Hauptseminar-Arbeit

Software für technische Systeme

Thema: Rechnen auf der Grafikkarte

Juni 2011

Autor: Florian Rückershäuser
Studiengang: Bachelor of Science Informatik
Fachsemester: 5

Inhaltsverzeichnis

1	Ziel dieser Arbeit	3
2	Funktionsweise und Aufbau von Grafikkarten	3
2.1	GPU	3
2.2	Bussystem	4
2.3	Grafikspeicher	4
2.4	RAMDAC	4
2.5	Sonstiges	5
3	Historische Entwicklung	5
3.1	Die Ursprünge der Grafikkarte	5
3.2	Parallele Mikroprozessoren	6
3.3	GPGPU	7
4	Parallelität	8
4.1	Zeitalter der Parallelität	8
4.2	Werkzeuge der Parallelität	10
5	CUDA	11
5.1	Einführung in CUDA	11
5.2	Funktionsweise von CUDA	12
5.3	Interne parallele Abarbeitung	13
5.4	CUDA Code	14
5.5	CUDA Einschränkungen	16
5.6	Alternativen zu CUDA	17
6	Fallbeispiele für die Berechnung auf der Grafikkarte	17
6.1	Fallbeispiel 1 Medizin	17
6.2	Fallbeispiel 2 Phyisk	18
6.3	Weitere Projekte	18

1 Ziele dieser Arbeit

Ziel dieser Arbeit ist ein grundlegendes Wissen und Verständnis über den Aufbau von Grafikkarten zu vermitteln. Hierbei soll lediglich eine reine Basis geschaffen werden, um weiterführende Beispiele und Entwicklung moderner Grafikkarten besser zu verstehen. Ebenfalls wird ein historische Überblick über die Entwicklungen neuer Grafikkarten und deren Nutzungsmöglichkeiten gegeben. Das Kernthema beschäftigt sich mit der Nutzung der Parallele Prozessoren für rechenintensive Algorithmen und gibt einen kleinen Einstieg in die dazugehörigen APIs. Abschließend wird diese Hausarbeit Fallbeispiele anführen, welche darlegen zu welcher Rechenleistung moderne Grafikkarten fähig sind.

2 Funktionsweise und Aufbau der Grafikkarte

Um die Grundprinzipien der Berechnungen einer Grafikkarte zu verstehen, empfiehlt es sich den prinzipiellen Aufbau einer Grafikkarte zu kennen. Im Folgenden werden die Hauptbestandteile einer Grafikkarte und die Kernkomponenten der GPU (Graphics Processing Unit) erläutert.

2.1 GPU

Die GPU ist die Hauptkomponente jeder Grafikkarte und ist maßgeblich für alle grafischen Berechnungen zuständig. Die GPU ist damit das Hauptrechenwerk für die Grafikberechnung und ist organisatorisch das Äquivalent zu der CPU (Central Processing Unit). Sie kann auf dem Motherboard (Integrated Graphics Processor) installiert sein, aber auch als externe Lösung, in Form einer Grafikkarte. Im Wesentlichen bestehen ihre Aufgaben aus 2D und 3D Grafikberechnungen, so wie auch komplexe mathematische Berechnungen in Form von GPGPU (General Purpose on Graphics Processor Unit). Prinzipiell beschränkt sich die GPU auf 2 Hauptaufgaben: Geometrie Erzeugung / Berechnung, welche in Form von Polygonen (bzw. eine Ansammlung von Dreiecken bei der Geforce Graphics Pipeline) und die Erzeugung von Pixeln [KiHw10, S. 22f.]. Diese Erkenntnis hat dazu geführt, dass man zwei wichtige Komponenten in die GPU integriert hat. Den Programmable vertex processor, welcher für die Geometrie zuständig ist und der Programmable fragment processor, welcher die Pixelerzeugung übernimmt.[KiHw10, S. 26ff.]
Durch die Einführung einer höheren Softwareabstraktionsschicht und die damit verbundenen APIs (DirectX und OpenGL) kann man diese beiden Prozessoren direkt ansprechen und mit ihnen Berechnungen durchführen. Unter anderem bieten moderne GPUs die Möglichkeit des Antialiasings, welche die Kanten der Polygone glättet und Anisotropes Filtern, welches für Rasterung der Texturen zuständig ist.

Moderne GPUs bestehen aus mehreren SMs (Streaming Multiprocessors), welche im Fall von Nvidia CUDA-Kerne genannt werden. Mehrere SMs werden zu einem Block zusammengefasst, bei der Geforce 8800er Reihe bilden zwei SMs einen Block. Jeder SM hat eine bestimmte Anzahl von SPs (Streaming Processors), welche sich die Steuerungslogik wie auch den Programmspeicher teilen [KiHw10, S. 8]. Aktuelle Grafikkarten besitzen bis zu 512 Stream Prozessoren.

2.2 Bussystem

Das Bussystem verbindet logisch die GPU mit den anderen Busteilnehmern. Der heute übliche Standart ist PCIe (Peripheral Component Interconnect express), welcher den AGP (Accelerated Graphics Port) ersetzt hat. Bei aktuellen Grafikkarten sind mit PCIe 2.0 und mit 16 Lanes theoretisch Datenraten bis zu 8000 MB/s möglich [Whit04, S.93ff].

2.3 Grafikspeicher

Der Grafikspeicher ist für die temporäre Speicherung der grafikrelevanten Daten zuständig. Es gibt ihn in 3 verschiedenen Ausführungen, als Reservierung des Hauptspeichers, als eigenen Speicherbaustein auf der Grafikkarte oder als Kombination aus den beidem. Der Grafikspeicher unterscheidet sich von dem Arbeitsspeicher im wesentlichen in der erhöhte Taktrate. Dadurch wird die Datentransferrate erhöht. Moderne Grafikkarten haben Datentransferraten von bis zu 177 Gbyte/s [iX,2011] **.** Diese erhöhte Geschwindigkeit ermöglicht Grafikkarten eine deutlich schnellere Berechnung von komplexen Daten, da diese schneller nachgeladen werden können. Durch diesen Vorteil und durch die Einführung von GPGPU ist im Vergleich zu einem Programm, welches auf der CPU alleine ausgeführt wird ein 10x Speed-Up möglich [KiHw10, S. 10f.]. Weitere Optimierungen am Code, welche die Datenzugriffe des Arbeitsspeichers wie auch die des Grafikspeichers mindern, und eine Verminderung des sequentiellen Teils des Codes, ermöglicht einen bis zu 100x Speed-Up [KiHw10, S. 12]**.** Der heute übliche Standard für Grafikkarten ist GDDR5 (Graphics Double Data Rate), welcher eine Weiterentwicklung des GDDR Grafikspeicher ist. Allgemein gehört der GDDR Speicher zu dem DDR-SDRAM-Standard, in dem die Taktfrequenz erhöht wurde. GDDR5 Speicher hat Taktfrequenzen von bis zu 4 GHz[Davi, 2011]. Die Größe des Speichers liegt im Durchschnitt der aktuellen Grafikkarten bei 2 GB. Dieser wird größtenteils zur Bildspeicherung benutzt aber auch für Berechnung wie zum Beispiel beim Rendern.

