HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Mario Gienapp

GPGPU Computing - Parallele Programmierung mit CUDA und OpenCL

2012

Fakultät **Elektro- und Informationstechnik**

BACHELORARBEIT

GPGPU Computing - Parallele Programmierung mit CUDA und OpenCL

Autor:
Mario Gienapp

Studiengang:
Multimediatechnik

Seminargruppe:
MK08

Erstprüfer:
Prof. Dr.-Ing. Alexander Lampe

Zweitprüfer:
Dipl.-Ing. Norbert Göbel

Mittweida, 2012

Bibliografische Angaben

Gienapp, Mario: GPGPU Computing - Parallele Programmierung mit CUDA und OpenCL, 85 Seiten, 12 Abbildungen, Hochschule Mittweida (FH), Fakultät Elektro- und Informationstechnik

Bachelorarbeit, 2012

Referat

Die Parallelisierung stellt einen wichtigen Faktor zur Geschwindigkeitssteigerung von Prozessoren dar. GPGPU Computing bezeichnet hierbei die erweiterte Nutzung der GPU für allgemeine Aufgaben - zusätzlich zum ursprünglichen Einsatzbereich des Grafikrenderings. Aufgrund ihrer Architektur bietet sie sich optimal für eine parallele Verarbeitung von Berechnungen an.

Die vorliegende Arbeit liefert einen Überblick in die Möglichkeiten der GPU Programmierung. Dazu werden die Hardwarearchitektur der GPU, sowie Unterschiede zur CPU erläutert. Vorhandene Schnittstellen verglichen und mit Codebeispielen dargestellt. Der Einsatz in der Praxis, wird anhand verfügbarer Entwicklungswerkzeuge und Anbindung an verschiedene Programmiersprachen untersucht.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

ALU	Arithmetic Logic Unit
CGA	Color Graphics Adapter
DDR	Double Data Rate
DLP	Data Level Parallelism
EGA	Enhanced Graphics Adapter
FMA	Fused Multiplay Addition
FPU	Floating Point Unit
GPC	Graphic Processing Cluster
HGC	Hercules Graphics Card
ILP	Instuction Level Parallelism
JNI	Java Native Interface
MADD	Multiply-Addition
MDA	Monochrome Display Adapter
OEM	Original Equipment Manufacturer
SFU	Special Function Unit
SM	Streaming Multiprocessor
SPU	Stream Processing Unit
TLP	Thread Level Parallelism
VGA	Video Graphics Array

1 Einleitung

1.1 Vorwort

Noch vor wenigen Jahren, erfüllten allein Supercomputer in großen Rechenzentren, die Anforderungen von ”High Performance Computing”-Aufgaben. Doch dies hat sich gewandelt. So gut wie jeder aktuelle PC ist eine Art ”Supercomputer”, ausgestattet mit hunderten Recheneinheiten und eigenem Speicher. Denn er besitzt eine GPU, die ”Graphic Processing Unit”. [OHL+08]

Bisher war ihre Rechenleistung vor allem für die Unterhaltungsindustrie, durch die Entwicklung immer realistischer aussehender Spiele oder Animationsfilme, von Bedeutung. Speziell für diesen Einsatz wurde die GPU auch entworfen[1]. Zum Programmieren existieren dafür schon seit einiger Zeit[2] die beiden Programmierschnittstellen *OpenGL* und *DirectX*. Um sie zu nutzen ist allerdings einige Einarbeitungszeit von Nöten. Es liegt auf der Hand, dass durch die Spezialisierung auf die Entwicklung von Grafikanwendungen, ein anderer Ansatz als in der CPU-Programmierung verfolgt wird.

Doch aktuell ist das Wort GPGPU, eine Abkürzung für ”General Purpose Computation on Graphics Processing Unit”, zu deutsch ”Allzweck Berechnung auf Grafikprozessoren”, im Umlauf. Dahinter verbergen sich neue Möglichkeiten die GPU auch für andere Aufgaben als die Grafikberechnung zu verwenden. Gemeint sind die beiden Schnittstellen *CUDA* und *OpenCL*, die eine CPU ähnliche Programmierung ermöglichen. Kenntnisse von Grafikprogrammierung sind damit nicht mehr notwendig.

1.2 Motivation

Durch Verwendung paralleler Berechnungen kann die Ausführungsgeschwindigkeit eines Algorithmus enorm beschleunigt werden. Die GPU bietet sich aufgrund ihrer Hardwarearchitektur optimal dafür an. Diese Arbeit untersucht, welche Möglichkeiten, zur Programmierung der GPU für den Entwickleralltag verfügbar sind. Dazu wird zuerst auf die allgemeine Vorgehensweise bzw. Besonderheiten der GPU-Programmierung, sowie die beiden Schnittstellen CUDA und OpenCL, eingegangen. Der Leser ist anschließend in der Lage in eigenen Projekten auf die Schnittstellen zurückzugreifen. Allerdings ist der Einsatz der GPU nicht immer sinnvoll, weshalb auch untersucht wird, in welchen Fällen die Verwendung lohnenswert ist bzw. wann ggf. optimiert werden kann.

[1] siehe Kapitel 2.1
[2] Seit Anfang/Mitte der Neunziger Jahre, siehe Kapitel 2.1

1.3 Aufbau der Arbeit

Neben dieser Einleitung besteht die Arbeit aus 5 Kapiteln. Am Anfang wird auf die Entwicklung der GPU und die daraus resultierende heutige Hardwarearchitektur eingegangen. In diesem Kapitel soll vermittelt werden, warum sich die GPU gut für die Parallelisierung eignet und wie im Vergleich die CPU Berechnungen parallel ausführt. Das darauf folgende Kapitel geht, resultierend aus der Hardware, auf die Besonderheiten der GPU- im Vergleich zur CPU-Programmierung ein. Im nächsten Kapitel werden dann speziell die beiden Schnittstellen CUDA und OpenCL erläutert und verglichen. Kapitel 5 untersucht die Praxistauglichkeit anhand verfügbarer Werkzeuge, sowie den sinnvollen Einsatz und Optimierungsmöglichkeiten der GPU-Programmierung. Mit einer Zusammenfassung und einem Ausblick wird die Arbeit in Kapitel 6 abgeschlossen.

Zum besseren Verständnis dieser Arbeit, sind grundlegende Kenntnisse in Hardwarearchitektur und der Funktionsweise einer CPU hilfreich. Zudem sollte man Quellcode der Programmiersprache C lesen können und vielleicht auch schon selbst kleine Programme geschrieben haben. Für Java Programmierer sei auf das Buch ”C für Java Programmierer” von Prof. Carsten Vogt [Vog07] verwiesen, welches die wichtigsten Unterschiede aufzeigt und so einen schnellen Einstieg bietet.

Tauchen neue Begriffe im Text auf, so sind diese *hervorgehoben*. Die Bedeutung der Begriffe kann dann im Glossar nachgeschlagen werden.

2 GPU-Architektur

Die Hardwarearchitektur einer GPU unterscheidet sich in Aufbau und Funktionsweise wesentlich von einer CPU. Daraus ergeben sich neue Verfahrensweisen, welche für die Programmierung zu beachten sind. Dieses Kapitel stellt die Besonderheiten, sowie Unterschiede zur CPU vor.

2.1 Rückblick

Zum besseren Verständnis moderner GPU Computing Architekturen, soll zuerst die Entstehung der ersten Grafikprozessoren und die darauf folgende rapide Weiterentwicklung zu parallelen Multiprozessoren betrachtet werden.

2.1.1 Erste grafische Adapter

Die Vorläufer heutiger Grafikkarten waren MDA - Steckkarten. Diese stellten Text in 25 Zeilen mit jeweils 80 Zeichen in einer Auflösung von 720 x 350 Pixeln dar. 2 Farben, schwarz und je nach Ausführung des Monitors grün, bernsteinfarben oder selten sogar weiß konnten dargestellt werden. Die Karte konnte keine Pixel direkt ansteuern, die Anzeige erfolgte durch eine Ansammlung von Bildpunkten. Eine damals sehr bekannte Weiterentwicklung war der HGC - Adapter der Firma Hercules. Parallel dazu brachte IBM einen weiteren Standard auf den Markt, den CGA. 3 Anzeigemodi, 4 Farben bei einer Auflösung von 320 x 200, 2 Farben mit 640 x 320 Pixeln und 16 Farben mit 160 x 100 Pixeln waren verfügbar. Letzterer wurde allerdings aufgrund der geringen Auflösung und Problemen bei der Auswertung des Intensitätsbits[3] sehr selten verwendet. Der EGA - Standard erschien kurze Zeit später. Damit war es mit einem unterstützenden Monitor möglich, 16 Farben bei 320 x 200 oder 640 x 350 Pixeln darzustellen. Abwärtskompatibilität zum CGA - Standard war vorhanden. Die Farben wurden im R-G-B+Intensity Format übertragen. Durch die Überlagerung der Farben Rot, Grün und Blau konnten 8, mit der Hinzunahme des Intensity-Bits 16 Farben erzeugt werden. Erstmals wurde außerdem ein Grafik-*BIOS* eingeführt. Zuvor erfolgte die Steuerung noch über das PC-BIOS. Bis zu diesem Standard, wurden alle Bilder noch digital übertragen. Dies änderte sich mit dem Erscheinen der ersten VGA-Karten. Nun war es möglich 256 Farben aus einer Palette von ca. 250000 Farben darzustellen. Da man aber noch keine Möglichkeit sah ein digitales Protokoll zu entwickeln, welches diese Farbpalette überträgt, begann man analoge Monitore und Übertragungswege zu verwenden. Damit war es praktisch möglich unbegrenzt viele Farben darzustellen. Mit S-VGA (S für

[3] Für Darstellung einer zusätzlichen Helligkeitsstufe notwendig

Super) und den darauf folgenden Weiterentwicklungen wurde dies dann auch umgesetzt. [Sch01] [Gr110] [MeG]

2.1.2 Weitere Entwicklung

Bis zu diesem Zeitpunkt, war die Hauptaufgabe der Grafikkarten, berechnete Daten der CPU zu empfangen und diese als Bildsignal auszugeben. Einen Grafikprozessor gab es noch nicht. Die ersten Ansätze, Berechnungen auf Grafikkarten durchzuführen, fanden in Form von Windows-Beschleunigern in Produkten der Firma S3[4] statt. Diese konnten einfache Befehle, wie z.B. ”Viereck zeichnen” ausführen. Das Modell 911 war im Jahr 1991 die erste Karte die 2D - Beschleunigungsfunktionen durchführen konnte. [MeG]

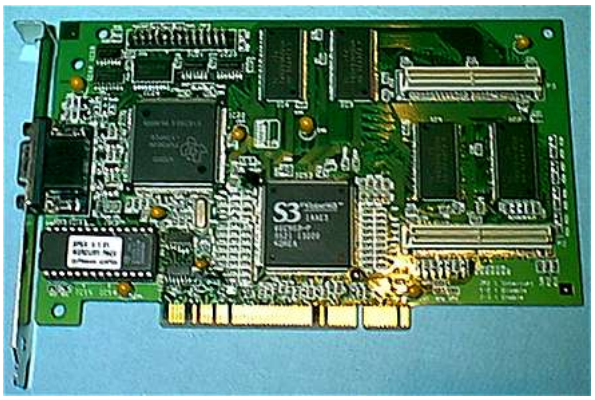

Abbildung 2.1: Das Modell 911 der Firma S3, [Sch01]

Erste 3D-Beschleuniger

Ein weiterer Entwicklungssprung war die Einführung des ersten 3D-Chips, der damals noch unbekannten Firma Nvidia. Dieser nannte sich NV1, konnte sich aber nicht durchsetzen, da sich die Programmierung schwierig gestaltete. [Sch01] Unter dem Namen ”Virge” brachte dann S3 1996 ihren ersten Grafikchip mit 3D-Beschleuniger heraus. Dieser war allerdings ebenfalls noch nicht ausgereift und brachte keine wirkliche Beschleunigung - eher das Gegenteil war der Fall. Bei Verwendung von bilinearer Filterung[5] traten erhebliche Geschwindigkeitsverluste auf. Ein Berechnung über die CPU lief schneller. Dasselbe Problem trat auch bei Karten mit dem Rage-Chip der Firma ATI[6] auf. Aufgrund dieser Schwierigkeiten wurde die 3D-Beschleunigung bereits abgeschrieben. [MeG] Doch ein kleines Unternehmen namens 3dfx[7] entwickelte mit den

[4] Das Unternehmen produziert heute unter dem Namen ”S3 Graphics”. http://www.s3graphics.com/en/company/index.aspx

[5] Ein Grafikfilter. Weitere Infos unter http://alt.3dcenter.org/artikel/grafikfilter/

[6] ehem. Grafikchiphersteller, wurde im Jahr 2006 von der Firma AMD übernommen. Aktuelle Firmenwebsite: http://ati.amd.com

[7] ehem. Grafikchiphersteller, wurde im Jahr 2000 vom Konkurrenten Nvidia übernommen, Informationen unter: http://www.historycorner.de/3dfx1.html

Voodoo-Chips die ersten brauchbaren 3D-Beschleuniger. Diese berechneten 3D Grafiken mittels *Polygon*en. Damit war eine einfachere Programmierung möglich. Allerdings besaß der Chip keine 2D - Berechnungsfunktionen, weshalb zusätzlich noch eine 2D - Karte benutzt werden musste. Genau dieses Manko konnte Nvidia zum Vorteil nutzen. Das Modell Riva128, dass auf dem NV3 Chip basierte, kombinierte 2D und 3D Berechnungsfunktionen in einer Karte. Es verfügte über neuste Beschleunigungsfunktionen, sowie einer 128bit Speicheranbindung. Und obwohl 3dfx weiterhin bessere Ergebnisse erzielte, wurden Nvidia und auch ATI Produkte immer populärer. Dies lag vor allem an geschickter Vermarktung. Die Karten wurden an *OEM*-PC-Hersteller günstig verkauft. In Komplettrechnern befanden sich deshalb fast nur Grafikkarten der beiden Hersteller. [MeG] Den endgültigen Schritt zum Marktführer schaffte Nvidia mit Einführung der Geforce256.

2.1.3 Die erste GPU

Das Grafik-Pipeline Modell

An dieser Stelle soll nun kurz beschrieben werden, wie die berechneten Bilddaten auf den Bildschirm gelangen. Diesen Vorgang bezeichnet man als Bildsynthese und speziell im 3D-Bereich als Rendern (engl. *to render* = wiedergeben). Es existieren die beiden Arten Echtzeit - und Realistisches Rendern. Beim Letztgenannten wird vor allem auf physikalische Korrektheit und realistische Beleuchtung wert gelegt. Das Endergebnis ist wichtiger als die Berechnungsgeschwindigkeit. Der Verwendungszweck umfasst dabei die Visualisierung physikalischer Modelle, sowie die Erstellung von Bildern und Filmen. Beim Echtzeitrendern jedoch müssen Änderungen im Bild nach Interaktion mit dem Nutzer schnell, also quasi in Echtzeit, dargestellt werden. Angewendet wird dies vor allem bei wissenschaftlichen Animationen, CAD-Programmen oder Computerspielen. Für diesen Vorgang existiert ein Modell namens Grafik-Pipeline, welches die dafür notwendigen Schritte beschreibt. Pipeline bedeutet soviel wie Leitung oder Kanal, die Daten werden praktisch von einem Schritt zum nächsten weitergeleitet. In der einfachsten Form lässt sie sich in die 3 abstrakten Schritte Anwendung, Geometrie und Rasterung einteilen, wobei jeder Schritt wieder eine Pipeline aus mehreren Teilschritten beinhaltet. [MHH08] Der Schritt Anwendung wird in Software, d.h. auf der CPU ausgeführt. Bei einem Computerspiel wird dabei z.B. auf Benutzereingaben reagiert und bei darauf folgenden Änderungen an einer Szene, diese an den nächsten Schritt übertragen. Ergeben sich daraufhin u.U. Soundeffekte, werden die akustischen Signale an entsprechende Ausgabegeräte weitergeleitet. Auch mögliche Animationen bzw. programmierte Bewegungsabläufe der Spielfiguren werden hier berechnet. Im darauf folgenden Schritt Geometrie, wird die 3D-Szene mit den beinhaltenden Objekten durch verschiedene ”Koordinationsräume”, welche die Berechnung vereinfachen, transformiert. Am Ende ist dann ein für die Ausgabe notwendiges 2D-Bild vorhanden. Während des Transformierens, wird mit den Techniken *Clipping* und *Culling* geprüft, welche 3D - Objekte (oder

Teile dieser) sichtbar sind. Alles was im aktuellen Bild nicht gesehen werden kann, wird entfernt. Der dritte Schritt Rasterung konvertiert das 2D-Bild in Pixel, berechnet die entsprechenden Farbwerte und leitet es an das Ausgabegerät, meistens ein Bildschirm, weiter. Überlagern sich Objekte, erfolgt durch *Z-Buffering* die Ermittlung des Obersten.

Entlastung der CPU

All diesen Vorgängen liegen aufwändige Algorithmen zugrunde. Die dabei rechenintensivsten befinden sich im zweiten Schritt Geometrie. Bis zu diesem Zeitpunkt wurden diese von der CPU berechnet und nur der letzte Schritt wurde von der Grafikkarte übernommen. Dies änderte sich mit Einführung der Geforce 256. Der zugrunde liegende NV10 Chip war, neben dem Savage 3D Chip von S3, der Erste, der mit einer *Transform & Lighting* Recheneinheit ausgestattet war. [MeG] Nvidia führte dafür den Begriff GPU ein und definierte ihn folgendermaßen:

> "...ein Einzelchip-Prozessor mit integrierten Transform-, Lighting-, Triangle-Setup/-Clipping und Rendering-Engines, der in der Lage ist ein Minimum von 10 Millionen Polygonen in der Sekunde zu verarbeiten." [G25]

Damit konnten nun die im Schritt Geometrie beinhaltenden Pipeline-Schritte unabhängig von der CPU berechnet werden. Abbildung 2.2 stellt diese Neuerung grafisch dar. Links wird die bisherige Unterteilung der Grafikpipeline in CPU/GPU Berechnung dargestellt, rechts nach Einführung der T & L Einheit.

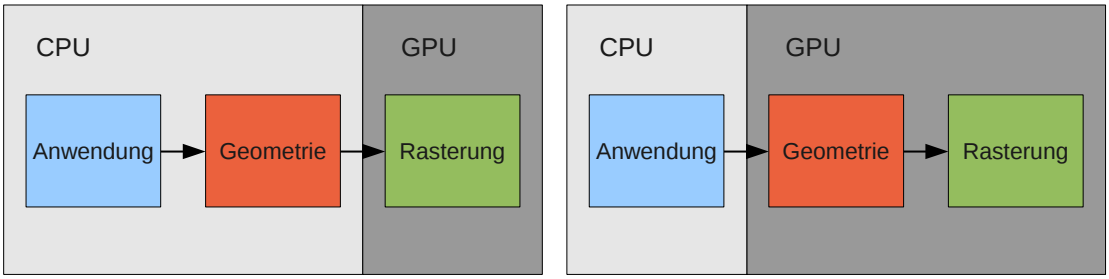

Abbildung 2.2: Vergleich Echtzeitrenderingpipeline vor und nach Einführung der T & L Einheit

Die T & L Einheit rechnete mit einer Genauigkeit von 32 Bit Fließkommazahlen und konnte mit den Schnittstellen *OpenGL* und *DirectX* programmiert werden. [ND10] Mit der neuen Genauigkeit wurde nun, neben einer verbesserten Berechnung der 3D Geometrie, *High Dynamic Range* Rendering möglich. Damit konnten Farbübergänge wesentlich genauer und realistischer dargestellt werden. Wirklich nutzbar wurden die neuen Beschleunigungsfunktionen allerdings erst mit dem Geforce 256 DDR Modell, bei welcher der Zugriff auf den Grafikspeicher nun mittels *DDR* erfolgte. Zuvor bremste die geringe Bandbreite den neu gewonnenen Geschwindigkeitsvorteil aus. [MeG] Die

Recheneinheit bestand allerdings immer noch aus fest in Hardware implementierten Funktionen. Von flexibler Programmierung konnte man noch nicht sprechen.

