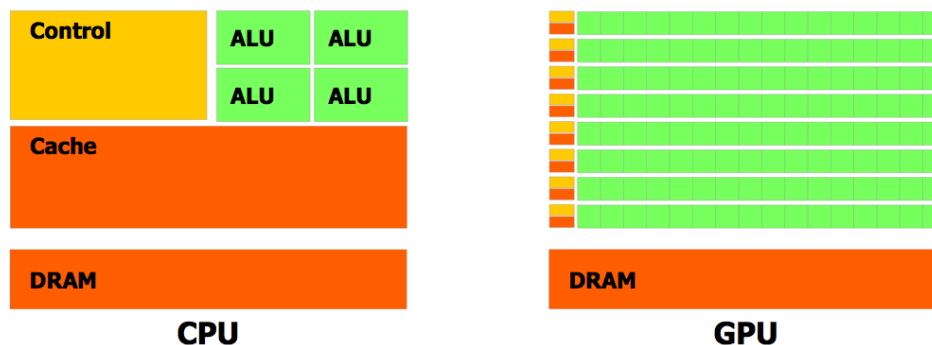


Abbildung 2.2: CPU- und GPU- Architektur [4].

2.5.1 Die *Compute Capability*

Die *Compute Capability* beschreibt die Charakteristik der Hardware und die unterstützten Befehle. Zudem gibt sie weitere Spezifikationen wie die maximale Anzahl von Threads pro Block und Register pro Multiprozessor an. Der erste Teil der Revisionsnummer beschreibt die Architektur während die zweite Ziffer für eine stufenweise Optimierung der Bauart und das Hinzufügen von neuen Features steht. Allerdings gibt es auch Hardwarecharakteristika, die nicht von der *Compute Capability* abhängen. Ein Beispiel hierfür ist die Möglichkeit zur zeitgleichen Ausführung eines Kernels mit einem asynchronen Datentransfer. Solche Eigenschaften können jedoch mit einem entsprechenden Befehl direkt abgefragt werden.

2.5.2 Die Architektur von Grafikkarten mit einer *Compute Capability* von 1.x

Die GPU besteht aus mehreren Multiprozessoren, die sich einen globalen Speicherbereich teilen. Bei Devices mit einer *Compute Capability* von 1.x besteht ein Multiprozessor aus 8 CUDA-Grafikkernen, welche für Integer und Gleitkommaarithmetik mit einfacher Genauigkeit verwendet werden. Des Weiteren verfügt die Grafikkarte über eine Einheit für Gleitkommaoperationen mit doppelter Genauigkeit und 2 Spezialeinheiten für Transzendente Gleitkommafunktionen mit einfacher Genauigkeit. Die letzte Einheit kann auch für Gleitkommamultiplikationen mit einfacher Genauigkeit verwendet werden. Des Weiteren besitzt der Multiprozessor einen Warp Scheduler, welcher die Ausführung der einzelnen Warps plant. Sobald weitere Anweisungen abgearbeitet werden können, sucht dieser nach einem Warp mit aktiven Threads welche bereit sind Operationen durchzuführen.

Der Warp Scheduler plant 4 Taktsignale für eine Integer- bzw. Gleitkommaoperation mit einfacher Genauigkeit, 32 Taktsignale für eine Gleitkommaoperation mit doppelter Genauigkeit und 16 Taktsignale für eine Transzendente Gleitkommafunktion mit einfacher Genauigkeit ein. Des Weiteren verfügt jeder Multiprozessor über einen konstanten Read-Only Cache, auf welchen alle Threads eines Multiprozessors zugreifen können. Multiprozessoren sind gruppiert in so genannten *Texture Processor Clusters*, kurz TPCs. Bei einer *Compute Capability* von 1.0 und 1.1 sind in jedem Cluster 2, bei 1.2 und 1.3 in jedem 3 enthalten. Jeder TCP hat einen Texturcache, den sich die

Multiprozessoren teilen um das Lesen aus dem Texturspeicher zu beschleunigen. Für den Zugriff auf diesen Cache verfügt jeder Multiprozessor über eine Textureinheit, welche die verschiedenen Adressierungsmethoden und Datenfilterungen ermöglicht.

2.5.3 Die Architektur von Grafikkarten mit einer *Compute Capability* von 2.0

Bei Devices mit einer *Compute Capability* von 2.0 besteht der Multiprozessor aus 32 CUDA-Grafikkernen für arithmetische Operationen, 4 Spezialeinheiten für Transzendente Gleitkommafunktionen mit einfacher Genauigkeit und 2 Warp Schemulern. Zu jedem Ausführungszeitpunkt bearbeitet der erste Scheduler einen Befehl aus einem Warp mit gerader ID und der Zweite eine Operation aus einem mit ungerader ID. Die einzige Ausnahme ist, wenn eine Gleitkommaoperation mit doppelter Genauigkeit ausgeführt werden soll. In diesem Fall kann nur einer arbeiten, da dieser alle Grafikkerne benötigt. Wird ein Befehl für alle Threads in einem Warp ausgeführt so plant der Scheduler 2 Taktsignale für eine Integer- bzw. Gleitkommaoperation mit einfacher Genauigkeit, 2 für eine mit doppelter Genauigkeit und 8 für eine Transzendente Gleitkommafunktion mit einfacher Genauigkeit ein. Des Weiteren verfügt jeder Multiprozessor über einen Read-Only Cache zum Beschleunigen der Lesezugriffe auf den konstanten Speicher, welchen sich alle funktionellen Einheiten teilen.

Jeder Multiprozessor besitzt zudem einen eigenen L1 Cache und teilt sich einen L2 Cache mit allen anderen Prozessoren. Beide werden benutzt, um den Zugriff auf den lokalen und globalen Speicher zu cachieren. Der On-Chip Speicher kann als 48 KB Shared Memory mit 16 KB L1 Cache oder anders herum konfiguriert werden. Multiprozessoren werden als *Graphik Processor Clusters*, kurz GPCs, gruppiert und jedes dieser Cluster enthält 4 Multiprozessoren. Einige Devices mit einer *Compute Capability* von 2.0 können mehrere Kernel zur gleichen Zeit ausführen. Die maximale Anzahl liegt aktuell bei 16.

2.5.4 Der Grafikkartenspeicher

Die Grafikkarte fungiert als Co-Prozessor zum CPU-Host-System. Beide Einheiten besitzen einen separaten Speicher, welcher als Host Memory bzw. Device Memory bezeichnet wird. Da ein Kernel nur mit Daten arbeiten kann,

die sich auf der Grafikkarte befinden müssen diese zuvor vom Host auf das Device übertragen und im dortigen Speicher abgelegt werden. In der CUDA Architektur ist der RAM virtuell und physisch in verschiedene Arten von Speichern unterteilt. Diese verfügen über unterschiedliche Eigenschaften, Lebensdauer und Größe (siehe Tabelle 2.1) um verschiedenen Anwendungsbereichen gerecht zu werden. Der jedem Thread automatisch während seiner Ausführung zugewiesene private Speicher ist nicht aufgelistet.

Die folgende Abbildung 2.3 gibt einen Überblick darüber, wo sich die verschiedenen Arten von Speicher befinden und wie sie miteinander in Verbindung stehen.

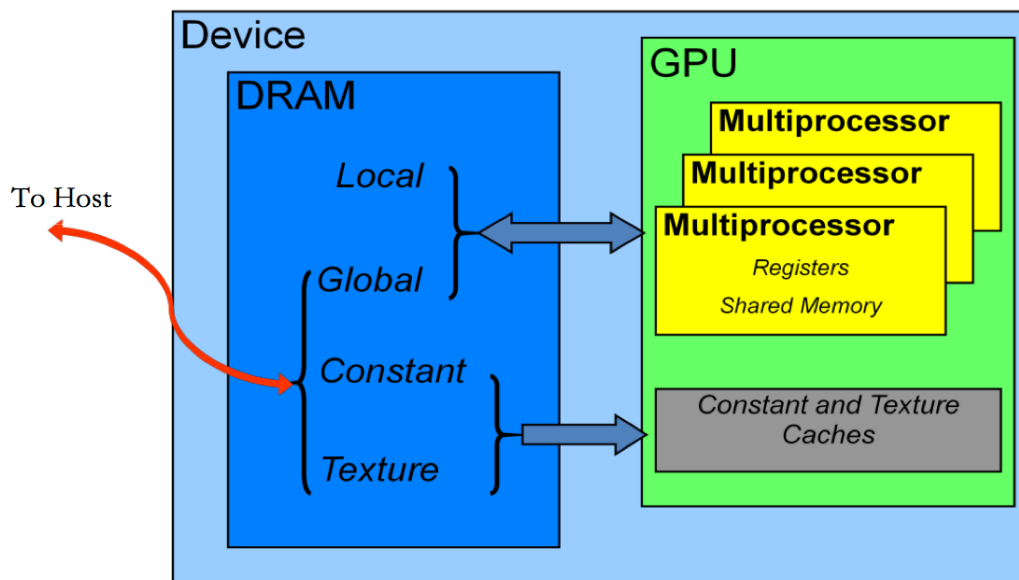


Abbildung 2.3: Verschiedene Speicherbereiche auf einem CUDA-Device [5].

Tabelle 2.1: Attribute der verwendeten Speicher.

Speicher	Lebensdauer	Zugriff	Anwendungsbereich
Globaler Speicher	Wird vom Host belegt und freigegeben	R/W	Alle Threads und Host
Lokaler Speicher	Lebensdauer des zugehörigen Threads	R/W	Zugehöriger Thread
Texturspeicher	Wird vom Host belegt und freigegeben	R	Alle Threads und Host
Konstanter Speicher	Wird vom Host belegt und freigegeben	R	Alle Threads und Host
Gemeinsamer Speicher	Lebensdauer des zugehörigen Blocks	R/W	Alle Threads und Host
Register	Lebensdauer des zugehörigen Threads	R/W	Zugehöriger Thread

Speicher	Lage	Gecached	Größe
Globaler Speicher	Off-Chip	Nein	bis zu 2048 *
Lokaler Speicher	Off-Chip	Nein	16 KB - 512 KB pro Thread **
Texturspeicher	Off-Chip, Cache ist On-Chip	Ja	Cache 6 KB - 8 KB pro MP *
Konstanter Speicher	Off-Chip, Cache ist On-Chip	Ja	64 KB, 8 KB Cache pro MP
Gemeinsamer Speicher	On-Chip	Nein	16 KB - 48 KB pro MP **
Register	On-Chip	Nein	8 K - 32 K **

*Hängt von der *Compute Capability* ab.

**Deviceabhängig

Der Globale Speicher

Der globale Speicher ist bis zu 2048 MiB groß und neben dem Konstanten- und Texturspeicher der einzige, welcher von allen Threads und dem Host adressiert werden kann. Ein Bereich im globalen Speicher kann entweder als CUDA Array oder linearer Speicherbereich belegt werden. CUDA Arrays sind optimiert für die Verwendung von Texturen, haben aber den Nachteil, dass nur die CPU die Möglichkeit hat diesen zu beschreiben. Der lineare Speicher hingegen kann auch von der GPU zur Datensicherung benutzt werden. Bei Devices mit einer *Compute Capability* von 1.x wird ein 32 Bit Adressraum verwendet während die neueren mit einem 40 Bit Raum arbeiten. Im Normalfall werden alle Daten, bei denen es sich nicht um Konstanten handelt, zuerst in den globalen Speicher übertragen. Später ist es möglich bestimmte Bereiche als Texturen zu definieren.

Wenn ein Warp den globalen Speicher ansteuert, wird dies in so wenigen Transaktionen wie möglich zusammengefasst. Für Devices mit einer *Compute Capability* von 1.x werden mindestens zwei Zugriffe, je einer pro halben Warp, benötigt. Bei Grafikkarten mit einer *Compute Capability* von 2.0 wird der Warp nicht unterteilt und somit im besten Fall nur eine Transaktion durchgeführt.

Pro Zugriff auf den globalen Speicher ist es möglich ein komplettes Segment der Größe 32, 64 oder 128 Byte auszulesen bzw. zu beschreiben. Dabei werden alle Threads eines Warps bei einer *Compute Capability* von 2.0 und alle eines halben bei einer *Compute Capability* von 1.x bedient, deren angesteuerte Adressen sich innerhalb dieses Abschnitts befinden. Die erste Adresse des Segments ist immer ein Vielfaches seiner Größe. Der Anfang des Abschnitts wird also nicht durch die Position des ersten Wortes, sondern durch die Einteilung des Adressraumes bestimmt. Wie viele Aufrufe nötig sind, um den ganzen Warp abzuarbeiten, hängt von der Wortgröße pro Thread und deren Verteilung im globalen Speicher ab. Nicht zusammengefasste Zugriffe reduzieren durch die Anzahl der zusätzlichen Transaktionen und dem Auslesen von nicht benötigten Daten den Befehlsdurchsatz. Aufgrund dieser möglichen Performanzreduzierung ist das Organisieren der Speicherzugriffe ein sehr wichtiger Aspekt der Optimierung. Für Devices mit einer *Compute Capability* von 1.2 und 1.3 sind die Bedingungen zwar weniger streng dennoch ist die Differenz in der Bandbreite gerade bei einer großen Anzahl von Daten deutlich spürbar.

Aus diesem Grund ist es wichtig die Voraussetzungen zu erfüllen, unter denen die Anfragen von Threads in einem Warp zusammengefasst werden können. Die Rahmenbedingungen hängen von der *Compute Capability* der verwendeten Grafikkarte ab.

Bei Devices mit einer *Compute Capability* von 1.0 und 1.1 wird die Transaktion für einen Warp in zwei Zugriffe unterteilt und jede Hälfte einzeln bearbeitet. Die Anfrage der 16 Threads können zusammengefasst werden, wenn die Wortlänge in jedem Thread nur 4,8 oder 16 Byte beträgt. Bei einer Wortgröße von 4 bzw. 8 Byte müssen sich alle 16 Wörter in dem selben 64 bzw. 128 Byte großen Segment befinden. Steuert jeder Thread Wörter der Größe 16 Byte an, so werden insgesamt 256 Byte angesprochen, von denen sich die ersten 8 Wörter im selben 128 Byte großen Segment befinden müssen und die letzten 8 in den darauf folgenden 128 Byte. Des Weiteren müssen die Threads auf die Wörter sequentiell zugreifen. Das heißt, dass der k .te Thread das k .te Wort in dem selben Segment adressiert. Dabei ist es jedoch nicht nötig, dass alle Threads auf ihre Daten zugreifen.

Sind diese Bedingungen erfüllt so können alle Zugriffe in einer einzigen Transaktion der Größe 64 bzw. 128 Byte oder wie im letzten Fall zwei Transaktionen der Größe 128 Byte ausgeführt werden. Sind diese Bedingungen nicht erfüllt, so erfolgt der Zugriff in 16 separaten 32 Byte großen Speicherzugriffen. Wird dann beispielsweise pro Thread nur 4 Byte benötigt, reduziert sich die Bandbreite um $1/8$. Dieser Fall tritt auch auf, wenn die Wörter zwar aufeinander folgen, sich aber in verschiedenen Segmenten befinden.

Bei Devices mit einer *Compute Capability* von 1.2 und 1.3 müssen bei einem zusammengefassten Zugriff die einzelnen Wörter von den Threads nicht mehr der Reihe nach adressiert werden und es ist möglich mehr als einmal auf die selbe Adresse zuzugreifen. Tritt der Fall ein, dass die Wörter zwar sequentiell ausgelesen werden, sich aber in verschiedenen Segmenten befinden, so richtet sich die Zahl der Abfragen nicht mehr nach der Anzahl von Threads in dem halben Warp, sondern nach den verschiedenen Segmenten, auf die sich die Daten verteilen. Des Weiteren können auch Zugriffe zusammengefasst werden, wenn die Wörter eine Größe von 1 oder 2 Byte haben. Das genaue Protokoll für diesen Vorgang sieht folgendermaßen aus: Zuerst wird in dem aktuellen Warp der Thread mit der kleinsten ID ausgewählt und das Segment im globalen Speicher zu seiner Adresse ermittelt. Die Segmentgröße beträgt

32 Byte, wenn jeder Thread auf 1 Byte zugreift, 64 Byte, bei 2 Byte und 128 Byte bei 4,8 oder 16 Byte. Anschließend werden alle Threads gesammelt, die Daten aus dem selben Segment bearbeiten. Stellt sich dabei heraus, dass nur die Wörter der oberen oder unteren Hälfte des Segments benötigt werden, wird die Segmentgröße entsprechend von 128 Byte auf 64 oder 32 bzw. von 64 auf 32 Byte reduziert. Anschließend wird das Segment ausgelesen und alle Threads markiert, die ihre Transaktion durchgeführt haben. Dies wiederholt sich so lange bis der gesamte Warp als bearbeitet markiert ist.

Ab Devices mit einer *Compute Capability* von 2.0 wird der Zugriff auf den globalen Speicher gecached und es ist möglich einzustellen, ob hierfür nur der L2 Cache oder zusätzlich der L1 Cache benutzt werden soll.

Jede Cache-Line ist 128 Byte lang und zeigt auf das zugehörige gleichlange Segment im Device Memory. Ein weiterer Unterschied ist, dass für einen Warp die Abfrage nicht mehr aufgeteilt wird. Es können also alle 32 Threads ihre Daten zeitgleich bearbeiten. Speicherzugriffe werden in einer minimalen Anzahl von Transaktionen durchgeführt, deren Größe der der Cache Line entspricht. Wenn pro Thread auf 8 Byte, also insgesamt auf 256 Byte zugegriffen wird, so werden zwei Anfragen benötigt. Eine pro halben Warp. Benutzt jeder Thread 16 Byte, so sind dementsprechend vier Transaktionen notwendig. Diese Zugriffe können unabhängig voneinander bearbeitet werden.

