Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3178

Abschätzung des Ressourcenverbrauchs und Analyse der Echtzeitfähigkeit von CUDA- und OpenCL-Befehlen

Fabian Römhild

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dipl.-Inf. Stephan Schnitzer
Externer Betreuer:	Dipl.-Inf. Simon Gansel (Daimler AG)
begonnen am:	16. Mai 2011
beendet am:	15. November 2011
CR-Klassifikation:	C.1.2, D.1.3

Kurzfassung

CUDA und OpenCl ermöglichen die Grafikkarte für Berechnungen abseits der Grafikausgabe zu verwenden. Gerade bei parallelisierbaren Berechnungen kann so ein enormer Geschwindigkeitszuwachs erreicht werden. CUDA und OpenCL geben keine Zeitgarantien, d.h. für die Laufzeit von Programmen gibt es keine Beschränkung. Des Weiteren sind einmal gestartete Berechnungen nicht unterbrechbar. Für Echtzeitgarantien ist dies allerdings zwingend erforderlich. In dieser Diplomarbeit wird untersucht, ob die GPU auch für Berechnungen in Echtzeitsystemen verwendet werden kann. Es wird nach Möglichkeiten gesucht den Kontextwechsel zwischen verschiedenen CUDA- und OpenCL-Programmen zu steuern. Ferner wird die Laufzeit und der Speicherverbrauch abgeschätzt und die für Echtzeit wesentlichen Einflussfaktoren ermittelt. Durch Evaluation wird das spezifische Verhalten analysiert und mit Hinblick auf Isolation und Echtzeitgarantien bewertet. Diese Arbeit zeigt auf, dass es in gewissem Umfang möglich ist, bezüglich Laufzeit und Ressourcenverbrauch, Garantien zu gewährleisten.

Abstract

CUDA and OpenCL enable the use of the video card for computing besides the output to the display. There is an enourmous speedup achievable especially in parallel computing. CUDA and OpenCL don't guarantee response time, so there is no limitation for the calculations. Furthermore there is no preemption for calculations once started. This would be necessary for real-time guarantees. This Diploma Thesis researches the opportunity to use the GPU for calculations in real-time systems. It is looking for possibilities to control the context switch between different CUDA and OpenCL programs. In addition, the running time and memory consumption and the estimated essential factors for real-time are determined. Through evaluation, we analyze the specific behavior and rate them with regard to isolation and real-time guarantees. This Thesis shows that it is to some extent possible to ensure guarantees with respect to time and resource consumption.

Danksagung

Bei der Erstellung dieser Diplomarbeit gab es zahlreiche Hilfestellungen von verschiedenen Personen. Einen besonderen Dank möchte ich allen aussprechen. Namentlich sollen hier einige genannt werden:

- Prof. Kurt Rothermel für die Ermöglichung der Diplomarbeit und der Übergabe eines interessanten Themas.
- Simon Gansel und Stephan Schnitzer für die sehr gute Betreuung, die vielen hilfreichen Tipps und die anregenden Diskussionen.
- Der Daimler AG und den freundlichen Mitarbeitern, besonders der Abteilung GR/PTA, für die Unterstützung in jeglicher Hinsicht.
- Meiner Frau Maria-Paola für ihre Geduld die letzten 6 Monate über und die seelische Unterstützung, die sie mir jederzeit entgegenbrachte.
- Ganz besonderer Dank gilt auch meinen Eltern für ihre Liebe und Zustimmung in allen Lebenslagen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

1.1 Motivation

Mit Intels Pentium 4 und seiner Netburst-Architektur, die zum Ziel hatte, möglichst hohe Taktraten zu erreichen, wurde schnell klar, dass die Erhöhung des CPU-Taktes immer ineffizienter wird. Mit steigendem Takt kam ein überproportionales Wachstum an Stromverbrauch und Abwärme hinzu. Zudem verbesserte sich die Leistung nicht linear mit der Takterhöhung. Mittlerweile geht der Trend auf dem CPU-Markt stark in Richtung Mehrkernprozessoren. Für Desktop-PCs sind CPUs mit acht Kernen bereits verfügbar, im Servermarkt auch mit zwölf. Bis diese aber flächendeckend auch in Desktop PCs vertreten sein werden, wird noch einige Zeit vergehen. Trotzdem lohnt es sich jetzt schon Programme massiv auf Multithreading auszulegen, denn in den letzten vier Jahren hat sich eine andere Komponente stark in diese Richtung weiterentwickelt: die Grafikkarte. Aktuelle Grafikkarten besitzen schon jetzt mehrere dutzend Kerne, die völlig unabhängig voneinander Berechnungen durchführen können. Dadurch sind aktuelle GPUs theoretisch in der Lage, über 1500 GFLOPS (Gleitkommaoperationen pro Sekunde) mit einfacher Präzision durchzuführen. CPUs im gleichen Preissegment erreichen etwa 200 GFLOPS, siehe Abbildung 1.1.

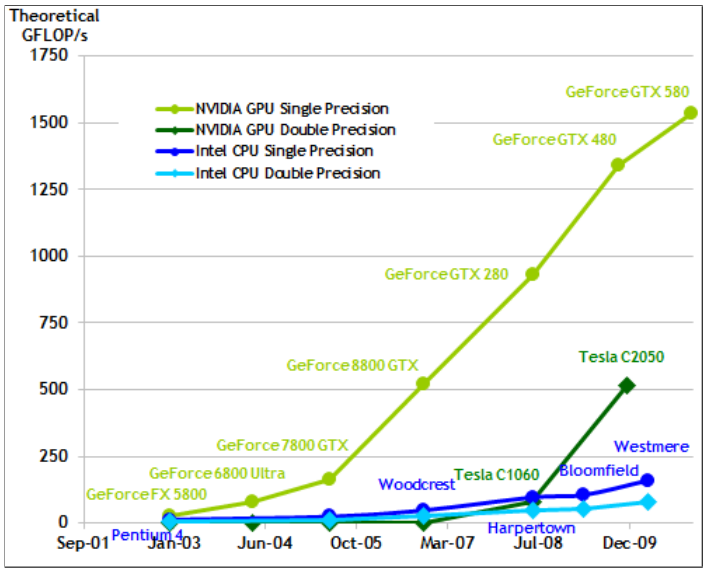

Abbildung 1.1: Vergleich der Rechenleistung von CPUs mit GPUs (Quelle: [NVI11c])

Grafikkarten werden zwar immer noch hauptsächlich für das Rendering (Berechnung der Bildausgabe) benutzt, aber die beiden führenden Grafikchiphersteller AMD und Nvidia investieren viel Geld, um Werkzeuge zur Allzwecknutzung von Grafikkarten bereitzustellen und ihre Chips auf diesem Gebiet weiter zu verbessern. NVIDIAs CUDA und das Herstellerübergreifende OpenCL repräsentieren jeweils einen Weg, Grafikkaten für allgemeinere Zwecke zu verwenden, beide ähneln der Programmiersprache C um den Einstieg zu erleichtern. Die CUDA Api funktioniert nur mit eigenen Chips, ist also auf AMD Grafikkarten nicht lauffähig. Um flexibler bei der Auswahl der Hardware zu sein, kann OpenCL verwendet werden. Dieser noch relativ neue, offene Standard ist in der Lage, viele Komponenten des PCs anzusprechen, um Programme auszuführen. Dabei können nicht nur CPU und Grafikkarte einbezogen werden, sondern auch andere Chips und Geräte die mit dem PC verbunden sind, solange sie den OpenCL Standard unterstützen und entsprechende Treiber zur Verfügung stehen. Momentan lassen sich CPUs und die gängigsten Grafikkarten zuverlässig mit OpenCL nutzen. Beide Schnittstellen bieten Möglichkeiten die deutliche höhere Rechenkapazität von GPUs gegenüber CPUs in eigenen Anwendungen zu nutzen. Dies trifft im Allgemeinen nur bei parallelen Programmabläufen zu. Als weiteres Einsatzfeld bieten CUDA und OpenCL die Fähigkeit OpenGL Befehle zu ersetzen und so die hardwarebeschleunigte Darstellung von Bildern zu unterstützen.

1.2 Problemstellung

Ziel dieser Diplomarbeit ist die Analyse der Funktionsweise von CUDA- und OpenCL-Befehlen. Da die Grafikkarten Hersteller über die genauen Funktionen und Arbeitsweisen ihrer Treiberimplementierungen keine Aussage treffen, soll anhand prototypischer Implementierungen ein Einblick in dieses Gebiet gegeben werden. Um einen Überblick zu gewinnen, sollen die beiden Schnittstellen auf ihre Unterschiede bei der Programmierung und der Performance untersucht werden. Die Anforderungen zur Echtzeitfähigkeit beinhalten die Evaluierung der Reihenfolge bei der Abarbeitung und Verzögerungszeit von Anfragen an die Grafikkarte, sowie der Erarbeitung von Möglichkeiten diese zu steuern. Ein weiterer wichtiger Aspekt ist die Speicherverwaltung, sie hat direkten Einfluss auf die Performance einer Anwendung und den Ressourcenverbrauch. Werden CUDA- oder OpenCL-Programme in einem System eingesetzt, so ist der Zugriff auf die Grafikkarte zumeist nicht exklusiv. Das macht es notwendig das Konkurrenzverhalten zwischen verschiedenen Programmen zu evaluieren, so wie es in einem realen Anwendungsszenario der Fall ist.

1.3 Verwandte Arbeiten

Bisherige Arbeiten die sich mit dem Thema CUDA und OpenCL beschäftigt haben, gehen meist konkret auf die Implementierung eines Algorithmus oder die Beschleunigung eines Programms ein. Christian Löwen [Löw09] untersuchte in seiner Diplomarbeit die „Parallele Berechnung Kombinatorische Vektorfelder mit CUDA“ und konnte diese mithilfe von CUDA um den Faktor 30 beschleunigen.

In der Master Thesis von Ashok Dwarakinath [DWA08] mit dem Titel „A Fair-Share Scheduler for the Graphics Processing Unit“ wird versucht das Scheduling der Befehle an die Grafikkarte zu verbessern. Dies gelingt allerdings nur für eine spezielle Grafikkarte und setzt direkt an dem bereitgestellten Open Source Treiber an. Dieser wird soweit verändert, dass die Befehle an die GPU abgefangen und auf ihre Komplexität hin untersucht werden, bevor sie zur Ausführung weitergeleit werden. Durch den Ansatz auf Treiberebene ist es weiter möglich die Befehle durch einen eigenen Scheduling Algorithmus, hier „deficit Round Robin“, zu bearbeiten und das sonst verwendete „First-come first-served“ Prinzip des Treibers zu umgehen. Um dies zu realisieren, werden die Befehle verschiedener Prozesse umgruppiert und ein eigener Kontextwechsel eingesetzt.

In der Veröffentlichung [GGHS09] „Enabling Task Parallelism in the CUDA Scheduler“ ist ein ähnlicher Ansatz gewählt um Einfluss auf das Scheduling zu nehmen. Die Kernel werden dabei, bevor sie zur GPU gelangen, ebenfalls untersucht und nach Möglichkeit in einer eigenen Befehlsqueue zusammengefasst, um so parallele Kernel ausführen zu können.

1.4 Aufbau der Diplomarbeit

In Kapitel 2 werden zunächst die wichtigsten Grundlagen des GPGPU anhand von CUDA und OpenCL vorgestellt. Mit Hilfe eines Vergleichs der Programmiersprache C werden die Details der Programmierung vorgestellt. In Kapitel 3 wird in die Echtzeitfähigkeit eingeführt und es werden die von den jeweiligen Herstellern bereitgestellten Analysewerkzeuge besprochen. Die Vorgehensweise und Messmethoden zur Evaluation sind ebenso in diesem Kapitel untergebracht.

In Kapitel 4 werden zunächst die Testsysteme, die in dieser Arbeit verwendet werden, vorgestellt und anschließend die bisher besprochenen Messmethoden in der Evaluation angewandt. Die Evaluation verzweigt sich in weitere vier Abschnitte. Zunächst wird die Performance von CUDA und OpenCL verglichen. In Abschnitt 4.3 wird die genaue Verwaltung des Grafikartenspeichers untersucht. Der dritte Abschnitt erläutert in welcher Reihenfolge die Befehle auf der Grafikkarte abgearbeitet werden (Scheduling) und welche Möglichkeiten zur Einflussnahme auf das Scheduling gegeben sind. Im letzten Abschnitt der Evaluierung, 4.5, wird auf die Verwaltung der Kontexte und die Interoperabilität zwischen CUDA, OpenCL und OpenGL eingegangen. Abgeschlossen wird diese Arbeit mit einer Zusammenfassung und einem Ausblick.

2 Einführung in GPGPU

In diesem Kapitel wird dem Leser ein Überblick über das Umfeld der in dieser Arbeit behandelten Themen geboten. Es werden die allgemeinen Grundlagen des GPGPU (General Purpose Computation on Graphics Processing Unit) erklärt und auf dessen Besonderheiten eingegangen. Darauf aufbauend, werden die beiden Schnittstellen CUDA (Compute Unified Device Architecture) und OpenCL (Open Computing Language) näher vorgestellt und deren Eigenschaften erläutert. Die Informationen dazu stammen, wenn nicht anders angegeben, aus [NVI11c, SK10] für CUDA und [KG11, MGM+11] für OpenCL.

2.1 Vergleich von CPU- und GPU-Architektur

Grafikkarten sind besser auf parallele als auf sequintelle Anwendungen ausgelegt. Dies ergibt sich aus der geschichtlichen Entwicklung von Grafikkarten, beziehungsweise den Anforderungen an das 3D Rendering, die computergestützte Erzeugung von Bildern.

Dort müssen für viele Pixel und Vertices (Eckpunkt eines Polygons) sehr oft dieselben Operationen durchgeführt werden. Dieser Aufgabenbereich erfordert wesentlich weniger Sprungvorhersagen und das Puffern von Daten, als die meisten anderen Berechnungen, die CPUs bearbeiten. Denn dabei ist die Abfolge eines Programms meist von Faktoren abhängig, die erst während der Laufzeit feststehen. Wohingegen bei einem Bild jedes einzelne Pixel vor der Ausgabe berechnet werden muss.

Die dafür notwendigen Ressourcen können also in andere Komponenten investiert werden, die das Rendering stärker beschleunigen, wie zum Beispiel in mehr Rechenwerke. Abbildung 2.1 vergleicht die Architekturen von heutigen CPUs und GPUs.

GPUs bieten sich bei Berechnungen an, bei welchen für die Verarbeitung einer großen Menge verschiedener Datenelemente immer derselbe Algorithmus angewandt wird und viele arithmetische Operationen durchgeführt werden. Dies entspricht am ehesten ihrer ursprünglichen Aufgabe, dem 3D Rendering. Je mehr Rechenoperationen ausgeführt werden, desto besser können Latenzzeiten durch Speicherzugriffe verborgen werden und entsprechend größer wird der Geschwindigkeitsvorteil gegenüber CPUs. Denn durch die hohe Anzahl an Registern stehen meist genügend Daten für die Berechnung zur Verfügung. Da außerdem immer derselbe Code ausgeführt wird, muss sich die Grafikkarte weniger um die Programmablaufsteuerung kümmern, sondern vor allem um die Zuführung der Daten. Diese Rechenleistung für nichtgrafische Anwendungen zu nutzen, ist Gegenstand des GPGPU. Ein weiterer Vorteil gegenüber der CPU liegt in der höheren Speicherbandbreite. Diese Zusammenhänge sind in Tabelle 2.1 noch einmal zusammengefasst.

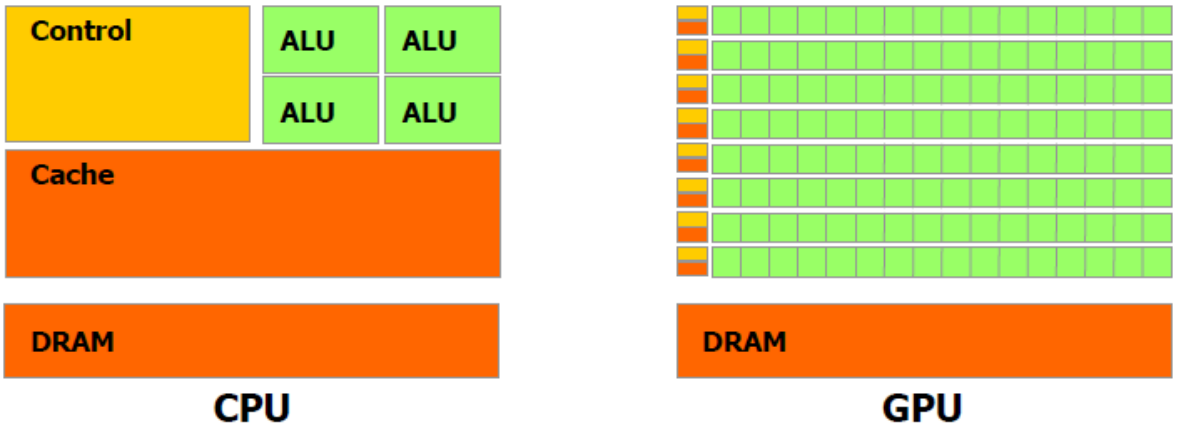

Abbildung 2.1: Schematischer Vergleich der CPU- mit der GPU-Architektur (Quelle: [NVI11c]). Grüne Blöcke repräsentieren Komponenten die Berechnungen ausführen, gelbe die Programmablaufsteuerung und orangene den Speicher.

Architektur	Rechenleistung	Speicherbandbreite
GPU AMD Fire Pro V5900	610 GFLOPs	64 GB/s
GPU NVIDA Quadro 2000D	480 GFLOPs	41,7 GB/s
GPU NVIDIA GeForce GTX 580	1580 GFLOPs	192 GB/s
CPU Intel i7 920	47 GFLOPs	25 GB/s

Tabelle 2.1: Vergleich zwischen CPU und GPU mit Daten der jeweiligen Hersteller, daher als theoretischer Maximalwert anzusehen[NVI11c, AMD11a, Int11]

Die bekanntesten Vertreter des GPGPU sind CUDA und OpenCL, auf die im folgenden Abschnitt genauer eingegangen wird. Eine weitere Alternative ist „DirectCompute“ von Microsoft [NVI11d], damit kann allerdings nur unter Windows Vista und Windows 7 gearbeitet werden und die Verbreitung ist bis heute zu vernachlässigen.

2.2 CUDA

CUDA wurde 2006 von NVIDIA als Plattform für paralleles Rechnen vorgestellt. Dadurch wurde es erstmals ermöglicht, allgemeine Berechnungen auf den spezialisierten GPUs durchzuführen, anstatt wie bisher auf der CPU. Bei früheren Grafikchipgenerationen waren die Rechenressourcen auf Vertex- und Pixel-Shader aufgeteilt, was sie nur schwer für andere Berechnungen, abseits des Rendering, einsetzbar machte. Der erste Chipsatz mit CUDA Unterstützung, der GeForce 8800 GTX, brachte als Neuerung, und gleichzeitig Voraussetzung für CUDA, die „unified shader pipeline“ mit sich. In dieser wurden die bisherigen Vertex- und Pixel-Shader programmierbar vereinigt. Diese Pipeline erlaubt es jede ALU (arithmetisch-logische Einheit) auf dem Grafikchip programmierbar zu steuern und so auch für allgemeine

Berechnungen zu verwenden. Des Weiteren wurde der bisherige Befehlssatz der GPU, der auf Grafikberechnungen ausgelegt war, um Befehle für universelle Berechnungen erweitert, damit war der Grundstein für CUDA gelegt. Ein weiterer wichtiger Faktor ist die Fähigkeit der GPU auf die verschiedenen Speicherbereiche der Grafikkarte schreibend und lesend zugreifen zu können. Um die neu eingeführten Funktionen leichter verwenden zu können und nicht auf andere Firmen angewiesen zu sein, wurde eine auf C basierende Programmiersprache und eine Laufzeitumgebung entwickelt. Davor waren Berechnungen auf der GPU nur mit auf Grafik optimierten Shadersprachen wie OpenGL (Open Graphics Library) und HLSL (High Level Shading Language) möglich. Momentan aktuell ist CUDA in der Version 4.0, die Ende Mai 2011 veröffentlicht wurde.

Die NVIDIA CUDA Architektur besteht im Prinzip aus zwei unterschiedlichen Plattformen:

- Ein Host Sytem (CPU)
- Ein CUDA kompatibles Gerät (GPU)

Die CPU führt bei dieser Zusammenstellung ein Programm aus und kann die parallelen Teile, z.B. eine Vektoraddition, als so genannte CUDA Kernel an die GPU zur Berechnung weitergeben. Die CUDA Kernel werden von dem eigens von NVIDIA entwickelten Compiler „nvcc“ für die Grafikkarte übersetzt. Der Host Teil des Programms wird mit dem jeweils verwendeten C Compiler übersetzt. Davor müssen die Daten an die GPU geschickt werden und ebenso nach der Berechnung das Ergebnis wieder zurück zum Host. Die GPU fungiert in diesem Fall nur als Koprozessor für die CPU.

2.2.1 Das Plattform-Modell

Eine CUDA kompatible GPU besteht aus mehreren so genannten Multiprozessoren, die je nach Modell und Leistungsfähigkeit eine unterschiedliche Anzahl an Multiprozessoren enthält. Diese teilen sich den gesamten Arbeitsspeicher der GPU. Der Arbeitsspeicher wird weiter differenziert in den Globalen Speicher, dieser wird Standardmäßig verwendet und zwei kleine Spezialbereiche, den Konstanten Speicher (48 KB) und den Textur Speicher (6-8 KB). Auf die zwei letztgenannten kann von der GPU aus nur lesend zugegriffen werden, sie werden durch interne Caches in jedem Multiprozessor gepuffert und erreichen dadurch eine geringere Verzögerungszeit. Als weiteren Speicher besitzt jeder Multiprozessor exklusiv einen gemeinsamen Speicher (48 KB). Die Multiprozessoren selbst bestehen aus 16 CUDA Kernen, den verschiedenen Speicherbereichen und einer Instruktionseinheit, die für die Verteilung der Befehle auf die einzelnen Kerne zuständig ist. Zu beachten ist hierbei, dass alle CUDA Kerne eines Multiprozessors ihre verschiedenen Speicherbereiche (gemeinsamer Speicher, Textur Speicher Cache, Cache des Konstanten Speichers) teilen, einzige Ausnahme dabei sind die Register die selbstverständlich für jeden Kern exklusiv vorhanden sind. Abbildung 2.2 verdeutlicht diese Zusammenhänge. Die genauen Eigenschaften einer CUDA fähigen Grafikkarte verbergen sich hinter der „Compute Capability“, dies ist mit einer Versionsnummer vergleichbar. Diese wird mit jeder neuen Chipgeneration erweitert. Für nähere Information sei hier auf [NVI11c] Seite 158 ff. verwiesen.

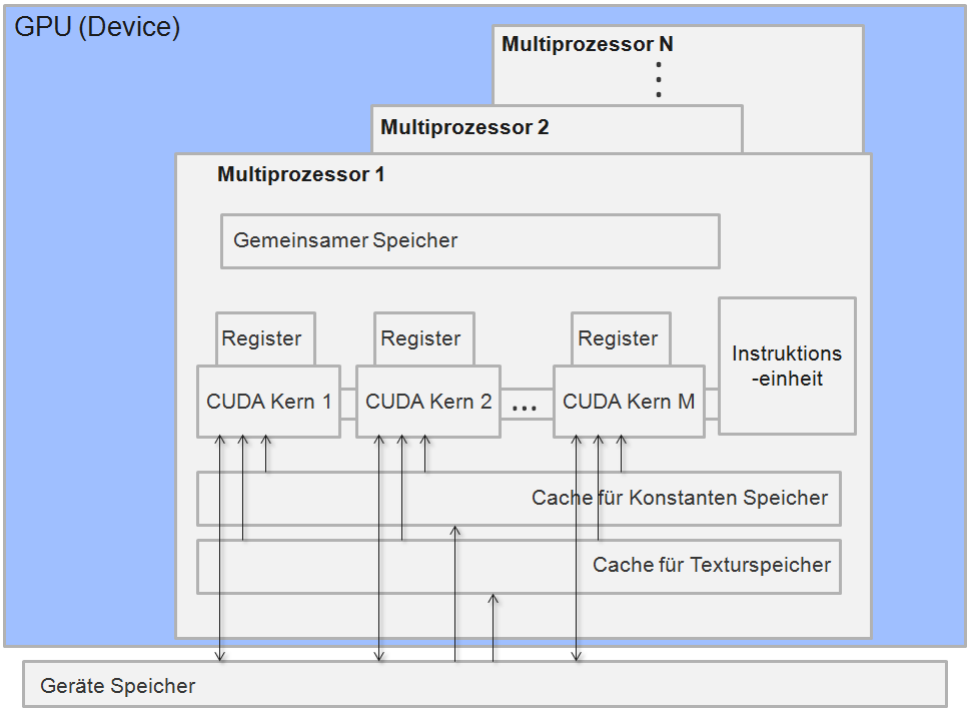

Abbildung 2.2: Das Hardwarelayout von CUDA

2.2.2 Das Ausführungs-Modell

NVIDIA erweitert mit CUDA die Programmiersprache C um die Möglichkeit neue Funktionen zu deklarieren, die parallel auf den CUDA Kernen der GPU ausgeführt werden. Diese Funktionen werden „Kernel“ genannt, die einzelnen Ausführungseinheiten, also der Teil der von einem Kern berechnet wird, werden „Threads“ genannt.

Wie in Abbildung 2.3 zu sehen, werden bei CUDA mehrere Threads zu Blöcken (Block) und diese wiederum zu Grids zusammengefasst. Die Anzahl der Threads je Block muss über alle Blöcke gleich sein und wird direkt beim Aufruf des Kernel festgelegt. Jeder Thread ist durch eine eindeutige „Thread ID“ Global identifizierbar. Diese ID, und damit jeder Thread, ist durch die von der CUDA-Laufzeitumgebung bereitgestellten Variablen „threadIdx und blockIdx“ eindeutig lokalisierbar. Diese Variablen bestehen aus einem Vektor, da der Index für den Block und den Grid bis zu drei Dimensionen fassen kann, je nach gewünschter Berechnung. Gerade bei Vektor- oder Matrizenberechnungen vereinfacht es die Koordination erheblich, da man die Dimensionen so direkt von der entsprechenden Eingabe auf die Anzahl der Threads und Blöcke übertragen kann. Ein Kernelaufruf unterscheidet sich nur geringfügig von einer normalen C Funktion. Beim Aufruf wird direkt die Anzahl der Threads und der Blöcke, auf die die Berechnung verteilt werden soll, angegeben.

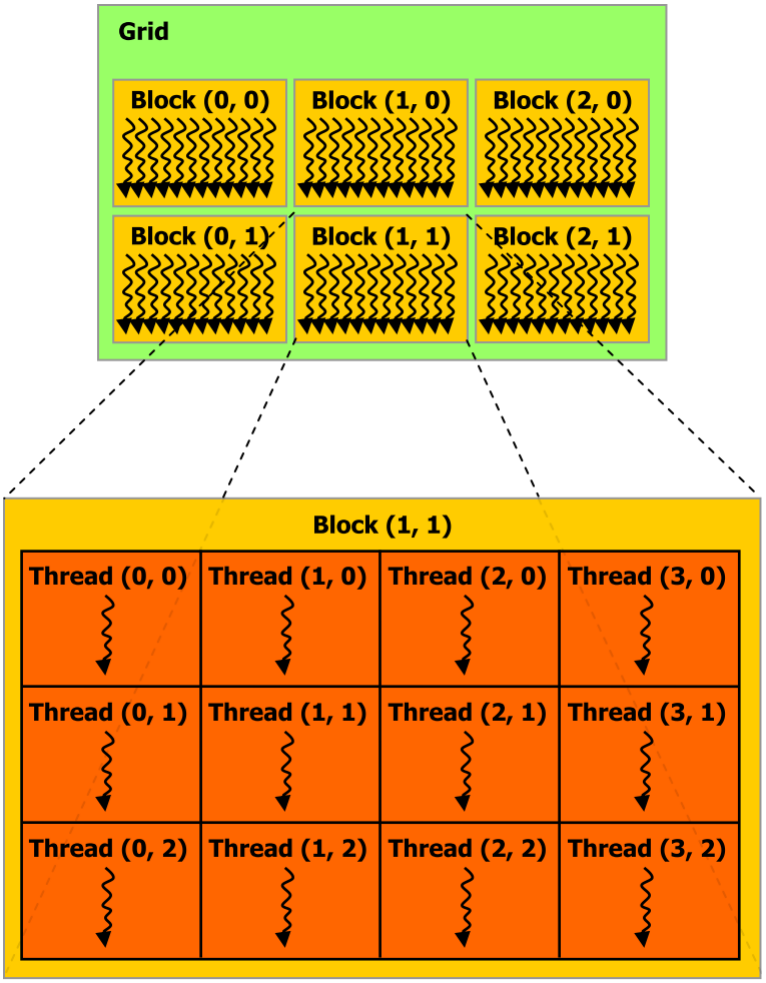

Abbildung 2.3: Threads bei CUDA (Quelle: [NVI11c])

2.3 OpenCL

OpenCL ist ein offener Industriestandard zur Programmierung von heterogenen Prozessor Systemen, bestehend nicht nur aus GPUs sondern ebenso CPUs und anderen Prozessoren. Es setzt sich, ebenso wie CUDA, aus einer auf C aufbauenden Programmiersprache, Bibliotheken und einer Laufzeitumgebung zusammen. Ursprünglich von Apple entwickelt, wurde OpenCL in Zusammenarbeit mit IBM, AMD, Intel und NVIDIA bei der Khronos Group zur Standardisierung eingereicht. Die Khronos Group ist ein Industriekonsortium zur Erstellung und Verwaltung von offenen Standards im Multimedia-Bereich wie z.B. OpenGL. Im Dezember 2008 wurde Version 1.0 veröffentlicht, aktuell ist die im Juni 2010 veröffentlichte Version 1.1 [KG11]. NVIDIA unterstützt mit ihren Treibern allerdings bis dato nur die Version 1.0, von AMD ist die aktuellste Version in ihren Treibern implementiert.