2.1.4 Eine wichtige Neuerung - Die Shader

Mit der Geforce 3 präsentierte Nvidia ein neues Konzept. Kleine Recheneinheiten, sogenannte Shader sind seitdem für das Rendern verantwortlich. [MeG] Das äquivalente Konkurrenzprodukt von ATI war die Radeon 8500. Für alle Berechnungen die auf der Geometrie (Eckpunkte, Kanten, Polygone) von Objekten ausgeführt werden, wurden die Vertex-Shader, für alle auf Pixel bezogenen die Pixel-Shader eingeführt. Der große Vorteil besteht darin, dass nun für jeden einzelnen Eckpunkt (engl. Vertex) bzw. jeden Pixel separate Programme geschrieben werden können. Dadurch ist eine sehr flexible und vor allem sehr gut parallelisierbare Programmierung möglich. Im Jahr 2006 kamen zusätzlich noch die Geometrie-Shader hinzu. [ND10] Diese können zusätzlich zu den Vertex-Shadern, die Geometrie einer Szene noch sehr flexibel bearbeiten und z.B. sogar neue Polygone hinzufügen. [PS1] Trotz dieser neu gewonnenen Möglichkeiten, gab es noch einen Mangel: die Shader waren nicht gleichmäßig ausgelastet. Pixel-Shader müssen meistens mehr Daten verarbeiten als Vertex-Shader, da in der Regel mehr Pixel als Eckpunkte vorhanden sind. [ES] Aus diesem Grund wurde die hardwareseitige Unterteilung der Shader schließlich ganz aufgehoben. Das neue Modell nennt sich Unified-Shader. Damit ist nun jeder Shader in der Lage, Berechnungen aller Art durchzuführen. Es obliegt dem Programmierer, welche Aufgaben er welchem Shader zuteilt. [US]

2.2 Die moderne GPU

Anhand zweier Grafikkartenmodelle der beiden Hersteller Nvidia und AMD, soll nun die Architektur und Funktionsweise der GPU ausführlich erläutert werden. Gleichzeitig wird ein Vergleich mit der CPU durchgeführt, um alle Unterschiede und Gemeinsamkeiten darzustellen. Die Betrachtungen beziehen sich auf die Grafikchips:

- GF100 von Nvidia
- Cypress (RV870) von AMD

2.2.1 Vergleich GPU und CPU

Die Entstehungsgeschichte der GPU macht deutlich, dass sie durch die Spezialisierung auf das Rendern, nach einem völlig anderem Prinzip konstruiert ist als die CPU. Eine CPU versucht grob gesagt, ein einzelnes Programm so schnell wie möglich auszuführen, während eine GPU viele ”kleine” Programme zur gleichen Zeit ausführt. Die Angaben im folgenden Absatz sind größtenteils aus [GPG] entnommen.

Parallelität bei der CPU

In einer modernen CPU wird Parallelität unter anderem mittels ILP (Instruction Level Parallelism) erreicht. Damit wird versucht, unabhängige Instruktionen in einem Programm zu finden und diese gleichzeitig auszuführen. Man bezeichnet CPUs mit dieser Technik als superskalar [CiZ]. Allerdings ist die für weitere Performancesteigerungen nötige Kontrolllogik im Vergleich zum Gewinn mittlerweile sehr aufwändig geworden. [GPG] Ein weiterer Weg sind natürlich die Mehrkernprozessoren, welche TLP (Thread Level Parallelism), also die Ausführung mehrerer Threads oder Programme parallel, ausnutzen. Außerdem wird, wenn möglich, auf DLP (Data Level Parallelism) zurückgegriffen. Auf mehrere Daten wird die gleiche Operation angewendet. Es wird ersichtlich, dass das Design der CPU auf die Performance eines einzelnen Threads ausgelegt ist. [GPG] **Ein** Thread wird so **schnell** wie möglich ausgeführt.

Parallelität bei der GPU

Bei der GPU hingegen ist nur wichtig, wie lang die Berechnung eines kompletten Bildes dauert. Die Ausführungsgeschwindigkeit der einzelnen Berechnungen ist dagegen eher egal. Es zählt nur das Endresultat, dass aus vielen voneinander unabhängigen Berechnungen über die Shader entsteht. Der Gesamtdurchsatz ist wichtig. Beim Pipelineschritt Rasterung berechnen z.B. immer mehrere Shader gleichzeitig und unabhängig voneinander Pixel. Sind für einen Pixel noch berechnete Daten aus dem Speicher notwendig, wird einfach ein anderer berechnet. Dadurch kann man auch auf große Caches verzichten, denn die Shader werden ja nicht blockiert bis ein Wert vorhanden ist. Zusammengefasst: **Viele** Threads werden **langsam parallel** ausgeführt.

Tabelle 2.1 stellt eine Gegenüberstellung der Prozessoren anhand von Kenndaten[8] dar. Verglichen wird ein Chip aus der Intel Core i7, i5, i3 2xxx Modellfamilie (Codename Sandy Bridge) mit den Grafikchips GF100 (Nvidia) und RV870 (AMD).

[8] entnommen aus [SaB] [Anda] [RV8] [wkG] [wkR]

	Sandy Bridge	**GF100**	**RV870**
Die-Größe	131 bis 216 mm^2	520 mm^2	389 mm^2
Transistorenanzahl	1,16 Milliarden	3 Milliarden	2,92 Milliarden
Fertigungstechnik	32 nm	40 nm	40 nm
Kernanzahl [a]	4	16	20
Integereinheiten (ALU)	12	512	1600
Gleitkommaeinheiten (FPU)	12	in Integereinheiten integriert	in Integereinheiten integriert
Ausführung Integer und Gleitkommapipelines	getrennt, bei Verwendung von SSE/AVX gemeinsam	gemeinsam	gemeinsam
L1-Cache	512 KB pro Kern	64 KB pro Kern	2 Speicher jeweils 8 KB + 32 KB pro Kern
L2-Cache	2048 KB	768 KB	4 x 128 KB sowie 64 KB pro Kern, insgesamt 1792 KB
L3-Cache	max. 8 MB	-	-
Speichercontroller	2 x 64 Bit DDR3 (21,2 GB/s bei 1333 MHz)	384 Bit GDDR5 (177,4 GB/s bei 924 MHz)	256 Bit GDDR5 (153,6 GB/s bei 1200 MHz)

[a] Die Herstellerbezeichnung ”Stream Processing Unit” (AMD) und ”CUDA Core” (Nvidia) ist nicht mit dem Kern einer CPU gleichzusetzen. Um den Begriff auf einen Nenner zu bringen, wurde die Kernanzahl nach wirklich unabhängigen Recheneinheiten berechnet.

Tabelle 2.1: Kenndaten CPU und GPU

Die in der Tabelle zu sehenden Differenzen, geben die unterschiedlichen Architekturen wieder. Besonders in der Anzahl der Kerne und der Größe der Caches zeigen sich die Unterschiede.

Des weiteren fällt auch auf, dass selbst die beiden Grafikchips Unterschiede aufweisen. Den Prozessoren liegen verschiedene Architekturen zugrunde. Der GF100 Chip von Nvidia ist nach der ”Fermi-Architektur” konstruiert, das Konkurrenzprodukt von AMD (RV870) verwendet die ”Terascale 2 Architektur”. Ein direkter Vergleich anhand der Kenndaten ist schwierig. Im folgenden werden die beiden Architekturen deshalb ausführlicher erläutert.

2.2.2 Die Fermi-Architektur

Wie bereits in Tabelle 2.1 zu sehen, befinden sich 16 Kerne auf dem Chip. Genauer gesagt, 4 Graphic Processing Cluster (kurz GPC) mit jeweils 4 Streaming Processors (kurz SM). Die folgende Grafik zeigt ein Blockdiagramm des GF100 Chips.

Abbildung 2.3: Blockdiagramm Fermi Architektur [Anda]

Jeder SM enthält wiederum 32 Ausführungseinheiten, die von Nvidia Cuda Cores genannt werden. Die Bezeichnung Core ist allerdings irreführend, denn ein Cuda Core entspricht nicht dem Begriff Core einer CPU. Ein CPU Kern ist ein selbstständiger Mikroprozessor mit eigenem Register, der unabhängig von anderen Kernen Daten liest und schreibt und für Berechnungen auf mehrere ALUs und ggf. FPUs zurückgreift. Ein Cuda Core enthält ”nur” eine einzelne ALU, sowie eine FPU. Von daher passt der Begriff Kern eher zu einem SM. Denn dieser besitzt, neben den Cuda Cores, ein eigenes Register, sowie 16 Load-Store-Einheiten, die Quell- und Zieladressen in Cache und VRAM speichern. Außerdem 4 Special Function Units (SFUs), die z.B. für Sinus und Cosinus Berechnung verwendet werden. Abbildung 2.4 zeigt ein Blockdiagramm eines einzelnen SM.

Jeder Cuda Core kann pro Takt eine FMA (Fused Multiply ADD) Operation durchführen. Damit können die Werte in einer hohen Genauigkeit berechnet werden, es ist vollständige IEEE 754-2008-Standard[9] Kompatibilität vorhanden. Auf dem Chip befinden sich zusätzlich 48 ROPs (Raster Operation Processors), sowie 64 TMUs (Texture Mapping

[9] Eine Norm für die Darstellung von Gleitkommazahlen

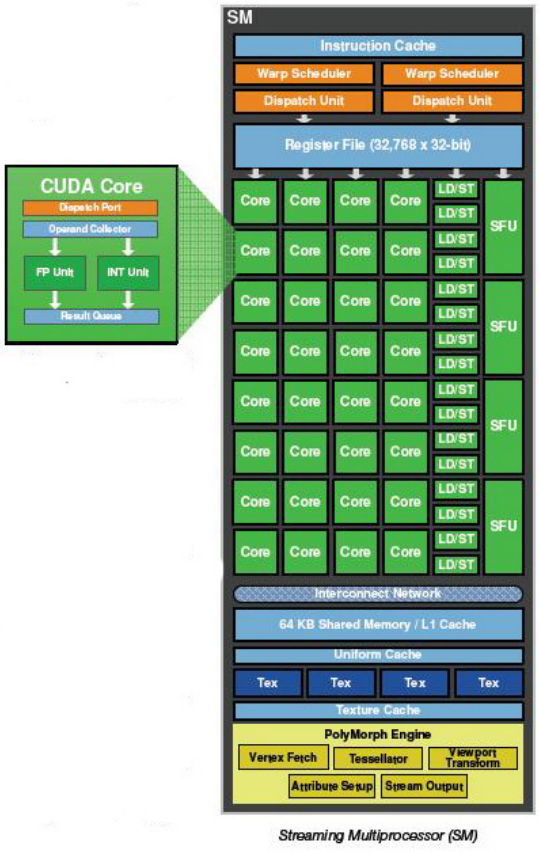

Abbildung 2.4: Blockdiagramm Streaming Processor [Anda]

Units). Die zuerst genannten werden für Veränderungen am fertigen Bild (z.B. Antialiasing) genutzt, die letzteren für Texturberechnungen.

Damit die Daten auf den Chip gelangen, existiert ein Host-Interface, dass die Befehle der CPU abfängt. Anschließend kümmert sich ein Verwaltungssystem namens ”GigaThread Engine” um die weitere Verarbeitung. [Anda] Die Daten[10] werden aus dem Systemspeicher in einen Frame Buffer geladen, in Blöcke geteilt, und an die SMs weitergeleitet. Dort werden die Daten nochmals in Gruppen von jeweils 32 aufgespalten. So eine Gruppe bezeichnet Nvidia als Warp. Um die Effektivität noch weiter zu steigern, unterstützt jeder SM Multithreading. 48 dieser Warps können zur gleichen Zeit ausgeführt werden. [Anda] ”Wartet” ein Warp auf Daten aus dem Speicher, wird ein anderer Warp verwendet, welcher alle Daten zur Verfügung hat. Alle Warps teilen sich dabei, wie in Abbildung 2.4 zu sehen, einen gemeinsamen Registerspeicher. Rechnet man alles zusammen, ergibt sich ein Wert von insgesamt 24.576 Datenelementen. Jeder SM besitzt 2 Warp Scheduler und 2 sogenannte ”Instruction Dispatch Units”. Die Scheduler verwalten jeweils ein Warp. Für 16 Datenelemente können Quell- und Zieladressen

[10] Sowohl Nvidia als auch AMD verwenden für ein einzelnes Datenelement auch den Begriff Thread. Zum besseren Verständnis, wird aber weiterhin die Bezeichnung Datenelement verwendet

über die Load/Store Einheiten zur Verfügung gestellt und anschließend berechnet werden. Wie bereits erwähnt, besitzt ein SM aber 32 Ausführungseinheiten und 4 SFUs, es wären also entweder nur die Hälfte der Kerne oder die SFUs in Betrieb. Anhand der Zahlen könnte man eine niedrige Auslastung vermuten. Allerdings existiert ein Register, welches die Dispatch Units nutzen um die Ausführungseinheiten, die Load/Store und die SF Einheiten zu verwalten. So kann, z.B. während für ein Warp die Adressen geladen werden, ein anderes bereits die Cuda Cores nutzen oder aus den 2 parallelen Warps auf jeweils 16 Datenelemente auf den Cuda Cores Befehle ausgeführt werden. Das geschieht jeweils in einem Takt. Anders verhält es sich mit den SFUs. Hier werden für die Berechnung eines Warps 8 Takte benötigt. [Anda] Der Einsatz der Dispatch Units ist also gerade hierfür besonders erforderlich, da in der Zeit ein anderes Warp die verfügbaren Teile der SM nutzen kann. Zusammen mit den Schedulern, d die Warps verwalten, ist also eine gute Auslastung gegeben. Abbildung 2.5 veranschaulicht das Verhalten der Warp Scheduler.

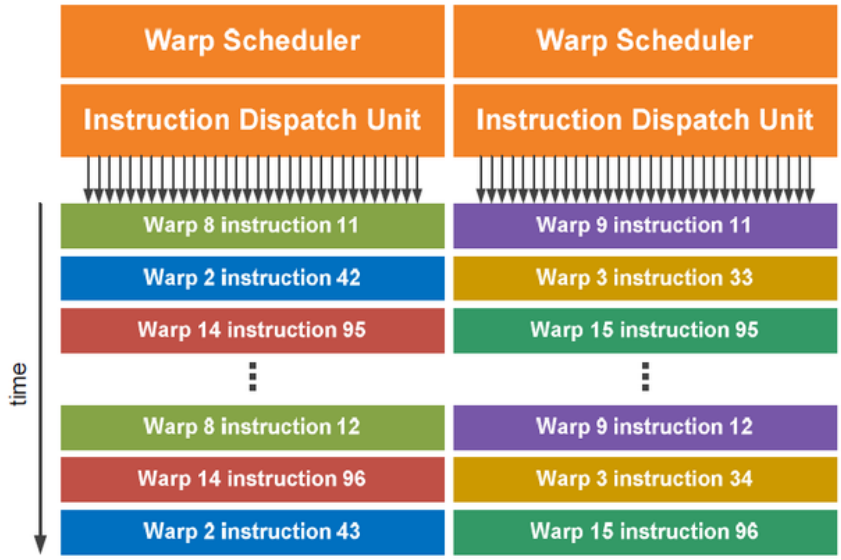

Abbildung 2.5: Verhalten des Warp Schedulers [Anda]

In Tabelle 2.2 wird die Anzahl der Operationen pro Takt, abhängig von der Komplexität der verwendeten Zahlenwerte bzw. der Berechnung gegenübergestellt.

	FP32	FP64	INT	SFU	LD/ST
Operationen pro Takt	32	16	32	4	16

Tabelle 2.2: Anzahl der Operationen pro Takt eines einzelnen Stream Multiprocessors [Smi]

Bei Verwendung ganzer Zahlen (INT), sowie Gleitkommazahlen mit einfacher Genauigkeit (FP32), können doppelt soviel Werte als bei Gleitzahlen mit doppelter Genauigkeit (FP64) berechnet werden. Ein Warp, das nur aus FP64 Zahlen besteht, braucht also

2 Takte für die komplette Berechnung. Werden nur die SFU Einheiten benutzt, werden wie bereits erwähnt 8 Takte benötigt.

Speicher

Jeder SM verfügt über einen Shared-Memory und einen L1-Cache. Der Shared Memory wird für Algorithmen mit klar definiertem Speicherzugriff verwendet, der L1-Cache für das Gegenteil (unbekanntes Speicherziel). Die Speicher sind zusammen 64 KB, aufgeteilt in 48 und 12 KB, groß. Die Verteilung ist dabei frei konfigurierbar, entweder L1-Cache 48 KB und Shared Memory 12 KB oder umgedreht. Zusätzlich existiert noch ein 12 KB Cache speziell für Texturen. Für alle SMs zusammen existiert ein 768 KB großer L2-Cache auf den ein gleichzeitiger Zugriff für Load/Store Anfragen möglich ist. [Anda]

2.2.3 Die Terascale 2 Architektur

Die Architektur von AMD enthält 20 SIMD-Einheiten, die wiederum 16 SPUs (Stream Processing Units) mit jeweils 5 ALUs enthalten. Eine einzelne SIMD-Einheit lässt sich dabei am ehesten mit einem CPU-Kern gleichsetzen. Abbildung 2.6 zeigt die Terrascale 2 Architektur auf dem RV870 Chip. Im Gegensatz zu Nvidia, verwenden die ALUs

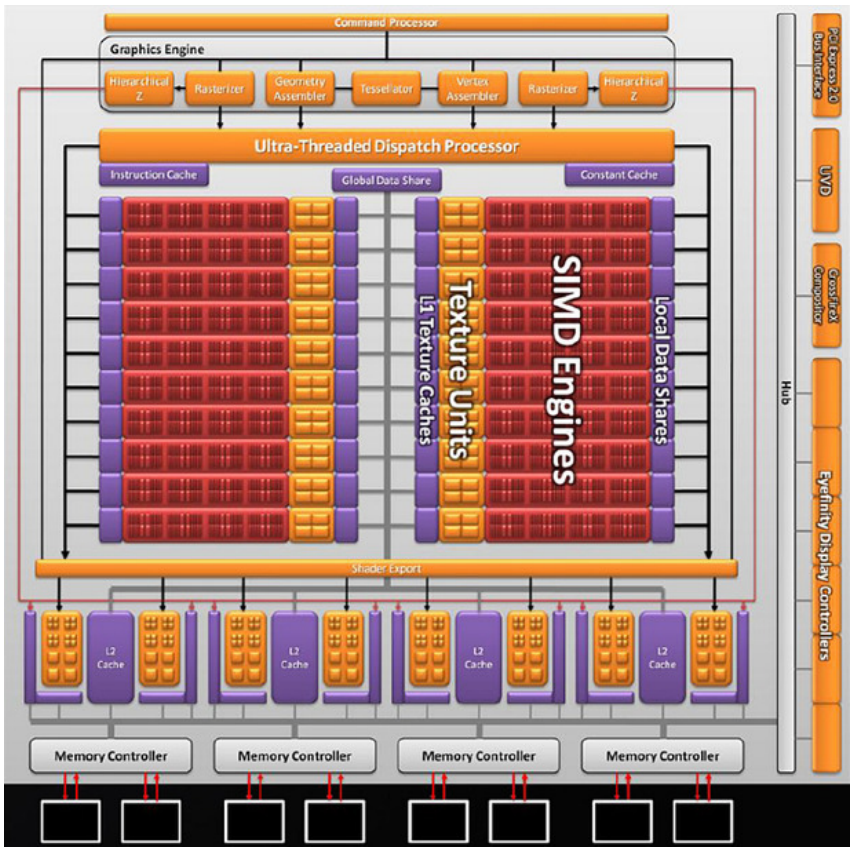

Abbildung 2.6: Blockdiagramm RV870 Chip [RV8]

MADD (Multiply-Addition) anstatt FMA. Die Ergebnisse liegen also in einer etwas geringeren Genauigkeit vor. Eine der 5 ALUs kann dabei als SFU agieren. Um die einzelnen ALUs als skalare Einheiten zu verwenden, müssen die Berechnungen unabhängig voneinander sein. Ist dies nicht der Fall, werden nicht alle ALUs ausgelastet. In jeder SPU gibt es außerdem eine Branch-Execution-Einheit, die durch vorzeitiges Erkennen von ”if” oder ”while” Konstrukten für eine hohe Auslastung der ALUs sorgt. Zusätzlich existiert ein ”General Purpose Register”, dass als kleiner Zwischenspeicher besonders für das GPU-Computing nützlich ist. [Andb] In Abbildung 2.7 ist eine einzelne SPU mit den beinhalteten Einheiten zu sehen. Jedes ALU kann pro Takt ein MADD mit FP32 Genauigkeit durchführen. Bei FP64 Werten werden vier ALUs für ein MADD benötigt. Die theoretische Rechenleistung beträgt bei einfacher Genauigkeit 2,72 TFLOPs. Da bei doppelter Genauigkeit 4 ALUs verwendet werden, reduziert sich die Rechenleistung für FP64 demzufolge auf ein Fünftel, sie beträgt 544 GFLOPs. [Andb]

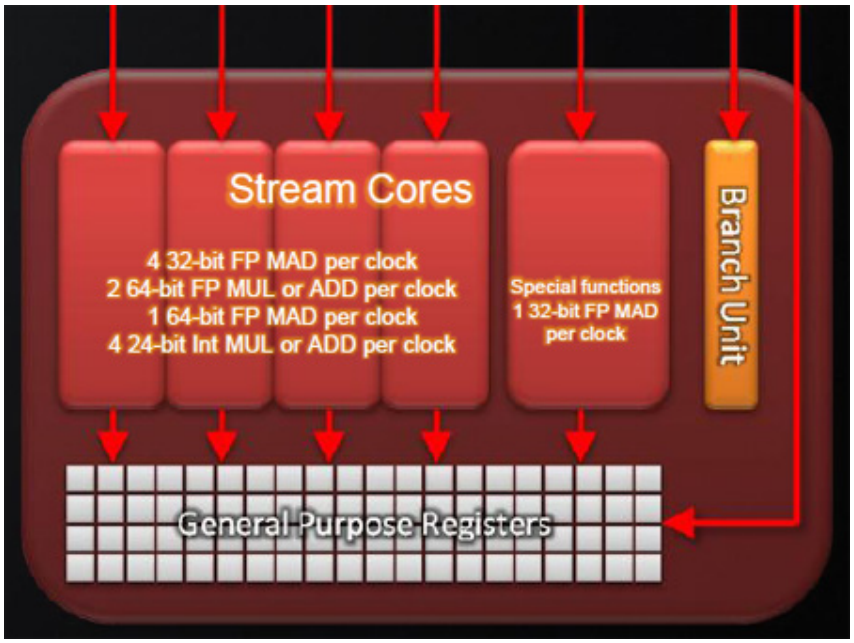

Abbildung 2.7: Blockdiagramm SPU [RV8]

Jede SIMD-Einheit besitzt einen 32 KB großen ”Local Data Share” in welchem Ergebnisse zwischengespeichert werden können, sowie einen 8 KB L1 Texture Cache, speziell für Texturen. Außerdem einen 64 KB ”Global Data Share” auf den zusätzlich alle anderen SIMD-Einheiten zugreifen können. Schließlich noch vier 128 KB große L2-Caches.