Abbildung 2.4 aus dem Programming Guide von NVIDIA [4] verdeutlicht nochmals die möglichen Zugriffsszenarien auf den globalen Speicher und die Unterschiede in der Handhabung dieser zwischen den einzelnen Devices unterschiedlicher *Compute Capability*. Besonders beachtet werden muss, ob die Daten mit einer Schrittweite ausgelesen werden, die größer als 1 ist. Beträgt diese beispielsweise 2, so werden nicht alle Daten des angesprochenen Segments benötigt. Dies reduziert die Bandbreite um die Hälfte. Ist die Schrittweite größer, so steigt dadurch auch die Zahl der benötigten Segmente bis für jedes Element eine eigene Transaktion benötigt wird.

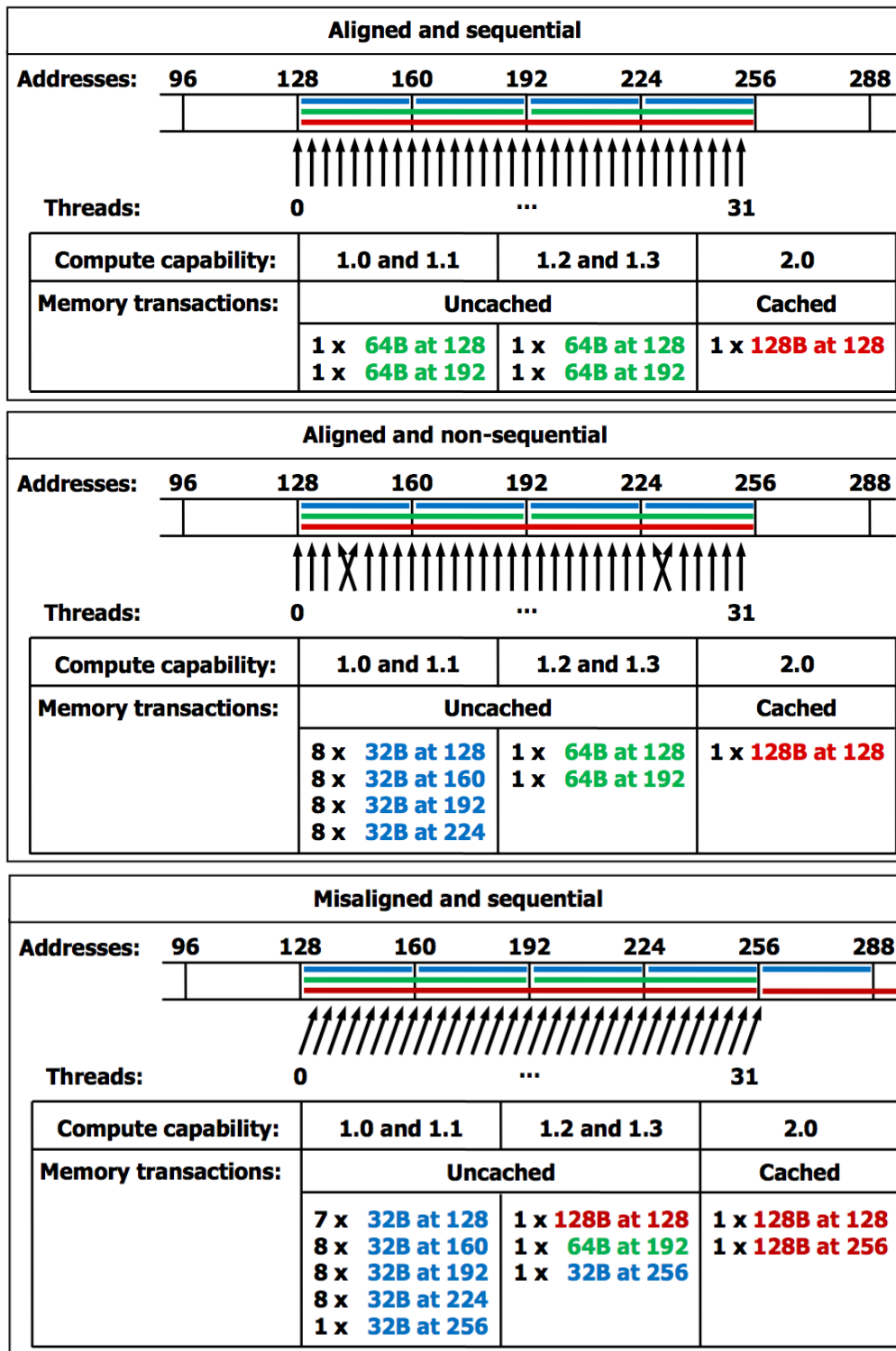


Abbildung 2.4: Mögliche Zugriffsszenarien eines Warps auf den globalen Speicher und die daraus resultierenden Speicherzugriffe basierend auf der *Compute Capability* [5].

Der Lokale Speicher

Der lokale Speicher wird nicht von dem Anwender, sondern von dem Kernel selber belegt. Dies geschieht wenn er beschließt, dass die zur Verfügung stehenden Register für die Variablen des Threads nicht ausreichend sind. Ursache dafür kann beispielsweise ein zu großer Datentyp sein. Da sich der lokale wie auch der globale Speicher nicht auf dem Chip befindet, haben sie die selben Verzögerungszeiten. Seinen Namen hat dieser Adressraum nicht durch seine Lage, sondern durch die Tatsache, dass er lokal zu einem Thread gehört.

Der lokale Speicher ist in 32 Bit Worten organisiert und seine Zugriffe können anders als bei dem globalen Speicher nur dann zusammengefasst werden, wenn die Threads auf die selben Daten zugreifen.

Der Gemeinsame Speicher

Bei dem gemeinsamen Speicher handelt es sich um einen Cache für alle Threads eines Blocks, welcher durch den Anwender organisiert wird. Somit ermöglicht er Kommunikation und einen gemeinsamen Zugriff auf Daten. Dieser Shared Memory hat eine geringe Verzögerungszeit, da er sich in der Nähe des Prozessorkerns befindet. Seine Lebensdauer beginnt und endet mit der des Blocks. Um eine möglichst hohe Bandbreite zu erzielen ist der Speicher in einzelne Module, so genannte Bänke, unterteilt auf die zeitgleich zugegriffen werden kann. Daher sollte versucht werden den Speicher so zu adressieren, dass es zu keinem Konflikt kommt. Dies geschieht beispielsweise wenn verschiedene Threads mit unterschiedlichen Wörtern in der selben Bank arbeiten. In diesem Fall wird der Aufruf in so viele konfliktfreie Transaktionen unterteilt, wie nötig. Daher empfiehlt es sich eine Wortgröße von mindestens 32 Bit zu wählen, da CUDA aufeinander folgende Daten mit einer Länge von 16 und 8 Bit jeweils zu zweit bzw. zu viert in dem selben Modul speichert. Der gemeinsame Speicher eignet sich nur dann, wenn die dort abgelegten Variablen mehr als einmal verwendet werden, da seine Lebensdauer erst mit der des Blocks beginnt und die Daten vorher aus dem globalen Speicher geladen werden müssen. Neben der Möglichkeit redundante Zugriffe auf einen langsamen Speicher zu vermeiden kann er auch verwendet werden um nicht zusammenfassbare Transaktionen zu verhindern. Dazu müssen im ersten Schritt, den Rahmenbedingungen entsprechend, die Daten sequentiell aus dem globalen Speicher in den gemeinsamen Speicher übertragen werden. Anschließend

ist es möglich in jeder beliebigen Reihenfolge darauf zuzugreifen, da es abgesehen von Bankkonflikten keine Beschränkungen für Daten von sequentiellen Threads gibt. Verwendet werden kann dies, wenn eine Matrix im globalen Speicher zeilenweise gespeichert wurde und für Berechnungen spaltenweise ausgelesen werden muss. Es ist nicht möglich diesen Zugriff für mehrere Threads zusammenzufassen, da die angeforderten Daten nicht sequentiell aufeinander folgen. Wenn ohnehin alle Spalten der Matrix ausgelesen werden müssen, können die Daten zeilenweise in den gemeinsamen Speicher übertragen und dort spaltenweise darauf zugegriffen werden. Zusammengefasst hat die Verwendung des gemeinsamen Speichers folgende Vorteile. Zum einen ist es möglich Transaktionen auf den globalen Speicher zusammenzufassen auch wenn die Threads die Rahmenbedingungen beim Zugriff nicht einhalten können und zum anderen werden redundante Abfragen verhindert. Die Größe des gemeinsamen Speichers für jeden Block ist begrenzt, da die Grafikkarten pro Multiprozessor nur über 16 KB - 48 KB verfügen. Aus diesem Grund hängt die Dimension des Blocks auch von der benötigten Menge an Ressourcen ab. Ein Teil des zur Verfügung stehenden gemeinsamen Speichers wird von dem Kernel belegt, um dort die Funktionsparameter zu sichern. Eine Möglichkeit dies bei langen Parameterlisten zu reduzieren ist, die Werte als Konstanten anzulegen und drauf zu verweisen.

Devices mit einer *Compute Capability* von 1.x verfügen über 16 Module mit einer Bandbreite von jeweils 32 Bits pro Taktrate und sind so organisiert, dass aufeinander folgende 32 Bit lange Wörter auf verschiedene Bänke verteilt werden. Beim Zugriff auf den Speicher wird wie bei dem globalen Speicher der Warp in zwei Teile unterteilt und jede Hälfte separat und unabhängig voneinander bearbeitet. Zwischen Threads, die sich in verschiedenen Warphälften befinden, kann es zu keinem Bankkonflikt kommen. Schreiben mehrere Threads auf die selbe Adresse so wird nur ein Zugriff ausgeführt. Welcher die Transaktion durchführt ist dabei unklar.

Bei Devices mit einer *Compute Capability* von 2.0 hat der gemeinsame Speicher 32 Bänke, die so organisiert sind, dass 32 Bit Wörter, die aufeinander folgen, in verschiedenen Modulen gespeichert werden. Jede Bank hat eine Bandbreite von 32 Bits pro 2 Taktzyklen. Ähnlich wie bei Devices von geringerer *Compute Capability* kann es auch hier zu Konflikten kommen. Greifen jedoch mehrere Threads auf das selbe Wort in einer Bank zu, so wird dieses

nur einmal ausgelesen und an alle Threads geschickt, die die Daten angefordert haben. Versuchen mehrere Threads auf die selbe Adresse zu schreiben, so wird auch bei diesen Grafikkarten nur ein Zugriff ausgeführt. Welcher ist dabei undefiniert.

Bei einem einfachen Zugriffsszenario liest zuerst jeder Thread einen bestimmten Bereich aus dem globalen Speicher und überträgt diesen in den gemeinsamen Adressraum. Anschließend werden alle Threads synchronisiert, um sicher zu gehen, dass die Transaktionen vollständig durchgeführt sind. Nachdem alle Threads die Daten bearbeitet haben, können die Ergebnisse wieder im globalen Speicher gesichert werden. Die einfachste, konfliktfreie Art bei einem Device mit einer *Compute Capability* von 1.x den Speicher zu nutzen ist, dass jeder Thread auf ein Wort mit einer Länge von 32 Bit zugreift, wobei dieses durch die *ThreadID* bestimmt wird. Auf diese Weise kann es zu keinem Bankkonflikt kommen, da zu jedem Ausführungszeitpunkt ein halber Warp bedient wird und sich die 16 Wörter für die 16 Threads in 16 verschiedenen Bänken befinden. Bei Devices mit einer *Compute Capability* von 2.0 bleibt dies bestehen. Der einzige Unterschied ist, dass hier 32 Threads gleichzeitig bearbeitet werden und für diese 32 Bänke zur Verfügung stehen.

Der gemeinsame Speicher unterstützt auch einen Broadcast-Aufruf, bei dem ein 32 Bit Wort gelesen und zeitgleich an alle Threads weitergereicht wird, die dieses benötigen.

Der Konstante Speicher

Der Speicher für Konstanten befindet sich zwar im Device Memory wird jedoch On-Chip gecached. Die Latenz tritt also nur bei einem Cache Miss auf. Das Lesen aus dem Puffer selbst ist genauso schnell wie der Zugriff auf ein Register. Bei Karten mit einer *Compute Capability* von 1.x wird wie bei dem globalen Speicher der Aufruf in zwei Transaktionen unterteilt. Allerdings können Zugriffe auf verschiedene Adressen nicht zusammengefasst werden. Zusätzlich zum konstanten Speicher, welcher von allen Devices unterstützt wird, verfügen Devices mit einer *Compute Capability* von 2.0 über eine *Load Uniform*-, kurz *LDU*-, Operation. Mit dieser Funktion lädt der Compiler alle Variablen, die in den globalen Speicher zeigen, nicht von der *ThreadId* abhängen oder auf die von dem Kernel aus nur lesend zugegriffen wird.

Der Texturspeicher

Der Texturspeicher befindet sich im Device Memory und wird ebenfalls gecached. Um diesen zu nutzen legt der Anwender über Teile des linearen Speicherbereichs oder CUDA Arrays eine Textur, wodurch die Zugriffsverzögerung nur bei einem Cache Miss auftritt. Die beste Performanz wird erreicht, wenn benachbarte Threads, die sich gemeinsam in einem Warp befinden, auf Adressen zugreifen, die ebenfalls benachbart sind. Ein weiterer Vorteil ist, dass die Adressberechnung außerhalb des Kernels mit bestimmten Einheiten durchgeführt wird. Des Weiteren können komprimierte Daten in einer Operation auf verschiedene Variablen verteilt werden. Allerdings muss beachtet werden, dass Texturen, die in einem Kernel gesetzt wurden beim Auslesen im selbigen noch undefiniert sind. Außerdem kann die Referenz auf eine Textur nur als statische globale Variable definiert werden und es ist nicht möglich sie als Parameter einer Funktion zu übergeben.

Die Register

Verzögerungen bei Zugriffen auf Register können nur durch write-after-read Abhängigkeiten entstehen. In diesem Fall beträgt sie 24 Taktzyklen, kann aber ausgeglichen werden, wenn die Auslastung der Karte bei 25% liegt (Siehe auch 2.6.1). Der Compiler und der Hardware Thread Scheduler planen den Zugriff so optimal wie möglich um Bankkonflikte bei den Registern zu vermeiden. Die besten Resultate werden erreicht wenn die Anzahl der Threads pro Block durch 64 teilbar ist.

2.5.5 Der CPU-Speicher

Neben Funktionen, mit deren Hilfe man einen Speicherbereich auf dem Device reservieren kann, unterstützt die Laufzeitbibliothek auch solche zum Belegen von Adressraum auf dem Host. Dazu gehören Befehle für *Pageable* aber auch für *Paged-Locked* bzw. *Pinned Memory*. Letztere haben den Vorteil, dass Kopiervorgänge zwischen Host und Device unter Verwendung von Streams gleichzeitig mit Kernelausführungen statt finden können. Bei der normalen Datenübertragungen wird die Kontrolle an den Host Thread erst zurück gegeben, wenn der Transfer abgeschlossen ist. Es gibt Grafikkarten bei denen der Anwender zudem die Möglichkeit hat diesen *Page-Locked* Bereich des Host-Speichers in den Adressraum des Device zu mappen, was eine

Übertragung überflüssig macht und bei Systemen mit einem Front Side Bus die Bandbreite erhöht.

Der Nachteil ist, dass es sich bei *Page-Locked* Memory um eine knappe Ressource handelt. Wird hiervon zu viel verwendet dann verringert sich die Systemperformanz, da der physikalische Speicher, der dem Betriebssystem zur Verfügung steht, reduziert wird. Alle Threads des Hosts haben die Möglichkeit auf einen Block in diesem Speicher zuzugreifen. Standardmäßig kann nur der Thread, welcher den Bereich belegt hat, seine Vorteile nutzen. Durch setzen eines Flags kann dies jedoch auf alle Threads übertragen werden.

Standardmäßig wird *Page-Locked* Memory als *cachable* definiert. Jedoch kann die Einstellung *write-combining* optimaler sein. In diesem Fall werden die Ressourcen der L1 und L2 Caches freigegeben wodurch anderen Anwendungen mehr Puffer-Speicher zur Verfügung steht. Allerdings sollte dieser nur für Daten verwendet werden, die der Host schreibt und nicht liest, da ein Lesezugriff von Seiten der CPU langsam ist.

Wie schon erwähnt kann ein *Page-Locked* Speicherbereich in den Adressraum des Devices gemapped werden. In diesem Fall hat ein solcher Block eine Adresse im Host und im Device Speicher, welche mit entsprechenden Funktionen abgefragt werden können. Von dem Kernel aus direkt auf den Adressraum des Hostsystems zuzugreifen hat mehrere Vorteile. Es muss kein Speicherbereich auf dem Device belegt oder Daten kopiert werden. Die benötigten Transaktionen führt der Kernel selbständig durch. Wurden die Daten in den Adressraum des Devices gemapped, so überlappt der Datentransfer automatisch mit der Ausführung des Kernels. Da aber in diesem Fall Host und Device auf den selben Speicher zugreifen, müssen Streams oder Events sicher stellen, dass Daten, die gelesen werden sollen, vorher vollständig geschrieben wurden.

2.6 Optimiertes Programmieren mit CUDA

Die erreichte Performanz eines Programms hängt von der Umsetzung des Algorithmus ab. Besonders bei wiederholten Kernelaufrufen unter Einsatz großer Datenmengen wirkt sich eine suboptimale Programmierung deutlich auf die Gesamtzeit aus. Ein Beispiel ist die Tatsache, dass die Verwendung von Kontrollstrukturen zu einer seriellen Abarbeitung aller Threads in einem Warp führen kann. Dies erhöht die Gesamtzeit um einen Faktor von 32. Zum Erreichen einer optimalen Laufzeit wird empfohlen, die im Folgenden

aufgezeigten Strategien anzuwenden. In diesem Kapitel werden vor allem die für diese Arbeit wichtigen Punkte beschrieben. Weitere Empfehlungen zur Optimierung der Laufzeit kann Kapitel 4 und 5 aus [5] entnommen werden.

1. Die Anzahl von parallelen Befehlen maximieren, um maximale Leistung zu erzielen.
2. Den Zugriff auf die Speicher optimieren, um die Bandbreite möglichst gut auszunutzen.
3. Bei der Verwendung von Anweisungen einen maximalen Datendurchsatz anstreben.

Wie stark sich Verbesserungen im Programmcode auf die Performanz auswirken ist von Fall zu Fall verschieden.

