Hochschule Neubrandenburg
University of Applied Sciences

Hochschule Neubrandenburg
Studiengang Geoinformatik

Untersuchungen zur Parallelisierung des JPEG-Baseline-Verfahrens mit einer NVIDIA-Grafikkarte

Bachelorarbeit

vorgelegt von: Tobias Lochmann

Zum Erlangen des akademischen Grades
'Bachelor of Engineering (B.Eng.)'

Erstprüfer: Prof. Dr. Andreas Wehrenpfennig
Zweitprüfer: Prof. Dr. Welf Löwe
Abgabetermin: 10. September 2010

urn:nbn:de:gbv:519-thesis2010-0476-9

Hiermit versichere ich, die vorliegende Bachelorarbeit 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 worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Neubrandenburg, den 9. September 2010

. . . . . .

(Unterschrift des Kandidaten)

Kurzfassung

Im Rahmen dieser Bachelorarbeit werden Nutzen und Potential der Parallelprogrammierung auf NVIDIA Grafikkarten, anhand des JPEG Algorithmus Baseline, aufgezeigt. Dazu werden zunächst die theoretischen Grundlagen der parallelen Datenverarbeitung erläutert. Im zweiten Kapital wird die Entwicklungsumgebung CUDA vorgestellt. Anschließend wird in Kapital drei, auf den JPEG Algorithmus eingegangen und sein Parallelisierungspotential beschrieben. Abschließend wird ein Modul des JPEG Algorithmus in CUDA umgesetzt und mit dem seriell korrespondierenden C++ Code verglichen.

Schlüsselwörter: CUDA, JPEG, Parallelisierung, Grafikkarten, GPGPU

Abstract

This bachelor thesis describes the benefits and potential of parallel programming on NVIDIA graphics cards, shown by reference of the JPEG baseline algorithm. The first instance discussed the theoretical foundations of parallel computing. In the second capital present the CUDA development environment. Then, in capital three recorded the JPEG algorithm and described the parallelization potential. Finally, a individual modul of the JPEG algorithm are implemented in CUDA to be compared with the corresponding serial C++ code.

Keywords: CUDA, JPEG, Parallelization, Graphic Card, GPGPU

Inhaltsverzeichnis

1 Einleitung

In Zeiten, wo die Masse an Informationen wächst, muss diese auch möglichst effizient verarbeitet werden, dabei gilt es, eine kostengünstige Alternative zu Supercomputern oder andren Multicore-CPU-Workstations zu finden. Eine viel versprechende Lösung ist der Einsatz von Grafikchips in der Parallelprogrammierung. AMD und NVIDIA haben diese Marktnische erkannt. Die Rechenleistung ihrer Grafikkarten, nicht nur ausschließlich zur Berechnung von Pixel-Shader in 3D-Spielen zu benutzen, sondern auch für den Einsatz rechenintensive Aufgaben einzusetzen. Diese Möglichkeit allgemeine Aufgaben auf der Grafikkarte zu berechnen, bezeichnet man auch als General Purpose Computation on Graphics Processing Unit (GPGPU). In Grafikkarten von heute steckt viel Potenzial, hier arbeiten mehr als 800 Rechenwerke parallel. Dabei kommen sie auf eine Rechenleistung von mehreren Billionen Gleitkommaoperationen pro Sekunde (TFlops). Das ist das Zehnfache dessen, was ein Quad-Core-Prozessor Intel Core i7-965 zu leisten vermag (102 GFlops). Allerdings müssen Anwendungen eine sehr hohe Zahl von parallel ausführbaren Berechnungen enthalten, damit sich der Einsatz von Grafikkarten lohnt. Was auf die typische Alltagssoftware nicht zu trifft, da diese meisten, sequenzieller Natur sind. Im Bereich von Technik/Wissenschaft hat GPGPU einen sehr hohen Stellenwert. Das liegt einerseits daran das viele wichtige mathematische Verfahren, wie Matrizen-Multiplikation oder Fourier-Transformation, schon per Definition eine sehr hohe Datenparallelität besitzen und sich daher sehr gut zu Parallelisierung eignen. Des Weiteren sind sie noch vergleichsweise preiswert (ab 150 Euro), daher haben sie eine sehr hohe Wirtschaftlichkeit. Heutige Einsatzbereiche sind sehr breit gefächert, z.B. in der Berechnung von physikalischen Simulationen aller Art (Strömung, Gravitation, Temperatur, Crash-Tests), komplexe Klimamodelle, Wettervorhersagen, Datenanalysen und Finanzmathematik, um nur einige zu nennen.

2 Grundlagen

2.1 Was bedeutet parallele Programmierung?

Parallele Programmierung ist eine Form des Rechnens, in der viele Berechnungen gleichzeitig ausgeführt werden. Sie beruht auf dem Grundsatz, dass oft große Probleme in kleinere abstrahiert, und dann gleichzeitig (parallel) gelöst werden können. Dadurch kommt es zu einen enormen Geschwindigkeitsgewinn. Es gibt verschiedene Formen des parallelen Rechnens: Bit-Ebene, Befehls-Ebene, Daten und Task-Parallelität.

Datenparallelität: Viele Prozesse bzw. Prozessoren führen zeitgleich, äquivalente die gleichen Operationen auf verschiedenen Daten aus. Angewendet wird dieses im technisch - wissenschaftlichen Rechnen z . B. Strömungsmechanik und Bildverarbeitung.

Bit-Ebene : Diese Parallelität steigt mit der Größe der sogenannten Prozessor Datenwort/-Binärwort. Jene ist die Grundverarbeitungsdatengröße bei einem Computer. Die Größe dieser Dateneinheit wird in Bit angegeben. Gängige CPUs haben eine Datenwort Größe von 32 bis 64 Bit.

Befehls-Ebenen(Instruction level parallelism ILP): Ist eine Maßeinheit, die angibt, wie viele Operationen in einen Programm gleichzeitig ausgeführt werden können. z . B.

1. a=e+f
2. b=g+h
3. c=a+b

Operation 3 hängt von den ersten beiden Operationen ab, daher starten sie erst, wenn 1. und 2. beendet sind. 1. und 2. sind von keiner weiteren Operation abhängig, dadurch können sie simultan berechnet werden. Wenn man jetzt annimmt, dass jede Operation eine Zeiteinheit verbraucht, so haben diese drei Operationen einen totalen Zeitverbrauch von 2 Zeiteinheiten. Dieses ergibt einen ILP von 3/2.

Task Parallelität: Ist erreicht, wenn in einem Mikroprozessorsystem jeder Prozessor einen anderen Thread (Prozess) auf den selben oder verschiedenen Daten ausführt.

2.2 Architekturen

Nach der Flynnschen Klassifikation gibt es 4 Kategorien von Rechnern. Dabei werden die Architekturen nach der Anzahl der vorhandenen Befehle und Datenströme unterteilt (siehe Tabelle 1). SISD steht für Single Instruction Single Data und ist die traditionelle Architektur eines Personal-Computer (PC). MISD (Multiple Instruction Single Data) beschreibt die Funktionsweise eines Schachcomputers. Zur Parallel-Programmierung sind nur die beiden

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multi Data	SIMd	MIMD

Tabelle 2.1: Flynnsche Klassifikation, Beschreibung der verschiedenen Rechnerarchitekturen

unteren Architekturen: SIMD (Single Instruction, Multiple Data) und MIMD (Multiple Instruction, Multiple Data) geeignet. Diese findet sie in Großrechnern bzw. Supercomputern und heute auch schon in einigen PCs.[M.F09] Des Weiteren lassen sich Parallel-Computer grob nach dem Level, auf dem die Hardware unterstützte Parallelisierung stattfindet, einstufen. Multi-Core oder auch Multi-Prozessor-Computer, sind z . B. Computer mit mehreren Verarbeitungselementen in einer einzigen Maschine. Während dagegen Cluster, MPP (Massively Parallel Processing) und Grids mit mehreren Computern verbunden Netzwerke, mit der gleichen Arbeitsaufgabe sind. [PI03]

2.3 Theoretische Grundlagen zur Beschleunigung von Programmen durch parallele Ausführung

Es gibt zwei wesentliche Untersuchungen zur parallelen Programmierung. Zum einen das Amdahlsche Gesetz, das pessimistisch der massiven Parallelisierung von Programmen gegen übersteht und zum anderen, das Gustafsonsche Gesetz, das diesen eher positiv gegenüber steht.

2.3.1 Amdahlsches Gesetz

Das Amdahlsche Gesetz geht von einem sehr einfachen Modell des zu parallelisierenden Programmcodes aus. Es nimmt an, dass jedes Programm zu einem gewissen Teil beliebig stark parallelisierbar ist und zu einem gewissen Teil sequenziell ablaufen muss. Daher kann ein Programm, nie vollständig parallel ausgeführt werden. Folglich zerlegt er den Programmablauf in Abschnitte, die entweder vollständig sequentiell oder vollständig parallel ablaufen und fasst sie zu jeweils einer Gruppe zusammen. Wenn man nun ein fiktives Programm starten würde und seine Laufzeit mit 1 definiert, gleichermaßen den parallelisierbaren Teil mit P bezeichnet, so erhält man die serielle Laufzeit durch $(1 - P)$. Auf einen parallelisierten System mit N Prozessoren ergibt sich die gesamte Laufzeit durch:

$$(\underbrace{(1-P)}_{sequentiell} + \underbrace{\left(\frac{P}{N}\right)}_{parallel})$$

Dadurch lässt sich der Zeitgewinn errechnen:

$$S = \frac{urspruenglicheLaufzeit}{neueLaufzeit} = \frac{1}{(1-P) - \left(\frac{P}{N}\right)}$$

Hierbei würde z. B. ein Zeitgewinn (S) von 2 bedeuten, dass das Programm nun doppelt so schnell läuft wie vorher. Welches auch als “Speedup“ oder “Geschwindigkeitsgewinn“

bezeichnet wird. Daraus ergibt sich folgende Konsequenz des Amdahlschen Gesetzes: Ein Programm kann nur bis zu einer bestimmten Grenze beschleunigt werden. Es gilt folgende Ungleichung:

$$S \leq \frac{1}{(1-P)}$$

1-P ist genau der Laufzeit-Anteil, der nicht parallelisiert werden kann. Die Grenzen können auch durch setzen der Anzahl der Prozessoren N auf ∞ hergeleitet werden.[Shi09]

$$\lim_{N\to\infty} S = \lim_{N\to\infty} \frac{1}{(1-P) - \left(\frac{P}{N}\right)}$$

Das heißt, selbst mit tausenden Prozessoren können wir ein zu 95 % parallelisierbares Programm höchstens um den Faktor 20 beschleunigen.

$$\left(\frac{1}{1-P} = \frac{1}{1-0,95} = 20\right)$$

Dabei berücksichtigt das Amdahlschen Gesetz folgende Aspekte nicht:

1. Je mehr Prozessoren an einem Problem arbeiten, umso aufwändiger wird die Kommunikation zwischen ihnen. Dieses beansprucht zusätzlichen Verwaltungsaufwand.

2. Neben der Anzahl der Prozessoren wachsen auch weitere Ressourcen mit der Parallelisierung. So bringt jeder zusätzliche Prozessor auch Cache mit, sodass insgesamt für die Berechnungen viel mehr Prozessor-Cache zur Verfügung steht. In einem Rechencluster kommt sogar noch mehr RAM und Festplattenspeicher hinzu.

3. Es geht nicht nur um die Geschwindigkeit, wie schnell ein Problem gelöst wird, sondern auch darum größere Probleme in der bisherigen Zeit zu lösen.

2.3.2 Gustafsons Gesetz

Gustafson verwendet einen anderen Ansatz, der zwar auf dem von Amdahl aufbaut, aber aufzeigt, dass massive Parallelisierung doch effektiver ist. Zwar kann man nicht beliebig schneller werden, aber in der gleicher Zeit beliebig große Probleme lösen. Gustafson erweitert den Ansatz von Amdahl dahingehend, dass er die Problemgröße mit einbezieht, die er mit K bezeichnet. Ursprüngliche Probleme haben die Größe K=1. Die Laufzeit eines Programms bei der Problemgröße K=1 auf einem 1-Prozessor ergibt wieder 1. Der Parallelanteil an der Problemgröße K von P‘ an der Laufzeit. Nach Gustafson bleibt der sequentielle Teil eines Problems konstant, während der verteilbare Teil des Problems linear mit der Prozessorzahl wächst. Denn dieser besteht aus vielen Operationen, deren Dimension sich direkt aus der Größe der Eingabedaten ergeben. Der sequenzielle Programmteil hingegen, besteht überwiegend aus Initialisierungs-Schritten, die unabhängig von der Problemgröße sind. Daher ist die Laufzeit für ein 1-Prozessor-System : [PI03] [Shi09]

$$(\underbrace{(1-P')}_{sequentiell} + \underbrace{(K*P')}_{parallel})$$

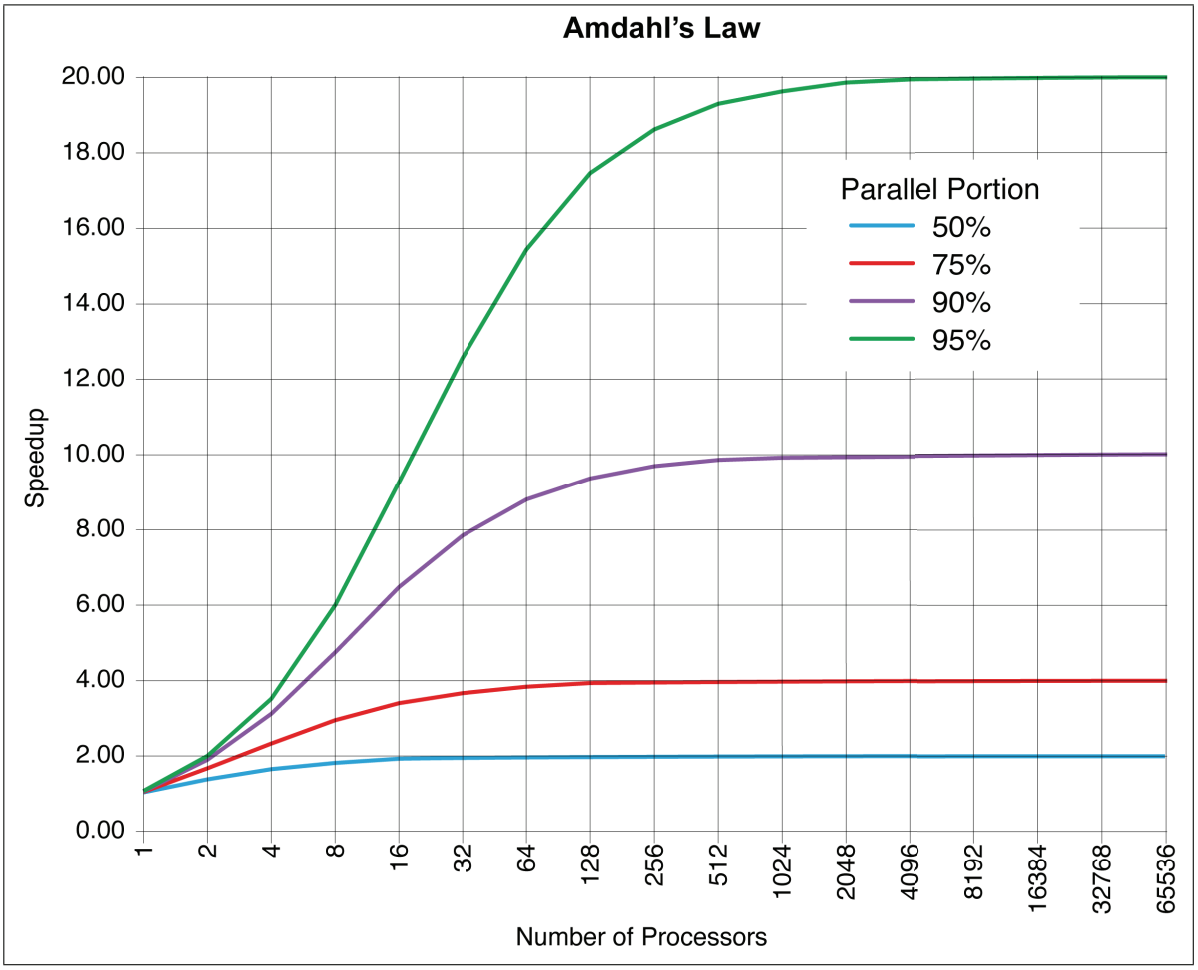

Abbildung 2.1: Der Geschwindigkeitsgewinn bei der Verwendung von parallelarbeitenden Prozessoren, bei der Bearbeitung eines parallelen Problems

Für ein N -Prozessor-System wir der parallelisierbare Programmteil N-mal so schnell:

$$(\underbrace{(1-P')}_{sequentiell}+\underbrace{\left(\frac{K*P'}{N}\right)}_{parallel})$$

und der Zeitgewinn entspricht:

$$\frac{(1-P')+K*P'}{(1-P')+\frac{K*P'}{N}}$$

2.3.3 Weitere Begriffe

Neben dem Zeitgewinn werden weitere Maße zur Bewertung von Parallelrechnern verwendet, die aber in der Praxis nur eine geringe Bedeutung erlangt haben.