Über einen Commandprozessor erhält der Chip Befehle von der CPU. Diese werden an eine ”Graphics Engine” weitergeleitet, in welcher bei Verwendung von Grafikdaten bereits Anpassungen und Vorberechnungen vorgenommen werden. Ein sogenannter ”Ultra-Threaded Dispatch Processor” kümmert sich anschließend um die Datenversorgung der Recheneinheiten und sorgt zudem für eine hohe Auslastung der einzelnen SPUs. [Andb]

2.2.4 Zusammenfassung

Nvdia baut auf eine superskalare Architektur, besitzt also fest in Hardware implementierte Kontroll- und Optimierungsstrukturen. Die Folgen davon sind weniger Platz auf dem Chip und eine niedrigere Taktrate. AMD hingegen optimiert mittels *VLIW* (Very Long Instruction Word). Es wird also kein zusätzlicher Platz auf dem Chip benötigt. Daraus resultieren die unterschiedlichen ALU Zahlen und Taktfrequenzen.

Bei der Berechnung von Grafikdaten zeigen die beiden Architekturen ähnliche Leistungen. [Ben] Allerdings gibt es Unterschiede, wenn der Chip für andere Aufgaben eingesetzt wird. Deutlich sieht man das z.B. bei der Bitcoin Berechnung [Bit]. Hier wirken sich

höhere ALU- und Taktzahl fast potentiell auf die Geschwindigkeitssteigerung aus. Die Verwendung von AMD GPUs, kann eine Steigerung von 200-300 Prozent bewirken.

Das nächste Kapitel geht deutlicher auf die Besonderheiten der GPU und damit verbunden der Programmierung ein.

3 GPU-Computing

Aus der im vorhergehenden Kapitel dargestellten GPU Architektur wird ersichtlich, dass sich hinter einer GPU ein hohes Leistungspotential verbirgt. Dieses beruht vor allem auf drei charakteristischen Merkmalen. Frei übersetzt nach [OHL+08]:

- *hohe Rechenleistung:* Echtzeit-Rendering fordert Milliarden Pixel pro Sekunde, wobei für jedes Pixel mehr als 100 Operationen möglich sind. Eine hohe Rechenleistung muss demzufolge gewährleistet sein, um diese Anforderungen zu erfüllen.
- *Parallelität:* Beim Abarbeiten der Grafikpipeline, werden Pixel und Eckpunkte gleichzeitig von vielen Shadern berechnet, um eine schnelle Bildgenerierung zu ermöglichen. Die Daten werden also parallel berechnet.
- *hoher Datendurchsatz:* Das menschliche Auge erfasst Bildänderungen im Millisekundenbereich, während GPU Operationen im Nanobereich ablaufen. Daraus folgt, dass die Latenzen zwischen einzelnen Berechnungen unwichtig sind. Mehrere tausend Durchläufe eines Befehl sind also möglich. Es existieren außerdem keine (bzw. wenige) Kontrollstrukturen, damit erhöht sich der Datendurchsatz zusätzlich.

3.1 Definition - zugrundeliegendes Konzept

Bereits seit einiger Zeit[11] wurde, vor allem aufgrund dieser Merkmale, versucht die GPU auch für allgemeine Rechneranwendungen zu nutzen. Dadurch entstand der Begriff GPGPU (General Purpose Computing on GPUs, zu deutsch etwa ”allgemeine Berechnungen auf Grafikprozessoren”). Allerdings konnte man bei der Programmierung nur auf eine der sogenannten Shadersprachen GLSL[12], HLSL[13] oder Cg[14] zurückgreifen. Die Problemstellung musste erst in ”Grafikdaten”, bestehend aus z.B. Pixelfarben oder Eckpunkten, umformuliert werden. Dies erforderte zusätzliches Wissen in den genannten Sprachen und zudem einen erhöhten Arbeitsaufwand. Die Leistungsfähigkeit der GPU war unter diesen Umständen nur sehr schwer nutzbar.

Dieses Problem wurde erkannt und es wurden Lösungen geschaffen, um die Programmierung zu vereinfachen und damit einer breiteren Masse an Entwicklern zugänglich zu machen. Nvidia machte den Anfang und veröffentlichte CUDA (Compute Unified Device Architecture). Die Firma Apple entwickelte in Zusammenarbeit mit den Firmen AMD, IBM, Intel sowie Nvidia den OpenCL (Open Computing Language) Standard und reichte

[11] in den Jahren 1999-2000, Quelle: http://www.nvidia.de/page/gpu_computing.html
[12] OpenGL Shading Language
[13] High Level Shading Language (Komponente von DirectX)
[14] C for graphics

ihn bei der Khronos Group[15] ein. Bei beiden Schnittstellen handelt es sich um Abwandlungen der Programmiersprache C. Als dritte Möglichkeit existiert Direct Compute von Microsoft, welches seit der 10ner Version in der Grafikschnittstelle DirectX integriert ist. Eine ausführliche Erläuterung der beiden Erstgenannten erfolgt in Kapitel 4.

3.2 Berechnungen auf der GPU - grundlegende Vorgehensweise

Mithilfe dieser beiden Schnittstellen ist es nun möglich, die GPU gänzlich ohne Kenntnisse in der Grafikprogrammierung für Berechnungen zu nutzen. Aufgrund der hardwareseitigen Unterschiede zwischen CPU und GPU gestaltet sich die Herangehensweise allerdings etwas anders. Bei dem Entwurf von Algorithmen auf der CPU, ist man normalerweise auf ein einzelnes Datenelement (z.B. ein Integer Wert) fixiert und versucht die Berechnung dieses Elements möglichst effizient umzusetzen. Der Compiler kann dann unter Umständen erkennen, ob bestimmte Teile des Algorithmus parallelisiert ausgeführt werden können. Als Beispiel sei hier z.B. eine Schleife genannt, in der für jedes Element eines Arrays die gleichen Berechnungen durchgeführt werden. Listing 3.1 zeigt eine Pseudocode Implementierung eines Algorithmus in dem zu jedem Wert eines Arrays der Wert 1 addiert wird.

Listing 3.1: Pseudo CPU-Code

```
1 Array[N] //Array mit Groesse N anlegen
2 von x = 0 bis N-1
3 {
4   Array[x] = Array[x] + 1
5 }
```

Bei der GPU sieht das etwas anders aus. Zwar entwirft man im einfachsten Fall auch hier einen Algorithmus entsprechend eines einzelnen Datenelements. Beim Aufruf übergibt man aber nicht ein einzelnes Datenelement sondern ein ganzes Array. Entsprechend der GPU-Hardware mit ihren vielen ALUs oder SIMD Einheiten, wird nun für alle Elemente der Algorithmus parallel durchgeführt. Kontrolle über die Reihenfolge der Abarbeitung der Elemente hat man als Programmierer nicht. Die Elemente dürfen deshalb keine (bzw. nur minimale) Abhängigkeiten voneinander haben. Listing 3.2 zeigt eine Pseudeocode Implementierung des oben beschriebenen Algorithmus auf der GPU.

Listing 3.2: Pseudo GPU-Code

```
1 // Berechnung welche auf der GPU ausgefuehrt wird
2 gpuMethode(Array[]) {
3     Array[N] = Array[N] + 1
4 }
```

[15] Ein Industriekonsortium, welches sich für offene Standards im Multimediabereich einsetzt, siehe http://www.khronos.org/about/

```
5
6 // CPU Code welcher den Ablauf steuert
7 Array[N]                  //Array mit Groesse N anlegen
8 gpuMethode(Array[N])      // Auruf der auf der GPU auszufuehrenden Methode
```

Grob zusammengefasst, werden also einzelne Methoden gekennzeichnet, welche dann auf der GPU ausführt werden. Der restliche Programmcode wird ganz ”normal” von der CPU verarbeitet. Auf diese Weise ist eine einfache Kombination von CPU und GPU Berechnungen möglich. Die Einarbeitungszeit ist für den fortgeschrittenen Programmierer recht gering, da neben den Besonderheiten der GPU nur einige neue Bezeichner und Funktionen hinzukommen.

4 Schnittstellen

4.1 Überblick

In Kapitel 3 wurde beschrieben, dass mithilfe aktueller Programmierschnittstellen (geläufiger ist der Begriff API[16], der im folgenden Teil der Arbeit verwendet wird) die Nutzung der GPU, ohne Kenntnisse in der Grafikprogrammierung, möglich ist. Zur Auswahl stehen CUDA, OpenCL und Direct Compute. Jede besitzt ihre eigenen Vor- und Nachteile, sodass die Wahl von verschiedenen Faktoren abhängt. Im Folgenden werden nun die beiden erstgenannten APIs und ihre wichtigsten Funktionen, anhand von Beispielen, vorgestellt. Eine Betrachtung von Direct Compute würde den Umfang dieser Arbeit überschreiten, es wird deshalb nicht weiter erläutert.

4.2 CUDA

CUDA bezeichnet einerseits die parallele Rechnerarchitektur der neueren[17] Nvidia Grafikkarten, andererseits unter der erweiterten Bezeichnung ”CUDA C” die entsprechende API mit Funktionen zum Zugriff auf die GPU. Dabei existieren 2 Versionen: Driver API und Runtime API. Die erstgenannte ist hardwarenäher (bietet also etwas mehr ”Kontrolle” und einige zusätzliche Funktionen), erfordert aber mehr API-Aufrufe, was wiederum zu mehr Quellcode und höherer Fehleranfälligkeit führt. Aus diesen Gründen, und da die Runtime API intern auf die Methoden der Driver API zugreift, wird im weiteren Teil der Arbeit auf die Runtime API zurückgegriffen.

4.2.1 Einrichtung

Vor der ersten Verwendung von CUDA C ist die Einrichtung einer entsprechenden Entwicklungsumgebung vonnöten. Zu dieser gehört:

- Eine CUDA-fähige Grafikkarte
- Ein NVIDIA Grafiktreiber
- Das CUDA Toolkit
- Ein C Compiler

CUDA-fähige Grafikkarte: Die erste Grafikkarte mit CUDA Unterstützung war die Geforce 8800 GTX, die im Jahr 2006 erschien. Alle nachfolgenden Produkte von Nvidia ver-

[16] engl. ”application programming interface”
[17] siehe Absatz Einrichtung

wenden die CUDA Architektur. Unter http://www.nvidia.de/object/cuda_gpus_de.html befindet sich eine aktuelle Liste aller unterstützten Modelle.

NVIDIA Grafiktreiber: Der aktuelle Grafiktreiber kann unter http://www.nvidia.de/Download/index.aspx?lang=de heruntergeladen werden.

CUDA Toolkit: Das CUDA Toolkit enthält den CUDA Compiler, Bibliotheken und Dokumentationen. Der Download ist unter http://developer.nvidia.com/cuda-downloads verfügbar.

C Compiler: Neben dem CUDA Compiler, welcher den GPU Code kompiliert, wird auch ein Compiler für CPU Code benötigt. Dieser ist abhängig vom verwendeten Betriebssystem:

- Windows: Es bietet sich der Visual Studio C Compiler an. Dieser ist in allen Visual Studio Versionen, inklusive der kostenlosen Express Variante, enthalten. Weitere Informationen gibt es unter http://www.microsoft.com/germany/visualstudio/
- Linux: Linuxsysteme verwenden den GNU C Compiler, der in den meisten Distributionen bereits enthalten ist. Prüfen kann man dies, indem man in der Konsole ”gcc -v” eingibt. Ist der Compiler nicht vorhanden, kann man ihn in den meisten Fällen einfach über den Paketmanager installieren. Hinweis: Der aktuelle CUDA C Compiler[18] unterstützt nur gcc bis Version 4.6. Unter Umständen ist ein Downgrade erforderlich. Ein Workaround zur Verwendung mehrerer Versionen wird in Anhang A.1 beschrieben.
- Macintosh OS X: Das Einrichten einer Entwicklungsumgebung mittels eines Mac OS X Betriebssystems konnte im Rahmen dieser Arbeit nicht durchgeführt werden. Hinweise finden sich im Netz, z.B. unter http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_Getting_Started_Mac.pdf

In Anhang A.1 befinden sich erweiterte Anleitungen zur Einrichtung einer CUDA Entwicklungsumgebung im gewünschten Betriebssystem.

[18] enthalten im CUDA Toolkit 4.2

4.2.2 Das erste CUDA Programm

Nach erfolgreicher Einrichtung der Entwicklungsumgebung, werden nun anhand einer GPU-Implementierung des bekannten ”Game of Life”[19] Algorithmus, die elementaren Funktionen der CUDA-API vorgestellt. Dabei handelt es sich um einen Algorithmus, welcher in einem zweidimensionales System, Veränderungen der darin enthaltenen Objekte berechnet. Die Veränderungen resultieren aus den Abhängigkeiten der einzelnen Objekte untereinander. Das System beruht auf dem Modell des ”zellulären Automaten”[20], die Objekte des Systems werden dabei als lebendige oder tote Zellen betrachtet. Einfacher ausgedrückt, kann man das System als eine Matrix betrachten, die aus `true` und `false` Werten besteht. Ein `true` gesetzter Wert entspricht dabei einer lebenden Zelle, ein `false` Wert einer Toten. Der Algorithmus legt neue ”Zellgenerationen”, resultierend aus den Abhängigkeiten der Zellen, an:

- Eine tote Zelle, die genau 3 lebende Nachbarzellen besitzt, erwacht in der Folgegeneration wieder zum Leben
- Lebendige Zellen mit weniger als 2 und mehr als 3 lebenden Nachbarzellen sterben in der nachfolgenden Generation
- Lebendige Zellen mit 2 oder 3 lebenden Nachbarzellen bleiben lebendig

Abbildung 4.1 zeigt eine zufällig generierte Startmatrix des ”Game of Life”.

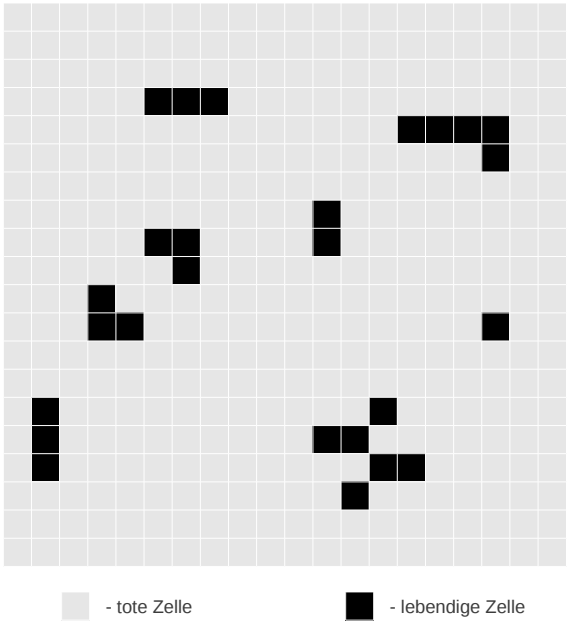

Abbildung 4.1: Zufällige Startmatrix ”Game of Life”

[19] http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens
[20] http://de.wikipedia.org/wiki/Zellulärer_Automat

CPU-Implementierung

Bevor der Algorithmus auf der GPU implementiert wird, soll zum besseren Verständnis zuerst eine einfache Variante[21] auf der CPU in Standard C Code betrachtet werden. Das komplette Beispiel befindet sich in Anhang B.

Über die beiden `for` Schleifen, wird jedes Feld des Arrays der Reihe nach abgearbeitet und die Nachfolgegeneration anhand der oben beschriebenen Bedingungen in einem neuen Array gespeichert.

```
1  for (int x = 1; x <= DIM; x++) {
2    for (int y = 1; y <= DIM; y++) {
3      neighbors = getNeighbors(array, x, y)
4      if (neighbors == 3 || (neighbors == 2 && array[x][y])) {
5        array_new[x][y] = 1;
6      } else {
7        array_new[x][y] = 0;
8      }
9    }
10 }
```

In der Methode `getNeighbors`, findet eine Prüfung der umliegenden Felder, inklusive des eigenen, statt. Ist ein ”Nachbar” vorhanden, wird die entsprechende Variable hochgezählt und anschließend die Anzahl der Nachbarn zurückgegeben.

```
1  int getNeighbors(int array[][DIM], int x, int y) {
2    int neighbors;
3    neighbors = array[x+1][y] + array[x-1][y]
4                + array[x][y+1] + array[x][y-1]
5                + array[x+1][y+1] + array[x-1][y-1]
6                + array[x-1][y+1] + array[x+1][y-1];
7    return neighbors;
8  }
```

Um eine Prüfung der ”Ränder” des Arrays zu ermöglichen, werden vier Temp-Spalten verwendet, welche den jeweils gegenüberliegenden Rand beinhalten.

```
1    //Kopieren der aeussersten Spalten zu den gegenueberliegenden Temp
       Spalten
2    for (i = 1; i<=DIM; i++) {
3       array[i][0] = array[i][DIM];
4       array[i][DIM+1] = array[i][1];
5    }
6
7    //Kopieren der aeussersten Zeilen zu den gegenueberliegenden Temp Zeilen
8    for (j = 0; j<=DIM+1; j++) {
9       array[0][j] = array[DIM][j];
```

[21] als Vorlage diente das Beispiel unter http://www.olcf.ornl.gov/training_articles/cpu-game-of-life/

```
10          array[DIM+1][j] = array[1][j];
11      }
```

GPU-Implementierung

Die Berechnung einer neuen Zellgeneration, soll nun auf die GPU ”ausgelagert” werden. Dazu muss eine Möglichkeit existieren, dem Compiler mitzuteilen, die Funktion `next_generation` zur Ausführung auf der GPU zu compilieren. CUDA verwendet dafür den Bezeichner `__global__`. Alle Funktionen, welche diese Kennzeichnung besitzen, werden mit dem nvcc Compiler compiliert und verwenden die GPU samt dem zugehörigen Speicher. Funktionen ohne spezielle Kennzeichnung, werden wie ”üblicher” C Code auf CPU und Arbeitsspeicher ausgeführt. Für eine eindeutigere Unterteilung, werden auch die Begriffe Host(CPU) und Device(GPU) verwendet. Listing C.1 in Anhang C.1 zeigt eine erste Version[22] des Game of Life unter Verwendung von CUDA. Alle verwendeten CUDA Funktionen werden im Folgenden erläutert.

Dateien mit CUDA Quellcode werden unter der Endung ”.cu” gespeichert. Genau wie ”.c” Dateien besitzen auch sie eine main Funktion, die den Einstieg ins Programm bildet. Es ist jedoch ein Aufruf einzelner CUDA Funktionen von einer C Quellcode Datei möglich. In Anhang E befindet sich eine Erweiterung dieses Beispiels, welche die Zellgeneration mittele GTK+[23] darstellt. Dort wird ein solcher Aufruf angewandt.

Die erste Funktion aus der CUDA-API befindet sich in Zeile 82 und besitzt den Namen `cudaMalloc()`. Dabei handelt es sich um nichts anderes als ein Pendant zur `malloc()` Funktion aus Standard-C. Der auf dem Host erzeugte Zeiger `dev_array` wird der Funktion übergeben und zeigt damit auf den Beginn des reservierten Speicherbereichs auf der GPU. Der zweite Parameter bestimmt wie in malloc die Größe des zu reservierenden Bereichs. Beim zweiten Aufruf der Funktion, wird ein weiterer Speicherbereich mit dem Zeiger `dev_array_new` auf der GPU erzeugt, in welchem später die berechnete neue Zellgeneration gespeichert wird.

```
1  cudaMalloc((void**)&dev_array, (DIM+2)*(DIM+2)*sizeof(int));
2  cudaMalloc((void**)&dev_array_new, (DIM+2)*(DIM+2)*sizeof(int));
```

Bei der Arbeit mit Devicezeigern ist folgendes zu beachten:

- Man kann sie an Host **und** Device(`__global__`) Funktionen übergeben
- Sie können **nur** zum Lesen und Schreiben von Device Speicher verwendet werden

[22] als Vorlage diente das Beispiel unter http://www.olcf.ornl.gov/training_articles/cuda-gol/

[23] Ein Framework zur Erstellung von GUIs, weitere Informationen unter http://www.gtk.org/

Der mit `cudaMalloc()` erzeugte Speicherbereich ist linear, eine Adressierung mittels eines zweidimensionalen Arrays ist nicht möglich. Daher kann im Beispiel nur ein einfacher Index verwendet werden. Die einzelnen Zellen werden nicht über x/y Werte adressiert, sondern über einen fortlaufenden Index. Der Wert in der linken oberen Ecke, wird über den Index 1 (0 ist die Temp Spalte), der rechte untere über die quadrierte Dimensionsgröße (im Beispiel 100) angesprochen.

