Studienarbeit

Berechnung der Wellengleichung mit Hilfe der maxwellschen Gleichungen auf der Grafikkarte durch Nvidia's CUDA("Compute Unified Device Architecture") und anschließender Performanceanalyse

Matthias Heine
Matrikelnmmer: 21166928
Friedrich-Alexander-Universität Erlangen

23. Februar 2008

Betreut von
Prof. Dr. Christoph Pflaum
Lehrstuhl für Informatik 10 – Systemsimulation

Inhaltsverzeichnis

1 Einleitung

In unserer heutigen Zeit, in der Algorithmen immer weiter optimiert werden und somit langsam an die Grenzen ihrer Leistungfähigkeit stoßen, ist es an der Hardware, die Berechnungen diverser Algorithmen weiter zu beschleunigen. Natürlich werden Prozessoren, RAM's etc. immer weiterenwickelt, aber sollte man nicht wirklich auch alle Resourcen die einem gegeben sind auch nutzen?
Durch die Spieleindustrie verstärkt geförderte Entwicklung der Grafikkarte, macht diese zum rechenstarken Kraftpaket, das bislang relativ ungenutzt im wissenschaftlichen Sinn blieb. Auf Grund ihrer spezialisierten Fähigkeiten hoch-parallele Probleme schnell abzuwickeln(größtenteils grafisches Rendering), wäre es also von Vorteil eine API zu besitzen, die es ermöglicht einfach Zugriff auf das genannte Potential zu erlangen.
Nivida's CUDA("Compute Unified Device Architecture") bietet seit ca. einem Jahr diese Möglichkeit für einige der eigenen Grafikkarten an. Meine Studienarbeit wird sich im folgenden Text, damit beschäftigen was CUDA ist und welche zusätzliche Eigenschaften CUDA anbietet, aber auch welche Einschränkungen man eingehen muss. Dies geschieht am Beispiel der Berechnung der Wellengleichung, indem ich Methoden anbiete, die es ermöglichen den gleichen Algorithmus auf der CPU oder auf der GPU zu berechnen. Von diesen Berechnungen werden die Berechnungsdauern gemessen und analysiert.

2 Wellengleichung

Eine Wellengleichung ist eine partielle Differenzialgleichung, deren Lösung eine sich ausbreitende Wellen darstellt. Diese Gleichung findet viele Einsatzmöglichkeiten. Zum Beispiel in

- Akustik
- Elektromagnetismus
- Laser-Technik
- ...

2.1 Herleitung der Wellengleichung

Die allgemeine Wellengleichung lautet:

$$\left(\Delta - \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\right) u = 0$$

Die Ausbreitungsgeschwindigkeit ist c.
Δ ist der Laplaceoperator.

Oder mit $\mu\varepsilon = \frac{1}{c^2}$:

$$\mu\varepsilon\frac{\partial^2\mu}{\partial t^2} = \Delta\mu$$

Für die Herleitung der Wellengleichung benötigen wir zuerst einmal die Maxwellschen Gleichungen:

$$\nabla \circ \vec{E} = 0 \tag{1}$$

$$\nabla \circ \vec{B} = 0 \tag{2}$$

$$\nabla \times \vec{E} = -\frac{\partial}{\partial t}\vec{B} \tag{3}$$

$$\nabla \times \vec{E} = \mu_0 \varepsilon_0 \frac{\partial}{\partial t}\vec{E} \tag{4}$$

Unter Anwendung der Rotation auf (3) erhält man:

$$\nabla \times \left(\nabla \times \vec{E}\right) = \nabla \times \left(-\frac{\partial}{\partial t}\vec{B}\right)$$

$$\Rightarrow \nabla \circ \left(\nabla \circ \vec{E}\right) - \nabla^2 \vec{E} = -\frac{\partial}{\partial t}\left(\nabla \times \vec{B}\right) = -\frac{\partial}{\partial t}\left(\mu_0 \varepsilon_0 \frac{\partial}{\partial t}\vec{E}\right)$$

$$\Rightarrow \nabla^2 \vec{E} = \mu_0 \varepsilon_0 \frac{\partial^2}{\partial t^2}\vec{E} \quad mit \quad c^2 = \frac{1}{\mu_0 \varepsilon_0}$$

$$\Rightarrow \nabla^2 \vec{E} = \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\vec{E}$$

$$\Rightarrow \left(\nabla^2 - \frac{1}{c^2}\frac{\partial^2}{\partial t^2}\right)\vec{E} = 0$$

Analog könnte man so auch die Wellengleichung für $\vec{B}$ bestimmen.

2.2 Iterationsvorschrift zur Berechnung der Wellengleichung

Zur Berechnung eines Zeitschritts der Wellengleichung wird im Programmcode folgende Iterationsvorschrift benutzt:

$$\begin{aligned} matrix_{new\,ij} = {} & x_{factor}\frac{-2 * matrix_{ij} + matrix_{i-1j} * matrix_{i+1j}}{2} \\ & + y_{factor}\frac{-2 * matrix_{ij} + matrix_{ij-1} * matrix_{ij+1}}{2} \\ & + 2 * matrix_{ij} + matrix_{old\,ij} \end{aligned}$$

mit:

$$x_{factor} = \frac{c^2}{h_x^2}\tau^2$$

$$y_{factor} = \frac{c^2}{h_y^2}\tau^2$$

wobei:

$$h_x = \frac{length_x \lambda}{size_x}$$

$$h_y = \frac{2\lambda}{size_y}$$

und:

$$\tau = \begin{cases} \frac{0.5h_x}{c} & \text{für } h_x < h_y \\ \frac{0.5h_y}{c} & \text{sonst} \end{cases}$$

Die Ausbreitungsgeschwindigkeit ist c.
Die Wellenlänge wird von λ beschrieben.