2.3.1 Das Plattform-Modell

Prinzipiell wird bei OpenCL zwischen einem Host, der die Ausführung steuert, und einem oder mehreren mit ihm verbundenen OpenCL Geräten unterschieden. Letztere werden „Compute Devices“ genannt, welche weiter in „Compute Units“ unterteilt sind, die wiederum „Processing Elements“ enthalten. Processing Elements sind letztendlich die Teile der Hardware, die die Berechnungen durchführen, siehe hierzu Abbildung 2.4.

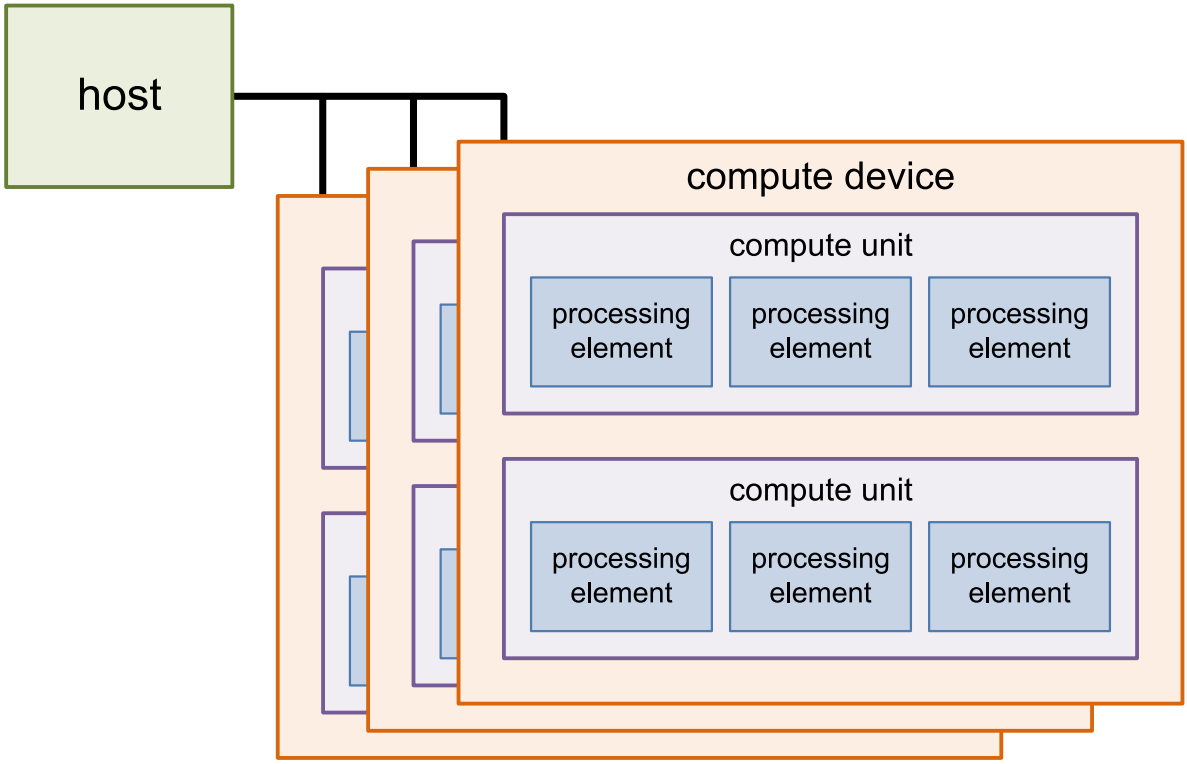

Abbildung 2.4: Das Plattform-Modell von OpenCL (Quelle: [Kön09])

Beim Host handelt es sich, wie bei CUDA, um die CPU. Als „Compute Device“ kommen alle Geräte in Frage, die OpenCL unterstützen (vornehmlich Grafikkarten und CPUs). Eine „Compute-Unit“ entspricht einem Mehrkernprozessor und jeder einzelne Prozessor einem „Processing Element“. Tabelle 2.2 zeigt einen weiteren Vergleich.

Platform Modell	Anwendungsbeispiel	CUDA
Compute-Device	GPU	GPU
Compute-Unit	Mehrkernprozessor innerhalb der GPU	Multiprozessor
Processing-Element	Prozessor innerhalb des Mehrkernprozessors	CUDA Kern

Tabelle 2.2: Vergleich des Standards mit einer Grafikkarte und CUDA

Um die Verbreitung zu vereinfachen, ist OpenCL komplett abwärtskompatibel konzipiert. Der mögliche OpenCL-Standard wird dabei nicht nur von den verwendeten Geräten, son-

dern auch von dem jeweils genutzten Treiber limitiert. Das OpenCL Gerät beeinflusst die Programmierung auf zwei Arten: Zum einen muss die Geräteversion (Hardwarespezifisch, diese muss beim Hersteller erfragt werden) berücksichtigt werden, zum anderen die Sprachversion (abhängig vom installierten Treiber des OpenCL Geräts). Falls ein Gerät mehrere Sprachversionen unterstützt, so wählt der OpenCL Compiler automatisch die neueste, wenn gewünscht, kann die Version auch explizit angegeben werden. Laut Spezifikation gibt die Geräteversion Auskunft über Ressourcen-Grenzen und „erweiterte Funktionalität“ siehe hierzu [KG11] Seite 22. Was das genau bedeutet und welche Version welche Eigenschaften mit sich bringt, wird nicht weiter erläutert. Es ist denkbar, dass sich hinter dieser Versionsnummer ein ähnliches Konzept verbirgt, wie hinter der „Compute-Capability“. Diese gibt bei NVIDIA Grafikkarten beispielsweise an, welche Erweiterungen verfügbar sind.

Die Sprachversion hat ebenfalls Auswirkungen darauf, welche Funktionen und Operationen zur Verfügung stehen. OpenCL 1.1 bringt hier unter anderem neue Datentypen (Drei-Komponenten Vektoren) mit sich. Zudem wurden vorher optionale Erweiterungen fest in den Standard aufgenommen. Zu beachten ist, dass die Sprachversion größer als die Geräteversion sein kann, aber höchstens so groß wie die Treiberversion sein darf. Die Treiberversion beeinflusst die Kommunikation zwischen Host und Device, da sie das Bindeglied zwischen den beiden darstellt. Seit OpenCL 1.1 können zum Beispiel mehrdimensionale Buffer-Objekte vom Host alloziert werden. Vorher waren diese auf eine Dimension beschränkt.

2.3.2 Das Ausführungs-Modell

Die Ausführung ist, wie bei CUDA, in zwei Bereiche unterteilt: Host-Programm und Kernel. Das Host-Programm steuert die Ausführung (beinhaltet unter anderem Speicherreservierung, Compiler-Konfiguration und Kernel-Parameterübergabe) der Kernel, welche letztendlich die gewünschten Berechnungen auf dem OpenCL Gerät durchführen. Wenn ein Kernel aufgerufen wird, wird eine Anzahl an Kernelinstanzen gestartet, die alle denselben Binärcode ausführen. Kernel Instanzen werden „Work Items“ genannt und weiter in „Work Groups“ (Arbeitsgruppen) eingeteilt. „Work Items“ besitzen zwei Indizes, die Globale ID und die Lokale ID. Der erstgenannte Index ermöglicht die eindeutige Zuordnung in der gesamten Menge der „Work-Items“, für die Lokale ID trifft das nur innerhalb einer Arbeitsgruppe zu. Arbeitsgruppen besitzen ebenfalls einen Index, siehe hierzu Abbildung 2.5.

Die Größe einer Arbeitsgruppe gibt die Anzahl an „Work Items“ innerhalb einer Arbeitsgruppe an. Die Größe muss dabei immer ein Vielfaches der „NDRange“, die beim Kernelaufruf angegeben wird, sein. Je nach Problemstellung und Hardware können „Work Items“ auch mehrdimensional angeordnet werden. Es gibt also analog zu CUDA eine mehrdimensionale Organisation der einzelnen Berechnungen.

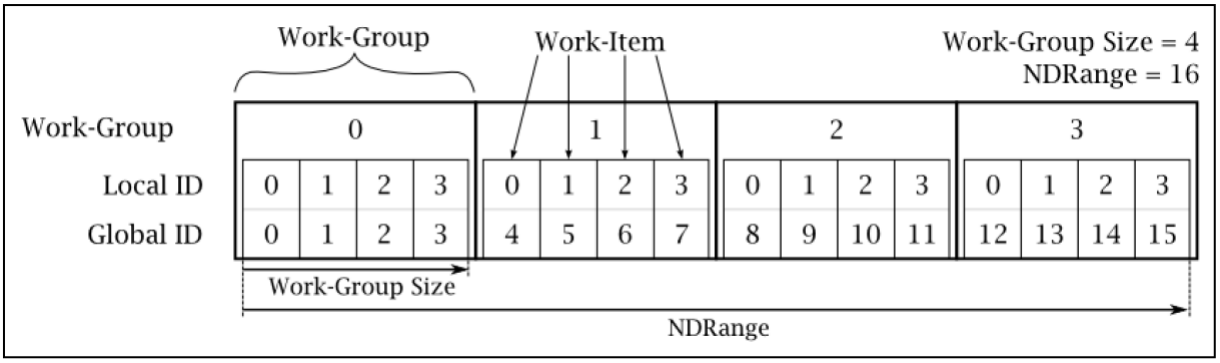

Abbildung 2.5: OpenCL Einteilung (Quelle: [KG11])

2.4 Das Speicher-Modell von CUDA und OpenCL

Sowohl CUDA, als auch OpenCL, teilen den Speicher der Compute Devices in mehrere Bereiche mit verschiedenen Geschwindigkeiten, Sichtbarkeiten und Zugriffsberechtigungen auf. Abbildung 2.6 gibt einen Überblick über die einzelnen Speicherbereiche und ihre Zusammenhänge.

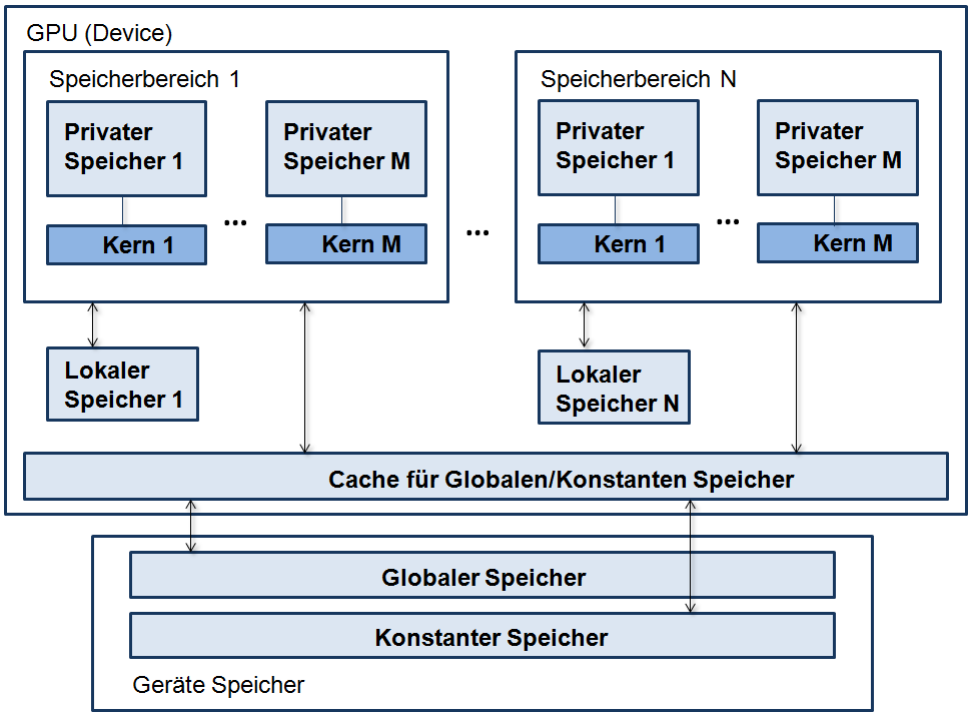

Abbildung 2.6: Speicherbereiche

Geräte Speicher ist ein vom Hauptspeicher des Host getrennter Speicher, der sich auf der Grafikkarte befindet. CUDA- und OpenCL-Kernel können nur aus dem Geräte Speicher aus-

geführt werden. Deshalb bieten die CUDA- und OpenCL-Laufzeitumgebungen Funktionen zum zum Transferieren von Daten zwischen Host Speicher und Geräte Speicher, sowie zum Allozieren, Freigeben und Kopieren innerhalb des Geräte Speichers an.

Globaler Speicher ist der Speicher auf den von jedem Thread eines jeden CUDA-Blocks oder einer OpenCL Arbeitsgruppe aus zugegriffen werden kann. Dieser Bereich erlaubt lesenden und schreibenden Speicherzugriff. Handelt es sich bei dem Device um eine CPU (nur bei OpenCL möglich), so ist der RAM auf dem Mainboard der globale Speicher.

Privater Speicher bei OpenCL ist exklusiv für jeden einzelnen Thread und wird meist nur für temporäre Variablen (wie zum Beispiel ein Schleifenindex) verwendet, die nur für einen Thread sichtbar sind.

Lokaler Speicher bei CUDA entspricht dem Privaten Speicher bei OpenCL. Allerdings nicht zu verwechseln mit dem Lokalen Speicher bei OpenCL.

Konstanter Speicher ist ein Speicherbereich, der während der Ausführung eines Kernels unverändert bleibt. Die Host Applikation alloziert und initialisiert diesen Speicherbereich vor dem Ausführen des Kernels, die einzelnen Threads eines Kernels wiederum können nur lesend darauf zugreifen.

Texturspeicher ist der von CUDA und OpenCL unterstütze Teilbereich der Textur Hardware von GPUs. Daten vom Texturspeicher zu lesen kann Performance Vorteile haben, da dieser immer im Cache vorliegt.

Gemeinsamer Speicher bei CUDA befindet sich auf dem Chip der Multiprozessoren. Zugriffe auf gemeinsamen Speicher sind dadurch um ein Vielfaches schneller, als Zugriffe auf den Globalen Speicher. Dieser Speicherbereich ist für jeden Block privat, die Threads eines Blocks teilen sich jedoch den gleichen Speicherbereich. Der gemeinsame Speicher ist in mehrere Module aufgeteilt, die Bänke genannt werden. Auf jede dieser Bänke kann zeitgleich zugegriffen werden. Dadurch können alle ausführenden Threads auf den Multiprozessoren gleichzeitig und ohne Geschwindigkeitsverlust auf den gemeinsamen Speicher zugreifen. Dies ist allerdings nur solange möglich, wie alle Threads auf verschiedene Bänke zugreifen. Wenn zwei zeitgleiche Zugriffe auf die gleiche Bank erfolgen, werden diese serialisiert.

Lokaler Speicher bei OpenCL entspricht dem gemeinsamen Speicher bei CUDA. Im Gegensatz zu CUDA kann der Lokale Speicher bei OpenCL jedoch, abhängig vom Hardware Design des jeweiligen OpenCL Geräts, entweder ein seperater Speicherbereich sein oder ein Bereich des Globalen Speichers.

Globaler Speicher, Konstanter Speicher und Textur Speicher stehen während der gesamten Ausführung des Host-Programms dauerhaft für jeden Kernelaufruf zur Verfügung.

2.5 Unterschiede zu C

Sowohl bei CUDA als auch bei OpenCL wird auf die bewährte Sprache C aufgebaut und diese wird um eigene Schlüsselwörter erweitert.

CUDA Begriff	OpenCL Begriff	Erklärung
Geräte Speicher	Geräte Speicher	Gesamtheit des Speichers
Globaler Speicher	Globaler Speicher	Arbeitsspeicher
Lokaler Speicher	Privater Speicher	für jeden Thread exklusiv
Konstanter Speicher	Konstanter Speicher	schnell, nur vom Host zu beschreiben
Texturspeicher	Texturspeicher	schnell, nur vom Host zu beschreiben
Gemeinsamer Speicher	Lokaler Speicher	auf dem Chip, privat pro Block bzw. Arbeitsgruppe

Tabelle 2.3: Aufteilung des Arbeitsspeicher bei CUDA und OpenCL

2.5.1 Unterschiede zwischen CUDA und C

CUDA besteht aus kleinen Erweiterungen der Programmiersprache C. Im Gegensatz zu OpenCL gibt es keine explizite Initialisierung für die CUDA-Laufzeitumgebung. Die Initialisierung erfolgt bei dem ersten Aufruf einer Funktion der CUDA-Laufzeitumgebung, die keine Funktion für die Versions- oder Geräteverwaltung ist. Das kann das Prüfen nach vorhandenen CUDA fähigen Geräten sein oder die Abfrage welche CUDA Version von dem entsprechenden Gerät unterstützt wird. Die eigentliche Initialisierung findet also beim ersten Aufruf statt, dies ist zu beachten wenn Untersuchungen des Laufzeitverhaltens durchgeführt werden, da die Initialisierung eine gewisse Zeit benötigt und dadurch der erste Aufruf verzögert wird. Damit diese Verzögerung keinen Einfluss auf die Tests hat, wird vor jeder Messung ein zusätzlicher Aufruf platziert. Ebenfalls muss dies beachtet werden, wenn Fehlercodes des ersten Aufrufs einer Funktion der CUDA-Laufzeitumgebung interpretiert werden, da diese auch von der Initialisierung der CUDA-Laufzeitumgebung stammen könnten. Sobald die CUDA Laufzeitumgebung initialisiert wird, ist der dadurch erzeugte Kontext und seine Ressourcen (wie zum Beispiel allozierter Gerätespeicher) innerhalb des Host-Prozesses gültig und kann von den Threads des Host-Programms verwendet werden.

Einschränkungen

CUDA unterliegt einigen Einschränkungen, die meist durch das Design der GPUs bestimmt wird. Diese Einschränkungen gelten jedoch nur für Algorithmen die auf einem Device ausgeführt wird. Code der auf dem Host ausgeführt wird, kann den vollen Funktionsumfang von C und C++ verwenden. Einige der wichtigen Einschränkungen sind nachfolgend aufgelistet.

- Funktionen mit dem Schlüsselwort „__global__“ unterstützen keine Rekursion.
- CUDA-Kernel und Funktionen, die auf einem CUDA-Device ausgeführt werden, können keine variable Anzahl an Argumenten besitzen.
- CUDA-Kernel müssen den Rückgabewert void haben.

- Funktionsaufrufe von CUDA-Kernel sind asynchron, das bedeutet, dass die Funktion die Kontrolle an die aufrufende Funktion zurück gibt, bevor die Ausführung beendet wurde.
- Argumente für CUDA-Kernel sind auf 4096 Byte Größe beschränkt.
- Parallele Kernelausführung ist nur innerhalb eines Kontextes möglich.
- Ein gerade ausgeführter Kernel kann nicht unterbrochen werden.

Für weitere Details siehe [NVI11c] Seite 148 ff.

Funktionstypen

NVIDIA CUDA kennt für die CUDA Laufzeitumgebung verschiedene Funktionstypen, die bestimmen welche Funktionen auf einem Device und welche Funktionen auf dem Host ausgeführt werden. Ebenfalls unterscheiden sich die Funktionen darin, ob sie durch ein Device oder den Host aufgerufen werden können. Die Funktionstypen werden mit Hilfe spezieller Schlüsselwörter, die der Funktion vorangestellt werden, bei der Deklaration der Funktion bestimmt.

__device__ Dieser Funktionstyp beschreibt eine Funktion, die durch ein Device ausgeführt wird und nur durch ein Device aufgerufen werden kann.

__global__ Mit diesem Schlüsselwort wird eine Funktion als CUDA-Kernel deklariert. Das bedeutet die Funktion wird auf einem Device ausgeführt und kann nur vom Host aufgerufen werden.

__host__ Dieser Funktionstyp beschreibt eine Funktion, die durch den Host ausgeführt wird und nur durch den Host aufgerufen werden kann. Wird kein Funktionstyp angegeben, dies entspricht einer C Funktion, wird automatisch „__host__“ angenommen.

Der Funktionstyp kann auch in Verbindung mit „__device__“ verwendet werden. Dabei wird die Funktion sowohl für den Host, als auch für das Device übersetzt.

Variablentypen

Durch die Verwendung spezieller Schlüsselwörter kann bestimmt werden, in welchem Speicherbereich eine Variable erstellt und alloziert werden soll. Diese werden der normalen Deklaration, wie in C, vorangestellt. Ebenso ist es möglich die Standard C Variablentypen direkt zu verwenden, dann liegt die Verwaltung bei CUDA. Die verschiedenen Typen sind:

__device__ Dieses Schlüsselwort bestimmt, dass eine Variable im Speicher eines Device alloziert wird. Zusätzlich zu „__device__“ kann ein weiterer Variablentyp angegeben werden, der näher bestimmt in welchem Speicherbereich des Device die Variable erstellt wird. Ohne weitere Angabe befindet sich die Variable im Globalen Speicher und ist für die gesamte Laufzeit der Applikation gültig. Variablen mit dem Schlüsselwort „__device__“ können von

allen CUDA-Threads und vom Host mit Hilfe der CUDA Laufzeitumgebung gelesen und verändert werden.

__constant__ Dieser Variablentyp beschreibt eine Variable, die sich im „Konstanten Speicher“ befindet. Die Variable ist ebenso für die gesamte Laufzeit der Applikation gültig und kann von allen CUDA-Threads und vom Host, mit Hilfe der CUDA Laufzeitumgebung, gelesen werden.

__shared__ Mit diesem Schlüsselwort wird eine Variable beschrieben, die im „gemeinsamen Speicher“ (CUDA) erstellt wird. Variablen mit dem Schlüsselwort „__shared__“ haben die Lebensdauer des Blocks in dem sie sich befinden, und können nur von Threads innerhalb dieses Blocks gelesen oder verändert werden.

Typischer Ablauf eines CUDA Programms

Der typische Ablauf einer CUDA-Applikation, welche die CUDA Runtime API verwendet, setzt sich aus Daten an das Device senden, CUDA Kernel ausführen und die Resultate von dem Device lesen, zusammen. Dieser Ablauf soll anhand des Beispiels der Vektor-Addition erläutert werden.

Speicherobjekte anlegen und auf das Device übertragen Um einen CUDA Kernel ausführen zu können, müssen sich alle notwendigen Daten auf dem ausführenden Gerät befinden. In Algorithmus 2.1 ist dieser Vorgang für die Vektoren dargestellt.

Algorithmus 2.1: Speicher allozieren und Daten kopieren mit CUDA

```
1   int N = 50000; //Anzahl der Elemente
2   size_t size = N * sizeof(float); //Größe festlegen
3   //Speicher im Host allozieren
4   h_A = (float*)malloc(size);
5   h_B = (float*)malloc(size);
6   h_C = (float*)malloc(size);
7   //Arrays füllen
8   RandomInit(h_A, N);
9   RandomInit(h_B, N);
10  //Speicher auf dem Device allozieren
11  cudaMalloc((void**)&d_A, size);
12  cudaMalloc((void**)&d_B, size);
13  cudaMalloc((void**)&d_C, size);
14  //Daten auf die GPU kopieren
15  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
16  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Als Besonderheit gibt es noch die Möglichkeit so genannten „Pinned Memory“ anzulegen. „Pinned“ bedeutet in diesem Zusammenhang, dass der auf dem Host allozierte Speicher nicht ausgelagert wird, dadurch kann die Anfrage zum Kopieren in jedem Fall direkt abgearbeitet werden und man erreicht dadurch eine höhere Geschwindigkeit bei der Übertragung von Daten.

Algorithmus 2.2: CUDA Pinned Memory

```
1 cudaMalloc( (void**) &d_data, size)); //Speicher auf der GPU allozieren
2 cudaMallocHost( (void**)&h_data, size ); //Pinned Memory im Host anlegen
3 cudaMemcpyAsync( h_data, d_data, size, cudaMemcpyHostToDevice, 0); //kopieren
```

CUDA Kernel Der CUDA Kernel wird N-mal parallell gestartet. Jede dieser parallelen Einheiten wird Thread genannt. Durch die in CUDA integrierten Variablen „threadIdx“ und „blockIdx“ wird für jeden Thread bestimmt, welchen Index der Variablen A, B und C er berechnen soll. Der Algorithmus 2.3 kann als eine for-Schleife betrachtet werden, bei der die Berechnungen, in diesem Beispiel die Addition einer Komponente der Vektoren, innerhalb des for-Blocks parallel erfolgen.

Algorithmus 2.3: CUDA Kernel zur Vektoraddition

```
1  __global__ void VectorAdd(const float* A, const float* B, float* C, int N)
2  {
3    //Index für die jeweilige Berechnung
4    int id = blockDim.x * blockIdx.x + threadIdx.x;
5    if (id < N) //Bereichsprüfung
6       C[id] = A[id] + B[id];
7  }
```

Die Bereichsprüfung in Zeile 5 ist nötig, da es je nach Anzahl der zu berechnenden Elemente vorkommen kann, dass mehr Threads als nötig gestartet werden. Dies hängt von der Angabe beim Aufruf des Kernel ab, dies wird im folgenden Abschnitt über die Kernelausführung nochmal genauer erläutert.

Kernel ausführen Um die Anzahl der zu verwendenden Threads und Blöcke bestimmen zu können, wurde C um eine eigene Syntax erweitert. Diese Syntax erweitert einen Funktionsaufruf, um die zwei Parameter „gridDim“ und „blockDim“. Diese werden innerhalb dreier spitzer Klammern vor den normalen Argumenten der Funktion übergeben. Im Algorithmus 2.4 wird der CUDA-Kernel „VectorAdd“ in 256 Blöcken zu je 256 Threads ausgeführt. NVIDIA gibt als Empfehlung für diese Größen ein Vielfaches von 64 an, um eine gute Auslastung der CUDA Kerne zu erreichen. Da die Anzahl der so erzeugten Threads nicht immer mit der Anzahl der zu berechnenden Elemente übereinstimmt, ist die in Algorithmus 2.3 gezeigte Bereichsprüfung notwendig.

Algorithmus 2.4: CUDA Kernelaufruf

```
1    //Aufruf des Kernel im Host Code
2    VectorAdd<<<256, 256, 0, stream0>>>(d_A, d_B, d_C, N);
```

Der Parameter „stream“ beim Kernelaufruf ist vom Typ „cudaStream_t“ und ermöglicht die Parallelisierung einer Berechnungen und einer Datenübertragung in Verbindung mit „cudaMemcpyAsync“ aus Algorithmus 2.2. Die Berechnung muss dann einem Stream zugeordnet werden und die Übertragung einem anderen Stream. Seit der CUDA Version 4.0 können damit auch Berechnungen parallelisiert werden, wenn es die Ressourcen der GPU zulassen.

Dabei muss die Anzahl der Threads je Kernelaufruf kleiner sein, als die Gesamtzahl der CUDA Kerne des entsprechenden Geräts. Der dritte Parameter, in diesem Fall die Null, kann dazu genutzt werden dynamisch gemeinsamen Speicher zu allozieren. Die beiden letztgenannten Parameter sind optional und sind in der Standardeinstellung beide auf Null gesetzt.

Ergebnisse zurück zum Host übertragen Nach Ausführen des Kernels ist es noch notwendig, die Daten welche die Resultate der Berechnungen repräsentieren, wieder auf den Host zu transferieren. Dieser Schritt kann mit dem Algorithmus 2.5 erfolgen.

Algorithmus 2.5: Ergebnisse zurück zum Host übertragen

```
1  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

CUDA Objekte freigeben Mit dem Code Abschnitt 2.6 werden die zuvor allozierten Ressourcen wieder freigegeben.

Algorithmus 2.6: Variablen freigeben

```
1    cudaFree(d_A); //CUDA Objekte freigeben
2    cudaFree(d_B);
3    cudaFree(d_C);
4    free(h_A); //Host Speicher freigeben
5    free(h_B);
6    free(h_C);
```

Wird der CUDA Kontext beendet, so werden auch alle ihm zugeordneten Speicherbereiche freigegeben. So ist sichergestellt, dass durch abgebrochene Kernel nicht Teile des Gerätespeichers belegt bleiben.

Verwendung Auf dem Markt gibt es etliche Produkte die mit CUDA Unterstützung ausgestattet sind. Das bekannteste ist sicherlich die Adobe Creative Suite, dort wird die Technik mit „MERCURY Playback Engine“ betitelt und beschleunigt vornehmlich die Video- und Bildbearbeitung [NVIa]. Im Bereich der Videobearbeitung gibt es noch etliche weitere Programme mit CUDA Unterstützung wie z.B Badaboom und TMPGEnc. Für wissenschaftliche Berechnungen gibt es ein Matlab Plugin. Der BOINC Client [Ber10] für verteiltes Rechnen der Universität Berkeley ist auch in einer Version mit CUDA Unterstützung erhältlich.

2.5.2 Unterschiede zwischen OpenCL und C

Wie bei NVIDIAs CUDA Architektur, wird der Device Code in einer modifizierten Version der Programmiersprache C entwickelt. Diese OpenCL C genannte Variante unterliegt, ähnlich wie CUDA, einigen Einschränkungen. Der Host Code kann wie bei CUDA mit Hilfe des gesamten Funktionsumfangs von C und C++ entwickelt werden. Im Gegensatz zu CUDA führt OpenCL keine erweiterte Syntax für das Ausführen eines Kernel ein, dies macht

einen speziellen Compiler nicht notwendig. Der Aufruf eines Kernel und die Übergabe von Argumenten an diesen erfolgen durch Funktionsaufrufe an die OpenCL API und werden dort übersetzt.

Einschränkungen

Die Einschränkungen von OpenCL überschneiden sich mit denen von CUDA, da meist dieselbe Hardware zugrunde liegt. Einige Einschränkungen sind [KG11] entnommen und folgend aufgelistet:

- Die Standardbibliothek von C kann nicht im Code der Kernel verwendet werden.
- OpenCL Funktionen unterstützen keine Rekursion.
- OpenCL Kernel müssen als Rückgabewert void besitzen.
- Schreibzugriffe auf Variablen mit einer Größe kleiner 32 Bit, wie zum Beispiel die Datentypen char, uchar, char2, uchar2, short, ushort oder half, werden erst seit Version 1.1 unterstützt. Also nicht von der aktuellen NVIDIA Implementierung.
- OpenCL Kernel und Funktionen, die auf einem OpenCL Device ausgeführt werden, können keine Variable Anzahl an Argumenten besitzen.
- Ein gerade ausgeführter Kernel kann nicht unterbrochen werden.