Mithilfe der Funktion `cudaMemcpy()`, wird Speicher zwischen Host und Device oder Device intern kopiert. Es handelt sich praktisch um die ”GPU Version” von `memcpy()`. Im Beispiel wird ein auf dem Host mit zufälligen Werten angelegtes Array, in den gerade neu erzeugten Device-Speicherbereich kopiert. Die Funktion erwartet als Parameter in folgender Reihenfolge: Zeiger auf Ziel-Speicher, Zeiger auf Quell-Speicher, Größe des Speicherbereichs und Art der beiden Zeiger. Der letzte Parameter wird über eine von drei Konstanten aus der CUDA-API angegeben: `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToDevice` und `cudaMemcpyDeviceToDevice`. Die Namen der Konstanten sprechen für sich.

```
1  cudaMemcpy(dev_array, h_array, (DIM+2)*(DIM+2)*sizeof(int),
2                cudaMemcpyHostToDevice);
```

In den Zeilen 99/100 erfolgen die ersten Aufrufe von GPU Funktionen.

```
1  tempRows<<<DIM, 1>>>(dev_array);
2  tempCols<<<DIM, 1>>>(dev_array);
```

Wie man sehen kann, werden Parameter genau wie in Standard C übergeben. Allerdings befindet sich zusätzlich der Ausdruck `<<<DIM, 1>>>` im Funktionsaufruf. Dieser gibt an, wie die GPU die Berechnung ausführen soll und bringt die Parallelisierbarkeit ins Spiel. Der erste Parameter innerhalb des Ausdrucks, gibt die Anzahl der parallelen Blöcke auf der GPU an. In diesem Fall werden also 10 Kopien der Funktion angelegt. Natürlich können weitaus mehr angelegt werden, die Höchstzahl ist hardwareseitig auf 65535 limitiert [SK10]. Hier sieht man, wie einfach sich die parallele Programmierung mittels CUDA gestaltet. Doch damit muss auch in der Funktion selbst ein Zugriff auf die einzelnen Blöcke gewährleistet sein. Dies geschieht mittels der CUDA-Variablen `blockIdx.x`. Diese enthält automatisch immer die Nummer des aktuellen Blocks. Zur Verdeutlichung zeigen die Listings 4.1 bis 4.4 die ersten 4 Blöcke bei Aufruf der Funktion `tempCols`. Die Variable id entspricht dabei jeweils der blockId.x CUDA Variable, siehe Anhang C.1.

Listing 4.1: Funktion tempCols Block1

```
1  __global__ void
2  tempCols(int *array)
3  {
4    //erster Block blockIdx.x = 0
5    int id = 0;
6
7    if (id <= DIM+1)
8    {
9      array[id*(DIM+2)+DIM+1] = array[
           id*(DIM+2)+1];
10     array[id*(DIM+2)] = array[id*(
           DIM+2) + DIM];
11   }
12 }
```

Listing 4.2: Funktion tempCols Block2

```
1  __global__ void
2  tempCols(int *array)
3  {
4    //zweiter Block blockIdx.x = 1
5    int id = 1;
6
7    if (id <= DIM+1)
8    {
9      array[id*(DIM+2)+DIM+1] = array[
           id*(DIM+2)+1];
10     array[id*(DIM+2)] = array[id*(
           DIM+2) + DIM];
11   }
12 }
```

Listing 4.3: Funktion tempCols Block3

```
1  __global__ void
2  tempCols(int *array)
3  {
4    //dritter Block blockIdx.x = 2
5    int id = 2;
6
7    if (id <= DIM+1)
8    {
9      array[id*(DIM+2)+DIM+1] = array[
           id*(DIM+2)+1];
10     array[id*(DIM+2)] = array[id*(
           DIM+2) + DIM];
11   }
12 }
```

Listing 4.4: Funktion tempCols Block4

```
1  __global__ void
2  tempCols(int *array)
3  {
4    //vierter Block blockIdx.x = 3
5    int id = 3;
6
7    if (id <= DIM+1)
8    {
9      array[id*(DIM+2)+DIM+1] = array[
           id*(DIM+2)+1];
10     array[id*(DIM+2)] = array[id*(
           DIM+2) + DIM];
11   }
12 }
```

Die Funktion `next_generation<<<blocks, 1>>>(dev_array, dev_array_new)` durchläuft das Array und prüft für jeden Wert die Anzahl seiner Nachbarn. Dabei ist eine zweite ”Dimension” hilfreich. CUDA liefert die Möglichkeit Blockdimensionen zu generieren. Man spricht dabei auch von einem ”grid” [SK10]. In Zeile 90 wird mittels des Variablentyps `dim3` ein zweidimensionales[24] grid deklariert und anschließend als ”Blockparameter” übergeben.

```
1  //Groesse der Bloecke festlegen
2    dim3 blocks(DIM,DIM);
```

Die Funktion `next_generation` ist folgendermaßen aufgebaut:

```
1  __global__ void next_generation(int *array, int *array_new) {
2    int neighbors;
```

[24] Der Typ ”dim3” deutet auf drei Dimensionen hin. Allerdings ist die Verwendung von drei Dimensionen erst bei GPUs, welche ”Compute capability” Version 2.x besitzen anwendbar. Alle älteren Versionen können maximal zwei Dimensionen verwenden.

```
3   int iy = blockIdx.y + 1;
4   int ix = blockIdx.x + 1;
5   int id = iy *(DIM + 2) + ix;
6
7   if (iy <= DIM && ix <= DIM) {
8     neighbors = getNeighbors(array, id);
9
10    if (neighbors == 3 || (neighbors == 2 && array[x][y])) {
11      array_new[id] = 1;
12    } else {
13      array_new[id] = 0;
14    }
15  }
16 }
```

Über `blockIdx.y` wird auf die zweite Dimension zugegriffen und die nächste Generation für jede einzelne Zelle berechnet. Die Dimensionen stellt man sich am Besten als eine Matrix vor. In Abbildung 4.2 wird eine zweifache Verwendung der Dimensionsgröße 5 dargestellt.

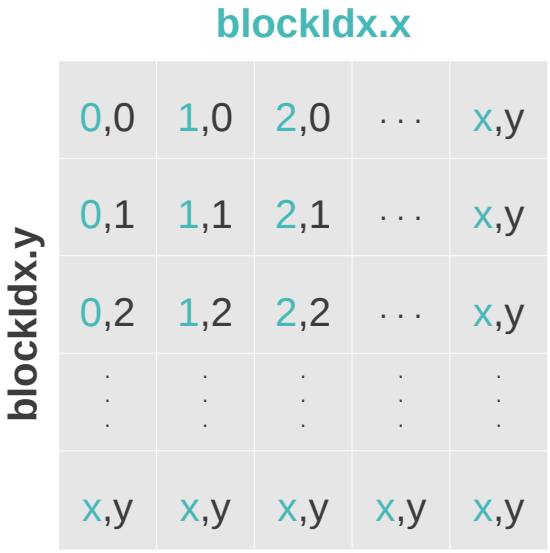

Abbildung 4.2: Darstellung Blöcke mit 2 Dimensionen

Zur besseren Übersicht ist das Zählen der Nachbarn nochmals in eine eigene Funktion `getNeighbors(int *array, int id)` ausgliedert, die den Bezeichner `__device__` besitzt. Dieser legt fest, dass die Funktion nur von Funktionen, welche ebenfalls auf der Device laufen, aufgerufen werden kann.

In Zeile 116/117 befindet sich mit `cudaFree` das Pendant zur C Funktion `free`. Der auf dem Device reservierte Speicher wird damit wieder freigegeben.

erweiterte GPU-Implementierung

Der im letzten Teil beschriebene Algorithmus ist nicht besonders performant. Denn für jede Zelle, deren nächste Generation berechnet wird, muss zur Bestimmung der Nachbaranzahl 8x auf den Grafikspeicher[25] zugegriffen werden. Dieser ist recht langsam und deshalb oft der ”Flaschenhals” eines Programms. Allerdings besitzen die GPUs eine Art schnellen Registerspeicher[26], der als ”Shared Memory” bezeichnet wird [SK10]. Der Name resultiert aus einem gemeinsamen Speicher, der von mehreren Threads gemeinsam genutzt wird. Threads werden in CUDA mittels des zweiten Parameters beim Aufruf einer `__global__` Funktion definiert. Zum Beispiel würde die Funktion `next_generation<<<16,32>>>(dev_array, dev_array_new)` mit 16 Blöcken und jeweils 32 Threads ausgeführt werden. Der Vorteil an der Verwendung von Threads ist einerseits deren Kommunikationsfähigkeit über einen gemeinsamen Speicherbereich, sowie andererseits die Umgehung der festgelegten Höchstanzahl der Blöcke. Die Anzahl der maximalen Threads pro Block, liegt je nach Grafikkartengeneration bei 512 oder 1024, daraus resultiert eine maximale mögliche Anzahl von 512/1024 x 65535 Blöcken. Listing C.2 zeigt den abgeänderten Algorithmus mit der Verwendung von Threads und Shared Memory.

Über das Makro THREAD_NUMBER wird die Anzahl der zu verwendenden Threads angegeben, die Anzahl der Blöcke wird dann anhand der Dimensionsgröße automatisch berechnet. Zusätzlich wird später geprüft, ob die eingegebene Threadzahl nicht die maximale, durch die Hardware beschränkte Threadzahl pro Block übersteigt.

```
1  #define THREAD_NUMBER 5
2      .
3      .
4      .
5  int main(void) {
6      .
7      .
8      .
9    //Pruefen ob gewaehlte Threadzahl zulaessig ist
10   cudaDeviceProp deviceProp;
11   cudaGetDeviceProperties(&deviceProp, 0);
```

[25] auch Global Memory
[26] siehe Kapitel 2.2.2

```
12   int maxThreads = deviceProp.maxThreadsPerBlock;
13   int threadNumberMax = (int)sqrt((double)maxThreads);
14   if (THREAD_NUMBER > threadNumberMax) {
15   printf("Die gewaehlte Threadzahl ueberschreitet die maximale Threadzahl
         der
16   verfuegbaren CUDA Device!\n");
17   printf("Maximal %d Threads sind moeglich.", threadNumberMax);
18   return -1;
19   }
20     .
21     .
22     .
23 }
```

Genauso wie `blockIdx` die Nummer des aktuellen Blocks angibt, gibt `threadIdx` die Nummer des aktuellen Threads an. Auch Threads können über mehrere Dimensionen verfügen, welche genau wie die der Blöcke angesprochen werden.

```
1   int iy = (blockDim.y-2) * blockIdx.y + threadIdx.y;
2   int ix = (blockDim.x-2) * blockIdx.x + threadIdx.x;
3   int id = iy * (DIM+2) + ix;
4
5   int i = threadIdx.y;
6   int j = threadIdx.x;
```

Die Variable `blockDim` gibt die Anzahl der Threads in den Blöcken der Dimension an, besitzt im Beispiel daher immer den Wert 5. Zur Verdeutlichung der Block<->Thread Beziehung stellt Abbildung 4.3 den Zusammenhang als Matrix dar. Mittels des Bezeichners `__shared__` wird ein zweidimensionales Array im Shared Memory, auf das alle Threads eines Blocks Zugriff haben deklariert. In dieses werden die Werte des Arrays aus dem Global Memory kopiert. Um sicherzustellen, dass der ”Kopiervorgang” vor der Weiterführung des Programms beendet ist, wird die Funktion `__syncthreads()` aufgerufen. Erst wenn alle Threads innerhalb des Blocks diese Funktion aufgerufen haben, erfolgt die Fortführung des Programms.

```
1  // shared array
2   __shared__ int shared_array[THREAD_NUMBER][THREAD_NUMBER];
3
4   // Kopieren der Zellen ins shared array
5   if (ix <= DIM+1 && iy <= DIM+1)
6     shared_array[i][j] = array[id];
7
8   // threads synchronisieren
9     __syncthreads();
```

In Zeile 122 und 124 werden die zweidimensionalen Blöcke/Threads deklariert und in Zeile 137 beim Aufruf der Funktion `next_generation` übergeben. Hinweis: Trotz der Verwendung von Shared Memory, lief diese Version auf eine Nvidia Quadro 140M Grafikkarte, langsamer als die erste Version mit Global Memory. Das Kopieren des Arrays in

den Shared Memory ist ein zusätzlicher Zeitaufwand, der durch Verringerung der Global Memory Zugriffe in diesem Fall nicht ausgeglichen wird. Die Berechnung einer Generation benötigte bei Verwendung von Global Memory ca. 50ms , bei Shared Memory ca. 65ms.

Abbildung 4.3: Darstellung Blöcke und Threads mit 2 Dimensionen

4.3 OpenCL

OpenCL steht für ”Open Computing Language” und ist ein offener Standard, der vom Industriekonsortium Khronos Group entwickelt wurde. Ebenso wie CUDA besitzt es eine eigene Sprache ”OpenCL C”, sowie eine API, die aus einer Plattformschicht-API und einer Laufzeit-API besteht. Die Erstgenannte bildet eine Hardwareabstraktionsschicht und ermöglicht den Zugriff auf verschiedenste Geräte. Die Zweite verwaltet die Rechen- und Speicherressourcen und bietet dem Entwickler die Möglichkeit sogenannte ”Rechenkernel” in Form einer Warteschlange zu verwalten. Das Besondere ist jedoch die Tatsache, dass OpenCL als offener Standard entwickelt wurde. Dadurch ist das Framework auf keinen bestimmten Hersteller zugeschnitten und ermöglicht eine Implementierung verschiedenster Hardware oder Betriebssysteme. Der Standard ist also nicht nur an die Besonderheiten der GPU-Programmierung angepasst, sondern umfasst auch CPUs, DSPs oder spezielle Prozessoren wie den ”Cell”[27] Prozessor. Dadurch kann OpenCL auch für die Entwicklung von Smartphone Anwendungen genutzt werden. Es lassen sich datenparallele (SIMD), als auch aufgabenparallele (Threads) Probleme beschreiben. Durch eine vorgegebene Spezifikation, werden einheitliche Implementierungen für den Nutzer sichergestellt. [opea] [opeb] [opec]

4.3.1 Einrichtung

Für die Verwendung von OpenCL benötigt man zwei Dinge:

- Eine OpenCL SDK
- Einen C Compiler

OpenCL SDK: OpenCL SDKs werden von AMD, Intel, Nvidia, sowie IBM bereitgestellt. Damit wird eine große Bandbreite an Hardwarekonstillationen abgedeckt. Sie enthalten alle für die Verwendung von OpenCL benötigten Funktionen.

C Compiler: Ein C Compiler ist zusätzlich zur Ausführung nötig. Weitere Informationen dazu siehe Abschnitt 4.2.1.

Bei Nutzung der GPU muss diese OpenCL unterstützen. Außerdem muss ein aktueller, unterstützender Treiber installiert sein.

[27] eine Prozessorreihe von IBM, siehe http://de.wikipedia.org/wiki/Cell_(Prozessor)

4.3.2 Das erste OpenCL Programm

Zur Erläuterung von OpenCL, soll auch hier wieder auf den Game of Life - Algorithmus zurückgegriffen werden. Das komplette Beispiel[28] befindet sich in Anhang D.1. Mehr Informationen zum Game of Life im Allgemeinen, siehe Kapitel 4.2.2. Bedingt durch die Plattformunabhängigkeit, ist mehr Code vonnöten, als bei der CUDA Version. Allerdings sind die daraus resultierenden Vorteile größer. Die OpenCL Laufzeitumgebung ermittelt auf dem Zielrechner die Konfiguration und führt die Berechnungen auf den gewünschten "Devices" aus. Ist keine GPU vorhanden, wird die CPU zur Berechnung verwendet. Die Ermittlung der Konfiguration, sowie Steuerung des Programms erfolgt in einer Standard C-Datei, die als Kernel bezeichneten, zu parallelisierenden Berechnungen in einer Datei mit ".cl" Endung. Die Datei wird im Hauptprogramm eingelesen und erst zur Laufzeit kompiliert. Somit ist eine Verwendung auf vielen verschiedenen Zielsystemen gewährleistet. Im Folgenden werden nun alle OpenCL Bestandteile des Beispiels erläutert.

Speicherplatz auf Devices wird mittels des OpenCL Typs `cl_mem` definiert. Zur Ermittlung der OpenCL Konfiguration und Erzeugung der Kernels stellt OpenCL weitere Typen bereit.

```
1  // Speicher auf Device
2  cl_mem dev_array;
3  cl_mem dev_array_new;
4
5  // OpenCL platform
6  cl_platform_id cpPlatform;
7  // device ID
8  cl_device_id device_id;
9  // context
10 cl_context context;
11 // command queue
12 cl_command_queue queue;
13 // program
14 cl_program program;
15
16 //OpenCL kernels
17 cl_kernel k_gol, k_tempRows, k_tempCols;
```

Allen erzeugten Variablen, werden im weiteren Quelltext Werte zugewiesen. Als erstes wird mittels der Funktion `clGetPlatformIDs` die OpenCL Plattform des Rechners ermittelt und in der Variablen cpPlatform gespeichert. Als Plattform werden die verfügbaren OpenCL Implementationen, z.B. von Nvidia, AMD, IBM und Intel, bezeichnet. Sollten mehrere auf einem Rechner zur Verfügung stehen, wird im Beispiel die erstgenannte gewählt.

```
1  // Ermitteln der (erstmoeglichen) OpenCL platform
```

[28] als Vorlage diente das Beispiel unter http://www.olcf.ornl.gov/training_articles/opencl-gol/

```
2   err = clGetPlatformIDs(1, &cpPlatform, NULL);
3   if (err != CL_SUCCESS) {
4     printf( "Error: Es konnte keine OpenCL platform gefunden werden!\n");
5     return EXIT_FAILURE;
6   }
```

Die Funktion `clGetDeviceIDs` sucht mittels der Variablen `CL_DEVICE_TYPE_GPU` nach einer vorhandenen GPU, um sie als Device zu verwenden. Sollte keine gefunden werden, wird stattdessen die CPU verwendet. Neben diesen beiden Typen, existieren allerdings noch weitaus mehr in der OpenCL Spezifikation[29].

```
1   // Ermitteln der (erstmoeglichen) gpu device
2   err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL)
        ;
3   if (err != CL_SUCCESS) {
4     printf("Warnung: Keine GPU konnte gefunden werden! Zur Berechnung wird
          die CPU verwendet. \n");
5     err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_CPU, 1, &device_id,
          NULL);
6     if (err != CL_SUCCESS) {
7       return EXIT_FAILURE;
8     }
9   }
```

Des weiteren verwaltet OpenCL die Systemumgebung (Anzahl der Devices und deren verfügbarer Speicher) in einem ”context”. Der context ist notwendig, um gemeinsame Speicherobjekte zwischen den Geräten zu teilen. [opea] Über die Funktion `clCreateContext` wird dieser erstellt.

```
1   // OpenCL context erstellen
2   context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
3   if (!context) {
4     printf("Error: Es konnte kein context erstellt werden!\n");
5     return EXIT_FAILURE;
6   }
```

Um Befehle an die Devices weiterzuleiten, benutzt OpenCL Command Queues (dt. Befehlswarteschlangen). Erstellt wird die Warteschlange mittels der Funktion `clCreateCommandQueue`.

```
1   // command queue fuer das gewaehlte device erstellen
2   queue = clCreateCommandQueue(context, device_id, 0, &err);
3   if (!queue) {
4     printf("Error: Es konnte keine command queue erstellt werden!\n");
5     return EXIT_FAILURE;
6   }
```

[29] siehe http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/

Wie bereits erwähnt, werden Kernels in einer .cl Datei ausgelagert. Die Datei wird eingelesen und in einem String gespeichert. Mit `clCreateProgramWithSource` wird daraus ein OpenCL program (Menge von OpenCL Kerneln) erstellt.

```
1  // externe OpenCL kernel quellcode datei oeffnen und daraus ein OpenCL
       Programm erstellen.
2  char *fileName = "kernels.cl";
3  FILE *fh = fopen(fileName, "r");
4  if(!fh) {
5    printf("Error: Datei \"kernels.cl\" konnte nicht geoeffnet werden!\n");
6    return 0;
7  }
8  struct stat statbuf;
9  stat(fileName, &statbuf);
10 char *kernelSource = (char *) malloc(statbuf.st_size + 1);
11 fread(kernelSource, statbuf.st_size, 1, fh);
12 kernelSource[statbuf.st_size] = '\0';
13 program = clCreateProgramWithSource(context, 1, (const char **) &
       kernelSource, NULL, &err);
14 if (!program) {
15   printf("Error: Es konnte kein OpenCL program aus dem Kernel erstellt
         werden \n");
16   return EXIT_FAILURE;
17 }
```