2.6.1 Optimierung des Befehlsdurchsatzes

Der häufigste Grund für Verzögerungszeiten ist das Warten der Threads auf Eingangswerte. Diese Latenz ist besonders hoch sobald Daten aus dem globalen Speicher angefordert werden. Eine Grafikkarte kann solche Zugriffe ausgleichen, wenn entsprechend viele Operationen anderer Threads in der Zwischenzeit bearbeitet werden können, die mit Werten aus einem prozessornahen Speicher arbeiten. Zu jedem Ausführungszeitpunkt wählt der Scheduler eines Multiprozessors einen Warp aus, dessen Anweisungen bereit sind durchgeführt zu werden und gibt die nächste Instruktion der Threads in Auftrag. Die Anzahl der Taktzyklen zwischen den Ausführungszeitpunkten zweier Anweisungen auf einem Prozessor wird als Verzögerungszeit bezeichnet. Mit einer entsprechend großen Menge an Warps kann diese Wartezeit verborgen werden. Wie viele Instruktionen weiterer Warps dafür verwendet werden müssen hängt von dem Datendurchsatz des Befehls ab. Soll beispielsweise eine Verzögerungszeit L mit einer Operation, die x Taktzyklen arbeitet, verborgen werden, so sind L/x Anweisungen notwendig.

Threads unterschiedlicher Blöcke können weder synchronisiert werden noch Daten über den gemeinsamen Speicher austauschen. Aus diesem Grund muss der Informationsaustausch über den globalen Speicher erfolgen. Da die Threads nicht synchronisiert werden können wird ein Kernel benötigt der die Daten schreibt und ein weiterer, welcher die Werte anschließend ausliest. Das

Programm erzielt eine deutlich schlechtere Performanz, da sich die Gesamtzeit durch die Ausführungszeit des zweiten Kernels, sowie die Verzögerungszeit des globalen Speichers, erhöht. Es empfiehlt sich daher die Verwendung atomarer Blöcke ohne Kommunikation einzelner Threads. Auch Synchronisationspunkte können die Ausführung eines Warps verzögern. In diesem Fall werden Threads bearbeitet, die diese Barriere nicht betrifft.

Die Anzahl der Threads und Blöcke sollten so gewählt werden, dass die Hardware ausreichend ausgelastet ist. Die Metrik Auslastung besteht aus dem Verhältnis von aktiven Warps pro Multiprozessor zur maximal möglichen Menge.

Die Anzahl von Blöcken und Warps die sich zeitgleich auf einem Multiprozessor befinden können hängt nicht nur von den Beschränkungen der Grafikkarte ab, sondern auch von den benötigten Ressourcen. Tritt der Fall ein, dass ein Block das Limit überschreitet, dann wird der Kernel nicht bearbeitet. Die Menge der benötigten Ressourcen, wie Register und Größe des gemeinsamen Speichers pro Block, kann berechnet werden.

Berechnung der Anzahl von Warps

Bei Devices mit einer *Compute Capability* von 1.x wird immer eine gerade Anzahl von Warps für einen Block zur Verfügung gestellt. Um die Menge benötigter Warps zu berechnen muss die Anzahl der Threads durch die Warpgröße geteilt und anschließend auf den nächst höheren geraden Wert aufgerundet werden.

Berechnung der Anzahl benötigter Register

Einer der Faktoren, welcher den Grad der Auslastung bestimmt, ist die Registerverfügbarkeit. Alle aktiven Threads eines Multiprozessors müssen sich die Menge der zur Verfügung stehenden Register teilen. Ist dies nicht möglich, so sind weniger Blöcke aktiv und es werden gegebenenfalls Daten in den lokalen Speicher ausgelagert, was zu erheblichen Verzögerungszeiten beim Abruf führt.

Zuerst wird die Anzahl der Warps mit der Warpgröße sowie der Menge benötigter Registern pro Thread multipliziert. Grafikkarten mit einer *Compute Capability* von 1.0 und 1.1 stellen Register in Gruppen zu 256 Stück bereit. Bei Devices mit einer *Compute Capability* von 1.2 und 1.3 bestehen

diese aus 512 und bei 2.0 aus 64 Registern. Um den eigentlichen Wert zu erhalten muss man daher den berechneten Zwischenwert auf das nächst höhere Vielfache aufrunden.

Berechnung des verwendeten gemeinsamen Speichers

Auch der gemeinsame Speicher wird in bestimmten Mindestgrößen bereit gestellt. Bei einer *Compute Capability* von 1.x liegt die Granularität bei 512 und bei 128 ab 2.0. Um die tatsächlich belegte Menge zu erhalten muss man den Wert für jeden Kernel aufrunden. In vielen Fällen hängt die Größe des verwendeten gemeinsamen Speichers von der Blockgröße ab. Da ein Warp aus 32 Threads besteht ist es sinnvoll immer Matrizen der Größe 32x32 in den Speicher zu laden. Allerdings muss dann aufgrund der Beschränkungen für aktive Threads (Siehe Abbildung B.2) jeder Prozess mehr als ein Element bearbeiten.

Die Auslastung

Um die Abhängigkeiten zwischen Registern zu verbergen ist eine minimale Auslastung von 25% notwendig. Auf den aktuellen GPUs muss eine Anwendung 24 Taktzyklen warten, bis ein Rechenergebnis genutzt werden kann. Sind auf einem Device mit einer *Compute Capability* kleiner als 1.2 mindestens 192 von 768 (25%), 768 von 1024 (18,75%) bis 1.3 und ab 2.0 384 von 1024 (25%) Threads aktiv, so kann diese Verzögerung verborgen werden. Normalerweise ist eine Auslastung von 50% ausreichend da sonst riskiert wird, dass Variablen von dem Kernel in den lokalen Speicher ausgelagert werden. Des Weiteren wird laut NVIDIA ab diesem Wert keine weitere Verbesserung der Performanz erreicht.

Größe und Dimension von Blöcken und Grids

Damit jeder Prozessor einen Block bearbeiten kann, sollten in einem Grid mindestens so viele Blöcke wie Multiprozessoren vorhanden sein. Es empfiehlt sich jedoch die Anzahl so zu wählen, dass Mehrere pro Prozessor aktiv sind, damit solche, die beispielsweise wegen Synchronisationen und Speicherzugriffen warten müssen von anderen abgelöst werden können. Dabei sollte sparsam genug mit den Ressourcen umgegangen werden, damit jeder Multiprozessor mehr als einen Block zeitgleich bearbeiten kann. Dieser Faktor

ist auch bei der Wahl der Blockgröße wichtig. Je nach Menge des benötigten gemeinsamen Speichers und der Register kann die Dimension entsprechend angepasst werden.

Größere Blöcke führen nicht automatisch zu einer höheren Auslastung. Ein Block mit 512 Threads auf einem Device mit einer *Compute Capability* von 1.1 führt zu einer Auslastung von 66%, da bei einem Maximum von 768 Threads pro Multiprozessor nur ein Block aktiv sein kann. Besteht dagegen ein Block aus 256 Threads, so beträgt die Auslastung 100%, da drei Blöcke geladen werden können (Vergleiche B.2).

Die Anzahl der Threads pro Block sollte durch die Warpgröße teilbar sein um ungenutzte Berechnungsressourcen durch Unterbelegung zu vermeiden. Wenn mehrere Blöcke pro Prozessor verarbeitet werden ist es außerdem empfehlenswert, dass jeder über mindestens 64 Threads verfügt. Besser ist jedoch ein Wert zwischen 128 und 256. Die Blockbreite und Blockhöhe sollte bei Devices mit einer *Compute Capability* von 2.0 durch die Warpgröße und bei Grafikkarten mit einer *Compute Capability* von 1.x durch die halbe Warpgröße teilbar sein. Dies liegt darin begründet, dass im besten Fall der erste Thread eines aktiven Warps auf das erste Element in dem Segment des globalen Speichers zugreift (Siehe auch 2.5.4).

2.6.2 Optimierung von Speicherzugriffen

Das Verbessern von Zugriffen auf die jeweiligen Speicher ist eines der wichtigsten Gebiete der Optimierung. Der erste Schritt ist Datenverkehr mit einer geringen Bandbreite zu meiden. Dazu gehört zum Beispiel der Transfer zwischen CPU und GPU, welcher über einen PCI Express (PCIe) Bus läuft. Des Weiteren sollten Daten in möglichst großen Paketen übertragen werden.

Die peak Bandbreite des Grafikkartenspeichers ist höher, als die zwischen Host und Device. Daher kann in einigen Fällen eine bessere Gesamtzeit erreicht werden, wenn ein Kernel auch Anweisungen bearbeitet, die nicht parallel durchgeführt werden können anstatt diese von der CPU berechnen zu lassen. Vor allem sollten Datenstrukturen auf der Grafikkarte verbleiben, die nur zum Speichern von Zwischenschritten benötigt werden.

Anfragen an den globalen Speicher haben durch die auftretende Latenz eine geringe Bandbreite, wodurch es empfehlenswert ist öfter Daten in den gemeinsamen Speicher zu übertragen. Außerdem sollten so oft wie möglich Adressräume verwendet werden, die über einen Cache verfügen, wie der für Texturen und Konstanten.

In Kapitel 2.5.4 wird die optimale Verwendung der verschiedenen Speicher genauer erläutert.

Bei einem Zugriff auf den globalen oder lokalen Speicher, welche nicht gecached werden, muss mit einer Verzögerungszeit von 400 bis 600 Taktzyklen gerechnet werden. Viel von dieser Latenz kann der Warp Scheduler ausgleichen, wenn genug unabhängige arithmetische Operationen in der Wartezeit durchgeführt werden können.

2.6.3 Optimierung des Datendurchsatzes

Für einen hohen Datendurchsatz muss darauf geachtet werden, welche Operatoren das Programm verwendet. Vor allem in häufig aufgerufenen Funktionen sollte die Benutzung von teuren Operationen wie Divisionen und Moduloberechnungen durch schnellere wie Shift ersetzt werden. Außerdem sollte der Anwender schnellere, speziellere Mathefunktionen gegenüber langsameren bevorzugen. Beispielsweise gibt es neben der allgemeinen Funktion zur Berechnung der Exponentialfunktion `pow` auch die Funktionen `exp2` und `exp10`, bei denen die Basis auf 2 bzw. 10 festgelegt ist. Diese liefern zügiger Ergebnisse, da sie nicht die verschiedenen Fälle berücksichtigen müssen. Außerdem ist der Zahlenbereich durch Basis und Exponent kleiner weshalb es einfacher ist mit einer hohen Genauigkeit zu rechnen und es werden weniger Register für die Abspeicherung von Zwischenwerten gebraucht. Die Wahl der eingesetzten Bibliotheken sollte abhängig von der benötigten Rechengenauigkeit ausgewählt werden. Neben Funktionen die Ergebnisse sehr exakt berechnen, gibt es auch solche deren Resultate zwar ungenauer aber schneller ermittelt sind.

Die Datentypen `Double` und `Float`

Da Berechnungen mit doppelter Genauigkeit rechenaufwendiger sind sollten diese nur wenn nötig verwendet werden. Konstanten ohne Typ-Suffix interpretiert CUDA als `Double`. Auch bei Berechnungen kann es vorkommen, dass Ergebnisse automatisch von `Double` in `Float` oder von `Char` und `Short` in `Integer` umgewandelt werden.

Die Datentypen Integer und Unsigned Integer

Damit es zu keinem undefinierten Ergebnis kommen kann, sind in der Standard C Programmiersprache *Overflow Semantics* für Unsigned Integer Variablen enthalten. Daher kann der Compiler besser bei der Verwendung von vorzeichenbehafteter Arithmetik optimieren als bei solcher, die nicht vorzeichenbehaftet ist. Aus diesem Grund sollte darauf geachtet werden, auch Variablen, die nur positive Werte speichern als Integer zu deklarieren.

Verwendung von Kontrollstrukturen

Warps, in denen durch If- oder Case- Anweisungen die weitere Befehlssequenz divergiert, sollten unbedingt vermieden werden. Der Multiprozessor behandelt jeden Pfad separat, was im schlechtesten Fall zu einer sequentiellen Ausführung der Threads eines Warps führen kann. Erst nachdem alle Verzweigungen bearbeitet wurden kehrt der Kernel zurück zur parallelen Berechnung. Statt bei einer Verzweigung Threads zu überspringen wird eine Bedingung hinzugefügt. Ist die Voraussetzung der Kontrollstruktur nicht erfüllt, so markiert CUDA den Prozess als *false*. Diese Threads werden zwar weiterhin dem Scheduler zur Bearbeitung freigegeben, schreiben bzw. lesen jedoch keine Daten.

Programmierbeschränkungen

Bei der Programmierung mit CUDA sollten folgende Einschränkungen beachtet werden.

1. Der Kernel gibt die Kontrolle an den CPU Thread zurück, bevor dieser seine Berechnungen beendet hat.
2. Wenn in dem gemeinsamen Speicher mehrere Variable verwendet werden muss der Benutzer das Offset zur Startadresse selbst setzen.
3. Erst nach Threadfence(), TheadBlock() oder Syncthreads() ist es sicher, dass vorherige Schreibzugriffe auf den globalen oder gemeinsamen Speicher auch für die anderen Threads sichtbar sind.

Speicherbelegung

Die Belegung von Speicher durch *cudaMalloc* und *cudaFree* ist eine teure Operation. Daher sollte einmal reservierter Adressraum so oft wie möglich wiederverwendet werden.

Kapitel 3

Die Umsetzung des M4RI in CUDA

Kapitel 3 ist der Hauptbestandteil der Arbeit und enthält alle Informationen zum Programm und zu den erreichten Ergebnissen.

In Kapitel 3.1 wird zunächst der verwendete Algorithmus genau beschrieben. Wie dieser für die Ausführung auf einer Grafikkarte modifiziert wurde und welche Ergebnisse damit erzielt werden konnten, wird in Kapitel 3.2 dargestellt. Dazu informiert dieses Unterkapitel zuerst darüber, welche GPU und API das Programm benutzt. Anschließend wird die Parallelisierung des Algorithmus beschrieben und erklärt, wie der Speicher organisiert und welche Parametereinstellungen gewählt wurden. Im späteren Verlauf konnte das Programm optimiert und somit die Berechnungszeit verbessert werden. Die dafür vorgenommenen Veränderungen und die Analyse der Ergebnisse sind ebenfalls dokumentiert. Da der Zeitvergleich mit Berechnungen auf der CPU zeigt, dass mit GPUs eine schnellere Bearbeitung möglich ist, sollte in Betracht gezogen werden, das Programm so zu ändern, dass die Möglichkeit besteht mit mehreren Grafikkarten zu arbeiten, um auch größere lineare Gleichungssysteme lösen zu können. Die Arbeit enthält zu diesem Ansatz keinen Programmcode, zeigt und diskutiert aber in Kapitel 3.3 mehrere Wege, wie dies theoretisch realisiert werden kann und gibt einen Einstieg in die Umsetzung mit CUDA.

3.1 Die “Method of Four Russians“ Inversion

Es gibt verschiedene Möglichkeiten, ein lineares Gleichungssystem der Form $Ax = b$ zu lösen. Die Methoden werden in iterative und direkte Verfahren unterteilt. Die erste Gruppe ist für Gleichungssysteme über diskreten Mengen nicht geeignet, da bei diesen Algorithmen die Matrix so verändert wird, dass sie sich der Inversen annähert. Ein direktes Verfahren ist die Matrix zu invertieren und anschließend mit b zu multiplizieren. Dabei handelt es sich jedoch um einen sehr rechenintensiven Weg. Bei dem schnelleren Gaußschen Eliminationsverfahren wird das Gleichungssystem in eine Stufenform gebracht und die Lösung durch sukzessives Eliminieren der Unbekannten erreicht. Soll dieses Verfahren als Computerprogramm umgesetzt werden, so empfiehlt es sich, dies mit Hilfe der LR-Zerlegung zu realisieren. Dafür muss die Matrix A des Gleichungssystems durch Faktorisierung in eine linke untere sowie rechte obere Dreiecksmatrix unterteilt und anschließend durch Vorwärts- und Rückwärtseinsetzen die Lösung ermittelt werden.

Das Lösen eines linearen Gleichungssystems mit dem Gauß-Verfahren ist für dicht besetzte Matrizen in der Praxis durch seine kubische Zeitkomplexität zu langsam. Aus diesem Grund besteht das Interesse an Verfahren, die diese Aufgabe mit Hilfe von Computern zügig bewältigen können.

Die “Method of Four Russians“ Bibliothek implementiert verschiedene arithmetische Funktionen auf einer dicht besetzten Matrix über einem Galois-körper mit zwei Elementen. Die Basisidee dieser Algorithmen ist wiederkehrende Berechnungen nur einmal durchzuführen und mehrfach zu verwenden. Dazu werden alle Linearkombinationen vorausberechnet und gespeichert. Auf CPUs kann die “Method of Four Russians“ Inversion, kurz M4RI, das Lösen von linearen Gleichungssystemen bei einer Matrixgröße von 32.000 x 32.000 um einen Faktor von 3,36 beschleunigen (Siehe auch [2]). Die theoretische Komplexität des Algorithmus ist $O(n^3/\log n)$. Dabei muss jedoch bedacht werden, dass n einen sechsstelligen Wert annehmen kann.

Der M4RI Algorithmus über $GF(2)$ besteht im wesentlichen aus 3 Schritten.

Schritt 1: Zuerst wird die reduzierte Stufenform über k Zeilen ab der aktuellen Zeile i berechnet. Dafür wird eine Submatrix mit $3k$ Zeilen betrachtet,

um die Nicht-Singularität zu gewährleisten. Dies ist möglich, da die Wahrscheinlichkeit, dass $3k$ Vektoren der Länge k Rang k haben, sehr hoch ist. Einen entsprechenden Beweis kann in [1] Kapitel 9.4 gefunden werden. Wäre dies nicht gegeben, würde der Algorithmus an dieser Stelle versagen, sobald er auf die erste Submatrix stößt, die nicht in eine obere Stufenform gebracht werden kann.