3 CUDA("Compute Unified Device Architecture")

Die Compute Unified Device Architecture (CUDA) von Nvidia versucht mittels der eigen entworfenen API die Grafikkarte als Co-Prozessor bereitzustellen. Folgendes stellt CUDA zur Verfügung:

- nvcc C compiler
- CUDA FFT und BLAS Bibliotheken für die GPU
- Profiler
- gdb debugger für die GPU (alpha erscheint im März, 2008)
- CUDA Laufzeittreiber
- CUDA Programmierhandbuch

3.1 Allgemeine Struktur

Das Softwarepacket von CUDA ist in drei Teile aufgeteilt. Die Basis bildet die Treiber-Ebene, darauf aufbauend existiert die Laufzeitbibliothek und an der Spitze die Anwendungsbibliothek. In der Anwendungsbibliothek befinden sich unter anderem zwei Mathematikbibliotheken(CUFFT und CUBLAS). Hierzu werden von Nvidia selber ausreichend Informationen und Dokumentationen auf der offiziellen Homepage angeboten und mit dem CUDA Toolkit mitgeliefert.

3.1.1 Thread-Block

Ein sogenannter "Thread-Block"beschreibt nichts weiteres als ein Sammelsorium an Threads, die zu einem Block zusammengefasst werden. Es sind maximal 512 Threads pro Block zulässig.
Nicht jeder Thread kann mit den anderen Threads interagieren. Die einzige Möglichkeit Daten zwischen verschiedenen Threads innerhalb des zugehörigen Blocks auszutauschen

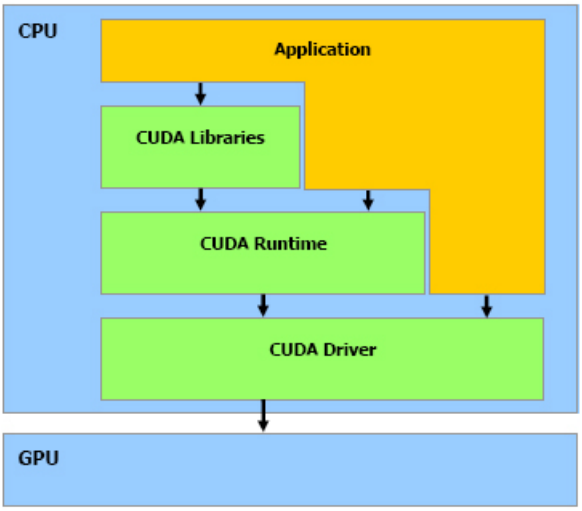

Abbildung 1: CUDA's Software Architektur

ist der Shared Memory. Daten zwischen Threads verschiedenen Blöcke auszutauschen ist nicht möglich. Zugriffe verschiedener Threads auf diesen shared memory müssen sycronisiert werden. Hierfür kann man sogenannte Sycronisationspunkte setzen, an denen alle Threads warten bis alle laufenden Threads an diesem Punkt angekommen sind(sh. Abschnitt 3.3.5).
Um die Adressierung der jeweiligen Threads zu erleichtern ist es möglich einen Block als ein-, zwei- oder dreidimensionale Array anzulegen. Die zugehörigen ThreadID's sehen zum Beispiel für ein dreidimensionales Array so aus (x, y, z).

3.1.2 Grid von Thread-Blöcken

Mehrere Thread-Blöcke bilden wiederum ein Grid. Dieses Gridgröße beläuft sich auf maximal 65535 Blöcke. Im Gegensatz zu den Threads werden die Blöcke unter Umständen sequenziell durchlaufen und abgearbeitet, was auch die Erklärung dafür ist, dass es nicht möglich ist Daten zwischen verschiedenen Threads verschiedener Blöcke auszutauschen. Dies geschieht, falls die verwendete Grafikkarte nicht genügend Ressourcen besitzt, um das zu lösende Problem parallel zu lösen. Es ist auch möglich, das ein Teild er Blöcke parallel läuft und eine anderer squenziell. Desweiteren kann man auch nicht verschiedene Threads aus verschiedenen Blocks syncronisieren.
Die Adressierung folgt dem selben Schema wie bei den Threads. Man hat also die Wahl, ob man das Grid ein-, zwei- oder dreidimensionales Array anlegt. BlockID's haben werden wie ThreadID's dargestellt (x, y, z) (dreidimensionales Array).

3.1.3 Speicherarten

Die Grafikkarte besitzt sechs verschiedene Speicherarten:

- register(lesen/schreiben pro Thread)

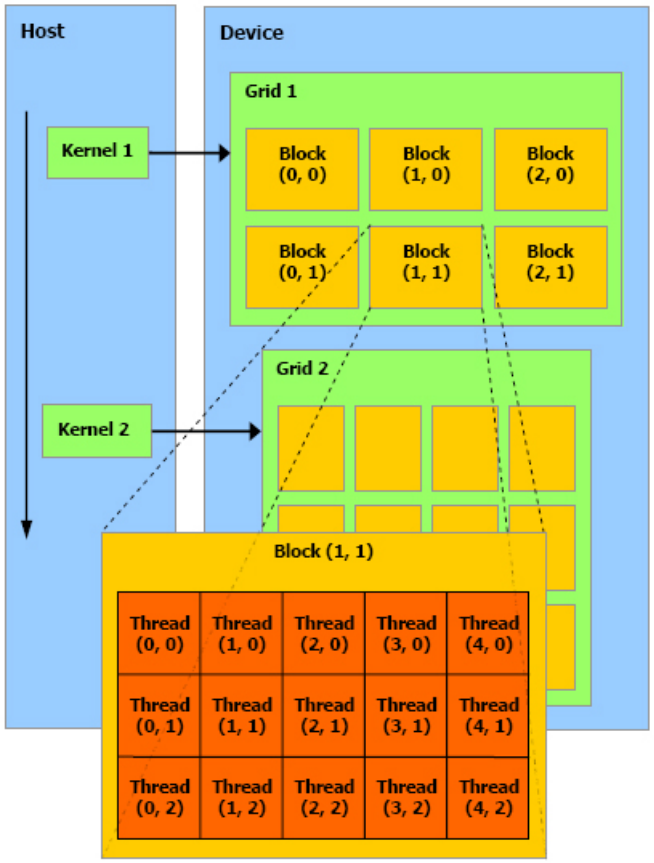

Abbildung 2: Thread- und Blockstruktur

- local memory (lesen/schreiben pro Thread)
- shared memory (lesen/schreiben pro Block)
- global memory (lesen/schreiben pro Grid)
- constant memory (nur lesen pro Grid)
- texture memory (nur lesen pro Grid)