Das erstellte Programm muss anschließend kompiliert werden, dies geschieht mit der Funktion `clBuildProgram`.

```
1  // das erstellte Programm kompilieren
2  err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
3  if (err != CL_SUCCESS) {
4    printf("Error: OpenCL program konnte nicht kompiliert werden!
         Fehlercode: %d\n", err);
5    return EXIT_FAILURE;
6  }
```

Um die Funktionen im erstellten Programm einzeln zu verwenden, existiert die Funktion `clCreateKernel`. Sie erstellt einen Kernel und legt ihn in der am Anfang definierten k_gol Variablen ab.

```
1  // GOL kernel aus dem erstellten Programm bauen
2  k_gol = clCreateKernel(program, "GOL", &err);
3  if (!k_gol || err != CL_SUCCESS) {
4    printf("Error: GOL kernel konnte nicht angelegt werden! \n");
5    return EXIT_FAILURE;
6  }
```

Das Anlegen und der Zugriff auf Speicher auf dem Device, erfolgt nicht wie bei CUDA über Zeiger, sondern über sogenannte ”Buffer”. Die Funktion `clCreateBuffer` erzeugt die Buffer, `clEnqueueWriteBuffer` schreibt Werte in den erzeugten Bereich.

```
1  // Initialisieren der arrays auf dem device
2  dev_array = clCreateBuffer(context, CL_MEM_READ_WRITE,
       number_cells_in_bytes, NULL, NULL);
3  dev_array_new = clCreateBuffer(context, CL_MEM_READ_WRITE,
       number_cells_in_bytes, NULL, NULL);
4  if (!dev_array || !dev_array_new) {
5    printf("Error: Es konnte kein Speicher auf dem device angelegt werden!
       \n");
6    return EXIT_FAILURE;
7  }
8
9  // Schreiben der Werte auf dem host array ins device array
10 err = clEnqueueWriteBuffer(queue, dev_array, CL_TRUE, 0,
       number_cells_in_bytes, h_array, 0, NULL, NULL);
11 if (err != CL_SUCCESS) {
12   printf("Error: Uebertragung der Daten vom Host zur Device nicht
       erfolgreich! \n");
13   return EXIT_FAILURE;
14 }
```

`clSetKernelArg` übergibt schlussendlich Parameter an die erstellten Kernel, bevor diese mit `clEnqueueNDRangeKernel` ausgeführt werden. Die Variable `GolGlobalSize` ist die Anzahl sogenannter Work Groups und entspricht den Blöcken von CUDA. `GolLocalSize` beinhaltet Work Items und entspricht CUDA Threads. Deklariert werden sie in einem ”Array” vom Typ `size_t`. Die Größe des Arrays gibt dabei die Anzahl der Dimensionen an. Abbildung 4.4 gibt eine Übersicht über das OpenCL Speichermodell.

```
1  // Uebergeben der Parameter an GOL kernel
2  err  = clSetKernelArg(k_gol, 0, sizeof(int), &dim);
3
4  size_t GolLocalSize[2] = {localSize, localSize};
5  size_t roundedGlobal = (size_t)ceil(dim/(float)localSize)*localSize;
6  size_t GolGlobalSize[2] = {roundedGlobal, roundedGlobal};
7
8  err = clEnqueueNDRangeKernel(queue, k_gol, 2, NULL, GolGlobalSize,
       GolLocalSize, 0, NULL, NULL);
```

Die Funktion next_generation in der kernels.cl Datei verwendet die Funktion `get_global_id()` zur Bestimmung der Datenelemente. Die Funktion berechnet einen globalen Index aus allen Workgroups. Zum Vergleich, bei CUDA erfolgt die Indexermittlung mittels den Strukturen `threadId`, `blockId` und `blockDim`. Der Bezeichner `__kernel` entspricht dem `__global__` Bezeichner von CUDA zur Kennzeichnung von Device Funktionen. Zeiger auf Device Speicher tragen in OpenCL den Bezeichner `__global`.

```
1  //Berechnung der neuen Generation und Speicherung in neuem Array
2  __kernel void next_generation(const int dim, __global int *array,
       __global int  *array_new)
3  {
4    int ix = get_global_id(0) + 1;
5    int iy = get_global_id(1) + 1;
```

```
6     int id = iy * (dim+2) + ix;
7
8     int neighbors;
9
10    if (iy <= dim && ix <= dim) {
11
12      neighbors = array[id+(dim+2)] + array[id-(dim+2)]
13                + array[id+1] + array[id-1]
14                + array[id+(dim+3)] + array[id-(dim+3)]
15                + array[id-(dim+1)] + array[id+(dim+1)];
16
17      if (neighbors == 3 || (neighbors == 2 && array[id])) {
18        array_new[id] = 1;
19      } else {
20        array_new[id] = 0;
21      }
22  }
```

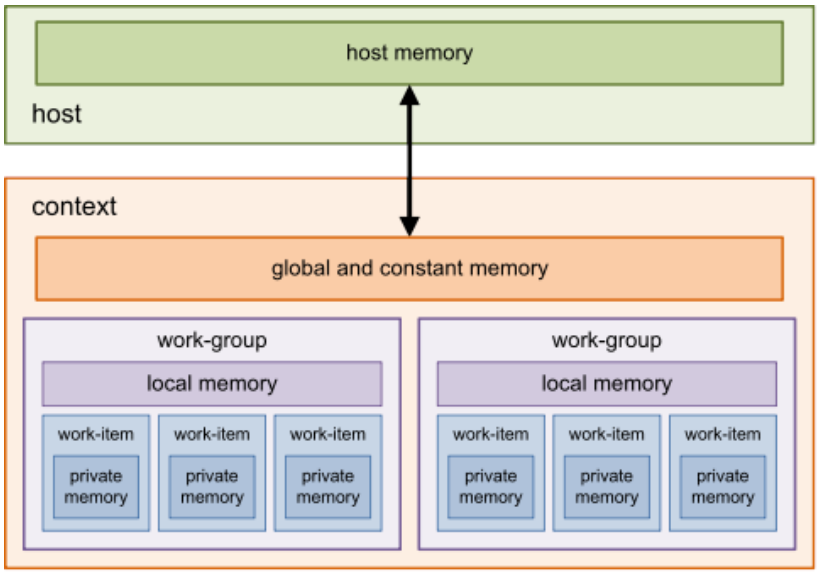

Abbildung 4.4: OpenCL Modell [opea]

4.4 Vergleich

Beide Frameworks ermöglichen durch Hardwareabstraktion einen einfachen Zugriff auf die GPU und deren Speicher. Der größte Unterschied zeigt sich in der Plattformunabhängigkeit. OpenCL ermöglicht eine einfache Implementierung für viele Zielsysteme und ist nicht nur für den Einsatz auf GPUs beschränkt, es ist z.B. auch eine Optimierung für Multicore CPUs möglich. Ein weiterer Vorteil ist die Möglichkeit, ein bereits vorhandenes C-Programm, bzw. rechenintensive Teile dessen, relativ einfach nach OpenCL zu portieren. Zusätzlich kann eine Berechnung einfach auf die CPU ”umgeleitet” werden,

falls im Zielsystem keine GPU vorhanden ist. Im Vergleich dazu, kann CUDA derzeit[30] nur auf Nvidia GPUs eingesetzt werden und bietet auch kein ”Fallback”, wenn im Zielsystem keine entsprechende Hardware vorhanden ist. Für CUDA spricht allerdings die etwas einfachere und näher an C orientierte Syntax, z.B. die Speicherallokation des GPU Speicher mittels Zeigern. Zudem zeigt sich bei Verwendung des CUDA Compilers auf Nvidia GPUs eine bessere Performance, als mit dem OpenCL Compiler[31]. CUDA befindet sich schon länger auf dem Markt als OpenCL, daher fallen die Dokumentation, Beispiele und Tutorials etwas ausführlicher aus. Da zur Kompilierung einer .cu Datei der nvcc Compiler von CUDA verwendet wird, ist die Einrichtung eine Entwicklungsumgebung jedoch komplizierter als bei OpenCL, dass beim Kompilierungsvorgang lediglich die Angabe einiger Parameter mit Pfaden zum installierten SDK benötigt. In Tabelle 4.1 erfolgt eine Gegenüberstellung anhand verschiedener Faktoren.

[30] Stand Mai 2012, Der Quellcode des CUDA Compilers wurde im Dezember 2011 für Entwickler freigegeben, sodass Portierungen auf andere Zielsysteme in naher Zukunft folgen könnten, siehe http://www.golem.de/1112/88414.html

[31] Die Performancevorteile liegen zwischen 5 und 50%, siehe http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf und http://www.sisoftware.net/?d=qa&f=gpgpu_gpu_perf&l=de&a=

	CUDA	**OpenCL**
einfach zu erlernende Syntax	++	+
Plattformunabhängigkeit	-	++
Dokumentation, Tutorials	++	+
Debugging[a]	++	o
universelle Verwendung, unabhängig ob eine GPU im Zielsystem vorhanden ist	- -	++
einfache Einrichtung auf Entwicklungsrechner	+	++

Legende

++	+	o	-	- -
erfüllt die Anforderung vollständig	erfüllt die Anforderung teilweise	Anforderung wird erfüllt, wurde aber nicht getestet	Anforderung unzureichend erfüllt	Anforderung nicht erfüllt

[a] siehe Kapitel 5.2

Tabelle 4.1: Vergleich CUDA und OpenCL

5 Praxistauglichkeit von GPU Computing

Für die Verwendung von CUDA und OpenCL existieren bereits verschiedenste Wrapper und Tools um die GPU Entwicklung zu erleichtern bzw. zu erweitern. In diesem Kapitel wird ein Überblick gegeben. Außerdem wird untersucht, wann der Einsatz der GPU lohnenswert ist und welche Optimierungsmöglichkeiten vorhanden sind.

5.1 Wrapper/APIs

5.1.1 MATLAB

Die Software MATLAB[32] ist besonders auf die Berechnung von Matrizen ausgelegt. Dabei wird oft ein Rechenbefehl auf alle Elemente einer Matrix angewendet. Gerade hierfür bietet sich der Einsatz der GPU an. Derzeit existieren zwei kostenpflichtige Plugins, welche einen GPU-Zugriff in Matlab integrieren und eine einfache Nutzung von CUDA und OpenCL aus Matlab heraus bieten: die Parallel Computing Toolbox, direkt entwickelt von MathWorks, sowie das Jacket Plugin der Firma Accelereyes.

Parallel Computing Toolbox

Die Parallel Computing Toolbox integriert Funktionen in Matlab, die einen einfachen Zugriff auf die GPU bieten. Beispielsweise kopiert die Funktion `gpuArray()`, eine Matrix in den Speicher der GPU. Alle auf diese Matrix angewandten Operationen, werden nun auf der GPU ausgeführt. Desweiteren besteht die Möglichkeit, bereits vorhandene CUDA kernel, aus einer ".cu" Datei, mit der Funktion `parallel.gpu.CUDAKernel()`, einzulesen. Das Plugin unterstützt allerdings kein OpenCL und funktioniert erst mit CUDA GPUs, welche mindestens "Compute Capability Version 1.3"[33] besitzen. Aufgrund fehlender Verfügbarkeit der entsprechenden Hardware, konnte im Rahmen dieser Arbeit kein weiteres Beispiel mittels des Plugins getestet werden. Ausführliche Beispiele sind unter http://www.mathworks.de/products/parallel-computing/demos.html zu finden.

Jacket

Das Jacket Plugin bietet ähnliche GPU Nutzung wie die Parallel Computing Toolbox, beinhaltet aber mehr Funktionen und ist laut eigener Aussage[34] um bis zu 50% schnel-

[32] kommerzielle Software zur Lösung mathematischer Probleme, siehe http://www.mathworks.de/
[33] gibt Auskunft über die Funktionsmenge der GPU
[34] siehe http://www.accelereyes.com/products/compare, aufgerufen am 27.04.2012

ler. Weiterhin existiert bereits eine BETA Version, welche OpenCL unterstützt. Das folgende Code Beispiel[35] zeigt eine einfache Game of Life CPU Matlabversion.

```
len=50; GRID=int8(rand(len,len));
up=[2:len 1]; down=[len 1:len-1];
colormap(gray(2));
for i=1:100
    neighbours=GRID(up,:)+GRID(down,:)+GRID(:,up)+GRID(:,down)+...
        GRID(up,up)+GRID(up,down)+GRID(down,up)+GRID(down,down);
    GRID = neighbours==3 | GRID & neighbours==2;
    image(GRID*2); pause(0.02);
end
```

Um nun das Beispiel mittels Jacket auf der GPU auszuführen, muss lediglich die Deklaration der GRID Variable geändert werden.

```
GRID = gint8(rand(len,len));
```

Die Funktion gint8() erzeugt eine Matrix im GPU-Speicher. Die darauf folgenden Operationen, werden automatisch auf der GPU ausgeführt. Das Plugin kann nach Anmeldung 15 Tage kostenlos getestet werden. Für akademische Einrichtungen kostet eine Lizenz aktuell[36] 350$ und standardmäßig 999$ je verwendeter GPU.

5.1.2 ArrayFire

ArrayFire ist eine Bibliothek der Firma AccelerEyes, welche einen einfachen Zugriff auf CUDA und OpenCL ermöglicht. Unterstützt werden die Programmiersprachen C, C++, Fortran und Python. Es wird eine wesentlich vereinfachte Syntax verwendet. Code welcher in den beiden Original APIs viele Zeilen umfasst, kann auf wenige Zeilen reduziert werden. Siehe dazu folgendes Beispiel:

```
//erzeugt eine 4x5 Matrix mit zufaelligen float Werten auf der GPU
array A = randu(4,5);
//erzeugt eine neue Matrix gefuellt mit Werten aus Fourier Transformation
    der ersten Matrix
array B = fft(A);
//erzeugt neue Matrix aus der letzten Zeile von Matrix B
array C = B.row(end);

print(A);
print(B);
print(C);
```

Die Bibliothek ist in einer nur gering abgespeckten Version (u.A. keine Multi-GPU Unterstützung) kostenlos verfügbar.

[35] siehe http://www.exolete.com/code/life
[36] Stand Mai 2012, siehe http://www.accelereyes.com/products/jacket_licensing

5.1.3 Java

Die Programmierung der GPU ist nicht nur auf die von C abgeleiteten Sprachen CUDA und OpenCL beschränkt. So existieren für die Programmiersprache Java verschiedene Wrapper und seit September 2011 das Opensourceprojekt ”Aparapi” von AMD.

jcuda/jocl

Unter www.jcuda.org und www.jocl.org werden Schnittstellen angeboten, welche die Ausführung von CUDA und OpenCL Funktionen aus Java heraus ermöglichen. Die Steuerung und Konfiguration erfolgt dabei komplett in Java mittels eigens dafür vorgesehenen Klassen. Die GPU Funktionen (siehe __global__ bzw. __kernel), müssen weiterhin in .cu bzw. .cl Dateien geschrieben und separat kompiliert werden. Die kompilierten Dateien können dann in Java aufgerufen und verarbeitet werden. Da die dafür nötigen Cuda/OpenCL Systemdateien mittels *JNI* aufgerufen werden, ist allerdings keine Plattformunabhängigkeit vorhanden.

Aparapi

Eine im Vergleich zu jocl vollständige Javaimplementierung von OpenCL, bietet Aparapi (”A parallel API”) von AMD, welche auch die für Java stehende Plattformunabhängigkeit gewährleistet. Dazu wird der Java Bytecode zur Laufzeit in OpenCL umgewandelt und ausgeführt. Wenn keine unterstützende GPU gefunden wird, erfolgt die Ausführung über den Java thread pool. Die Realisierung erfolgt über eine Klasse ”Kernel”. Diese enthält eine zu überschreibende run() Methode, in der die benötigte Berechnung formuliert wird. Indem man von Kernel erbende Klassen implementiert, kann man so mehrere OpenCL Kernel erstellen. Ausgeführt werden die Kernel dann mittels der execute Methode, die als Parameter die Anzahl der auszuführenden Workgroups erwartet. Der nun folgenden Codeabschnitt stellt das beschriebene dar:

```
1  int A[] = new int[1024];
2  int B[] = new int[1024];
3  int C[] = new int[A.length];
4
5  //Berechnung wie gewoehnlich auf der CPU
6  for (int i=0; i<C.length; i++){
7      C[i]=A[i]+B[i];
8  }
9
10 //Aparapi Version
11 Kernel kernel = new Kernel(){
12     @Override
13     public void run(){
14         int i= getGlobalId();
15         C[i]=A[i]+B[i];
```

```
16          }
17      };
18      kernel.execute(C.length());
```

5.1.4 andere Wrapper

Auch für andere Sprachen existieren Wrapper, so gibt es mehrere Implementierungen für Microsofts .NET Runtime:

- CUDAfy .NET - siehe http://www.hybriddsp.com/Products/CUDAfyNET.aspx
- GPU.NET - siehe http://www.tidepowerd.com/gpu-net
- CUDA.NET - siehe http://www.hoopoe-cloud.com/Solutions/cuda.net/

Unter dem Namen WebCL wird ein Javascript Binding angeboten, welches die Nutzung von OpenCL in Webapplikationen möglich macht. Derzeit existieren 2 Prototypimplementierungen: für Firefox von Nokia[37], sowie für die WebKit Engine (Safari, Chrome) von Samsung[38].

Weitere Sprachen zu denen Wrapper existieren sind: Python, Ruby und Pearl.

5.2 Software-Entwicklungstools

5.2.1 NVIDIA Visual Profiler

Der NVIDIA Visual Profiler ist im CUDA Toolkit enthalten und stellt Statistiken der ausgeführten CUDA oder OpenCL Berechnungen in Diagrammform dar. Zudem erfolgt eine Analyse nach eventuellen ”Geschwindigkeitsbremsen”, mit anschließenden Tipps diese zu beheben. Eine vollständige Unterstützung ist allerdings erst ab GPUs mit Compute Capability 2.x gegeben. [opt] Abbildung 5.2.1 zeigt eine Auswertung des Game of Life Beispiels unter der Verwendung von shared memory und einer Arraydimension von 1024x1024. Unter (1) sieht man alle auf der GPU durchgeführten Berechnungen und deren prozentualer Anteil an der Gesamtberechnungsdauer. Diese Verteilung und der zeitliche Ablauf werden im Balkendiagramm rechts daneben optisch dargestellt. In der untenstehenden Tabelle erfolgt die Auflistung der Funktionen (2), mit vielen weiteren Informationen in Tabellenform. Das Kopieren der Daten vom Host zum Device Speicher benötigte 2,634 ms(3) und belegte einen Speicherplatz von 4,016 MB (4).

[37] http://webcl.nokiaresearch.com/
[38] http://code.google.com/p/webcl/

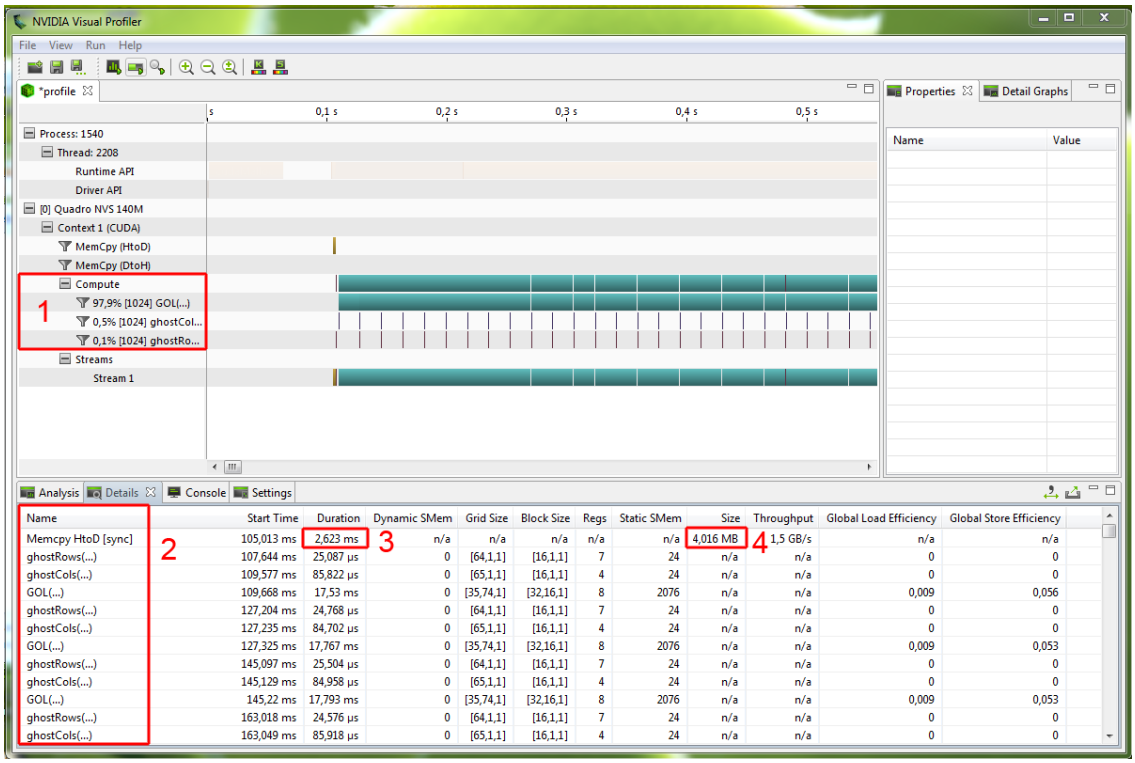

Abbildung 5.1: Screenshot NVIDIA Visual Profile - Game of Life Beispiel

5.2.2 Nsight

Für Windows Betriebssysteme existiert mit dem Visual Studio Plugin Nsight von Nvidia, ein bereits sehr ausgereiftes Entwicklungstool für CUDA Anwendungen. Es ist nach einer kostenlosen Registrierung in der Nvidia Developer Zone unter `http://developer.nvidia.com/nvidia-nsight-visual-studio-edition` erhältlich. Bei der Installation wird es automatisch in die auf dem System installierte Visual Studio Version[39] integriert. Das Tool beinhaltet Debugging und Profiling Funktionen. Das Profiling ähnelt dem des CUDA Visual Profilers und bietet ähnliche Funktionalitäten. Das Debuggen erfolgt über einen Monitor, der auf dem Rechner des zu untersuchenden Programms gestartet sein muss. Damit ist auch das Debuggen von Programmen auf anderen Rechnern, vom Entwicklungsrechner aus möglich. Die Vorgehensweise ist dabei die Gleiche wie beim regulären Debuggen mit Visual Studio. Haltepunkte können gesetzt werden, das Programm wird dementsprechend angehalten und kann auf Fehler untersucht werden.