(System-) Effizienz eines Parallelrechners
$E(P)$=Zeitgewinn/Prozessoren

Redundanz eines parallelen Programms
$R(P)$=Operationen des sequentiellen Programm/Operationen im parallelen Programm bei P parallelen Verarbeitungseinheiten

Es gilt $R(P) => 1$, wobei große Werte von R(P) auf eine schlechte Eignung des Programms zur Parallelverarbeitung hindeuten. Sofern die Wartezyklen nicht als Operationen betrachtet werden, bedeutet eine geringe Redundanz nicht gleichzeitig auch eine gute Parallelisierbarkeit. Vielmehr bewertet die Redundanz den Overhead (Aufwand), der bei der Parallelisierung auftritt.[PI03]

Qualität eines parallelen Programms auf einem Parallelrechner
$Q(P)$=(Zeitgewinn*Effizienz)/Redundanz

2.4 Grafikkarten

Um die Möglichkeiten der Parallel Programmierung auf Grafikkarten zu untersuchen, ist es erforderlich sich kurz mit dem Aufbau einer Grafikkarte, deren Grundbegriffe und dessen Funktionsweise, speziell mit der Datenverarbeitung zu befassen. Die Entwicklung von Grafikkarten ist einer der am schnellsten wachsenden Sektoren im Bereich der Computerentwicklung. Am besten nachweisen lässt sich diese Tendenz, anhand der Anzahl der Transistoren auf einem handelsüblichen Prozessor. Bei CPUs (Centraling Processor Units) galt hierbei meist das 'Mooresche Gesetz', nach welchem sich die Transistorenanzahl alle 18 Monate verdoppelt. Bei GPUs (Graphics Processoring Units) wird dieses Gesetz um ein Vielfaches überboten, da sich die Transistorenanzahl zirka alle sechs Monate verdoppelt. Auch die Anzahl der Prozessoren übertrifft die einer CPU bei weitem. Vergleicht man zum Beispiel einen aktuellen Intel Prozessor Core 2 Duo (Merom mit 291 Millionen Transistoren), mit dem aktuellen Prozessor von NVIDIA, dem GT200, mit 1.400 Mio. Transistoren, sieht man das sie jetzt schon die Anzahl von CPU Prozessoren bei weiten übertrifft. Dennoch kann ein Grafikprozessor nicht mit einem Zentralprozessor verglichen werden, da er speziell für seinen Einsatzzweck ausgelegt ist. GPUs bauen meistens auf einer Stream-Processing-Architektur

auf. Bei dieser Architektur werden die Daten als Ströme über eine Verarbeitungskette, sogenannte Pipelining, mit einzelnen Verarbeitungsstationen hindurch geschleust. Dazu müssen die Daten eine gewisse Analogie besitzen und unabhängig voneinander verarbeitet werden können. [Hoe08]

2.4.1 Aufbau einer Grafikkarte

Der Aufbau einer Grafikkarte ist heutzutage immer einheitlich. Hauptbestandteil sind der Grafikchip, der Bild-/Videospeicher, RAMDAC (der für die Umwandlung von digitalen in analoge Signale sorgt), AGP- oder PCI-Express-Schnittstellen, sowie die verschiedenen Ausgänge (DVI- oder VGA-Schnittstelle für einen Monitor, S-Video Ausgang für Fernseher) (siehe Abbildung 2.2).

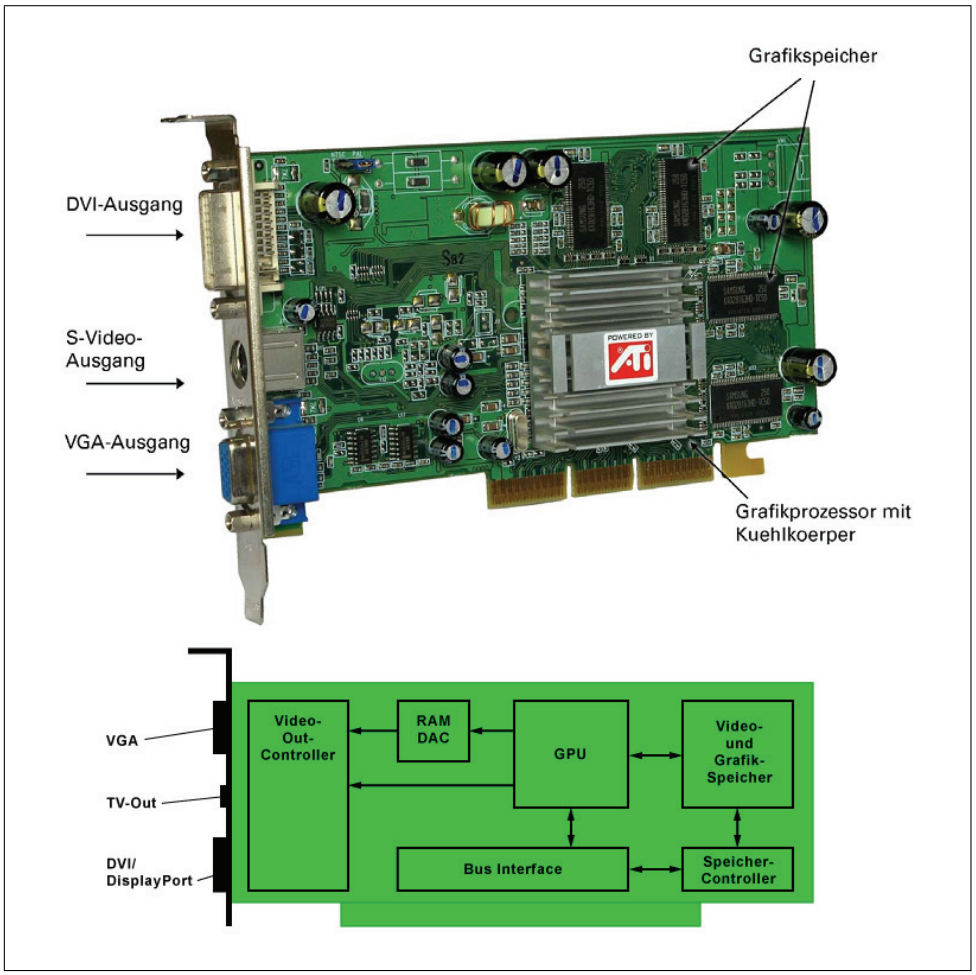

Abbildung 2.2: Aufbau einer Grafikkarten(ATI-Radeon 9000)

Grafikspeicher

In den ersten Jahren bestimmte die Größe des Grafikspeicher noch die mögliche Auflösung und Farbtiefe. Sie wurde nur als Framebuffer benutzt, in welcher die berechneten Pixel ab-

gelegt wurden. Heute ist weit mehr Grafikspeicher verfügbar (256MB bis 2GB) und er wird zur Speicherung von Texturen und 3D-Modellen verwendet. VRAM (Video Random Access Memory) und MDRAM (Multibank Dynamic Random Access Memory) waren spezielle Speicher, die nur auf Grafikkarten verwendet wurden. Dies war notwendig, da die damals als Hauptspeicher üblichen DRAMs (Dynamic Random Access Memory) nicht erlaubten, Daten gleichzeitig zu lesen und zu schreiben. Diese war aber für die Grafikverarbeitung wichtig, da ein neues Bild während der Ausgabe des vorhergehenden erzeugt wurde. Heute werden in Grafikkarten GDDR (Graphics Double Data Rate)-RAM verwendet, die von dem im Hauptspeicher üblichen DDR-Rams abstammt.[Hoe08]

Grafikprozessor

Der Grafikprozessor dient zur Transformierung von Vektordarstellungen des Computers in ein 2D-Pixelbild, welches später auf dem Bildschirm ausgegeben wird. Erst mit den sogenannten Windows-Beschleuniger-Grafikkarten waren eigene Berechnungen möglich. Grund war der Übergang von den Text-basierten Anwendungen (DOS) zu grafischen Benutzeroberflächen (Windows). Durch Entwicklung von 3D-Anwendungen wurden Grafikprozessoren immer weiter an diese Spezialanwendung angepasst. So kamen nach und nach immer mehr Schritte hinzu, die die Grafikkarte von der CPU übernahm.[Hoe08]

2.5 Was ist GPGPU?

GPGPU steht für *General-Purpose computation on Graphics Processing Units*, auch bekannt als GPU-Computing. Graphics Processing Units (GPUs) sind High-Performance-many-Core-Prozessoren mit einer sehr hohen Rechen und Datendurchsatzrate. Die nicht mehr nur primär der Bildanzeige dienen, sondern auch immer mehr Verwendung für allgemeine wissenschaftliche und technische Berechnungen finden. [MH09] Beim GPU-Computing werden CPU und Grafikprozessor gemeinsam für Berechnungen eingesetzt. Der sequenzielle Teil der Anwendung läuft auf der CPU und der rechenintensive Teil auf dem Grafikprozessor. Aus Sicht des Nutzers läuft die Anwendung dadurch schneller, da die Verwendung des leistungsstarken Grafikprozessors die Ausführung beschleunigt.[nvi09] Erstmalig im Jahre 2002 erfüllten die Pixel-Shader-Einheiten der Grafikchips, Radeon HD 9700 und GeForce FX 5800 die Voraussetzungen für technisch wissenschaftliche Anwendungen. Zu dieser Zeit wurde noch über die OpenGL oder DierectX-Grafikschnittstellen programmiert und mathematische Algorithmen umständlich in Pixel-Shader-Programme und Texturen verpackt. Im Jahre 2004 wurde erstmalig an der Stanford-Universität (USA) eine Programmierschnittstelle für allgemeine Berechnungen auf Grafikkarten vorgestellt, mit dem Namen BrookGPU. Der Brook-Compiler verwendet die gängige Programmiersprache C und erweitert diese um einige spezielle Funktionen, mit denen man einen Rechenalgorithmus mit relativ wenig Aufwand auf hunderte Shader-Prozessoren von Grafikchips verteilen konnte. Diese schleusen dann einen Strom von Daten durch den immer gleichen Algorithmus. Stream Computing war geboren. Das weckte damals die Aufmerksamkeit von AMD und NVIDIA. Dadurch ging ein Teil der BrookGPU-Entwickler zu NVIDIA und sie entwickelten dort die GPU-Programmierschnittstelle CUDA (Compute Unified Device Architecture). Andere Mitglieder der Gruppe entwickelten bei AMD eine an Radeon-Grafikkarten angepasste Version vom BrookGPU namens Brook+. [Ber09]

2.6 Stream-Processing

Stream-Processing ist ein Modell, das die Parallelisierung geeigneter Probleme in Hard- und Software vereinfacht beschreibt. Dabei wird zwischen zwei grundsätzlichen Elementen unterschieden: Streams und Kerneln. Ein Stream oder auch Datenstrom genannt, ist eine geordnete Menge von Daten vom gleichen Typ, wobei der Datentyp nicht notwendigerweise ein „Primitiver“ sein muss und die Länge eines Steams nicht beschränkt ist. Des Weiteren sind die erlaubten Operationen auf einem Stream eingeschränkt. Es ist erlaubt einen Stream zu kopieren, einen Stream in mehrere Substreams aufzuteilen, mit Hilfe eines Index-Streams einen Stream zu indizieren und Berechnungen auf ihm auszuführen (mit Hilfe eines Kernels). Ein Stream ist in der Programmierung vergleichbar mit einem Array, der aber die genannten Anforderungen erfüllen muss. Als Kernel werden im Stream-Processing die Funktionen bezeichnet. Sie verarbeiten den Stream und erzeugen wiederum einen neuen Ausgabe-Stream oder mehrere. Das Besondere eines Kernels ist, dass die Ausgabe nur von der Eingabe abhängt und immer auf dem kompletten Stream angewendet wird. Außerdem sind die Berechnungen eines Elementes im Ausgabestream unabhängig von den Berechnungen anderer Elemente. Durch diese recht starken Einschränkungen bieten mögliche Berechnungen allerdings ein sehr hohes Optimierungspotential, insbesondere im Bereich der Kommunikation und Berechnung. Bei der Kommunikation wird zwischen zwei Arten unterschieden. Zum einen die off-chip-Kommunikation, die die Kommunikation zwischen dem Prozessor und einem externen Partner, z. B. RAM-Speicher beschreibt und der on-chip-Kommunikation, die die Kommunikation innerhalb des Prozessors selbst beschreibt. Durch Stream-Processing können die Kosten für off-chip-Kommunikation reduziert werden, beziehungsweise auf on-chip-Kommunikation umgelagert werden. Dieses geschieht z. B. dadurch, dass die Kommunikation zwischen Kernels innerhalb des Prozessors erfolgen kann. Darüber hinaus arbeiten Kernels im Regelfall auf sehr großen Streams. Der Vorteil der Übertragung sehr großer Datenmengen besteht darin, dass die Kosten der Initialisierung im Vergleich zur Speicherbewegung sehr gering sind. Durch die Einschränkungen des Kernels, dass die Berechnung eines Elementes von den anderen Berechnungen unabhängig sein muss, können theoretisch alle Elemente gleichzeitig bearbeitet werden. Diese Datenparallelität ermöglicht einen sehr hohen Geschwindigkeitszuwachs. Weiterhin wird durch die Unabhängigkeit der Berechnung eines Elementes im Kernel gewährleistet, dass mehrere Kernel gleichzeitig ausgeführt werden können (Tasklevelparallelität). Folgedessen wird die Ausgabe eines Kernels für einen anderen Kernel als Eingabe verwendet. Somit kann der zweite Kernel bereits mit den Berechnungen beginnen, sobald der erste Kernel die Elemente in den Ausgabestream schreibt. Folglich ist es nicht mehr nötig die Daten zwischen zu speichern. Zusätzlich zu diesen Eigenschaften bieten Kernel die Möglichkeit, da sie häufig mehrere Eingabe Streams haben können, unterschiedliche Berechnungen auf den verschiedenen Streams gleichzeitig durchführen (Instructionlevelparallelität).[Bre06]

2.7 Die Grafikpipeline

Die Grafikpipeline ist ein Modell in der Computervorstellung, die beschreibt, welche Schritte ein Grafiksystem beim Rendern durchführen muss. Da diese Schritte sowohl von der Soft- und Hardware als auch von den gewünschten Darstellungseigenschaften abhängen, gibt es keinen allgemein gültigen Aufbau. Zur Ansteuerung werden die Grafik-APIs Direct3D oder

OpenGL verwendet, die die zugrunde liegende Hardware abstrahieren und so das Programmieren vereinfachen. Im Grunde lässt sich eine Grafikpipeline in drei große Schritte aufteilen: Anwendung, Geometrie und Rasterung.