Da CUDA noch nicht ausgereift ist, sei hier schon einmal angemerkt, dass man für Performance mit den verschiedenen Speicherarten herumspielen muss. Genrell kann man aber sagen je spezieller die Art des Speichers ist und man diese in seiner Berechnung vernünftig unterbringen kann, desto mehr Performance erhält man am Ende. Beispielsweise Konstanten sollte man auch in den constant memory laden, shared memory zur Verständigung der Threads benutzen und global memory möglichst vermeiden. Sollen natürlich nur Richtlinien sein und kein Garant für Performance.

3.2 Hardware

Die Verwendung von CUDA ist nur auf den Grafikkarten der GeForce 8x00 Serie, Quadro FX 5600/4600, und Tesla möglich, anderfalls wird ein Emulation-Modus verwendet. Hierbei gehen jedoch sämtlich Performancevorteile verloren. Jede dieser Grafikkarten besitzt mehrere Multiprozessoren, welche eine Single Instruction Multiple Data(SIMD) Architektur besitzen. Alle Multiprozessoren besitzen vier der oben genannten Speicherarten(32-bit register, shared memory, constant memory, texture memory).

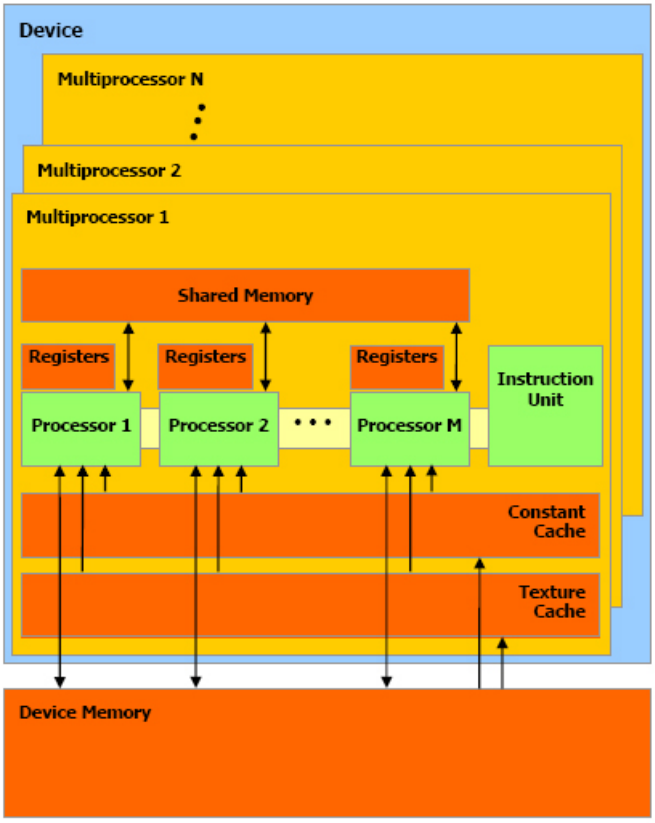

Abbildung 3: Single Instruction, Multiple Data architecture (SIMD

Die Anzahl der Multiprozessoren hängt vom Typ der Grafikkarte ab. Performancetechnisch gilt je mehr Multiprozessoren, desto schneller. Jeder dieser Multiprozessoren berechnet zum selben Zeitzyklus das Gleiche, aber auf verschiedenen Daten.

3.2.1 Art und Weise der Abarbeitung

Jeder Block wird in Gruppen von Thread SIMD's zerlegt. Diese nennt man wraps. Jeder wrap enhält die gleiche Anzahl von Threads und wird mit wrap size bezeichnet. Ein sogenannter Threadscheduler wechselt von einem wrap zum anderen um die Auslastung

der Multiprozessoren zu maximieren. Ein Wrap enthält immer Threads mit aufeinanderfolgenden ThreadID's beginnend bei der ThreadID 0.
Falls die Anzahl der benutzten Register pro Thread multiplizert mit der Anzahl der Threads im Block größer ist als die Anzahl der Register pro Multiprozessor, kann das Programm nicht ausgführt werden. Außerdem wird ein Block auch immer nur von einem Multiproessor abgearbeitet, was dazu führt dass man Speicher benutzen sollte, der auf einem Multiprozessor ist. Blöcke können nicht syncronisiert werden, man kann also sicheren Datenaustausch über den globalen Speicher nicht gewährleisten.

3.2.2 Mehrere Grafikkarten

Will man mehrere Grafikkarten zusammen nutzen ist es erforderlich, dass die Grafikkarten vom gleichen Typ sind. Will man jedoch mehrere Grafikkarten als verschiedene Co-Prozessoren nutzen, muss man sicher stellen dass der SLI Modus der Grafikkarten abgestellt ist.