Schritt 2: Mit den Zeilen in reduzierter Stufenform erhält man k linear unabhängige Vektoren, welche einen Untervektorraum aufspannen. Mit diesen werden nun alle möglichen Linearkombinationen erzeugt. Um die Berechnungen auf CPUs möglichst optimal durchführen zu können, wird eine Gray-Code Tabelle verwendet. Dieser Code hat den Vorteil, dass er sich bei aufeinander folgenden Vektoren nur um eine Stelle unterscheidet. Bei der Berechnung der Tabelle wirkt sich das so aus, dass nur eine Addition vorgenommen werden muss. Das heißt, dass alle 2^k Linearkombinationen aus den k Zeilen in 2^k Schritten berechnet werden können.

Schritt 3: Im letzten Schritt kann die berechnete Tabelle auf alle Zeilen unter bzw. unter und über den k Zeilen angewendet werden. Dafür wird das Präfix ausgelesen und die entsprechende Zeile der Lookup-Tabelle addiert. Um den richtigen Eintrag zu finden gibt es mehrere Möglichkeiten. Zum einen kann die Tabellenzeile mit dem selben Präfix gesucht und zum anderen das Präfix als Dezimalzahl interpretiert, in die zugehörige Gray-Code-Zahl umgerechnet und der entsprechende Eintrag ausgewählt werden. So wird erreicht, dass mit einer einzigen Rechenoperation auf die Zeile an k Stellen eine 0 erzwungen werden kann.

Beispiel für k=4:Schritt 1:
Matrix

```

1000011000...
0100011101...
0010110011...
0001101010...
0000011001...
0000111111...
0000100100...
0000110011...
0000101111...
0000110001...
...

```

→

Schritt 2:
Matrix

```

1000011000...
0100011101...
0010110011...
0001101010...
000010001...
0000010011...
0000001000...
0000000111...
0000101111...
0000110001...
...

```

→

Schritt 3:
Gray-Code
Tabelle

```

000111...
001000...
001111...
010011...
010100...
011011...
011100...
...

```

Schritt 4
Gray-Code Tabelle

```

000111...
001000...
001111...
010011...
010100...
011011...
011100...
...

```

Matrix:

```

1000011000...
0100011101...
0010110011...
0001101010...
0000100001...
0000010011...
0000001000...
0000000111...
0000101111...
0000110001...
...

```

→

```

1000000011...
010000001...
001000001...
000100001...
0000100001...
0000010011...
0000001000...
0000000111...
000000001...
000000001...
...

```

Abbildung 3.1: Die "Method of Four Russians" Inversion.

INPUT: Einen Parameter k und eine Matrix A der Größe $m \times n$.
OUTPUT: Matrix A in Stufenform.

```

begin
  for  $o = 1, k+1, 2k+1, 3k+1, \dots, \min(m,n)$  do
    Führe eine Gauß-Eliminierung auf die Zeilen  $i, i+1, \dots, i+3k-1$ 
    durch, um eine  $k \times k$  Einheitsmatrix in den Spalten
     $a_{i,i} \dots a_{i+k-1,i+k-1}$  zu erhalten.
    Erzeuge eine Gray-Code Tabelle, um alle  $2^k - 1$  vom
    Nullvektor verschiedenen Vektoren aufzuzählen, die sich
    im Untervektorraum befinden, welcher von den Zeilen  $i$ 
    bis  $i+k-1$  aufgespannt wird.
    for jede Zeile  $j = i+3k \dots m$  do
      Lese die Einträge in den  $k$  Spalten  $i, i+1, \dots, i+k-1$  der
      Zeile  $j$  und behandle diese als eine binäre Zahl
       $x$  mit  $k$  Bit.
      Addiere den Eintrag der Gray- Code Tabelle, welcher
      ebenfalls Zahl  $x$  als Präfix hat, zur Zeile Nummer  $j$ .
    end
  end
end
  
```

Abbildung 3.2: M4RI in Pseudocode (Vergleiche [1]).

Werden die ersten beiden Schritte vernachlässigt, kann man sagen, dass dieser Algorithmus k mal schneller arbeitet als der von Gauß, da in einem Durchlauf k Spalten bearbeitet werden können. Außerdem ist es sinnvoll, alle Linearkombinationen vorher zu berechnen, da beispielsweise bei einer Matrix der Größe 32000×32000 und einem k von 8 die Wahrscheinlichkeit hoch ist, dass eine Kombination mehrfach vorkommt.

Obwohl in dem Namen des Algorithmus von einer Inversion die Rede ist, kann er auch für eine Triangulation und eine Back Substitution verwendet werden. Am effektivsten ist es jedoch, wenn der b -Vektor als eine Spalte an die Matrix A gehängt und die obere Dreiecksform gebildet wird. M4RI ist auf eine optimale Berechnung von Matrizen im $GF(2)$ ausgelegt. Durch Modifikation kann er jedoch auch für andere Galoiskörper mit einer kleinen Anzahl von Elementen verwendet werden (Vergleiche [1] und [2]).

3.2 Die Umsetzung von M4RI mit CUDA

Um die Vorteile von Grafikkarten als Co-Prozessoren nutzen zu können müssen andere Aufgaben gelöst und Rahmenbedingungen beachtet werden als bei der Verwendung einer CPU. Die GPU eignet sich, um alle rechenintensiven und vor allem die datenparallelen Teile einer Anwendung zu berechnen. Um sie dafür einsetzen zu können ist es notwendig das Ausgangsproblem so zu formulieren, dass es in Aufgabenteile zerlegt werden kann, die auf verschiedenen Prozessoren unabhängig voneinander bearbeitet werden können. In diesem Kapitel sollen Details des Programms und die Designentscheidungen erläutert werden.

3.2.1 Beschreibung der verwendeten Grafikkarten

In diesem Kapitel wird nur ein Überblick über die Ausstattung der Grafikkarten gegeben. Genauere Informationen über die Funktionsweise der einzelnen Hardwaremodule kann man Kapitel 2.5 und [3] entnehmen.

Für die ersten Tests wurde eine GeForce GTX 295 verwendet. Diese Karte kam im Januar 2009 auf den Markt und hat eine *Compute Capability* von 1.3. Anfangs bestand sie aus zwei einzelnen Platinen, bis Mai 2009 zu einem einfacheren Einzelplatinenlayout gewechselt wurde, auf dem sich beide Grafikkern befanden. Sie verfügt über 2x30 Multiprozessoren mit 2x1400 Transistoren. Der Speicher hat eine Größe von 2x 896 MiB, eine Bandbreite von 2x111.9 GB/s und eine Busbreite von 2x448 Bit. Die Kommunikation zwischen CPU und GPU läuft über ein PCIe 2.0 x16 Bus Interface.

Später wurden die Tests nochmals auf einer GeForce GTX 480 durchgeführt. Diese Karte kam erst März 2010 auf den Markt und hat eine *Compute Capability* von 2.0. Außerdem verfügt sie über eine neue Architektur namens Fermi (Siehe auch Kapitel 2.5). Diese kommt auch auf den Quadro- und Teslakarten zum Einsatz.

Die Grafikprozessoren auf Basis dieser Architektur bestehen primär aus Graphik Processing Clusters, welche neben der Raster Engine auch vier Shader-Cluster bzw. Streaming Multiprozessoren beherbergen. Die Karte verfügt über 15 Multiprozessoren, 480 Cuda-Cores und 3000 Transistoren. Des Weiteren hat der Speicher eine Größe von 1536 MB mit einer Bandbreite von 177.4 GB/s und einer Busbreite von 384 Bit. Für die Kommunikation zwischen GPU und CPU wird derselbe Bus wie bei der GTX 295 verwendet.

3.2.2 Die verwendete API

Das Programm wurde mit C Runtime For CUDA geschrieben, da es gerade für Einsteiger empfehlenswert ist die High-Level API zu verwenden. Sie verfügt über einen Emulationsmodus und enthält Erweiterungen zur Programmiersprache C. Durch diese Erweiterungen ist es einfacher Kernel aufzurufen und zu initialisieren. Die Unterschiede treten nur im Host Code auf und bei Bedarf ist es kein großer Aufwand das Programm für die Verwendung einer Low-Level API umzuschreiben. Bei der neuen Grafikkarte besteht die Möglichkeit beide APIs in einem Programm zu benutzen. Genauere Informationen wurden in Kapitel 2.4 gegeben.

Während meiner Arbeit mit dem Programm erschien die neue Grafikkarte GeForce GTX 480. Bisher war der Emulationsmodus besonders für Anfänger wichtig, da dieser dem Kernel die Möglichkeit bot während des Ablaufs Textausgaben zu machen. Devices ab einer *Compute Capability* von 2.0 verfügen nicht mehr über diesen Modus, erlauben aber Ausgaben auf der Konsole ohne Beschränkungen.

3.2.3 Der umgesetzte Algorithmus

Als Algorithmus wurde die “Method of Four Russians“ Inversion, kurz M4RI, verwendet. Zuerst müssen k Zeilen in eine reduzierte Stufenform gebracht werden. Anschließend wird eine Lookup-Tabelle berechnet und im letzten Schritt auf die gesamte Matrix angewendet. Genauer ist das Verfahren in Kapitel 3.1 erläutert. Der M4RI Algorithmus kann dazu verwendet werden eine obere Dreiecksmatrix zu bilden oder gleich die gesamte Matrix in eine reduzierte Stufenform zu bringen. Das Auflösen eines Dreiecksungleichungssystems ist sehr schlecht parallelisierbar, da der Reihe nach die in den letzten Schritten berechneten Komponenten benötigt werden, um die nächste zu ermitteln. Aus diesem Grund und weil es wenig Mehraufwand bedeutet wird die reduzierte Stufenform gebildet.

Da mit Matrizen über einem Endlichen Körper mit 2 Elementen gearbeitet wird, kann XOR und AND statt Plus und Mal verwendet werden. Des Weiteren tauchen dadurch keine Kommawerte auf, weshalb das Programm mit einer einfachen Genauigkeit rechnet. Durch die verwendeten Operationen ist es möglich die einzelnen Bits in Gruppen zu 32 Stück in einem Unsigned Integer abzulegen. Eine Alternative wäre es einen Datentyp mit 64 Bit zu

verwenden. Da der gemeinsame Speicher jedoch mit Bänken der Größe 32 Bit arbeitet und Bankkonflikte in einem Warp unbedingt vermieden werden sollten (Siehe 2.5.4) wurde entschieden den Datentyp in der selben Größe zu wählen. Im ersten Schritt des Algorithmus müssen mit AND und Shift einzelne Bits ausgewählt werden. Dafür reduziert sich durch die Variablengröße der Aufwand deutlich in den folgenden Schritten.

Bei den durchgeführten Berechnungen handelt es sich zwar um einfache Anweisungen, welche jedoch auf eine große Menge von Daten angewendet werden müssen. Aus diesem Grund machen die Speicherzugriffe einen deutlichen Teil der Berechnungszeit aus. Hier besteht auch ein Problem in der Computerarchitektur. Nicht die gesamte Matrix kann in den Cache des Multiprozessors geladen werden, was zur Folge hat, dass des öfteren auf den Hauptspeicher zugegriffen werden muss. Die Verzögerungszeit dieser Transaktionen ist sehr hoch, was sich negativ auf die Performanz des Programms auswirkt.

Um den Algorithmus mit CUDA umsetzen zu können musste dieser zuvor für die Anwendung auf einer CPU mit GPU als Co-Prozessor abgeändert werden. Die Berechnung der reduzierten Stufenform mit Gauß führt der Host durch. Die restlichen Schritte des M4RI, wie die Erzeugung der Lookup-Tabelle sowie die Anwendung dieser auf alle Daten, finden auf dem Device statt (Siehe auch Abbildung 3.3).

CUDA bietet die Möglichkeit explizit eine Grafikkarte auszuwählen, welche als Co-Prozessor verwendet wird. Als Defaulteinstellung benutzt das Programm das Device mit dem Index 0. Verfügt dieses schon über einen aktiven Kontext dann scheitert der erste Funktionsaufruf, der den Zustand ändern möchte, und erzeugt eine Fehlermeldung.

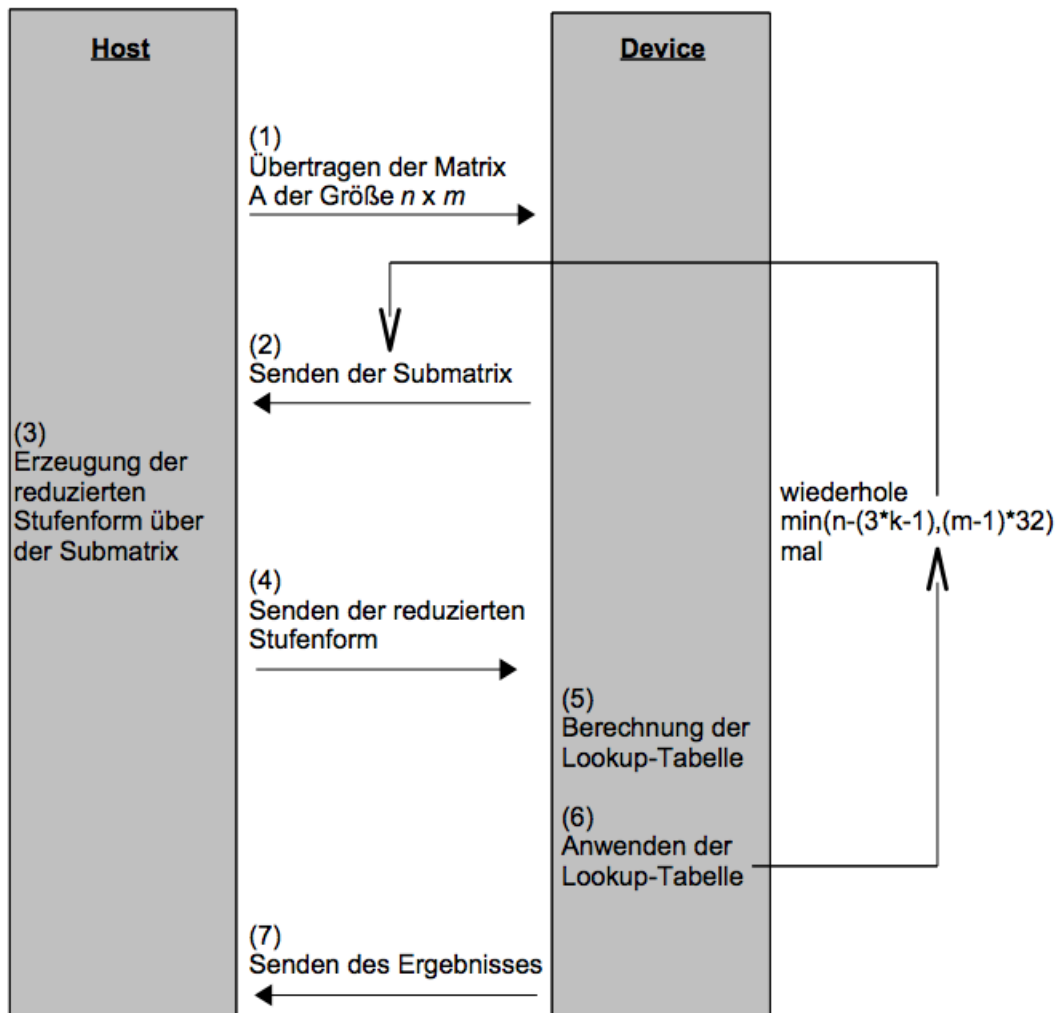


Abbildung 3.3: Kommunikation zwischen Host und Device.

3.2.4 Übertragung der Daten

Bevor die Matrix auf dem Device bearbeitet werden kann, muss diese zuerst über den PCI Express (PCIe) Bus übertragen werden. Ab einer gewissen Größe können die Daten nicht mehr in einem Schritt erzeugt und verschickt werden. Daher findet die Erstellung der Zufallsmatrix und deren Übertragung im Programm stückweise statt und wird auf der Grafikkarte zu einem großen Gleichungssystem zusammen gesetzt.

Durch die Verwendung von Page-Locked (bzw. Pinned) Memory kann das Programm eine höhere Bandbreite erzielen (Siehe auch 2.5.5). Pinned Memory sollte jedoch nicht zu viel benutzt werden, da dies die Performanz des gesamten Systems reduzieren kann. Weil die Datenmenge relativ groß ist wurde daher auf diese Art von Speicher verzichtet.

In den weiteren Schritten werden nur noch $3k-1$ Zeilen an den Host übertragen, um diese dort in eine reduzierte Stufenform zu bringen. Auch hier wird weder Pinned Memory noch ein asynchroner Datentransfer verwendet. Dies liegt darin begründet, dass für die Berechnung der Lookup-Tabelle alle Zeilen übertragen sein müssen und sich somit die Kernelausführung nicht mit dem Transfer überschneiden kann. Für eine stückweise Datenübertragung, um eine asynchrone Bearbeitung zu ermöglichen, wie in Kapitel 2.5.5 beschrieben, ist die Anzahl der versendeten Zeilen zu niedrig. Mit Mapped Pinned Memory würde die Übertragung komplett entfallen. Dies ist allerdings nur in bestimmten Fällen von Vorteil. Da die Daten auf der GPU für Devices mit einer *Compute Capability* von 1.x nicht gecached werden, sollte man bei Mapped Pinned Memory nur einmal lesend und schreibend auf die Werte zugreifen. Des Weiteren reduziert dies ebenfalls die Systemperformanz, da anderen Anwendungen weniger physikalischer Speicher zur Verfügung steht. Aus diesen Gründen wurde diese Art der Datenspeicherung nicht verwendet.

3.2.5 Speicherverwaltung

Die linearen Gleichungen werden als Wertevektor im globalen Speicher der GPU in Form von Unsigned Integern abgelegt.

Das Best Practices Guide von NVIDIA [5] gibt an, dass jeder Zugriff auf den globalen oder lokalen Speicher eine Verzögerungszeit von 400 bis 600 Taktzyklen hat. Dies liegt darin begründet, dass GPUs auf rechenintensive parallele Berechnungen spezialisiert sind und dafür mehr Transistoren zur Verfügung

stehen als für das Cachen von Daten.

Die einzige Möglichkeit diese Latenz zu verbergen ist es genügend unabhängige arithmetische Operationen zu haben, welche bearbeitet werden können, solange ein Thread auf seine Daten wartet.