2.4 RAMDAC

Der RAMDAC-Chip (Random Access Memory Digital/Analog Converter) liest in regelmäßigen Abständen digitale Daten aus dem Framebuffer im Grafikspeicher aus und wandelt diese in analoge Bildsignale um, welche von Monitoren wiedergegeben werden können. Durch die Einführung von

digitalen Ausgängen wie zum Beispiel HDMI und DVI verlor der RAMDAC an Bedeutung. Die Bilder müssen daher nicht mehr Digital/Analog umgewandelt werden. Bei dieser Art von Ausgängen kommt ein TMDS-Modul (Transition Minimized Differential Signaling) zu tragen, welches die digitalen Daten aus dem Grafikspeicher zu anderen Geräten transportiert.

2.5 Sonstiges

Erwähnenswert sind die Ausgänge. Heute übliche Standards sind VGA (Video Graphics Array), als heute noch einziger analoge Ausgang, DVI (Digital Visual Interface), HDMI (High Definiton Multimedia Interface) und DisplayPort. Ebenso nennenswert sind die Kühlervarianten, welche die Grafikkarte vor Überhitzung schützt. Grafikkartenkühlsysteme können aus elektrisch angetriebenen Lüftern (Axial- Radiallüfter), aus passiven Kühlkörper oder als Wasserkühlung bestehen. [Vött,2011, S.8]. Keine direkte Hardware, aber dennoch sehr elementar für das Verständnis von Grafikkarten, ist die sogenannte Grafik-Pipeline. Die Grafik-Pipeline ist eine Abarbeitungskette, welche für die Erzeugung von Grafik nötig ist. Sie besteht aus mehreren Unterschritten. Im Prinzip ist es notwendig, das die CPU der GPU Geometrie-und Pixel-Daten zur Verarbeitung übergibt und die Grafikkarte, nach Ablauf dieser Kette, Grafik erzeugt. Im nächsten Kapitel wird u.a. darauf eingegangen, an welchen Stellen der Abarbeitungskette eingegriffen werden kann um die Grafikerzeugung zu optimieren bzw. zu vereinfachen [KiHw10, S. 23f.].

3 Historische Entwicklung

Im folgenden Kapitel wird auf die Entwicklung von Grafikkarten und CPUs eingegangen. Ebenso auf die parallele Entwicklung von Software welche die Grafikkarte nutzen kann. Explizit wird darauf eingegangen wie der GPGPU Trend zustande kam.

3.1 Die Ursprünge der Grafikkarte

Bis in die frühen 90er wurde Grafik in handelsüblichen Computern von der CPU erzeugt. Es gab zwar bereits Computerspiele und Büroanwendungen die 3D und 2D Grafik verwendeten, dennoch musste diese aufwendig erzeugt werden. Ein weiterer Aspekt war der Halbleiter Preis bzw. der allgemeine Preis für Hardware, wodurch sich eine Entwicklung und Benutzung einer Grafikkarte nicht rentierte. Im Jahr 1992 wurde die erste OpenGL-API von SGI (Silicon Graphics) auf den Markt gebracht. Mit dieser API hatte man sich eine Erleichterung der Grafik-Erzeugung erhofft. Dennoch musste die Grafik aufwendig mit der CPU erzeugt werden, wodurch sich die Berechnungszeit für andere Prozesse verminderte und das Gesamtsystem verlangsamte [SaKa11,S.4]. Diesen Nachteil behob 1996 die Firma 3dfx, indem sie den ersten 3D-Beschleuniger auf den Markt gebracht hatte, die Voodoo Graphics oder auch Voodoo 1 [Vött,2011, S.6] . Dieser 3D-Beschleuniger war als Erweiterungskarte verfügbar und war die erste klassische Grafikkarte. Mit der Einführung die-

ser Karte konnte man die CPU entlasten und die Grafikberechnung beschleunigen, da diese Grafikkarte bereits einen schnelleren Grafikspeicher wie auch eine geänderte GPU bzw. Pixelprozessor besaß. Diese Grafikkarte wurde größtenteils für Computerspiele benutzt, welche auch noch auf die Voodoo 1 angepasst werden konnten, mit Hilfe der von 3dfx bereit gestellten API Glide [Smit, 1998]. Die Voodoo 1 hatte allerdings Nachteile, da sie die Geometrie-Manipulationen von der CPU berechnen lassen musste. 1 Jahr vor der Veröffentlichung der Voodoo 1 veröffentlichte im Zuge von Windows 95 Microsoft ihre erste API DirectX, welche auch in der Version 2.0 3D-Funktionalität mit sich brachte. Bis zum Jahre 1999 beschränkten sich Grafikkarten darauf Geometriedaten, von der CPU kommend, mit Pixel zu bestücken und diese ruckelfrei darzustellen. Dies änderte sich mit der Veröffentlichung der Nvidia Geforce. Mit dieser Grafikkarte war es möglich Pixel zu transformieren und zu beleuchten. Dies ermöglichte die Transform & Lightning-Einheit, welche dann zum elementaren Bestandteil der Nvidia Graphics Pipeline wurde. Mit dieser Einheit wurde die CPU nahezu komplett von der Grafikerzeugung entlastet. Ebenso konnte die Grafikkarte für diese Art von Matrix- und Vektor-Berechnungen angepasst und optimiert werden. Durch die Einführung der Transform & Lightning-Einheit entwickelte Microsoft ihr DirectX weiter. Mit der Veröffentlichung von DirectX 8.0 im Jahr 2001 war es möglich, den Vertex- und Pixel-Shader per API-Schnittstelle anzusprechen. Bereits hier gab es Versuche das Potenzial und die Geschwindigkeit der Grafikkarte ausnutzen, indem man die Shader Berechnungen durchführen lies, welche nicht an der Grafikerzeugung beteiligt waren [SaKa11,S.5]. Es stellte sich aber schnell heraus, dass dies sehr komplex war, da die zu berechnenden Daten in Geometrie- und Pixel-Werte umgewandelt werden mussten. Ebenfalls war das Ergebnis neu zu interpretieren, da dies meist Grafik-Daten entsprach. Als weiteres Problem erwies sich das Ganz- und Gleitkommazahlen verschiedene und auch ungewohnte Größen aufwiesen und von Hersteller zu Hersteller unterschiedlich groß waren. So war es zum Beispiel möglich bei einer Ati Radeon 9700 einen 24-bit Gleitkomma Pixelprozessor anzusprechen und bei einer gleichwertigen Geforce FX einen 32-bit Gleitkomma Prozessor. Durch dieses Problem musste ein höherer Arbeitsaufwand geleistet werden, um bei verschiedenen Grafikkarten den selben Effekt zu erzielen. Schließlich stagnierte dahingehend die Entwicklung, Berechnungen auf der Grafikkarte ausführen zu wollen [KiHw10, S. 26f.].