3.2.3 Spezifikationen

Nachfolgende Spezifikationen sind zu beachten und einzuhalten:

- Maximal 512 Threads pro Block
- Maximal 65535 Blocks pro Grid
- Wrap size ist 32
- 8192 Register pro Multiprozessor
- 16KB shared memory pro Multiprozessor
- 64KB constant memory, 8KB maximal pro Multiporzessor
- Maximal 2 Millionen Instruktionen pro Kernel
- Floating-Point Standard

3.3 Software

CUDA bildet eine Erweiterung der Programmiersprache C. Dennoch muss man beachten, dass nicht alle C-Funktionen unterstützt werden. Die Laufzeitbibliothek wird unterteilt in einen Host-Teil , einen Device-Teil und einen Common Teil. Im Host Abschnitt werden Funktionen beschrieben, die vom Host(CPU) aufgerufen werden können. Analog dazu gibt es den Device Abschnitt. Dieser bietet Funktionen die auf dem Device (GPU) ausgeführt werden. Der Common Abschnitt stellt die Schnittstelle zwischen den anderen beiden Abschnitten da und enthält deswegen Funktionen, die sowohl auf CPU als auch auf GPU ausgeführt werden können.

3.3.1 Funktionen

Von CUDA werden drei Funktionserweiterungen angeboten(__host__ , __device__ , __global__).

- __host__ : die Funktion ist nur vom Host aufrufbar und wird auch nur dort ausgeführt
- __device__ : die Funktion ist nur vom Device aufrufbar und wird auch nur dort ausgeführt
- __global__ : die Funktion ist nur vom Host aufrufbar und wird auf dem Device ausgeführt (zB. Kernel-Aufruf)

Man kann die Funktionserweiterungen kombinieren, leider sind nicht alle Kombinationen zulässig und bei manchen muss man mit Einschränkungen rechnen. Hier sei noch erwähnt, dass __device__ Funktionen immer inlined sein müssen und __global__ Funktionen immer wie in 3.3.6. beschrieben wird ausgeführt werden müssen. __global__ Funktionen werden Kernel gennant und haben einen speziellen Aufrufmechanismus.

3.3.2 Variablen

Variablen in CUDA können maximal 32bit-Float-Genauigkeit besitzen. Den Typ double gibt es in CUDA nicht. Sonst können alle Typen genutzt werden, die es in C auch gibt. Bei den VAriablen gibt es drei Erweiterungen:

- __constant__ : kann mit __device__ kombiniert werden , Variable wird im constant memory abgelegt, existiert so lange wie das Programm , Zugriffe können von allen Threads getätigt werden , sowie vom Host durch die Laufzeitbibliothek
- __device__ : Varibale wird im global memory gepseichert, existiert so lange wie das Programm, Zugriffe können von allen Threads getätigt werden , sowie vom Host durch die Laufzeitbibliothek
- __shared__ : kann mit __device__ kombiniert werden , Vaiable wird im shared memory eines Blocks abgelegt, existiert so lange wie das Programm, Zugriffe sind nur von Threads des zugehörigen Blocks möglich

3.3.3 Built-in Variablen

Built-in Variablen stehen vier verschiedene zur Verfügung. Eine davon ist gridDim und bezeichnet die Größe des Blockarrays. Mit blockDim legt man die Größe des Threadarrays fest. Beide sind vom Typ dim3. blockDim braucht als Parameter Werte die über #define festgelegt wurden. Man kann also die Blockgröße zur Laufzeit nicht mehr beeinflussen. dim3 beschreibt einen dreidimensionelen Vektor von unsigned Integern.

Die letzten beiden Variablen sind Indexvariablen vom Typ uint3. Den Blockindex beinhaltet blockIdx und den ThreadIndex beinhaltet threadIdx(threadIdx.x enhält also beispielsweise den x-Wert des Threadindex).
Zusätzlich zu den built-in Variablen existieren auch noch built-in Vektoren, die mit typex bezeichnet werden, wober type der Typ der Variablen ist(es gibt kein double) und x die Anzahl der Komponenten(möglich sind 1, 2, 3 oder 4). Um auf die verschieden Komponenten zuzugreifen benutzt man die Felder x, y, z, oder w.

3.3.4 Mathematische Funktionen

Beim Rechnen mit CUDA muss man beachten, dass es zum ersten nur Float-Genauigkeit gibt und zum zweiten verschiedene mathematischen Funktionen angeboten werden, die etwas ungenauer sind als die von beispielsweise <math.h> bereitgestellten, aber dafür bringen sie mehr Performancen. Solche Funktionen erhalten das Prefix __ und sind in der Dokumentation von CUDA aufgelistet. Um genrell diese Funktionen zu benutzen kann man seinen Code mit Option -use_fast_math kompilieren.
Falls man die auf performance ausgerichtet mathematischen Funktionen jedoch nicht nutzen möchte, bietet CUDA auch eine normale Mathematikbibliothek an, in der die Funktionen aber eben nur mit Float-Genauigkeit ausgestattet sind. Ungenauigkeiten und Fehlerraten der normalen mathematischen Funktionen werden ebenfalls in der Dokumentation beschrieben.

3.3.5 Syncronisation

Um in CUDA Threads zu syncronisieren muss man sich der Funktion __syncthreads() bedienen. Diese Funktion lässt alle Threads pausieren bis alle an diesem Punkt angekommen sind. Wichtig ist dies deshalb, da es beim Zugriff auf den shared memory sonst zu Hazards kommen könnte. Man stelle sich einfach eine if-Abfrage vor bei der die eine Möglichkeit 100000 Intruktionen besitzt und die andere 10. So könnte es leicht zu unzulässigen Zuständen im Speicher kommen.

3.3.6 Ausführung

Wie oben schon erwähnt besitzt eine als __global__ deklarierte Funktion einen speziellen Aufruf. Für diesen Aufruf benötigt man die built-in Variablen blockDim und gridDim. Folgenden Funktion

__global__ void foo(void);

ist so aufzurufen

foo< < < gridDim, blockDim, Ns > > >(void);

Ns: vom Typ size_t und beschreibt die Anzahl an Bytes die für den shred memory gebraucht werden

3.3.7 Integration in C++

Da sicherlich viele die Vorteile von C++ nicht verlieren wollen, kann man seine CUDA Funktionen auch in C++ einbetten. Dies kann durch die Funktionserweiterung extern "Cërreichen. Eine Möglichkeit wäre also:

C++ Code(.cpp):

```
extern "C"void fooCUDA(void);
```

Im C++ Code gibt es also nur den Methodenrumpf.