Bei der "Method of Four Russians" Inversion ist die am meisten verwendete arithmetische Operation XOR, welche nicht sehr rechenintensiv ist. Des Weiteren operiert sie auf Daten, die sich im globalen Speicher befinden. Dies führt zu hohen Verzögerungszeiten, welche nur schwer ausgeglichen werden können. Aus diesem Grund ist es sehr wichtig die Benutzung des globalen Speichers so gut wie möglich zu reduzieren.

Der Gemeinsame Speicher

Die Schwierigkeit am gemeinsamen Speicher ist, dass er die selbe Lebenszeit wie der Block hat, der ihn verwendet, und nur von den Threads gelesen werden kann, welche sich in diesem Block befinden. Außerdem ist er auf 16 KB pro Multiprozessor beschränkt. Aus diesem Grund kann man ihn in dem besonders zeitaufwändigen Teil des Programms, in dem die vorberechnete Tabelle auf die gesamte Matrix angewendet wird, nicht verwenden. Würde jeder Block seinen Teil der Gleichungen in den gemeinsamen Speicher laden, so wäre die Datenmenge zu groß. Außerdem werden alle Werte nur einmal benötigt.

Auch die Lookup-Tabelle in den gemeinsamen Speicher zu übertragen macht wenig Sinn. Zum einen ist auch diese relativ groß und zum anderen wird gegebenenfalls nicht jede Zeile verwendet. Des Weiteren ist es dann notwendig die Datenmatrix auch horizontal in Blöcke zu unterteilen. Warum dies suboptimal ist wird genauer in Kapitel 3.2.6 erläutert.

Allerdings kann bei der Berechnung der Lookup-Tabelle der gemeinsame Speicher verwendet werden. Dafür teilt der Kernel die k Zeilen, welche zuvor auf der CPU in eine reduzierte Stufenform gebracht wurden, horizontal und verteilt die Daten auf die jeweiligen Blöcke. Die Anzahl dieser wird im wesentlichen durch die vorhandene Menge an gemeinsamen Speicher bestimmt, welche jedem Multiprozessor zur Verfügung steht. Die Zwischenergebnisse werden von den Threads in Registern abgelegt und erst die resultierenden Werte selber müssen in den globalen Speicher übertragen werden.

Texturspeicher

Da die Anwendung der Lookup-Tabelle auf das gesamte Gleichungssystem nicht unter Verwendung des gemeinsamen Speichers beschleunigt werden konnte, fanden hier Texturen ihre Anwendung. Dieser *read only* Speicher wird gecached und die Verzögerungszeiten des globalen Speichers treten nur bei einem Cache Miss auf.

Tabelle 3.1: Zeiten mit und ohne Verwendung des Texturspeichers auf der GeForce GTX 295 (in Sek.).

Matrixgröße	Mit Texturspeicher	Ohne Texturspeicher
9.984 x 10.240	1,04	1,08
16.384 x 16.384	3,08	3,35
20.000 x 20.480	5,49	5,96
32.000 x 32.768	19,63	20,97

Die Zeiten in Tabelle 1 stammen von Berechnungen mit einer dicht besetzten Matrix. Da die Verzögerungszeiten nur bei einem Cache Miss auftreten kann sich die Anwendung einer Textur bei dünnbesetzten Matrizen noch stärker positiv auf die Performanz auswirken. Die Wahrscheinlichkeit auf einen Cache Hit erhöht sich dadurch, dass Zeilen mit vielen Nullen häufiger vorkommen als solche mit wenigen.

Einstellungen auf der GTX 480

Die neue GeForce GTX 480 verfügt über eine verbesserte Speicherarchitektur. Zusätzlich zum L1 Cache in jedem Multiprozessor besitzt die Grafikkarte auch einen L2 Cache, welchen sich alle Prozessoren teilen. Der Anwender kann einstellen, ob für die Lesezugriffe vom lokalen und globalen Speicher beide oder nur der L2 Cache benutzt werden soll. Da sich der L1 Cache und der gemeinsame Speicher den On-Chip Speicher teilen kann ebenfalls eingestellt werden, welcher von beiden 48 KB bzw. 16 KB zugewiesen bekommt.

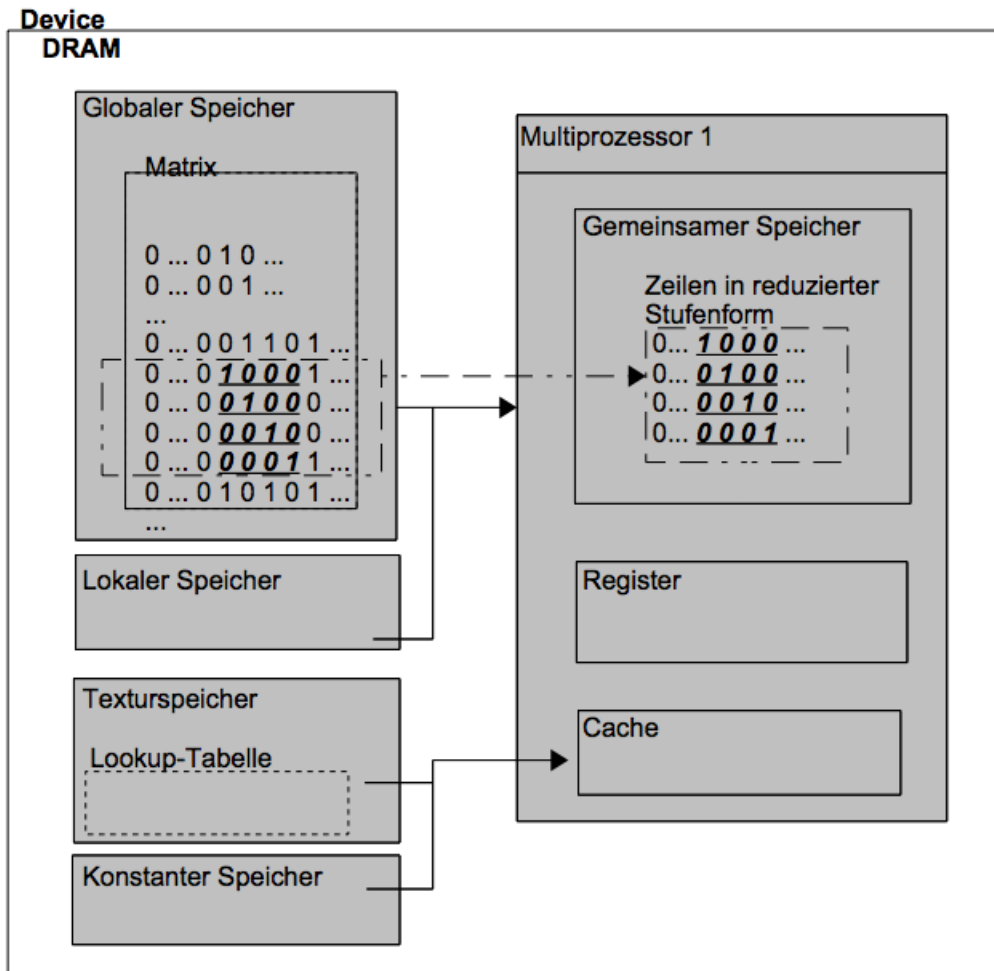


Abbildung 3.4: Speicherverwaltung auf der Grafikkarte.

3.2.6 Parameter

Dieses Kapitel stellt alle Parameter vor und zeigt wie sie die Gesamtzeit beeinflussen.

Unterteilung der Datenmenge in Blöcke und Grids

Da die Matrix mit mehreren Threads bearbeitet werden soll, müssen die Daten zuerst in Blöcke unterteilt werden. Der Kernel, welcher die Linearkombinationen berechnet, splittet die gesamte Matrix vertikal. Anschließend lesen alle Threads eines Blocks die benötigten Spalten der k Zeilen in ihren gemeinsamen Speicher und berechnen ihren Teil der Lookup-Tabelle. Da im besten Fall die Daten für alle Zeilen verwendet werden können findet keine horizontale Unterteilung statt.

Beispiel für $k=4$

	0 0 0 0	0 0 0 0	1 0 1 1	0 1 1 0	1 0 1 0	1 ...
	...					
	0 0 0 0	0 0 0 0	<u>1 0 0 0</u>	1 0 1 0	1 1 1 0	1 ...
	0 0 0 0	0 0 0 0	<u>0 1 0 0</u>	0 1 0 1	1 0 1 0	1 ...
	0 0 0 0	0 0 0 0	<u>0 0 1 0</u>	1 1 0 1	0 1 1 0	1 ...
	0 0 0 0	0 0 0 0	<u>0 0 0 1</u>	1 1 0 1	0 1 1 0	0 ...
	0 0 0 0	0 0 0 0	0 1 0 1	1 0 1 0	1 1 0 1	0 ...
	0 0 0 0	0 0 0 0	1 1 0 1	0 1 1 1	0 1 0 0	1 ...
	...					
Block 0	Block 1	Block 2	Block 3	Block 4		
↓	↓	↓	↓	↓		
0 0 0 0	0 0 0 0	1 0 0 0	1 0 1 0	1 1 1 0		
0 0 0 0	0 0 0 0	0 1 0 0	0 1 0 1	1 0 1 0		
0 0 0 0	0 0 0 0	0 0 1 0	1 1 0 1	0 1 1 0		
0 0 0 0	0 0 0 0	0 0 0 1	1 1 0 1	0 1 1 0		

Abbildung 3.5: Unterteilung der Datenmenge für die Berechnung der Lookup-Tabelle.

Im nächsten Schritt wird die Matrix horizontal in Blöcke gesplittet. Um vertikale Blöcke zu erlauben müsste ein Kernel das Präfix bevor es geändert wird im globalen Speicher sichern, damit es für alle anderen Blöcke sichtbar ist. Wie wir aber in Tabelle 4 sehen können führt dies zu einer schlechteren

Performanz. Aus diesem Grund wird die Matrix nur horizontal unterteilt.

```

Block 0   →   000000000101101101010101...
              00000000110110101010111...

Block 1   →   000000000011101110110110...
              000000001101101101101101...

Block 2   →   000000001000101011101...
              000000000100010110101...

Block 3   →   000000000010110101101...
              000000000001110101100...

Block 4   →   000000000101101011010...
              000000001101011101001...
...

```

Abbildung 3.6: Unterteilung der Datenmenge für die Anwendung der Lookup-Tabelle.

Ein paar Regeln sind sehr wichtig, um die richtige Block- und Gridgröße auszuwählen. Zuerst muss bedacht werden, dass jeder Multiprozessor eine maximale Anzahl von aktiven Threads und Registern hat. Wenn ein Block zu groß ist, in dem er beispielsweise zu viel gemeinsamen Speicher benutzt, wird er nicht bearbeitet. Des Weiteren kann es passieren, dass der Kernel bei der Verwendung von zu vielen Registern die Variablen auslagert, was zu einer schlechten Performanz führt. Es gibt noch weitere Bedingungen. Diese wurden bereits in Kapitel 2.6.1 im Detail beschrieben.

Anhand der Informationen und durch Tests wurden für die verwendeten Grafikkarten optimale Block- und Gridgrößen ermittelt. Diese kann man Tabelle 3.2 und 3.3 entnehmen. Dabei gibt `dimGridy` die Anzahl der Unterteilungen auf der y-Achse und `dimGridx` die Anzahl der Blöcke auf der x-Achse an.

Tabelle 3.2: Parameter auf der GeForce GTX 295.

Matrixgröße	Offset (Schritt 2)	y-Dim. des Grid (Schritt 2)	Offset (Schritt 3)	x-Dim. des Grid (Schritt 3)
9.984 x 10.240	8	40	10	9.984
16.384 x 16.384	8	32	8	16.384
20.000 x 20.480	5	128	10	20.000
32.000 x 32.768	8	128	16	32.000
64.000 x 65.536	8	256	32	64.000

Tabelle 3.3: Parameter auf der GeForce GTX 480.

Matrixgröße	Offset (Schritt 2)	y-Dim. des Grid (Schritt 2)	Offset (Schritt 3)	x-Dim. des Grid (Schritt 3)
9.984 x 10.240	4	20	5	4.992
16.384 x 16.384	8	32	4	8.192
20.000 x 20.480	5	128	5	10.000
32.000 x 32.768	4	128	8	16.000
64.000 x 65.536	8	256	16	32.000

Organisation der Threads

Da nur eine beschränkte Anzahl von aktiven Threads pro Multiprozessor zur Verfügung steht, ist es notwendig, dass von jedem mehr als ein Integer bearbeitet wird. Aus diesem Grund gibt es einen Offsetparameter, welcher angibt wie viele Wörter jeder Prozeß berechnen soll.

Um eine gute Bandbreite zu gewährleisten ist die Reihenfolge in welcher jeder Thread auf die Informationen im globalen Speicher zugreift sehr wichtig. Die Threads werden in Warps ausgeführt und verwaltet (Siehe 2.3.2).

Wie in Kapitel 2.5.4 beschrieben, können unter bestimmten Rahmenbedingungen die Zugriffe auf den globalen Speicher zusammengefasst werden. Aus diesem Grund greifen Threads eines Warps immer auf aufeinander folgende Daten zu. Des Weiteren ist die Anzahl der Wörter einer Zeile und die Distanz zwischen den Elementen, die von einem Thread bearbeitet werden, durch die Warpgröße teilbar.

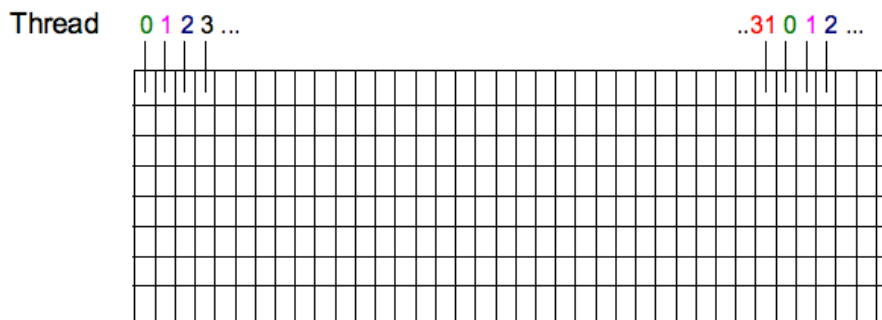


Abbildung 3.7: Zugriff der Threads auf die Matrix.

Tabelle 3.4 zeigt die Zeiten für verschiedene Zugriffsmöglichkeiten. Bei A) sind die Wörter, mit denen ein Thread arbeitet, im globalen Speicher benachbart. Bei B) greifen alle Threads eines Warps auf sequentielle Daten zu (Siehe Abbildung 3.7).

Tabelle 3.4: Zeiten mit und ohne Speichern des Präfixes für sequentielle Wörter A) und solche, die nicht aufeinander folgen B) auf der GeForce GTX 295 (in Sek.).

Matrixgröße	A) Mit Speichern des Präfixes	A) Ohne Speichern des Präfixes	B) Ohne Speichern des Präfixes
9.984 x 10.240	6,46	5,43	0,9
16.384 x 16.384	20,06	14,62	2,47
20.000 x 20.480	37,15	22,24	4,63
32.000 x 32.768	148,18	94,16	13,3

Tabelle 3.4 zeigt, dass die Zeiten für B) besser sind. Das liegt daran, dass der Transfer für einen Warp zusammengefasst werden kann.

In dem Programmcode führt nicht jeder Thread Berechnungen durch da sich im n -ten Durchlauf $k \cdot n$ Spalten bereits in reduzierter Stufenform befinden. Zuvor ermittelt jeder Thread, ob er sich in einem Warp befindet, der noch nicht vollständig bearbeitet wurde. Nur wenn dies der Fall ist führt der Thread einen Zugriff auf den globalen Speicher aus, was zu einer erheblichen Zeitersparnis führt (Siehe Abbildung 3.8).

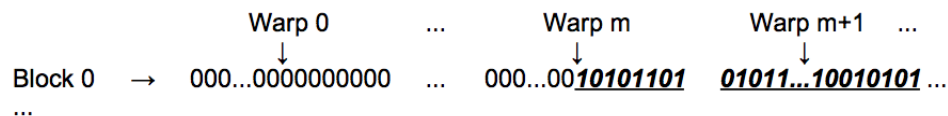


Abbildung 3.8: Zugriff der Warps auf die Datenmenge.

Der Parameter k

Parameter k ist variabel und gibt an, wie groß die vorberechnete Lookup-Tabelle ist. Ihn zu erhöhen steigert auch den Rechenaufwand. Das liegt daran, dass der Algorithmus von Gauß die reduzierte Stufenform über einer größeren Matrix bilden muss. Wenn aber auf der anderen Seite k verringert wird, sind mehr Zugriffe auf den globalen Speicher nötig, was ebenfalls

Tabelle 3.5: Gesamtzeit für verschiedene Größen von k auf der GeForce GTX 295.

Matrixgröße	$k=8$ (in Sek.)	$k=4$ (in Sek.)
9.984 x 10.240	1,04	1,31
16.384 x 16.384	3,08	3,66
20.000 x 20.480	5,49	6,52
32.000 x 32.768	19,63	23,27

die Berechnungszeit erhöht. Tabelle 3.5 zeigt, dass ein größeres k zu einer Performanzverbesserung führt. Allerdings sollte die Lookup-Tabelle in einem angemessenen Verhältnis zur Größe der Gesamtmatrix stehen, da bei einem k von 16 beispielsweise diese über 65536 Zeilen verfügt.

3.2.7 Optimierungen

Die Modifikation der Lookup-Tabelle

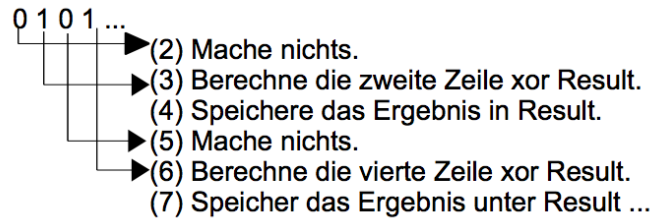
Da der “Method of Four Russians“ Algorithmus für CPUs ausgelegt ist, wurde er nicht eins zu eins übernommen sondern modifiziert, damit er besser parallelisiert werden kann. Die Grundidee, die Eliminierung der Unbekannten zu beschleunigen, indem mehrere Spalten pro Durchlauf bearbeitet werden, wurde jedoch beibehalten.