3.2 Parallele Mikroprozessoren

Anfang 2000 bemerkte man, dass eine stetige Erhöhung der Taktfrequenz der CPU nicht möglich war. Die Erhöhung der Taktfrequenz stieß auf das Problem, dass die Kühlung der CPU mit handelsüblichen Kühlmethoden, nicht ausreichte. Ebenfalls war aus Energieeffizienzgründen ein proportionales Problem aufgetreten, da durch die steigende Taktrate eine steigende Spannung auftrat, welche dazu führte das der Stromverbrauch anstieg. Durch die parallele Entwicklung im Server-Bereich war bekannt, dass man durch mehrere CPUs bzw. CPU-Kerne und angepasster Software mit der gleichen CPU bei gleichbleibender Taktrate, Erfolge erzielen konnte. Dies führte 2003 dazu, dass im CPU Bereich der erste Doppelkern-Prozessor in Entwicklung ging. Mit diesem erhoffte man sich, trotz gleichbleibender CPU-Taktfrequenz, eine Leistungssteigerung. 2005 kam der erste

Dual Core Prozessor von Intel auf den Markt. Durch die Einführung eines parallelarbeitenden Mikroprozessors stiegen auch die Anforderungen an parallele Programmierung. Betriebssysteme und Programme wurden an diese neue Anforderungen angepasst [KiHw10,S.1f]. Dies führte dazu, dass Nvidia Forschung betrieb und versuchte Berechnungen auf der Grafikkarte auszuführen und diese so einfach und so portabel, also auf jeder Nvidia Grafikkarte, wie möglich anzubieten. 2006 brachte Nvidia daher die Geforce 8800 auf den Markt. Mit der Geforce 8800 wurde eine Architektur entwickelt, welche sich zur Berechnung von komplexen Daten eignete. Zunächst wurde die Grafik-Pipeline so abgeändert, dass die einzeln ansprechbaren Prozessoren, wie Vertex und Pixelshader, zu einem Verbund von vereinheitlichten Prozessoren zusammengefasst wurden. Die Pipeline wurde dadurch physikalisch gesehen zu einem wiederkehrenden Kreislauf durch die Prozessoren. Insgesamt werden bei einer Grafikberechnung und Erzeugung diese einheitlichen Prozessoren dreimal angesprochen, mit verschiedenen vorgegeben grafischen Veränderungen bzw. Funktionen zwischen den Durchläufen. Durch die Einführung des Verbunds der Einheitsprozessoren, konnte man nun zum Beispiel dynamisch unterschiedliche Render-Algorithmen verwenden und die Prozessoren unterschiedlich stark und zeitlich unabhängig voneinander ausnutzen. Dies führt auch zu einer Verbesserung der Grafikspeicherzugriffe, da in jedem Durchlauf dynamisch Speicher allokiert werden und bei Bedarf angepasst werden konnte. Zeitgleich mit der Geforce 8800 wurde auch DirectX 10 veröffentlich. Mit DirectX 10 war es möglich Geometry-Shader zu benutzen. Dies war aus rein effizienter Sicht ein großer Schritt, da es mit Hilfe von einem Geomety-Shader möglich war neue Geometrie eigenständig zu erzeugen, was bis dato nicht möglich war. Durch diese neue Möglichkeit entwickelte Nvidia ihre Einheitsprozessoren. Nvidia erhöhte die Taktfrequenz der Prozessoren und passte diese den neuen Gegebenheiten an, verschiedenste Render-Algorithmen ausführen zu lassen[KiHw10,S.29].

3.3 GPGPU

Nvidia entwickelte die Einheitsprozessoren um eine hohen Grad an Parallelität zu gewinnen und dadurch einen höheren Speedup zu erzielen. Im Hinblick auf die Ausführung dynamischer Render-Algorithmen entwickelte Nvidia eine API, welche nicht nur Render-Algorithmen sondern nahezu alle rechenintensive Algorithmen mit einem hohen Grad an Parallelität ausführen lässt. Nvidia nannte diese API CUDA (Compute Unified Device Architecture), welche 2007 offiziell vorgestellt wurde. Mit CUDA ist es möglich die Rechenkapazität einer Grafikkarte ohne großes Einwirken auf die Hardware auszunutzen[KiHw10,S.7]. Bereits nach kurzer Entwicklungszeit entstanden die ersten angepassten Algorithmen für diese API und nach der Erkenntnis des Potenzials, brachte Nvidia ein Jahr später die erste Generation der Nvidia Tesla-Grafikkarten auf den Markt. An Tesla Grafikkarten ist die Besonderheit, dass diese ohne Ausgänge ausgeliefert werden, d.h. ohne die Möglichkeit Monitore anschließen zu können. Daher sind Tesla-Karten nur für den Gebrauch von GPGPU gedacht und können zur Rechenunterstüzung in handelsüblichen PCs verbaut werden, aber auch in Geräten, welche 3D Ultraschall erzeugen. Ati versuchte sich ebenfalls auf diesem Gebiet und stellte im selben Jahr ihre API vor, welche FireStream genannt wurde. FireStream gilt

aber als längst überholt, da der Hersteller nur sporadisch Erweiterungen auf den Markt bringt. Infolge der großen Akzeptanz und der enormen Leistungsausbeute, versprach sich Apple davon Teile ihres Betriebssytems von der Grafikkarte berechnen zu lassen, ohne aber auf einen Grafikkartenhersteller angewiesen zu sein. Daher beauftragte Apple die Firma Khronos Group einen Standart zu erarbeiten, der herstellerunabhängig ist, welcher aber auf die Bedürfnisse und die Erfahrungen der jeweiligen Hersteller eingeht. Das Ergebnis war OpenCL (Open Computing Language), woran die Großen Grafikkartenhersteller AMD(ATI) und Nvidia maßgeblich beteiligt waren, aber auch Firmen wie IBM und Intel ihren Beitrag dazu lieferten [KiHw10,S.206]. Die Erkenntnis Teile des Betriebssystems von der Grafikkarte berechnen lassen zu können bewegte Microsoft dazu ebenfalls eine eigene API zu veröffentlichen. Sie wurde 2010 unter dem Namen DirectCompute vorgestellt. Zum jetzigen Zeitpunkt ist aber relativ unbekannt, welchen Umfang diese API bekommt und wie man diese benutzen kann. [KiHw10,S.205].

4 Parallelität

Im folgenden Kapitel, wird erläutert warum Parallelität einen enormen Faktor zur Leistungssteigerung beiträgt, wie man sie anwendet und versteht und wie diese Übertragbar ist auf Grafikkarten. Ebenfalls wird ein Vergleich zwischen CPUs und GPUs genannt, welcher verdeutlicht zu welcher Rechenleistung Grafikkarten im derzeitigen Stadium fähig sind.