Funktionstypen

Bei OpenCL gibt es im Vergleich zu CUDA nur einen Funtktionstyp, den Kernel. Dieser wird mithilfe eines speziellen Schlüsselworts „**__kernel**“ definiert. Dieses wird bei der Deklaration der Funktion vorangestellt. Der Kernel muss zusätzlich in einer seperaten *.cl Datei vorliegen und angegeben werden, um von der OpenCL API korrekt erkannt zu werden.

Variablentypen

Bei OpenCL ist es, analog zu CUDA, ebenso möglich durch die Verwendung definierter Schlüsselwörter den Speicherbereich zu bestimmen. Diese werden ebenfalls der C Deklaration vorangestellt.

__global Dieses Schlüsselwort bestimmt, dass eine Variable im Globalen Speicher eines Device alloziert wird.

__constant Dieser Variablentyp beschreibt eine Variable, die sich ebenso im Globalen Speicher befindet, jedoch innerhalb eines OpenCL Kernel nur lesend verwendet werden kann. Diese Variablen können von allen Work-Items innerhalb eines Kernel gelesen werden.

__local Mit diesem Schlüsselwort wird eine Variable beschrieben, die im Lokalen Speicher (OpenCL) alloziert wird. Variablen mit dem Schlüsselwort „__local“ können von allen „Work Items“ innerhalb der gleichen Arbeitsgruppe gelesen und geschrieben werden.

__private Dieses Schlüsselwort bestimmt, dass eine Variable im Privaten Speicher (OpenCL) alloziert wird und daher nur innerhalb eines einzelnen Thread gelesen und beschrieben werden kann.

Typischer Ablauf eines OpenCL Programms

Der typische Ablauf einer OpenCL-Applikation setzt sich zusammen aus: Daten an das Device senden, OpenCL-Kernel ausführen und Resultate vom Device lesen. Dieser Ablauf soll anhand eines Beispiels zur Vektor Addition erläutert werden. Im Gegensatz zu CUDA sind anfangs jedoch mehr Schritte notwendig, um die OpenCL Laufzeitumgebung zu initialisieren.

OpenCL Gerät auswählen Der erste Schritt, der in einer OpenCL Anwendung durchgeführt werden muss, ist eine Liste an OpenCL Plattformen abzufragen. Dies erlaubt es dem Host-Programm OpenCL fähige Geräte zu erkennen und ihre Eigenschaften abzufragen. Im nächsten Schritt wird ein Gerät ausgewählt, wie in Algorithmus 2.7 dargestellt. Durch den zweiten Parameter der Funktion „clGetDeviceIDs“ kann bestimmt werden, welche Art von Gerät ausgewählt wird. Soll die GPU verwendet werden, so ist der Parameter „CL_DEVICE_TYPE_GPU“ anzugeben. Analog dazu kann „CL_DEVICE_TYPE_CPU“ verwendet werden, um die CPU auszuwählen.

Algorithmus 2.7: OpenCL Gerät auswählen

```
1  clGetPlatformIDs(1, &cpPlatform, NULL);
2  clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
```

Kontext und „Command Queue“ erzeugen Nachdem ein OpenCL Gerät ausgewählt wurde, ist es notwendig einen Kontext zu erzeugen, siehe hierzu Algorithmus 2.8. Ein OpenCL Kontext wird verwendet, um Objekte wie „Command Queues“ (Befehlswarteschlange), den allozierten Gerätespeicher oder Kernelfunktionen zu verwalten. Anschließend wird eine „Command Queue“, wie in Algorithmus 2.8 angegeben, erstellt. Die „Command Queue“ wird von OpenCL verwendet, um das Gerät zu kontrollieren. Jeder Befehl des Host an das Gerät, wie zum Beispiel ein Kernelaufruf oder ein Speichertransfer, wird durch die „Command Queue“ verwaltet. Für jedes Gerät muss mindestens eine oder auch mehrere „Command Queues“ erstellt werden, um mit ihm Berechnungen durchführen zu können.

Algorithmus 2.8: Kontext und Command Queue erzeugen

```
1 cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
2 cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
```

Speicherobjekte erstellen und zum Device übertragen Um einen OpenCL Kernel ausführen zu können, müssen sich analog zu CUDA alle notwendigen Daten auf dem ausführenden Gerät befinden. Da der Kernel nicht auf Speicherbereiche außerhalb des Device zugreifen kann, muss das Kopieren der Daten auf Host Seite passieren. Um dies zu ermöglichen, muss ein Speicherobjekt erstellt werden, mit dem es möglich ist Daten zwischen Host und Device auszutauschen, hierbei wird noch kein Speicherplatz belegt. Erst im zweiten Schritt, beim Übertragen der Daten, wird dann auch der tatsächliche Speicher auf der GPU belegt. Dieser Prozess mit anschließendem Kopieren der Daten ist in Algorithmus 2.9 dargestellt.

Algorithmus 2.9: OpenCL Buffer anlegen und Daten kopieren

```
1   // OpenCL Buffer Speicher Objekt auf der GPU allozieren
2   d_A = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) *
        szGlobalWorkSize, NULL, &ciErr1);
3   d_B = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) *
        szGlobalWorkSize, NULL, &ciErr2);
4   d_C = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, sizeof(cl_float) *
        szGlobalWorkSize, NULL, &ciErr2);
5   // Daten auf die GPU kopieren
6   clEnqueueWriteBuffer(cqCommandQueue, d_A, CL_FALSE, 0, sizeof(cl_float) *
        szGlobalWorkSize, h_A, 0, NULL, NULL);
7   clEnqueueWriteBuffer(cqCommandQueue, d_B, CL_FALSE, 0, sizeof(cl_float) *
        szGlobalWorkSize, h_B, 0, NULL, NULL);
```

Vor dem Kopieren müssen die Daten, wie in Algorithmus 2.1 schon gezeigt, auf dem Host alloziert und initialisiert werden. Bei OpenCL sind die Datentransfers stets Asynchron, es kann aber, wenn es relevant ist, mit dem Befehl „clFinish(cqCommandQueue)“ ein Warten auf die Übertragung erzwungen werden.

Bei OpenCL gibt es wie bei CUDA die Möglichkeit, wie in Algorithmus 2.10 dargestellt, Speicher im Host RAM zu allozieren, der nicht ausgelagert wird. Auch bei OpenCL wird dadurch die Geschwindigkeit bei der Übertragung erhöht.

Algorithmus 2.10: OpenCL pinned Memory

```
1       // Host Buffer anlegen
2       Pinned_Data = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE |
            CL_MEM_ALLOC_HOST_PTR, memSize, NULL, &ciErrNum);
3       // Pointer speichern
4       Host_data = (unsigned char*)clEnqueueMapBuffer(cqCommandQueue,
            cmPinnedData, CL_TRUE, CL_MAP_WRITE, 0, memSize, 0, NULL, NULL,
            &ciErrNum);
5       //Speicher auf der GPU anfragen
6       Dev_data = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, memSize,
            NULL, &ciErrNum);
7       //Daten kopieren
8       clEnqueueWriteBuffer(cqCommandQueue, Dev_data, CL_FALSE, 0, memSize,
            Host_data, 0, NULL, NULL);
```

OpenCL Kernel Der Aufbau des OpenCL Kernel ähnelt, wie in Algorithmus 2.11 zu sehen, sehr stark dem des CUDA Kernel. Durch die Verwendung von „clEnqueueNDRangeKernel“

beim Aufruf der OpenCL Kernel werden diese ebenfalls N-mal parallel ausgeführt. Der Index innerhalb des Kernel für die einzelnen Threads, kann in OpenCL mit der integrierten Funktion „get_global_id“ bestimmt werden.

Algorithmus 2.11: OpenCL Kernel zur Vektoraddition

```
1 __kernel void VectorAdd(__global const float* a, __global const float* b,
      __global float* c, int N)
2 {
3    //Index für jeweilige Berechnung
4    int id = get_global_id(0);
5    // Bereichsprüfung
6    if (id < N)
7    c[id] = a[id] + b[id];
8 }
```

Kernel Code einlesen und erstellen Um einen OpenCL-Kernel ausführen zu können, muss dieser erst gelesen werden. Dies kann in Form einer ausführbaren Binärdatei sein oder in Form des Quellcodes, der für das Gerät erst übersetzt werden muss. In diesem Beispiel wurde der Quellcode bereits in die Variable „cSourceCL“ eingelesen. Um den Quellcode übersetzen zu können, muss dieser zuerst zu einem Kernel Programm verarbeitet werden, dargestellt in Algorithmus 2.12 Zeile 1. Anschließend wird der Quellcode durch die OpenCL API speziell für das Gerät übersetzt. Dieser Schritt ist notwendig, da der Quellcode mehrere Funktionen besitzen kann. Um den OpenCL-Kernel aufzurufen muss zuerst, wie in Algorithmus 2.12 Zeile 3 zu sehen, ein Kernel Objekt erstellt werden. Jedes Kernel Objekt entspricht dabei einer einzelnen Kernel Funktion.

Algorithmus 2.12: OpenCL Kernel zur Vektoraddition

```
1    // Programm verarbeiten
2    cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char
         **)&cSourceCL, &szKernelLength, &ciErr1);
3    //Programm übersetzen
4    clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
5    // Kernel erzeugen
6    ckKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);
```

Argumente an den Kernel übergeben Um Argumente an den OpenCL Kernel übergeben zu können muss, wie in Algorithmus 2.13 angegeben, vorgegangen werden.

Algorithmus 2.13: Argumente an den Kernel übergeben

```
1    ciErr1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&d_A);
2    ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&d_B);
3    ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&d_C);
4    ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_int), (void*)&N);
```

Das erste Argument dieser Funktion ist der Kernel an den die Argumente übergeben werden sollen. Das zweite Argument bestimmt die Position des Parameters innerhalb des Kernelaufrufs. Im dritten Argument ist die Größe des Pointers, der im vierten Argument

übergeben wird, angegeben. Dies muss für jedes Argument eines Kernel durchgeführt werden.

Kernel ausführen Um die eigentlichen Berechnungen auf dem OpenCL Gerät durchzuführen, ist es notwendig die Kernel Funktion, wie in Algorithmus 2.14 zu sehen, aufzurufen. Die Variable „Global“ gibt dabei die Gesamtanzahl an „Work Items“ an. Die Variable Lokal die Anzahl an „Work-Items“ pro Arbeitsgruppe. Bei CUDA sind, wie in Abschnitt 2.5.1 gezeigt, die Parameter Blockgröße und Threadanzahl pro Block anzugeben. Durch diese Ähnlichkeit beim Aufruf ist eine Portierung in die eine oder andere Richtung gewährleistet.

Algorithmus 2.14: OpenCL Kernlaufruf

```
1    size_t Lokal = 256, Global = N;
2    clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, &Global, &Lokal,
         0, NULL, NULL);
```

Ergebnisse zurück zum Host übertragen Nach Ausführen des Kernels ist es erneut notwendig das Resultat der Berechnungen, wie in Algorithmus 2.15 dargestellt, wieder auf den Host zu transferieren.

Algorithmus 2.15: Ergebnisse zurück zum Host übertragen

```
1  clEnqueueReadBuffer(cqCommandQueue, d_C, CL_TRUE, 0, sizeof(cl_float) *
       szGlobalWorkSize, h_C, 0, NULL, NULL);
```

OpenCL Objekte freigeben Nach erfolgreicher Berechnung, sollten die nicht mehr benötigten Ressourcen, wie in in Algorithmus 2.16 gezeigt, freigegeben werden.

Algorithmus 2.16: Ressourcen freigeben

```
1    free(cSourceCL); //OpenCL Objekte freigeben
2    clReleaseKernel(ckKernel);
3    clReleaseProgram(cpProgram);
4    clReleaseCommandQueue(cqCommandQueue);
5    clReleaseContext(cxGPUContext);
6    clReleaseMemObject(d_A);
7    clReleaseMemObject(d_B);
8    clReleaseMemObject(d_C);
9    free(h_A); //Host Variablen freigeben
10   free(h_B);
11   free(h_C);
```

2.5.3 Hardwarenähe

Es gibt sowohl bei CUDA als auch bei OpenCL die Möglichkeit, in einer Maschinensprache die jeweiligen Kernel zu analysieren und zu manipulieren. Bei NVIDIA nennt sich diese Sprache „Parallel Thread Execution“ (PTX) [NVI11h] und findet sowohl bei CUDA als auch

bei OpenCL Verwendung, da es sich um eine hardwarespezifische Sprache handelt, siehe hierzu Abbildung 2.7. Bei AMD gibt es äquivalent dazu die „Intermediate Language“ (IL) [AMD11b], ebenfalls in Abbildung 2.7 zu sehen. Dies entspricht dem Assembler Code bei CPUs und sei hier nur als Möglichkeit erwähnt.

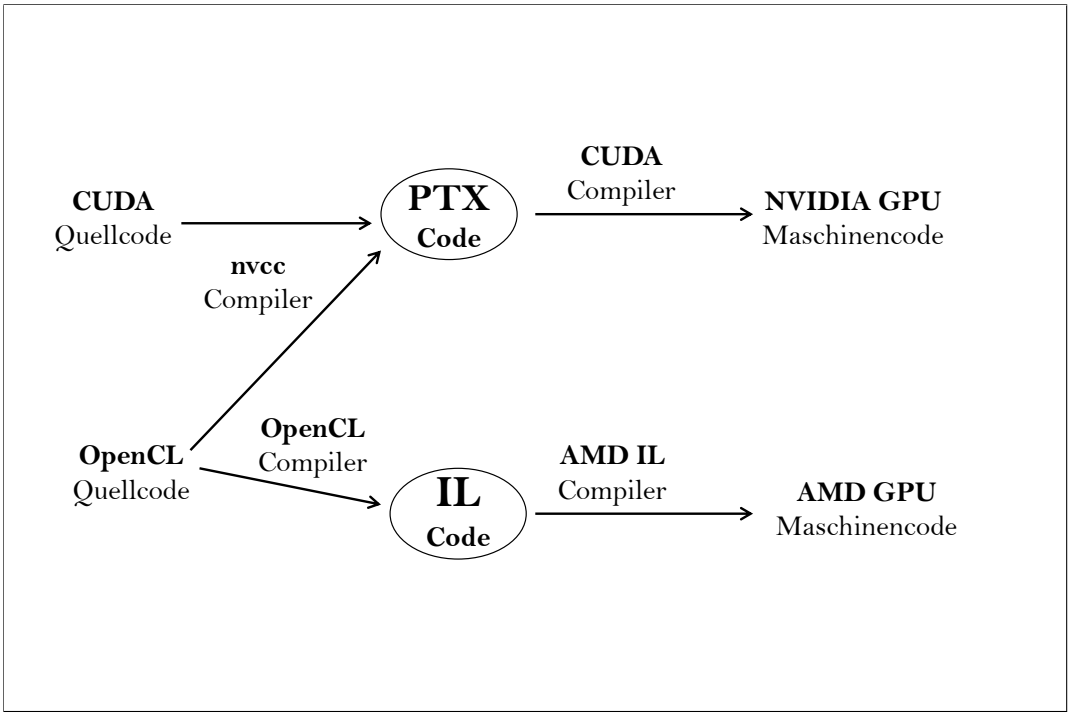

Abbildung 2.7: CUDA PTX, AMD IL

2.6 Unterschiede zwischen CUDA und OpenCL

CUDA und OpenCL ermöglichen beide die Verwendung der GPU abseits der Grafikausgabe. CUDA lässt sich allerdings zum jetzigen Zeitpunkt ausschließlich auf NVIDA Grafikkarten ausführen. Im Gegensatz dazu steht OpenCL mit seiner Möglichkeit, jegliche Art von Recheneinheiten, z.B. auch den CELL Prozessor der Playstation 3, verwenden zu können. Insofern vom Hersteller eine entsprechende Implementierung bereitgestellt und vor allem auch gepflegt wird. Der NVIDIA Treiber ist immer noch bei Version 1.0, die Intel Treiber noch in der Betaphase ebenso die IBM Treiber für den CELL Prozessor. Die Einsatzbereiche sind bei OpenCL dadurch vielfältiger, es bringt aber auch mehr Aufwand bei der Programmierung mit sich. Die Eigenheiten der Hardware müssen berücksichtigt werden, die Verwaltung der Geräte liegt beim Benutzer und man ist immer auf die jeweilige Implementierung des Herstellers angewiesen, welche Erweiterungen der Spezifikation letztendlich umgesetzt sind. Die Ausführung mehrerer Kernel beispielsweise ist in der OpenCL Spezifikation aufgeführt, wird aber bisher noch von keiner Implementierung bereitgestellt. Bei CUDA

ist durch die Konzentration auf denselben Hersteller der Hardware und der Software eine bessere Unterstützung des Standards gegeben, die soeben erwähnten parallelen Kernel sind mit CUDA durch die Verwendung von Streams möglich. Auf der Habenseite von CUDA stehen auch die zahlreich vorhandenen Bibliotheken im mathematischen Bereich und die besseren Möglichkeiten des Debugging (Fehleranalyse) durch mehr zur Verfügung stehende Programme. Für OpenCL werden diese mit der Zeit immer besser werden und der Unterschied geringer. Durch den Mehraufwand bei der Verwaltung der Geräte, hat man aber auch mehr Kontrolle über sie. Bei CUDA passiert vieles automatisch und dadurch für den Benutzer nicht sichtbar.

Erklärung	CUDA	OpenCL	Eigenschaft
Gerät, Grafikkarte	CUDA GPU	Device	
Funtkionstypen	__device__	__kernel	keine Differenzierung bei OpenCL
	__global__	__kernel	keine Differenzierung bei OpenCL
	__host__	__kernel	keine Differenzierung bei OpenCL
Variablentypen	__device__	__global	im globalen Speicher
	__constant__	__local	im konstanten Speicher
	__shared__	__constant	im gemeinsamen Speicher
Ausführung	Thread	Work-Item	kleinste Zerlegegung
	Block		
	Grid	Work-Group	Arbeitsgruppe

Tabelle 2.4: Unterschiede einiger Begriffe von CUDA und OpenCL

3 Echtzeitfähigkeit und Ressourcenverbrauch bei GPGPU

3.1 Einführung Echtzeitfähigkeit

Ein Echtzeitsystem ist nach ([Kop11] Seite 2) definiert als:

> ... a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced.

Soll ein System Echtzeitanforderungen erfüllen, so müssen die korrekten Ergebnisse von Berechnungen demnach innerhalb einer gewissen Zeit garantiert vorliegen, also bevor eine zuvor festgelegte Frist abgelaufen ist. Da die Zeit, in dessen Rahmen das System antworten muss, selbst definiert wird, ist dies nicht unbedingt mit einem möglichst schnellen System gleichzusetzen. Vielmehr steht die Sicherheit im Vordergrund, mit der man Aussagen über das zeitliche Verhalten treffen kann. Ein einfaches Beispiel für ein Echtzeitsystem im Alltag, ist eine funktionierende Uhr. Diese muss, wenn sie denn korrekt arbeitet, einmal in der Minute ihren Minutenzeiger bewegen und einmal in der Stunde ihren Stundenzeiger. Die festgelegte Zeit an den Minutenzeiger liegt bei dieser Uhr also bei einer Minute, und für den Stundenzeiger bei einer Stunde. Bei einem Echzeitsystem muss zusätzlich zur Korrektheit des Ergebnisses, auch die Antwortzeit eingehalten werden, sonst ist es fehlerhaft. Durch diese Anforderungen für das System, ergeben sich zusätzliche Anforderungen an die Umgebung. Denn es reicht nicht aus, wenn das System schnell genug ist, d.h. entsprechende Berechnungen in einer vorgegeben Zeit abarbeiten kann. Es muss auch sichergestellt werden, dass benötigte Daten in der vorgegeben Zeit zum System gelangen können bzw. von entsprechenden Sensoren bereitgestellt werden. Daher muss für jedes Teil eines Gesamtsystems die Echtzeitfähigkeit sichergestellt sein. Ein weiterer wichtiger Faktor sind die zur Verfügung stehenden Ressourcen, denn ein Echtzeitsystem, das mit einem Benutzer oder einer Anfrage korrekt arbeitet, kann bei einem Mehrbenutzerbetrieb oder mehreren gleichzeitigen Anfragen evtl. nicht mehr fehlerfrei arbeiten.

Diese Einhaltung von Vorgaben ist im Hinblick auf sicherheitsrelevante Anwendungen unumgänglich, insbesondere wenn Menschenleben von dessen Korrektheit abhängen. Solch ein System wird auch als hartes Echtzeitsystem bezeichnet. Eine Ausgabe nach der festgelegten Frist ist in solch einem Fall zu verwerfen und kann das gesamte System in Frage stellen. In den meisten heute eingesetzten Systemen sind die Echtzeitanforderungen jedoch nicht ganz so streng wie in der Theorie beschrieben. Man findet meistens eine Mischung verschiedener Anforderungen die definieren inwieweit eine Abweichung davon noch gestattet ist. Bei einem

Videoschnittprogramm ist es z.B. ausreichend in einer gewissen Zeitspanne auf eine Anfrage des Benutzers zu reagieren. Systeme dieser Klassifikation werden weiche Echtzeitsysteme genannt. Zum besseren Verständnis ist diese Einteilung noch einmal in Tabelle 3.1 aus [Kop11] dargestellt.

Eigenschaft	harte Echtzeit	weiche Echtzeit
Antwortzeit	immer einzuhalten	versucht einzuhalten
Verhalten unter Vollast	berechenbar	vermindert
Zeitvorgabe	Umwelt	Benutzer
Sicherheit	meist kritisch	unkritisch
Fehlererkennung	selbständig	Benutzerunterstützt

Tabelle 3.1: Harte und weiche Echtzeit im Vergleich

3.2 Anwendungsfälle

Für Echtzeitsysteme gibt es vielfältige Einsatzbereiche, ohne die im täglichen Leben vieles nicht so einfach funktionieren würde. Angefangen bei der Motorsteuerung in einem Auto, die sehr starke Anforderungen an die Reaktionszeit stellt. In modernen Fahrzeugen muss für jede Einspritzung der Zeitpunkt und die Benzinmenge berechnet werden. Bei einem Achtzylinder Motor mit 6000 Umdrehungen in der Minute sind das 800 Berechnungen pro Sekunde. Das lebensrettende Antiblockiersystem (ABS) ist ebenso ein Echtzeitsystem. Über Sensoren wird das Blockieren der Reifen festgestellt. Diese Information gelangt zu einem Steuergerät und es muss innerhalb kürzester Zeit berechnet werden, welches Rad wie angesprochen werden muss, um das Blockieren zu beseitigen. Dauert die Berechnung zu lange kann es ein Sicherheitsrisiko darstellen.

Inzwischen gibt es auch Fahrzeuge bei denen der Tachometer in digitaler Form auf einem Display ausgegeben wird. Die Aufgabe der Ausgabe übernimmt eine Grafikkarte, die ihre Daten zur Geschwindigkeit von einem Sensor erhält. Nicht nur in der Automobilindustrie, auch in der Medizintechnik und vielen anderen Bereichen ist die Echtzeitanforderung eine Voraussetzung. Durch GPGPU kann die Berechnungszeit im Idealfall gesenkt werden und dadurch die Antwortzeit eines Systems erhöht werden, um die Echtzeitfähigkeit für mehr Anwendungsbereiche sicherzustellen. Das bedeutet z.B. mit CUDA und OpenCL die hardwarebeschleunigte 2D/3D-Darstellung des Tachometers gezielt zu unterstützen oder zu ergänzen. Üblicherweise werden dazu OpenGL-Befehle verwendet, diese können mittels CUDA- oder OpenCL-Befehlen durch vergleichbare Berechnungen ersetzt werden.

3.3 Einflussfaktoren

Die relevanten Einflussfaktoren bei Grafikkarten für das Echtzeitverhalten und eine mögliche Interoperabilität, lassen sich grob in 4 Bereiche einordnen die es genau zu analysieren gilt.

1. Analyse der Geschwindigkeit bei der Datenübertragung
2. Speicherverwaltung
3. Reihenfolge bei der Abarbeitung von Befehlen (das Scheduling)
4. Kontextverwaltung

Die Analyse der Geschwindigkeit bei der Datenübertragung ist ein wichtiges Kriterium, denn ohne Daten kann nichts berechnet werden. Nach [NVI11c] und [KG11] ist der Engpass bei GPGPU-Anwendungen der Speichertransfer vom Host zur GPU und wieder zurück. Dieser Transfer kann demnach den Vorteil einer schnelleren Berechnung wieder zunichte machen. Für die eigentliche Berechnungsdauer der einzelnen arithmetischen Befehle sei an dieser Stelle auf die Spezifikation des jeweiligen Herstellers verwiesen, eine der wenigen Informationen die gerne zur Verfügung gestellt werden, und wie in [WPSAM10] gezeigt, auch zuverlässig sind. Der Einfluss auf die Echtzeitfähigket ist jedoch vernachlässigbar, da die anderen Faktoren deutlich überwiegen.

Um Echtzeitgarantien in Bezug auf den Speicher geben zu können, müssen die zugrundeliegenden Daten stets korrekt sein. Denn falsche Ergebnisse nützen nichts, auch wenn sie innerhalb der festgelegten Zeit ausgegeben werden. Dies ist besonders wichtig in Konkurrenzsituationen, wie sie entstehen wenn mehrere Anwendungen die Grafikkarte gleichzeitig verwenden. Da nur eine begrenzte Menge an Speicher auf der Grafikkarte vorhanden ist, muss untersucht werden, wie sich die verschiedenen Implementierungen (CUDA, OpenCL, OpenGL) gegenseitig beeinflussen und ob ein, wie auch immer gearteter, Speicherschutz zwischen verschiedenen Kontexten existiert. Der Ressourcenverbrauch, der beim Allozieren von Daten auf der Grafikkarte entsteht, ist ein Kriterium das großen Einfluss haben kann. Denn die Verwaltung der Speicherseiten auf der Grafikkarte und die Aufteilung auf Blöcke sind nicht bekannt. Es kann aber ein sehr großer Unterschied zwischen der phsysikalischen Speichermenge und der effektiv nutzbaren Menge durch die Art der Organisation entstehen. Mit geeigneten Testfällen kann man davon ausgehen feste Werte für die Blockgröße und die interne Organisation bestimmen zu können und damit die genaue Organisation der Speicherseiten und die Ablage von Daten weiter zu untersuchen.

Das Scheduling der Grafikkarte wirkt sich direkt auf das Antwortverhalten aus, und kann die Laufzeit sehr stark beeinflussen wenn viele Anfragen gleichzeitig an die Grafikkarte geschickt werden. Die Anforderungen an den Scheduler sind in einem Echtzeitsystem die Vorhersagbarkeit und die Einhaltung einer Zeitvorgabe. Ziel der Evaluierung ist es daher die Funktionsweise des Schedulers zu untersuchen und Möglichkeiten der Kontrolle zu finden. Die Befehle werden, bevor sie den Scheduler erreichen, in einer Warteschlange (Queue) verwaltet, bei OpenCL wird diese Queue direkt mit angegeben wenn ein Befehl abgesetzt wird, bei CUDA übernimmt dies die Laufzeitumgebung. Um Aussagen über das Scheduling treffen zu können, ist es nicht ausreichend nur den Scheduler zu untersuchen sondern ebenso notwendig die Verwaltung der Befehle in der Queue zu kennen. Bei CUDA gibt es durch die Funktionalität der Streams, die Möglichkeiten Kernel parallel von der Grafikkarte berechnen zu lassen. Es wird daher untersucht inwieweit man damit auf die Reihenfolge beim Scheduling Einfluss nehmen kann.

Die Auswirkungen der Kontextverwaltung auf die Echtzeitfähigkeit sind im Hinblick auf den Ressourcenverbrauch durch die Erstellung und eine mögliche maximale Anzahl von Interesse. Es gilt zu untersuchen inwieweit man Einfluss auf die Verwaltung nehmen kann. Auch im Hinblick auf die Synchronisierung von Daten zwischen verschiedenen Kontexten von CUDA und OpenCL mit OpenGL.

3.4 Messmethoden

Um Engpässe zu finden und Aussagen über eine erfolgreiche Implementierung der Testfälle machen zu können, sind Evaluierungen notwendig. Diese können nur sinnvoll durchgeführt werden, wenn es festgelegte Szenarien gibt, in denen die benötigte Rechenleistung und der Ressourcenverbrauch erfasst sind. Es werden sogenannte Benchmarks erstellt, die mehrmals hintereinander durchgeführt werden, um zuverlässige Werte zu erhalten. In dieser Arbeit werden die Benchmarks, wenn nicht anders angegeben, jeweils zehnmal hintereinander durchlaufen. So werden zufällige Spitzen und Senken in den Messungen ausgeglichen und für die Auswertung verlässliche Werte ermittelt.

Wie bei GPGPU-Programmen üblich, werden die Anfragen an die GPU von einem Host-Programm aufgerufen. Vor und nach dieser Anfrage an die GPU wird ein Zeitstempel in einem Array gespeichert, um die Laufzeit einfach mitprotokollieren zu können. Dies geschieht innerhalb einer Schleife, um die Quantität der Messungen einfach erhöhen zu können. Erst nach der vollständigen Berechnung werden die Werte des Arrays in eine Logdatei geschrieben, um die Auswirkungen auf die Messung gering zu halten und qualitativ gute Ergebnisse zu bekommen. In ersten Tests war der Overhead ohne diese Vorgehensweise deutlich höher und musste zeitintensiv aus den Messungen herausgerechnet werden.

3.4.1 Zeitmessung im Detail

Zur Verifikation der Zeitmessung wurde die Software NVIDIA Parallel Nsight, für die Tests mit NVIDIA Karten, und der AMD APP Profiler für die Tests mit der Fire Pro Karte angewandt. Durch die Anwendung der Profiling-Tools wird allerdings die Ausführung der Kernel beeinflusst und so musste eine eigene Zeitmessung für valide Messungen gefunden werden. Diese ist in Algorithmus 3.1 zu sehen.