Die ursprüngliche Version des Algorithmus verwendet als Lookup-Tabelle eine Gray-Code Tabelle. Das heißt, dass die Linearkombinationen in der Reihenfolge der Gray-Code Zahlen erzeugt werden (siehe dazu Kapitel 3.1). Die Idee dahinter ist, dass in jedem Schritt auf der CPU nur eine Berechnung statt finden muss. Dies ist eine sehr wichtige Beschleunigung, da so die Tabelle in 2^k Schritten erzeugt werden kann. Die GPU hat jedoch die Möglichkeit die einzelnen Elemente der Tabelle parallel in $2 \cdot k$ Schritten zu berechnen. Würde auch hier das Prinzip des Gray-Code seine Verwendung finden, so wäre dies nicht möglich, da die einzelnen Zeilen voneinander abhängen würden. Der parallele Algorithmus erzeugt die Tabelle daher so, dass jeder Thread seine vertikale Position bestimmt, die Zeilennummer anschließend in eine binäre Zahl umwandelt und die entsprechende Linearkombination für diese berechnet. Da immer nur eine bestimmte Spaltenmenge pro Block in den gemeinsamen Speicher geladen werden kann arbeiten mehrere Threads

an einer Zeile. Dabei berechnet jeder nur einen Eintrag, damit genügend Prozesse aktiv sind.

Abbildung 3.9 zeigt das verwendete Prinzip für Thread Nummer 5.

(1) Setze Variable Result auf den Nullvektor



(8) Der Ergebnisvektor steht in Result.

Abbildung 3.9: Modifikation der Lookup-Tabelle.

Das Speichern der resultierenden Zeilen an der entsprechenden binären Position hat weitere Vorteile. Dadurch ist es nicht mehr nötig den ausgelesenen Zeilenanfang im nächsten Schritt in eine Gray-Code Zahl umzuwandeln oder in der Tabelle nach dem Eintrag mit dem selben Präfix zu suchen. Bei dieser Version lesen die Threads das Präfix aus, interpretieren es als Integerzahl und nehmen die Zeile mit der selben Nummer (Siehe Abbildung 3.10).

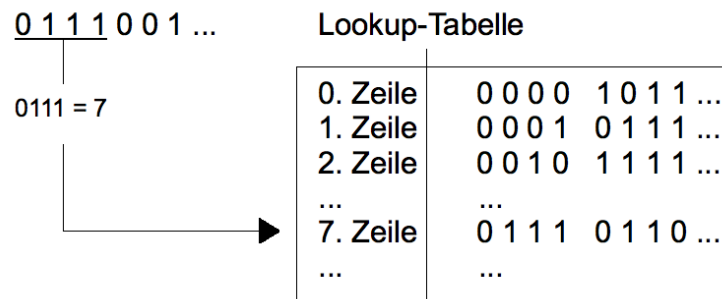


Abbildung 3.10: Modifikation der Lookup-Tabelle.

Durchführung des Gauß-Verfahrens auf der CPU

Testweise wurde die reduzierte Stufenform auf der CPU gebildet und anschließend an die GPU weitergereicht. Obwohl NVIDIA davon abrät zu oft Daten über den PCI-Bus zu schicken, welcher die CPU mit der GPU verbindet, waren die resultierenden Zeiten besser. Dies lässt sich dadurch erklären, dass bei der Verwendung des Gauß-Algorithmus oft auf den globalen Speicher zugegriffen werden muss und die ausgeführten Operationen nicht ausreichend rechenintensiv sind um die auftretenden Verzögerungszeiten zu verbergen.

Tabelle 3.6: Berechnung des Gauß-Algorithmus (Schritt 1) auf dem Host vs. mit einem Kernel auf der GeForce GTX 295.

Matrixgröße	Berechnung auf dem Host (in Sek.)	Berechnung auf dem Device (in Sek.)
9.984 x 10.240	1,04	2,04
16.384 x 16.384	3,08	5,53
20.000 x 20.480	5,49	8,9
32.000 x 32.768	19,63	28,0

Tabelle 3.7: Zeit (in Sek.) bei Berechnung aller Schritte durch die GeForce GTX 295.

Matrixgröße	Gesamtzeit	Schritt 1	Schritt 2	Schritt 3
9.984 x 10.240	2,04	1,34	0,05	0,48
16.384 x 16.384	5,53	3,12	0,28	1,73
20.000 x 20.480	8,9	4,52	0,31	8,9
32.000 x 32.768	28,0	10,6	1,27	14,78

Ein weiterer Vorteil ist, dass das Programm einfacher für die Nutzung mehrerer GPUs umgeschrieben werden kann. In diesem Fall ist es möglich die Ergebnisse der CPU gleichzeitig an mehrere GPUs zu übertragen, die dann voneinander unabhängig die Lookup-Tabelle berechnen und sie auf ihren Teil des linearen Gleichungssystems anwenden. Siehe dazu auch Kapitel 3.3.

3.2.8 Der Programmaufbau

In diesem Kapitel soll mit Hilfe von Pseudocode die grobe Struktur des Programms erklärt werden. Zuerst wird die CPU-Funktion `FourRussians` erläutert und anschließend die Kernel `kernel_gray` und `kernel_gray_xor`, welche auf der GPU ausgeführt werden. Für Testzwecke wird die reduzierte Stufenform über einer von CUDA erzeugten Randommatrix gebildet. Das Übergeben eines Zeigers auf ein schon vorhandenes lineares Gleichungssystem kann später ohne großen Aufwand hinzugefügt werden.

FourRussians(int numrow, int numcol)

ZWECK: Berechnet die reduzierte Stufenform einer Zufallsmatrix.

INPUT: numrow \leftarrow Anzahl der Zeilen.

INPUT: numcol \leftarrow Anzahl der Spalten.

```

begin
  // Variablendeklaration
  sizeelem  $\leftarrow$  Anzahl der benutzten Bits pro Integerwort.
  e  $\leftarrow$  aktuelle Spalte.
  dim1, dim2  $\leftarrow$  Definieren der Dimensionen für die Kernel.
  Setzen der Devicenummer.
  Reservieren des Speicherbereichs d_m auf der Grafikkarte für das
    Gleichungssystem.
  Reservieren des Speicherbereichs d_g auf der Grafikkarte für die
    Lookup-Tabelle.
  Senden des Gleichungssystems an die GPU und ablegen unter d_m.

  for i = 0,k,2k,3k,... min((numrow)-(3*k-1),(numcol-1)*sizeelem) do
    // Berechne die reduzierte Stufenform über einer Submatrix der
      Größe k x k.
    Kopieren der aktuellen 3*k-1 Zeilen vom Device auf den Host.
    for m=0,1,2..k do
      e  $\leftarrow$  (i+m) / sizeelem // aktuelle Spalte
      n  $\leftarrow$  ((e+1)*sizeelem)-(i+m) // aktuelles Bit

      if (In Zeile m an Pos. n in Spalte e befindet sich keine 1) then
        Gehe alle folgenden Zeilen durch und wähle die erste
          aus, deren Bit n in Spalte e eine 1 ist.
        Berechnen der aktuellen Zeile xor der gefundenen
          Zeile.

        for: l=0,1,2,... 3*k-1 do
          if (Das Präfix der Zeile l ist eine 1) then
            Berechne Zeile l xor Masterzeile m um eine 0 zu
              erzwingen.
          end
        end
      end
    end

    e  $\leftarrow$  i/sizeelem //aktuelle Spalte
    Übertragen der 3*k-1 veränderten Zeilen auf das Device.
    Das Device berechnet die Lookup-Tabelle mit Gridgröße dim1 durch
      Aufruf von kenel_gray(d_g, d_m, i).
    Definieren einer Textur über der Lookup-Tabelle.
    int offsetgray  $\leftarrow$  sizeelem - k - (i%sizeelem) //Berechnet das Offset
      in dem Wort.
    Die Lookup-Tabelle wird durch Aufruf von
      kernel_gray_xor(d_m,e,offsetgray,i, offset) mit Gridgröße
      dim2 auf die gesamte Matrix angewendet.
  end
  Kopieren des Ergebnisses von dem Device auf den Host.
  Freigeben der belegten Speicherbereiche auf dem Device.
end

```

Abbildung 3.11: Hauptprogramm.

```

Kernel_gray(unsigned int* d_g*, unsigned int* matr, int start)
ZWECK: Berechnet die Lookup-Tabelle auf der GPU.
INPUT: d_g* ← Zeiger auf die Lookup-Tabelle;
INPUT: matr* ← Zeiger auf die Zeilen in reduzierter Stufenform;
INPUT: start ← aktuelle Zeile;
begin
    tcol ← y-Position des Threads * y-Blockoffset.
    value ← Vektor zum Speichern der Zwischenwerte.

    Reservieren eines gemeinsamen Speicherbereichs unter redForm.
    Übertragen des entsprechenden Teils der k Zeilen in reduzierter
        Stufenform in redForm.
    Value mit 0 initialisieren.

    Threads synchronisieren.

    for m=0,1,2,...k do
        if (An Position m befindet sich eine 1) then
            for l=0, 1, ... y-Blockoffset do
                value[l] xor redForm[k-m-1][tcol +l].
            end
        end
    end

    for l=0,1,2..y-Blockoffset
        Ermitteln des entsprechenden Eintrags in der Lookup-Tabelle über
            Block- und ThreadId sowie Anzahl der Spalten und Zeilen im
            Grid.
        Speichern von value[l] an der zuvor ermittelten Position.
    end
end

```

Abbildung 3.12: Kernel zur Berechnung der Lookup-Tabelle.

Kernel_gray_xor(unsigned int* matr, int e, int off, int i, int offset)

ZWECK: Wendet die Lookup-Tabelle auf die gesamte Matrix an.

INPUT: matr* ← Zeiger auf das Gleichungssystem;

INPUT: e ← aktuelle Spalte;

INPUT: off ← Offset in dem aktuellen Wort;

INPUT: i ← aktuelle Zeile;

INPUT: offset ← Wörter, die pro Thread bearbeitet werden sollen;

begin

Setze den Zeiger matr auf die aktuelle Zeile in der Matrix.

Auslesen des Zeilenpräfixes.

Synchronisieren der Threads.

Speichern der Position des ersten Elements, welches noch nicht 0 ist in start.

if (Präfix > 1) **and**

(der Zeiger zeigt nicht auf eine der 3*k-1 Zeilen, welche bereits auf der CPU bearbeitet wurden) **then**

Ermitteln der Zeile, welche der Thread bearbeiten soll.

Berechnen der Position in der Zeile.

for l=start, start+1, ... offset **do**

Berechne den neuen Eintrag der Matrix durch den alten Wert, xor dem Eintrag der Lookup-Tabelle, welcher durch das Präfix bestimmt wird.

end

end

end

Abbildung 3.13: Kernel zur Anwendung der Lookup-Tabelle.

```

Kernelcalc(unsigned int i, int k, int numrow, unsigned int* row, int e)
INPUT: i ← aktuelles Bit;
INPUT: k ← aktuelle Zeile;
INPUT: numrow ← Anzahl der Zeilen;
INPUT: row* ← Zeiger auf die aktuelle Zeile;
INPUT: e ← aktuelle Spalte;
begin
  for m=0,1,2..k do
    e ← (i+m) / sizeelem // aktuelle Spalte
    n ← ((e+1)*sizeelem)-(i+m) // aktuelles Bit

    if (an Pos. i in Spalte e in Zeile k befindet sich keine 1) then
      Gehe alle folgenden Zeilen durch und wähle die erste aus,
      deren Bit n in Spalte e eine 1 ist.
      Berechne die aktuelle Zeile xor der gefundenen Zeile.
    end
  end
end

GaussKernel(int masterrow, unsigned int* matr, int offset, unsigned int i, int e)
INPUT: masterrow ← aktuelle Zeile;
INPUT: matr* ← Zeiger auf den Anfang der 3*k-1 Zeilen in der Matrix;
INPUT: Offset ← Anzahl der Wörter, die ein Thread bearbeiten soll;
INPUT: i ← aktuelles Bit;
INPUT: e ← aktuelle Spalte;
begin
  Positionieren von matr auf den Eintrag in der Matrix, mit dem der Thread
  arbeiten soll.
  Speichern des Präfixes in calc.
  Warte, bis alle Threads ihre Schreiboperationen beendet haben.

  Berechnen der ersten Spalte, in der eine 1 auftauchen kann und setzen
  von start auf diesen Wert.

  if (Präfix ist nicht 0) then
    Berechnen der aktuellen Zeile in Abhängigkeit der ThreadID und
    speichern des Ergebnisses in i.
    Berechnen der Position in der Masterzeile in Abhängigkeit der
    ThreadID und speichern des Ergebnisses in j.

    for l=start, start+1, .., offset do
      Berechnen des l.ten Elements in Zeile i aus dem alten Wert in
      matr xor dem j.ten Element der aktuellen Masterzeile.
    end
  end
end
end

```

Abbildung 3.14: Kernel zur Berechnung des Gauß-Algorithmus auf dem Device.

3.2.9 Analyse

Um die erreichte Performanz zu untersuchen wurde mit *cudaEvent* die Ausführungszeit einzelner Kernel und Programmschritte gemessen. Vorteil dieser Methode ist, dass die Ergebnisse mit der GPU-Clock erzeugt werden und somit betriebssystemunabhängig sind. Des Weiteren müsste bei Verwendung der CPU-Clock der GPU- mit dem CPU-Thread synchronisiert werden, was ein Drosseln in der GPU Pipeline und somit ein Verfälschen des Resultats zur Folge haben kann. Siehe auch Kapitel 2.4.2.

Tabelle 3.8: Zeit (in Sek.) für jeden Schritt auf der GeForce GTX 295.

Matrixgröße	Gesamtzeit	Schritt 1	Schritt 2	Schritt 3
9.984 x 10.240	0,9	0,25	0,15	0,41
16.384 x 16.384	2,47	0,44	0,35	1,23
20.000 x 20.480	4,63	0,46	0,38	2,26
32.000 x 32.768	13,3	1,74	1,18	8,5

Tabelle 3.9: Zeit (in Sek.) für jeden Schritt auf der GeForce 480.

Matrixgröße	Gesamtzeit	Schritt 1	Schritt 2	Schritt 3
9.984 x 10.240	1,2	0,4	0,05	0,22
16.384 x 16.384	2,9	0,72	0,16	1,07
20.000 x 20.480	4,63	1,45	0,19	1,83
32.000 x 32.768	12,2	2,6	0,35	6,58
64.000 x 65.536	70,74	11,63	1,82	48,58

Die Tabellen 3.8 und 3.9 zeigen, dass der *Bottleneck* des Programms die Verwendung der Lookup-Tabelle ist. Aus diesem Grund wurde im Folgenden hauptsächlich der Kernel für den 3. Schritt untersucht.

Die Bandbreite

Auch wenn die Grafikkarte viele Berechnungen parallel durchführen kann

verliert die Anwendung Zeit durch den Zugriff auf den globalen Speicher. Aus diesem Grund sollte die erreichte Bandbreite gemessen werden. Sie bezeichnet die Rate, in der Daten transportiert werden können und ist einer der wichtigsten Faktoren für die Performanz. Bei allen Änderungen am Code muss deren Auswirkung auf die Bandbreite ebenfalls beachtet werden. Dabei spielt beispielsweise der verwendete Speicher eine Rolle, wie die Daten dort angeordnet sind und in welcher Reihenfolge sie ausgelesen werden. Für mehr Informationen siehe Kapitel 2.5.4.

Um die Performanz genau messen zu können muss die theoretische und tatsächliche Bandbreite ermittelt werden.

Die theoretische Bandbreite der NVIDIA GeForce GTX 295 ist 2x111.9 GB/s. Diese Information kann man der offiziellen NVIDIA Homepage entnehmen. [13]

Um die tatsächliche Bandbreite in GB/s zu berechnen wird die Anzahl der gelesenen und geschriebenen Bytes addiert und anschließend das Ergebnis durch 10^9 und die Zeit in Sekunden geteilt.

Wenn ein Kernel beispielsweise den 3. Schritt auf einer Matrix der Größe $n \times m$ durchführt, so liest er $(n - (3 \cdot k - 1)) \cdot ((m/8) \cdot 2 + 1)$ Bytes und schreibt $(n - (3 \cdot k - 1)) \cdot (m/8)$ Bytes. Die tatsächliche Bandbreite kann daher berechnet werden durch $(n - 3 \cdot k + 1) \cdot ((m/8) \cdot 3 + 1) / 10^9 / time$.

Tabelle 3.10: Tatsächliche Bandbreite des 3. Schritts auf der GeForce GTX 295 (für $k=8$).

Matrixgröße	Zeit (in Sek.)	Bandbreite in GB/s
9.984 x 10.240	0,000438	87,352
16.384 x 16.384	0,00091	110,482
20.000 x 20.480	0,001391	110,3115
32.000 x 32.768	0,003719	105,66

Die erreichte Auslastung

Wie in Kapitel 2.6.1 beschrieben sollte die Anzahl benötigter Ressourcen für jeden Block und die Auslastung des Programms berechnet werden. Mit der Operation `-ptxas-options=-v` wurden folgende Werte ermittelt:

Der Kernel, welcher die Gray-Code Tabelle berechnet, benutzt 14 Register, 276+16 Byte gemeinsamen Speicher und 20 Byte konstanten Speicher. Das bedeutet es wurden insgesamt 276 Byte gemeinsamer Speicher benutzt wobei das System 16 Byte belegt hat.

Der Kernel, welcher den nächsten Schritt ausführt, benutzt 12 Register, 24+16 Byte gemeinsamen Speicher und 8 Byte konstanten Speicher.