5.2.3 Weitere Tools

AMD bietet mit dem AMD APP Profiler ein Pendant zum Visual Profiler, für die Verwendung von OpenCL auf AMD Hardware an. Auch hier ist eine Visualisierung der Be-

[39] unterstützt werden Version 2008 und 2010

rechnung möglich. Das Programm kann ebenfalls in Visual Studio integriert werden.[40] Weitere Profiler sind z.B. CAPS HMPP Wizard[41], Vampir[42] und TAU[43]. Auch weitere Debugger stehen zur Verfügung. So bietet AMD den gDEBugger[44] an. Dieser ist ebenfalls als Visual Studio Plugin oder als Standalone Version für Linux Systeme verfügbar. Auch Intel bietet einen Debugger und Profiler in der SDK for OpenCL Applications[45] an.

5.2.4 Sinnvoller Einsatz der GPU - Optimierungsmöglichkeiten

Allgemein

Zuallererst sollte immer versucht werden, den vorhandenen Quellcode soweit wie möglich zu parallelisieren. Hierbei findet das Amdahlsche Gesetz Anwendung:

$$Zeitgewinn = \frac{1}{(1-P)+\frac{P}{N}}$$

P ist dabei der parallele Anteil am Gesamtprogramm, während N die Anzahl der Prozessoren darstellt. Bei einer hohen Zahl an Recheneinheiten wie auf der GPU, tendiert der $\frac{P}{N}$ Anteil gegen null. Wenn ca. $\frac{3}{4}$ des Programms parallel auf der GPU ausgeführt werden, ergibt sich also ein Zeitgewinn von Faktor 4, während bei $P = \frac{1}{4}$ lediglich ein Zeitgewinn von $1,\overline{3}$ erreicht wird. Zusätzlich sollte die Menge der zu parallel verarbeiteten Daten geprüft werden. Da der Kopiervorgang vom Hauptspeicher in den Speicher der GPU sehr viel Zeit in Anspruch nimmt, ist bei geringer zu berechnender Datenmenge ggf. die CPU schneller. Sequentieller Code sollte generell immer auf der CPU berechnet werden, der Einsatz der GPU erfüllt hierfür keinen Nutzen. Das Kopieren zwischen den Speichern ist zu vermeiden, es stellt oft die größte Geschwindigkeitsbremse dar. Einmal mittels `cudaMalloc` erzeugter Speicher, sollte wenn möglich ”wiederverwendet” werden.

Speicherwahl auf der GPU

Wie bereits erwähnt, existieren auf der GPU mehrere Speicher. Neben Global und Shared Memory können bei CUDA[46] zusätzlich noch Constant und Texture Memory verwendet werden. Constant Memory ist ein 64 KB großer readonly Speicher direkt auf

[40] siehe http://developer.amd.com/tools/amdappprofiler/pages/default.aspx
[41] siehe http://www.caps-entreprise.com/fr/page/index.php?id=120&p_p=36
[42] siehe http://www.vampir.eu/
[43] siehe http://www.cs.uoregon.edu/Research/tau/home.php
[44] siehe http://developer.amd.com/tools/gDEBugger/Pages/default.aspx
[45] siehe http://software.intel.com/de/articles/programming-with-the-intel-sdk-for-opencl-applications-development-tools/
[46] auch bei OpenCL, jedoch unter anderer Bezeichnung

dem Grafikchip. Lesezugriffe werden stark gecached, sodass ein schneller Zugriff möglich ist. Zusätzlich wird der Zugriff noch durch die Hardware optimiert. Ein halber Warp, d.h. 16 Threads (siehe Kapitel 2.2.2), kann mit einem einzelnen Lesezugriff auf den Speicher zugreifen. Wenn alle Threads die selben Daten benötigen, bringt dies nochmals einen ordentlichen Geschwindigkeitszuwachs, bewirkt aber bei unterschiedlicher Datenanforderung je Thread das Gegenteil, da keine Threads parallel Daten anfordern können und deshalb nacheinander auf den Speicher zugreifen müssen. Texture Memory ist ebenfalls ein readonly Speicher auf dem Grafikchip, der bei Grafikberechnungen zum Speichern von Texturen dient. Dort werden die Pixel der gespeicherten Texturen einzeln angesteuert. Dies ermöglicht unter CUDA einen einfachen Zugriff von Threads die ”nahe” (im Adressbereich) beieinander liegen. Dem Speicher kann unter CUDA auch ein zweidimensionales Array übergeben werden, somit ist eine einfache Ansteuerung aller Elemente des Arrays möglich. Alle 4 verschiedenen Speicher, können je nach Anwendungsfall einen Performancegewinn bringen.

Blöcke und Threads

Entscheidend für die Performance, ist ebenfalls die Größe der Blöcke und Threads. Für Nvidia Karten gilt dabei folgendes[47]:

- Da ein Warp 32 Threads umfasst, sollte die Anzahl der Threads ein Vielfaches von 32 betragen, um eine optimale Auslastung zu ermöglichen.
- Es sollten stets mehrere Blöcke mit wenigen Threads, anstatt ein Block mit vielen Threads verwendet werden. So können andere Blocks ausgeführt werden, während einige gerade aufgrund von `__syncthreads()` warten.

[47] siehe [opt]

6 Fazit und Ausblick

6.1 Fazit

Durch die Einbeziehung der GPU, ergeben sich für die Lösung von Programmierproblemen ganz neue Möglichkeiten. Die über die Jahre immer weiterentwickelte Hardware bietet Rechenpower, die einst nur Supercomputern vorenthalten war und ist nun relativ günstig für den Endverbrauchermarkt verfügbar. Zudem gestaltet sich die Nutzung, durch die in den letzten Kapiteln beschriebenen Schnittstellen und Tools, relativ einfach. Großes Potential liegt dabei besonders in den Java und .NET Wrappern, da sie den Einsatz der GPU für eine weitere Entwicklergruppe verfügbar machen.

Dokumentationen und Tutorials sind ausreichend[48] vorhanden. Sowohl Nvidia, als auch AMD bieten auf ihren Websiten ”Getting Started” Dokumente, Schnelleinstiege und ausführliche Referenzen zu den jeweiligen Frameworks an. Empfehlenswert für den Einstieg ist außerdem das Buch ”CUDA by Example”, siehe [SK10]. Für Dozenten und Studenten ist das OpenCL University Kit[49] von AMD interessant. Dieses bietet genug Material, um einen Kurs oder ein Modul in OpenCL Programmierung anzubieten oder sich per Selbststudium in die Thematik einzuarbeiten.

Um einen sinnvollen Einsatz der GPU, durch Berechnungen mit hohen Datenmengen zu ermöglichen, sollten mindestens 256 MB Speicher auf der GPU vorhanden sein. Ab einer Dimensionsgröße von 2800 - 2900 traten im gewählten Game of Life Beispiel, bei einer Geforce Quadro 140M Grafikkarte mit 128 MB Speicher, Anzeige- und Speicherfehler auf. Diese sind möglicherweise auf zu geringe Speichergröße zurückzuführen. Zwar sind bei dieser Dimensionsgröße erst ca. 30 von 128 MB belegt, allerdings wird auch für andere Anwendungen, z.B. zur Anzeige des Desktops, Grafikspeicher benötigt.

6.2 Ausblick

Die Entwicklung hat noch längst nicht das Maximum erreicht, immer mehr Firmen erkennen das Potential. So integriert Microsoft in den kommenden Visual C++ 11 Compiler mit C++ AMP[50], GPU Unterstützung direkt in C++. Die Implementierung benutzt derzeit die Direct Compute API aus DirectX, ist jedoch als offener Standard geplant, sodass auch Implementierungen für andere Betriebssysteme möglich sind. [C++] Viele

[48] in englischer Sprache, deutschsprachige Literatur zum Thema ist, abgesehen von einigen Artikeln in Fachzeitschriften, (noch) nicht verfügbar

[49] siehe http://developer.amd.com/zones/OpenCLZone/universities/Pages/default.aspx

[50] Accelerated Massive Parallism

bekannte Programme greifen für bestimmte Berechnungen bereits auf die GPU zurück, so z.B. Gimp und Photoshop für die Erzeugung von aufwendigen Filtern und Bildmanipulationen oder die kürzlich erschienene WinZIP Version 16.5 zur Dateikompression. Da GPGPU-fähige GPUs schon seit einigen Jahren auf dem Markt sind[51], kann der überwiegende Teil der Endverbraucher von der Rechenleistung profitieren. Doch nicht nur ”normale” Anwendungsprogramme nutzen die Entwicklung. Auch Medizin und Forschung kommt die Rechenleistung zugute. So konnte das NAMD[52] Programm der University of Illinois eine 3D-Simulation von Viren um das 12fache beschleunigen. [CUDa] Ein Algorithmus, welcher 3D-Modelle aus Ultraschallaufnahmen berechnet, wurde vom Unternehmen TechniScan auf eine Berechnungszeit von weniger als 20 min reduziert. Die Berechnung benötigte beim Einsatz von CPUs bis zu 3 mal so lang. [CUDb]

Eine weitere interessante Entwicklung ist der Versuch CPU und GPU zu ”verschmelzen”. AMD Fusion ist der Markenname für das Konzept, beide Modelle auf einem Chip zu vereinen. Diese Konstruktion wird von AMD APU (Accelerated Processing Unit) genannt. Auch Intel benutzt seit der Sandy Bridge - Prozessorgeneration, dieses Konzept für seine integrierten GPUs. Damit können die Vorteile beider Prozessoren kombiniert und je nach zu parallelisierendem Problem gewählt werden. Bei einer Berechnung mit vielen parallelen Aufgaben (Threads - Thread Level Parallism) die CPU, bei einer Berechnung mit vielen Daten (Data Level Parallism) die GPU. Anders ausgedrückt, das Hauptaugenmerk der CPU liegt bei Verwaltungs- und Steueraufgaben, während rechenintensive Prozesse auf die GPU ausgelagert werden. Da sich beide Prozessoren den Speicher teilen, entfällt der ansonsten zeitaufwändige Kopiervorgang zwischen den Speichern, allerdings wird die Bandbreite durch die gemeinsame Nutzung des Speicherinterfaces verlangsamt, sodass eine externe Grafikkarte mit eigenem Grafikspeicher unter Umständen trotzdem schneller ist. [MF12]

Bei den Frameworks wird auf längere Sicht der Einsatz von CUDA abnehmen. Eine Tendenz ist dabei schon jetzt zu beobachten, als Beispiel sei die Firma Adobe genannt, welche von CUDA auf OpenCL als neuen GPGPU Standard wechselte. [Ado] Die Verfügbarkeit allein für Nvidia GPUs, sowie das fehlende ”CPU Fallback”, sind für den praktischen Einsatz zu große Nachteile. Die in Kapitel 4.4 erwähnte Freigabe des CUDA Compilers zur Portierung auf ”Nicht Nvidia Systeme” erfolgte zu spät. Auf Windowssystemen wird DirectCompute als Bestandteil von DirectX Anwendung finden, während OpenCL, ähnlich wie OpenGL, den plattformunabhängigen Standard bilden wird.

[51] siehe Kapitel 4.2.1
[52] Nanoscale Molecular Dynamics

Anhang A: Einrichtung

A.1 CUDA

A.1.1 Windows

Um CUDA unter Windows einzurichten, muss neben einem aktuellen NVIDIA Treiber, lediglich das CUDA Toolkit heruntergeladen werden, sowie ggf. noch die GPU Computing SDK, welche viele Codebeispiele enthält. Alle Downloads befinden sich unter http://developer.nvidia.com/cuda-downloads. Als Entwicklungsumgebung bietet sich das Visual Studio von Microsoft an, denn hier kann nach der Installation des CUDA Toolkits ein neues CUDA Projekt mit allen nötigen Einstellungen über den Projektexplorer angelegt werden.

A.1.2 Linux

Unter Linux ist für die Einrichtung einer CUDA Entwicklungsumgebung etwas mehr Arbeit nötig. Nachfolgend werden nun die auszuführenden Schritte beschrieben.

1. **Download:** Unter http://developer.nvidia.com/cuda-downloads müssen aktueller Treiber, CUDA Toolkit und GPU Computing SDK für die entsprechende Distribution heruntergeladen werden. Treiber und Toolkit müssen mit root Rechten installiert werden.

   ```
   $ chmod 755 cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
   $ sudo ./cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
   $ chmod 755 devdriver_4.2_linux_64_295.41.run
   $ sudo ./devdriver_4.2_linux_64_295.41.run
   ```

2. **Setzen der $PATH Variable:** Der CUDA bin Ordner muss der $PATH Variable hinzugefügt werden. Dazu muss folgende Zeile

   ```
   $ export PATH=\$PATH:/usr/local/cuda/bin
   ```

   der .bashrc Datei hinzugefügt werden.

3. **Hinzufügen der CUDA Bibliotheken zum library pfad:** Eine Datei mit .conf Endung muss im Ordner

   ```
   $ /etc/ld.so.conf.d/
   ```

   angelegt werden. Sie muss folgende Einträge enthalten:

   ```
   $/usr/local/cuda/lib64
   $/usr/local/cuda/lib
   ```

   Mit dem Befehl

   ```
   $ sudo ldconfig
   ```

werden die Bibliotheken neu eingelesen.

4. **Einrichten des Compilers:** Das aktuelle CUDA Toolkit 4.2 unterstützt nur den gcc Compiler bis Version 4.6. Unter Umständen ist also ein Downgrade erforderlich oder eine Installation mehrerer Versionen nötig. Bei Verwendung mehrerer Versionen, sollte die für CUDA verwendete Version verlinkt und dem Compiler bekannt gemacht werden. Dazu erstellt man am Besten einen Ordner, in welchen die Links gesetzt werden:

   ```
   $ mkdir gcc-4.6
   $ cd gcc-4.6
   $ ln -s /usr/bin/g++-4.6 g++
   $ ln -s /usr/bin/gcc-4.6 gcc
   ```

   Anschließend sucht man im MakeFile common.mk unter

   ```
   $ ~/NVIDIA_GPU_Computing_SDK/C/common
   ```

   nach dem Eintrag

   ```
   $ NVCCFLAGS :=
   ```

   und trägt dort folgenden Wert ein:

   ```
   $ NVCCFLAGS := -ccbin ~/gcc-4.6/
   ```

   Damit wird der nvcc Compiler zur gcc Version 4.6 verlinkt.

Nun können mit dem nvcc Compiler .cu Dateien kompiliert werden.

```
$ nvcc game_of_life_gpu_shared.cu
```

A.2 OpenCL

Für die Verwendung von OpenCL, sind mehrere Implementationen verschiedener Hersteller verfügbar:

1. **Nvidia:** Die OpenCL Implementierung ist im CUDA Toolkit enthalten, siehe dazu Anhang A.1
2. **AMD:** Unter dem Namen Accelerated Parallel Processing (APP) SDK vertreibt AMD seine OpenCL Implementierung. Der Download erfolgt unter http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx. Damit kann OpenCL auf allen AMD CPUs genutzt werden, welche mindestens *SSE3* unterstützen. Bei Verwendung einer AMD GPU ist zusätzlich ein aktueller Grafiktreiber nötig, siehe http://support.amd.com/us/gpudownload/Pages/index.aspx. Eine Liste mit allen unterstützten AMD Grafikkarten befindet sich unter http://developer.amd.com/sdks/AMDAPPSDK/pages/DriverCompatibility.aspx
3. **Intel:** Um OpenCL auf Intel HD Grafikchips zu nutzen, wird unter http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/ eine entsprechende Entwicklungsumgebung angeboten.

4. **IBM:** Für IBM Hardware gibt es unter https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1 eine OpenCL Entwicklungsumgebung für Linux Systeme.

Je nach verfügbarer Hardware, ist die entsprechende Implementierung zu verwenden. Es ist auch möglich mehrere Implementierungen (z.B. Nvidia und Intel) einzurichten. Im OpenCL Quellcode, muss dann die zu verwendende Plattform mittels `clGetPlatformIDs` ausgewählt werden. Unter Linux wird eine OpenCL Anwendung mit dem Attribut -lOpenCL und dem gcc Compiler kompiliert.

```
$ gcc -lOpenCL game_of_life_opencl.c
```

Anhang B: Game of Life - CPU

Listing B.1: CPU-Implementierung ”Game of Life”

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define DIM 10
#define NUMBER_GENERATIONS 20

//Anzahl der Nachbarn einer Zelle bestimmen
int getNeighbors(int array[][DIM+2], int x, int y) {
  int neighbors;
  neighbors = array[x+1][y] + array[x-1][y]
              + array[x][y+1] + array[x][y-1]
              + array[x+1][y+1] + array[x-1][y-1]
              + array[x-1][y+1] + array[x+1][y-1];
  return neighbors;
}

void next_generation(int array[][DIM+2]) {
  int neighbors;
  int array_new[DIM+2][DIM+2] = {0};

  for (int x = 1; x <= DIM; x++) {
    for (int y = 1; y <= DIM; y++) {
      neighbors = getNeighbors(array, x, y);
      if (neighbors == 3 || (neighbors == 2 && array[x][y])) {
        array_new[x][y] = 1;
      } else {
        array_new[x][y] = 0;
      }
    }
  }
  for (int x = 0; x < DIM; x++) {
    for (int y = 0; y < DIM; y++) {
      array[x][y] = array_new[x][y];
    }
  }
}

int main(void) {
  int array[DIM+2][DIM+2] = {0};

  //befuellen des Arrays mit zufaelligen Werten
  srand( (unsigned)time(NULL) );
  for (int x = 1; x <= DIM; x++) {
      for (int y = 1; y <= DIM; y++) {
        array[x][y] = rand() % 2;
      }
```