4.1 Zeitalter der Parallelität

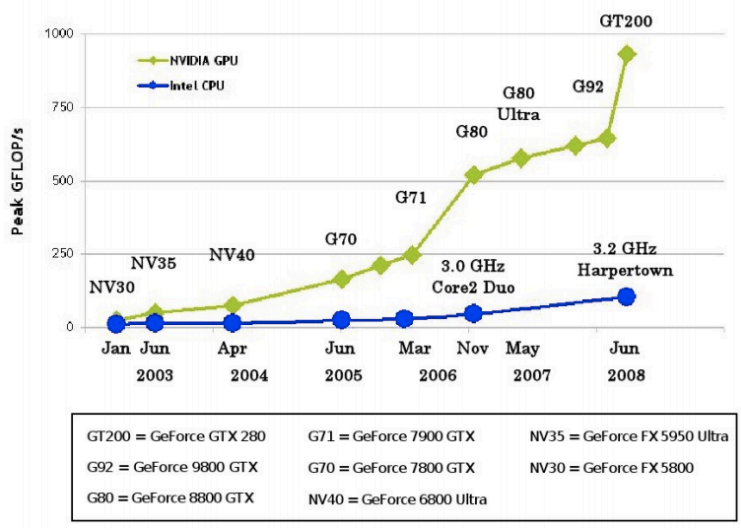
Einführend zu der historischen Entwicklung von GPGPU sei ein Zitat genannt welches sich auf die Entwicklung von CUDA bezieht: „*There was a time in the not-so distant past when parallel computing was looked upon as an „exotic“ pursuit and typically got compartmentalized as a specialty within the field of computer science This perception has changed in recent years. The computing world has shifted to the point where far from being an esoteric pursuit, nearly every aspering programmer needs training in parrallel programming to be fully effective in computer science“.* [SaKa11,S.1] Im Laufe der Entwicklung von CPUs im Desktop-Bereich stieß man an Grenzen, da die Taktrate nicht beliebig erhöht werden konnte. Infolge dieser Problematik wurde die Entwicklung von Mehrkern-Prozessoren vorangetrieben, welche bereits bekannt waren aus dem Server-Bereich. Man erhoffte sich theoretisch eine erhöhte Leistungssteigerung, welche aber nicht ohne Anpassung am Code möglich war. In den Entwicklungsphasen der CPU, wurde zeitgleich die Entwicklung des ausführbaren Codes von Betriebssystemen vorangetrieben. Mit der Entwicklung dieser angepassten Betriebssysteme konnte die CPU parallel mehrere Prozesse ausführen. Aber nicht nur die Hersteller solcher Betriebssysteme profitierten von dieser Leistungssteigerung, sondern auch die Entwickler von Anwendersoftware, da diese größtenteils Threads benutzen, welche ebenfalls auf Kerne der CPU verlagert werden können und damit Echt-Parallel ausgeführt werden können. Die Entwicklung und Akzeptanz solcher parallelen Software, führte die großen Hardwarehersteller dazu vermehrt Mehrkern-Prozessoren zu verbauen um eine Verbesserung der Leistung

herbeizuführen. Wie bereits oben erwähnt führte gerade das Nvidia dazu, ihre Grafik-Pipeline so umzubauen, dass sie die neuen Einheitsprozessoren mit Hunderten von diesen Prozessorkernen ausnutzen. In der folgenden Tabelle ist ein Vergleich zwischen einer modernen CPU und einer aktuellen Grafikkarte mit solch einer GPU dargestellt.

Prozessor/ Grafikkarten-typ	Intel Core i7-980X Gulftown	Nvidia Geforce GTX 580
Taktrate	3,33 GHz	772 MHz
Prozessorkerne	6	512
GFLOPS	107,58	1581

Quelle: In Anlehnung an SiSoftware Sandra Gflops Test und Pc Games Hardware Ausg. 05/11

Wie man an dieser Tabelle erkennt ist die Taktrate einer CPU wesentlich höher im Vergleich zu einer Grafikkarte. Dies lässt sich damit begründen, dass eine CPU mehrere Zykluswechsel, also mehreren Prozessen, Prozessorzeit abgeben muss und dies sich bei einer hohen Taktrate positiv auswirkt. Wohingegen bei einer Grafikkarte nahezu keine Zykluswechsel passieren. Ein gravierender Unterschied lässt sich bei den Prozessorkernen feststellen. Während es bei modernen Desktop-CPUs üblich ist 6-8 Kerne zu haben gibt es im Grafikkarten-Bereich bis zu 512 Kerne. Wie bereits oben erwähnt, ist dies eine Umbaumaßnahme von Nvidia gewesen, um verschiedenste Render-Algorithmen in einem höchstmöglichen Grad der Parallelität ausführen zu lassen. Weit mehr aussagekräftig als die Anzahl der Kerne ist die GFLOPs Anzahl. Die GFLOPS (Giga Floating Point Operations Per Second) sind, wie der Name schon sagt, die Anzahl der Gleitkomma-Berechnungen pro Sekunde mit dem Faktor 10^9. Anhand dieses Beispiels sieht man schnell zu welcher Rechenleistung Grafikkarten fähig sind. Logischerweise ist dies ein Grund der massiv parallelen Grafikkarten-Kerne. Kurze Zeit nach Veröffentlichung von CUDA wurde in den Medien davon berichtet das die CPU bald von der GPU abgelöst wird, da sie so eine enorme Rechenleistung hat [Gart,2008]**.** Diese Annahme ist natürlich im Hinblick auf die unterschiedliche Arbeitsweise der CPU im Gegensatz zur GPU falsch. Ein neuer Ansatz ist es CPUs mit GPUs zu kombinieren bzw. zu verschmelzen, in dem man diese beiden Komponenten in einem Chip zusammenfügt, welcher dann APU (Accelerated Processing Unit) genannt wird. Diese Idee heißt AMD „AMD Fusion“ und wurde bereits 2011 als Innovation für Netbooks vorgestellt [Jung2011,S.21f]. In der folgenden Grafik wird dargestellt wie sich im Laufe der Entwicklung der Grafikkarten sich die GFLOPS vermehrt haben im Vergleich zu Desktop-CPUs. Es werden folglich nur Grafikkarten der Marke Nvidia, wie auch nur CPUs des Herstellers Intel angeführt, da diese maßgeblich an der Entwicklung im Bereich der GPGPU beteiligt waren.



Quelle: Cuda Präsentation Uni Kassel.

Wie man sehen kann ist ein deutlicher Sprung von dem Grafikkartenchip G71 zu G80 vorhanden, welcher sich auf die Entwicklung der Nvidia Einheitsprozessoren wie auch die Abänderung der Grafikpipeline zurückführen lässt. Ebenfalls die zweite Steigerung von dem Grafikchip G92 zu GT200 lässt sich auf die Anzahl der Einheitsprozessoren zurückführen. Während der G92 128 Prozessoren hat, verfügt der GT200 über 240 Prozessoren. Ebenfalls besitzt der GT200 eine Speicheranbindung von 448 Bit im Gegensatz zu den nur 256 Bit des G92. Zudem ist in der Grafik erkennbar, dass eine vergleichbare CPU von der Rechenleistung äquivalent zu einer Grafikkarte ist, welche 4 Jahre älter ist. Aber wie bereits oben erwähnt war dies kein Ziel der Entwicklung der CPUs.