Algorithmus 3.1: Art der Zeitmessung

```
1 #include <windows.h>
2 QueryPerformanceFrequency(&ticksPerSecond);
3 for(int i=0; i<X; i++)
4 {
5  QueryPerformanceCounter(&start_ticks);
6  //GPGPU Berechnung, Speichertransfer etc.
7  //dieser Teil wird gemessen
8  QueryPerformanceCounter(&end_ticks);
9  //vergangene Zeit in Sekunden
```

```
10   zeit[i] = ((double)(end_ticks.QuadPart-
         start_ticks.QuadPart)/(double)ticksPerSecond.QuadPart);
11   start[i] = start_ticks; //Zeitstempel vor dem Aufruf
12   ende[i] = end_ticks; //Zeitstempel nach dem Aufruf
13 }
```

In der Schleife des Algorithmus 3.1 wird mit Hilfe des High-Resolution Performance Counters [Net07] ein Zeitstempel abgefragt und gesichert. Die Genauigkeit des Zählers liegt bei einer Mikrosekunde aber die Auflösung noch einmal deutlich höher. Dadurch kann auch bei mehreren Threads die Abfolge der Aufrufe sehr genau bestimmt werden, selbst wenn dazwischen weniger als eine Mikrosekunde vergangen ist. Jeder Kernelaufruf lässt sich so, mithilfe der Logdatei, einer bestimmten Position in der Abfolge zuordnen.

3.4.2 Konkurrenzsituationen mit Hilfe von Threads

Um Konkurrenzsituationen zwischen zwei Kernelaufrufen zu simulieren, wenn mehrere Anwendungen die Grafikkarte gleichzeitig verwenden wollen, wurde auf OpenMP (Open Multi-Processing) [Boa] zur Threaderzeugung zurückgegriffen. Da es sich sowohl für CUDA als auch OpenCL gleichermaßen eignet. In Algorithmus 3.2 sieht man Beispielhaft die Erzeugung von zwei Threads und die Zuordnung über ihre ID zu zwei unterschiedlichen Programmabschnitten. Für die Tests wurden dann in den Verzweigungen entsprechende Aufrufe an die GPU geschickt.

Algorithmus 3.2: Ein Beispielcode für OpenMP

```
1  #include <omp.h> //OpenMP einbinden
2  int main( void ) {
3    //Initialisierung der CUDA oder OpenCl spezifischen Daten
4    omp_set_num_threads(2); //zwei Threads
5      #pragma omp parallel for //die folgende for-Schleife wird auf 2 Threads
           verteilt
6    for(int i=0; i<x; i++)
7    {
8       unsigned int cpu_thread_id = thread_id; //Thread-ID abfragen
9       if (cpu_thread_id==0)
10     {
11         //der erste Thread macht einen Kernel Aufruf
12      }
13      else if (cpu_thread_id==1)
14     {
15          //der zweite Thread macht seinen Kernel Aufruf
16        //die Daten müssen dabei unabhängig vom ersten sein
17        }
18   }
19 }
```

3.4.3 Bewertung von Analysewerkzeugen

NVIDIA Parallel Nsight

Parallel Nsight von NVIDIA ist eine Profiling Umgebung für NVIDIA Karten, das sich in die Entwicklungsumgebung Microsoft Visual Studio integriert. Parallel Nsight enthält einen CUDA Debugger und Profiling-Werkzeuge. Die Profiling-Werkzeuge von Parallel Nsight helfen zu analysieren, wie die Arbeitslast innerhalb einer Applikation auf die einzelnen GPU Kerne verteilt ist. Das Ausführen von CUDA Kernel und CPU Ereignissen kann wie in in Abbildung 3.1 zu sehen ist, in einer grafischen Zeitlinie erfolgen.

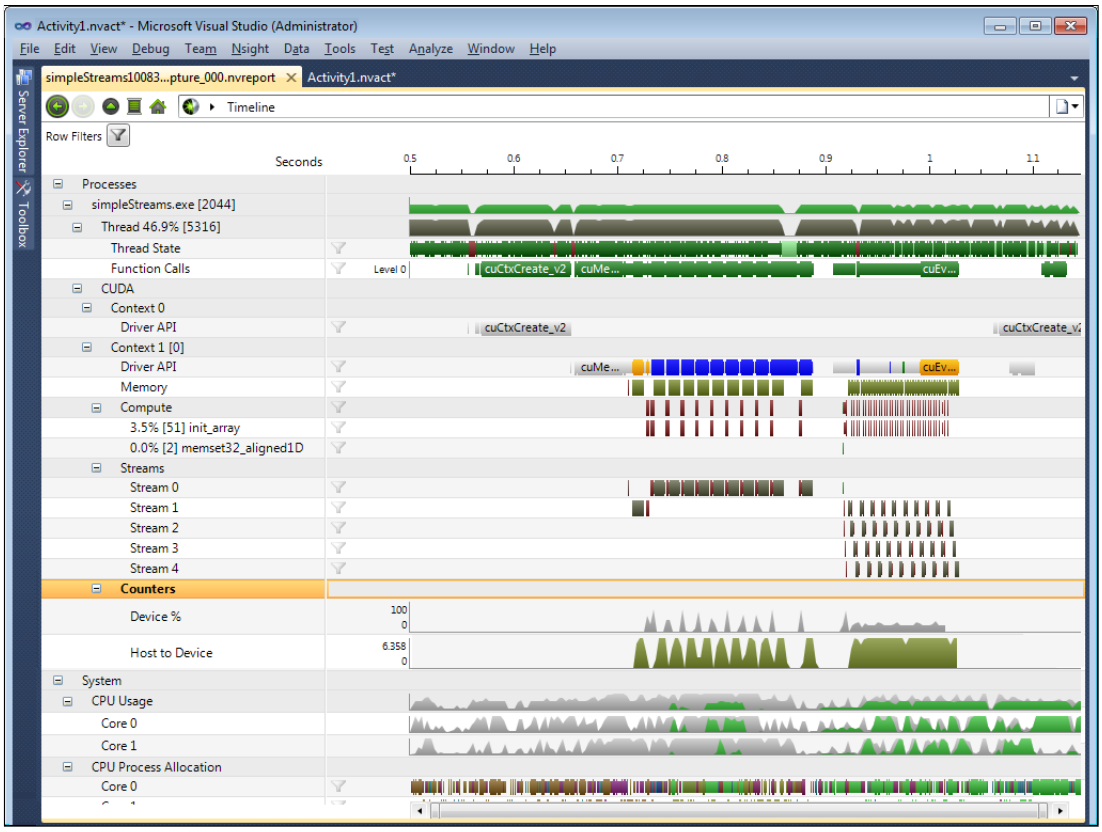

Abbildung 3.1: Nvidia Parallel Nsight Analyse Werkzeug

AMD APP Profiler

Von AMD wird der „APP Profiler“ für das Profiling von OpenCL Kernel zur Verfügung gestellt. Der „APP Profiler“ wird zur Laufzeit von Programmen angewendet und bietet ähnlich wie Parallel Nsight von NVIDIA eine grafische Darstellung des zeitlichen Ablaufs, siehe Abbildung 3.2, eines Programms.

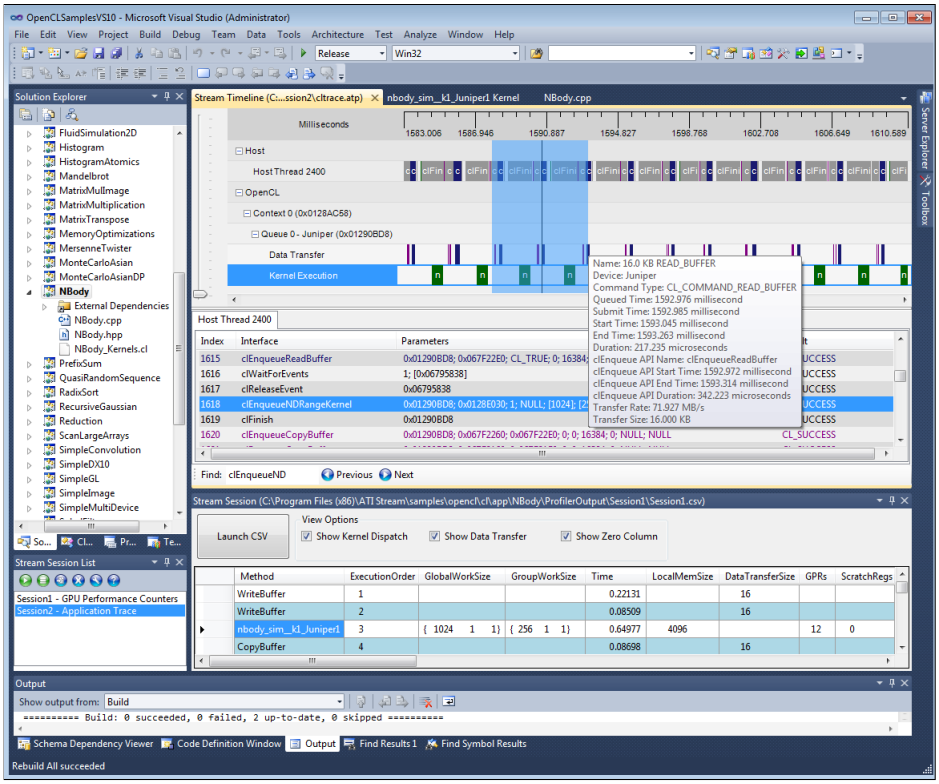

Abbildung 3.2: AMD APP Profiler

GPU-Z

Um Auskünfte über die Auslastung der Grafikkarte zu bekommen, wird ein weiteres Tool, GPU-Z [W1z11], verwendet. Dieses gibt Auskunft über den Speicherverbrauch, siehe Abbildung 3.3. Diese Daten lassen sich in einer Logdatei festhalten und wurden vornehmlich für die Untersuchung des Speichers verwendet.

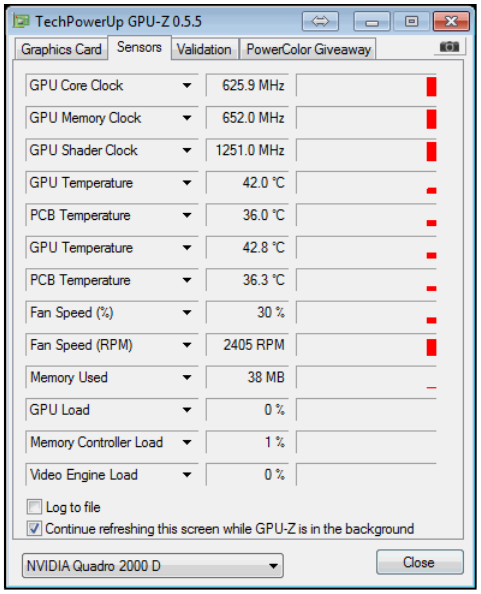

Abbildung 3.3: GPU-Z

Mit diesem Analysetool kann man allerdings nur eine Näherung erreichen. Da zum einen die Genauigkeit nur bei einem Megabyte liegt und zum anderen die Funktionsweise auch nach Rücksprache mit dem Autor nicht ganz transparent ist, da die Funktionen zum Teil unter einem Geheimhaltungsvertrag (NDA) stehen.

Zusammenfassend sind folgende wichtigen Punkte zu eruieren:

- Performanceanalyse: Die Datenübertragung zwischen Host und Grafikkarte, und der Vorteil von „Pinned Memory“, Speicher der auf Seite des Host nicht ausgelagert wird.
- Speicherverwaltung:Es ist sehr wichtig den Ressourcenverbrauch bei der Speicherverwaltung zu kennen, da alle Daten die zur Berechnung verwendet werden, zunächst auf die Grafikkarte kopiert und dort verwaltet werden müssen.
- Scheduling: Das Scheduling, also die Reihenfolge wie Befehle abgearbeitet werden und die Möglichkeiten der Einflussnahme darauf, sind wichtige Faktoren um eine Aussage über die Echtzeitfähigkeit treffen zu können. Damit eng verbunden ist das Vorhandensein und die Analyse der Eigenschaften einer Queue (Warteschlange) für die Befehle an die Grafikkarte.
- Kontextverwaltung: Inwieweit gibt es eine Handhabe bei der Verwaltung verschiedener Kontexte auf der GPU und kann darauf Einfluss genommen werden. Dies ist auch wichtig im Hinblick auf die Synchronisierung von Daten zwischen verschiedenen Kontexten von CUDA, OpenCL und OpenGL.

4 Evaluierung

4.1 Die Testumgebung

Für die Evaluation wurden folgende Systeme verwendet:

4.1.1 Testsysteme

- **AMD-Fire-Pro:** AMD Fire Pro V5900 Grafikkarte mit 2 GB RAM, Intel Q8600 Quadcore 4 GB RAM, Treiber: AMD Catalyst 11.9 erschienen am 28. September 2011, unterstützt OpenCL in Version 1.1, *AMD Haupttestsystem*
- **NVIDA-Quadro:** NVIDIA Quadro 2000 D Grafikkarte mit 1 GB RAM, Intel i7 Quadcore 6 GB RAM, Treiber: NVIDIA Quadro/Tesla 275.89 am 11. August 2011 erschienen, unterstützt CUDA 4.0 und OpenCL 1.0, *NVIDIA Haupttestsystem*
- **NVIDIA-GTX:** Nvidia GTX 260 Grafikkarte mit 896 MB RAM, Intel E8400 Dualcore 8 GB RAM, Treiber: NVIDIA GeForce 280.26 am 9. August 2011 erschienen, unterstützt CUDA 4.0 und OpenCL 1.0, zur Verifikation bestimmter Tests
- **NVIDIA-NVS:** NVIDIA NVS 3100M mit 512 MB RAM, i7 Dualcore 4 GB RAM, Treiber: NVIDIA Verde 280.26 am 9. August 2011 erschienen, unterstützt CUDA 4.0 und OpenCL 1.0, zur Verifikation bestimmter Tests

Auf allen Systemen ist das Betriebssystem Windows 7 x64 Professional mit Service Pack 1 von Microsoft und die Entwicklungsumgebung Microsoft Visual Studio 2008 mit Service Pack 1 installiert. An zusätzlichen Programmen kamen die Folgenden zum Einsatz.

4.1.2 Software

- NVIDIA Parallel Nsight 2.0, zum Profiling der GPU
- NVIDIA CUDA Toolkit 4.0, mit CUDA 4.0 und OpenCL 1.0 Unterstützung
- NVIDIA GPU-Computing SDK 4.0
- NVIDIA CUDA Tools 4.0
- AMD APP SDK v2.5, mit OpenCL 1.1 Unterstützung
- GPU-Z zur Überwachung der GPU Auslastung

Eine Besonderheit von Windows 7 ist die integrierte Überwachung der Grafikkarte, reagiert diese über mehrere Sekunden nicht, wird der Treiber neu gestartet und dadurch auch jedes Programm auf der GPU beendet [Mico6]. Für die Untersuchungen ist es erforderlich dieses Verhalten zu unterbinden. Mit einer Änderung in der Windows Registrierungsdatenbank, wie in [Mico9] beschrieben, ist es möglich diese Überwachung zu deaktivieren.

4.2 Performance

In diesem Abschnitt wird untersucht, inwieweit das Host-System Einfluss auf die Messungen hat, um Störungen ausschließen zu können. In Unterabschnitt 4.2.2 ist die Geschwindigkeit bei der Übertragung von Daten zwischen dem Host und der GPU und beim Kopieren im Grafikkartenspeicher Gegenstand der Betrachtung. Es werden Möglichkeiten gesucht diese Geschwindigkeit zu erhöhen, um die Antwortzeit des Systems zu verbessern.

4.2.1 Overhead des Host-Programms

Wie schon in Kapitel 3.4 erläutert wird die Messung bei jedem Testfall auf Hostseite durchgeführt um Auswirkungen auf die Messungen zu verhindern, die durch Profiling Tools oder sonstige Manipulationen auf der Grafikkarte entstehen. Obwohl viel Aufmerksamkeit darauf verwendet wurde die Einflüsse des Host gering zu halten, kann man sie nicht ausschließen. Von daher ist es für die folgenden Untersuchungen wichtig, den erzeugten Mehraufwand zu kennen, um ihn im Bedarfsfall herausrechnen zu können.

Zeitmessung im Detail

Wie dem Algorithmus 3.1 zu entnehmen, wird die Zeitmessung, wenn nicht anders angegeben, bei den Evaluierungen dieser Diplomarbeit durchgeführt. Für die Einschätzung des Ressourcenbedarfs auf der Host Seite wird die Messung wie in Algorithmus 4.1 zu sehen ist angepasst, um den Overhead zu überprüfen.

Algorithmus 4.1: Overhead des Host

```
1 QueryPerformanceFrequency(&ticksPerSecond);
2 QueryPerformanceCounter(&start_ticks);
3 QueryPerformanceCounter(&end_ticks);
4 for(int i=0; i<x; i++)
5 {
6  QueryPerformanceCounter(&end_ticks);
7  // Zeit in Sekunden
8  time0[i] = ((double)(end_ticks.QuadPart-
       start_ticks.QuadPart)/(double)ticksPerSecond.QuadPart);
9  logfileS[i]=start_ticks; //Zeitstempel vor dem Aufruf
10 logfileE[i]=end_ticks; //Zeitstempel nach dem Aufruf
11 add<<<64,64>>>(dev_a, dev_b, dev_c); //GPGPU Aufruf
12 QueryPerformanceCounter(&start_ticks);
```

```
13 }
```

Um die Verzögerung des Kernelaufrufs respektive des Speichertransfers zu eruieren, wird die Zeitmessung aus Algorithmus 3.1 auf alle verwendeten Kernel (CUDA und OpenCL) angewendet.

Analyse

Die Schleife erzeugt einen nicht messbaren Overhead. Der Aufruf des Kernel bei CUDA hingegen benötigt 0,002 ms - 0,003 ms. Diese Schwankung hängt nicht mit der Größe des Kernel zusammen. Sie ergibt sich über mehr als 10.000 durchgeführte Messungen. Bei OpenCL ist eine Verzögerung durch den Aufruf des Kernel nicht zu Messen. Diese Werte treffen selbstredend nur für das asynchrone absetzen von Befehlen oder das Kopieren von Daten zu. Im Gegensatz dazu steht das synchrone Kopieren bzw. Ausführen von Kernel. Hierbei wird das Host-Programm für die gesamte Dauer der entsprechenden Operation blockiert, respektive es wartet auf die Fertigstellung der Berechnung auf der Grafikkarte.

Durch die gewonnen Ergebnisse kann ein Verfälschen der Ergebnisse durch den Host minimiert werden, und die Messergebnisse lassen sich ohne aufwendiges herausrechnen eines Overhead meist direkt verwenden.

4.2.2 Speicherdurchsatz

Zunächst wird die Durchsatzrate, welche Menge an Daten pro Sekunde kopiert werden können, bei der Datenübertragung von CUDA und OpenCL mit dem NVIDIA-Quadro System gemessen und die Auswirkungen des „Pinned Memory“ untersucht. Hierzu werden in mehreren Durchgängen verschieden große Speicherbereiche zwischen GPU und Host, bzw. direkt auf der GPU zwischen verschiedenen Bereichen kopiert. Mit Hilfe der gemessenen Zeit und dem Wissen der Größe der einzelnen Bereiche kann damit die Durchsatzrate ermittelt werden. Um aussagekräftige Ergebnisse zu erhalten werden verschiedene Speichergrößen (1 KB bis 64 MB) verwendet und jeweils 10 Durchläufe gemacht. Die so gewonnenen Ergebnisse werden gemittelt und die jeweilige Abweichung mit angegeben.

Bei der Option „Pinned“ wird die Besonderheit, Speicher im Host anlegen zu können der nicht ausgelagert wird, bei der Allokation genutzt. Siehe hierzu Kapitel 2.5.1 für die Vorgehensweise bei CUDA und Kapitel 2.5.2 für OpenCL. Um die Untersuchung zu vervollständigen wird unter OpenCL zum Vergleich dieselbe Messung mit dem AMD-Fire-Pro System durchgeführt, mangels CUDA Implementierung für AMD Grafikkarten kann diese nur für OpenCL durchgeführt werden.

Zur Messung wurde der in Algorithmus 4.2 angegebene Ansatz verwendet.

Algorithmus 4.2: Speicherdurchsatz Pseudocode

```
1 FOR Daten 1 KB to 64 MB DO
2  FOR i in 1 to 10 DO
3    Beginn Zeitmessung
4      Daten übertragen
5    Ende Zeitmessung
6    Durchsatz berechnen und speichern
7  END
8  Gesamtdurchsatz berechnen und die Summe ausgeben
9 END
```

Die Übertragung der Daten aus dem RAM des Hosts auf den Speicher der Grafikkarte, geschieht über den PCI-Express Bus. Bei den zwei verwendeten Testsystemen handelt es sich dabei um die Version 2.0. Daraus resultiert nach Spezifikation [SIG11] eine theoretische Maximalleistung von 8 GB/s zwischen Host und Grafikkarte, die Anbindung auf der GPU an ihren eigenen Speicher liegt im Fall des NVIDIA-Quadro Systems bei theoretischen 41,7 GB/s, das AMD-Fire-Pro System mit theoretischen 64 GB/s liegt mit 53% deutlich darüber, diese Werte sind jeweils Herstellerangaben.

Testergebnisse

Die RAM Zugriffszeit des Host ist nicht immer konstant, denn es gibt Zugriffslatenzen die durch das Betriebssystem und die Hardware beeinflusst sind. Dies schlägt sich auch in der Streuung der Ergebnisse nieder und ist in der Abweichung angegeben. Die Abweichung wurde aus 10 Messergebnissen mit folgender Formel errechnet.

$$((GroessterWert - kleinsterWert) \cdot 100\%) / GroessterWert$$

und ist in Prozent angegeben.

Einstellung	CPU zu GPU	GPU zu CPU	auf der GPU	Abweichung
CUDA	3303 MB/s	3393 MB/s	26109 MB/s	< 3 %
CUDA pinned	5848 MB/s	6107 MB/s	26109 MB/s	< 3 %
OpenCL	3323 MB/s	3432 MB/s	20115 MB/s	< 3 %
OpenCL pinned	5628 MB/s	6096 MB/s	20115 MB/s	< 3 %

Tabelle 4.1: Bandbreitenvergleich des NVIDIA-Quadro Systems

Analyse

Durch Analyse der Testergebnisse aus Tabelle 4.1 erkennt man keine großen Unterschiede zwischen CUDA und OpenCL. Einzig beim Kopieren auf der GPU gibt es einen deutlichen

Einstellung	CPU zu GPU	GPU zu CPU	auf der GPU	Abweichung
OpenCL	2529 MB/s	2740 MB/s	45775 MB/s	< 3 %
OpenCL pinned	4320 MB/s	5205 MB/s	45775 MB/s	< 3 %

Tabelle 4.2: Bandbreitenvergleich des AMD-Fire-Pro Systems mit OpenCL

Unterschied von 6 GB/s. Die OpenCL Implementierung von AMD liegt, wie in Tabelle 4.2 zu sehen, bei Kopiervorgängen nochmal ein wenig darunter, und dies trotz der theoretisch höheren Durchsatzrate der AMD-Fire-Pro. Beim Kopieren auf der GPU kann sie das NVIDA-Quadro System jedoch weit hinter sich lassen. Dies deutet auf die etwas vernachlässigende Behandlung von OpenCL auf der Seite von NVIDIA hin. Die Verwendung von „pinned memory“ zeigt über die gesamten Untersuchungen einen enormen Geschwindigkeitszuwachs gegenüber dem Standardverfahren und bietet sich daher für zeitkritische Übertragungen geradezu an. Denn trotz der relativ schnellen Übertragung, ist dieser Faktor für viele Berechnungen auf der GPU limitierend und sollte daher bei einer Implementierung immer mit im Vordergrund stehen. Durch die gewonnen Werte lässt sich ausrechnen wie viel Zeit benötigt wird, um eine bestimmte Menge an Daten auf die Grafikkarte zu kopieren.

4.3 Speicherverwaltung

Wie schon in Abschnitt 3.3 gezeigt, ist der Einfluss der Speicherverwaltung auf die Echtzeitfähigkeit ein entscheidender Faktor. In diesem Abschnitt wird untersucht wie sich das Verhalten in Konkurrenzsituationen zwischen verschiedenen Kontexten auswirkt, und ob man Aussagen über die Sicherheit des Speichers machen kann. Denn valide Daten sind ein entscheidender Faktor bei der Zusicherung zur Echtzeit. Im weiteren Verlauf wird die genaue Organisation des Speichers untersucht, um den Ressourcenverbrauch beim allozieren genau einschätzen zu können.

4.3.1 Konkurrierende Speicherallokation

Ziel dieser Untersuchung ist es ein reales Anwendungsverhalten zu simulieren. Es werden Konkurrenzsituationen zwischen CUDA, OpenCL und OpenGL konstruiert, die jeweils viel Speicher benötigen, um die Möglichkeiten der gegenseitigen Beeinflussung zu bestimmen. Wenn Speicher verdrängt werden kann, ändert sich die Zugriffszeit drastisch und hat direkt Einfluss auf das Echtzeitverhalten.

Vorgehensweise

Alloziere jeweils mit einem CUDA-, OpenCL- und OpenGL-Programm den gesamten Grafikspeicher. Dann wird mit den Anderen versucht, einen Bereich zu allozieren. Um mit OpenGL Speicher zu allozieren, wird ein kleines Programm gestartet, welches Texturen mit 100, 200

und 300 MB in den Speicher laden und anschließend auf einem sich drehenden Würfel darstellt. Es wird, um den gesamten Speicher zu füllen, mehrfach gestartet.

4.3.2 Auswertung

Konkurrenz zwichen CUDA und OpenCL Es wird mit CUDA der verfügbare Speicher auf der Grafikkarte alloziert, ein weiteres Allozieren von OpenCL war in diesem Fall nicht möglich. OpenCL kann also den mit CUDA allozierten Speicher nicht verdrängen.

Zuerst wird mit OpenCL der gesamte Speicher alloziert. Mit CUDA konnte daraufhin kein weiterer Speicher mehr angefordert werden. CUDA ist ebenso nicht in der Lage, OpenCL dazu zu veranlassen, Speicher freizugeben oder auszulagern.

Konkurrenz zwichen CUDA / OpenCL und OpenGL Das OpenGL Programm kann zwar nach Allokation von CUDA bzw, OpenCL noch gestartet werden, es arbeitet jedoch mit einer deutlich verringerten Performance. Die Daten werden dadurch nicht zum Auslagern gebracht. Verifiziert wurde dies durch eine anschließende Berechnung mit den Daten und einem Vergleich auf dem Host nach dem zurückkopieren. Umgekehrt sind CUDA und OpenCL nicht in der Lage, nachdem das OpenGL-Programm den Speicher belegt hat, eigenen Speicher zu allozieren.

Konkurrenz zwichen OpenGL und OpenGL Das OpenGL Testprogramm wurde mehrfach gestartet. Sobald der Speicher auf der Grafikkarte beinahe voll war, wurden Daten in den Ram des Hosts ausgelagert. Dadurch ging die Performance der einzelnen Anwendungen zurück, ausgeführt wurden sie dennoch. OpenGL ist demnach in der Lage, andere OpenGL-Anwendungen zum Auslagern zu bewegen.

Die verschiedenen Möglichkeiten der Verdrängung von Speicher sind Tabelle 4.3 noch einmal zusammengefasst.

zuerst alloziert	nächste Anforderung		
	CUDA	OpenCL	OpenGL
CUDA	Nicht möglich	Nicht möglich	Nicht möglich
OpenCL	Nicht möglich	Nicht möglich	Nicht möglich
OpenGL	Nicht möglich	Nicht möglich	möglich

Tabelle 4.3: Speicher Verdrängung

Demnach ist nur OpenGL in der Lage andere OpenGL Anwendungen zu verdrängen. Für CUDA und OpenCL kann also eine Sicherheit in Bezug auf einmal allozierten Speicher gegeben werden.

4.3.3 Speicherschutz

Der Speicherschutz ist ein weiterer wichtiger Aspekt für Anwendungen auf der GPU vor allem in Bezug auf die Sicherheit einer Anwendung. Denn wie schon in Abschnitt 3.3 erwähnt sind valide Daten essentiell für ein Echtzeitsystem.

CUDA

Sobald man innerhalb eines Kernels von einer Speicheradresse außerhalb des zugewiesenen Speichers lesen oder schreiben möchte, wird die gesamte Anwendung des Host beendet, siehe dazu auch [NVI11c] „segmentation fault and application termination“.

OpenCL

OpenCL auf NVIDIA-Quadro: man kann über eine Änderung der Zeigeradresse auf anderen Speicher im eigenen Bereich zugreifen, sobald man darüber hinaus geht, wird das Programm beendet oder das gesamte System stürzt. Also ähnliches Verhalten wie bei CUDA.

OpenCL auf AMD-Fire-Pro: hier scheint es keinen Speicherschutz zu geben, man kann problemlos eine Zeigeradresse erweitern und somit auf andere Speicheradressen lesend, sowie schreibend zugreifen. Ohne jegliche Rückmeldung über Fehlercodes oder von der Laufzeitumgebung.

Ergebnisse

Bei NVIDIA wird der Kernel bzw. die gesamte Anwendung direkt beendet, sobald sie auf Speicher außerhalb ihres Bereiches zugreift. Der Speicherschutz ist nach [BR10] bei CUDA in Hardware umgesetzt. Bei OpenCL stürzt meist sogar das gesamte System ab. Jede Speicherzugriffsverletzung muss unbedingt vermieden werden, um ein sicheres System zu haben. Bei AMD scheint es diese Kontrolle nicht zu geben, zumindest konnte durch mehrfaches Schreiben und Lesen außerhalb des eigenen Bereiches keine offensichtliche Anomalie hervorgerufen werden. Für das Echtzeitverhalten kann daraus geschlossen werden, dass der Speicherschutz von NVIDIA ein wichtiger Beitrag ist um Sicherheit zu garantieren. Bei AMD scheint diese Kontrolle zu fehlen und die Kernel müssen daher mit besonderer Vorsicht programmiert werden. Für Fremdanwendungen ist dies schwer möglich und muss daher berücksichtigt werden.