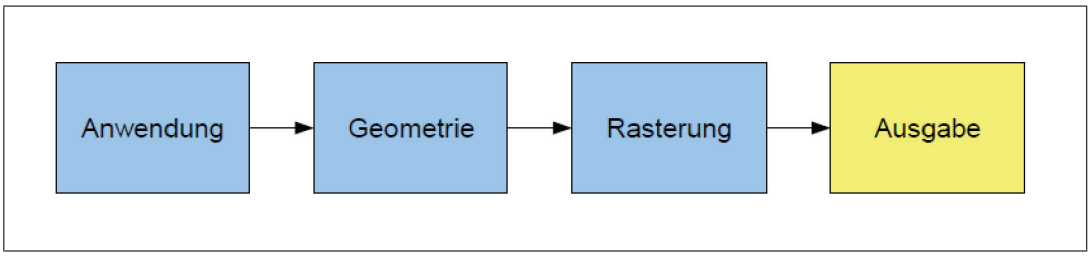

Abbildung 2.3: Einfache Darstellung einer Grafikpipeline

Anfang der Pipeline (FFP) stehen die Vertex Daten. Ein Vertex ist ein Eck- oder Scheitelpunkt eines Polygons. Es enthält seine Positionsangabe (Koordinatenpunkte x,y,z) und häufig auch noch weitere Attribute wie Farbwert oder Materialart. Die Polygone werden durch die Tesselation in so genannte primitive Flächen, wie z . B . Dreiecke oder Vierecke zerlegt, heutige Grafik-APIs und Grafikkarten sind meistens für Dreiecke optimiert, da solche primitive Flächen leichter zu handhaben sind als komplexe Polygone. In der Modellierung werden die Objekte, die aufgrund der einfachen Modellierbarkeit in einen lokalen Koordinatensystem angegeben sind, in ein globales Koordinatensystem, mittels Translation, Rotation oder Skalierung, transformiert. Im zweiten Schritt, der Beleuchtung (Lighting), werden anhand der Position der Lichtquellen und der Materialeigenschaften, sowie deren Textur, die Farben der Eckpunkte, des Dreiecks berechnet. Als Nächstes folgt das Clipping. Hier wird der Sichtkegel der Kamera berechnet und alle nicht sichtbaren Objekte abgeschnitten. Dadurch müssen weniger Daten in den nächsten Stufen bearbeitet werden. Die nachfolgenden Schritte der Pipeline (FFP) finden auf Pixel- und Vektorebene statt. Hierbei handelt es sich um Operationen, die auf jedem Pixel ausgeführt werden müssen. Beim Texture Mapping werden die 2D-Texturen anhand der definierten Texturkoordianten an der Oberfläche des Objektes angebracht. Beim Fog-Blending wird je weiter das Objekt entfernt ist, der Kontrast verringert, um so einen Tiefeneindruck zu vermitteln. Darauf folgt das Z-Buffering oder auch Tiefenpuffer genannt. Hier werden einzelne Vektoren die sich im Sichtkegel überlappen, ermittelt und so angeordnet, dass für den Betrachter die am naheliegendsten sichtbar sind. Zu Beginn werden die Einträge im Z-Buffer auf eine unendliche Entfernung gesetzt. Dann wird für jedes Pixel ein Entfernungswert überprüft. Dieser Wert wird gespeichert und zeigt an, wieweit sich das aktuelle Pixel vom Betrachter entfernt befindet. Nur wenn der Punkt nicht weiter weg vom Betrachter ist, als ein vor ihm stehender Punkt, wird er dem Framebuffer übergeben (siehe Abbildung 2.5). Nach dem Abschluss aller Berechnungen wird über den Doublebuffering das fertige Bild ausgegeben. Doublebuffering bedeutet, das der Framebuffer in einen Front und einen Backbuffer unterteilt ist. Hierbei wird der Frontbuffer auf dem Bildschirm ausgegeben und im Backbuffer wird das nächste Bild berechnet. Ist die Berechnung abgeschlossen wird der Inhalt wieder getauscht und der Prozess beginnt von vorn. Ziel des Verfahrens ist die Gewährleistung einer kontinuierlichen Bildfrequenz ohne Flackern.[Hoe08]

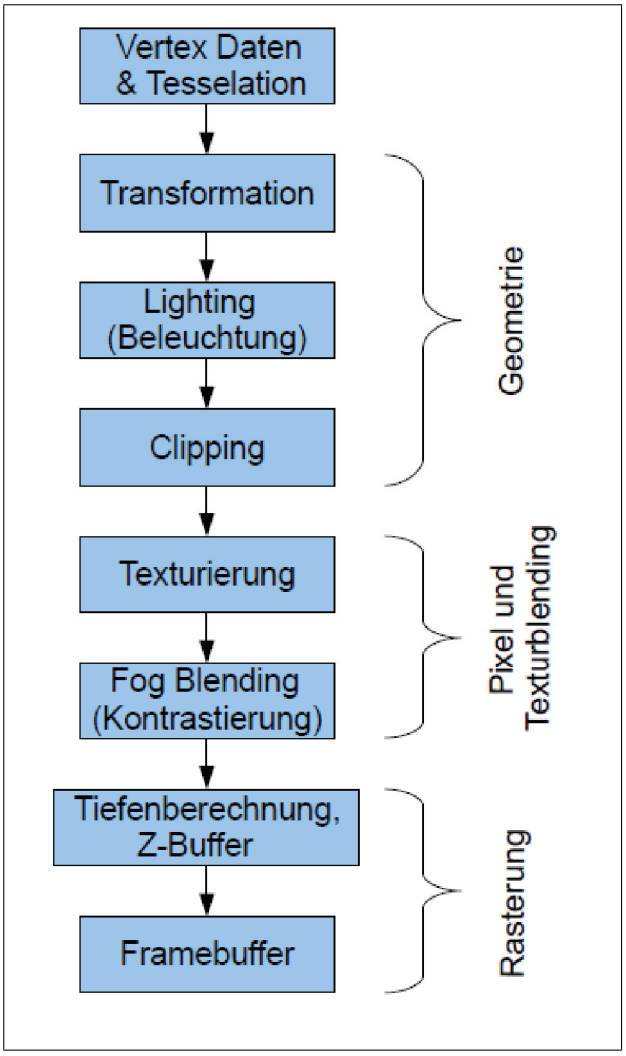

Abbildung 2.4: klassische 3D-Grafikpipeline oder Fixed-Function-Pipeline

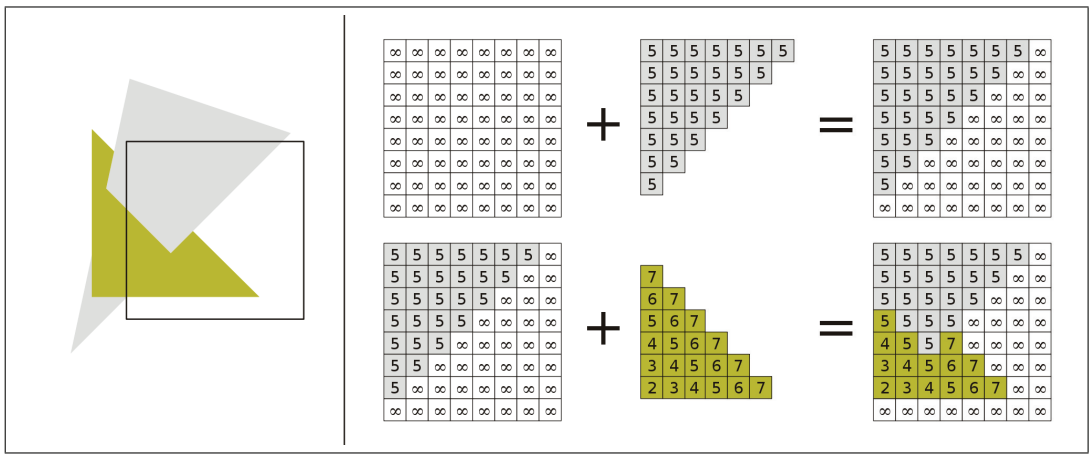

Abbildung 2.5: Z-Buffering

2.7.1 Shader

Seit dem Erscheinen der ersten 3dfx Vodoo-Karten, stieg die Performance immer mehr an, aber die grafischen Fähigkeiten verbesserten sich nicht. Der Grund für diese Verbesserung war, das die Grafikkartenentwickler, die Funktionen und die Algorithmen direkt in den Grafikchip gebrannt haben, daher kam es dazu das kein eigenen Algorithmen verwendet wurden. Aus diesem Grund wurden das Pixel- und Texturblending der Pipeline (FFP) mit einem eigenen Instruktionsspeicher zu SIMD-Prozessorarrays aufgebaut. Die sogenannten Shaders entsprechen spezialisierten Floating Point Units. Hierbei dienen die Pixel-Shader dazu, die zu rendernden Polygone zu verändern, um beispielsweise eine realistischere Darstellung von Oberflächen- und Materialeigenschaften zu erreichen. Kurz darauf folgte der Vertex-Shader am Eingang der Pipeline. Er verändert die Geometrie einer Szene, indem er die Vertices einer Oberfläche verschiebt und so die Form des Objektes verändert, demzufolge wird auch die Beleuchtung der Objekte beeinflusst. Allerdings kann ein Vertex-Shader nur bestehende Geometrie verändern, er kann keine entfernen, noch neue hinzufügen. Hierfür wurde der Geometry-Shader entwickelt, der seit DirectX10 unterstützt wird. Beide zusammen ersetzten somit die ältere Transformierung und Beleuchtung der Pipeline (FFP).[Jor08]

2.7.2 Unified Shader

Das Konzept Unified Shader wurden umgesetzt, da es häufig dazu kam, das die Rechnenkapazit von Pixel- und Vertexshadern nicht voll ausgenutzt wurden. Wobei meistens einen von beiden zum Nadelöhr wurde (meist die Pixelshader), während die anderen nicht ausgelastet waren. Zudem wurden sich die Instruktionssätze der drei Shadertypen im Laufe der Zeit immer ähnlicher. Der Unteschied zu anderen Shadern ist, dass jede Shadereinheit die gleichen Operationen ausführen kann. Die Verteilung der Ressourcen übernimmt hierbei ein Arbiter (eine Art Scheduler), daher kann eine bessere Gesamtauslastung erreicht werden. Unified Shader macht somit das Pipeline-Modell, eher zu einem Programmiermodell für Renderingaufgaben.[Jor08]

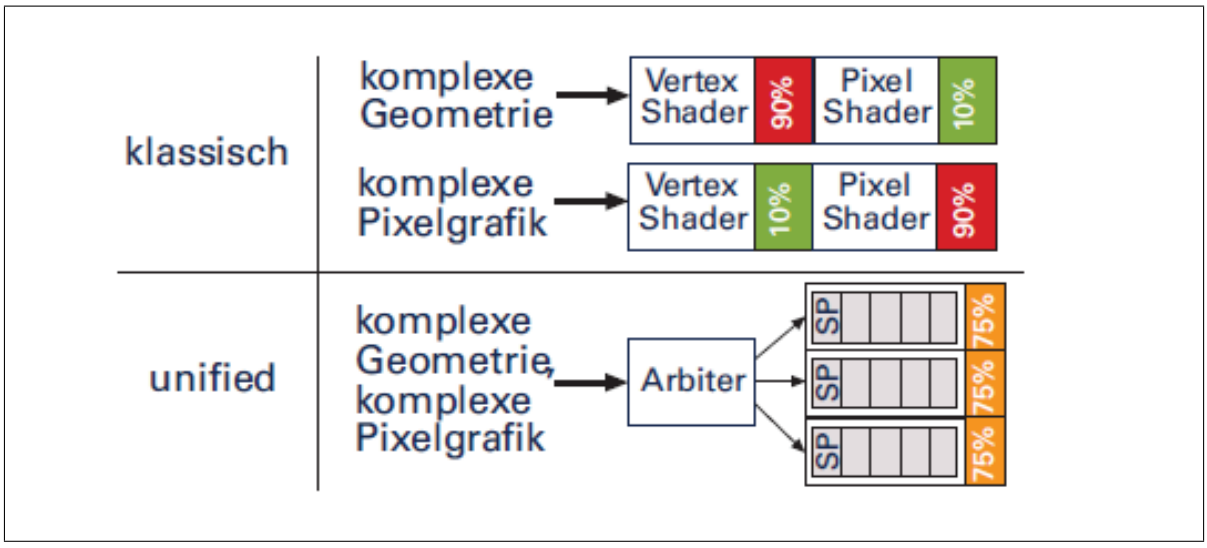

Abbildung 2.6: Unified Shader Architektur

3 CUDA

3.1 Einführung

CUDA (Compute Unified Device Architecture) ist eine parallele Hardware/Software Berechnungs Architektur von NVIDIA. Diese ermöglicht General Purpose Computing auf einer GPU. Hierbei fungiert die GPU als Coprozessor für die CPU und übernimmt alle Datenparallelen und rechenintensiven Teile einer Anwendung.

Definition:

- Device = GPU/Grafikkarte
- Host = CPU
- Kernel = Funktion, die vom Host aufgerufen und auf der GPU ausgeführt wird

Im Gegensatz zu der CPU besteht die GPU aus weit mehr ALU's (Arithmetic Logic Unit). Wie man in der Abbildung 3.1 sehen kann, ist die GPU im Vergleich zur CPU ganz anders konzipiert. Um eine möglichst hohe Rechenleistung zu erhalten, wurden Cache und Kontrolleinheiten klein gehalten. Dies begrenzt natürlich die Programmiermöglichkeiten der Prozessoren und verlangsamt den Speicherzugriff, ist aber optimal für die Berechnung von Algorithmen mit hoher Datenparallelität und Rechenintensität.

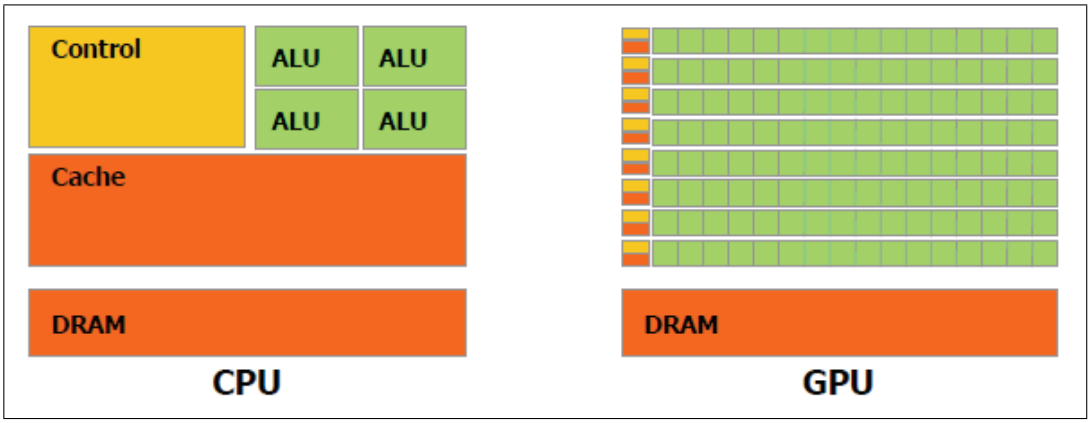

Abbildung 3.1: Vergleich CPU und GPU

[CUD10a] [CUD08]

3.2 CUDA-Software-Architektur

Die CUDA Software Architektur besteht aus mehreren Ebenen der Parallelität, die auf den einzelnen Threads eines parallelen Programms aufbaut. Hierbei besitzt jede Ebene ihren eigenen Speicher.