4.2 Werkzeuge der Parallelität

Um den hohen Grad der Parallelität von Grafikkarten ausnutzen zu können, bedarf es einiger Vorkenntnisse und Empfehlungen. In erster Linie ist ein Umdenken erforderlich. Parallele Programme müssen vom Aufbau her komplett neu organisiert werden. In der Literatur wird von einem Pfirsich-Modell gesprochen [KiHw10,S.12] **.** Das Pfirsich-Modell beschreibt wie Programme aufgebaut werden müssen, damit die Rechenleistung von parallelen Prozessoren bestmöglich ausgenutzt wird. Bestandteile des Pfirsich-Modells sind der Kern und das Fleisch. Das Fleisch ist der parallele Teil eines Programms. Im Gegensatz zu dem Kern ist dieser weitaus größer. Daher sollte ein Programm möglichst viele Berechnungen parallel ausführen, weshalb findet die Ausführung auf einer Grafikkarte oder ähnlichen Multiprozessorsystemen statt. Der Kern repräsentiert den sequentiellen Teil eines Programms, welcher von der CPU ausgeführt wird. Der sequentielle Teil sollte größtenteils sequentiell bleiben und man sollte sich bemühen keine Optimierung zu unternehmen, um diesen in irgendeiner Form zu einem parallelen Teil umzuformen. Das Pfirsich-Modell beschreibt wie eine Art Software zu entwickeln ist und diese auf verschiedene Komponenten eines Systems zu verlagern. Durch dieses Modell ist es möglich, die Stärken von CPU und GPU softwareseitig zu

kombinieren, denn die CPU ist dafür ausgelegt den sequentiellen Teil zu managen und auszuführen, während die GPU schnelle und massiv parallele Berechnungen durchführt. Ebenfalls empfehlenswert ist es den Grad der Parallelität in Programmen möglichst hoch zu halten. Der Grad der Parallelität bestimmt wieviel Berechnungen parallel ausgeführt werden können. Ist der Grad gering so ist die maximale Leistungsausbeutung geringer. Weitaus wichtiger als Programmierrichtlinien ist die zu programmierende Hardware. Wie bereits oben erwähnt benötigt man hierfür eine Multiprozessorsystem, welches sich durch Schnittstellen ansprechen lässt. Im besten Fall ist eine API vorhanden um sich einen Arbeitsaufwand von hardwarespezifischen Zugriffe zu vermindern.

5 CUDA

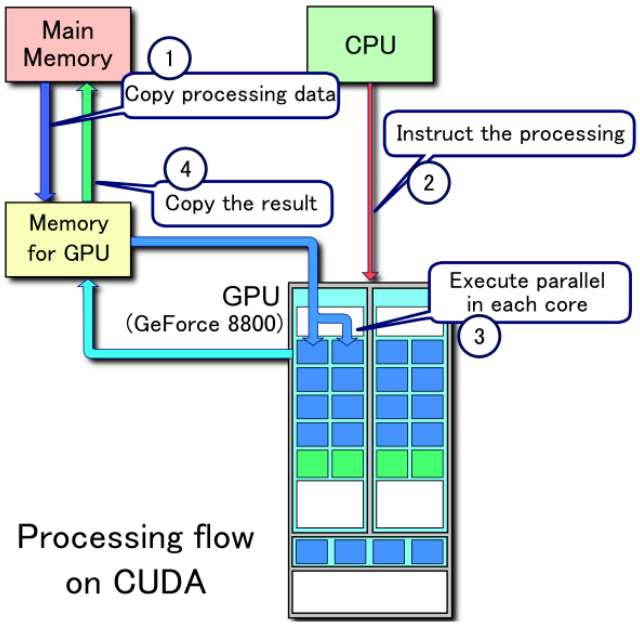
Als großes Beispiel für die Nutzung von GPGPU wird im folgenden Kapitel CUDA vorgestellt. CUDA gilt momentan als die größte Innovation für das Rechnen auf der Grafikkarte und ist zumindest in diesem Bereich weit verbreitet. Im Folgenden wird eine prinzipielle Übersicht, was CUDA ist und wie es funktioniert gegeben.

5.1 Einführung in CUDA

CUDA ist in erster Linie eine auf C basierende API, welche Funktion hinzufügt, aber auch Funktionen einschränkt. Mit CUDA ist es möglich, beliebige Algorithmen auf der Grafikkarte auszuführen und dies massiv parallel. Als Vorraussetzung für die Nutzung von CUDA benötigt man eine Grafikkarte mit der oben bereits erwähnten CUDA-Hardware-Architektur, welche aus einem Verbund von hunderten Einheitsprozessoren besteht. Eine CUDA-Hardware-Architektur bieten die Karten von Nvidia, welche nach der Geforce 8800er Reihe (8800er Reihe mit einbezogen) gefertigt wurden. Vor allem in der Forschung wurden bereits große Erfolge erzielt. So schreibt zum Beispiel der Hersteller selbst auf seiner Website: *„Im Bereich wissenschaftlicher Forschung wurde CUDA mit Begeisterung aufgenommen. CUDA beschleunigt jetzt zum Beispiel AMBER, ein Molekulardynamik-Simulationsprogramm, das weltweit von mehr als 60.000 Forschern an Universitäten und in Pharmaunternehmen zur Beschleunigung der Medikamentenentwicklung eingesetzt wird.“* [Nvidia,2011] und bezieht sich dabei auf eins von vielen Forschungsgebieten. Weitere CUDA Fallbeispiele werden in Kapitel 6 erläutert. Prinzipiell ist CUDA nur in C verfügbar, mit Wrapper kann man aber auch Programmiersprachen wie Python, Java oder eine .NET Sprache benutzen**.** Seit der neusten Architektur ist auch eine Benutzung von C++ vorgesehen [Nvidia2011].

5.2 Funktionsweise von CUDA

Abstrakt gesehen funktioniert die Berechnung auf der Grafikkarte mit Hilfe von CUDA in 4 Schritten. Im Folgenden wird eine Grafik dargestellt, welche dies sehr gut verdeutlicht.