4.3.4 Ermittlung der Blockgröße und dessen Eigenschaften

Ziele der Untersuchung

Um Aussagen über den Speicher treffen zu können, ist es sehr wichtig seine interne Organisation zu kennen. Bei dieser Untersuchung steht die einzelne Blöckgröße im Blickpunkt, also die kleinste Einheit des Speichers die sich bei einer Allokation egal welcher Granularität immer ändert. Im Idealfall ist es genau die Größe die alloziert wird, vom Verwaltungsaufwand ist dies bei sehr vielen kleinen Einheiten, im Byte Bereich, aber sehr aufwendig. Hier schließt sich direkt die zweite Fragestellung zur Organisation an. Nämlich ob kleine Elemente eine Mindestgröße an Speicher benötigen, das heißt jedes neue Objekt belegt dieselbe Größe oder kommt es zu so genanntem „Paging“. In diesem Fall benötigen kleine Objekte ebenso eine Mindestgröße an Speicher beim Allozieren. Neue Objekte können jedoch im selben Block mit abgelegt werden, es wird bei neuer Allokation also kein zusätzlicher Speicher verbraucht.

Vorgehen

Alloziere ausgehend von einem Byte bis hin zu 10 MB, verschieden große Objekte auf dem GPU Speicher. Für jeden Wert so oft, dass sich die Angabe über den freien Speicher mehrfach ändert, um eine belastbare Aussage über die Organisation treffen zu können.

CUDA Mit der in 4.3 gezeigten Abfrage lässt sich bei CUDA der gesamte und der freie Speicher auf ein Byte genau abfragen, so kann nach jedem Allozieren die Änderung direkt abgefragt und überprüft werden.

Algorithmus 4.3: Freien Speicher abfragen

```
1  size_t free_mem, total_mem;
2  cudaMemGetInfo(&free_mem, &total_mem);
3  printf("Free/Total %lu/%lu %u %%\n",free_mem, total_mem, (unsigned int)
       ((free_mem*100.0)/total_mem));
```

OpenCL Bei OpenCL gibt es keine direkte Abfrage für den freien Speicher, daher wird für die NVIDIA Karte eine OpenGL Funktion verwendet. Für die AMD Karte gibt es ebenso eine Abfrage von OpenGL, diese kann die Änderungen durch OpenCL jedoch nicht erkennen. Einzig „GPU-Z“ ist dazu in der Lage, den Speicherverbrauch von OpenCL zu auszulesen.

Ergebnisse

Die Ergebnisse haben den Paging Effekt klar aufgedeckt und sind für den Ressourcenverbrauch bei der Allokation sehr wichtig. Bei NVIDIA Grafikkarten liegt die Blöckgröße bei einem Megabyte, sowohl bei CUDA als auch bei OpenCL. Es gibt allerdings einen Unterschied in der Anzahl der Elemente die innerhalb eines Blocks abgelegt werden. Dieser Wert

liegt bei CUDA bei 2048, das bedeutet im Umkehrschluss, dass jedes Element mindestens 512 Byte belegt. Bei OpenCL hingegen kann man bis zu 4096 Objekte in einem Block allozieren, was wiederum auf eine Mindestbelegung von 256 Byte pro Einheit schlussfolgern lässt.

Bei der AMD Grafikkarte war leider ein so genaues Messen der Speicherauslastung mangels verfügbarer Abfrage nicht möglich. Mit GPU-Z konnte man die Werte zumindest annähern und erkennen, dass bei Speichereinheiten unter 32 Kilobyte immer direkt 2 MB belegt werden. Zwischen 32 Kilobyte bis hin zu 1 MB ist es genau 1 MB, bei größere Einheiten dann entsprechend mehr. Der Paging Effekt ist ebenso auszumachen, bei mehreren kleinen allozierten Einheiten blieb die Speicherbelastung bei 2 MB.

Für die weiteren Messungen gehen wir von einer Blockgröße von 1 MB aus.

4.3.5 Speicherseiten

Mit den in 4.3.4 gewonnen Ergebnissen kann in weiteren Untersuchungen, die genaue Organisation der Blöcke bestimmt werden. Damit der genaue Ressourcenverbrauch bestimmt werden kann. Im Detail geht es um folgende Fragestellungen.

1. Wie Werden beim Allozieren die Objekte auf verschiedene Blöcke verteilt?
2. Werden freie Bereiche in Blöcken aufgefüllt, auch wenn zwischenzeitlich andere Blöcke belegt wurden?
3. Werden Blöcke zusammengefasst, um mehr freien Speicher zu bekommen, wenn nur Teile davon wieder freigegeben werden?

Vorgehen

Um diese Fragen beantworten zu können, wurden folgende Testfälle konstruiert, angepasst an die Werte der Blockgröße von einem Megabyte, aus Abschnitt 4.3.4, und dem Vorhandensein von Paging im GPU-Speicher.

1. Der Speicher wird mit 8 mal 375 KB alloziert. 2 Einheiten passen in einen Block, für den jeweils dritten gibt es nun 2 Möglichkeiten, entweder wird er auf 2 Blöcke aufgeteilt oder in einem neuen abgelegt. Bei 8 angelegten Bereichen sind für den ersten Fall 3 MB belegt und im zweiten sind es 4. Zur besseren Verifikation wurde der Test auch mit 16, 32 und 64 angelegten Bereichen durchgeführt.
2. Der Speicher wird mit 2 mal 333 KB alloziert, diese passen in einen Block und es gibt noch Platz für einen weiteren. Danach werden 900 KB alloziert, um sicher einen neuen Block zu belegen. Als letztes wird wieder ein Bereich mit 333 KB alloziert, für diesen gibt es nun 2 Möglichkeiten beim Anlegen. Entweder wird er in einem neuen Block angelegt und somit ist ein weiteres MB belegt, oder es wird der erste Block aufgefüllt und es bleibt bei der Belegung von zwei MB. Bei diesem Test wurden zur Variation

dazwischen mehrere neue Bereiche angelegt, um die „Entfernung“ zum ersten Block zu erhöhen.

3. Der Speicher wird mehrmals mit 512 KB Objekten alloziert, genau die Hälfte der in Untersuchung 4.3.4 festgestellten Blockgröße von 1 MB. Es sollten sich in jedem Block also zwei Einheiten befinden. Nun wird jede Zweite wieder freigegeben. Erhöht sich dadurch der freie Speicher, so werden Blöcke zusammengefasst. Wenn nicht muss untersucht werden ob die Lücken wieder aufgefüllt werden können. Dazu wird genau die Hälfte an 512 KB Stückchen wieder angelegt und überprüft ob sich die Größe des freien Speichers verändert.

Ergebnisse

Aufgrund der mangelhaften Möglichkeiten der Abfrage nach dem freien Speicher auf der AMD-Karte konnten darauf nicht alle Tests durchgeführt werden und die Ergebnisse sind mit der nötigen Skepsis zu betrachten. Da gerade bei kleinen Einheiten die Messgenauigkeit entscheidend ist.

NVIDIA-Quadro System mit CUDA und OpenCL Da sich die Werte bei der NVIDIA Karte für CUDA und OpenCL nicht unterscheiden, werden die Ergebnisse zusammengefasst.

1. Nach der Allokation von 8 mal 375 KB sind wie in Abbildung 4.1 zu sehen 4 Blöcke belegt, es wird also nicht über Blockgrenzen hinweg verteilt wenn der Bereich nicht am Anfang des Blockes beginnt, wie es bei großen Einheiten (größer als 1 MB) der Fall ist. Das Verhalten konnte mit dem vielfachen an Objekten verifiziert werden, wie beim Vorgehen beschrieben.

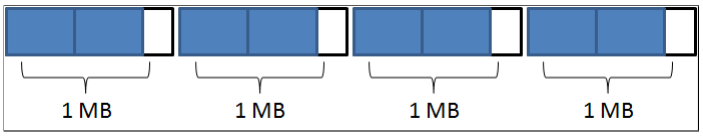

Abbildung 4.1: Verteilung der Objekte bei NVIDIA

2. Die Lücken werden immer aufgefüllt, auch wenn dazwischen mehrere andere Bereiche alloziert worden sind.
3. Nach der Freigabe jeder zweiten Einheit veränderte sich der zurückgegebene Wert für den freien Speicher nicht. Die vorhandenen Lücken konnten vollständig wieder gefüllt werden, ohne zusätzlichen Speicher zu verwenden. Wurde auch nur eine Einheit mehr als zuvor alloziert, so verringerte sich der freie Speicher um 1 MB.

AMD-Fire-Pro System mit OpenCL

1. Auf der AMD Karte sind nach der Allokation, wie in Abbildung 4.2 dargestellt nur 3 MB belegt. Bei der Variation mit 16 angelegten Blöcken sind es entsprechend 6 MB, bei 32 Einheiten werden 12MB als belegt gezeigt und bei 64 angelegten Objekten 24 MB. Mit der Besonderheit, dass beim ersten Allozieren direkt 2 MB belegt werden. Die Blöcke werden demnach bei AMD über die Grenzen hinweg befüllt.

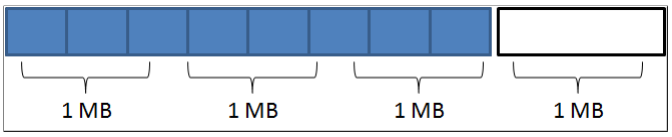

Abbildung 4.2: Verteilung der Objekte bei AMD

2. Nach diesem Test kann, so gut es möglich ist, auch von einem Ablegen in angefangenen Blöcken ausgegangen werden, selbst wenn dazwischen andere Blöcke ganz gefüllt werden.
3. Nach der Freigabe konnte keine Änderung des freien Speicher festgestellt werden, die Blöcke werden also auch bei der AMD Karte nicht zusammengefasst. Ein erneutes Allozieren war jedoch problemlos möglich und der vorhandene Speicher konnte vollständig genutzt werden.

Zusammenfassung

Die gewonnen Erkenntnisse zur Organisation der Speicherseiten ist für den Ressourcenverbrauch beim allozieren mit zu berechnen und für Zusicherungen an den Speicher zu verwenden. Gerade die Verteilung auf die einzelnen Blöcke muss berücksichtigt werden, wenn eine Anwendung viele kleinere Speicherbereiche alloziert. Die Tatsache, dass Blöcke nach dem freigeben nicht zusammengefasst werden, ist ein Umstand den man so hinnehmen muss. Wenn man sie selber erzeugt hat, kennt man die Größe und kann den Bereich trotzdem nutzen.

4.3.6 Speicherfragmentierung

Da bei den Tests in Abschnitt 4.3.4 festgestellt wurde, dass sowohl bei NVIDIA als auch bei AMD, Blöcke die nur teilweise befüllt sind, nicht zusammengefasst werden, stellt sich die Frage nach der Speicherfragmentierung. Diese tritt unweigerlich auf, wenn von einem Host-Programm mehrfach Speicher alloziert und wieder freigegeben wird. Je kleiner die Einheiten sind desto stärker der Effekt. Im Fall wie Abschnitt 4.3.5 Untersuchung drei, wenn jeder zweite Bereich freigeben wird, ist theoretisch die Hälfte des Speichers frei, er kann aber aufgrund der Fragmentierung für größere Bereiche eventuell nicht genutzt werden. Es gilt also zu bestimmen, ob die wieder freigegebenen ganzen Blöcke zusammengefasst werden. Denn bei teilen eines Blocks ist dies nach Untersuchung 4.3.5 nicht möglich. Man kann sich

diese Fragestellung auf einen voll belegten Saal übertragen. Stehen einige Personen zufällig verteilt auf und verlassen den Sall, kann dann eine Gruppe derselben Anzahl zusammen sitzen, da die anderen Menschen sich umsetzen, oder nicht?

Vorgehen

Der Speicher wird vollständig mit 1 MB Objekten belegt, dies entspricht der in 4.3.4 bestimmten Blockgröße. Durch löschen jedes zweiten Elements wird eine Fragmentierung auf dem Grafikkartenspeicher erzeugt und es sollte die Hälfte des Globalen Speichers frei sein. Im nächsten Schritt wird versucht ein 200 MB großes Objekt respektive mehrere 10 MB große Objekte anzulegen und die Daten zu verifizieren. Zur weiteren Variation wurden verschiedenste Größen verwendet, aber immer so, dass die vorhandenen Lücken kleiner waren, als die gewünschten neu anzulegenden Bereiche. Um Anomalien bei der Allokation oder eine Umorganisation des Speichers erkennen zu können wurde, zusätzlich zur Verifikaton der Daten, die Dauer mitprotokolliert. Die Vorgehensweise ist in Algorithmus 4.4 beispielhaft für CUDA dargestellt.

Algorithmus 4.4: Speicherfragmentierung bei CUDA

```
int size = 1024*1024; // 1 MB
cudaMemGetInfo( &maximal, &gesamt );
int x = ((maximal/1024)/1024) //auf MB bringen
for(int i=1; i<=x; i++){
  QueryPerformanceCounter(&start_ticks);
  cudaMalloc( (void**)&dev_ptr[i], size ) ;
  QueryPerformanceCounter(&end_ticks);
  zeit[i] = ((double)(end_ticks.QuadPart-
      start_ticks.QuadPart)/(double)ticksPerSecond.QuadPart);
}
//jeden zweiten Block freigeben
for (int i = 1; i<=x; i +=2){
  cudaFree(dev_ptr[i]);
}
x = x/10; // entsprechend der neuen Objektgröße teilen
size=1024*1024*10; //neue Größe
for(int i=1; i<=x; i++){
  QueryPerformanceCounter(&start_ticks2);
  //neue Objekte anlegen
  cudaMalloc( (void**)&dev_ptr2[i], size ) ;
  QueryPerformanceCounter(&end_ticks2);
  //Zeit der Übertragung speichern
  zeit2[i] = ((double)(end_ticks2.QuadPart-
      start_ticks2.QuadPart)/(double)ticksPerSecond.QuadPart);
}
//Daten zurück zum Host kopieren und verifizieren
```

Ergebnisse

CUDA Auf dem NVIDIA-Quadro System kam es zu Anomalien bei der Speicherallokation, den gesamten Speicher mit 1 MB Stückchen zu füllen war nicht möglich, auch nicht mit 10 MB Einheiten. Wie sich herausstellte, kam der Fehler immer an derselben „Stelle“ und es spricht alles für einen Hardwaredefekt. Daher wurde dieser Test auf dem NVIDA-NVS und dem NVIDIA-GTX Testsystem durchgeführt.

Bei den NVIDIA Karten war es in jeglicher Konstellation von freien Bereichen immer möglich den gesamten Speicher zu nutzen und auch sehr große Bereiche zu allozieren. Als Besonderheit ist aufgefallen, dass der zweite „cudaMalloc“ Befehl in Algorithmus 4.4 Zeile 19 allerdings deutlich mehr Zeit in Anspruch nahm, was sich auf eine Umorganisation bzw. einen erhöhten Verwaltungsaufwand auf der Grafikkarte zurückführen lässt. Der zeitliche Mehraufwand war teilweise enorm, so dauert eine Allokation normalerweise 3 ms. Bei starker Fragmentierung stieg dieser Wert bis auf 0,2 s an.

OpenCL Bei OpenCL konnte diese Umorganisation nicht beobachtet werden, man kann nur die vorhandenen Lücken allozieren und keine größeren Bereiche. Hat man z.B. nach der Freigabe 5 MB Lücken und möchte mehrmals 10 MB allozieren, wird der Befehl zwar ohne Fehler abgesetzt, möchte man aber auf die Daten zugreifen kommt es zu Speicherfehlern.

Zusammenfassung

Bei CUDA kann durch die Fragmentierung die Zeit des Allozierens erhöht sein. Dies beeinflusst das Echtzeitverhalten und muss daher berücksichtigt werden.

Bei OpenCL ist es nicht mehr möglich bei starker Fragmentierung große Bereiche zu allozieren. Es müsste die gesamte Anwendung neu gestartet werden.

4.3.7 Größtmögliche zu allozierende Einheit

Diese Untersuchung hat das Ziel die Möglichkeiten, Art und Weise der Speicherallokation weiter zu untersuchen. Die Frage nach der größten Einheit die im GPU-Speicher alloziert werden kann stellt sich unmittelbar durch eine vermeintliche Restriktion von OpenCL. Mit der Funktion „CL_DEVICE_MAX_MEM_ALLOC_SIZE“ kann dieser Wert direkt abgefragt werden. Für CUDA gibt es keine Entsprechung und wird daher direkt eruiert. Wenn man einen sehr großen Bereich allozieren kann, ergibt sich die Möglichkeit diesen selber zu verwalten und selbst in kleinere Einheiten zu unterteilen. Daraus ergibt sicher der Vorteil einer größeren Kontrolle über die Aufteilung und die Sicherstellung einer bestimmten Größe an Speicher, für die gesamte Laufzeit eines Programms oder so lange, wie gewünscht.

CUDA

Bei CUDA ist die Größe nur abhängig vom gesamten Speicher, denn es kann der komplette freie Bereich auf einmal alloziert werden.

Bei dem NVIDIA-NVS System liegt dieser Wert bei 422 MB, der Rest wird von Windows verwendet, wie mit GPU-Z überprüft wird. Beim NVIDIA-Quadro System bei 937 MB, auch hier also nicht die gesamten 1024 MB da Windows den Rest verwendet. Mit dem Befehl „cuMemGetInfo(&frei, &gesamt)“ kann die Größe des gesamten und des freien Speichers abgefragt werden. Zwischen der Abfrage des freien Speichers und dem allozieren kann von einem konkurrierenden Programm, z.B. das Betriebssystem, Speicher alloziert worden sein. Um diesem Verhalten zu begegnen kann mit dem in 4.5 angegebenen Algorithmus der maximal mögliche Bereich im GPU Speicher alloziert werden.

Algorithmus 4.5: CUDA maximale Speicherallokation

```
1  const size_t MB = 1024*1024;
2  int *dev_Speicher;
3  size_t maximal, gesamt;
4  cudaMemGetInfo( &maximal, &gesamt );
5  char fail = 0;
6  while( cudaMalloc( (void**)&dev_Speicher, maximal ) != cudaSuccess )
7  {
8     maximal -= MB;
9     if( maximal < MB )
10    {
11       fail = 1;
12       break;
13    }
14 }
```

OpenCL

Bei OpenCL gibt es, wie schon erwähnt, mit dem Befehl „CL_DEVICE_MAX_MEM_ALLOC_SIZE“ die Möglichkeit, genau die gewünschte Größe direkt abzufragen. Nach eigenen Messungen stellte sich dieser Wert allerdings als nicht sehr zuverlässig heraus, die tatsächlichen Werte sind Tabelle 4.4 zu entnehmen. Die realen Werte bei OpenCL wurden durch sukzessive Erhöhung der Größe beim allozieren, siehe Abschnitt 2.5.2, und Überprüfen auf valide Daten ermittelt.

Testsystem	Wert der Abfrage	tatsächlicher Wert
NVIDIA-NVS	128 MB	gesamter Bereich (422 MB)
NVIDIA-Quadro	240 Mb	gesamter Bereich (937 MB)
AMD-Fire-Pro	256 MB	512 MB

Tabelle 4.4: OpenCL maximale Speicherallokation

Ergebnisse

Sowohl bei CUDA als auch bei OpenCL lassen sich sehr große Bereiche, bei den NVIDIA Karten sogar bis hin zum gesamten freien Speicher, direkt allozieren. Somit kann man, wenn dies am Anfang einer Anwendung direkt geschieht, über die gewünschte Laufzeit eine bestimmte Menge an Speicher verwenden und selber verwalten, oder den Start der Anwendung verschieben, bis die gewünschte Menge verfügbar ist. Damit kann die Kontrolle über die Speicherverwaltung auf Seite des Benutzers gestellt werden und richtet sich nicht mehr zur Gänze nach den Vorgaben der Hersteller.

4.4 Scheduling

Das Scheduling, die Reihenfolge wie Befehle abgearbeitet werden, und die Möglichkeiten der Einflussnahme darauf sind wichtige Faktoren um eine Aussage über die Echtzeitfähigkeit treffen zu können. Eng damit verbunden ist das Vorhandensein und die Analyse der Eigenschaften einer Queue (Warteschlange) für die Befehle an die Grafikkarte.

4.4.1 Analyse der Speicherqueue

Motivation für die Untersuchung

Bei den Untersuchungen in Abschnitt 4.3 zum Speicher konnte bei sehr vielen asynchronen Kopiervorgängen manchmal eine Verzögerung des Host Programms festgestellt werden. Zum besseren Verständnis der Funktionsweise von Kopiervorgängen wird daher in dieser Untersuchung die Betrachtungsweise auf die Speicherqueue, eine Warteschlange für die Speicherbefehle an die GPU, verlagert. Hier schließen sich folgende Fragestellungen an.

- Wie viele Befehle kann man ohne Verzögerung des Host-Programms an die Grafikkarte schicken?
- Wie lange, respektive auf wie viele Befehle muss gewartet werden, bis erneut eine Anfrage zum Kopieren an die GPU geschickt werden kann?

Vorgehen

Da bei synchronem Kopieren von einem Programm immer auf die Bearbeitung des Befehls gewartet wird, erhält die Queue nicht mehr als einen Befehl. Daher genügt es die Tests nur für den asynchronen Fall durchzuführen. Die Untersuchung wird wie in Algorithmus 4.6 dargestellt durchgeführt. Zur Variation und dem Ausschluss möglicher Faktoren wird das Programm um Threads erweitert, die parallel Anfragen abschicken. Bei CUDA wird die Verwendung von Streams als weitere Modifikation eingeführt.

Algorithmus 4.6: Beispielcode zur Speicherqueue

```
1 //Variablen initialisieren
2 for(int i=0; i<x; i++) //x entsprechend groß gewählt
3 {
4    QueryPerformanceCounter(&start_ticks);
5  cudaMemcpyAsync( dev_b, b, größe, cudaMemcpyHostToDevice, stream0) ;
6  QueryPerformanceCounter(&end_ticks);
7  zeit = ((double)(end_ticks.QuadPart-
       start_ticks.QuadPart)/(double)ticksPerSecond.QuadPart);
8  time0[i] = zeit;
9  logfileS[i]=start_ticks;
10 logfileE[i]=end_ticks;
11 }
```

In einer Schleife werden mehrere Tausend Kopiervorgänge gestartet und jedes Mal die Zeit mitprotokolliert. Das Absetzen des Befehls benötigt ca. 1 ms, die eigentliche Übertragung ist so gewählt, dass sie ca. 0,1 s dauert, also konnten in der Zeit einer Übertragung mehrere Tausend Befehle abgesetzt werden. Durch die gemessene Wartezeit bei einer Verzögerung, konnte die Anzahl der abgearbeiteten Befehle ermittelt werden. Um Auswirkungen der zu übertragenden Daten auszuschließen, werden die Tests mit verschiedensten Größen durchgeführt, siehe Tabelle 4.5. Wobei sich dann jeweils die entsprechende Dauer der Übertragung ändert und die Auswertung angepasst werden muss. Pro Test wurden 10 Durchgänge gestartet, um etwaige Störungen erkennen zu können.

Analyse

CUDA Beim ersten Test mit dem Algorithmus 4.6 konnten die Werte der Tabelle 4.5 evaluiert werden.

Speichergröße	Größe der Queue	Sprunggröße
50 MB	682	136
100 MB	682	136
200 MB	682	136
400 MB	682	136
800 MB	682	136

Tabelle 4.5: Speicher Queue

Das Programm kann also zunächst 682 Kopierbefehle absetzen bis es blockiert wird. Erst nach der Abarbeitung von 136 Kopiervorgängen fährt das Programm fort und die Queue kann, wie in Abbildung 4.3 zu sehen, dieses Mal mit genau 136 Befehlen erneut befüllt werden. Dieses Verhalten wiederholt sich bis zum Ende des Programms.

Bei 2 Threads mit Streams gibt es leichte Abweichungen, allerdings auch hier wieder konsistent über 10 Durchläufe und Variation in der Größe der zu übertragenden Daten. Gemessen wurde, wie in Tabelle 4.6 aufgezeigt, für die Queuegröße der Wert 664 und für

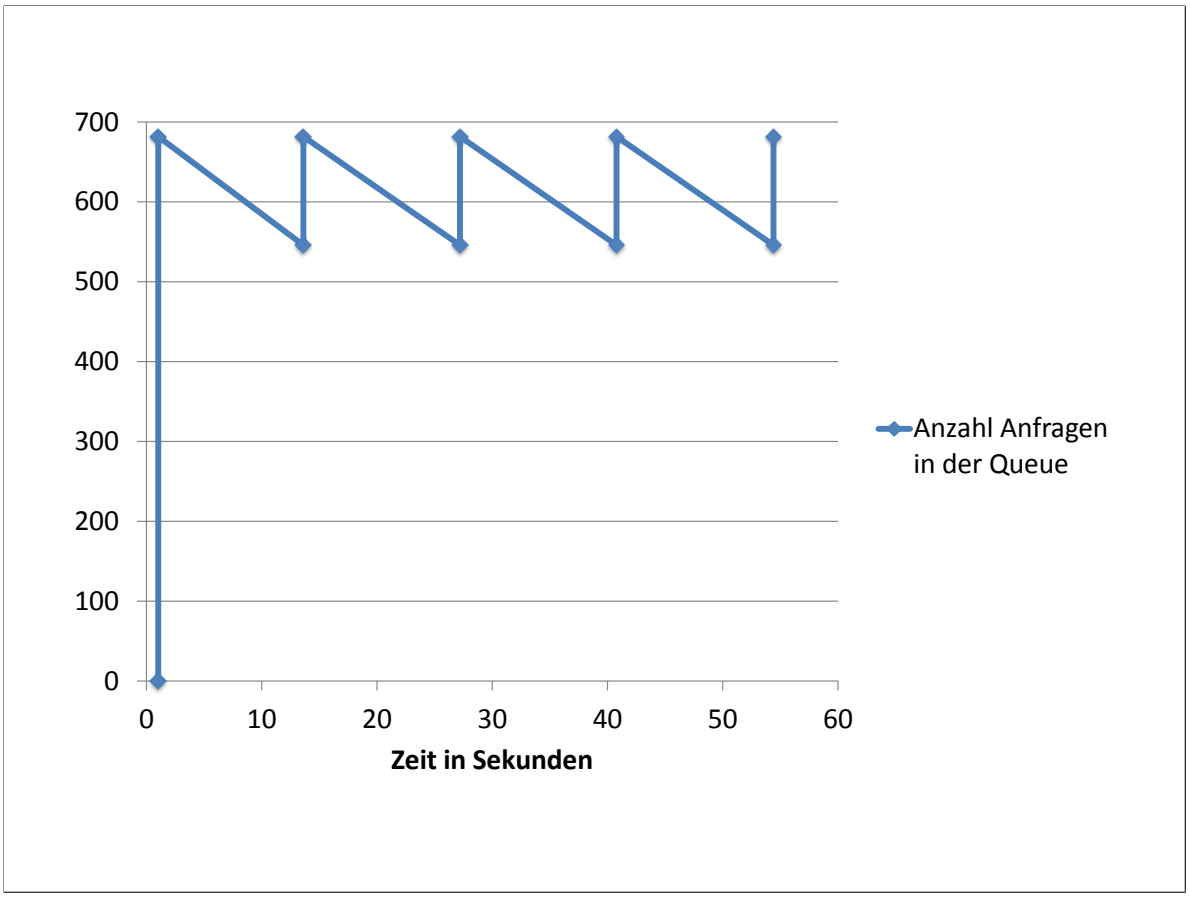

Abbildung 4.3: Speicherqueue bei CUDA

die Sprunggröße 132. In Abbildung 4.4 ist ein Auschnitt der Abarbeitung zu sehen, wie die Queue über die Zeit befüllt wird. Durch die Erzeugung und Zuordnung der Streams scheint also ein gewisser Overhead zu entstehen, der entsprechend kompensiert wird. Bei einer Erweiterung auf 4 Threads mit jeweils eigenem Stream werden dieselben Werte gemessen, also nochmals 664 für die Queuegröße und 132 für die Sprünge. Ebenso bei einem einzigen Aufruf mit einem Stream. Die Größe hängt also nicht von der Anzahl der Threads ab, sondern ob Streams verwendet werden oder nicht.

Durch die gemessenen Ergebnisse lässt sich zeigen, dass bei der Verwendung des asynchronen Kopierens die Größe der Queue eine wichtige Rolle spielt und deren Größe bei der Implementierung berücksichtigt werden muss. Dies ist wichtig um sicher gehen zu können, dass auch alle erforderlichen Daten im Hauptspeicher der GPU zum Zeitpunkt der Berechnung vorliegen. Ebenso um ein Blockieren des Host-Programms zu verhindern. Insbesondere bei mehreren Threads wird dies eine Rolle spielen, denn bei einer Anwendung allein wird die Grenze der Queue nicht so schnell erreicht werden.