3.2.1 Thread Hierarchie

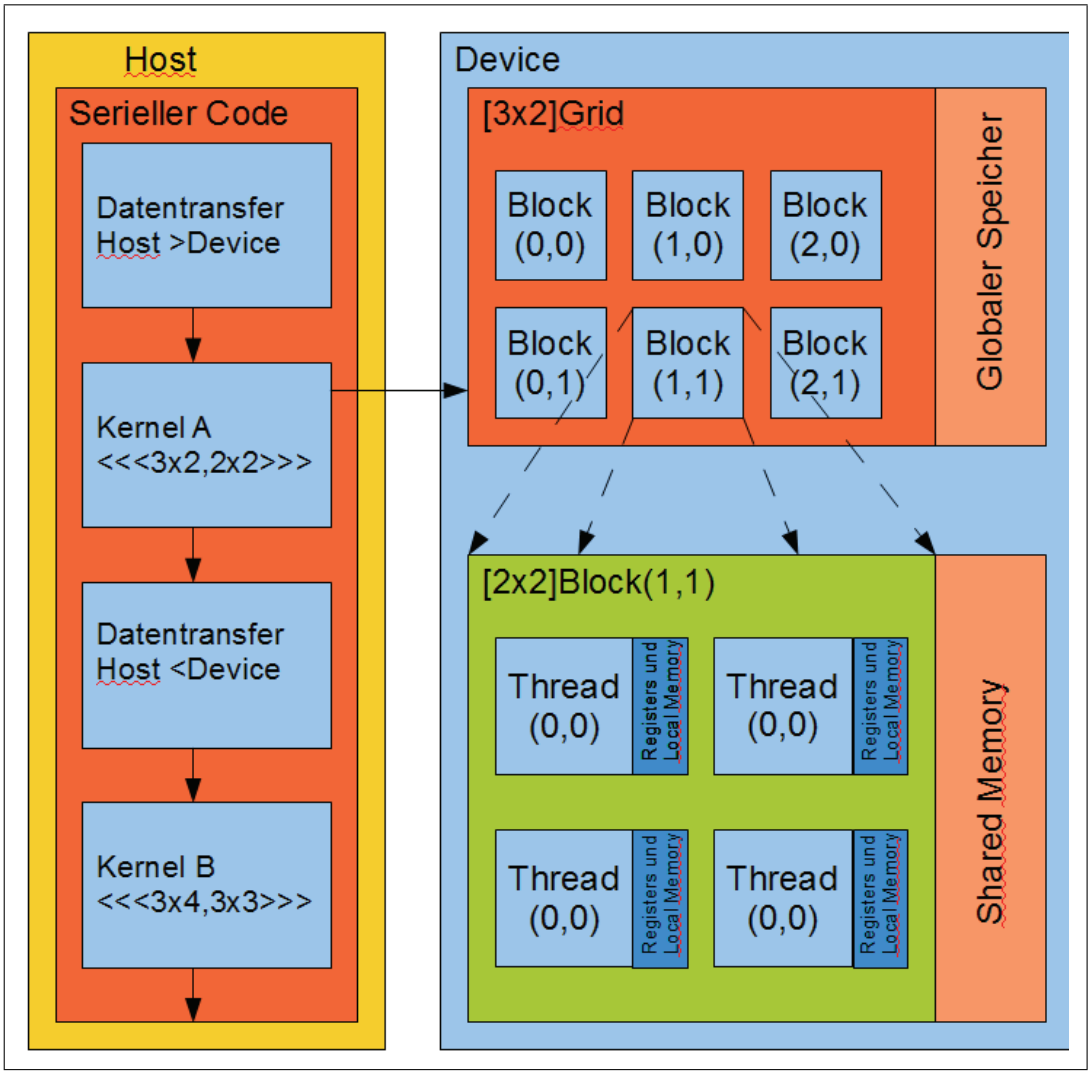

Abbildung 3.2: Strukturhierachie der Software-Architektur. Der Kernelaufruf wird auf dem Device mit einem 3x2 Grid, bestehend aus 2x2 Blöcken, ausgeführt.

Wie in Abbildung 3.2 dargestellt, wird der Seriellencode auf dem Host ausgeführt. Dieser leitet den Kernelaufruf an die GPU. Jeder dieser Kernel wird von mehreren Threads bearbeitet, welche in einem Gitternetz (Grid) von Thread-Blöcken organisiert sind. Hierbei führen die Threads den selben Programmcode aus.

Begriffserläuterung:

Thread Eine Thread ist ein Ausführungsstrang, der ein Programmbefehl für Befehl ab abarbeitet. Dabei kann er auf dem zugehörigen lokalen Speicher Operationen ausführen. Jeder Thread besitzt eine eigene ID, die aus drei Dimensionen besteht.

(Thread-)Block Ein Thread-Block ist die logische Gruppierung mehrerer Threads, die in einem Gitter geordnet werden. Threads aus dem gleichen Block können miteinander kooperieren, in dem sie ihren Ablauf synchronisieren oder die Daten über Shared Memory austauschen. Das ist bei Threads aus verschiedenen Blöcken nicht möglich.

Grid Ein Grid ist die logische Gruppe von Thread-Blöcken. Es organisiert die Blöcke in ein oder zweidimensionalen Gitternetzen. Alle Threads eines Grids führen den selben Programmcode aus.

Kernel Ein Kernel ist der Teil der Anwendung, der sehr oft parallel auf der Grafikkarte ausgeführt wird, aber unabhängig auf verschiedenen Daten operiert. Er wird über `__global__` deklariert und über `<<<..>>>` mit den entsprechenden Parametern aufgerufen. Mehr dazu im Kapitel 3.2.3 Kernausführung.

Stream Ein Stream ist eine frei definierbare Abfolge von Kernel aufrufen und der Datenübertragungen zwischen Host und Device, welche sequentiell durchlaufen werden. Jeder Stream hat sein eigene, vom Programmierer gegebene ID. Es lassen sich mehrere Streams gleichzeitig starten. In der CUDA-Runtime-Library gibt es Möglichkeiten die Lebenszeit und Synchronisation zu überwachen und zu verwalten.

Warp Ein Warp besteht aus 32 Threads, von aufeinander folgender IDs eines Blocks, die physisch parallel ausgeführt werden. Ist die Anzahl der Threads eines Blocks nicht durch 32 teilbar, so wird der letzte Warp mit auswirkungslosen Threads aufgefüllt.
[CUD10a] [CUD08] [Zel08] [GR08]

3.2.2 Speicher-Hierarchie

Wie man in der Abbildung 3.2. sieht, haben Threads verschiedene Speicherplätze zur Verfügung. Jeder Thread hat einen privaten lokalen Speicher. Alle Threads eines Blocks haben einen gemeinsamen Speicher (Shared memory), der die gleiche Lebenserwartung hat wie der Block selbst. Sämtliche Threads haben Zugriff auf dieselben globalen Speicher (Global memory). Des Weiteren gibt es noch zwei zusätzliche read-only Speicher, die wiederum von allen Threads genutzt werden können. Sie werden hauptsächlich zur Verbesserung der Performance eingesetzt: Konstant (constant memory) und Textur (texture memory)-Speicher. Eine Übersicht über die verschiedenen Zugriffsmöglichkeiten befinden sich in der Tabelle 3.1. Der globale, konstante und Textur-Speicher sind für verschiedene Speichernutzungen optimiert.

Begriffserläuterung

Local memory Der lokale Speicher wird anstelle von Registern genutzt, z . b . für Arrays, bei denen die Größe zur Kompilierungszeit noch unbekannt ist oder bei anderen größeren Strukturen. Des Weiteren wird der lokale Speicher als eine Art Überlauf für die Variablen verwendet, wenn der Kernel mehr Register nutzt als zu Verfügung stehen.

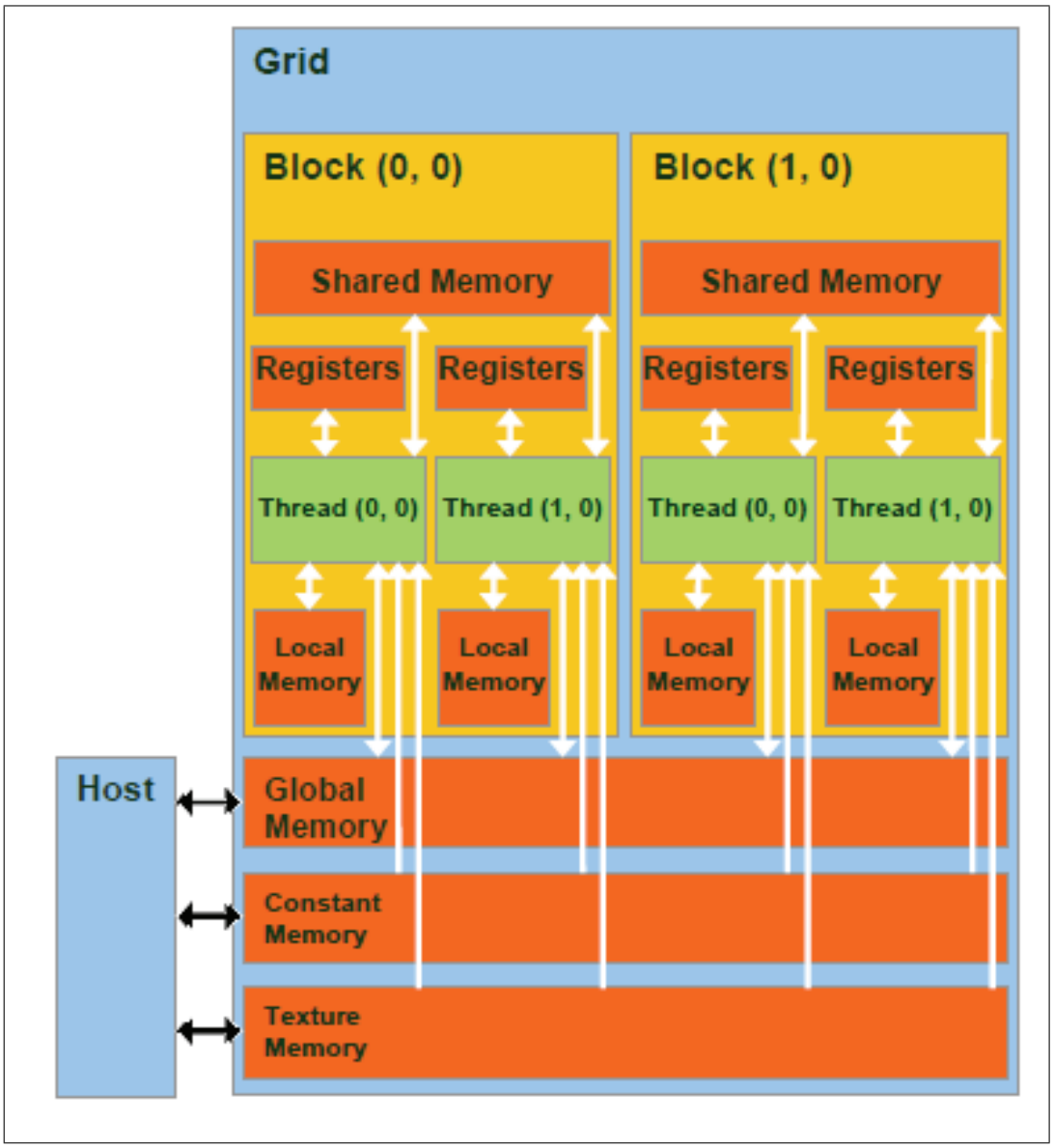

Abbildung 3.3: Aufbau des Speicher Modell einer CUDA-Karte

Texture Memory Der Texturspeicher wird als nicht-beschreibbar behandelt. Er kann nur vom Host alloziert werden, dafür werden aber einige statische Parameter benötigt. Er bietet verschiedene Adressierungsarten, sowie Daten-Filterung, für bestimmte Datenformate. Hat aber eine hohe Latenzzeit und wird gecacht.

Constant Memory Der konstante Speicher hat eine Größe von 64 kB und befindet sich im DRAM des Device. Dieser Speicherbereich kann auch wiederum nur vom Host beschrieben werden, hat aber eine sehr niedrige Latenzzeit im Vergleich zu anderen Speichern. Er kann statisch oder vom Host auch dynamisch alloziert und initialisiert werden. Wie der Texturspeicher wird er vorher gecacht.

Global Memory Auf diese Weise allozierter Speicher kann von allen Prozessorkernen auf der GPU gelesen und beschrieben werden. Der Zugriff wird nicht gecacht, somit muss für eine hohe Performance der Zugriff minimiert werden. Die Initialisierung kann statisch oder durch den Host dynamisch erfolgen.

Register Die Register dienen zum Zwischenspeichern von Befehlen, Speicheradressen und Rechenoperanden. Mit einer Größe von 32-Bit und 8 bis 32 tausend Register pro Multiprozessor, je nach Compute Capability(siehe Kapitel 3.2.6 Compute Capability), ist er der schnellste Speicher. Er benötigt keinen extra Taktzyklus pro Anweisung, aber Verzögerungen können aufgrund der read-after-write Regel auftreten oder durch Speicherbank-Konflikte.