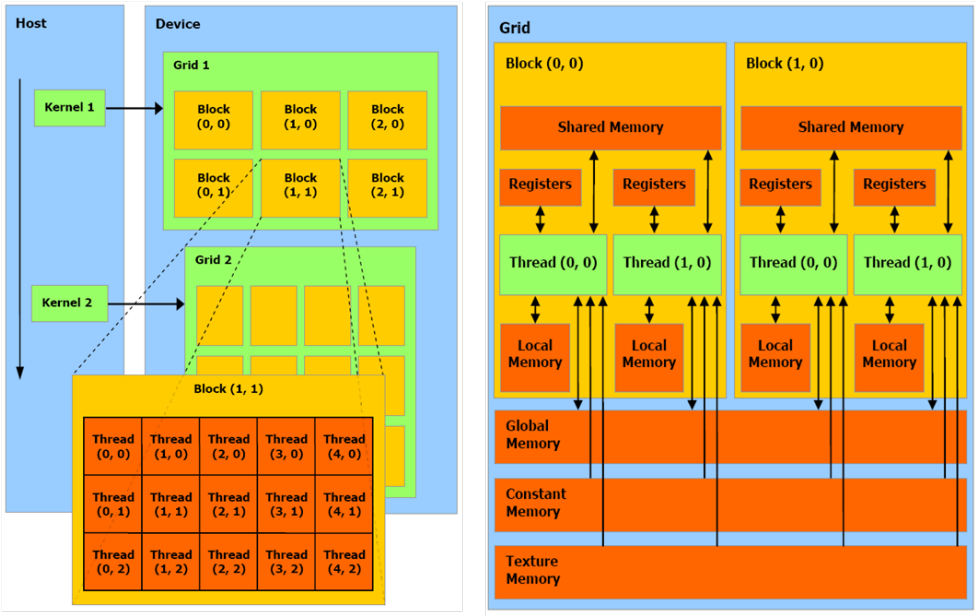


Quelle: Wikipedia Cuda englisch

Im folgenden werden die Schritte erläutert. Zuerst kopiert die CPU die zu berechnenden Daten von dem Hauptspeicher zu dem Grafikkartenspeicher, was nötig ist damit die Grafikkarte nicht permanente Speicherzugriffe auf den Hauptspeicher ausübt und somit der Hauptspeicher zu dem Flaschenhals des Gesamtsystems wird. Weiterhin ist zu Punkt 1 zu erwähnen, dass durch das kopieren von den zu berechnenden Daten in den Grafikkartenspeicher, die Zugriffsgeschwindigkeit des Grafikspeichers erst ausgenutzt wird und die CUDA-Hardware-Architektur Speicherzugriffe optimieren bzw. ermöglichen kann. Im 2. Schritt erteilt die CPU den Befehl den zu berechnenden Algorithmus auf der Grafikkarte auszuführen, womit die Grafikkarte die komplette Kontrolle über den auszuführenden Code bekommt. In Schritt 3 organisiert die Grafikkarte die parallele Abarbeitung. Dies geschieht in dem die Daten sinnvoll auf alle SMs verteilt werden, welche dann die interne Kontrolle über ihre SPs haben. Um dies genau zu erklären bedarf es eines Vorgriffs auf das nächste Kapitel. Kurz zusammengefasst teilt man zusammenhängende Daten wie es z.B. bei Arrays der Fall ist, auf verschiedene SMs auf, um somit möglichst viel der parallelen Architektur auszunutzen. In Schritt 4 werden die fertig abgearbeiteten Daten wieder der CPU übergeben, welche dann mit der sequentiellen Abarbeitung fortfährt. [KiHw10,S.48ff.]

5.3 Interne parallele Abarbeitung

Um die massive Parallelität der Einheitsprozessoren ausnutzen zu können Bedarf es einer Umstrukturierung des Codes und einem Umdenken bei der Organisierung des Codes. CUDA gibt drei Möglichkeiten Daten zu organisieren um sie möglichst effektiv ausführen zu lassen. Alle drei Möglichkeiten sind hierarchisch angeordnet. Anschließend ist ein Bild für die Organisation dieser Hierarchie dargestellt.



Quelle: In Anlehnung an Programming Massively Parallel Processors

Auf der obersten Ebene befinden sich die Grids. Grids belegen sozusagen die Zuteilung auf welchem SM die Berechnung ausgeführt wird. Eine Ebene darunter befinden sich die Blöcke welche sich auf ein Cluster von SPs beziehen. Und auf der untersten Ebene befinden sich die Threads welche im besten Fall von einem SP ausgeführt werden. Threads führen alle den selben Teil des Codes aus. Sie haben alle einen eigenen PC (Programm Counter), eigene Variablen und Prozessor Zustand. Für ein besseres Verständnis wird im nächsten Kapitel ein Beispiel Code gezeigt [KiHw10,S.49ff.]. Als kurzer Vermerk ist hier noch die Hausarbeit von Herrn Heimann zu nennen, welcher sich mit diesem Thema explizit beschäftigt.

5.4 CUDA Code

Nun soll verdeutlicht werden, wie CUDA angewendet wird. Gezeigt wird eine simple Vektor Addition. Zunächst ist ein Beispiel dargestellt, welches in C ohne CUDA programmiert wurde. Die Kommentare helfen zum Verständnis.

```
 1 #define N 10 // Größe des Vektors
 2
 3 void add(int *a, int *b, int *c) {
 4     int tid = 0; // Element des Vektors, initialisiert bei 0;
 5     while (tid < N) {
 6         c[tid] = a[tid] + b[tid];
 7         tid += 1;
 8     }
 9 }
10
11 int main(void) {
12     int a[N], b[N], c[N]; //Initialisierung der Vektoren.
13
14     /*Die beiden Vektoren, welche zusammen addiert werden sollen,
15      * werden mit Werte inititalisiert.
16      */
17     for (int i = 0; i < N; i++) {
18         a[i] = -i;
19         b[i] = i * i;
20     }
21     /*  Funktionsaufruf der Methode add.
22      * Alle Vektoren werden als Refernz übergeben.
23      */
24     add(a, b, c);
25     // Ausgabe
26     for (int i = 0; i < N; i++) {
27         printf("%d + %d = %d \n", a[i], b[i], c[i]);
28
29    }
30 }
31
```

Wie man sieht ist dies eine simple Vektor-Addition, welche vollkommen von der CPU ausgeführt wird. Das nächste Beispiel zeigt wie dieser Code in CUDA-C aussieht.

```
 1 #define N 10
 2
 3 __global__ void add( int *a, int *b, int *c){
 4     int tid = blockIdx.x;
 5     if (tid < N){
 6         c[tid]= a[tid] + b[tid];
 7
 8     }
 9 }
10
11
12
13 int main(void) {
14
15     int a[N], b[N], c[N];
```

```
16      int *dev_a, *dev_b, *dev_c;
17
18
19      HANDLE_ERROR( cudaMalloc (void **)&dev_a, N*sizeof(int));
20      HANDLE_ERROR( cudaMalloc (void **)&dev_b, N*sizeof(int));
21      HANDLE_ERROR( cudaMalloc (void **)&dev_c, N*sizeof(int));
22
23      for (int i=0; i < N; i++ ){
24          a[i] = -i;
25          b[i] = i * i;
26      }
27
28      HANDLE_ERROR(cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice));
29      HANDLE_ERROR(cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice));
30
31      add<<N,1>>(dev_a, dev_b, dev_c);
32
33      HANDLE_ERROR(cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost));
34
35
36      for (int i =0; i<N; i++){
37          printf("%d + %d = %d \n",a[i],b[i],c[i]);
38      }
39
40      cudaFree(dev_a);
41      cudaFree(dev_b);
42      cudaFree(dev_c);
43  }
```

Quelle: Der Code bezieht sich aus einem Code Beispiel von CUDA by Example

Wie man auf den ersten Blick erkennt ist die Funktion add durch ein voranstehendes __global__ ergänzt worden. Dies bedeutet für den CUDA-Compiler, dass dieser Teil vom Code auf der Grafikkarte ausgeführt wird. Man nennt dies auch eine Kernel-Funktion. Desweiteren fällt auf, dass die Schleife wegfällt und der ehemalige Zähler durch ein blockId.x ersetzt wurde. Dies ist das ausschlaggebende Merkmal bei CUDA. Nahezu alle Schleifen werden im besten Fall durch If-Anweisungen ersetzt. Dabei wird jeder ehemalige Iterationsschritt durch eine parallele Abarbeitung eines Blocks ersetzt. Weiterführend wird ein kurzer Einblick gegeben, wie CUDA dies zur Laufzeit organisiert. Die Main-Funktion bleibt größtenteils bestehen. Es ändert sich nur eine weitere Deklaration von Zeigern auf Integer, welche später auf die Speicherstelle im Grafikkartenspeicher verweist. Ebenfalls neu ist die Funktion cudaMalloc, die Speicher auf der Grafikkarte reserviert. Hierbei wird die zu referenzierende Stelle in Form der oben deklarierten Integer-Zeiger repräsentiert, welche zu einem für die CUDA-Hardware nötige Void Pointer auf Pointer gecasted wird. Ebenfalls benötigt cudaMalloc die Größe des Vektors, um genügend Speicher zu reservieren. Nachdem die Vektoren initialisiert wurden, werden die Vektoren in den reservierten Speicher auf der Grafikkarte kopiert. Dies geschieht mit der Funktion cudaMemcpy, welche die Stelle im Grafikkartenspeicher benötigt, den Vektor selber, die Größe und als letzten Parameter die Richtung in die kopiert werden soll. In diesem Fall soll von Hauptspeicher zu dem Grafikkartenspeicher kopiert werden. Dies geschieht mit der Konstanten cudaMemcpyHostToDevice. Der Funktionsaufruf wurde ebenfalls abgeändert. Die Angaben in den Spitzenklammern beziehen sich auf die Größe der Blöcke welche in Kapitel 5.3 erklärt wurden. In diesem Beispiel bedeutet das konkret, dass 10 Blöcke den selben Code ausführen. Nachdem die Funktion abgearbeitet wurde, wird das Ergebnis aus dem Grafikkartenspeicher in den Hauptspeicher kopiert. Dies geschieht mit dem bereits erklärten cudaMemcpy, welches