```
48    }
49
50    for (int i = 0; i < NUMBER_GENERATIONS; i++) {
51
52      //Kopieren der aeussersten Spalten zu den gegenueberliegenden Temp
          Spalten
53      for (int x = 1; x<=DIM; x++) {
54         array[x][0] = array[x][DIM];
55         array[x][DIM+1] = array[x][1];
56      }
57
58      //Kopieren der aeussersten Zeilen zu den gegenueberliegenden Temp
          Zeilen
59      for (int y = 0; y<=DIM+1; y++) {
60         array[0][y] = array[DIM][y];
61         array[DIM+1][y] = array[1][y];
62      }
63
64      //grafische Ausgabe in textform
65      int total = 0;
66      for (int x = 1; x<=DIM; x++) {
67          for (int y = 1; y<=DIM; y++) {
68        total += array[x][y];
69    }
70      }
71      printf("lebende Zellen: %d\n", total);
72
73      next_generation(array);
74    }
75 }
```

Anhang C: Game of Life - GPU (CUDA)

C.1 Version 1 - Global Memory

Listing C.1: GPU-Implementierung ”Game of Life” CUDA - Version 1

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4  #include "cuda_runtime.h"
5  #include "device_launch_parameters.h"
6
7  #define DIM 10
8  #define NUMBER_GENERATIONS 20
9
10 //Anzahl der Nachbarn einer Zelle bestimmen
11 __device__ int getNeighbors(int *array, int id) {
12   int neighbors;
13   neighbors = array[id+(DIM+2)] + array[id-(DIM+2)]
14         + array[id+1] + array[id-1]
15         + array[id+(DIM+3)] + array[id-(DIM+3)]
16         + array[id-(DIM+1)] + array[id+(DIM+1)];
17   return neighbors;
18 }
19
20 //Kopieren der aeussersten Zeilen zu den gegenueberliegenden Temp Zeilen
21 __global__ void tempRows(int *array)
22 {
23   // id von 1 bis DIM
24   int id =blockIdx.x + 1;
25
26   if (id <= DIM)
27   {
28     array[(DIM+2)*(DIM+1)+id] = array[(DIM+2)+id];
29     array[id] = array[(DIM+2)*DIM + id];
30   }
31 }
32
33 //Kopieren der aeussersten Spalten zu den gegenueberliegenden Temp Spalten
34 __global__ void tempCols(int *array)
35 {
36   // id von 0 bis DIM+1
37   int id = blockIdx.x;
38
39   if (id <= DIM+1)
40   {
41     array[id*(DIM+2)+DIM+1] = array[id*(DIM+2)+1];
42     array[id*(DIM+2)] = array[id*(DIM+2) + DIM];
43   }
```

```
44 }
45
46 //Berechnung der neuen Generation und Speicherung in neuem Array
47 __global__ void next_generation(int *array, int *array_new) {
48   int neighbors;
49   int iy = blockIdx.y + 1;
50   int ix = blockIdx.x + 1;
51   int id = iy *(DIM + 2) + ix;
52
53   if (iy <= DIM && ix <= DIM) {
54     neighbors = getNeighbors(array, id);
55
56     if (neighbors == 3 || (neighbors == 2 && array[id])) {
57       array_new[id] = 1;
58     } else {
59       array_new[id] = 0;
60     }
61   }
62 }
63
64 int main(void) {
65
66   int *h_array;
67   int *dev_array;
68   int *dev_array_new;
69   int *dev_array_temp;
70
71   h_array = (int*)malloc(sizeof(int)*(DIM+2)*(DIM+2));
72
73   //befuellen des Arrays mit zufaelligen Werten
74   srand( (unsigned)time(NULL) );
75   for(int x = 1; x<=DIM; x++) {
76     for(int y = 1; y<=DIM; y++) {
77         h_array[x*(DIM+2)+y] = rand() % 2;
78     }
79   }
80
81   //Speicher auf der GPU reservieren
82   cudaMalloc((void**)&dev_array, (DIM+2)*(DIM+2)*sizeof(int));
83   cudaMalloc((void**)&dev_array_new, (DIM+2)*(DIM+2)*sizeof(int));
84
85   //Inhalt des host arrays in device array kopieren
86   cudaMemcpy(dev_array, h_array, (DIM+2)*(DIM+2)*sizeof(int),
87              cudaMemcpyHostToDevice);
88
89   //Groesse der Bloecke festlegen
90   dim3 blocks(DIM,DIM);
91
92   for (int n = 0; n < NUMBER_GENERATIONS; n++) {
93     //grafische Ausgabe in textform
94     int total = 0;
95     for (int x = 1; x<=DIM; x++) {
```

```
96        for (int y = 1; y<=DIM; y++) {
97          total += h_array[x*(DIM+2)+y];
98        }
99      }
100     printf("lebende Zellen: %d\n", total);
101
102     //Kopieren der temp Zeilen/Spalten
103     tempRows<<<DIM, 1>>>(dev_array);
104     tempCols<<<DIM, 1>>>(dev_array);
105
106     //Berechnung der naechsten Generation
107     next_generation<<<blocks,1>>>(dev_array, dev_array_new);
108
109     dev_array_temp = dev_array;
110     dev_array = dev_array_new;
111     dev_array_new = dev_array_temp;
112
113     //Zurueckkopieren der berechneten Werte im device array in host array
114     cudaMemcpy(h_array, dev_array, (DIM+2)*(DIM+2)*sizeof(int),
115                cudaMemcpyDeviceToHost);
116   }
117
118   //belegten Speicher wieder freigeben
119   free(h_array);
120   cudaFree(dev_array);
121   cudaFree(dev_array_new);
122
123   return 0;
124 }
```

C.2 Version 2 - Shared Memory

Listing C.2: GPU-Implementierung ”Game of Life” CUDA - Version 2

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <math.h>
5  #include "cuda_runtime.h"
6  #include "device_launch_parameters.h"
7
8  #define DIM 10
9  #define THREAD_NUMBER 5
10 #define NUMBER_GENERATIONS 20
11
12 //Anzahl der Nachbarn einer Zelle bestimmen
13 __device__ int getNeighbors(int array[][THREAD_NUMBER], int i, int j) {
14   int neighbors;
15   neighbors = array[i+1][j] + array[i-1][j]
16           + array[i][j+1] + array[i][j-1]
17           + array[i+1][j+1] + array[i-1][j-1]
```

```
18            + array[i-1][j+1] + array[i+1][j-1];
19    return neighbors;
20  }
21
22  //Kopieren der aeussersten Zeilen zu den gegenueberliegenden Temp Zeilen
23  __global__ void tempRows(int *array)
24  {
25    // id von 1 bis DIM
26    int id =blockIdx.x + 1;
27
28    if (id <= DIM)
29    {
30      array[(DIM+2)*(DIM+1)+id] = array[(DIM+2)+id];
31      array[id] = array[(DIM+2)*DIM + id];
32    }
33  }
34
35  //Kopieren der aeussersten Spalten zu den gegenueberliegenden Temp Spalten
36  __global__ void tempCols(int *array)
37  {
38    // id von 0 bis DIM+1
39    int id = blockIdx.x;
40
41    if (id <= DIM+1)
42    {
43      array[id*(DIM+2)+DIM+1] = array[id*(DIM+2)+1];
44      array[id*(DIM+2)] = array[id*(DIM+2) + DIM];
45    }
46  }
47
48  //Berechnung der neuen Generation und Speicherung in neuem Array
49  __global__ void next_generation(int *array, int *array_new) {
50    int neighbors;
51
52    int iy = (blockDim.y-2) * blockIdx.y + threadIdx.y;
53    int ix = (blockDim.x-2) * blockIdx.x + threadIdx.x;
54    int id = iy * (DIM+2) + ix;
55
56    int i = threadIdx.y;
57    int j = threadIdx.x;
58
59    // shared array
60    __shared__ int shared_array[THREAD_NUMBER][THREAD_NUMBER];
61
62    // Kopieren der Zellen ins shared array
63    if (ix <= DIM+1 && iy <= DIM+1)
64      shared_array[i][j] = array[id];
65
66    // threads synchronisieren
67      __syncthreads();
68
69    if (iy <= DIM && ix <= DIM) {
```

```
70      if(i != 0 && i !=blockDim.y-1 && j != 0 && j !=blockDim.x-1) {
71      neighbors = getNeighbors(shared_array, i, j);
72
73        if (neighbors == 3 || (neighbors == 2 && array[id])) {
74          array_new[id] = 1;
75        } else {
76          array_new[id] = 0;
77        }
78      }
79    }
80  }
81
82  int main(void) {
83
84    int *h_array;
85    int *dev_array;
86    int *dev_array_new;
87    int *dev_array_temp;
88
89    //Pruefen ob gewaehlte Threadzahl zulaessig ist
90    cudaDeviceProp deviceProp;
91    cudaGetDeviceProperties(&deviceProp, 0);
92    int maxThreads = deviceProp.maxThreadsPerBlock;
93    int threadNumberMax = (int)sqrt((double)maxThreads);
94    if (THREAD_NUMBER > threadNumberMax) {
95    printf("Die gewaehlte Threadzahl ueberschreitet die maximale Threadzahl
          der
96    verfuegbaren CUDA Device!\n");
97    printf("Maximal %d Threads sind moeglich.", threadNumberMax);
98    return -1;
99    }
100
101   h_array = (int*)malloc(sizeof(int)*(DIM+2)*(DIM+2));
102
103   //befuellen des Arrays mit zufaelligen Werten
104   srand( (unsigned)time(NULL) );
105   for(int i = 1; i<=DIM; i++) {
106     for(int j = 1; j<=DIM; j++) {
107         h_array[i*(DIM+2)+j] = rand() % 2;
108     }
109   }
110
111   //Speicher auf der GPU reservieren
112   cudaMalloc((void**)&dev_array, (DIM+2)*(DIM+2)*sizeof(int));
113   cudaMalloc((void**)&dev_array_new, (DIM+2)*(DIM+2)*sizeof(int));
114
115   //Inhalt des host arrays in device array kopieren
116   cudaMemcpy(dev_array, h_array, (DIM+2)*(DIM+2)*sizeof(int),
117              cudaMemcpyHostToDevice);
118
119
120   //Groesse der Bloecke bestimmen und festlegen
```

```
121   int blockSize = (int) ceil((float)DIM/THREAD_NUMBER);
122   dim3 blocks(blockSize, blockSize);
123   //Groesse der Threads festlegen
124   dim3 threads(THREAD_NUMBER,THREAD_NUMBER);
125
126   for (int n = 0; n < NUMBER_GENERATIONS; n++) {
127     //grafische Ausgabe in textform
128     int total = 0;
129     for (int x = 1; x<=DIM; x++) {
130       for (int y = 1; y<=DIM; y++) {
131         total += h_array[x*(DIM+2)+y];
132       }
133     }
134     printf("lebende Zellen: %d\n", total);
135
136     //Kopieren der temp Zeilen/Spalten
137     tempRows<<<DIM, 1>>>(dev_array);
138     tempCols<<<DIM, 1>>>(dev_array);
139
140     //Berechnung der naechsten Generation
141     next_generation<<<blocks,threads>>>(dev_array, dev_array_new);
142
143     dev_array_temp = dev_array;
144     dev_array = dev_array_new;
145     dev_array_new = dev_array_temp;
146
147     //Zurueckkopieren der berechneten Werte im device array in host array
148     cudaMemcpy(h_array, dev_array, (DIM+2)*(DIM+2)*sizeof(int),
149                cudaMemcpyDeviceToHost);
150   }
151
152   //belegten Speicher wieder freigeben
153   free(h_array);
154   cudaFree(dev_array);
155   cudaFree(dev_array_new);
156
157   return 0;
158 }
```

Anhang D: Game of Life - GPU (OpenCL)

D.1 Datei ”game_of_life_opencl.c”

Listing D.1: GPU-Implementierung ”Game of Life” - OpenCL

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <math.h>
5  #include <CL/opencl.h>
6  #include <sys/stat.h>
7
8  #define LOCAL_SIZE 5
9  #define NUMBER_GENERATIONS 20
10
11 int main(int argc, char* argv[])
12 {
13   int i,j;
14   int *h_array;
15   cl_mem dev_array;
16   cl_mem dev_array_new;
17   cl_mem dev_array_temp;
18
19   // Groesse der Dimension
20   int dim = 10;
21
22   // notwendiger Speicherplatz fuer Array
23   size_t number_cells_in_bytes = sizeof(int)*(dim+2)*(dim+2);
24
25   // array auf dem host speicher anlegen
26   h_array = (int*)malloc(number_cells_in_bytes);
27
28   cl_platform_id cpPlatform;          // OpenCL platform
29   cl_device_id device_id;             // device ID
30   cl_context context;                 // context
31   cl_command_queue queue;             // command queue
32   cl_program program;                 // program
33
34   //OpenCL kernels
35   cl_kernel k_next_generation, k_ghostRows, k_ghostCols;
36
37   // befuellen des Arrays mit zufaelligen Werten
38   srand((unsigned)time(NULL));
39   for(i = 1; i<=dim; i++) {
40       for(j = 1; j<=dim; j++) {
41           h_array[i*(dim+2)+j] = rand() % 2;
42       }
43   }
```

```
44
45  // Errorcode
46  cl_int err;
47
48  // Ermitteln der (erstmoeglichen) OpenCL platform
49  err = clGetPlatformIDs(1, &cpPlatform, NULL);
50  if (err != CL_SUCCESS) {
51    printf( "Error: Es konnte keine OpenCL platform gefunden werden!\n");
52    return EXIT_FAILURE;
53  }
54
55  // Ermitteln der (erstmoeglichen) gpu device
56  err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL)
        ;
57  if (err != CL_SUCCESS) {
58    printf("Warnung: Keine GPU konnte gefunden werden! Zur Berechnung wird
        die CPU verwendet. \n");
59    err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_CPU, 1, &device_id,
        NULL);
60    if (err != CL_SUCCESS) {
61      return EXIT_FAILURE;
62    }
63  }
64
65  // OpenCL context erstellen
66  context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
67  if (!context) {
68    printf("Error: Es konnte kein context erstellt werden!\n");
69    return EXIT_FAILURE;
70  }
71
72  // command queue fuer das gewaehlte device erstellen
73  queue = clCreateCommandQueue(context, device_id, 0, &err);
74  if (!queue) {
75    printf("Error: Es konnte keine command queue erstellt werden!\n");
76    return EXIT_FAILURE;
77  }
78
79  // externe OpenCL kernel quellcode datei oeffnen und daraus ein OpenCL
      Programm erstellen.
80  char *fileName = "kernels.cl";
81  FILE *fh = fopen(fileName, "r");
82  if(!fh) {
83    printf("Error: Datei \"kernels.cl\" konnte nicht geoeffnet werden!\n");
84    return 0;
85  }
86  struct stat statbuf;
87  stat(fileName, &statbuf);
88  char *kernelSource = (char *) malloc(statbuf.st_size + 1);
89  fread(kernelSource, statbuf.st_size, 1, fh);
90  kernelSource[statbuf.st_size] = '\0';
```

```
91  program = clCreateProgramWithSource(context, 1, (const char **) &
        kernelSource, NULL, &err);
92  if (!program) {
93    printf("Error: Es konnte kein OpenCL program aus dem Kernel erstellt
          werden \n");
94    return EXIT_FAILURE;
95  }
96
97  // das erstellte Programm kompilieren
98  err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
99  if (err != CL_SUCCESS) {
100   printf("Error: OpenCL program konnte nicht kompiliert werden!
          Fehlercode: %d\n", err);
101   return EXIT_FAILURE;
102 }
103
104 // GOL kernel aus dem erstellten Programm bauen
105 k_next_generation = clCreateKernel(program, "next_generation", &err);
106 if (!k_next_generation || err != CL_SUCCESS) {
107   printf("Error: GOL kernel konnte nicht angelegt werden! \n");
108   return EXIT_FAILURE;
109 }
110
111 // tempRows kernel aus dem erstellten Programm bauen
112 k_ghostRows = clCreateKernel(program, "ghostRows", &err);
113 if (!k_ghostRows || err != CL_SUCCESS) {
114   printf("Error: ghostRows kernel konnte nicht angelegt werden! \n");
115   return EXIT_FAILURE;
116 }
117
118 //  tempCols kernel aus dem erstellten Programm bauen
119 k_ghostCols = clCreateKernel(program, "ghostCols", &err);
120 if (!k_ghostCols || err != CL_SUCCESS) {
121   printf("Error: ghostCols kernel konnte nicht angelegt werden! \n");
122   return EXIT_FAILURE;
123 }
124
125 // Initialisieren der arrays auf dem device
126 dev_array = clCreateBuffer(context, CL_MEM_READ_WRITE,
        number_cells_in_bytes, NULL, NULL);
127 dev_array_new = clCreateBuffer(context, CL_MEM_READ_WRITE,
        number_cells_in_bytes, NULL, NULL);
128 if (!dev_array || !dev_array_new) {
129   printf("Error: Es konnte kein Speicher auf dem device angelegt werden!
          \n");
130   return EXIT_FAILURE;
131 }
132
133 // Schreiben der Werte auf dem host array ins device array
134 err = clEnqueueWriteBuffer(queue, dev_array, CL_TRUE, 0,
135     number_cells_in_bytes, h_array, 0, NULL, NULL);
136 if (err != CL_SUCCESS) {
```

```
137     printf("Error: Uebertragung der Daten vom Host zur Device nicht
            erfolgreich!\n");
138     return EXIT_FAILURE;
139   }
140
141   // Uebergeben der Parameter an GOL kernel
142   err  = clSetKernelArg(k_next_generation, 0, sizeof(int), &dim);
143   err |= clSetKernelArg(k_next_generation, 1, sizeof(cl_mem), &dev_array);
144   err |= clSetKernelArg(k_next_generation, 2, sizeof(cl_mem), &
        dev_array_new);
145   if (err != CL_SUCCESS) {
146     printf("Error: Kernel Parameter fuer GOL Kernel konnten nicht gesetzt
            werden!\n");
147     return EXIT_FAILURE;
148   }
149
150   // Uebergeben der Parameter an tempRows kernel
151   err  = clSetKernelArg(k_ghostRows, 0, sizeof(int), &dim);
152   err |= clSetKernelArg(k_ghostRows, 1, sizeof(cl_mem), &dev_array);
153   if (err != CL_SUCCESS) {
154     printf("Error: Kernel Parameter fuer tempRows Kernel konnten nicht
            gesetzt werden! \n");
155     return EXIT_FAILURE;
156   }
157
158   // Uebergeben der Parameter an tempCols kernel
159   err  = clSetKernelArg(k_ghostCols, 0, sizeof(int), &dim);
160   err |= clSetKernelArg(k_ghostCols, 1, sizeof(cl_mem), &dev_array);
161   if (err != CL_SUCCESS) {
162     printf("Error: Kernel Parameter fuer tempCols Kernel konnten nicht
            gesetzt werden! \n");
163     return EXIT_FAILURE;
164   }
165
166   // Setzen der kernel local und global sizes
167   size_t tempRowsGlobalSize, tempColsGlobalSize, localSize;
168
169   localSize = LOCAL_SIZE;
170
171   // Bestimmung der Workgroupanzahl
172   tempRowsGlobalSize = (size_t)ceil((float)dim/(float)localSize)*localSize;
173   tempColsGlobalSize = (size_t)ceil((float)(dim+2)/(float)localSize)*
        localSize;
174
175   size_t GolLocalSize[2] = {localSize, localSize};
176   size_t roundedGlobal = (size_t)ceil(dim/(float)localSize)*localSize;
177   size_t GolGlobalSize[2] = {roundedGlobal, roundedGlobal};
178
179
180   for (int n = 0; n<NUMBER_GENERATIONS; n++) {
181     // Ausfuehren der Kernel
```

```
182     err = clEnqueueNDRangeKernel(queue, k_ghostRows, 1, NULL, &
            tempRowsGlobalSize, &localSize, 0, NULL, NULL);
183     err |= clEnqueueNDRangeKernel(queue, k_ghostCols, 1, NULL, &
            tempColsGlobalSize, &localSize, 0, NULL, NULL);
184     err |= clEnqueueNDRangeKernel(queue, k_next_generation, 2, NULL,
            GolGlobalSize, GolLocalSize, 0, NULL, NULL);
185
186     if(n%2 == 1) {
187       err |= clSetKernelArg(k_ghostRows, 1, sizeof(cl_mem), &dev_array);
188       err |= clSetKernelArg(k_ghostCols, 1, sizeof(cl_mem), &dev_array);
189       err |= clSetKernelArg(k_next_generation, 1, sizeof(cl_mem), &
              dev_array);
190       err |= clSetKernelArg(k_next_generation, 2, sizeof(cl_mem), &
              dev_array_new);
191     } else {
192       err |= clSetKernelArg(k_ghostRows, 1, sizeof(cl_mem), &dev_array_new)
              ;
193       err |= clSetKernelArg(k_ghostCols, 1, sizeof(cl_mem), &dev_array_new)
              ;
194       err |= clSetKernelArg(k_next_generation, 1, sizeof(cl_mem), &
              dev_array_new);
195       err |= clSetKernelArg(k_next_generation, 2, sizeof(cl_mem), &
              dev_array);
196     }
197
198     if (err != CL_SUCCESS) {
199       printf("Error: Kernel konnte nicht erfolgreich ausgefuehrt werden!
              Fehlercode: %d\n",err);
200       return EXIT_FAILURE;
201     }
202
203     // Synchronisiert alle commands
204     clFinish(queue);
205
206     // Zurueckkopieren der Werte vom Device in den Hostspeicher
207     err =  clEnqueueReadBuffer(queue, dev_array, CL_TRUE, 0,
            number_cells_in_bytes, h_array, 0, NULL, NULL );
208     if (err != CL_SUCCESS) {
209       printf("Error: Uebertragung der Daten von Device zum Host nicht
              erfolgreich \n");
210       return EXIT_FAILURE;
211     }
212     //grafische Ausgabe in textform
213     int total = 0;
214     for (int x = 1; x<=dim; x++) {
215       for (int y = 1; y<=dim; y++) {
216         total += h_array[x*(dim+2)+y];
217       }
218     }
219     printf("lebende Zellen: %d\n", total);
220   }
221
```

```
222   // Speicher auf dem Host wieder freigeben
223   free(h_array);
224   // Speicher auf Device freigeben
225   clReleaseMemObject(dev_array);
226   clReleaseMemObject(dev_array_new);
227
228   return 0;
229 }
```

D.2 Datei ”kernels.cl”

Listing D.2: GPU-Implementierung ”Game of Life” - OpenCL (kernel)

```
1
2 //Kopieren der aeussersten Zeilen zu den gegenueberliegenden Temp Zeilen
3 __kernel void ghostRows(const int dim, __global *array) {
4   // id von 1 bis DIM
5   int id = get_global_id(0) + 1;
6
7   if (id <= dim)
8   {
9     array[(dim+2)*(dim+1)+id] = array[(dim+2)+id];
10    array[id] = array[(dim+2)*dim + id];
11  }
12 }
13
14 //Kopieren der aeussersten Spalten zu den gegenueberliegenden Temp Spalten
15 __kernel void ghostCols(const int dim, __global *array) {
16   // id von 0 bis DIM+1
17   int id = get_global_id(0);
18
19   if (id <= dim+1)
20   {
21     array[id*(dim+2)+dim+1] = array[id*(dim+2)+1];
22     array[id*(dim+2)] = array[id*(dim+2) + dim];
23   }
24 }
25
26 //Berechnung der neuen Generation und Speicherung in neuem Array
27 __kernel void next_generation(const int dim, __global int *array, __global
      int *array_new) {
28   int ix = get_global_id(0) + 1;
29   int iy = get_global_id(1) + 1;
30   int id = iy * (dim+2) + ix;
31
32   int neighbors;
33
34   if (iy <= dim && ix <= dim) {
35
36     neighbors = array[id+(dim+2)] + array[id-(dim+2)]
37                 + array[id+1] + array[id-1]
```

```
38                  + array[id+(dim+3)] + array[id-(dim+3)]
39                  + array[id-(dim+1)] + array[id+(dim+1)];
40
41      if (neighbors == 3 || (neighbors == 2 && array[id])) {
42        array_new[id] = 1;
43      } else {
44        array_new[id] = 0;
45      }
46    }
47 }
```