Shared Memory Der parallele Datencache oder Shared Memory wird von allen Prozessorkernen (SIMT) eines Blocks gemeinsam genutzt. Jeder Multiprozessor verfügt über 16 kB oder 48 kB, abhängig von der Compute Capability. Von allen Kernen kann er parallel gelesen und beschrieben werden. Er ist in 16 oder 32 Speicherbänken organisiert und ermöglicht dieselben Zugriffszeiten wie auf Register, sofern keine Bank-Konflikte entstehen.
[CUD10a][Zel08]

Speicher	Thread greift zu via	Zugriffsmöglichkeiten
Register	Thread	Lese/Schreiben
Shared Memory	Block	Lese/Schreiben
Global Memory	Grid	Lese/Schreiben
Local Memory	Thread	Lese/Schreiben
Constant Memory	Grid	Lese
Texture Memory	Grid	Lese

Tabelle 3.1: Übersicht über die Zugriffsmöglichkeiten auf die verschiedenen Speicher

3.2.3 Software Stack

Der CUDA-Software-Stack besteht aus mehreren zusammengesetzt Schichten, wie in der Abbildung 3.4 dargestellt: Beim Schreiben von CUDA-Anwendungen können diese entweder auf der Treiberebene (CUDA Driver API) arbeiten oder in der Runtime-Ebene (CUDA Runtime API). Es sollte aber immer nur eine von beiden API's genutzt werden. Die CUDA Laufzeit verlinkt und kompiliert die CUDA Kernels ohne das man sich um das Laden von

Treiber und generieren von CUBIN Dateien(beinhalten Spezifikationen der Ziel Architektur) kümmern muss. Es gibt keinen spürbaren Performance-Unterschied zwischen den API's. Es fällt mehr ins Gewicht, wie die Verteilung der Threads und der Blöcke innerhalb eines Kernels gelöst wird. Eine weitere Schicht bilden die Bibliotheken (CUDA Libraries), zu den wichtigsten gehören: [sta09] [CUD08]

CUFFT CUFFT ist die Implementierung des FFT (Fast Fourier Transform) Algorithmus in CUDA. Die FFT ist ein Divide-and-Conquer-Algorithmus für die effiziente Berechnung diskreter Fourier-Transformation von komplexen oder Echtzeit-Daten und ist einer der wichtigsten und am weitesten vorbereiteten numerischen Algorithmen, z.B. in der Signalverarbeitung. [CUD10c]

CUBLAS CUBLAS ist die Implementierung der BLAS (Basic Linear Algebra Subprograms) Softwarebibliothek. BLAS ist eine Softwarebibliothek, die elementare Operationen der linearen Algebra wie Vektor- und Matrixmultiplikationen implementiert. [CUD10b]

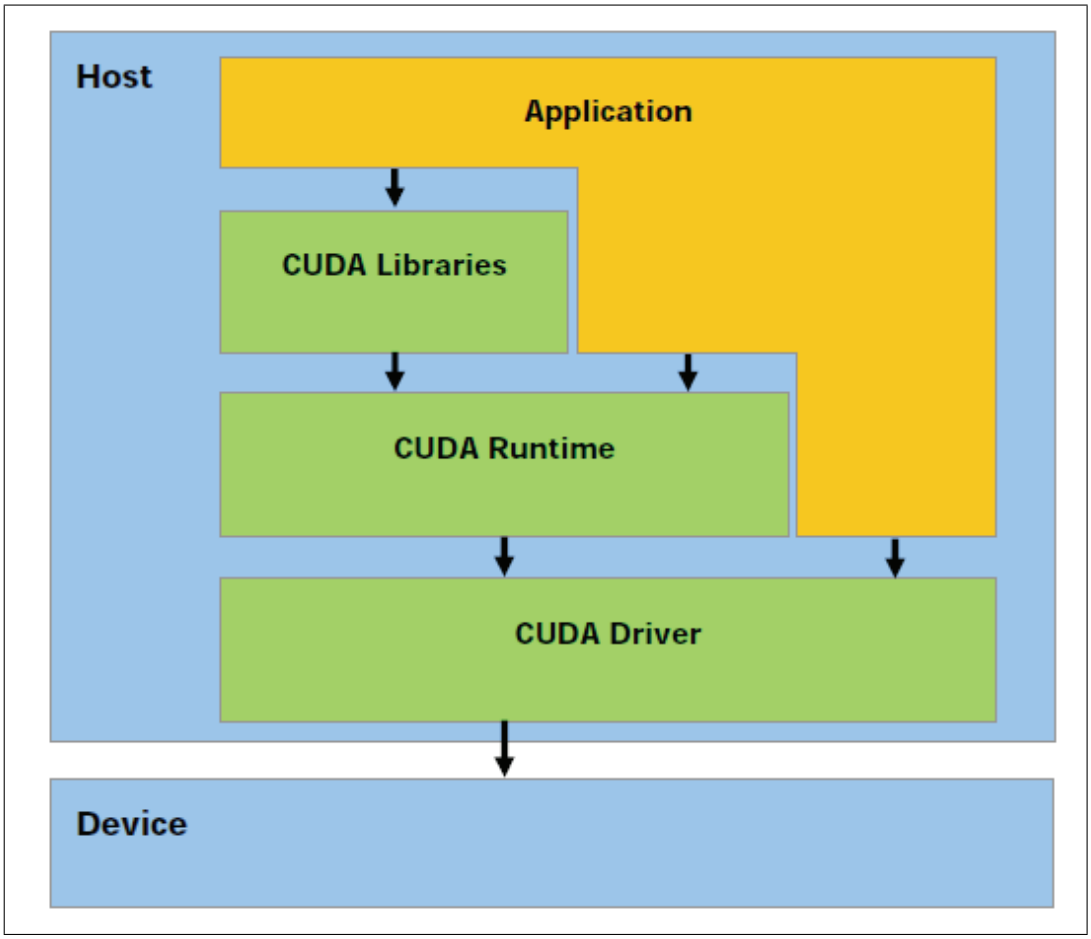

Abbildung 3.4: Modell des CUDA Software Stack

3.2.4 Prinzipieller Aufbau von Cuda-Programmen

Funktionen Qualifiers: gibt an ob eine Funktion auf dem Host oder auf dem Gerät ausgeführt wird und ob es Callable vom Host oder vom Gerät ist.

- `__global__` : Wird auf der GPU ausgeführt, nur vom Host aufrufbar.
- `__device__` : Wird auf der GPU ausgeführt, nur vom Device aufrufbar.
- `__host__` : Wird auf der CPU ausgeführt, nur vom Host aufrufbar, ist äquivalent zu den Aufrufe ohne Qualifier.

Built-In Variablentypen:

- gridDim: Dimension des Grids
- blockDim: Dimension eines einzelnen Blocks
- blockIdx: Index des Blocks in einem Grid
- threadIdx: ThreadIndex in einem Block
- warpsize: Größe eines Warps

Variablentypen: Zusätzliche Qualifier für Variablen

- `__Device__`: Variable verbleibt für die gesamte Zeit im globalen Speicher der Grafikkarte.
- `__shared__`: Variable verbleibt für die Lebenszeit eines Blocks im Shared Speicher der GPU. Zugriff nur durch Threads aus dem gleichen Block.
- `__constant__`: Variable verbleibt im konstantem Speicherbereich.

Kernel Ausführung: Kernel sind C-Funktionen mit einigen Einschränkungen. Sie dürfen nicht auf den Host-Speicher zugreifen, der Rückgabetyp muss void sein, nicht rekursiv sein und keine statischen Variablen besitzen. Alle `__global__` Funktionen müssen mit einer Kernel-Konfiguration aufgerufen werden. Spezifiziert wird diese durch das Einfügen von`<<< Dg, Db, Ns, S >>>` zwischen Funktionsname und Parameterliste.

- Dg: Angabe über Größe und Dimension des Grids
- Db: Angabe über Größe und Dimension eines einzelnen Blocks
- Ns: Anzahl an Bytes, die bei jedem Aufruf in den Shared-Speicher geladen werden (optional)
- S: Zusätzliche Angabe eines Streams vom typ cudaStream (optional)

[Roh08] [CUD10a]

3.2.5 Compiler

Wie in der Abbildung dargestellt, bestehen die grundlegenden Arbeitsschritte des NVCC's, in der Trennung des Device-Codes vom Host-Code und das Kompilieren des Device Codes in binäre Form oder in Cubin Objekte. Der C-Code wird dann vom Standard C-Compiler verarbeitet. Für den Host-Code wird die komplette C++ Syntax unterstützt. Für den Device-Code wird jedoch nur eine kleine Teilmenge unterstützt, z . B . werden keine Klassen oder Vererbungen unterstützt. NVCC beinhaltet zwei Compiler-Direktiven:

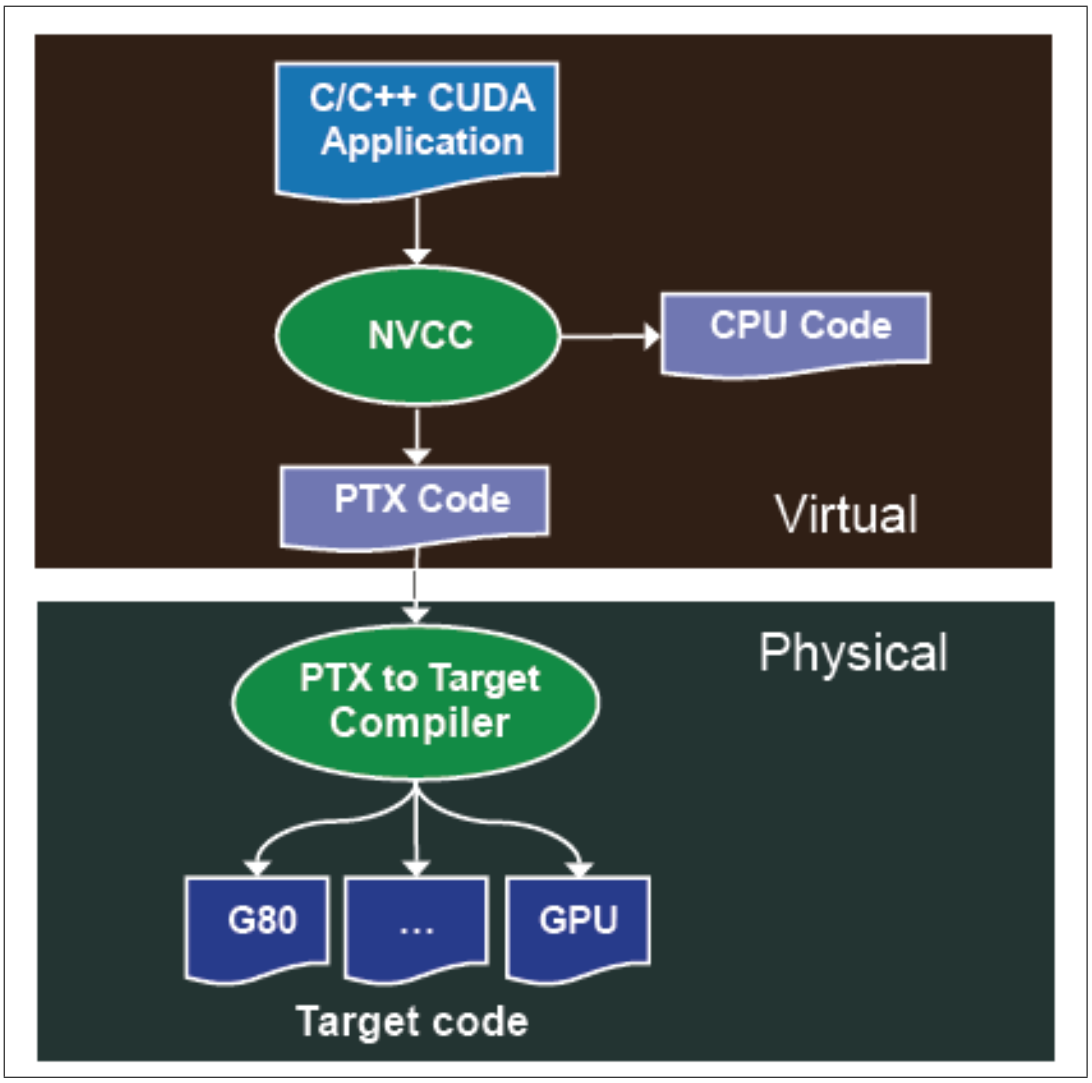

Abbildung 3.5: Ablauf einer Kompilierung mit NVCC

- `__noinline__` : Standardmäßig ist eine `__device__` Funktion immer inline. Die `__noinline__`Qualifier Funktion kann jedoch als Hinweis für den Compiler verwendet werden, die Funktion nicht als Inline zu nutzen, sofern es möglich ist. Der Compiler wird dessen nicht nach kommen, wenn in der Funktion Pointer oder längere Parameterlisten sind.

- `#pragma unroll`: Bei kleineren Schleifen, deren Durchlaufzahl bekannt ist, werden die Schleifen entrollt. Dieses kann mit dem Befehl unroll verhindert werden.

[CUD08]

3.2.6 Compute Capability

Die compute capability beschreibt die Berechnungs-Fähigkeit einer CUDA-Karte. Wobei Karten mit der gleichen Haupt-Revisions-Nummer die gleiche Kern-Architektur haben. Die zweite Ziffer beschreibt die inkrementellen Verbesserungen der Kern-Architektur. Für genauere Informationen siehe Anhang A und G des NVIDIA CUDA C Programming Guide 3.

[CUD10a]

3.3 CUDA Implementation auf der GPU

Die zwei wesentlichen Komponenten sind die Graphics Processing Unit (GPU) und der Device Memory (DRAM). Der DRAM dient insbesondere dem Datenaustausch zwischen Host und GPU. Dieser kann statisch oder dynamisch vom Host alloziert werden. Der Host kann mittels Direct Memory Access (DMA) auf sämtlichen allozierten Speicher zugreifen (lesend und schreibend). Es gibt drei Alternativen (Texture Memory, Constant memory, Global Memory), die im Punkt 3.2.2 näher beschrieben sind.

3.3.1 Graphics Processing Unit (GPU)

Die CUDA-Architektur ist aufgebaut aus skalierbaren Arrays, von Multithread-Streaming Multiprozessoren. Wobei ein Multiprozessor aus 8 oder 32, je nach Compute Capability, Bit-Streaming-Prozessoren besteht. Die aus zwei speziellen Funktionseinheiten bestehen, eine für Transzendentalien (nicht algebraische Funktion) und eine Multithread-Instruction-Unit. Jeder Prozessorkern verfügt über eine eigene Arithmetic Logical Unit (ALU), die auf den Registern des Kerns operiert. Außer den üblichen Basisoperationen stehen ihr Hardwareimplementierungen häufig genutzter mathematischer Funktionen zur Verfügung. Der Multiprozessor erstellt, verwaltet und führt die Threads in Gruppen von 32 parallelen Threads aus, die auch als Warp bezeichnet werden. Threads in dem selben Warp starten mit der gleichen Programmadresse, haben aber ihren eigenen Adresszähler und Registerstand und sind daher unabhängig in der Ausführung und Verzweigung. Dieses wird auch als SIMT (Single-Instruction, Multiple-Thread) Architektur bezeichnet, die stark an der SIMD (Single Instruction, Multiple Data) Architektur angelehnt ist. Im Gegensatz zu der SIMD Architektur, ermöglicht SIMT den Programmierer Thread-Level-parallelen-Code für unabhängige, skalare Threads zu schreiben, sowie Daten-parallelen-Code für koordinierte Threads. Innerhalb jedes Multiprozessors gibt es vier Speicherressourcen, die von den Prozessorkernen genutzt werden können. Der Globale-(global memory) und der Verteilte Speicher (shared Memory) sind im Punkt 3.2.2 beschrieben:

Constant Cache Constant Cache ist ein read-only Speicher und wird von allen Prozessoren auf der Karte genutzt. Dieser 8 kB-Cache, pro Multiprozessor hält zuvor angeforderte Daten aus dem Constant Memory, um dessen Zugriff zu beschleunigen. Seine Zugriffszeiten entsprechen dann dem eines Registerzugriffs.