CUDA Code(.cu):

```
extern "C"void fooCUDA(void){
...
}
```

Im CUDA Code kann dann zum Beispiel der Kernel aufgrufen werden oder Speicher auf der Grafikkarte angelegt werden.

3.3.8 Der Compiler NVCC

NVCC stellt den hauseigenen Compiler zu CUDA dar. Mit ihm kann man auch C und C++ Dateien kompilieren, kann sich aber auch dafür entscheiden nur den CUDA-Code vom NVCC kompiliren zu lassen und die restlichen Dateien mit dem bevorzugten Compiler. Was man mit NVCC kompilieren muss sind Dateien, die die Funktions- und/oder Variablenerweiterungen beinhalten. Diesen Dateien gibt man die Endung .cu. Die Benutzung wurde den gewöhnlichen Compilern angepasst, womit er leicht zu bedienen ist. Falls man für seinen eigenen nicht-CUDA-Code einen anderen Compiler benutzen will, so erhält man durch den NVCC für seine .cu-Datei eine gewöhnliche Objektdatei, die mit jedem Linker wieder problemlos nutzbar ist.
NVCC hat aber Einschränkungen in Bezug auf das Kompilieren von C/C++ Dateien, welche gesondert und detailiert von Nvidia in einer extra Dokumentation zum Compiler beschrieben werden. Der NVCC untersützt unter Linnux jegliche Shells, unter Windows werden die DOS shell, CygWin shells und MinGW shells unterstützt.

4 Performanceanalyse

In diesem Kapitel werden ich nun konkret auf meine Studienarbeit und deren Programmcode eingehen. Zu Beginn möchte ich erwähnen, dass beide Algorihtmen, der für die CPU und der für die GPU, so implementiert sind, dass sich sich in der Art und Weise wie sie das Problem lösen kaum unterscheiden. Sicherlich ist in beiden Algorithmen von der Performance her noch einiges zu optimieren, aber dennoch war im Sinne meiner Studienarbeit nicht die Performance der einzelnen Alogirthmen gefragt, sondern der Vergleich zwischen der Performance auf der CPU und der GPU. Logisch ist es, dass man hierfür

sich in der Bearbeitung des Problems ähnelnde Algorithmen verwenden sollte.
Die Zeiten die angeben werden, beschreiben die Zeiten, die nötig sind vom Aufruf der calculate-Funktion bis zu ihrer Wiederkehr. Dadurch schließe ich also auch die Kopiervorgänge von Host-to-Device und Device-to-Host mit ein. Der Grund hierfür ist, dass es sich nicht vermeiden lässt solche Kopiervorgänge zu benutzen wenn man die GPU als Co-Prozessor benutzen möchte.

4.1 Systemspezifikation des Testrechners

Als Testrechner habe ich einen Grafikrechner vom Lehrstuhl 10 Systemsimulation der Friedrich-Alexander-Universität benutzt. Auf dem Rechner ist eine Version von Linux installiert und er besitzt folgende Spezifikationen:

2x Opteron 2222 3.0 GHz
4x 4GB DDR2-667 rg ECC
2x Geforce 8800 Ultra 768MB
2x HDD SATA II 500GB 7.2k
1x AT03 Systemboard D2533

4.2 Tabelle

NX:	NY:	LENGTH-X:	ITERATIONS:	BLOCK_SIZE:	TIME(GPU/CPU):
100	100	2	1	22	40ms/0ms
100	100	2	10	22	10ms/10ms
100	100	2	100	22	0ms/60ms
100	100	2	1000	22	50ms/520ms
1000	100	2	1	22	0ms/10ms
1000	100	2	10	22	0ms/70ms
1000	100	2	100	22	30ms/650ms
1000	100	2	1000	22	340ms/6480ms
10000	100	2	1	22	20ms/80ms
10000	100	2	10	22	40ms/720ms
10000	100	2	100	22	350ms/7090ms
10000	100	2	1000	22	3420ms/70770ms
100000	100	2	1	22	100ms/1380ms
100000	100	2	10	22	370ms/10580ms
100000	100	2	100	22	3190ms/103180ms
100000	100	2	1000	22	31390ms/1.02702e+06ms
100	100	2	1	16	50ms/0ms
100	100	2	10	16	0ms/10ms
100	100	2	100	16	10ms/50ms
100	100	2	1000	16	50ms/520ms

1000	100	2	1	16	0ms/10ms
1000	100	2	10	16	10ms/60ms
1000	100	2	100	16	40ms/650ms
1000	100	2	1000	16	320ms/6620ms
10000	100	2	1	16	10ms/80ms
10000	100	2	10	16	40ms/720ms
10000	100	2	100	16	330ms/7080ms
10000	100	2	1000	16	3220ms/71070ms
100000	100	2	1	16	100ms/1400ms
100000	100	2	10	16	350ms/10610ms
100000	100	2	100	16	3050ms/103250ms
100000	100	2	1000	16	30070ms/1.03031e+06ms
100	100	2	1	8	50ms/10ms
100	100	2	10	8	0ms/0ms
100	100	2	100	8	10ms/50ms
100	100	2	1000	8	50ms/530ms
1000	100	2	1	8	0ms/10ms
1000	100	2	10	8	0ms/60ms
1000	100	2	100	8	40ms/640ms
1000	100	2	1000	8	320ms/6440ms
10000	100	2	1	8	10ms/90ms
10000	100	2	10	8	40ms/730ms
10000	100	2	100	8	340ms/7120ms
10000	100	2	1000	8	3410ms/71050ms
100000	100	2	1	8	110ms/1410ms
100000	100	2	10	8	350ms/10680ms
100000	100	2	100	8	3110ms/103740ms
100000	100	2	1000	8	30600ms/1.03425e+06ms
100	100	2	1	4	50ms/0ms
100	100	2	10	4	0ms/10ms
100	100	2	100	4	10ms/50ms
100	100	2	1000	4	50ms/520ms
1000	100	2	1	4	0ms/10ms
1000	100	2	10	4	10ms/60ms
1000	100	2	100	4	40ms/650ms
1000	100	2	1000	4	330ms/6560ms
10000	100	2	1	4	20ms/90ms
10000	100	2	10	4	40ms/720ms
10000	100	2	100	4	350ms/7130ms
10000	100	2	1000	4	3470ms/70860ms
100000	100	2	1	4	120ms/1400ms
100000	100	2	10	4	370ms/10660ms
100000	100	2	100	4	3200ms/103770ms