Speichergröße	Größe der Queue	Sprunggröße	Anzahl Threads
50 MB	664	132	1
100 MB	664	132	1
200 MB	664	132	1
50 MB	664	132	2
100 MB	664	132	2
200 MB	664	132	2
50 MB	664	132	4
100 MB	664	132	4
200 MB	664	132	4

Tabelle 4.6: Speicher Queue mit Threads und Streams

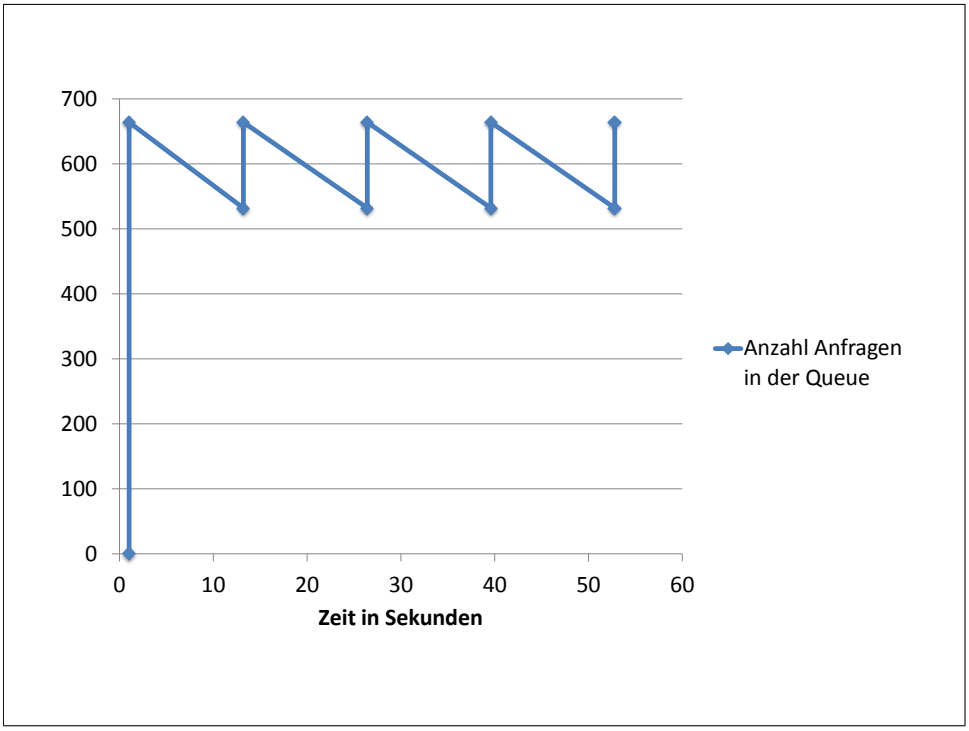

Abbildung 4.4: Speicherqueue bei CUDA mit Streams

OpenCL Bei OpenCL ist das Kopieren normalerweise asynchron und konnte daher direkt untersucht werden. Es kam allerdings zu keiner Zeit zu einer außergewöhnlichen Verzögerung. Die Queue wird also vermutlich im Treiber auf Seite des Host verwaltet und ist nur durch dessen Ressourcen beschränkt.

4.4.2 Scheduling

Motivation für die Untersuchung

Das Scheduling kontrolliert die Reihenfolge der Abarbeitung wenn mehrere Kernel an die GPU geschickt werden. Da ein Kernel, unabhängig von der Laufzeit, nicht unterbrochen wird [NVI11c, MGM+11], sondern immer komplett durchläuft, ist es interessant zu wissen ob die Laufzeit Einfluss beim Scheduling hat oder nicht. Die Evaluierung des GPU-Scheduling ist auch besonders wichtig im Hinblick auf Konkurrenzsituationen von verschiedenen Applikationen, die die Grafikkarte gemeinsam verwenden. Es gibt keine Interrupts für die Kernel auf der GPU, die einzige Möglichkeit einer Unterbrechung ist eine komplette Rücksetzung des Treibers, dabei gehen jedoch alle Daten auf der GPU verloren. Wenn es die primäre Grafikkarte ist, so würde auch die Anzeige schwarz werden.

Vorgehen

Zur Verifikation der Arbeitsweise des Scheduling wurden verschiedene Tests durchgeführt, die im Folgenden kurz vorgestellt werden. Zur Simulation von verschiedenen Anfragen wurden zunächst 2 Threads verwendet. Diese Vorgehensweise wurde dann um 2 getrennte Programme erweitert, um Abhängigkeiten zwischen den Threads ausschließen zu können. Die Zuordnung der Reihenfolge wurde über die mitprotokollierten Zeitstempel realisiert. Die allgemeine Variation ist wie in Tabelle 4.7 angegeben.

Test	Thread0	Thread1
1	add	add
2	add,add	add
3	addi (M = 3)	add
4	addi (M = 10)	add
5	addi (M = 20)	add

add – einfacher addKernel zu sehen in Algorithmus 4.7 Zeile 39 ff.

addi – Kernel mit deutlich erhöhter Laufzeit, entsprechend dem Schleifenzähler M, siehe Algorithmus 4.7 Zeile 48 ff.

Tabelle 4.7: Varianz

Diese Tests werden in der sechsten Untersuchung mit Synchronisierung erweitert, um festzustellen ob damit eine Einflussnahme auf den Ablauf möglich ist.

CUDA Bei CUDA wurde die Varianz noch durch Streams erweitert, Thread 0 war dann Stream 0 zugeordnet und Thread 1 analog dazu Stream 1.

Algorithmus 4.7: Scheduling CUDA

```
1 int M = 10; //Schleifenzähler für den Kernel addi, wird entsprechend dem Test
      initialisiert
```

```
 2 omp_set_num_threads(2); //Anzahl der Threads auf 2 setzen
 3 #pragma omp parallel for //die folgende Schleife in Threads aufteilen
 4 for(int thread_id=0; thread_id<=1; thread_id++) {
 5  for(int i=0; i<x; i++)
 6  {
 7  unsigned int cpu_thread_id = thread_id;//die aktuelle Thread ID abfragen
 8    //und entsprechend der ID den ersten oder zweiten
 9    //Bereich ausführen
10    if (cpu_thread_id==0) {//Thread 0
11      //Zeitmessung am Anfang
12      QueryPerformanceCounter(&start_ticks);
13      //Kernel starten
14      addi<<<64,64,0,stream0>>>( dev_a, dev_b, dev_c );
15      //addi<<<64,64,0,stream0>>>( dev_a, dev_b, dev_c );
16      //Zeitmessung am Ende
17      QueryPerformanceCounter(&end_ticks);
18      zeit = ((double)(end_ticks.QuadPart-
            start_ticks.QuadPart)/(double)ticksPerSecond.QuadPart);
19      time0[i]=zeit;
20      logfile0[i]=end_ticks;
21    }
22    else if (cpu_thread_id==1) {//Thread 1
23      //Zeitmessung am Anfang
24      QueryPerformanceCounter(&start_ticks2);
25      //Kernel starten
26      add<<<64,64,0,stream1>>>( dev_a2, dev_b2, dev_d2 );
27      //Zeitmessung am Ende
28      QueryPerformanceCounter(&end_ticks2);
29      zeit2 = ((double)(end_ticks2.QuadPart-
            start_ticks2.QuadPart)/(double)ticksPerSecond2.QuadPart);
30      time1[i] = zeit2;
31      logfile1[i]=end_ticks2;
32    }
33  }
34 }
35 //Ein einfacher Kernel mit kurzer Laufzeit
36 __global__ void add(const int *a,const int *b, int *c ) {
37    int tid = threadIdx.x + blockIdx.x * blockDim.x;
38    if (tid < N) {
39       c[tid] = a[tid] + b[tid];
40       tid += blockDim.x * gridDim.x;
41    }
42 }
43 //Ein Kernel mit Schleife und langer Laufzeit
44 //mit der Schleife kann die Laufzeit einfach angepasst werden
45 __global__ void addi(const int *a,const int *b, int *c ) {
46    int tid = threadIdx.x + blockIdx.x * blockDim.x;
47    if (tid < N) {
48    for (int i = 0; i < M; i++){
49      c[tid] = (2* a[tid]) + b[tid] * (a[tid] + b[tid]);
50      c[tid] = (2* a[tid]) + b[tid] + (a[tid] + b[tid]);
51      c[tid] = (3* a[tid]) * b[tid] / (a[tid] + b[tid]);
52    }
53    c[tid] = a[tid] + b[tid];
54       tid += blockDim.x * gridDim.x;
```

```
55    }
56 }
```

OpenCL Bei OpenCL werden die Tests noch um verschiedene Queues und verschiedene Kontexte in den jeweiligen Threads ergänzt. Thread 0 arbeitet dann auf der ersten Queue und Thread 1 auf Queue 2. Analog dazu wird es bei 2 Kontexten durchgeführt, also Thread 0 dem ersten Kontext zugewiesen und Thread 1 zu Kontext 2.

Algorithmus 4.8: Scheduling OpenCL

```
 1 //Threaderstellung analog zu CUDA
 2 int M = 10; //Schleifenzähler für den Kernel addi, wird entsprechend dem Test
       initialisiert
 3 omp_set_num_threads(2);
 4 #pragma omp parallel for
 5 for(int thread_id=0; thread_id<=1; thread_id++) {
 6   for(int i=0; i<x; i++)
 7   {
 8     unsigned int cpu_thread_id = thread_id;
 9     if (cpu_thread_id==0) {
10        //Zeitmessung am Anfang
11       QueryPerformanceCounter(&start_ticks);
12       //Kernel starten
13       clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL,
             &szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);
14       //clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL,
             &szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);
15       //die Befehle an die GPU senden
16       clFlush(cqCommandQueue);
17       //Zeitmessung am Ende
18       QueryPerformanceCounter(&end_ticks);
19       zeit = ((double)(end_ticks.QuadPart-
             start_ticks.QuadPart)/(double)ticksPerSecond.QuadPart);
20       time0[i] = zeit;
21       logfileS[i]=start_ticks;
22       logfileE[i]=end_ticks;
23     }
24     else if (cpu_thread_id==1) {
25       //Zeitmessung am Anfang
26       QueryPerformanceCounter(&start_ticks2);
27       //Kernel starten
28       clEnqueueNDRangeKernel(cqCommandQueue2, ckKernel2, 1, NULL,
             &szGlobalWorkSize2, &szLocalWorkSize2, 0, NULL, NULL);
29       //die Befehle an die GPU senden
30       clFlush(cqCommandQueue2);
31       //Zeitmessung am Ende
32       QueryPerformanceCounter(&end_ticks2);
33       zeit = ((double)(end_ticks2.QuadPart-
             start_ticks2.QuadPart)/(double)ticksPerSecond2.QuadPart);
34       time1[i] = zeit;
35       logfileS2[i]=start_ticks2;
36       logfileE2[i]=end_ticks2;
37     }
```

```
38   }
39 }
40 //Ein einfacher Kernel mit kurzer Laufzeit
41 __kernel void Add(__global const float* a, __global const float* b, __global
       float* c, int iNumElements)
42 {
43   int iGID = get_global_id(0);
44     if (iGID >= iNumElements)
45     {
46        return;
47     }
48   c[iGID] = a[iGID] + b[iGID]
49 }
50 //Kernel mit langer Laufzeit,
51 //mit der Schleife kann die Laufzeit einfach angepasst werden
52 __kernel void addi(__global const float* a, __global const float* b, __global
       float* c, int iNumElements)
53 {
54     int iGID = get_global_id(0);
55     if (iGID >= iNumElements)
56     {
57        return;
58     }
59   for (int i = 0; i < M; i++)
60   {
61     c[iGID] = a[iGID] + b[iGID] / a[iGID];
62     c[iGID] = a[iGID] + b[iGID] * a[iGID];
63     c[iGID] = a[iGID] + b[iGID] - a[iGID];
64     c[iGID] = a[iGID] + b[iGID];
65   }
66 }
```

Als letzte Variation werden verschiedene Prozesse die parallel Aufrufe an die GPU schicken untersucht. Damit kann eine eventuelle Beeinflussung, die innerhalb eines Programms stattfindet, ausgeschlossen werden.

Ergebnisse

Nach ersten Auswertungen lässt sich sagen, dass das Scheduling nach dem „First-Come First-Served“ Prinzip arbeitet, längere Kernel werden in gewisser Weise bevorzugt, da keine Unterbrechung der Ausführung stattfindet. Verzögerungen traten oft am Beginn oder am Ende auf, dies lag an der Threaderstellung und floss daher nicht in die Ergebnisse mit ein. Ein weiterer Einflussfaktor war bei CUDA die Befehls Queue, dieses Thema wird in Kapitel 4.4.3 behandelt. Die Auswirkungen konnten soweit eingeschränkt werden, um trotzdem belastbare Ergebnisse zu erhalten. Im Detail wurden folgende Resultate erzielt:

CUDA Die Reihenfolge der Auswertung ist wie in Tabelle 4.7 angegeben.

1. Test: 2 verschiedene Threads die beide jeweils einen Kernel mit derselben Laufzeit aufrufen. Hier kann ein ausgeglichenes Abarbeiten der Anfragen festgestellt werden, immer abwechselnd. Zu sehen in Abbildung 4.5

2. Test: Der erste Thread hat in der Schleife direkt 2 Aufrufe desselben Kernel. Der zweite ruft pro Durchlauf nur einmal den Kernel aus Algorithmus 4.7 Zeile 39 ff. auf. Hier kann dasselbe Verhalten wie bei Test 1 beoobachtet werden, also immer abwechselnd. Der Overhead des Host ist demnach zu gering um Auswirkungen auf die Abfolge bei der Bearbeitung zu haben, daher sei hier erneut auf Abbildung 4.5 verwiesen.

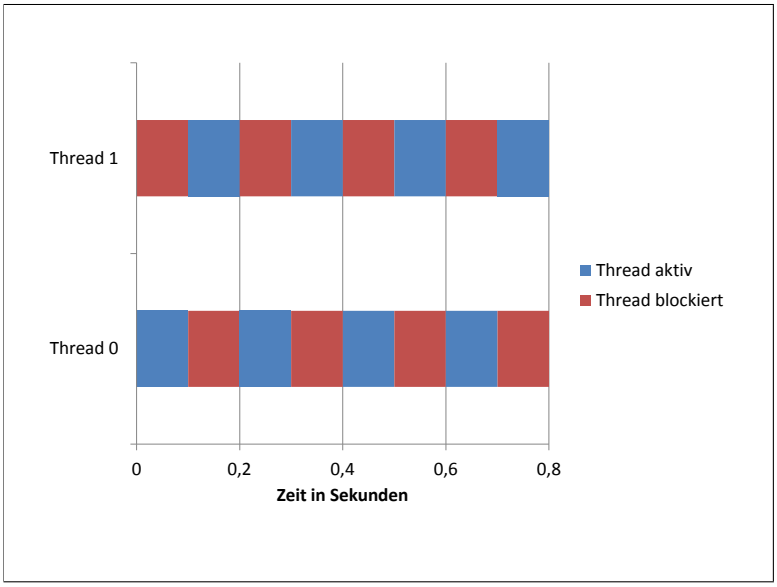

Abbildung 4.5: Scheduling zwischen 2 Threads, jeder Kernel mit einer Laufzeit von 0,1 s

3. Test: Der erste Thread ruft einen dreimal länger rechnenden Kernel auf, der zweite wieder denselben einfachen Kernel. Interessanterweise war auch bei diesem Test ein abwechselnder Ablauf festzustellen. Der schnelle Kernel wird rein von der Ausführungszeit betrachtet demnach benachteiligt, der Erste kann in derselben Zeit mehr berechnen.

4. Test: Der erste Thread ruft bei diesem Test einen zehnmal länger rechnenden Kernel auf, der zweite Thread wird wieder mit demselben Kernel gestartet. Auch hier kann keine Änderung bei der Reihenfolge festgestellt werden, der Unterschied liegt einzig in der deutlich längeren Verzögerung des schnellen Kernels. Der Ablauf ist in Abbildung 4.6 grafisch dargestellt. Der erste Thread bekommt demnach zehnmal mehr Ausführungszeit auf der Grafikkarte zugestanden als der Zweite.

5. Test: Bei diesem Test wird die Laufzeit des ersten Kernel bis auf das Zwanzigfache erhöht. Auch dieser extreme Unterschied der beiden Kernel erbrachte eine deutlich längere Verzögerung des ersten Kernel wie schon beim vierten Test. Siehe hierzu Abbildung 4.6.

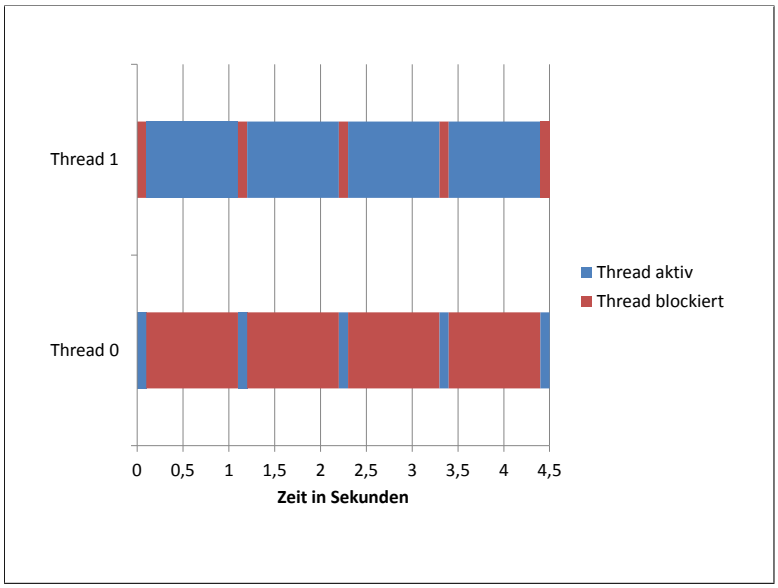

Abbildung 4.6: Scheduling zwischen 2 Threads, erster Kernel mit einer Laufzeit von 0,1 s der zweite 1 s

Eine weitere Möglichkeit der Einflussnahme auf das Scheduling ist eventuell mit Synchronisierung möglich.

6. Test mit Synchronisierung: Es werden die Tests 1 - 5 mit Synchronisierung wiederholt, dies hatte jedoch keinerlei Auswirkungen auf die Wirkungsweise des Scheduling, daher sei hier auf die Abbildungen der jeweiligen Tests ohne Synchronisierung verwiesen. Es kam wie vorher zu einer Abarbeitung nach dem „First-come first-served“ Prinzip.

Hat indessen nur ein Thread den Befehl zur Synchronisierung, so wird dieser stark verzögert, da der Andere in der Zwischenzeit seine Kernelanfragen weiter an die GPU schickt.

Erweiterung um Streams Mit Streams werden die Anfragen wie in unterschiedlichen Queues verwaltet. Daher kann von einer Auswirkung auf das Scheduling ausgegangen werden. So werden folgende Tests mit der Erweiterung um Streams durchgeführt.

Thread0	Thread1
stream 0, add	stream 1, add
stream 0, add	stream 0, add
stream 0, addi(3er Schleife)	stream 1, add
stream 0, addi(3er Schleife)	stream 0, add
stream 0, addi(10er Schleife)	stream 1, add
stream 0, addi(10er Schleife)	stream 0, add

Tabelle 4.8: CUDA Scheduling mit Streams

Mit den Streams werden erneut dieselben Ergebnisse wie ohne erzielt, sie haben also keinerlei Einfluss auf die Art des Schedulings. Daher sei an dieser Stelle auf die Auswertung der Tests ohne Streams verwiesen.

Die CUDA Streams haben dennoch einen Vorteil, sie ermöglichen unter CUDA das parallele Ausführen mehrerer Threads auf der GPU, wenn diese einzeln nicht die gesamte Karte auslasten. Möglich ist dies allerdings nur innerhalb desselben Kontextes und es hängt direkt von den Parametern beim Aufruf ab, dies lässt sich Anhand der Formel:

$$AnzahlCUDAKerne > AnzahlBloecke \cdot AnzahlThreads$$

Parameter beim Kernelaufruf

beschreiben. Es ist also eine sehr eingeschränkte Möglichkeit das Scheduling zu beeinflussen.

OpenCL Bei OpenCL kann genau dasselbe Abarbeitungsprinzip wie bei CUDA ausgemacht werden, „First-Come First-Served“. Ebenso ist es hierbei unabhängig von der Berechnungsdauer der einzelnen Kernel, sowie den gegebenen Kontrollmöglichkeiten. Als weitere Option wird das Flag „Out of Order Queue“ verwendet doch auch hier ist keine Abweichung der bisherigen Ergebnisse auszumachen, auch wenn dies gerade bei lang rechnenden Kernel zu erwarten gewesen wäre. In Abbildung 4.7 ist die Arbeitsweise dargestellt für eine Laufzeit des einen Kernels von einer Sekunde und des Zweiten mit 1,5 s.

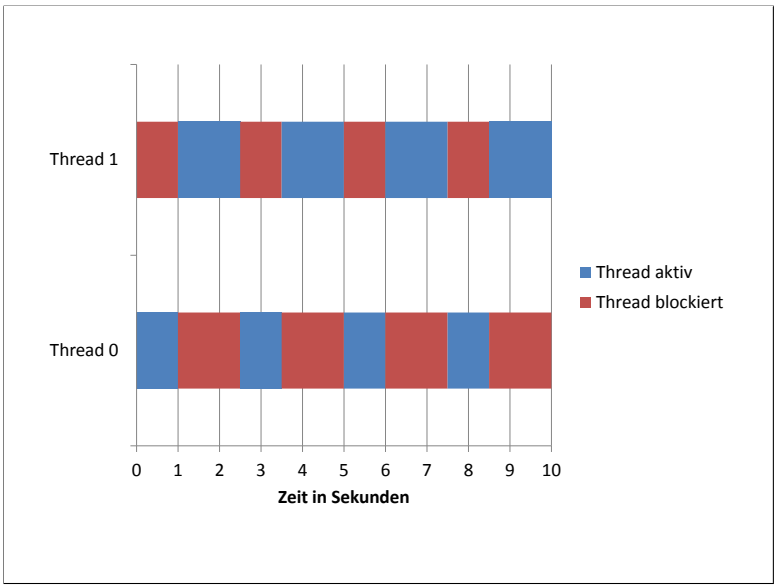

Abbildung 4.7: OpenCL Scheduling zwischen 2 Threads

Durch die verschiedenen Variationen kann nicht erreicht werden, dass die Abarbeitungsreihenfolge geändert wird. OpenCL erachtet jeden Kernel als gleichwertig und zeigt dies

konsequent durch das abwechselnde Bearbeiten auch bei unterschiedlichen Queues und selbst bei verschiedenen Kontexten.

Verschiedene Prozesse

Bei der Umsetzung dieser Untersuchung werden jeweils zwei bis drei Prozesse mit mehreren Kernelaufrufen gestartet um die gegenseitige Beeinflussung zu überprüfen.

CUDA Die Aufteilung auf verschiedene Prozesse änderte die Reihenfolge in einer unerwarteten Weise. Dies hängt jedoch direkt mit der Queue zusammen und weniger mit der Implementierung des Schedulings. Die Funktionsweise sei hier trotzdem kurz vorgestellt.

Nach dem Start ist es durch die Verzögerung beim Aufruf der einzelnen Prozesse zunächst ungleich verteilt, der Erste hat einen Vorteil und beginnt. Doch nicht wie anzunehmen mit einem Kernelaufruf, sondern direkt 96 in Folge abgearbeitet, es folgt der Zweite mit 64 und anschließend der Dritte mit 64. Im Anschluss ist wieder der Erste an der Reihe, dieses mal auch mit 64 Aufrufen, und von nun an geht diese Reihenfolge weiter bis zum Ende des Tests. Zum leichteren Verständnis des Ablaufs siehe Abbildung 4.8.

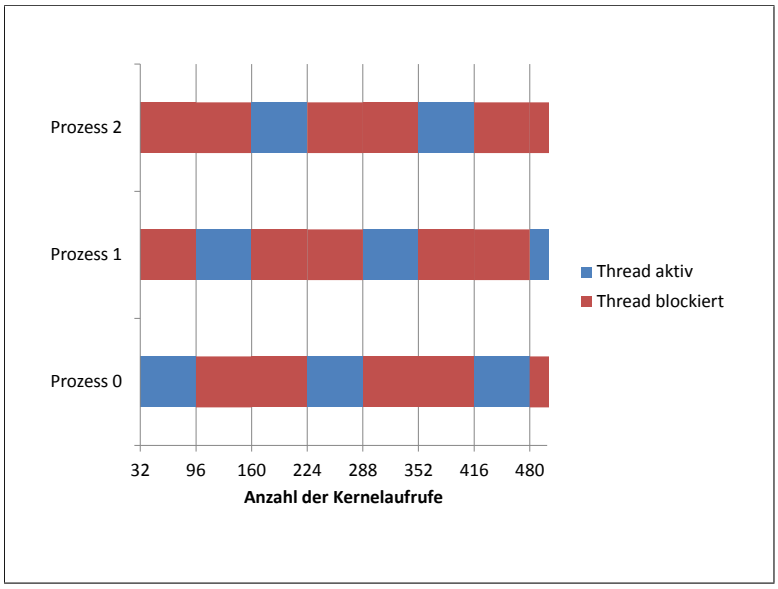

Abbildung 4.8: CUDA Scheduling zwischen 3 Prozessen

OpenCL Bei OpenCL hingegen bleibt die Reihenfolge der Abarbeitung auch bei verschiedenen Prozessen dieselbe. Es wird also nach jedem Aufruf die Kontrolle an den nächsten Prozess übergeben. Der Unterschied liegt lediglich in der Verzögerung, diese hängt direkt von der Anzahl der aktiven Prozesse ab. Bei drei Prozessen, wie in Abbildung 4.9 zu sehen, liegt die Verzögerung konsequenterweise bei zwei Aufrufen und bei mehreren Prozessen entsprechend darüber.

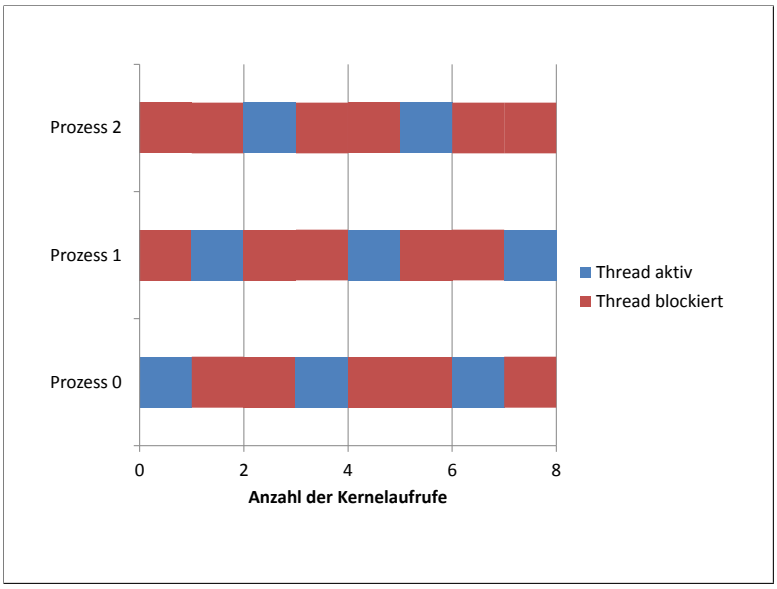

Abbildung 4.9: OpenCL Scheduling zwischen 3 Prozessen

Zusammenfassung zum Scheduling

Sowohl bei CUDA als auch bei OpenCL lässt sich wenig Einfluss auf die Reihenfolge beim Scheduling nehmen, die Abarbeitung findet nach dem „First-Come First-served“ Ansatz statt, also wer zuerst seine Anfrage schickt, wird auch zuerst abgearbeitet. Das deutet daraufhin, dass neue Jobs in eine globale Queue eingefügt werden und die Grafikkarte aus dieser bedient wird.

Bei CUDA gibt es mit dem Befehl „cudaThreadSynchronize“ die Möglichkeit einen Thread so einzuschränken, dass er nur einen Kernel an die Grafikkarte schickt und erst nach dessen Berechnung den nächsten. OpenCL bietet mit dem Kommando „clFinish(command_queue)“ eine äquivalente Möglichkeit wie bei CUDA. Mit diesen Befehlen lassen sich in Konkurrenzsituationen also gewünschte Threads einschränken. Die Schwierigkeit liegt hierbei, vor allem bei OpenCL, dass man fast keine Aussage darüber treffen kann, zu welchem Zeitpunkt ein solcher Thread wieder zum Zuge kommt. Denn der nächste Aufruf kommt frühestens in die Queue, sobald der erste zu Ende berechnet wurde. Schlimmstenfalls erst nachdem die anderen Threads ihre Arbeit vollständig beendet haben. Um darüber genauere Aussagen

treffen zu können, wird in Kapitel 4.4.3 die Queue der beiden Implementierungen genauer untersucht.

„First-Come First-served“ hat bei Echtzeitsystemen normalerweise keine Verwendung, denn die anderen Prozesse müssen solange auf die Grafikkarte warten, bis die Berechnung beendet wurde. Für Echtzeitgarantien müsste man die Ausführungszeit aller Kernel kennen, die je auf das Gerät gelangen.

4.4.3 Queue

Die Funktionsweise der Kernel Queue, eine Warteschlange für die Kernel, hat Einfluss auf die Reihenfolge beim Scheduling und den Ablauf des Host-Programms. Da bei einer vollen Queue keine neuen Anfragen mehr an die GPU geschickt werden können und das entsprechende Host-Programm an diesem Punkt blockiert wird. Des Weiteren ist es interessant zu wissen, nach wie vielen Aufrufen es wieder möglich ist Kernel an die GPU zur Bearbeitung weiterzureichen, in Abschnitt 4.4.2 ist dies schon kurz angesprochen und wird nun vertieft.

Bei OpenCL wird die Queue direkt als „command queue“ angelegt. Dieser werden dann die Kernel und Variablen zugewiesen und anschließend kann mit entsprechenden Parametern für die Anzahl der Threads der Kernel gestartet werden.

Bei CUDA geschieht dies alles implizit durch die CUDA Runtime.

Vorgehen

Um die Größe der Queue festzustellen, werden viele Kernelaufrufe in einer Schleife an die GPU geschickt. Jeder Kernel hat eine lange Laufzeit (1 s), denn die Grafikkarte fängt mit der Berechnung des ersten Kernels sofort an, um die Grenzen der Queue auch erreichen zu können. Das Absenden einer Anfrage benötigt in etwa 0,02 ms so können also in der Zeit, die für eine Berechnung benötigt wird, mehrere 10.000 Anfragen geschickt werden. Vor und nach jedem Absetzten wird ein Zeitstempel mitgeloggt, um ein mögliches Blockieren erkennen zu können, wenn eine Verzögerung auftritt. Die Anzahl der Anfragen in der Queue kann dann über die Zeitdifferenz ermittelt werden, da die Berechnungsdauer bekannt ist und die Anzahl der abgeschickten Kernel ebenfalls. Um Auswirkungen der Berechnungsdauer zu erkennen, werden die Tests mit verschiedensten Kerneln durchgeführt, wobei sich dann jeweils die entsprechende Dauer der Ausführung ändert und die Auswertung angepasst werden muss.