Mit diesen Werten ist es möglich die Auslastung zu berechnen. Wie in Tabelle 3.2 und 3.3 Kapitel 3.2.6 aufgelistet wurde die Datenmenge in eine bestimmte Anzahl Blöcke unterteilt. Im ersten Schritt wird die Gesamtzahl aktiver Warps pro Block berechnet und anschließend die Menge von verwendeten Ressourcen und somit die Auslastung ermittelt.

Die Tabelle 3.11 listet diese für den zweiten Kernel auf. Die Werte in den Klammern geben an, wie viel der jeweiligen Ressourcen vom Kernel benötigt werden. Die tatsächlich reservierte Menge ist meistens höher und hängt von der Granularität ab, mit der der Speicher und die Register bereit gestellt werden.

Da einem Multiprozessor bei einer *Compute Capability* von 1.3 insgesamt

Tabelle 3.11: Menge der benötigten Ressourcen pro Block auf der GeForce GTX 295.

Matrixgröße	Threads	Warps	Register	Gemeinsamer Speicher	Konstanter Speicher
9.984 x 10.240	32	2(1)	1024 (384) B	512 (24) B	512 B
16.384 x 16.384	64	2(2)	1024 (768) B	512 (24) B	512 B
20.000 x 20.480	64	2(2)	1024 (768) B	512 (24) B	512 B
32.000 x 32.768	64	2(2)	1024 (768) B	512 (24) B	512 B
64.000 x 65.536	64	2(2)	1024 (768) B	512 (24) B	512 B

*Die Zahlen in Klammern geben die tatsächlich benötigte Menge an.

nur 16 KB gemeinsamen Speicher sowie 16000 Register zur Verfügung stehen können 15 Blöcke in einen Multiprozessor geladen werden. Da dadurch 960 Threads zeitgleich aktiv sind beträgt die Auslastung 93,75 %, was zeigt, dass die Parameter optimal gewählt sind.

3.2.10 Zeitvergleiche mit Berechnungen auf der CPU

Die Gesamtzeit auf der GPU bezieht sich auf eine Zufallsmatrix, welche durch CUDA erstellt wird. Die Referenzwerte auf der CPU stammen von der Homepage <http://m4ri.sagemath.org/performance.html> [4]. Diese Seite beschäftigt sich mit der Berechnung von linearen Gleichungssystemen über GF(2). Seit der Veröffentlichung des M4RI durch Gregory Bard arbeitet das "SAGE/M4RI-Team" an einer schnellen Umsetzung dieses Algorithmus auf der CPU und veröffentlicht in regelmäßigen Abständen die aktuellen Ergebnisse. Wie man an den Zeiten in Tabelle 3.12 und 3.13 erkennen kann, ist es möglich ein lineares Gleichungssystem auf einer Grafikkarte schneller zu berechnen als auf der CPU. Das CUDA-Programm arbeitet vor allem mit großen Matrizen sehr effektiv, da bei einer großen Datenmenge die GPU besser ausgelastet ist.

Tabelle 3.12: Zeiten auf der GeForce GTX 295 und GeForce GTX 480.

Matrixgröße	GeForce GTX 295	GeForce GTX 480
9.984 x 10.240	0,9 Sek.	1,2 Sek.
16.384 x 16.384	2,47 Sek.	2,9 Sek.
20.000 x 20.480	4,63 Sek.	4,63 Sek.
32.000 x 32.768	13,3 Sek.	12,2 Sek.
64.000 x 65.536	-	70,74 Sek.

Tabelle 3.13: Zeiten auf der CPU [6].

Matrix Dimension	M4RI/M4RI 20090105 ¹	M4RI/M4RI 20100817 ²
10.000 x 10.000	1,532	1,050
16.384 x 16.384	6,597	3,890
20.000 x 20.000	12,031	7,250
32.000 x 32.000	40,768	22,560
64.000 x 64.000	241,017	124,480

[1] 64-bit Debian/GNU Linux, 2.33 Ghz Core2Duo (Macbook Pro, 2nd. Gen.)

[2] 64-bit Debian/GNU Linux, 2.6 Ghz Intel i7 (Macbook Pro 6,2)

3.3 Multiple GPUs

Lineare Gleichungssysteme über $\text{GF}(2)$, die im Rahmen einer kryptographischen Attacke gelöst werden müssen, können sehr groß ausfallen. Neben der Tatsache, dass der verwendete Algorithmus viel Zeit benötigt, da er auf einer großen Menge von Daten arbeitet, kann das Gleichungssystem auch so groß sein, dass es nicht auf einer einzigen Grafikkarte gespeichert werden kann. Um in diesem Fall den Datentransfer zwischen CPU und GPU möglichst gering zu halten, ist es sinnvoll das Gleichungssystem auf mehrere Devices zu verteilen. Dabei sollte beachtet werden, dass möglichst wenig redundante Daten entstehen. Bei Berechnungen auf verteilten Speichern kann sehr gut das Master/Slave-Prinzip verwendet werden, um den Zugriff auf die gemeinsamen Ressourcen hierarchisch zu verwalten. Dabei ist ein Teilnehmer der Master und die anderen sind Slaves. Diese Zuordnung ist nicht statisch und kann sich während der Laufzeit des Programms mehrfach ändern. Der Masterknoten ist der aktive Teil und hat die Aufgabe zu regeln und zu steuern, während die Slaveknoten als passive Partner lediglich Anweisungen ausführen (Vergleiche [18] Kapitel 5). Angewendet auf das Rechnen mit mehreren GPUs würde das bedeuten, dass eine Grafikkarte Master ist und bestimmt, welche Daten an die Slaves übergeben werden.

3.3.1 Theoretische Umsetzung

Auf der parallelisierten “Method of Four Russians“ Inversion kann das Master/Slave-Prinzip theoretisch auf 2 Wegen umgesetzt werden.

Vertikale Unterteilung

Das Gleichungssystem kann vertikal in mehrere Submatrizen unterteilt und auf verschiedenen Grafikkarten abgelegt werden. Das Device, auf welchem sich das aktuelle Zeilenpräfix befindet, übernimmt die Funktion des Masters. Dieser berechnet die reduzierte Stufenform und teilt seine Schritte den anderen Devices mit, damit diese die selben Berechnungen auf ihrem Teil der Zeilen ausführen können. Jede Grafikkarte kann ihre Tabelle unabhängig voneinander berechnen. Bei der Anwendung muss jedoch ein Informationsaustausch stattfinden. Der Master erkennt am Präfix, welche Zeile aus der Lookup-Tabelle verwendet werden muss und teilt dies anschließend den anderen Devices mit.

	Slave	Master	Slave	...	Slave
Matrix:	1 0 0 0	1 0 1 1	1 0 0 1	...	1 0 1 1
	0 1 0 0	0 1 1 1	0 0 1 0	...	1 1 1 1
	0 0 1 0	1 0 1 0	1 1 1 1	...	0 0 0 0
	0 0 0 1	0 0 1 1	0 0 0 0	...	1 0 1 0
	0 0 0 0	1 0 0 0	1 1 0 1	...	0 1 1 0
	0 0 0 0	0 1 0 0	0 1 1 1	...	1 0 0 1
	0 0 0 0	0 0 1 0	1 0 1 0	...	1 1 0 0
	0 0 0 0	0 0 0 1	0 1 1 0	...	0 0 1 1
	0 0 0 0	1 0 1 1 →	0 1 1 0	...	1 1 1 1

	0 0 0 0	1 1 1 0 →	1 0 1 0	...	1 0 1 0
 Lookup-Tabelle:					
	0 0 0 0	0 0 0 1	0 1 1 1	...	1 1 1 1
	0 0 0 0	0 0 1 0	0 1 1 0	...	0 1 1 1
	0 0 0 0	0 0 1 1	1 0 1 0	...	1 0 1 0

Abbildung 3.15: Vertikale Unterteilung der Datenmenge auf verschiedene Devices.

Horizontale Unterteilung

Bei dem 2. Weg wird das Gleichungssystem horizontal in mehrere Submatrizen unterteilt und diese auf verschiedenen Grafikkarten abgelegt. Das Device, auf dem sich die aktuell verwendeten k Zeilen befinden, übernimmt die Funktion des Masters. Dieser muss die reduzierte Stufenform bilden und das Ergebnis an alle anderen Devices übertragen. Diese können anschließend unabhängig voneinander die Lookup-Tabelle berechnen und sie auf ihren Teil des Gleichungssystems anwenden.



Abbildung 3.16: Horizontale Unterteilung der Datenmenge auf verschiedene Devices.

Bewertung

Bei einer vertikalen Unterteilung muss sehr viel Kommunikation zwischen Master und Slave statt finden. Grafikkarten können nicht direkt sondern nur über die CPU Daten austauschen, was den Algorithmus erheblich verlangsamt. Des Weiteren reduziert sich die Anzahl der verwendeten Devices während der Ausführung des Algorithmus. Sobald sich alle Spalten einer Grafikkarte in reduzierter Stufenform befinden, gibt es nichts, was diese noch berechnen muss.

Dagegen bietet sich eine horizontale Unterteilung, auch durch die aktuelle Programmierung, an. Die reduzierte Stufenform wird ohnehin auf der CPU gebildet und von dort aus an die GPU geschickt wodurch es möglich ist diese gleich an mehrere GPUs zu übertragen. Dadurch wird keine GPU-GPU Kommunikation nötig. Des Weiteren kann der Transfer der Zeilen in einer einzigen Transaktion statt finden, womit laut NVIDIA die beste Performanz erreicht wird [5]. Der Master hat die Funktion, den Slaves seine Zeilen zur Verfügung zu stellen. Der einzige Nachteil ist, dass die Lookup-Tabelle redundant erzeugt wird. Da die Zeitanalyse des Algorithmus jedoch gezeigt hat, dass deren Berechnung nur einen sehr geringen Teil der Gesamtzeit ausmacht, halte ich eine horizontale Unterteilung für den besseren Ansatz.

3.3.2 Umsetzung mit CUDA

CUDA ist komplett orthogonal zum CPU-Thread Management bzw. zu Message Passing APIs. Daher unterscheidet sich die Programmierung einer Multi-GPU Anwendung nicht von den Programmen, die mehrere Kerne oder Prozessorsockel verwenden. Es ist möglich einen existierenden multithreaded CPU-Code einfach um eine GPU-Beschleunigung zu erweitern, unabhängig davon, ob leichtgewichtige oder schwergewichtige Threads benutzt werden. Das vorhandene Programm wird nur um den Teil, in dem die Daten an die GPU geschickt werden, ergänzt und die Kommunikation bleibt unverändert.

Es gibt verschiedene Standards für Programmierschnittstellen, die den Nachrichtenaustausch bei parallelen Berechnungen beschreiben. Dazu gehört beispielsweise *OpenMP* und *threads* für leichtgewichtige, sowie *MPI* für schwergewichtige Threads.

Die *Native POSIX Thread Library* kurz *NPTL* ist eine Threading-Bibliothek

für Linux. Sie ermöglicht Linux-Programmen die Verwendung von *POSIX-Threads*, kurz *pthread*, unter Verwendung der *GNU C Library*. Der *OpenMP-Standard* dagegen wird bevorzugt auf Systemen mit einem gemeinsamen Hauptspeicher, den Shared-Memory-Maschinen, verwendet (Siehe auch Kapitel 2.1). Bei dem so genannten *Message Passing Interface* kurz *MPI* für schwergewichtige Threads handelt es sich um einen Standard, der eine Programmierschnittstelle für parallele Berechnungen auf verteilten Computersystemen definiert. Es gibt auch Anwendungen, zum Beispiel bei den Supercomputern, wo *OpenMP* und *MPI* gemeinsam zum Einsatz kommen.

Zwar ist es möglich, dass verschiedene Host-Threads Code auf dem selben Device ausführen, wie bei einem CPU-Thread kann jedoch auf einer GPU gleichzeitig nur ein Kontext aktiv sein. Das hat zur Folge, dass zu einem Zeitpunkt das Device von nur einem Prozess genutzt werden kann. Mit der ersten Funktion, welche den Zustand der GPU zum Beispiel durch reservieren eines Speicherbereichs ändert, ist ein Kontext angelegt und kann erst wieder durch `cudaThreadExit()` oder Beendigung des zugehörigen CPU-Threads gelöscht werden. Greifen mehrere Prozesse auf ein Device zu, so wird der GPU Driver Manager verwendet, um zwischen ihnen zu wechseln. Dabei muss jedoch beachtet werden, dass der Speicher, der von einem Kontext belegt wurde, auch nur von diesem benutzt werden kann.

Hat sich der Anwender bei der Wahl des Interfaces für die CUDA Driver API entschieden, so besteht die Möglichkeit, dass ein einzelner CPU-Thread durch pushing und popping mehrere Kontexte und somit mehrere GPUs verwalten kann. Die C Runtime for CUDA API hingegen erlaubt nur einen Kontext pro Host Thread. Wenn das Programm in diesem Fall mit p Devices arbeiten soll, so werden auch p leichtgewichtige oder schwergewichtige CPU-Threads benötigt.

Eine Möglichkeit dies umzusetzen ist, dass jeder Thread fest mit einer GPU arbeitet. In diesem Fall ist es beispielsweise möglich den GPU Index durch die ID des Prozesses abzuleiten. Bei *OpenMP* kann direkt auf die ThreadID zugegriffen werden und bei *MPI* verfügt jeder Prozess über eine Nummer, welche als Rang bezeichnet wird. Diese ist eindeutig, solange sich alle Threads auf einem gemeinsamen Host Knoten befinden.

Ab CUDA 2.2 unterstützen Linux Driver einen alternativen einfacheren Weg, jeder GPU einen aktiven Kontext zuzuweisen. Der Administrator kann

einen Exclusive Mode mit dem *System Management Interface Tool*, kurz *SMI*, auswählen, welches mit dem Driver mitgeliefert wird. Wenn der Anwender normalerweise nicht explizit mit `cudaSetDevice()` eine Grafikkarte auswählt, dann wird der Kontext auf Device 0 erstellt. In dem Exclusive Mode greift das Programm auf die nächste Grafikkarte ohne aktiven Kontext zu.

Die gesamte Kommunikation, welche zwischen GPUs statt findet läuft über den Host. Das Device und sein zugehöriger CPU-Thread kommunizieren mit *memcpys*, während die CPU-Threads untereinander auf die selbe Weise Daten austauschen, wie sie es auch ohne Verwendung von Grafikkarten tun würden. Für die Performanz ist es wichtig, dass die Kommunikation möglichst optimal organisiert wird. Dabei sollte bedacht werden, dass die CPU-GPU Kommunikation unabhängig von der CPU-CPU Kommunikation durchgeführt werden kann.

Leichtgewichtige CPU-Threads tauschen Daten am effizientesten über den gemeinsamen Speicher. Ein Kommunikationsszenario zwischen zwei GPUs sieht so aus, dass zuerst die GPU die Daten an den zugehörigen CPU-Thread überträgt, dieser lädt sie anschließend in den gemeinsamen Speicher, dort werden sie von einem anderen CPU-Thread gelesen, welcher sie dann an die GPU überträgt, mit der er arbeitet.

Bei schwergewichtigen Threads kann MPI verwendet werden. Sobald Daten von der GPU an die CPU übertragen wurden, können die entsprechenden MPI Funktionen aufgerufen werden, um die Kommunikation zwischen den einzelnen CPU-Prozessen zu organisieren.

Sobald Code Syntax der C Runtime for CUDA API enthält, muss er mit `nvcc` kompiliert werden. Der Compiler verwendet bei Linux `gcc` und bei Windows *Microsoft Visual C++* für den Teil, bei dem es sich nicht um GPU-Code handelt und es ist möglich, diesem durch die Operation `-Xcompiler` Optionen zu übergeben. Beispielsweise würde `nvcc -Xcompiler/openmp` aufgerufen werden, um die `-fopenmp` Operation an den *Host gcc-Compiler* weiterzureichen. *MPI* Anwendungen werden meistens mit `mpicc` kompiliert. Es ist zwar möglich alle notwendigen Flags auch hier durch `nvcc` zu übergeben, allerdings empfiehlt es sich, den Code in verschiedene Object Files zu schreiben und unabhängig voneinander zu kompilieren.

Mehr Information über die Verwendung Multipler GPUs kann man Kapitel 8 des Best Practices Guides von NVIDIA entnehmen [5].

Kapitel 4

Fazit und Ausblick

Diese Arbeit beschäftigte sich mit der Frage, ob das Lösen eines linearen Gleichungssystems im $GF(2)$ mit Hilfe von Grafikkarten zu kürzeren Laufzeiten führt als bei der Verwendung von CPUs. Als Algorithmus wurde die “Method of Four Russians“ Inversion verwendet, da dieser effizienter die reduzierte Stufenform bildet als das Gaußsche Eliminationsverfahren. Das Programm wurde mit Hilfe von CUDA geschrieben und anschließend auf den GPUs GeForce GTX 295 und GeForce GTX 480 ausgeführt. Aus diesem Grund bietet diese Arbeit neben dem Programmdesign auch einen Überblick über den Aufbau von Grafikkarten der Firma NVIDIA und eine Einführung in die verwendete parallele Berechnungsarchitektur.

GPUs sind besonders geeignet für “Massiv Parallel Data Problems“. Dabei handelt es sich um Algorithmen, die eine große Menge von Daten bearbeiten und deren Werte eigenständig berechnet werden können. Obwohl beim Lösen linearer Gleichungssysteme die Berechnungen von dem jeweiligen Zeilenpräfix abhängen, konnten die Bearbeitungsblöcke so gewählt werden, dass sie autonom arbeiten. Jede Zeile kann durch Threads unabhängig voneinander berechnet werden und auch die Erzeugung der Lookup-Tabelle profitiert von der parallelen Verarbeitung. Um eine parallele Durchführung des M4RI zu erreichen musste der Algorithmus zuvor für die Anwendung auf einer CPU mit GPU als Co-Prozessor abgeändert werden. Weitere Modifikationen an dem Verfahren wie etwa bei der Berechnung der Lookup-Tabelle konnten die Performanz zusätzlich steigern. Alle dafür vorgenommenen Veränderungen am Algorithmus sind in Kapitel 3.2 dokumentiert. Auch andere Faktoren, welche die Laufzeit beeinflussen, wie beispielsweise die verwendeten Parameter

und Speicher sowie die Organisation der Threads, sind mit den zugehörigen Zeitmessungen aufgeführt. Die Auswertung der Berechnungszeit zeigt, dass die Bandbreite gut ausgenutzt wird und die Parameter optimal gewählt sind. Die ermittelte Auslastung der Multiprozessoren ergab einen Wert von 93,75% (Siehe Kapitel 3.2.9).