100000	100	2	1000	4	31700ms/1.0329e+06ms

4.3 Analyse der Tabellen

Die Zeiten wurden mit Hilfe der Funktion clock() ermittelt. Vor und nach dem Funktionsaufruf von calculateCPU() oder calculateGPU() wird clock() aufgerufen, anschließend werden die beiden, pro calculate-Funktion genommen, Zeiten subtrahiert und bilden die Funktionsdauer. Diese Funktionsdauer beinhaltet somit die komplette Berechnung mit allen Abfragen und Kopiervorgängen. Umschlossen wird die Zeitnahme der beiden Funktionen von Schleifen, die die Parameter NX und ITERATIONS erhöhen. NY wird nicht erhöht, aufgrund der Tatsache, dass die Welle sich in meinem Code in x-Richtung ausbreitet.
Die Blockgröße muss per Hand im #define in der maxwellCUDA.cu verändert werden, weil sie nur durch ein #define festgelegt werden kann und somit zur Laufzeit nicht mehr änderbar ist. Der Wert 22 bildet die Obergrenze der möglichen internen Blockgröße, da die maximale zulässige Anzahl von Threads 512 beträgt(da $23 \times 23 = 529$). Falls man die Blockgröße verändern will, befolgt man entweder die Instruktionen aus dem Abschnitt 4.5 oder man wählt die Blockgröße so, dass sie eien Potenz von 2 bildet. Es macht allerdings wenig Sinn die Blockgröße klein zu wählen(kleiner als 8) und wird hier nur gemacht um den Performanceunterschied darzustellen.
Der erste Punkt, der einem auffällt bei den Zeiten ist der Fall, der in der ersten Zeitnahme einer neuen Blockgröße auftritt(50ms GPU gegenüber 0ms CPU). Hierzu sollte man daran denken, dass bei einer neuen Blockgröße das Programm neu kompiliert wurde und danach neugestartet werden musste. Die sich ergebenden 50ms kommen also von CUDA internen Initialisationsprozessen und ist damit zwar erwähnenswert, aber für uns uninteressant. Als zweiten Punkt sieht man, dass Anzahl der Iterationen und die Zeit direkt proportional sind und es hierbei auch keine Rolle spielt, wie man die Blockgröße wählt. Der Effekt bleibt, trotz sich ändernden Blockgröße, erhalten. Und der letzte Punkt ist, dass bei steigender Blockgröße die Berechnungsdauer der GPU abnimmt. Man kann generell sagen, dass eine größere Blockgröße mehr Performance bringt, dennoch kann es unter Umständen sinnig sein die Blockgröße nicht auf das Maximum zu setzen. Wie man die Blockgröße am besten wählt folgt im Abschnitt 4.5 .

4.4 Datentransfer zwischen Host und Device

Der Datentransfer zwischen Device und Host ist um einiges langsamer als der Datentransfer zwischen Device und Device. Aus diesem Grund sollte es forciert werden beispielsweise so vorzugehen, dass man erst alle nötigen Parameter auf die Grafikkarte lädt, dort das Problem berechnet und das Ergebnis wieder auf den Host zurückschickt. Hier könnte man in meinem Code durchaus noch optimieren, da ich jedes Zwischenergebnis wieder zurückkopiere und die neuen Parameter dann erst wieder auf die Grafikkarte schicke. Ich habe mich in meiner Arbeit dafür entschieden, da es um einiges leichter war so Fehler nachzuvollziehen und Zwischenergebnisse zu plotten. Genrell bringt es auch noch den Vorteil, dass man schon zur Laufzeit plotten könnte und eine Art Animation erstellen

könnte. Meiner Meinung nach ist dieser Punkt nicht unerheblich in der Performancesteigerung, aber dennoch ist er von der Applikation abhhängig und nicht immer vermeidbar. Hier kann man nur auf eine Steigerung der Performance der Mainboards hoffen.

4.5 Blockgröße

Bei der Blochgröße ist wohl mit einer richtigen Einstellung einiges an Leistungssteigerung herauszuholen. Grundsätzlich ist es sinnvoll mindestens so viele Blöcke zu haben, wie die Grafikkarte Multiprozessoren besitzt. Wenn nun aber nur ein Block auf einen Multiprozessor kommt kann es zu Wartezeiten bei der Threadsyncronisation und der Speicherlesezugriffe kommen. Um das zu verhindern sollte man die Anzahl der Blöcke so wählen, dass die Anzahl der Blöcke ungefähr doppelt so groß ist wie die Anzahl der Multiprozessoren der Grafikkarte. Dies führt dann zu einer Überlappung der Blöcke die warten müssen und denen die laufen können. Außerdem sollte die Größe des allokierten Speichers pro Block nicht die Hälfte des allokierbaren Speichers pro Multiporzessors übersteigen.
Zusätzlich kann man sagen, dass je mehr Threads pro Block desto besser sind, da zum Beispiel bei parallelen Problemen, daraus die Leistung resultiert, dennoch geht man damit eine Einschränkung an verfügbaren Registern ein. Die Anzahl der verfügbaren Register pro Thread kann folgendermaßen berechnet werden:

$$\frac{R}{B \times ceil(T, 32)}$$

R Anzahl der Register pro Multiprozessor.
B Anzahl der überlappenden Blöcke.
T Anzahl der Threads pro Block.