Algorithmus 4.9: Queuemessung CUDA

```
 1 for(int i=0; i<x; i++)
 2 {
 3   //Zeitstempel vor dem Aufruf
 4   QueryPerformanceCounter(&start_ticks);
 5   //Kernelaufruf
 6   Kernel<<<64,64>>>(dev_a, dev_b, dev_c);
 7   //Zeitstempel nach dem Aufruf
 8   QueryPerformanceCounter(&end_ticks);
 9   zeit[i] = ((double)(end_ticks.QuadPart-
         start_ticks.QuadPart)/(double)ticksPerSecond.QuadPart);
10   logfileS[i]=start_ticks;
11   logfileE[i]=end_ticks;
12 }
```

Um andere Faktoren ausschließen zu können, werden erneut einige Messungen zum Overhead durchgeführt. Der Overhead wird durch die Aufrufe selbst erzeugt, alle anderen Abschnitte wie Schleife und Zeitmessung haben eine nicht messbare Verzögerung des Codes zur Folge und können daher vernachlässigt werden.

Analyse

CUDA Bei CUDA kommt es nach der Anfrage von 162 Kernel zu einer ersten Verzögerung, an diesem Punkt ist die Queue voll und der erste abgesetzte Aufruf wird berechnet. Erst nach der Berechnungszeit von 32 Kernel können wieder neue Anfragen an die Grafikkarte geschickt werden, ebenso 32 an der Zahl. Dieser Rhythmus setzt sich bis zum Ende des Programms fort. Bei einer Variierung der Kernelgröße und damit auch der Berechnungsdauer werden dieselben Ergebnisse gemessen, ebensowenig hat die Anzahl der Aufrufparameter einen Einfluss bei der Auswertung gezeigt.

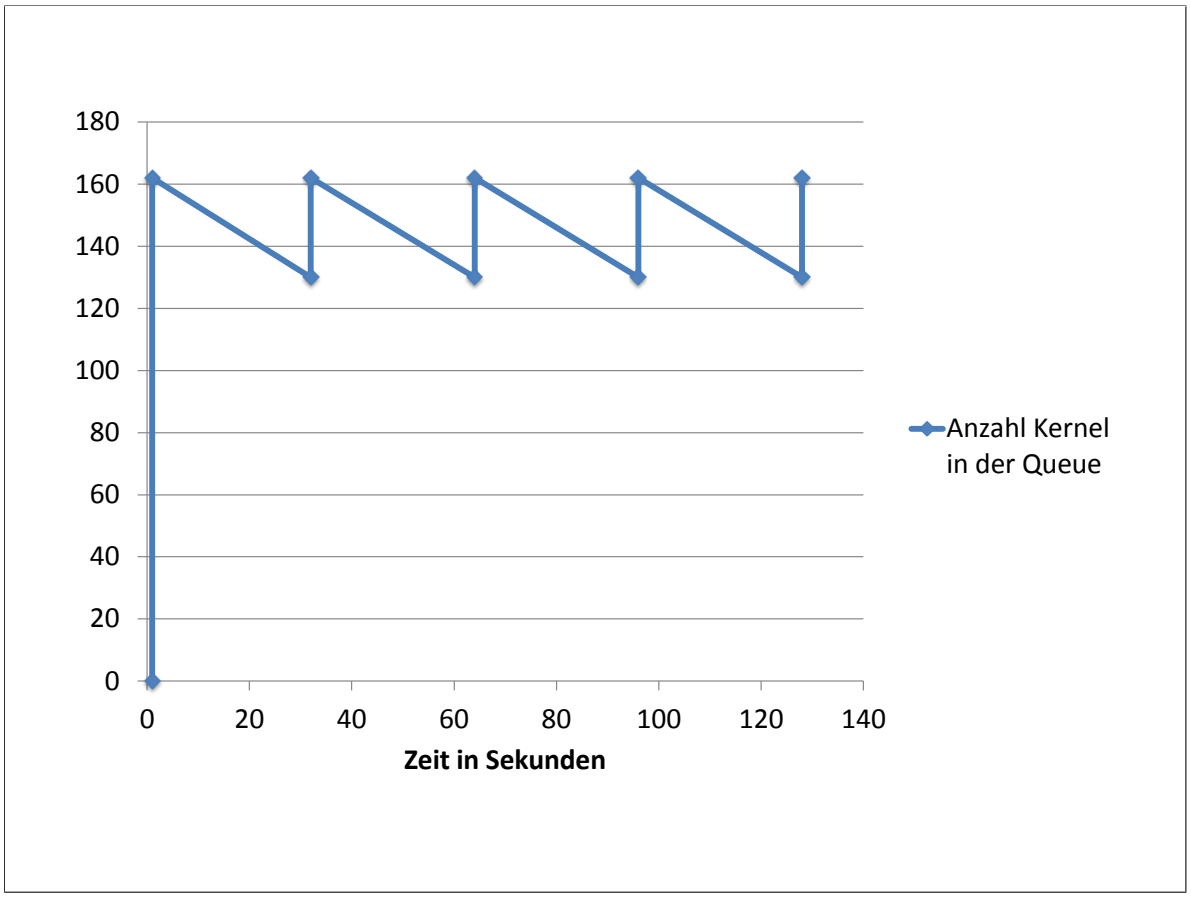

Abbildung 4.10: Queuegröße

Durch die Erweiterung um Streams kann ein verändertes Verhalten festgestellt werden. Dies wurde bei der Verwendung mehrerer Threads festgestellt, doch selbst wenn nur ein Thread aktiv ist und einen Stream verwendet, ist die veränderte Arbeitsweise der Queue zu erkennen. Die Queue ist dann schon nach 154 Anfragen gefüllt und die Verzögerung beträgt die Laufzeit von 30 Kernel, siehe hierzu Abbildung 4.11.

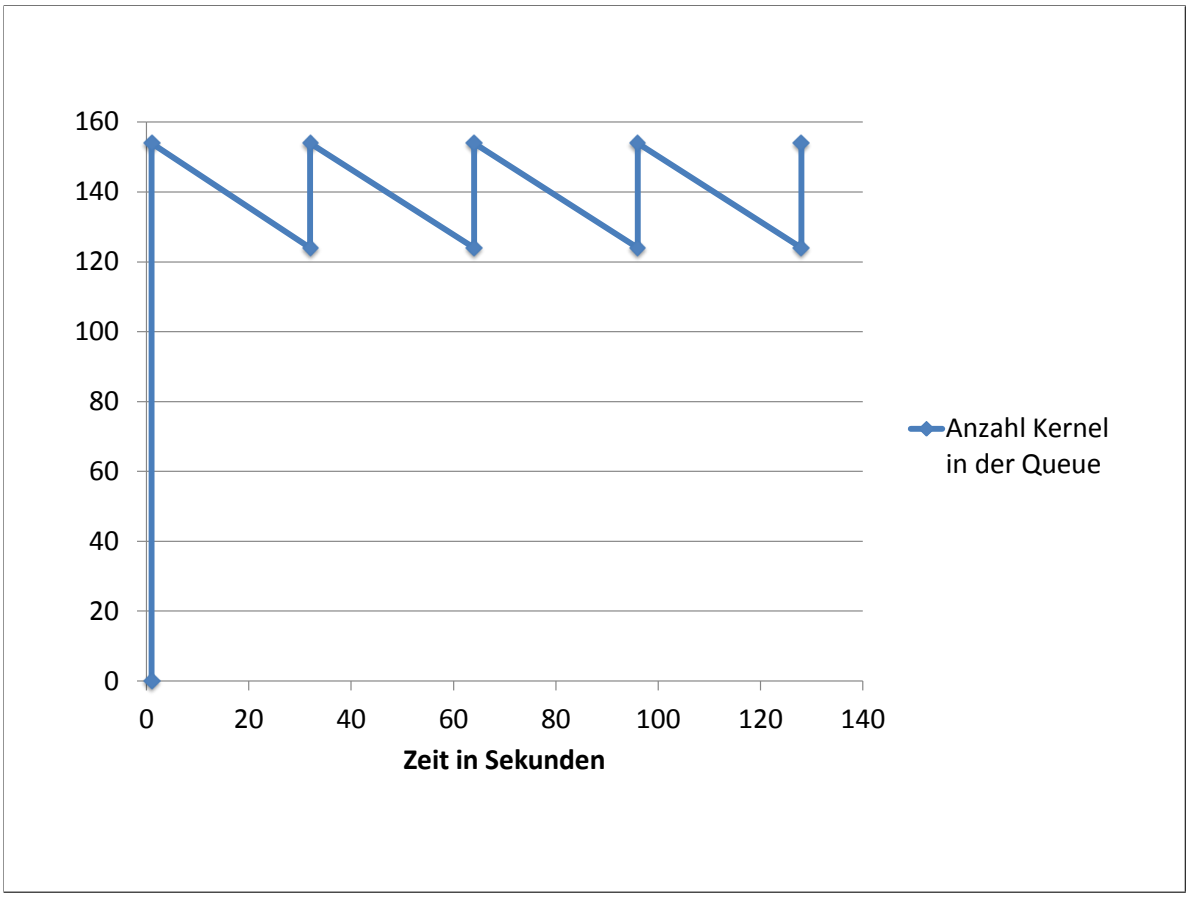

Abbildung 4.11: Queuegröße mit Streams

Durch Synchronisierung kann die Verzögerung umgangen werden. Es müssten dafür über 100 Threads parallel Anfragen an die GPU schicken. Denn jeder aufrufende Thread wartet auf das Ende seiner Berechnung und schickt bis dahin keinen erneuten Aufruf.

Zusammenfassung: Die Queuegröße liegt normalerweise bei 162 Kernelaufrufen und nach 32 abgearbeiteten wird erneut gesendet. Durch die Verwendung von Streams verringern sich diese Werte auf 154 Kernel und einer Wartezeit von 30. Es muss also darauf geachtet werden wie viele Kernel gesendet werden um das Anhalten zu verhindern. Als Alternative kann eine eigens definierte Position im Host-Programm eingesetzt werden, damit man sicher sein kann an welcher Stelle die Blockierung stattfindet, um nicht unvorhergesehen angehalten zu werden. Dies lässt sich einfach über den Befehl „cudaThreadSynchronize“ realisieren,

dadurch wird genau an der Stelle des Befehls im Code, auf die Abarbeitung aller vorheriger Anfragen gewartet. Für jegliche Form von Echtzeitgarantien ist es relevant die Größe und Funktionsweise der Queue zu kennen und zu berücksichtigen.

OpenCL Die Queue bei OpenCL lässt sich nicht zum blockieren bringen, die Grenze scheint also der Speicher des Systems zu sein und nicht eine feste Zahl wie bei CUDA. Der Befehl „clFinish“ blockiert bis zur Fertigstellung der Berechnung, äquivalent zu „cudaThreadSynchronize“. Dadurch kann auch unter OpenCL die Kontrolle über die Anzahl der Befehle in der Queue sichergestellt werden. Dies kann bei mehreren Anwendungen von Vorteil sein, denn das zweite Programm muss zuerst auf die Abarbeitung der Befehle des Ersten warten, bis die eigenen berechnet werden. Selbst bei mehreren 1000 Kernelaufrufen kam es zu keiner Unterbrechung im Host-Programm. Durch Einfügen des Befehls „clFlush“ nach jeder Anfrage, war keine Änderung des Verhaltens auszumachen. Dieser Befehl stellt sicher, dass die Anfragen davor an die GPU gesendet werden, ohne diesen Zusatz entscheidet der Treiber wann die Befehle an die GPU geschickt wird.

Verschiedene Prozesse

Als weitere Variation wird analog zu den Untersuchungen zum Scheduling die Untersuchung der Queue mit mehreren verschiedenen Prozessen durchgeführt. Dadurch wird eine eventuelle Beeinflussung die innerhalb des Programms stattfindet ausgeschlossen. Bei der effektiven Umsetzung werden jeweils zwei bis drei Prozesse mit mehreren Kernelaufrufen gestartet um die gegenseitige Beeinflussung zu überprüfen. Um Möglichst viele Faktoren die Einfluss haben können zu untersuchen, werden 3 verschiedene Prozesse erzeugt. Prozess 1 mit einem Kernelaufruf, Prozess 2 ebenso mit einem Kernelaufruf, jedoch einer doppelt so langen Berechnungszeit. Prozess 3 wiederum ruft Kernel auf die eine erneut um den Faktor 2 erhöhte Laufzeit, im Vergleich zu Prozess 2, haben. Im ersten Test wird zunächst Prozess 1 gestartet und nachdem dieser seine Queue gefüllt hat Prozess 2. Diese Untersuchung wird dann um den dritten Prozess erweitert und die Reihenfolge der einzelnen Prozesse variiert.

Erwartung: Queues sind unabhängig voneinander, pro CUDA-Kontext gibt es eine Queue mit statischer Länge. Scheduling erfolgt Round-Robin zwischen allen Queues.

CUDA Auswertung Bei jedem Testfall kann jeder Prozess eine Queue mit 162 Kernel füllen und wird dann blockiert. Darauf folgen, ähnlich wie bei einem Prozess, die Sprünge von 32 Kernel. Allerdings nicht direkt abwechselnd sondern etwas unerwartet. Der Erste Prozess kann, wie in Abbildung 4.12 zu sehen, drei mal 32 Kernel bearbeiten lassen, daraufhin der Zweite zwei mal 32 und bei drei Prozessen der Dritte ebenso zwei mal 32. Dann folgt wieder der erste Prozess, ebenso mit zwei mal 32 und so setzt es sich fort bis alle Prozesse ihre Kernel abgearbeitet haben. Die einzelnen Prozesse werden dabei entsprechend der Abarbeitungszeit der Kernel der anderen Prozesse verzögert. Bei einem Prozess allein auf der GPU ist es nur die Ausführungszeit der eigenen Kernel.

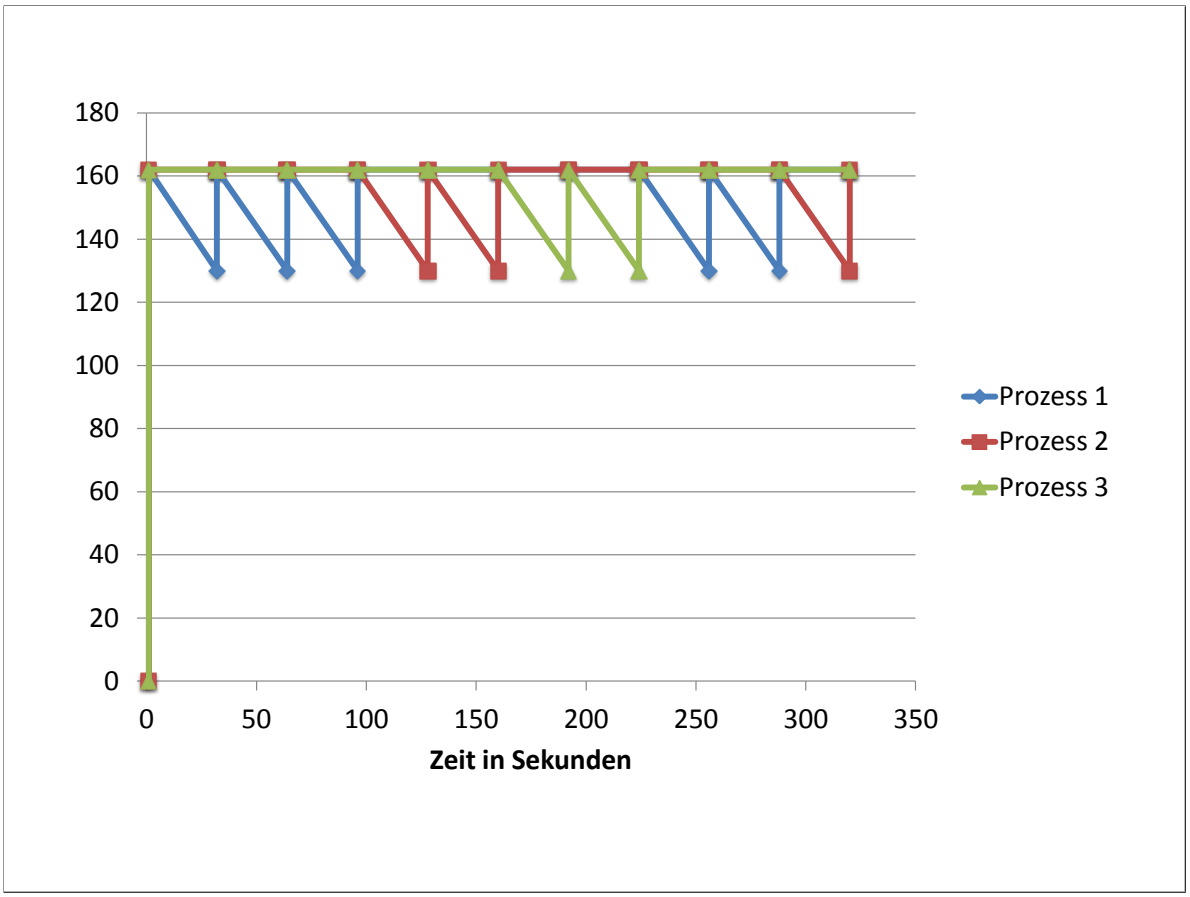

Abbildung 4.12: Queuegröße bei 3 Prozessen

OpenCL Auswertung Bei OpenCL änderte sich durch die Erweiterung auf verschiedene Prozesse nichts an der Arbeitsweise der Queue. Auch hier kommt es zu keinerlei zusätzlicher Verzögerung bei den Aufrufen der Kernel der einzelnen Prozesse.

4.4.4 CUDA Streams

Durch die Verwendung von Streams kann bei CUDA die Parallelität in 2 Bereichen erhöht werden. Momentan noch mit der Einschränkung auf einen Kontext. Zum einen kann dadurch parallel zu einer Datenübertragung eine von diesen Daten unabhängige Berechnungen stattfinden, wie in Abbildung 4.13 zu sehen.

Zum anderen können, bei geeigneter Wahl der Block- und Gridgröße, 2 Kernel in unterschiedlichen Streams parallel auf der Grafikkarte ausgeführt werden. Eine geeignete Größe kann nicht pauschal genannt werden, da dies von der Anzahl der Kerne der spezifischen GPU abhängt. Erfüllen 2 Prozesse Gleichung 4.4.4 so können diese parallel auf der Grafikkarte ausgeführt werden.

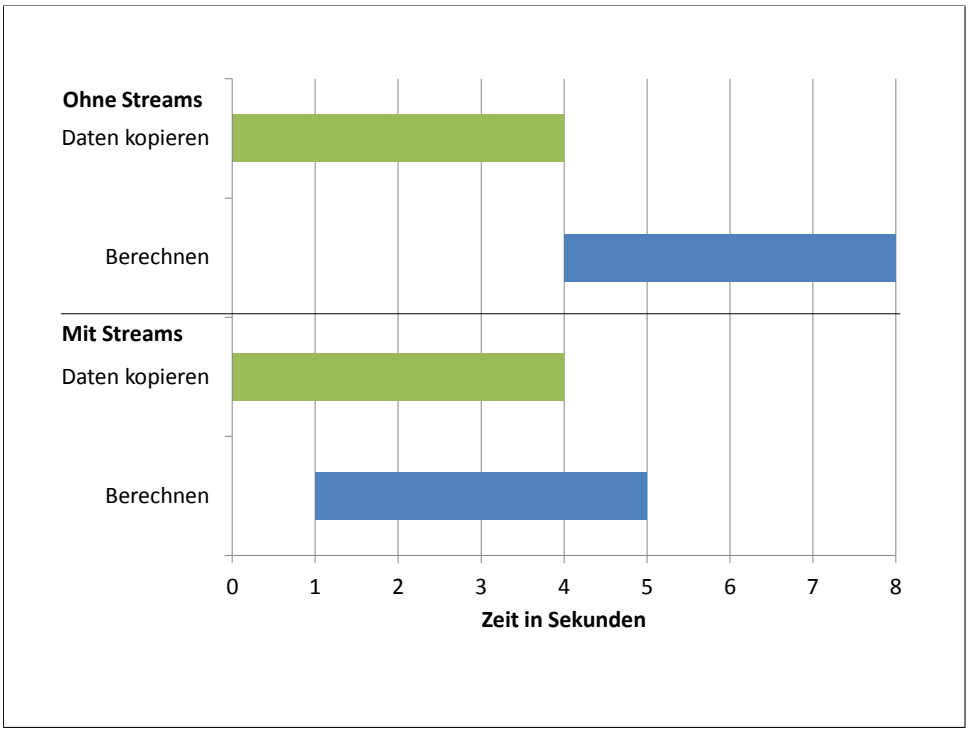

Abbildung 4.13: CUDA Streams beim Daten kopieren

$$halbeAnzahlderCUDAKerne >= AnzahlBloecke \cdot AnzahlThreads$$

Parameter beim Kernelaufruf

Im Idealfall, wie in Abbildung 4.14 zu sehen, kann dadurch die Berechnungsdauer halbiert werden. Dies gelingt allerdings nur in solchen Fällen, wo eine Erweiterung der einzelnen Anwendungen auf mehr Kerne keinen Vorteil bringt. Diese Aufteilung lässt sich auf bis zu vier parallele Streams erweitern, jeder weitere Stream wird auf dem NVIDIA-Quadro System sequentiell ausgeführt.

Ergebnisse

Durch die Verwendung von CUDA Streams ist die Möglichkeit gegeben, Kernel aus demselben Kontext parallel auszuführen und dadurch die Antwortzeit des Systems besser zu kontrollieren. Da nicht durch einen Kernelaufruf die Grafikkarte exklusiv genutzt wird, sonder Ressourcen für weitere bereithält, momentan mit der Einschränkung auf einen Kontext.

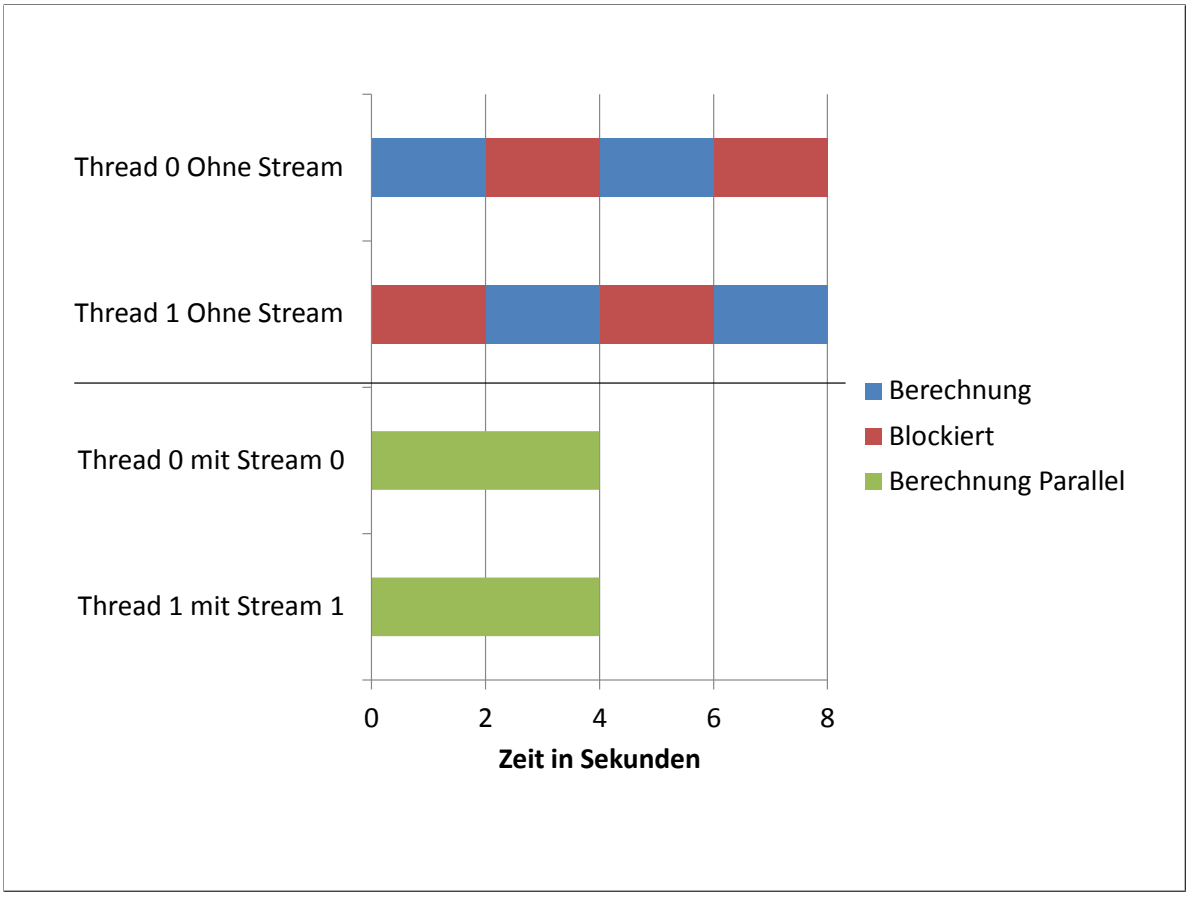

Abbildung 4.14: CUDA Streams beim parallelen Berechnen

4.5 Kontextverwaltung

Inwieweit gibt es eine Handhabe bei der Verwaltung verschiedener Kontexte und ihrer Ressourcen auf der GPU und inwieweit kann darauf Einfluss genommen werden? Auch im Hinblick auf die Synchronisierung von Daten zwischen verschiedenen Kontexten von CUDA, OpenCL und OpenGL, um Berechnungen auszulagern.

4.5.1 Kontexte erzeugen

Ziel dieser Untersuchung ist es den Ressourcenverbrauch bei der Erzeugung von Kontexten zu bestimmen. Des weiteren wird untersucht wie viele Kontexte gleichzeitig jeweils mit CUDA und mit OpenCL auf einer Grafikkarte angelegt werden können.

Vorgehen

Es werden nacheinander Kontexte erzeugt, der Ressourcenverbrauch überprüft und so versucht eine Obergrenze zu finden. Außerdem werden etwaige Möglichkeiten der Anpassung gesucht, um mehr Kontexte erzeugen zu können.

CUDA Bei CUDA liegt die Verwaltung der Kontexte standardmäßig bei der API, es können aber mit Befehlen der „Driver API“ ebenso wie bei OpenCL Kontexte direkt erzeugt werden.

Algorithmus 4.10: CUDA Kontext erzeugen

```
1  cuDeviceGet(&cuDevice, devID); //Default Device auswählen
2  cuCtxCreate(&cuContext1, 0, cuDevice); //Kontext erzeugen
```

Mit den in Algorithmus 4.10 angegebenen Befehlen wird der Kontext „cuContext1“ auf dem Gerät „cuDevice“ erzeugt.

OpenCL Bei OpenCL liegt die Kontextverwaltung immer auf der Seite des Programmierers. Algorithmus 4.11 zeigt die benötigten Befehlen, um den Kontext „clContext1“ zu erzeugen.

Algorithmus 4.11: OpenCL Kontext erzeugen

```
1  clGetPlatformIDs(1, &cpPlatform, NULL); //Platform wählen
2  clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
       //Default Device auswählen
3  clContext1 = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
       //Kontext erzeugen
```

Ergebnisse

CUDA Der Test wird auf dem NVIDIA-Quadro System mit 1024 MB Grafikspeicher durchgeführt.

Als maximale Anzahl an möglichen Kontexten stellt sich die Zahl 40 heraus, beim Versuch den 41. anzulegen wird der Fehler „out of memory“ ausgegeben. Jeder erzeugte Kontext belegt auf der NVIDIA-Quadro Karte rund 24,5 MB, von Windows sind von vornherein 84 MB belegt. Tabelle 4.9 sind die genauen Zahlen zu entnehmen.

Anzahl Kontexte	Speicherverbrauch
1	24 MB
10	243 MB
20	485 MB
30	728 MB
40	970 MB
50	nicht möglich

Tabelle 4.9: Anzahl möglicher CUDA Kontexte NVIDIA-Quadro System

Der Test wird auf dem NVIDIA-NVS System mit 512 MB Grafikspeicher ausgeführt.

Hier stellt sich als maximal mögliche Anzahl 42 heraus, beim Versuch den 43. anzulegen wird ebenso wie beim NVIDIA-Quadro System der Fehler „out of memory“ ausgegeben. Jeder erzeugte Kontext benötigt hier allerdings nur rund 12,5 MB, also etwa die Hälfte wie bei der NVIDIA-Quadro Karte. Von Windows waren hier 59 MB belegt. Der genaue Verlauf ist in Tabelle 4.10 dargestellt.

Anzahl Kontexte	Speicherverbrauch
1	13 MB
10	124 MB
20	247 MB
30	370 MB
40	421 MB
50	nicht möglich

Tabelle 4.10: Anzahl möglicher CUDA Kontexte NVS 3100M

Aus diesen zwei Untersuchung lässt sich ein Zusammenhang zwischen dem vorhandenen Grafikspeicher und der Größe eines Kontextes ableiten. Um diese Feststellung zu verfestigen wird derselbe Test auf einer NVIDIA GTX 260 mit 896 MB Speicher durchgeführt. Diese sollte demnach gerade zwischen den beiden Anderen liegen, ist jedoch erneut ganz unterschiedlich ausgefallen. Die maximale Anzahl an erzeugbaren Kontexten liegt bei 17, beim Versuch den 18. anzulegen wird ebenso der Fehler „out of memory“ ausgegeben. Jeder erzeugte Kontext benötigt auf dem NVIDIA-GTX System rund 44 MB, also deutlich mehr als auf der NVIDIA-Quadro oder dem NVIDIA-NVS System. Von Windows waren 127 MB belegt. Die genauen Werte des Verlaufs sind Tabelle 4.11 zu entnehmen.