sich aber von dem ersten Aufruf im 4. Parameter unterscheidet. Die Konstante um vom Grafikkartenspeicher zu dem Hauptspeicher zu kopieren ist hier cudaMemcpyDeviceToHost. Nachdem die Ausgabe getätigt wurde, wird der reservierte Grafikkartenspeicher wieder freigeben. Hierfür ist die Funktion cudaFree vorgesehen. Der folgende Code ist eine Verdeutlichung, was zur Laufzeit geschieht. Dafür wird die Variable blockIdx.x mit ihrem jeweiligen Wert ersetzt.

Block 1	Block 4

3 __global__ void add(int *a, int *b, int *c){
4 int tid = 0;
5 if (tid < N){
6 c[tid]= a[tid] + b[tid];
7
8 }
9 }
	3 __global__ void add(int *a, int *b, int *c){
	4 int tid = 3;
	5 if (tid < N){
	6 c[tid]= a[tid] + b[tid];
	7
	8 }
	9 }

An diesem Beispiel wird klar, dass jeder Iterationsschritt aus dem Standart-C Beispiel durch einen Block ersetzt wird. Also arbeitet jeder Block einen Teil dieser Vektor-Addition ab. Ferner erkennt man, das BlockIdx.x einen Block eindeutig identifiziert. Für eine genaue Erläuterung wie Blöcke intern organisiert sind und welche Dimensionen sie annehmen können ist noch einmal die Hausarbeit von Herrn Heimann zu nennen.

5.5 CUDA- Einschränkungen

Trotz der großen Beliebtheit bringt CUDA auch Nachteile mit sich. Mit Cuda ist keine Rekursion möglich da dies durch den begrenzten Speicher nicht möglich ist. Es würde schnell zu einem Stack-Overflow kommen. Außerdem sind in CUDA keine Funktionspointer und kein Textur-Rendering möglich. Ein großes Spezifikationsproblem ist vorhanden, da es zu Abweichungen bei Double-Gleitkomma-Zahlen zum IEEE Standart kommt. Dies tritt aber nur beim Runden vom Kehrwert, einer Division oder beim Wurzel ziehen auf. Mit der neusten Version von CUDA soll dies aber behoben werden. Deshalb kann es bei schlechter Programmierung oder bei einem ungeeigneten Algorithmus zu permanenten Speicherzugriffen kommen, sowohl von Arbeitsspeicher zu Grafikspeicher, als auch von Grafikspeicher zu einzeln SPs. Dies wird nach einiger Zeit zum Flaschenhals und bremst das System oder Berechnung der Grafikkarten drastisch aus. Der angestrebte Speedup ist somit nicht gegeben und die Berechnung würde vielleicht noch langsamer werden als wenn sie rein von der CPU ausgeführt werden würde. Eine weitere Einschränkung ist, das nur maximal 65.535 Threads gleichzeitig ausführbar sind. Aus Portablitätsgründen sei hier noch einmal gesagt, dass CUDA nur auf Nvidia Grafikkarten ausführbar ist.

5.6 Alternativen zu CUDA

Als eine der größten Alternativen gilt OpenCL. Wie bereits in der Historie erwähnt ist OpenCL herstellerunabhängig. Eine der besonderen Merkmale von OpenCL ist, dass es sich nicht auf Grafikkarten beschränkt. Mit OpenCL kann die Berechnung auf den unterschiedlichsten Mikroprozessoren ausgeführt werden. Sobald ein Hardwarehersteller einen Treiber für sein Gerät veröffentlicht, kann dieser in OpenCL eingebunden werden und so unterschiedlichste Berechnungen ausführen. So kann OpenCL die Berechnungen selbstverständlich auf einer Grafikkarte, aber auch von Kernen der CPU oder auch von FPGAs ausführen lassen. Codetechnisch unterscheidet sich OpenCL nur unwesentlich von CUDA, was zum größten Teil darauf zurückzuführen ist, dass Nvidia bei der Entwicklung von OpenCL mitgearbeitet hat. Von der Bezeichnung her heißt ein Cuda Thread in OpenCL Work item und ein Cuda Block in OpenCL Work group. Einen Unterschied hat OpenCL aber, denn das Programm muss wissen auf welcher Hardware es ausgeführt werden soll. Ebenfalls nennenswert ist der JIT-Compiler (Just in Time Compiler), welcher den Device Code zu Laufzeit kompiliert. Dies kostet logischerweise aber Rechenzeit. [KiHw10,S.206f.] Wie bereits erwähnt gibt es noch zwei weitere Konkurrenten für CUDA, DirectCompute von Microsoft, welches Teile von Windows Vista und Windows 7 im Hintergrund berechnet, und FireStream welches in Anlehnung an Cuda von Ati entwickelt wurde. Über beide Alternativen gibt es aber leider relativ wenig Informationen.

6 Fallbeispiele für die Berechnung auf der Grafikkarte

In dem letzten Kapitel wird nochmals anhand von Beispielen die Rechenleistung von GPGPU verdeutlicht. Aktuell sind vier große Anwendungsgebiete vorhanden. Physik, Finanzwesen, Biologie und Medizin. Die Fallbeispiele beziehen sich auf die Gebiete Physik und Medizin, daher sei kurz darauf verwiesen, welche Erfolge im Bereich Finanzwesen und Biologie vorzuzeigen sind. Im Bereich Finanzwesen geht es um Risikoanalysen für große Geldinstitute, welche zeitnah erfasst werden müssen. Hierfür wurden früher Rechenzentren beauftragt. Im Bereich Biologie sind im Bereich der Molekularmikroskopie Erfolge vorzuweisen. In dieser Disziplin unterstützen Grafikkarten die Bilderfassung und die Verwertung der Bilder dieser Mikroskope. [KiHw10,S.173ff.]