Anhang E: Game of Life GTK+

Um das Game of Life mittels GTK+ grafisch darzustellen, ist der Aufruf von CUDA Funktionen aus einer .c Datei nötig, da diese durch die Verwendung der GTK+ Bibliotheken separat kompiliert werden muss. Im folgenden Beispiel wird daher neben einer game_of_life.cu Datei eine gol_frontend.c Datei verwendet. Um die Dateien zu verknüpfen, muss eine Funktion mit gleichem Name in beiden Dateien vorhanden sein, in diesem Fall ist es die Funktion `wrapper_next_generation()`. Mittels des Bezeichners `extern`, wird jeweils auf die andere Datei verwiesen. Der Quellcode aus Beispiel C.1 muss dafür nur gering abgeändert werden:

Listing E.1: GPU-Implementierung ”Game of Life” CUDA GTK+ - Datei game_of_life.cu

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #define DIM 10
5
6  extern "C" int wrapper_next_generation(int h_array[]);
7
8  //Anzahl der Nachbarn einer Zelle bestimmen
9  __device__ int getNeighbors(int *array, int id) {
10   int neighbors;
11   neighbors = array[id + (DIM + 2)] + array[id - (DIM + 2)] + array[id + 1]
12       + array[id - 1] + array[id + (DIM + 3)] + array[id - (DIM + 3)]
13       + array[id - (DIM + 1)] + array[id + (DIM + 1)];
14   return neighbors;
15 }
16
17 //Kopieren der aeussersten Zeilen zu den gegenueberliegenden Temp Zeilen
18 __global__ void tempRows(int *array) {
19   // id von 1 bis DIM
20   int id = blockIdx.x + 1;
21
22   if (id <= DIM) {
23       array[(DIM + 2) * (DIM + 1) + id] = array[(DIM + 2) + id];
24     array[id] = array[(DIM + 2) * DIM + id];
25   }
26 }
27
28 //Kopieren der aeussersten Spalten zu den gegenueberliegenden Temp Spalten
29 __global__ void tempCols(int *array) {
30   // id von 0 bis DIM+1
31   int id = blockIdx.x;
32
33   if (id <= DIM + 1) {
34       array[id * (DIM + 2) + DIM + 1] = array[id * (DIM + 2) + 1];
35     array[id * (DIM + 2)] = array[id * (DIM + 2) + DIM];
36   }
```

```
37 }
38
39 //Berechnung der neuen Generation und Speicherung in neuem Array
40 __global__ void next_generation(int *array, int *array_new) {
41   int neighbors;
42   int iy = blockIdx.y + 1;
43   int ix = blockIdx.x + 1;
44   int id = iy * (DIM + 2) + ix;
45
46   if (iy <= DIM && ix <= DIM) {
47     neighbors = getNeighbors(array, id);
48
49     if (neighbors == 3 || (neighbors == 2 && array[id])) {
50       array_new[id] = 1;
51     } else {
52       array_new[id] = 0;
53     }
54   }
55 }
56
57 int wrapper_next_generation(int h_array[]) {
58
59   int *dev_array;
60   int *dev_array_new;
61
62   //Speicher auf der GPU reservieren
63   cudaMalloc((void**) &dev_array, (DIM + 2) * (DIM + 2) * sizeof(int));
64   cudaMalloc((void**) &dev_array_new, (DIM + 2) * (DIM + 2) * sizeof(int));
65
66   //Inhalt des host arrays in device array kopieren
67   cudaMemcpy(dev_array, h_array, (DIM + 2) * (DIM + 2) * sizeof(int),
68              cudaMemcpyHostToDevice);
69
70   //Groesse der Bloecke festlegen
71   dim3 blocks(DIM, DIM);
72
73   //Kopieren der temp Zeilen/Spalten
74   tempRows<<<DIM, 1>>>(dev_array);
75   tempCols<<<DIM, 1>>>(dev_array);
76
77   //Berechnung der naechsten Generation
78   next_generation<<<blocks,1>>>(dev_array, dev_array_new);
79
80   //Zurueckkopieren der berechneten Werte vom device array ins host array
81   cudaMemcpy(h_array, dev_array_new, (DIM + 2) * (DIM + 2) * sizeof(int),
82              cudaMemcpyDeviceToHost);
83
84
85   //belegten Speicher auf GPU wieder freigeben
86   cudaFree(dev_array);
87   cudaFree(dev_array_new);
88
```

```
89   return 0;
90 }
```

In Zeile 7 wird deklariert, dass die Funktion aus einer externen Datei aufgerufen wird. Die ehemalige main Funktion in Zeile 58, wurde nun dementsprechend umbenannt und erhält als Parameter ein Array, welches in der externen Datei ”befüllt” wird. Die gol_frontend.c Datei sieht folgendermaßen aus:

Listing E.2: GPU-Implementierung ”Game of Life” CUDA GTK+ - Datei gol_frontend.c

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <gtk-2.0/gtk/gtk.h>
6
7 #define DIM 10
8 #define NUMBER_CELLS (DIM+2)*(DIM+2)
9
10 //Verweis auf Methode in externer Datei
11 extern void wrapper_next_generation(int a[]);
12
13 //globals array enthaelt Informationen ueber lebende und tote Zellen
14 int array[NUMBER_CELLS] = {0};
15 //globales widget array enthaelt buttons welche die Zellen darstellen
16 GtkWidget *buttons[NUMBER_CELLS];
17
18
19 /**
20  * Callback funktion um naechste generation zu berechnen,
21  * Die Ausfuehrung erfolgt mittels der externen Methode
       wrapper_next_generation() auf der GPU
22  */
23 void next_generation(GtkWidget *button, gpointer user_data) {
24   wrapper_next_generation(array);
25   int i,j,n;
26   n=0;
27   for (i = 1; i <= DIM; i++) {
28     for (j = 1; j <= DIM; j++) {
29       if (array[i*(DIM+2)+j]) {
30         gtk_button_set_label(GTK_BUTTON(buttons[n]), "O");
31       } else {
32         gtk_button_set_label(GTK_BUTTON(buttons[n]), " ");
33       }
34       n++;
35       }
36    }
37 }
38
39 /**
40  * generiert zufaellige startpopulation im globalen array
41  */
```

```
42 void generate_random_start_population(void) {
43   int i,j;
44   srand( (unsigned)time(NULL) );
45   for(i = 1; i<=DIM; i++) {
46     for(j = 1; j<=DIM; j++) {
47       array[i*(DIM+2)+j] = rand() % 2;
48     }
49   }
50 }
51
52 /**
53  * Callback Funktion, zum Generieren einer neuen Generation per Button
54  */
55 void new_random_population(GtkWidget *button, gpointer user_data) {
56   generate_random_start_population();
57   int i,j,n;
58   n=0;
59   for (i = 1; i <= DIM; i++) {
60     for (j = 1; j <= DIM; j++) {
61       if (array[i*(DIM+2)+j]) {
62         gtk_button_set_label(GTK_BUTTON(buttons[n]), "O");
63       } else {
64         gtk_button_set_label(GTK_BUTTON(buttons[n]), " ");
65       }
66       n++;
67     }
68   }
69 }
70
71 /**
72  * main methode initialisiert grafische Darstellung und Callback Funktionen
73  */
74 int main(int argc, char *argv[])
75 {
76   int i,j,n;
77   GtkWidget *table;
78   GtkWidget *vbox;
79   GtkWidget *button;
80   GtkWidget *button2;
81   GtkWidget *label;
82   GtkWidget *window;
83
84   //buttons mit NULL initialisieren
85   for (i = 0; i < NUMBER_CELLS; i++) {
86     buttons[i] = NULL;
87   }
88   gtk_init(&argc, &argv);
89   window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
90   table = gtk_table_new(DIM, DIM, TRUE);
91   vbox = gtk_vbox_new(0, FALSE);
92   button = gtk_button_new_with_label("naechste Generation");
93   button2 = gtk_button_new_with_label("neue Zufallspopulation");
```

```
94
95    gtk_box_pack_start(GTK_BOX(vbox), table, FALSE, FALSE, 0);
96    gtk_box_pack_end(GTK_BOX(vbox), button, FALSE, FALSE, 0);
97    gtk_box_pack_end(GTK_BOX(vbox), button2, FALSE, FALSE, 0);
98    gtk_container_add(GTK_CONTAINER(window), vbox);
99
100   generate_random_start_population();
101
102   //generiertes Array auf Buttons uebertragen
103   n = 0;
104   for (i = 1; i <= DIM; i++) {
105     for (j = 1; j <= DIM; j++) {
106       if (array[i*(DIM+2)+j]) {
107         buttons[n] = gtk_button_new_with_label("O");
108         gtk_table_attach_defaults(GTK_TABLE(table), buttons[n], j-1,j,i-1,i
              );
109       } else {
110         buttons[n] = gtk_button_new_with_label(" ");
111         gtk_table_attach_defaults(GTK_TABLE(table), buttons[n], j-1,j,i-1,i
              );
112       }
113       n++;
114     }
115   }
116
117   //Callback Funktionen verknuepfen
118   g_signal_connect(button, "clicked", G_CALLBACK(next_generation), NULL);
119   g_signal_connect(button2, "clicked", G_CALLBACK(new_random_population),
        NULL);
120   g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);
121   gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
122   gtk_widget_show_all(window);
123
124
125   gtk_main();
126
127   return 0;
128 }
```

In Zeile 11 wird auf die externe Methode in der game_of_life.cu Datei verwiesen, bevor sie in Zeile 24 bei jedem Klick auf den Button aufgerufen wird.

Beim Kompilieren werden die beiden Dateien zuerst in Objektdateien kompiliert und anschließend in ein ausführbares Programm umgewandelt. Unter Linux sehen die dafür notwendigen Schritte folgendermaßen aus:

```
1 $  gcc -c gol_frontend.c `pkg-config --cflags --libs gtk+-2.0`
2 $  nvcc -c game_of_life.cu
3 $  gcc -o GOF -L/usr/local/cuda/lib64 -lcuda -lcudart `pkg-config --cflags
     --libs gtk+-2.0` *.o
```

Literaturverzeichnis

[Ado] *Adobe accelerates CS6 with OpenCL*. Website. http://semiaccurate.com/2012/04/24/adobe-accellerates-cs6-with-opencl/; Online am 24. Mai 2012.

[Anda] ANDERMAHR, WOLFGANG: *Bericht: Nvidia GF100: Technische Details*. Website. http://www.computerbase.de/artikel/grafikkarten/2010/bericht-nvidia-gf100-technische-details/3/; Online am 01. Dezember 2011.

[Andb] ANDERMAHR, WOLFGANG: *Test: ATi Radeon HD 5870*. Website. http://www.computerbase.de/artikel/grafikkarten/2009/test-ati-radeon-hd-5870/3/; Online am 01. Dezember 2011.

[Ben] *Geforce GTX 470 und GTX 480: Test der GF100-Generation mit SLI - Update: DX11-Techdemos von AMD*. Website. http://www.pcgameshardware.de/aid,743333/Geforce-GTX-470-und-GTX-480-Test-der-GF100-Generation-mit-SLI-Update-DX11-Techdemos-von-AMD/Grafikkarte/Test/; Online am 26. Januar 2012.

[Bit] *Why are AMD GPUs faster than Nvidia GPUs?* Website. https://en.bitcoin.it/wiki/Why_a_GPU_mines_faster_than_a_CPU#Why_are_AMD_GPUs_faster_than_Nvidia_GPUs.3F; Online am 26. Januar 2012.

[C++] *Microsoft publishes fancy-pants heterogeneous parallel GPGPU C++ AMP specification*. Website. http://arstechnica.com/information-technology/2012/02/microsoft-publishes-fancy-pants-heterogeneous-parallel-gpgpu-c-amp-specification/; Online am 24. Mai 2012.

[CiZ] *Computer in der Zeit der Wandlung*. Website. http://www.stickybit.de/wissen/computer/grundlagen/cpu/; Online am 01. Dezember 2011.

[CUDa] *CUDA + Forschung*. Website. http://www.nvidia.de/object/cuda_research_de.html; Online am 25. Mai 2012.

[CUDb] *CUDA + Medizin*. Website. http://www.nvidia.de/object/cuda_medical_de.html; Online am 25. Mai 2012.

[ES] *Die Entwicklung der Shader zu "4.0" (WGF)*. Website. http://alt.3dcenter.org/artikel/2004/09-28.php; Online am 24. November 2011.

[G25] *GeForce 256*. Website. http://www.nvidia.de/page/geforce256.html; Online am 03. November 2011.

[GPG] *GPGPU Computing - ein Überblick für Anfänger und Fortgeschrittene*. Website. http://www.planet3dnow.de/vbulletin/showthread.php?t=362621; Online am 25. November 2011.

[Gr110] *Grafikkarte/Grafikkarten*. Website, 2010. http://www.elektronik-kompendium.de/sites/com/0506191.htm; Online am 22. Oktober 2011.

[MeG] *Meilensteine in der Entwicklung moderner Grafikkarten*. Website. http://www.grafikkarten-rangliste.org/meilensteine-in-der-entwicklung-moderner-grafikkarten; Online am 22. Oktober 2011.

[MF12] MÜSSIG, FLORIAN und MARTIN FISCHER: *Raupentechnik - AMDs Notebook-Prozessor „Trinity" mit überarbeiteten Bulldozer-Kernen und flotter DirectX-11-Grafik*. c't Magazin für Computer Technik, (12):90–95, Mai 2012.

[MHH08] MÖLLER, TOMAS, ERIC HAINES und NATY HOFFMAN: *Real-Time Rendering*. A.K. Peters, 3 Auflage, 2008.

[ND10] NICKOLLS, JOHN und WILLIAM J. DALLY: *The GPU Computing Era*. IEEE Micro, March/April 2010.

[OHL+08] OWENS, JOHN D., MIKE HOUSTON, DAVID LUEBKE, JOHN E.STONE, SIMON GREEN und JAMES C. PHILLIPS: *GPU Computing*. Proceedings of the IEEE, 96(5), May 2008.

[opea] *Computing with NVDIA's CUDA and OpenCL*. Website. http://knol.google.com/k/computing-with-nvidia-s-cuda-and-opencl; Online am 16. April 2012.

[opeb] *Eine Einführung in OpenCL*. Website. http://www.amd.com/de/products/technologies/stream-technology/opencl/pages/opencl-intro.aspx; Online am 16. April 2012.

[opec] *OpenCL*. Website. http://faui36a.informatik.uni-erlangen.de/trac/puppet/wiki/OpenCL; Online am 16. April 2012.

[opt] *Visual Profiler Optimization Guide*. siehe Visual Profiler -> Hilfe.

[PS1] *Pipeline Stages (Direct3D 10)*. Website. http://msdn.microsoft.com/en-us/library/bb205123(VS.85).aspx; Online am 24. November 2011.

[RV8] *ATI Radeon HD 5870 Architecture Analysis*. Website. http://www.bit-tech.net/hardware/graphics/2009/09/30/ati-radeon-hd-5870-architecture-analysis/5; Online am 22. Dezember 2011.

[SaB] *Intel Core i7 3960X Extreme Edition im Test*. Website. http://ht4u.net/reviews/2011/intel_sandy_bridge_e_hexa_core/index8.php; Online am 01. Dezember 2011.

[Sch01] SCHAUFF, VOLKER: *Die Geschichte der Grafikkarte*. Website, 2001. http://alt.3dcenter.org/artikel/graka-geschichte/2.php; Online am 22. Oktober 2011.

[SK10] SANDERS, JASON und EDWARD KANDROT: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st Auflage, 2010.

[Smi] SMITH, RYAN: *NVIDIA's GeForce GTX 480 and GTX 470: 6 Months Late, Was It Worth the Wait?* Website. http://www.anandtech.com/show/2977/; Online am 22. Dezember 2011.

[US] *Unified Shader: Optimale Auslastung des Grafikchips*. Website. http://www.chip.de/artikel/DirectX-10-Realistische-3D-Szenen-in-Windows-Vista-6_24167441.html; Online am 24. November 2011.

[Vog07] VOGT, CARSTEN: *C für Java-Programmierer*. Hanser Fachbuchverlag 3-446-40797-8, 2007.

[wkG] *Nvidia-Geforce-400-Serie*. Website. http://de.wikipedia.org/wiki/Nvidia-Geforce-400-Serie; Online am 01. Dezember 2011.

[wkR] *ATI-Radeon-HD-5000-Serie*. Website. http://de.wikipedia.org/wiki/ATI-Radeon-HD-5000-Serie; Online am 01. Dezember 2011.

Stichwortverzeichnis

Glossar

BIOS *basic input/output system*, Firmware, welche im nichtflüchtigen Speicher der Hauptplatine abgelegt ist. Wird im Allgemeinen benötigt, um die Hardware ”funktionsfähig” zu machen.

Clipping In der Computergrafik das Abschneiden von Grundobjekten am Rand eines gewünschten Bildschirmausschnittes oder Fensters.

CUDA *Compute Unified Device Architecture*, eine von Nvidia entwickelte Technik, die es Programmierern erlaubt, Programmteile zu entwickeln, die durch den Grafikprozessor auf der Grafikkarte abgearbeitet werden. Die Programmierung erfolgt über das gleichnamige Framework und der Programmiersprache ”CUDA C”.

Culling ein Prinzip aus der 3D-Computergrafik. Hierbei wird getestet, ob ein betrachtetes Objekt aktuell sichtbar ist. Ziel ist es, nicht sichtbare Objekte möglichst früh zu verwerfen um dadurch einen Performancegewinn bei der Darstellung zu erreichen.

DDR *Double Data Rate*, Verfahren, mit dem Daten auf einem Datenbus mit doppelter Datenrate übertragen werden können.

DirectX Sammlung von Programmierschnittstellen für multimediaintensive Anwendungen (2D und 3D) auf der Windows-Plattform.

High Dynamic Range Rendering unter Berücksichtigung der in der Natur vorkommenden großen Helligkeitsschwankungen. Im Gegensatz zu den herkömmlichen 256 Helligkeitsabstufungen pro Farbkanal werden beim HDRR Farben intern mit ausreichend hoher Präzision repräsentiert, um einen sehr großen Bereich von Helligkeiten abzudecken. Dies ermöglicht die Darstellung starker Kontraste ohne übermäßigen Detailverlust und die Anwendung von Effekten wie der Simulation von Linsenstreuung.

JNI *Java Native Interface*, eine standardisierte Anwendungsprogrammierschnittstelle zum Aufruf von plattformspezifischen Funktionen bzw. Methoden aus der Programmiersprache Java heraus.

OEM *Original-Equipment-Manufacturer*, Hersteller von Komponenten oder Produkten, der diese in seinen eigenen Fabriken produziert, sie aber nicht selbst in den Einzelhandel bringt.

OpenCL *Open Computing Language* eine Schnittstelle für uneinheitliche Parallelrechner, die z.B. mit Haupt-, Grafik- und/oder digitalen Signalprozessoren ausgestattet sind, mit der zugehörigen Programmiersprache ”OpenCL C”.

OpenGL *Open Graphics Library*, Spezifikation für eine plattform- und programmiersprachenunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafik.

Polygon Ein Polygon ist eine geometrische Figur, die man erhält, indem man mindestens drei voneinander verschiedene Punkte in einer Zeichenebene (die Ecken) durch Strecken (die Kanten) miteinander verbindet, sodass durch den entstandenen Linienzug eine zusammenhängende Fläche umschlossen wird..

SSE3 *Streaming SIMD Extensions*, eine von Intel entwickelte Befehlssatzerweiterung der X86er-Architektur. Die Ziffer 3 bezeichnet dabei die 3.Version der Erweiterung.

Transform & Lighting Zwei Schritte der Grafikpipeline in der 3D-Computergrafik. ”Transform” bezeichnet die Transformation der Weltkoordinaten eines Vertex in zweidimensionale Bildschirmkoordinaten mittels einer Transformationsmatrix. ”Lighting” bezeichnet die Berechnung der Beleuchtung eines Bildpunktes, also ihrer Helligkeit und Farbe, nach dem Beleuchtungsmodell.

VLIW *Very Long Instruction Word*, eine Eigenschaft einer Befehlssatzarchitektur einer Familie von Mikroprozessoren. Ziel ist die Beschleunigung der Abarbeitung von sequentiellen Programmen durch Ausnutzung von Parallelität auf Befehls-Ebene.

Z-Buffering Verfahren der Computergrafik zur Verdeckungsberechnung, m die vom Betrachter aus sichtbaren dreidimensionalen Flächen in einer Computergrafik zu ermitteln.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 22. Juni 2012