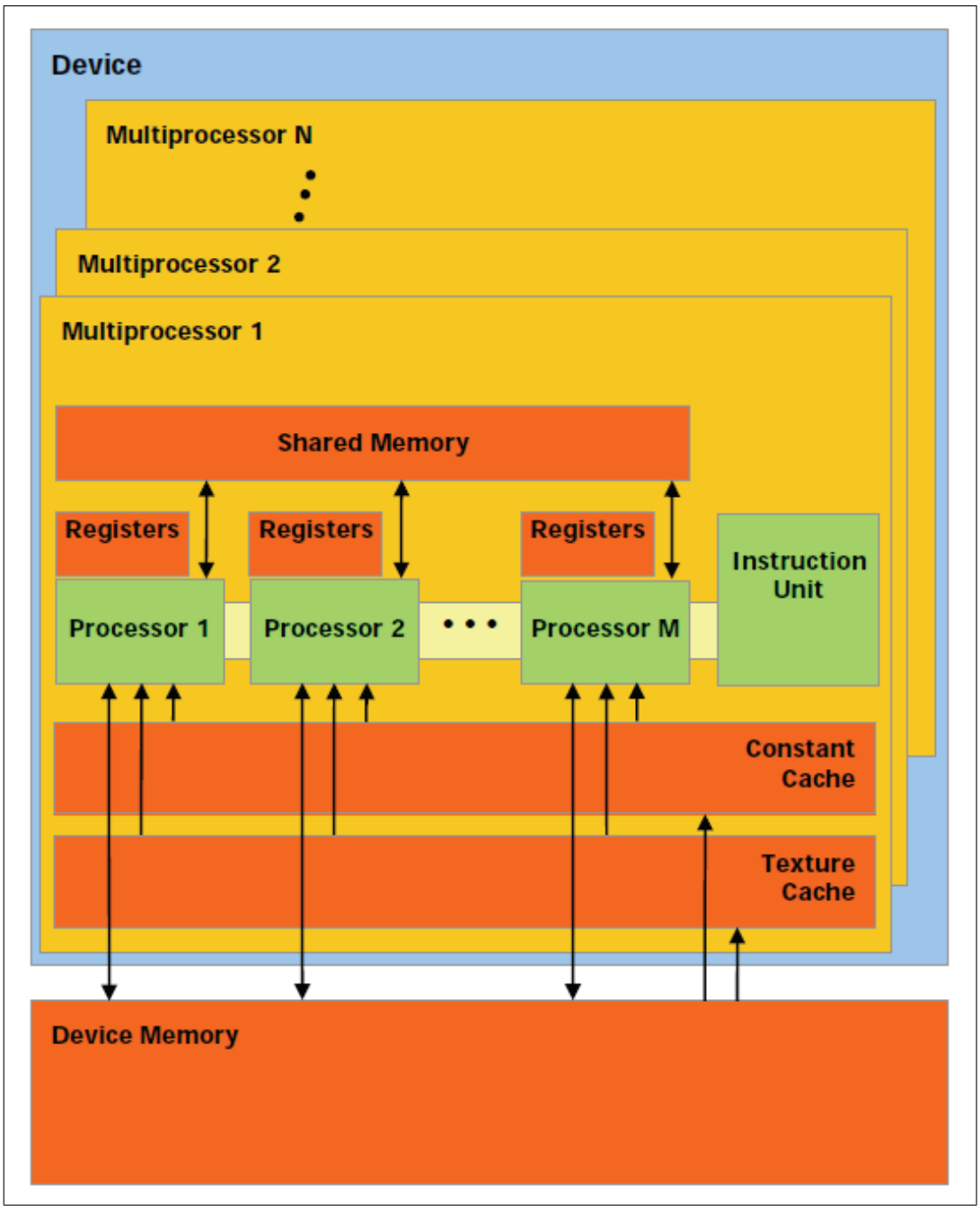

Abbildung 3.6: Abbildung von SIMT Multiprozessoren mit On-Chip-Shared-Memory.

Texture Cache Ist ebenfalls ein read-only Speicher der von allen Prozessoren genutzt werden kann. Seine Größe ist vom Gerät abhängig und beträgt zwischen 6 KB und 8 KB pro Multiprozessor. Er cacht die Daten aus dem Texturspeicher um ebenfalls den Zugriff zu beschleunigen. [CUD10a][CUD08]

4 JPEG

4.1 Einführung

Bei JPEG (Joint Photographic Expert Group) handelt es sich um ein Bildkomprimierungsalgorithmus, nicht um ein Dateiformat. JPEG ist heute einer der bekanntesten und verbreitetsten Standards zur Kompression von Bildern, besonders von Fotos. Das JPEG-Komitee ist ein Zusammenschluss verschiedener Standardisierungsorganisationen, die den Standard Anfang der 90er Jahre entwickelt und schließlich 1992 vorgestellt haben (ISO/IEC 10918-1). Die JPEG-Norm beschreibt lediglich den Komprimierungsalgorithmus, legt aber nicht fest, wie die Daten gespeichert werden sollen. Das gebräuchlichste Format einer JPEG-Datei ist das Grafikformat JPEG File Interchange Format (JFIF). Des weiteren gibt es noch SPIFF (Still Picture Interchange File Format) und JNG (JPEG Network Graphics). JPEG ist ein Standard, der möglichst allen Anforderungen bei der Bilddatenkompression gerecht werden soll. Da es nicht einen einzelnen Algorithmus gibt, der alles kann, legte das JPEG-Gremium (Joint Picture Expert Group) verschiedene Verfahren fest, die jeweils für einen bestimmten Teilbereich sehr gute Ergebnisse liefern. Der JPEG-Standard beschreibt zwei verschiedene Kompressionsverfahren. Das JPEG-Lossless-Verfahren ist ein verlustfreies Kompressionsverfahren, welche alle nach dem gleichen Schema arbeiten. Die anfallenden Bilddaten werden zuerst in Deskriptoren transformiert. Anhand dieser wird ein statisches Modell (Verschlüsselungstabelle) erstellt, wodurch die Deskriptoren später codiert werden. Als Transformationverfahren wird hier die differentielle Pulscodemodulation eingesetzt. Zum codieren der Daten benutzt man die Huffmann-Kodierung oder die arithmetische Kodierung. Das verlustfreie Kompressionsverfahren wird überall da eingesetzt, wo keine Fehler auftreten dürfen (z.B.bei mechanischer Bildauswertung). Bei der verlustbehafteten Kompression werden die zu komprimierenden Bildaten erst in einen anderen Farbraum, YCbCr oder YUV, gebracht. Danach wird es transformiert (Diskrete Cosinus Transformation), quantisiert und codiert. Die Informationverluste entstehen durch die Quantisierung und durch Rundungsfehler. Das verlustfreie Kompressionsverfahren besitzt 3 verschiedene Operationsmodi:

Sequential Modus Der Sequential Modus oder auch Baseline-JPEG genannt, wird heute für die meisten Anwendungen genutzt, die JPEG verwenden. Dieses Verfahren ermöglicht es, sowohl Graustufen als auch Echtfarbbilder mit sehr hohen Kompressionsraten und relativ geringen Qualitätsverlusten zu komprimieren. Bei diesem Modus werden die Bilder erst in einen anderen Farbraum umgewandelt, wo dann die einzelnen Farbebenen (Y-, Cb- und Cr-Ebene oder auch in das YUV-Farbmodell) horizontal und vertikal transformiert werden. Dieses hat den Vorteil, dass das Verfahren sowohl Graustufen als auch Echtfarbbilder mit sehr hohen Kompressionsraten und relativ geringen Qualitätsverlusten komprimieren kann.Abbildung 4.1.

Progressiv Modus Beim Progressiv Modus wird das Bild in mehreren Durchgängen kodiert, um so die Bildqualität schrittweise zu erhöhen. Dieser Modus ist nützlich, vor allem bei

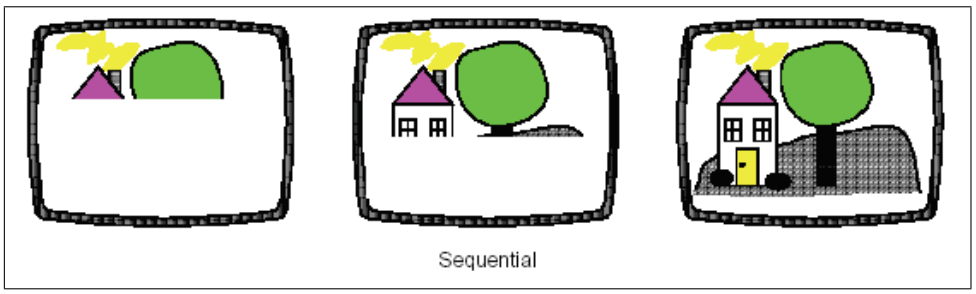

Abbildung 4.1: Sequential Modus

langsamen Datenübertragungskanälen. Der Empfänger bekommt schnell ein Bild geliefert, das dann etappenweise verbessert wird. So bekommt der Empfänger schnell einen Überblick und kann die Übertragung, bei ausreichender Qualität, abbrechen. Abbildung 4.2.

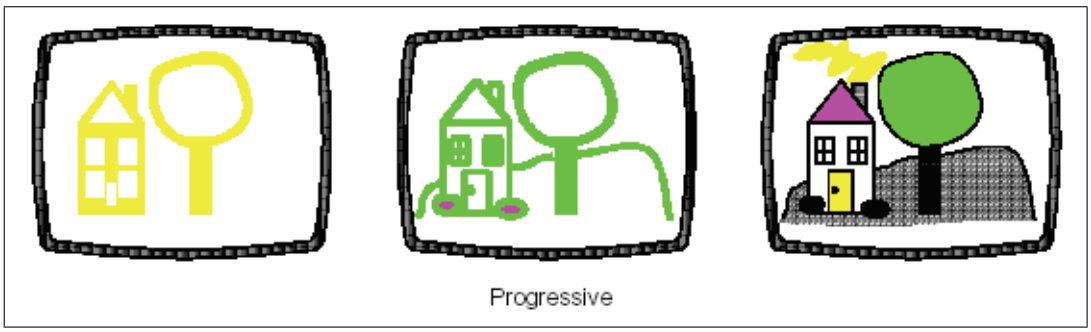

Abbildung 4.2: Progressiv Modus

Hierarchial Modus Dieser Modus ist eine Form des Progressive Modus. Hier wird das Bild sowohl in der vollen Auflösung, als auch in einer geringeren Auflösung gespeichert. Der Vorteil ist dass das gering aufgelöste Bild wesentlich schneller übertragen und dekodiert werden kann. Es eignet sich somit bestens als schnelle Vorschau auf das eigentliche Bild. Bei Interesse kann dann die voll aufgelöste Version geladen werden.
[Sch09] [uMG02] [Tem05] [Men99]

4.2 Baseline JPEG-Verfahren

Diese Arbeit befasst sich mit dem verlustbehafteten, sequentialen Modus. Er besteht aus 6 Schritten:

- Konvertierung des Bildes in den YUV-Farbraum
- Downsampling (optional)
- Diskrete Cosinus Transformation
- Quantisierung der DCT-Koeffizienten
- Zick-Zack-Sortierung

- Kodierung der Koeffizienten (Entropie Kodierung)

4.2.1 Konvertierung des Bildes in den YUV-Farbraum

Ein RGB Farbbild besitzt drei Farbkanäle (Rot-Grün-Blau), welche zusammen einen Farbraum bilden. Hier werden die Farben additiv über ihre Rot-, Grün- und Blauanteile dargestellt. Da das menschliche Auge Kontrastunterschiede viel genauer wahr nimmt, als Unterschiede im Farbverlauf, bietet sich die verlustbehaftete Umwandlung vom RGB-Farbraum zum YUV oder YcbCr-Farbraum an. Im RGB-Farbraum sind die Kontrastinformation in allen drei Farbkanälen gespeichert. Dabei wird bei der Transformation die gesamte Helligkeitsinformation in einem Y-Kanal gespeichert, der auch Luminanzkanal genannt wird. Die Farbinformation steckt in den U- und V-Kanälen, den Chrominanzkanälen.

Vom RGB- ins YUV-Modell:

$$Y = 0,299R + 0,587G + 0,114B$$

$$U = 0,493(B - Y)$$

$$V = 0,877(R - Y)$$

oder
vom RGB- ins YCbCr-Modell

$$Y = 0,299R + 0,587G + 0,114B$$

$$Cb = -0,1687R - 0,3313G + 0,5B + 128$$

$$Cr = 0,5R - 0,4187G - 0,0813B + 128$$

4.2.2 Downsampling

Downsampling (Sub-Sampling) bietet die Möglichkeit, diese Kompressions-Parameter zu beeinflussen: Die Luminanz bleibt dabei in der vollen Auflösung (1:1), die Chrominanz-Kanäle werden um Faktor zwei reduziert. In der JPEG-Terminologie wird dies RGB 4:4:4, YUV 4:2:1 oder YUV 4:2:2 genannt. Dieses geschieht durch einfaches Mitteln der Werte, die anstatt der gesamten Chrominanzwerte codiert werden. Dieser Schritt verkleinert ohne Qualitätsverluste das Datenvolumen um den Faktor zwei oder drei.

4.2.3 Diskrete Cosinus Transformation

Im nächsten Schritt wird das Bild in 8x8 große Pixelblöcke aufgeteilt. Jeder dieser Blöcke wird mit der diskreten Cosinus-Transformation bearbeitet. Diese wandelt die drei Kanäle (YUV) in Frequenzwerte um. Die Umwandlung selbst ist verlustfrei bis auf die Rundungsfehler. Auf diese Weise ergeben sich 64 DCT-Koeffizienten für jeden Pixelblock. Der oben links im Block stehende Koeffizient wird DC-Komponente (von Gleichstrom, engl. Direct Current) genannt. Je weiter man sich von ihm entfernt, desto höher werden die zugehörigen Frequenzen. Da das menschliche Auge für höhere Frequenzen weniger empfindlich ist, kann dieser Anteil je nach

Höhe des Kompressionsfaktors vernachlässigt werden. Die restlichen 63 DCT-Koeffizienten werden AC-Komponente (von Wechselstrom, engl. Alternating Current) genannt.

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos[\frac{(2x+1)i\pi)}{2N}] \cos[\frac{(2y+1)j\pi}{2N}]$$

falls X=0 oder Y=0

$$C_{i,j} = \frac{1}{\sqrt{2}}$$

sonst 1

4.2.4 Quantisierung der DCT-Koeffizienten

Um die Anzahl der DCT-Koeffizienten zu verringern, wird jeder Wert durch einen zugeordneten Quantisierungs Wert dividiert und auf die nächste ganze Zahl gerundet. Die Q-Werte werden dabei Tabellen entnommen, die das JPEG-Komitee anhand psycho-visueller Tests an einer Vielzahl von Personen ermittelt hat. Somit wird beim JPEG-Verfahren der Kompressionsfaktor nicht direkt eingestellt, sondern ein Q-Faktor ausgewählt, der einer bestimmten Kompression entspricht.

$$F^Q(u,v) = Round \frac{F(u,v)}{Q(u,v)}$$

4.2.5 Zick-Zack Sortierung

Durch die Abtrennung der hohen Frequenzanteile entsteht immer eine Reihe von Null-Werten, die durch Abzählen zusammengefasst werden. Die von Null verschiedenen quantisierten DCT-Koeffizienten werden durch ein Zick-Zack-Schema in eine lineare Zahlenfolge umgesetzt, die im letzten Schritt der Entropie-Kodierung bearbeitet werden.