Es sollten 64 Threads pro Block nicht unterschritten werden. Mehr Sinn machen die Blockgrößen 192 und 256, da hier die oben genannten Aspekte oft erfüllt sind. Um nun noch die gleichzeitig laufen wraps zu maximieren, ist es erforderlich seine built-in Variablen so zu wählen, dass die Anzahl der benutzten Register minimiert werden.

4.6 Speicher

Auf der Grafikkarte selber ist der Speicherzugriff auf den globalen Speicher wesentlich langsamer als der auf den on-chip Speicher(shared memory). Darum sollte der Gebrauch des globalen Grafikkartenpseichers minimiert werden und die Benutzung des shared memory maximiert. Um dies zu gewährleisten hat sich es bewährt folgendes Schema einzuhalten. Die Daten aus dem Graifkkarten speicher in den shared memory zu laden, alle Threads zu syncronisieren, Berechnung abwickeln, wieder syncronisieren und anschließend die Daten wieder zurückladen. Aber vorsicht, falls ihre Threads nicht untereinander kommunizieren müssen und ihr Problem relativ viel Speicher braucht(große 3D-Arrays), können sie so schnell in Speichermangel geraten. Ich empfehle, falls die Threads nicht untereinander Daten austauschen müssen, es über Register oder den Texturspeicher zu

versuchen.
Im Programm zu dieser Studienarbeit habe ich pro Threads 5 Variablen, die ich in Register lade. Da meine Threads nicht kommunizieren müssen, hielt ich es für unsinnig $22 \times 22 \times 5$ floats in den shared memory zu laden und diese Ladevorgänge zu syncronisieren. Ob es sich nicht doch lohnt sei hier dahin gestellt. Allerdings kann ich mir in meinem Fall sicher sein, dass ich wohl nicht so schnell Speicherprobleme bekommen werde.

4.6.1 Global Memory

Einen Zwischenspeicher besitzt der global memory nicht, deshalb ist es wichtig darauf zu achten wie man seine Zugriffe koordiniert. Der global memory ist in der Lage 32-bit, 64-bit, oder 128-bit words in einer Intruktion zu lesen, dazu ist es allerdings wichtig, dass der Typ der anglegten Variable eine Größe von 4, 8, oder 16 Bytes besitzt. CUDA bietet dazu seine built-in Typen an. Biespiele dafür wären float4 oder float2. Die gennanten Typen füllen die Variablen so auf, dass die Größe der Anforderung entspricht in eienr Intruktion gelesen werden zu können..

4.6.2 Constant Memory

Der constant memory ist in der Lage zwischenzuspeichern. Ein Lesezugriff auf den constant memory kostet einen Lesezugriff auf den Zwischenspeicher des constant memory. Falls das Zwischenspeichern fehlschlägt wird ein Lesezugriff auf den Grafikkartenspeicher benötigt. Wenn alle Threads auf die selbe Adresse zugreifen ist der Zugriff auf den constant memory so schnell wie ein Zugriff auf ein Register. Falls es mehrere Adresse gibt, steigern sich die Kosten linear zu der Anzahl der verschiedenen Adressen.

4.6.3 Shared Memory

Dadurch dass der shared memory ein on-chip Speicher ist, ist er wesentlich schneller als der lokale oder globale Speicher. Seine Leistung kommt dem eines Registers gleich, außer es liegen Bankkonflikte vor. Der Speicher ist in gleichgroße Bänke unterteilt. Diese können gleichzeitig gelesen und beschrieben werden. Damit ist klar, wenn nun beispielsweise zwei Speicherzugriffe zur selben Zeit auf die selbe Bank fallen, muss serialisiert werden. Dies bezeichnet den sogenannten Bankkonflikt. Es ist also beim Gebrauch des shared memory enorm wichtig, dass man sich im Klaren darüber ist, wie das eigene Programm auf die verschiedenen Bänke und in welcher Reihenfolge zugreift. Da eine Zugriffsanfrage eines wraps in zwei Zugriffsanfragen für den ersten Teil und den zweiten Teil des wraps aufgeteilt wird, kann es zwischen diesen beiden Teilen nicht zu einem Bankkonflikt kommen.

4.6.4 Texture Memory

Texture memory wird zwischengespeichert und verbraucht einen Lesezugriff auf den Zwischenspeicher. Bei einen Fehler beim Zwischenspeichern betragen die Kosten einen Zugriff auf den Graikkartenspeicher. Der texture memory ist für 2D Daten optimiert, woraus

folgt, dass Threads, des selben wraps, die beste Leistung erzielen, wenn die Adressen der jeweiligen Zugriffe nah beieinander liegen.

4.6.5 Register

Normalerweise verbrauchen Register keine zusätzliche Zeit bei einem Zugriff, dennoch kann es zu einer Verzögerung durch read-after-write Abhängigkeiten kommen oder durch Bankkonflikte. Der Threadscheduler versucht Bankkonflikte so gut wie möglich zu verhindern. Um dies zu erreichen ist es am besten, wenn man die Blockgröße so wählt, dass sie ein Vielfaches von 64 bildet.