Anzahl Kontexte	Speicherverbrauch
1	44 MB
10	443 MB
15	664 MB
20	nicht möglich

Tabelle 4.11: Anzahl möglicher CUDA Kontexte GTX 260

Es konnte keine Möglichkeit gefunden mit der man den Overhead an alloziertem Speicher auf der Grafikkarte verringern, geschweige denn umgehen konnte.

OpenCL Es folgen die Auswertungen für OpenCL auf den NVIDIA Karten und der AMD Karte.

Die erste Untersuchung findet auf dem NVIDIA-Quadro System mit 1024 MB Grafikspeicher statt. Hier liegt die Obergrenze bei 42, beim Versuch den 43. anzulegen, wird wie bei CUDA der Fehler „out of memory“ ausgegeben. Jeder erzeugte Kontext benötigt hier rund 23 MB, von Windows sind 84 MB belegt. Die Werte sind annähernd gleich der CUDA Auswertung, was auf eine ähnliche Implementierung seitens NVIDIA schließen lässt. Der genaue Verlauf bei der Erzeugung ist in Tabelle 4.12 dargestellt.

Anzahl Kontexte	Speicherverbrauch
1	23 MB
5	112 MB
10	223 MB
15	334 MB
20	445 MB
30	668 MB
40	890 MB
50	nicht möglich

Tabelle 4.12: Anzahl möglicher OpenCL Kontexte NVIDIA-Quadro System

Auswertung der NVIDIA NVS 3100M mit 512 MB Speicher. Hier liegt die Obergrenze bei 44, jeder erzeugte Kontext benötigt rund 11 MB, also auch wieder sehr ähnlich zur CUDA Implementierung. Von Windows waren 59 MB belegt.

Auswertung für die NVIDIA GTX 260 mit 896 MB Speicher. Hier war das Maximum mit 25 wieder schnell erreicht, jedoch doch deutlich über dem Wert für die CUDA Kontexte. Von Windows waren 128 MB belegt, daran liegt es folglich nicht. Jeder erzeugte OpenCL Kontext benötigt also rund 29 MB auf der GTX 260. Der genaue Verlauf ist in Tabelle 4.14 wiedergegeben.

AMD OpenCL Die Auswertung für die AMD Fire Pro Karte mit 2048 MB Speicher, hebt sich stark von den Anderen ab. Es stellt sich heraus, dass die Grenze deutlich höher liegen

Anzahl Kontexte	Speicherverbrauch
1	11 MB
5	56 MB
10	113 MB
20	226 MB
30	339 MB
40	421 MB
50	nicht möglich

Tabelle 4.13: Anzahl möglicher OpenCL Kontexte NVS 3100M

Anzahl Kontexte	Speicherverbrauch
1	29 MB
10	297 MB
15	445 MB
20	594 MB
25	742
30	nicht möglich

Tabelle 4.14: Anzahl möglicher OpenCL Kontexte GTX 260

muss, denn beim Anlegen eines Kontextes wird zunächst nicht direkt Speicher belegt. Doch auch bei AMD gibt es eine Obergrenze bei der Anzahl der Kontexte, dieser liegt genau bei 200. Beim Versuch den 201. anzulegen gibt es einen Fehler. Die Implementierungen für das Anlegen eines Kontextes unterscheiden sich hier also erheblich zwischen den beiden Herstellern.

Zusammenfassung zur Kontexterzeugung

Die Anzahl möglicher Kontexte hängt bei NVIDIA direkt von der Größe des Grafikspeichers ab, ist jedoch von Version zu Version nicht immer gleich. Denn beim Anlegen eines Kontextes wird direkt Grafikspeicher belegt. Es muss aber für die gewünschte Zielplattform und Version explizit eruiert werden, wie viel Speicher ein Kontext belegt. Versuche den Speicherverbrauch in irgendeiner Weise zu minimieren brachten keinen Erfolg. Der Ressourcenverbrauch je Kontext ist den jeweiligen Tabellen zu entnehmen.

Bei der AMD Fire Pro liegt dieser Wert fest bei 200, auch hier gab es keine Möglichkeit diesen zu erhöhen. Allerdings ist dies verglichen mit NVIDIA ein höherer Wert und diese Einschränkung somit leichter tragbar.

4.5.2 Synchronisierung zwischen CUDA / OpenCL und OpenGL Kontexten

Um OpenGL Daten mit CUDA respektive OpenCL bearbeiten zu können, muss bei beiden innerhalb des OpenGL Kontextes die entsprechende Funktion zur Registrierung der Objekte aufgerufen werden. Synchronisation zwischen verschiedenen Prozessen ist nicht direkt mit CUDA oder OpenCL möglich, zumindest bisher nicht implementiert. Dies kann jedoch im Host-Programm durch Interprozesskommunikation zum Beispiel mit einem gemeinsamen Speicher oder mit MPI (Message Passing Interface, Standard für den Nachrichtenaustausch) realisiert werden. Der Datenaustausch mit OpenGL innerhalb des selben Kontextes ist im Folgenden erklärt.

CUDA

Zunächst wird mit OpenGL ein Vertex Buffer Objekt (VBO, das Standard Datenformat in OpenGL um Daten auf die GPU zu laden) angelegt. Dieses muss dann für die Verwendung mit CUDA einmalig über den Befehl „cudaGraphicsGLRegisterBuffer“ für den CUDA Kontext registriert werden. Danach kann es beliebig oft mit dem Befehl „cudaGraphicsMapResources“ in den CUDA Bereich gemappt werden. Mittels „cudaGraphicsResourceGetMappedPointer“ wird die genaue Speicheradresse einem Pointer übergeben, mit dem dann ein CUDA Kernel aufgerufen werden kann um die Daten zu bearbeiten. In der Zwischenzeit darf das Objekt von OpenGL nicht verwendet werden, da es sonst zu einem undefinierten Fehler kommt. Nach der Berechnung mit CUDA kann mit dem Befehl „cudaGraphicsUnmapResources“ das Objekt wieder freigegeben werden um mit OpenGL weiterverarbeitet oder angezeigt zu werden. [NVI11c]

OpenCL

Bei OpenCL wird zur Verarbeitung von OpenGL Objekten folgende Vorgehensweise empfohlen. Es wird, wie bei CUDA, zuerst ein VBO im OpenGL Kontext erzeugt, mit diesem VBO als Parameter wird anschließend ein OpenCL Objekt mit dem Kommando „clCreateFromGLBuffer()“ erzeugt. Der Befehl „clEnqueueAcquireGLObjects“ übergibt die Zuständigkeit des Objektes an OpenCL davor muss zur Sicherheit im OpenGL Kontext der Befehl „glFinish“ abgesetzt werden. Die Bearbeitung der Daten durch einen OpenCL Kernel kann dann gestartet werden. Nach der Berechnung wird, analog zu CUDA, mit dem Befehl „clEnqueueReleaseGLObjects“ gefolgt von „clFinish“ das Objekt wieder für OpenGL freigegeben und kann in dessen Kontext wieder verwendet werden. [KG11]

4.5.3 Analyse und Zeitbedarf von Kontextwechseln

Motivation für die Untersuchung

Je mehr ein Kernel berechnet, desto länger ist die Grafikkarte für andere Berechnungen blockiert, daher stellt sich die Frage wie viel durch Auftrennung eines Kernel in mehrere Aufrufe, an Overhead erzeugt wird. Mit dem Gewinn einer besseren Antwortzeit des Systems.

Vorgehen

Es wurde nach Algorithmus 4.12 vorgegangen, um die Berechnung in kleinere Teile aufzuspalten.

Algorithmus 4.12: Kontextwechsel Overhead

```
__global__ void addt(const int *a,const int *b, int *c, int maxi ) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  for (int i = 0; i<(10000/maxi); i++)
  {
    c[tid] = a[tid] + b[tid] ;//c dazu zur variation sont wird evtl wegoptmiert
  }
}
for (int f =1; f<10001; f=f++) {
  int maxi = f;

  //Zeitmessunge davor
  for(int j=0; j<maxi; j++) {
    addt<<<128,128>>>(dev_a, dev_b, dev_c, maxi);
    cudaThreadSynchronize();
  }
  cudaThreadSynchronize();
  //Zeitmessung danach über die entsprechende Anzahl Aufrufe abspeichern
}
```

Der Kernel wurde also zuerst einmal mit einer 10.000 fachen Berechnung gestartet. Diese Berechnung wurde sukzessive verkleinert und die Anzahl der Aufrufe entsprechend erhöht.

Auswertung

CUDA Wie man der Abbildung 4.15 gut entnehmen kann, gibt es bei CUDA einen linearen Anstieg des Overheads durch zerlegen des Kernel. Bei der Zerlegung in 10.000 einzelne Berechnungen wird ein etwa fünffacher Overhead erzeugt der linear ansteigt. Dies stellt einen gangbaren Weg dar, um die Blockierung der Grafikkarte möglichst gering zu halten, und so eine bessere Nutzbarkeit für mehrere parallele Programme zu erreichen und die gesamte Antwortzeit des Systems zu erhöhen. Man muss im einzelnen Fall dann abwägen

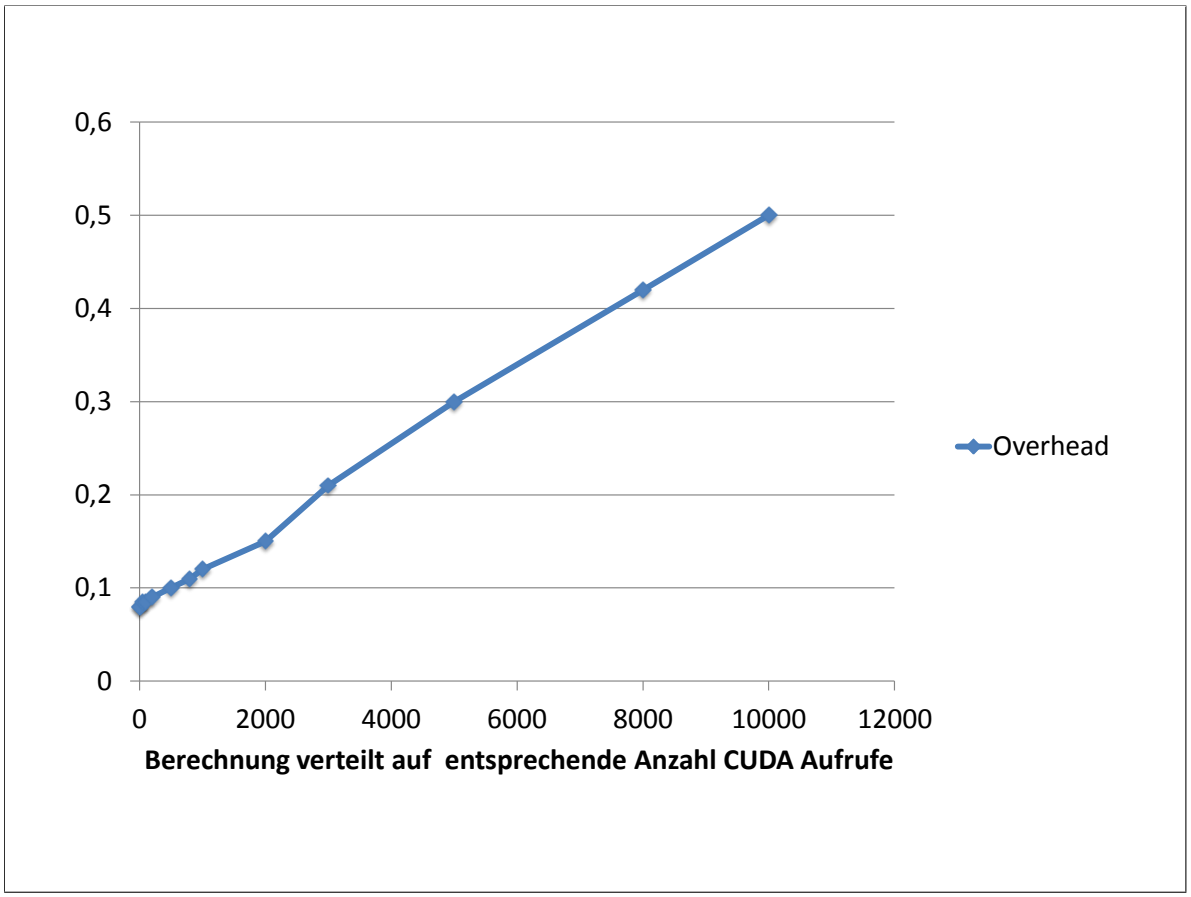

Abbildung 4.15: Overhead durch Kontextwechsel bei CUDA

wie weit man gehen möchte, um nicht durch den Overhead die gesamte Beschleunigung wieder zu egalisieren.

OpenCL Bei OpenCL ist, wie in Abbildung 4.16 zu sehen, der Overhead nicht ganz so ausgeprägt wie bei CUDA. Es wird bei der Zerlegung der Berechnung in 10.000 einzelne Kernelaufrufe, die Berechnungszeit verdoppelt. Ebenso wie bei CUDA ergibt sich ein linearer Anstieg. Auch bei OpenCL kann durch die Aufteilung einer Berechnung in mehrere kleinere Aufrufe, die Antwortzeit des Systems entsprechend verbessert werden.

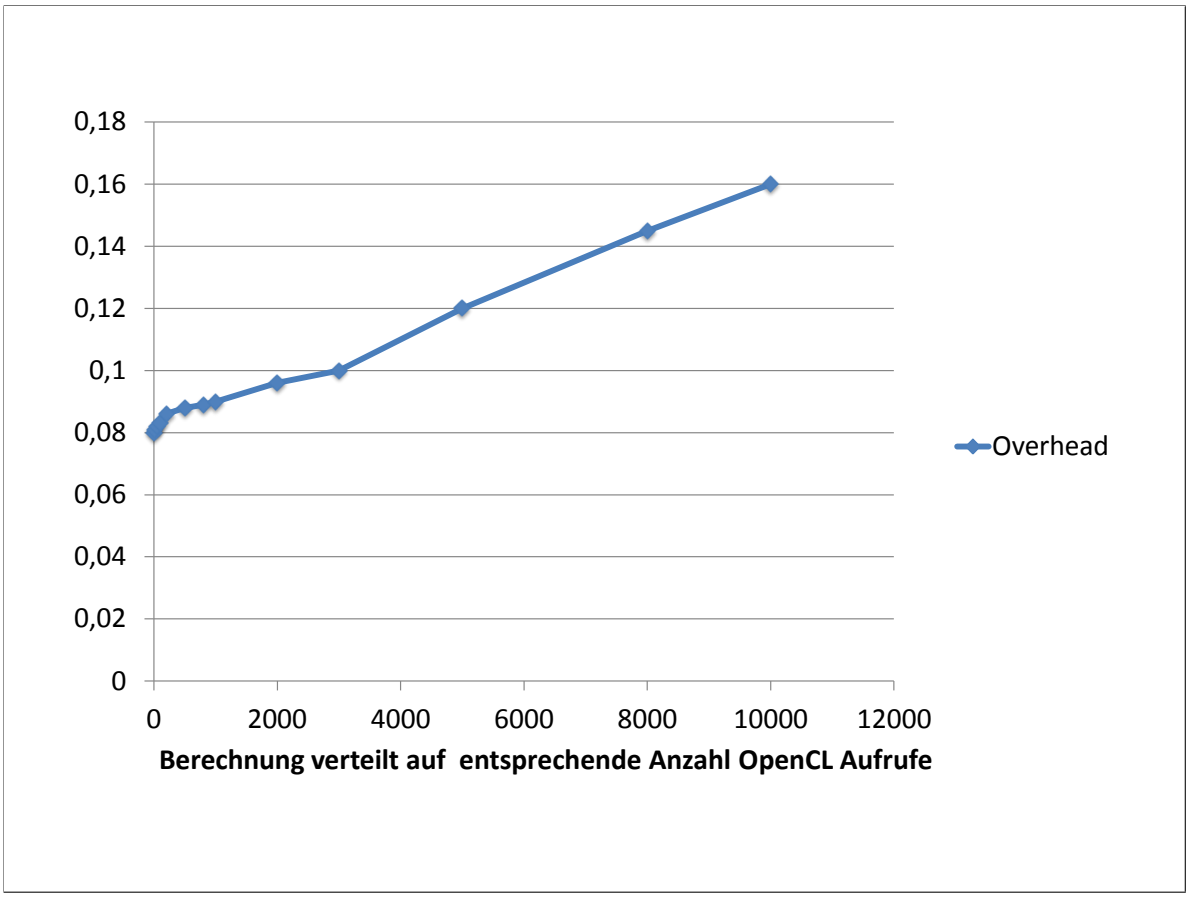

Abbildung 4.16: Overhead durch Kontextwechsel bei OpenCL

5 Zusammenfassung und Ausblick

Die GPGPU-Standards CUDA und OpenCL sind für Echtzeitsysteme nicht vorgesehen. Ziel dieser Arbeit war es daher, Möglichkeiten und Grenzen von OpenCL und CUDA bezüglich Echtzeit und Isolation zu ermitteln.

Um diese Aussagen treffen zu können, war es nötig zunächst die grundlegende Funktionalität der beiden Schnittstellen, CUDA und OpenCL, zu sichten und relevante Faktoren zu ermitteln. Um nachvollziehbare Werte zu erhalten, wurden während dieser Arbeit mehrere 10.000 Messungen durchgeführt, mit meist eigens erstellten Messmethoden und den von den GPU-Herstellern zur Verfügung stehenden Profiling-Tools.

Bei der Untersuchung der Möglichkeiten zum Datentransfer, wurde die Möglichkeit Daten im Arbeitsspeicher des Host derart anzulegen, dass ein Auslagern nicht stattfindet, untersucht. Diese Funktionsweise bieten beide Schnittstellen, sowohl CUDA als auch OpenCL. So kann damit eine höhere Geschwindigkeit bei der Übertragung der Daten zur Grafikkarte erzielt werden es wird also die Antwortzeit des Systems verringert.

Mit den Untersuchungen zur Speicherverwaltung auf der GPU konnten wichtige Größen bestimmt werden, die für eine Implementierung zu berücksichtigen sind. Sie sind in Tabelle 5.1 für NVIDIA noch einmal zusammengefasst.

Wert	CUDA	OpenCL	Zusatz
Blockgröße	1 MB	1 MB	kleinste Einheit die mindestens alloziert wird
Maximale Anzahl der Elemente pro Block	2048 Stück	4096 Stück	mehr als ein Objekt je Block ist möglich
Mindestgröße die belegt wird	512 Byte	256 Byte	auch wenn die Objektgröße darunter liegt
Blöcke werden immer aufgefüllt, jedoch nicht über mehrere Blöcke hinweg.			
Ganze Blöcke werden nach dem Freigeben für größere Bereiche wieder zusammengefasst, durch die Umorganisation entsteht jedoch ein Overhead.			

Tabelle 5.1: Zusammenfassung zur Speicherverwaltung bei NVIDIA

Die Speicherverwaltung kann durch die Möglichkeit, direkt große Bereiche zu allozieren, mit Einschränkungen selbst übernommen werden, um Isolation zu gewährleisten. Durch die Tatsache, dass angelegter Speicher von konkurrierenden Anwendungen nicht verdrängt werden kann ist dies eine zuverlässige Möglichkeit auf die Verwaltung Einfluss zu nehmen.

Der Speicherschutz ist bei NVIDIA in Hardware implementiert und Isolation zwischen zwei Kontexten ist daher gewährleistet.

Das Scheduling der Kernelaufrufe an die Grafikkarte, konnte nur mit der Möglichkeit der CUDA Streams für ein weiches Echtzeitverhalten beeinflusst werden. Dabei ist es jedoch nötig die Kernelaufrufe entsprechend anzupassen, damit von einem Kernelufruf nur ein Teil der Grafikkarte verwendet wird, und somit Ressourcen für folgende Anfragen zur Verfügung stehen. Momentan ist diese Möglichkeit noch auf einen Kontext beschränkt, bietet aber den stabilsten Ansatz. Bei OpenCL ist die Möglichkeit, Kernel parallel auszuführen, ebenfalls vorgesehen, wird jedoch von der Implementierung erst in Zukunft unterstützt werden. Die gewonnenen Ergebnisse über das Scheduling und die Queue, sind bei einer entsprechenden Umsetzung eines Algorithmus hilfreich.

Der Ressourcenverbrauch bei der Kontextverwaltung hat deutliche Unterschiede zwischen NVIDIA und AMD aufgezeigt, und weniger zwischen CUDA und OpenCL. Bei NVIDIA wird durch jede Kontexterzeugung einge gewisse Menge an Grafikkartenspeicher belegt. Bspw. bei dem NVIDIA-Quadro System waren es ca. 24 MB pro erzeugtem Kontext. Die Anzahl gleichzeitig erzeugbarer Kontexte war durch den Grafikspeicher beschränkt. Bei dem AMD-Fire-Pro System konnten 200 Kontexte erzeugt werden, ein Grafikspeicherverbrauch konnte für das Erzeugen von Kontexten nicht nachgewiesen werden.

Mit der abschließenden Untersuchung zum Overhead bei Kontextwechseln konnte ein wichtiger Ansatzpunkt geliefert werden, um die Antwortzeit, ein wichtiger Parameter des Echtzeitverhaltens, zu verringern. Es konnte gezeigt werden, dass mit der Aufteilung von Kernel in kleinere Teilprobleme der Overhead linear ansteigt. Mit diesem Wissen kann für einen gegebenen Kernel gut abgeschätzt werden, wie sich die Gesamt-Ausführungszeit bei einer entsprechenden Aufteilung erhöht und damit die Antwortzeit des Systems drastisch verringert wird.

Ausblick

Um CUDA und OpenCL in Echtzeitsystemen einsetzen zu können, ist die oberste Priorität die Unterbrechbarkeit von Kernel. Denn nur so kann verhindert werden, dass ein Kernel mit hohem Ressourcenverbrauch die GPU unbegrenzt lange exklusiv nutzen kann. Von NVIDIA ist diese Funktionalität schon angekündigt [NVIb] und soll in der neuen Generation an Grafikchips Anfang 2012 auf den Markt kommen. Die Echtzeitunterstützung von GPUs wird nicht zuletzt aufgrund dieser Entwicklung zunehmend an Bedeutung gewinnen.

Literaturverzeichnis

[AMD] AMD. URL http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl.aspx. OpenCL bei AMD, Zugriff am: 31.08.2011.

[AMD11a] AMD, 2011. URL http://www.amd.com/de/products/workstation/graphics/ati-firepro-3d/v5900/Pages/v5900.aspx. Zugriff am: 06.11.2011.

[AMD11b] AMD. *AMD Intermediate Language*. AMD, 2.4 edition, 2011.

[AMD11c] AMD. TOOLS, 2011. URL http://developer.amd.com/tools/Pages/default.aspx. Zugriff am: 06.11.2011.

[APP11] APPLE, 2011. URL http://support.apple.com/kb/HT4728?viewlocale=de_DE. Zugriff am: 07.11.2011.

[Ber10] Berkeley, 2010. URL http://boinc.berkeley.edu/trac/wiki/GPUApp. Zugriff am: 07.11.2011.

[Boa] O. A. R. Board. OpenMP. URL http://openmp.org/wp/. Zugriff am: 24.10.2011.

[Boa11] O. A. R. Board. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, 3.1 edition, 2011.

[BR10] N. Black, J. Rodzik. My Other Computer is your GPU: System-Centric CUDA Threat Modeling with CUBAR. *CS8803SS Project, Spring 2010*, 2010.

[BSW08] M. Boyer, K. Skadron, W. Weimer. Automated Dynamic Analysis of CUDA Programs. *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[BYF+09] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. *International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2009.*, 2009.

[Dev11] A. M. Devices. *AMD Accelerated Parallel Processing OpenCL*. AMD, One AMD Place P.O. Box 3453 Sunnyvale, CA 94088-3453, rev1.3f edition, 2011.

[DWA08] A. DWARAKINATH. *A Fair-Share Scheduler for the Graphics Processing Unit*. Master's thesis, STONY BROOK UNIVERSITY, 2008.

[GGHS09] M. Guevara, C. Gregg, K. Hazelwood, K. Skadron. Enabling Task Parallelism in the CUDA Scheduler. In *Workshop on Programming Models for Emerging Architectures*, pp. 69–76. 2009.

[Gün11] M. Günsch. AMD-GPU mit „Stacked Memory“ gesichtet, 2011. URL http://www.computerbase.de/news/2011-10/amd-gpu-mit-stacked-memory-gesichtet/.

[GPG] GPGPU.ORG. General-Purpose Computing on Graphics Hardware. URL http://gpgpu.org/. Zugriff am: 31.08.2011.

[Gro] K. Group. URL http://www.khronos.org/opencl/. Zugriff am: 31.08.2011.

[HTA08] P. H. Ha, P. Tsigas, O. J. Anshus. Wait-free Programming for General Purpose Computations on Graphics Processors, 2008.

[Int] Intel. URL http://software.intel.com/en-us/articles/opencl-sdk/. Zugriff am: 31.08.2011.

[Int11] Intel, 2011. URL http://www.intel.com/support/processors/sb/CS-032814.htm. Zugriff am: 06.11.2011.

[JWT+10] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, J. Gray. CUDACL: A Tool for CUDA and OpenCL Programmers. In *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada*. 2010.

[KG11] A. M. Khronos Group. *The OpenCL Specification*. Khronos Group, 1.1 edition, 2011.

[Kön09] B. König. OpenCL Platform-Modell, 2009. URL http://upload.wikimedia.org/wikipedia/de/9/96/Platform_architecture_2009-11-08.svg. Zugriff am 06.09.2011.

[Kop11] H. Kopetz. *Real-Time Systems*. Springer, 2011.

[Lux11] Luxrend, 2011. URL http://www.luxrender.net/wiki/Luxrender_and_OpenCL. Zugriff am: 07.11.2011.

[Löw09] C. Löwen. *PARALLELE BERECHNUNG KOMBINATORISCHER VEKTORFELDER MIT CUDA*. Master's thesis, Fachhochschule Südwestfalen, 2009.

[MGM+11] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, 2011.

[Mic06] Microsoft. Display driver model, 2006. URL http://msdn.microsoft.com/de-de/library/aa480220.aspx. Zugriff am 11.11.2011.

[Mic09] Microsoft, 2009. URL http://msdn.microsoft.com/en-us/windows/hardware/gg487368. Microsoft Website, Zugriff am: 16.07.2011.

[Muro9] G. S. Murthy. *Optimal Loop Unrolling For GPGPU programs*. Master's thesis, The Ohio State University, 2009.

[NBGS08] J. Nickolls, I. Buck, M. Garland, K. Skadron. Scalable Parallel Programming with CUDA. *GPUs for Parallel Programming*, Vol. 6, No. 2, 2008.

[Net07] M. D. Network. How To Use QueryPerformanceCounter to Time Code, Article ID: 172338, 2007. URL http://support.microsoft.com/kb/172338/de. Zugriff am: 03.08.2011.

[NVIa] NVIDIA. URL http://developer.nvidia.com/category/zone/cuda-zone. NVIDIA CUDA Website, Zugriff am: 31.08.2011.

[NVIb] NVIDIA. Preemption, GTC: Nvidia gibt Ausblick auf kommende Grafikchips. URL http://www.heise.de/newsticker/meldung/GTC-Nvidia-gibt-Ausblick-auf-kommende-Grafikchips-1083430.html.

[NVI08] NVIDIA. *The CUDA Compiler Driver NVCC*, 2.1 edition, 2008.

[NVI10] NVIDIA. *CUDA C Programming Guide Version 3.2*, 2010.

[NVI11a] NVIDIA. CUDA 4.1 Release Candidate, 2011. URL http://developer.nvidia.com/content/cuda-41-release-candidate-now-available.

[NVI11b] NVIDIA. *CUDA C BEST PRACTICES GUIDE*, 2011.

[NVI11c] NVIDIA. *CUDA C Programming Guide Version 4.0*, 2011.

[NVI11d] NVIDIA. DirectCompute, 2011. URL http://developer.nvidia.com/directcompute. Zugriff am 11.10.2011.

[NVI11e] NVIDIA. *OpenCL Best Practices Guide*, 2011.

[NVI11f] NVIDIA. *OpenCL Programming Guide for the CUDA Architecture*, 2011.

[NVI11g] NVIDIA. *Parallel Nsight 2.0 User Guide*. NVIDIA, 2.0 edition, 2011.

[NVI11h] NVIDIA. *PTX: Parallel Thread Execution*. NVIDIA, version 2.3 edition, 2011.

[PKL10] H. Peters, M. Köper, N. Luttenberger. Efficiently using a CUDA-enabled GPU as shared resource. *2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)*, 2010.

[PSHL10] H. Peters, O. Schulz-Hildebrandt, N. Luttenberger. Parallel external sorting for CUDA-enabled GPUs with load balancing and low transfer overhead. *2010 IEEE International Symposium on Parallel Distributed Processing Workshops and Phd Forum IPDPSW*, pp. 1–8, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5470833.

[SGS10] J. E. Stone, D. Gohara, G. Shi. OpenCL: A Parall el ProgrammingStandard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, pp. 66–72, 2010.

[SIG11] P. SIG. PCI Express Specification, 2011. URL http://www.pcisig.com/specifications/pciexpress/. Zugriff am 31.08.2011.

[SK10] J. Sanders, E. Kandrot. *CUDA by Example*. Addison -Wesley, 2010.

[SP] T. K. Steve Pronovost, Henry Moreton. Windows Display Driver Model (WDDM) v2 And Beyond. Microsoft Winhec.

[STE11] W. STERNA. *SOFTWARE RENDERER ACCELERATED BY CUDA TECHNOLOGY*. Master's thesis, FACULTY OF FUNDAMENTAL PROBLEMS OF TECHNOLOGY WROCAW UNIVERSITY OF TECHNOLOGY, 2011.

[W1z11] W1zzard. GPU-Z, 2011. URL http://www.techpowerup.com/gpuz/. Zugriff am 04.07.2011.

[WPSAM10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2010.*, 2010.

[YCW+11] C.-T. Yang, T.-C. Chang, H.-Y. Wang, W. C. Chu, C.-H. Chang. Performance Comparison with OpenMP Parallelization for Multi-core Systems. *Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 232–237, 2011.

Wenn nicht anders angegeben wurden alle URLs zuletzt am 01.09.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Fabian Römhild)