6.1 Fallbeispiel 1 Medizin

Eine für Frauen ernstzunehmende und gefährliche Erkrankung ist Brustkrebs. Die Anzahl der Diagnosen stieg in den letzten 20 Jahren drastisch an. Die Erkrankung muss daher rechtzeitig erkannt werden. Zu Beginn der Diagnostik, war vor allem die Mammografie das aufschlussreichste Mittel. Mit der Mammografie wurde die Brust der Patientin zwischen zwei Platten zusammengedrückt und anschließend geröngt. Diese Methode war unter anderem sehr schmerzhaft und erhöhte -ironischerweise- das Brustkrebsrisiko. Aber nicht nur für den Patienten war dies ein unange-

nehmes Unterfangen, sondern auch für die Ärzte. Für die Mammografie mussten Ärzte speziell geschult werden, um das Röntgenbild analysieren zu können. Die Schwierigkeit dabei war auf dem Röntgenbild etwas zu erkennen und die richtige Diagnose zu stellen, um damit den Patienten eine kostspielige, aber vor allem eine strapazierende Chemotherapie zu ersparen. Oft wurden diese Röntgenbilder daher fehlinterpretiert und neue Tests wurden beauftragt. Dieses Problem wollte die Firma TechniScan beheben, indem sie die Brust mit einem Ultraschallgerät abscannte. Da ein normaler Ultraschall aber nicht so stark in die Tiefe gehen konnte und die Bilderkennung auch eher schlecht war, entwickelte TechniScan den 3D Ultraschall, welcher diese Problematik umgeht. Leider war in den Anfängen dieser Technik die Rechenleistung zu gering und das Gerät setzte sich nicht durch. Dies entmutigte die Firma TechniScan aber nicht und sie entwickelt ein neues 3D- Ultraschall, welches die Berechnungen und Verwertung dieser Bilder mit einer Nvidia Tesla Karte und Cuda durchführte. Mit Hilfe dieser Rechenleistung war das System sogar zur Visualisierung von Brustgewebe fähig. Insgesamt werden 35 GB Daten nach 15 Minuten Scan erfasst und eine ausreichende Diagnose kann nach 20 Minuten gestellt werden [vgl. SaKa11,S.8f].

6.2 Fallbeispiel 2 Physik

Industrielle Rotoren und Schneidmesser wurden lange Zeit als „schwarze Kunst“ verkannt, da die Erfassung von Luft- und Flüssigkeitsverhalten nicht mit einem plastischen Model realisierbar war. Größere Computermodelle waren nur mit Supercomputern durchführbar, was dazu führte, dass die Entwicklung auf diesem Gebiet stagnierte. Angetrieben von diesem Problem untersuchte die Universität von Cambridge die Entwicklung von parallelen Berechnungen ohne darauf auf die Rechenleistung von GPGPU einzugehen. Nach mehreren Versuchen ein effektives und kostengünstiges System zu entwickeln stieß Dr. Graham Pullan aus reinem Zufall auf CUDA und entwickelte hiermit das erste Fluidmodell für Desktop-PCs. Nach Erkennen des Potenzials verbaute er mehrere Grafikkarten in einem Rechner zu einem CUDA-Cluster,was ihm eine effektive Modellerzeugung in nur 15 Sekunden erlaubte [vgl. SaKa11,S.9f].

6.3 Weitere Projekte

ATI und Nvidia haben jeweils zwei große Projekte, bei denen Menschen aus der ganzen Welt mithelfen können. Bei dem Projekt Folding@Home, welches von der Stanford University ins Leben gerufen wurde, geht es um die Krankheitserkennung von Alzheimer, BSE, Krebs und Weiteren Krankheiten. Dabei kann jeder, der eine Ati Grafikkarte der Reihe HD2000 oder höher, CUDA-fähige Grafikkarte oder Playstation 3 besitzt, mithelfen [Fold,2011]. Bei dem Projekt Seti@Home welches von der Berkley Universität geleitet wird, werden mit Radiosignalen nach außerirdischem Leben gesucht. Seit 1999 wurde mit Hilfe von fünf Millionen Benutzern 2,3 Millionen Jahre Rechenzeit erspart. Momentan kann man bei diesem Projekt aber nur mit einer CUDA-fähigen Karte mitarbeiten. [Seti,2011]

Literaturverzeichnis

- Kirk D. & Hwu W. (2010). Programming Massively Parallel Processors- A Hands-on Approach. Burlington, Elsevier.
- Sanders J. & Kandrot E. (2010). Cuda by Example- An Introduction to General-Purpose GPU Programmin. Boston, Perason Education Inc.
- White R. (2004). So funktionieren Computer- Ein visueller Streifzug durch den Computer & alles, was dazu gehört. München, Markt + Technik Verlag. Übersetzung: Lochmann C., Drewnitzki K. & Alkemper C.
- Vötter R. (2011). GPUs: Hintergrund in PC Games Hardware Sonderheft 01/2011 S. 6 bis 8
- Jungowsik M. (2011). CPUs: Hintergrund in PC Games Hardware Sonderheft 01/2011 S.21 bis 23
- Spille C. (2011). Multi-GPU, Runde 2001 in PC Games Hardware 05/2011 S. 36 bis 46
- Tesla M2090: 512 Kerne nun auch fürs Supercomputing, 18.05.2011, http://www.heise.de/ix/meldung/Tesla-M2090-512-Kerne-nun-auch-fuers-Supercomputing-1245209.html, (07.06.2011,1:20)
- Davis L. GDDR5 Pineout and Description, 2011, http://www.interfacebus.com/GDDR5-Memory-IC-Pinout.html, (07.06.2011,1:23)
- RAMDAC, 17.05.2011, http://en.wikipedia.org/w/index.php?title=RAMDAC&oldid=429605506, (07.06.2011,1:25)
- Smith T. 3dfx wraps up wrapper Web sites, 08.04.1999, http://www.theregister.co.uk/1999/04/08/3dfx_wraps_up_wrapper_web/,(07.06.2011,1:28)
- Nvidia Cuda Homepage http://www.nvidia.de/object/cuda_home_new_de.html (07.06.2011,1:29)
- Compute Unified Device Architecture , http://de.wikipedia.org/wiki/Compute_Unified_Device_Architecture, (07.06.2011,1:30)
- CUDA, http://en.wikipedia.org/wiki/CUDA, (07.06.2011,1:30)
- Raw Performance: SiSoftware Sandra 2010 (GFLOPS) http://www.tomshardware.de/charts/desktop-cpu-charts-2010/Raw-Performance-SiSoftware-Sandra-2010-Pro-GFLOPS,2409.html, (07.06.2011,1:34)
- Gartz J. GPU ersetzt CPU: Nvidia fordert Intel heraus, 26.11.2008, http://www.crn.de/panorama/artikel-48701.html, (07.06.2011,1:37)
- Wagner D. Terrain Geomorphing in the Vertex Shader, 20.04.2003, http://www.gamedev.net/page/resources/_/reference/programming/game-programming/landscapes-and-terrain/terrain-geomorphing-in-the-vertex-shader-r1936, (07.06.2011,1:38)
- CUDA Präsentation, http://www.plm.eecs.uni-kassel.de/plm/fileadmin/pm/courses/pvII09/pvII090518k.pdf, (07.06.2011,1:39)
- Wagner M & Luttmann E., Folding@Home, http://folding.stanford.edu/German/Main, (07.06.2011,1:43)
- Seti@Home, http://setiathome.berkeley.edu/, (07.06.2011,1:43)