Als Referenzwerte für die Ausführung der “Method of Four Russians“ Inversion auf der CPU dienten die Ergebnisse des “SAGE/M4RI-Teams“. Ein Vergleich in Kapitel 3.2.10 zeigte, dass Grafikkarten um einen Faktor von bis zu 1.87 schneller eine reduzierte Stufenform erzeugen können.

Ausblick

Wie in Kapitel 3.3 beschrieben, kann das Programm so erweitert werden, dass es das lineare Gleichungssystem mit mehr als einer Grafikkarte löst. Die Modifikation, Matrizen auf mehrere GPUs zu verteilen, ist vor allem dann sinnvoll, wenn die Datenmenge für den globalen Speicher einer einzelnen Grafikkarte zu groß ist. Des Weiteren haben bei einer horizontalen Unterteilung des Gleichungssystems die einzelnen Grafikkarten die Möglichkeit ihre Berechnungen nahezu unabhängig voneinander durchzuführen. Es findet keine GPU-GPU- sondern lediglich eine CPU-CPU- und GPU-CPU- Kommunikation statt.

Auf Grafikkarten wird der Großteil der Transistoren für Rechenaufgaben eingesetzt, anstatt wie bei der CPU für Steuerungsaufgaben und Datencaching. Die neueren GPUs von NVIDIA verfügen jedoch über eine verbesserte Speicherarchitektur, die ebenfalls einen Cache für Zugriffe auf den globalen und lokalen Speicher zur Verfügung stellt. Damit wird das Problem adressiert, dass das Laden und Speichern von Daten in den globalen Speicher der GPU zu hohen Verzögerungszeiten führt. Eine weitere Möglichkeit auftretende Latenzen auszugleichen ist es, unabhängige Berechnungen, die mit einem On-Chip-Speicher arbeiten, in der Wartezeit durchzuführen. Dies ist ein wesentlicher Vorteil von Grafikkarten und kommt am besten bei rechenintensiven Algorithmen zur Geltung. Aus diesem Grund könnte in Betracht gezogen werden GPUs zu verwenden, um auch lineare Gleichungssysteme über anderen endlichen Körpern zu lösen, deren Berechnung mehr Aufwand erfordert.

Da das Programm mit CUDA geschrieben wurde und nur auf Grafikkarten der Firma NVIDIA läuft bleibt die weitere Entwicklung des OpenCL-Compilers interessant. Sobald dieser die Hardwareeigenschaften verschiedener Grafikkarten besser ausnutzen und dadurch ähnlich gute Laufzeiten erreicht werden können bietet es sich an diese plattformunabhängige Programmierschnittstelle zu verwenden um das Programm auch auf anderen Grafikkarten ausführbar zu machen.

Anhang A

Programmcode

```
                                /Users/ddemirel/M4R1/FourRussians20.h


---


#ifdef _FOURRUSSIANS20_H_
#define _FOURRUSSIANS20_H_

#define hk 8 //4
#define hcol 320 //512 //640 //1024 // 2048
#define offset2 8 //8 //5 //8 //8
#define BlockDimCalcGray 40 //32 //128 //128 //256

#endif
```

/Users/ddemirel/M4R1/FourRussians20.cu

```

/*
 * Copyright 1993-2009 NVIDIA Corporation. All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation and
 * any modifications thereto. Any use, reproduction, disclosure, or distribution
 * of this software and related documentation without an express license
 * agreement from NVIDIA Corporation is strictly prohibited.
 *
 */

/* Function: calculate reduced row echelon form using M4RI.
   Code by: Denise Demirel
   Version: 2.1
   Date: 06.2010
*/

// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes, kernels
#include <FourRussians20_kernel.cu>
#include "FourRussians20.h"

const int pieces = 32;
const int row = 9984;
const int offset = 10; // number of words on thread execute. Appendix 2.
const int knumrow = 3*hk-1;
const int dimGridxgray = 9984;
const int devicenumber = 2;

//const int dimGridxgaussk = 1;
//const int dimGridygaussk = 1; only if Gauss is calculated by kernel

////////////////////////////////////
void
FourRussians(int numrow, int numcol);
////////////////////////////////////

// Functions
////////////////////////////////////

int
main( int argc, char** argv)
{
    clock_t clock1 = clock();
    FourRussians(row,hcol);
    clock_t clock2 = clock()-clock1;

```

```

/Users/ddemirel/M4R1/FourRussians20.cu
printf("Zeit in Sekunden: %f\n", (double)clock2 / CLOCKS_PER_SEC);
printf("Clocks: %f\n", (double)clock2);
}

void
FourRussians(int numrow, int numcol)
/* Funktion: Calculates reduced row echelon form
parameter:  numrow: number rows
            numcol: number columns
*/
{
    // Declaration
    uint sizeelem = 32; //number of used bits per word (Appendix 1)
    int e = 0; // active column
    unsigned int gauss[knumrow][hcol];

    // Size of one row
    unsigned int mem_size_Gauss = sizeof(unsigned int) * ((knumrow)*numcol);
    int i;

    // Definition of kernel dimensions

    //for Gauss k x k
    //dim3 dimBlock((int)(knumrow/dimGridxgaussk),
    //              (int)((numcol/offset)/dimGridygaussk));

    // kernel for pre-calculation of the table
    dim3 dimBlock2((int)(numrow/dimGridxgray), (int)(numcol/offset));

    // kernel for use of table on matrix
    dim3 dimBlockgraycalc((int)(1<<hk), (int)(hcol/offset2)/BlockDimCalcGray);

    //dim3 dimGridgaussk(dimGridxgaussk, dimGridygaussk);
    dim3 dimGridgray(dimGridxgray, 1);
    dim3 dimGridgraycalc(1, BlockDimCalcGray);

    //set device
    cudaSetDevice(devicenumber);

    //allocate memory on device
    unsigned int size_M = (numrow) * numcol;
    unsigned int mem_size_M = sizeof(unsigned int) * size_M; //Size of one row

    unsigned int size_grey = ((1<<hk) * numcol);
    unsigned int mem_size_grey = sizeof(unsigned int) * size_grey;
    unsigned int* d_g;
    cudaMalloc((void**)&d_g, mem_size_grey);

    unsigned int* d_m;
    cudaMalloc((void**)&d_m, mem_size_M);
    unsigned int dataorg[Row/pieces][hcol];

```

 /Users/ddemirel/M4R1/FourRussians20.cu

```

for (int l=0; l<pieces;l++){
  for (int i=0; i<(row/pieces); i++){
    for (int j=0; j< hcol-1; j++)
      // random number with 32 Bit
      dataorg[i][j]= (uint)((random()*(l+1)+1)*
                          pow((double)-1,(int)(random()%2)));
  }

  // send Matrix to device
  cudaMemcpy(d_m+l*(hcol*(numrow/pieces)),
            dataorg, mem_size_M/pieces, cudaMemcpyHostToDevice);
}

// calculate row echelon form of size k x k
for (i = 0; i < min((numrow)-knumrow,(numcol-1)*sizeelem); i=i+hk){
  cudaMemcpy(gauss, d_m+(i*numcol), mem_size_Gauss, cudaMemcpyDeviceToHost);

  for (int m=0; (m < hk); m++)
  {
    e = (i+m) / sizeelem; //master column
    int n = ((e+1)*sizeelem)-(i+m); //master bit

    int notchanged = 1;
    int count = m+1;
    int pos = 1 << (n-1); // 2^i
    if (((gauss[m][e]) & pos) != pos){

      do {
        if (((gauss[count][e]) & pos) == pos){
          notchanged = 0;
          for (int o= 0; o < hcol; o++){
            gauss[m][o] = gauss[m][o] ^ gauss[count][o] ;
          }
        }
        count++;
      }while ((notchanged) and (count < knumrow));

      if (notchanged) {
        count = 0;
      }
    }

    for (int o=0; o<knumrow; o++){
      if (((gauss[o][0+e]& 1<<(n-1))>>(n-1)) and (o != m))
        for (int p=0; p<numcol; p++)
          gauss[o][p]=gauss[o][p]^gauss[m][p];
    }

    //Kernelcalc<<<1,1>>>(n, m, knumrow, d_m+i*numcol, e);

    // Calculate Matrix
  }
}

```

```

/Users/ddemirel/M4R1/FourRussians20.cu
//GaussKernel<<<dimGridgaussk, dimBlock>>>(m, d_m+i*numcol,offset,n,e);
}

e=i/sizeelem;
cudaMemcpy(d_m+(i*numcol), gauss, mem_size_Gauss, cudaMemcpyHostToDevice);

kernel_gray<<<dimGridgraycalc, dimBlockgraycalc>>>(d_g, d_m, i);

cudaBindTexture(0,texRef,d_g, sizeof(unsigned int)* ((1<<hk) * hcol));

int offsetgray = sizeelem - hk - (i*sizeelem); //offset in one word
kernel_gray_xor<<<dimGridgray, dimBlock2>>>(d_m,e,offsetgray,i, offset);
}

// deallocate memory on device
cudaFree(d_g);

for (int p=0;p<pieces;p++){
// copy matrix back to host
cudaMemcpy(dataorg, d_m+(p*(row/pieces*hcol)),
            mem_size_M/pieces, cudaMemcpyDeviceToHost);

for (int i=0; i<(numrow/pieces); i++) // show matrix on the screen.
{
printf(" %u .row: %u", i+p*(row/pieces),dataorg[i][0]);
for (int l=1;l< numcol;l++){
printf(" %u",dataorg[i][l]);
}
printf("\n");
}

}
cudaFree(d_m); // deallocate memoryspace on the device.
}

/*
Appendix 1: To reduce the number of active bits make it easier to check the right
functionality of the program.
Appendix 2: If you want to change the dimension of the grid or the offset make sure
that the matrix-size is divisible without remainder by the dimension.
Otherwise this can occurs errors.
*/

```

```

/Users/ddemirel/M4R1/FourRussians20_kernel.cu


---


/*
 * Copyright 1993-2009 NVIDIA Corporation. All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation and
 * any modifications thereto. Any use, reproduction, disclosure, or distribution
 * of this software and related documentation without an express license
 * agreement from NVIDIA Corporation is strictly prohibited.
 *
 */

/* Code by: Denise Demirel
   Version 2.0
   Date: 06.2010
 */

#ifdef _FourRussians20_KERNEL_H_
#define _FourRussians20_KERNEL_H_

#include <stdio.h>
#include "FourRussians20.h"

#define SDATA( index)      cutilBankChecker(sdata, index)

texture<unsigned int, 1, cudaReadModeElementType> texRef; //dimension one
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

__global__ void
kernel_gray(unsigned int* d_g, unsigned int* matr, int start) {
    int trow = threadIdx.x;
    int tcol = threadIdx.y*offset2;
    int blockid = blockIdx.y;
    __shared__ unsigned int redForm[hk][hcol/BlockDimCalcGray];
    unsigned int value[offset2];
    int modrow= trow % hk; //reduced row echelon form has k rows.

    for (int l=0;l<offset2;l++){
        redForm[modrow][tcol+l] = matr[(start + modrow)*hcol+
            (blockid*(hcol/BlockDimCalcGray))+tcol+l];
        value[l]=0;
    }

    syncthreads();

    for(int m=0; m< hk; m++){
        if (((trow>>m)&1)==1){
            for (int l=0;l<offset2;l++){
                value[l]=value[l] ^ redForm[hk-m-1][tcol+l];
            }
        }
    }
}

```

 /Users/ddemirel/M4R1/FourRussians20_kernel.cu

```

    for (int l=0;l<offset2;l++){
        d_g[(trow)*hcol+(blockid*(hcol/BlockDimCalcGray))+tcol+l]= value[l];
    }
}

__global__ void
kernel_gray_xor(unsigned int* matr, int e, int off, int i, int offset) {

    matr = matr + blockDim.x*blockIdx.x*hcol; //add block-offset for row (x-dim)
    int result = (matr[(threadIdx.x)* hcol +e]>>off)&&((1 << hk)-1);
    syncthreads();

    int num= hcol/offset;
    int start = e/num;

    if (((threadIdx.x+blockDim.x*blockIdx.x) > (3*hk+i-2))or
        ((threadIdx.x+blockDim.x*blockIdx.x) < i))and(result)){
        int i = threadIdx.x * hcol; // calculate position in matr
        int j = threadIdx.y; // calculate position in actual row
        for (int l =start; l<offset; l++)
            matr[i+l*num+j]=matr[i+l*num+j]^
                tex1Dfetch(texRef, (result)*hcol+j+l*num);
    }
}

////////////////////////////////////old unused kernels////////////////////////////////////

/*__global__ void
Kernelcalc(uint i, int k, int numrow, unsigned int* row, int e)
{
    uint notchanged = 1;
    uint count = k+1;
    uint pos = 1 << (i-1); // 2^i

    if ((row[k*hcol+e] & pos) != pos){
        do {

            if (((row[hcol*count + e]) & pos) == pos){
                notchanged = 0;
                for (int m= 0; m < hcol; m++){
                    row[k* hcol +m] = row[k* hcol +m] ^ row[hcol *count + m];
                }
            }

            count++;

        }while ((notchanged) and (count < numrow));

        if (notchanged) {
            //printf("Matrix hat keinen vollen Rang? Spalte:%u, Pos: %u \n", e+1, i);

```

```

/Users/ddemirel/M4R1/FourRussians20_kernel.cu
//printf("\n");
count = 0;

    }
}

//for(int j=0; j<numrow;j++)
// calc[j]=(bool)((row[hcol*j + e] & 1<<(i-1))>>(i-1));
//calc[k]=0; // The masterrow mustn't be changed.
}*/

/*__global__ void
GaussKernel(int masterrow, unsigned int* matr, int offset, uint i, int e)
{
    matr = matr + blockDim.x*blockIdx.x*hcol;
    bool calc = (bool)((matr[hcol*threadIdx.x + e] & 1<<(i-1))>>(i-1));

    if (blockDim.x*blockIdx.x+threadIdx.x == masterrow)
        calc = 0;

    threadfence();

    int start = 0;
    if ((threadIdx.y*offset + blockDim.y*offset*blockIdx.y) <= e)
        start = e-(threadIdx.y*offset + blockDim.y*offset*blockIdx.y);

    if (calc){
        int i = threadIdx.x * hcol + offset*threadIdx.y; // calc pos in matr
        int j = offset*threadIdx.y; // calc position in row
        for (int l =start; l<offset; l++)
            matr[i+l]=matr[i+l]^matr[masterrow* hcol +j+l];
    }
}*/

#endif

```


Anhang B

Technische Spezifikationen

Feature Support (Unlisted features are supported for all compute capabilities)	Compute Capability				
	1.0	1.1	1.2	1.3	2.0
Integer atomic functions operating on 32-bit words in global memory (Section B.11)	No	yes			
Integer atomic functions operating on 64-bit words in global memory (Section B.11)	No		Yes		
Integer atomic functions operating on 32-bit words in shared memory (Section B.11)					
Warp vote functions (Section B.12)					
Double-precision floating-point numbers	No		Yes		
Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.11)	No				Yes
__ballot() (Section B.12)					
__threadfence_system() (Section B.5)					
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)					

Abbildung B.1: Unterstützte Funktionen [4].

Technical Specifications	Compute Capability				
	1.0	1.1	1.2	1.3	2.0
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24	32			48
Maximum number of resident threads per multiprocessor	768	1024			1536
Number of 32-bit registers per multiprocessor	8 K	16 K			32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture or surface reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or for a 2D texture or surface reference bound to a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of textures that can be bound to a kernel	128				
Maximum number of surfaces that can be bound to a kernel	8				
Maximum number of instructions per kernel	2 million				

Abbildung B.2: Technische Spezifikationen [4].

Literaturverzeichnis

- [1] Bard G.V., Algebraic Cryptanalysis, Springer, 2009
- [2] Gregory V. Bard, Accelerating Cryptanalysis with the Method of Four Russians, 2006
- [3] NVIDIA, www.nvidia.de/page/products.html
- [4] NVIDIA, Programming Guide 3.1, 2010
- [5] NVIDIA, Best Practices Guide 3.1, 2010
- [6] “SAGE/M4RI-Team“, M4RI, m4ri.sagemath.org/performance.html
- [7] Prof. Dr. Manfred Schimmler, Vorlesung: Parallele Architekturen und Algorithmen, Universität Kiel, 2010, www.informatik.uni-kiel.de/schimmler/lehre/vorlesungen/parallele-architekturen-und-algorithmen/vorlesungsfolien/
- [8] Prof. R. Hoffmann, Vorlesung: Rechnerarchitektur, Technische Universität Darmstadt, 2010, www.ra.informatik.tu-darmstadt.de/lehre/ra/
- [9] Andreas Frommer, Lösung linearer Gleichungssysteme auf Parallelrechnern, Universität Karlsruhe, 2007, www-ai.math.uni-wuppertal.de/~dfritzsche/lehre/parallel_ws07/buch90/buch90.pdf
- [10] Thomas Huckle, Stefan Schneider, Effiziente Lösung linearer Gleichungssysteme, 2006, www.springerlink.com/content/g278731132748770/

- [11] NVIDIA, Angaben zur Firma, www.nvidia.com/page/companyinfo.html
- [12] Intel, www.intel.com/technology/platform-technology/hyper-threading/index.htm
- [13] NVIDIA, www.nvidia.com/object/cuda_home_new.html
- [14] Prof. Dr. Michael Goesele, Vorlesung: Programming Massively Parallel Processors, 2010, www.gris.tu-darmstadt.de/teaching/courses/ss10/pmpp/index.en.htm
- [15] Raphaël Laurent, Jérémie Tharaud, Linear algebra over the field with two elements using GPUs
- [16] AMD, www.amd.com/stream
- [17] gpgpu.org/
- [18] Ralf-Philipp Weinmann, Algebraic Methods in Block Cipher Cryptanalysis, 2008