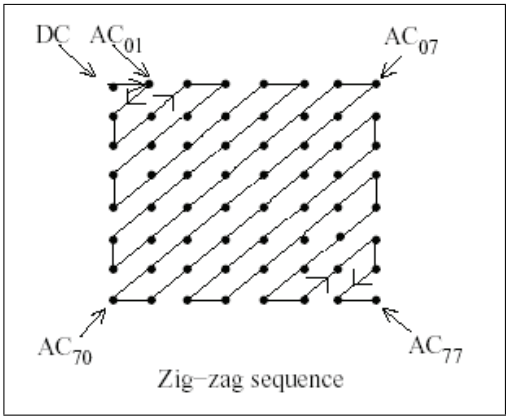

Abbildung 4.3: Abbildung der Zick-Zack-Sortierung

4.2.6 Kodierung der Koeffizienten (Entropie Kodierung)

Nach dem Zick-Zack-Verfahren wird zuerst die Lauflängen Kodierung und dann die Entropie Kodierung durchgeführt. Die Lauflängen Kodierung nützt es aus, dass die Sequenz der AC-Koeffizienten eine große Anzahl von Werten beinhaltet, die gleich Null sind. Jeder AC-Koeffizient, der ungleich Null ist, wird mit zwei Symbolen dargestellt, die folgendermaßen aufgebaut sind:

$$\begin{pmatrix} Symbol1 \\ Groesse/Lauflaenge \end{pmatrix} \begin{pmatrix} Symbol1 \\ Amplitude \end{pmatrix}$$

Wobei das zweite Symbol den AC-Koeffizient wieder gibt. Das erste Symbol beschreibt, wie viele Bits zur Kodierung des Amplituden Werts benötigt werden. Die abschließende Null wird mit einem End-of-Block-Code (EOB) kodiert. Darauf folgt die Entropie Kodierung. Grundsätzlich werden von JPEG zwei Entropie Kodierungen unterstützt:

- die Huffmann Kodierung
- die arithmetische Kodierung

Die arithmetische Kodierung, die um 5 % bis 15 % besser komprimiert als die Huffmann Kodierung, wird aber aus patentrechtlichen Gründen kaum verwendet. Beim Baseline JPEG benutzt man als Entropie Kodierung nur die Huffmann Kodierung. Bei dieser Kodierung wird für die Zuordnung des Codes zu den Symbolen keine externe Tabelle benötigt, wenn die Standard Huffmann Tabelle verwendet wird. Es können aber auch eigene Tabellen erstellt werden, die jedoch dem Decoder mitgeteilt werden müssen. [uMG02] [Sch09]

0	0		0
1	-1	1	10
2	$-3,-2$	$2,3$	110
3	$-7,\ldots,-4$	$4,\ldots,7$	1110
4	$-15,\ldots,-8$	$8,\ldots,15$	11110
5	$-31,\ldots,-16$	$16,\ldots,31$	111110
6	$-63,\ldots,-32$	$32,\ldots,63$	1111110
$\vdots$	$\vdots$	$\vdots$	$\vdots$
15	$-32767,\ldots,-16384$	$16384,\ldots,32767$	1111111111111110
16	32768		1111111111111111

Abbildung 4.4: Huffmann Tabelle: Einteilung der Zahlen in Klassen

4.3 JPEG Encoder Parallelisierungsmöglichleiten

Der JPEG Baseline Algorithmus eignet sich sehr gut zum Parallelisieren, denn es werden wenige Speicherverschiebungen von und zum Device benötigt. Nur zwei Aufrufe von `cudaMemcpy` werden benötigt, zum Speichern und Laden der Bilder. Bedingt dadurch, dass die Bilder klein genug sind, um komplett in den Device Memory zu passen. Des Weiteren kommt zugute, das sich Pixel von Natur aus gut zum Parallelisieren eignen, da keine Abhängigkeiten der Pixel untereinander bestehen.

4.3.1 CUDA Spezifische Planungsregeln zur Parallelisierung

Die Aufgaben sollten so eingeteilt sein, dass die Hardware möglichst voll ausgelastet wird. Das heißt, das mindestens so viele Blöcke wie die GPU Multiprozessoren (SIMT-Einheit) hat,

verwendet werden. Dadurch kann der Multiprozessor im Idealzustand die Blöcke wechseln, sobald ein `synchronize` bzw. ein Speicherzugriff auftritt. Das bedeutet, dass so Latenzzeit umgangen werden kann. Daher ist es besser mehrere Blöcke pro Multiprozessor einzuplanen, um zwischen Blöcken wechseln zu können. Des Weiteren sollte die Anzahl der Threads pro Block ein Vielfaches der Warp-Size sein, besser noch ein Vielfaches von 64 (laut NVIDIA). Der Compiler und der Thread-Scheduler versuchen die Instruktionen so gut wie möglich zu koordinieren um Register-Bank-Konflikte zu vermeiden. Dabei sollte aber beachtet werden, dass je mehr Threads benutzt werden, auch weniger Register pro Thread verfügbar sind.[Unb]

4.3.2 Parallelisierungsmöglichleiten

Sobald die drei Komponenten des Bildes in den globalen Speicher gebracht wurden, können alle Schritte von der Farbraumänderung bis zur Zick-Zack-Sortierung parallelisiert werden.

Farbraumumwandlung

Da schon vom JPEG Algorithmus die Bilder in 8x8 Blocke geteilt werden müssen und jedes Pixel aus drei Komponenten besteht, setzt sich jeder Block (8x8x3) aus 192 Threads zusammen. Die Berechnung der Farbräume erfolgt wie in Punkt 4.2.1 beschrieben. Die Größe des Grids ergibt sich aus der Breite/8 und der Höhe/8 des Bildes.

Downsampling

Kann im oberen Schritt mit angefügt werden. Berechnung wie in Punkt 4.2.2.

DCT

Das Parallelisieren der DCT ist ein bisschen komplizierter, da alle Elemente in einer 8x8 Matrix zur Berechnung benötigt werden. Hierzu wird die 2D DCT in 2*1D DCT (siehe Formel im Punkt 4.2.3) umgewandelt für jede Spalte und Zeile. Die am Ende wieder miteinander addiert werden, um die 2D DCT Koeffizienten zu erhalten. Für jede 8x8 Matrix werden 384(4*4*24) Threads benötigt.Es werden 4x4 1D DCT, für jede Spalte und Zeile, und 8x3 für jedes Pixel mit seinen drei Komponenten. Für diesen Kernel würde sich das Verschieben der Zwischenergebnisse in den Shared Memory Bereich anbieten. Infolgedessen das alle Elemente für die Berechnung mehrmals gebraucht werden. Die Gesamtanzahl an Threads berechnet sich aus Höhe/8 *Breite /8*24.

$$DCT(j) = \frac{1}{2} C_j \sum_{x=0}^{7} pixel_x \cos \frac{(2x+1)j\pi}{16}$$

falls X=0, sonst 1

$$C_j = \frac{1}{\sqrt{2}}$$

Quantisierung

Wird genauso behandelt wie im Schritt Farbraumumwandlung beschrieben, außer das jedes Pixel jetzt durch einen Quantisierungs-Koeffizienten geteilt wird. Die Quantisierungs-Koeffizienten sind in einer Quantisierungstabelle spezifiziert(JPEG Spezifikation).

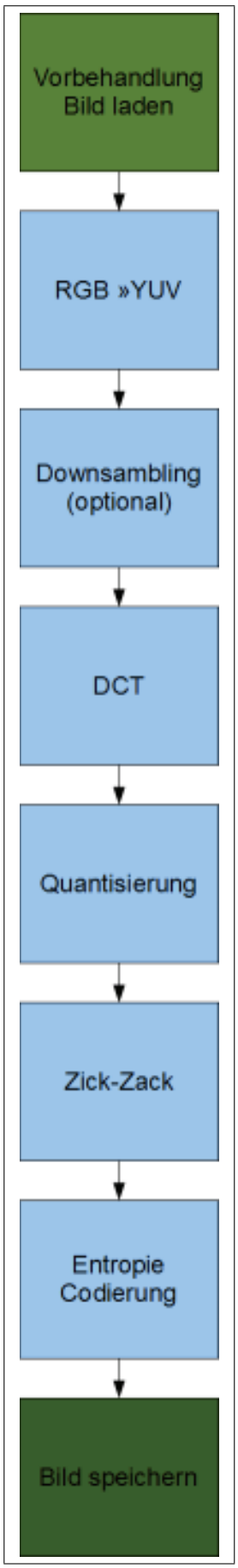

Abbildung 4.5: Grober Ablauf des JPEG Encoders

Zick-Zack

Ist wieder eine einfache Pixel-Operation, wie im Schritt Farbraumumwandlung beschrieben. Hierbei wird eine Lookup-Tabelle zur Positionbestimmung genutzt. Da dieser Teil sonst nicht parallelisiert werden kann und von der CPU bearbeitet werden müsste.

Entropie Kodierung

Bei der Entropie Kodierung ist die Parallelisierung nicht sinnvoll. Aufgrund, das die Huffmann Kodierung ein Serieller Prozess ist und viele Abhängigkeiten zu vorgängigen Pixel bestehen (siehe Lauflängen Kodierung). Des Weiteren ist der Algorithmus sehr kontrollintensiv. CUDA ist nicht für Kontroll-Strukturen ausgelegt, daher würde die Huffmann Kodierung langsamer laufen als auf der CPU.

5 Prototypische Umsetzung

Erst wurde der JPEG-Encoder in C++ bis zu Huffmann Kodierung umgesetzt, um dann einzelne Funktionen mit CUDA Kernels auszutauschen. Der C++ Quellcode befindet sich auf der CD. Zur Erstellung wurde Visual C++ 2008 Express von Microsoft genutzt.

5.1 Umsetzung der Parallelisierung des Quantisierung Moduls

Um den Geschwindigkeitsgewinn durch den Einsatz einer CUDA fähigen Grafikkarte zu analysieren, wurde das Quantisierung Modul des JPEG-Encoders parallelisiert. Bedingt dadurch, dass der Aufbau der anderen Module, außer das DCT Modul, sehr ähnlich ist und daher ähnlich viel Ressourcen verbraucht werden. Die Abbildung 5.1 zeigt der Programmablaufplan des Quantisierung Moduls.

5.2 Erläuterung des CUDA Quellcodes

Hier werden nur die Bereiche behandelt, die sich zum reinen C++ Code unterscheiden. Der komplette CUDA Code befindet sich im Anhang auf der CD.

5.2.1 HOST-Code

Hier wird die Größe des benötigten Speichers berechnet. Dazu wird die Bildhöhe und Bildbreite genutzt.

```
1 int n =picturehigh*picturewidth;
2 size_t size = n*3 * sizeof(int);
3 size_t size_i = 64 * sizeof(int);}
```

Initialisierung der einzelen Matrizen, wobei das 'h' anzeigt, dass es sich um einen Speicherplatz handelt, der sich im Host befindet und 'd' steht für Device.

```
1 int* h_outputmatrix= (int*)malloc(size);
2 int* d_A;
3 int* d_quantisationmatrixluminazen;
4 int* d_quantisationmatrixchrominazen;
5 int* d_outputmatrix;
```

Anlegen der Speicherbereiche auf der Karte im globalen Speicherplatz

```
1 cudaMalloc(&d_quantisationmatrixluminazen, size_i);
2 cudaMalloc(&d_quantisationmatrixchrominazen, size_i);
3 cudaMalloc(&d_outputmatrix, size);
```

Kopieren der Daten in den globalen Speicher

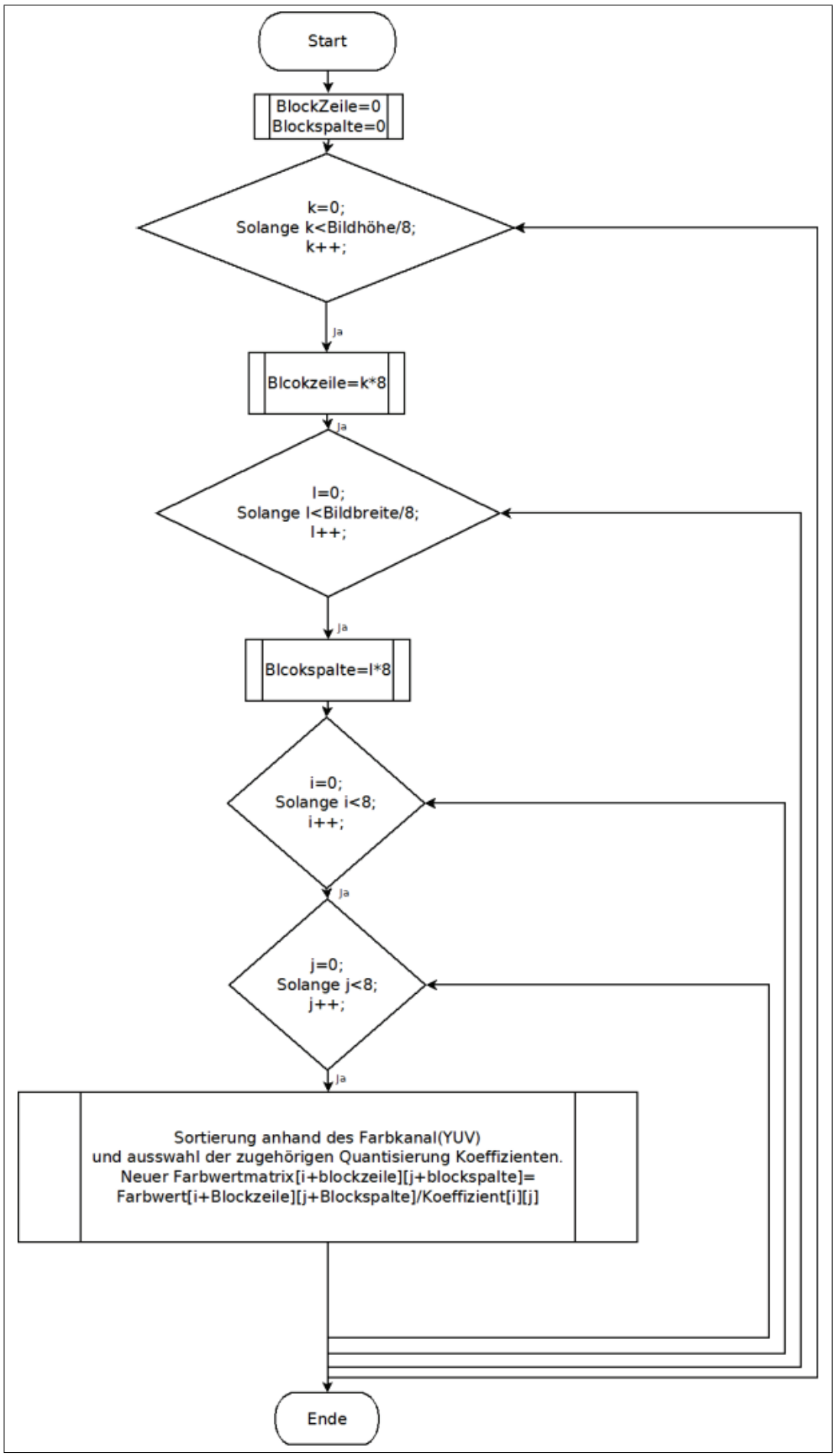

Abbildung 5.1: Programmablaufplan der Quantisierung

```
cudaMemcpy(d_A,dctYUV, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_quantisationmatrixluminazen,
quantisationmatrixluminazen, size_i, cudaMemcpyHostToDevice);
cudaMemcpy(d_quantisationmatrixchrominazen,
quantisationmatrixchrominazen, size_i, cudaMemcpyHostToDevice);
```

Anlegen der Block- und Gridgröße für die Quantisierung, wie in Punkt 4.2.4 beschrieben

```
dim3 threadsPerBlock(8,8,3);
dim3 numBlocks(picturehigh / threadsPerBlock.x, picturewidth
/threadsPerBlock.y);
```

Kernel Aufruf

```
quantisation<<<numBlocks,threadsPerBlock>>>
d_A,d_quantisationmatrixluminazen,d_quantisationmatrixchrominazen,
d_outputmatrix,picturehigh,picturewidth);
```

Kopieren des Speichers zum Host

```
cudaMemcpy(h_outputmatrix, d_outputmatrix, size,
cudaMemcpyDeviceToHost); |
```

Freigabe der Speicherbereiche auf der Grafikkarte

```
cudaFree(d_A);
cudaFree(d_quantisationmatrixluminazen);
cudaFree(d_quantisationmatrixchrominazen);
cudaFree(d_outputmatrix);
```

5.2.2 Kernel Code (Quantisierung)

Die Eingangsmatrix ist 3 Dimensionen aufgebaut, wobei die Chrominazwerte hinten auf der Z-Achse liegen.

```
__global__ void quantisation(int* Eingangsmatrix,
int* quantisationmatrixluminazen,
int* quantisationmatrixchrominazen,
int* quantisiertematrix,int high,int width)
{
```

Erstellen der Indizes für die einzelnen Threads

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int k = blockIdx.z * blockDim.z + threadIdx.z;
```

Abfrage, welche der beiden quantisierungsmatrizen zur Berechnung benötigt werden, mit anschließender Berechnung und Speicherung des neuen Wertes.

```
1 if(k==0)quantisiertematrix[high*width*k+i * width + j]=
2 Eingangsmatrix[high*width*k+i * width + j]
3 /quantisationmatrixluminazen[threadIdx.x*blockDim.x +threadIdx.y];
4 else quantisiertematrix[high*width*k+i *
5 width + j]= Eingangsmatrix[high*width*k+i * width + j]/
6 quantisationmatrixchrominazen[threadIdx.x*blockDim.x +threadIdx.y];
7                 }
```

5.3 Analyse

Die Programme wurden unter Hilfenahme des Analysierungs Tools "Compute Visual Profiler Version 3.1" von NVIDIA analysiert und es wurden Zeitmessungen mit der Bibliotheken time.h und cutil.h. durchgeführt. Die Zeitmessungen wurden mit vier verschieden Bildgrößen durchgeführt (256px , 512px, 1024px, 2048px). Die beiden Algorithmen wurden auf folgenden System ausgeführt.

5.3.1 Testsystem:

Prozessor: Intel Core2 Quad Prozessor Q6600 2,40 Ghz
Arbeitsspeicher: 4 GB (3,5 verwendbar)
Systemtyp: 32Bit-Betriebsystem
Grafikarte: GeForce 8800 GTS 512

CUDA Driver:	Version: 3.10
CUDA Runtime:	Version: 3.10
CUDA Capability:	Major Revision Nummer: 1
CUDA Capability:	Minor Revision Nummer: 1
Global memory:	521732096 bytes
Anzahl von Mikroprozessors:	16
Anzahl von Prozessor (SIMT):	128
Constant memory:	65536 bytes
Shared memory per block:	16384 bytes
Register pro Block:	8192
Maximale Anzahl von Threads pro Block:	512
Maximale Demension eines Blocks :	512 x 512 x 64
Maximale Demension eines Grid:	65535 x 65535 x 1
Clock rate:	1.62 GHz

Tabelle 5.1: Grafikkarten Attribute

5.3.2 Vergleich der Implementierungen

Die Ergebnisse sind in den unteren Diagrammen (Abbildung 5.2) zusammengefasst. Die dazugehörigen Tabellen befinden sich Anhang A. Es konnten leider keine Werte mit dem

“Compute Visual Profiler“ für die Bildgröße 2048 Pixel gemessen werden. In den beiden linken Diagrammen werden die beiden Quantisierungs-Implementierungen miteinander verglichen. Hier fällt auf das der SpeedUp der GPU Implementierung gegenüber der CPU Implementierung von 4,7 auf 3,5 zurück geht. Als Grund ist wahrscheinlich die gestiegende Datenmenge zu nennen, dadurch müssen intern einige Register in den lokalen Speicher ausgelagert werden, was mehr Zeit kostet. Auf der rechten Seite befindet sich die Gesamtlaufzeit, der beiden Implementierungen. Hier wird deutlich das die Parallelisierung des Quantisierungs Moduls nur einen leichten SpeedUp von maximale 1,07 zur Folge hat. Die vom “Compute Visual Profiler“ ausgewerteten Daten in der Abbildung 5.4 zeigen das sich der GPU rund 1/3 der Zeit mit dem Ein- und Auslese der Daten beschäftigt, unabhängig von der Bildgröße (siehe Abbildung 5.3). Die Schwache Bandbreite vom Host zum Device ist auch die größte Schwachstelle von CUDA.

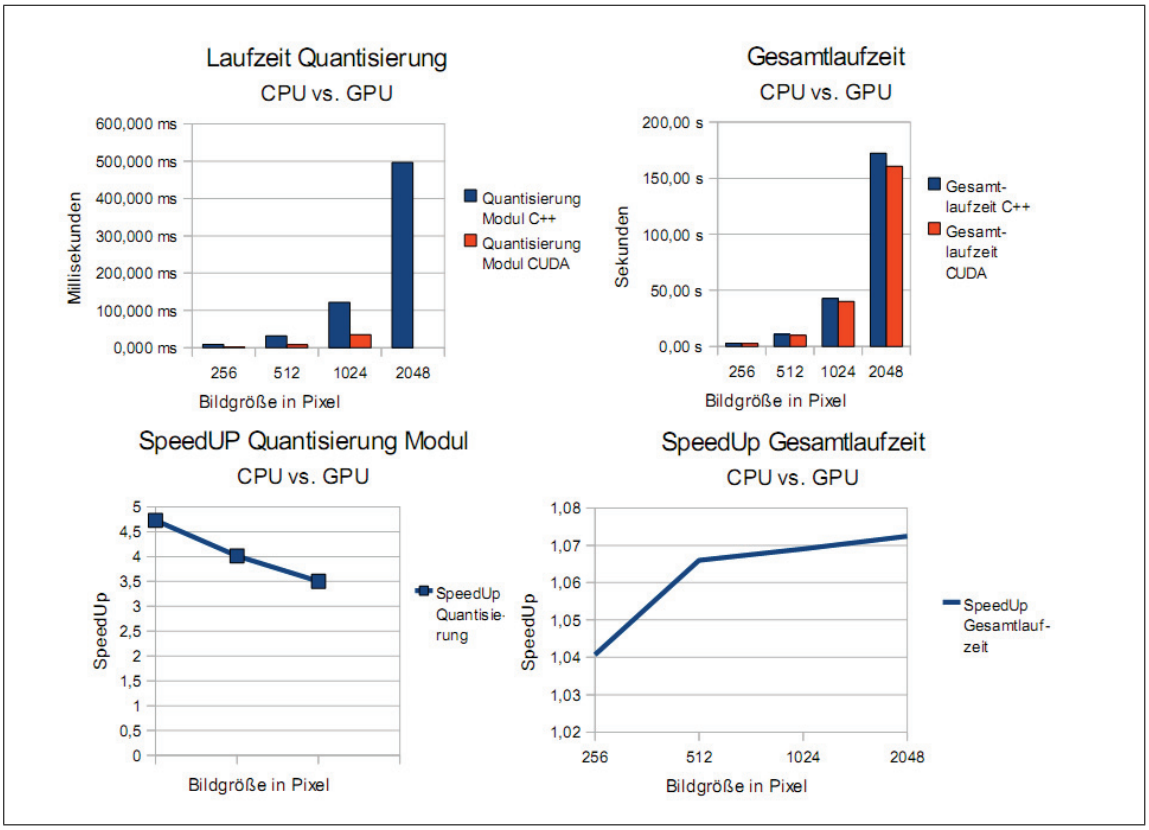

Abbildung 5.2: Gegenüberstellung des Laufzeit und des Geschwindigkeitsgewinn des Quantisierungs Moduls mit und ohne CUDA.

5.4 Fazit

Durch weitere Parallelisierung der anderen Module, die ähnlich gute Parallelisierung Eigenschaften besitzen wie der Quantisierungs Algorithmus, würde sich der SpeedUp der gesamten Implementierung stark erhöhen. Des Weiteren kommt zugute das die Daten ebenfalls nur einmal, in den Device Speicher ein und ausgelesen werden müssten, wodurch die geringe

		Aufrufe	GPU	% GPU
256	quantisation	1	1163,23 µs	61,12 %
	memcpyHtoD	3	306,11 µs	16,08 %
	memcpyDtoH	1	433,6 µs	22,78 %
	Summe		1,90 ms	
512	quantisation	1	4727,01 µs	59,28 %
	memcpyHtoD	3	1379,08 µs	17,29 %
	memcpyDtoH	1	1867,23 µs	23,41 %
	Summe 512		7,97 ms	
1024	quantisation	1	20754,6 µs	59,56 %
	memcpyHtoD	3	6155,72 µs	17,66 %
	memcpyDtoH	1	7931,33 µs	22,76 %
	Summe 1024		34,84 ms	

Abbildung 5.3: Compute Visual Profiler Version Messdaten

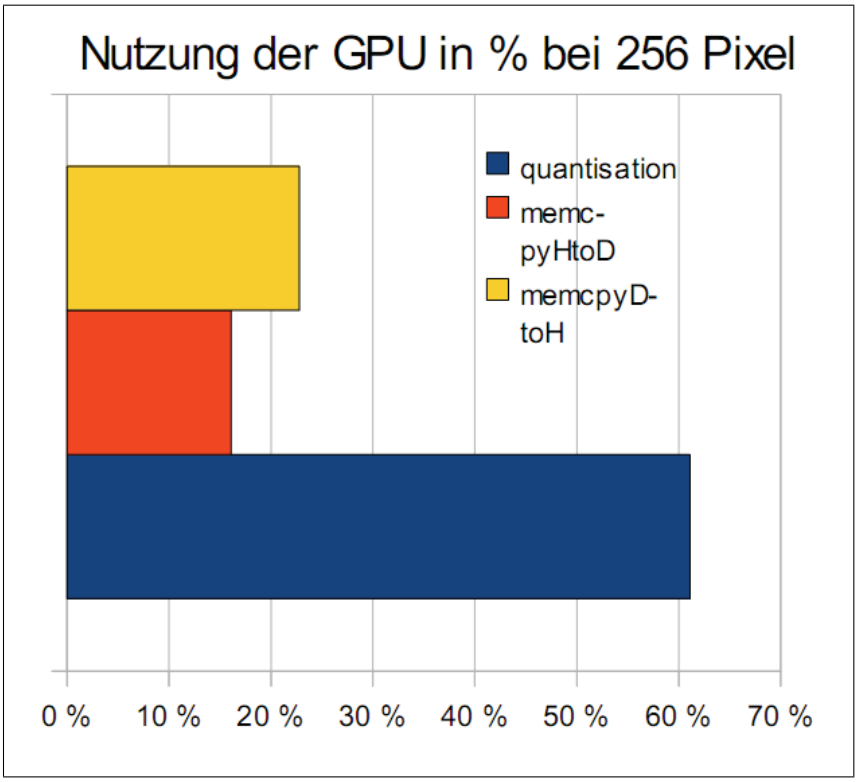

Abbildung 5.4: Visualisierung der Messdaten vom Compute Visual Profiler

Bandbreite zwischen der CPU und GPU nicht so stark zum tragen kommt. Dadurch würde der JPEG Algorithmus wahrscheinlich um das drei bis vier fache beschleunigt werden.

6 Anhang

6.1 Anhang A

C++

Bildgröße	Quantisierung Modul C++	Gesamtlaufzeit C++
256	9,00 ms	2,67 s
512	32,00 ms	10,67 s
1024	122,00 ms	42,70 s
2048	496,00 ms	172,36 s

CUDA

Bildgröße	Quantisierung Modul CUDA	Gesamtlaufzeit CUDA	CUDA Laufzeit in ms
256	1,903 ms	2,57 s	63,39 ms
512	7,973 ms	10,01 s	73,63 ms
1024	34,842 ms	39,94 s	121,82 ms
2048	Keine Daten	160,71 s	330,91 ms

Speedup

Bildgröße	SpeedUp Quantisierung	SpeedUp Gesamtlaufzeit
256	4,73	1,04
512	4,01	1,07
1024	3,5	1,07
2048		1,07

Abbildung 6.1: Tabellen Zeitmessung

Abbildungsverzeichnis

Literaturverzeichnis

[Ber09] Bertuch, Manfred: *c't Magazin für Computer Technik 11*. Heise Zeitschriften Verlag, 11.5.2009. Artikel:Parallel-Werkzeuge 'Datenverarbeitung auf Grafikkarten mit ATI Stream und Nvidia CUDA.

[Bre06] Breitbart, Jens: *Stream Processing und High-Level GPGPU Sprachen*. Universität Kassel, SS 2006.

[CUD08] CUDA, NVIDIA: *NVIDIA CUDA C Programming Guide 2*. NVIDIA CUDA, 7.6.2008. Version 2.0.

[CUD10a] CUDA, NVIDIA: *NVIDIA CUDA C Programming Guide 3*. NVIDIA CUDA, 21.07.2010. Version 3.1.1.

[CUD10b] CUDA, NVIDIA: *NVIDIA CUDA CUBLAS Library*. NVIDIA CUDA, May 2010.

[CUD10c] CUDA, NVIDIA: *NVIDIA CUDA CUFFT Library*. NVIDIA CUDA, May 2010.

[ek09] kompendium elektronik: *Grafikkarten*. Website, 2009. Available online at `http://www.elektronik-kompendium.de/sites/com/0506191.htm`; visited on August 5th 2010.

[GR08] Greg Ruetsch, Brent Oster: *Getting Started with CUDA/Vvision 08*. Präsentation, 2008. NVIDIA Developer Technology.

[Hoe08] Hoermann, Friedrich: *Grundlagen und Aufbau einer Grafikkarte*. Universitaet Ulm, SS 2008.

[Jor08] Joram, Niko: *Untersuchungund Vorstellung moderner Grafikchiparchitekturen*. Technische Universitaet Dresden, 4.Juni 2008.

[Men99] Menn, Ralph: *JPEG*. Website, 30.12.1999. Available online at `http://www.tecchannel.de/webtechnik/entwicklung/401190/jpeg/`; visited on August 31 2010.

[M.F09] M.Flynn: *Some Computer Organizations and Their Effectiveness*. IEEE Transactions on Computers, 10.11.2009. S. 948 – 960.

[MH09] Mark Harris, Dominik Göddeke.: *GPGPU*. Website, 2009. Available online at `http://gpgpu.org`; visited on August 5th 2010.

[nvi09] nvidia: *GPU-Computing*. Website, 2009. Available online at `http://www.nvidia.de/page/gpu_computing.html`; visited on August 5th 2010.

[PI03] Prof.Dr.-Ing.UweSchwiegelshohn: *Parallele Rechnersysteme*. Universitaet Dortmund, 29. September 2003.

[Roh08] Rohr, Stephan: *Cuda Programme mit C*. Präsentation, 2008. University of Heidelberg.

[Sch09] Schnattinger, Thomas: *JPEG-Seminararbeit am Institut für theoretische Informatik der Universität Ulm*. Universität Ulm, 6. Januar 2009.

[Shi09] Shi, Yuan: *Reevaluating Amdahls Law and Gustafsons Law*. Website, 2009. Available online at `http://www.cis.temple.edu/~shi/docs/amdahl/amdahl.html`; visited on juli 12th 2010.

[Sil02] Silberhorn, Norbert: *Das jpg-Bildformat*. Website, 2002. Available online at `http://www.itec.uka.de/seminare/rftk/jpeg/`; visited on September 1th 2010.

[sta09] stackoverflow: *CUDA Driver API vs. CUDA runtime*. Website, 2009. Available online at `http://stackoverflow.com/questions/242894/cuda-driver-api-vs-cuda-runtime`; visited on August 5th 2010.

[Tem05] Temerinac, Maja: *J.P.E.G = Joint Photographic Expert Group*. Präsentation, 2005. Albert-Ludwigs-Universität Freiburg.

[uMG02] Marc Glaus, Roman Steiner und: *Theorie und Implementation zu JPEG und JPEG2000*. Universität Ulm, 11.11.2002.

[Unb] Unbekannt: *CUDA Best practices*. Präsentation.

[wik10] wikipedia.org: *Z-Buffer*. Website, 29. Juni 2010. Available online at `http://de.wikipedia.org/wiki/Z-Buffer`; visited on August 31 2010.

[Zel08] Zeller, Cyril: *Tutorial CUDA*. Präsentation, 2008. NVIDIA Developer Technology.