4.7 Kontrollstrukturen

If, switch, do , while, for können auch zu echten Performancebremsen werden. Das liegt daran, dass der aus der Kontrollstruktur resultierende Entscheidungsbaum nacheinander abgearbeitet werden muss. So kann sich die Anzahl der Anweisungen pro Thread erhöhen. Es ist nicht immer möglich um solche Kontrollstrukturen herum zukommen, dennoch sollte man sie tunlichst vermeiden oder wenigstens so viel Abfragen wie möglich in eine von ihnen stecken.
Bei if und switch kann es zur branch predication kommen. Hier wird jeder branch mit einer Art von bool-Flag versehen das entscheidet ob der branch ausgeführt werden soll oder nicht. Dies passiert nur wenn die Anzahl der von der Kontrollstruktur kontrollierten Instruktionen einen bestimmten Schwellwert nicht übersteigt. Der Schwellwert wird auf 7 gesetzt, wenn der Compiler davon aus geht, dass viele verschiedene wraps entstehen, ansonsten beträgt der Schwellwert 4.
Ich habe Kontrollstrukturen nur dazu benutzt herauszufinden, welche Threads die Threads sind, in denen ich die Randbedingung setzen muss und welche Threads keine Randthreads sind. Für mehr sollte man Kontrollstrukturen meiner Meinung nach auch nicht brauchen.

4.8 Mathematische Funktionen

CUDA bietet wie bereits erwähnt verschiedene mathematische Funktionen an, welche jedoch unterschiedlich lang zur Berechnung brauchen. Floating-point Addition, floating-point Multiplikation, floating-point Multiplikation-Addition, integer Addition, bitweise Operationen, Vergleiche, Minimum, Maximum und Casts brauchen 4 Zeitzyklen und sind damit die schnellsten, der angebotenen mathematischen Funktionen. 16 Zeitzyklen brauchen hingegen Quadratwurzeln und der Logarithmus, genauso wie die 32-Bit Integermultiplikation. Um zum Beispiel bei der Integermultiplikation eine Leistungssteigerung zu erhalten kann hier das prefix __ benutzt werden. In diesem Fall handelt sich dann aber nur noch um eine 24-Bit Integermultiplikation. Bei diesem Funktionen ist es möglich dass sie in der Zukunft wiederum langasmer sind als ihr 32-Bit Pendant, aufgrund dessen ist es eine Überlegung wert, ob man sich die Mühe macht, zwei Kernels zu implementieren mit den jeweiligen Funktionen.
Integerdivision und Modulo sind sehr kostspielig und sollten deshalb vermieden werden. Floating-point Division verbraucht 36 Zeitzyklen ist durch die entsprechnende Funktion

auch wieder in der Leistung zu steigern. __sin(x), __cos(x), __exp(x) sind in 32 Zeitzyklen abgearbeitet. Um noch ein paar internen Fehlern aus dem Weg zu gehen, sollten floating-point Konstanten in single-precision ein f als suffix erhalten und die Funktionen mit dem f suffix benutzt werden(sinf(), etc ...).
Leider ließ es sich bei mir nicht vermeiden Modulo zu benutzen. Ich denke aber auch, dass es durch geschicktes Adressieren der Threads durchaus zu eleminieren wäre. Ich habe bewusst nicht die schnelleren Funktionen gewählt, weil ich die CPU und GPU Ergebnisse möglichst nahe beieinander halten wollte, damite die Ausgabe sich nicht beziehungsweise kaum unterscheidet.

5 Resumée

Nvidias CUDA("Compute Unified Device Architecture") ist eine noch relativ junge Entwicklungsumgebung, wodurch man manchmal ein bisschen Nachsicht mit der Reife haben sollte. Generell ist das Konzept aber relativ einfach zu durchschauen und gut um es sich selber anzueignen. Die Dokumentationen sind sehr hilfreich, könnten aber an manchen Stellen einfach noch ein wenig ausführlicher sein. Schade ist, dass man falls man noch weitere Fragen hat, sich eigentlich nur an das hauseigene Forum wenden kann, da man sonst im Internet nicht wirklich fündig wird. Persönlich hätte ich mich über mehr Literatur gefreut.Nichts desto trotz bleibt die Verwendbarkeit und der Nutzen von CUDAs außer Frage. Die Spieleindustrie steckt so viel Elan in die Weiterentwicklung der Grafikkarten, also warum sollte man diese Rechnenleistung im wissenschaftlichen Gebrauch außer Acht lassen? Ich möchte hier nicht sagen, dass die Grafikkarte die Lösung aller Performanceprobleme darstellt, aber sie könnte in Zukunft einen wichtigen Platz einnehmen. Man kann nur hoffen, dass andere Grafikkartenhersteller nachziehen und sowas ähnliches wie CUDA anbieten. Denn den Schritt den CUDA im Sinne der Erlernbarkeit der Grafikkartenprogrammierung gemacht hat, ist enorm. In der Zukunft könnte ich mir vorstellen, dass es Bibliotheken gibt in denen es möglich ist, Berechnungen nach Belieben auf der CPU oder auf der GPU laufen zu lassen. Ich könnte mir sogar vorstellen, dass es bei manchen Programmen Sinn machen würde, Teilbrechnungen auf der CPU und andere Teilbrechnungen auf der GPU laufen zu lassen. Das parallele Programmieren wird wohl durch Entwicklungsumgebungen wie CUDA immer leichter und verständlicher werden. Ich sehe die Schwierigkeiten in der Sycronisation der CPU und GPU und der auf den Prozessoren laufenden Threads. Eine weitere Performancesteigerung wird es wohl dann geben, wenn die Mainboardanbieter den gennannten Trend erkennen und die Übertragungsgeschwindigkeiten zwischen Hauptspeicher und Grafikkartenspeicher optimieren.
Persönlich für mich war das Thema meiner Studienarbeit sehr spannend, da ich zuvor noch nie etwas mit der direkten Programmierung auf der Grafikkarte zu tun hatte und man merkt wie wichtig dieses Thema in der Zukunft werden könnte. Zusätzlich hat es mir sehr viel Übung im Umgang mit mehreren Threads gebracht und mir tiefe Einblicke in die parallele Programmierung gewährt. Ich könnte mir vorstellen in diesem Themenbereich auch meine Diplomarbeit zu erarbeiten oder sogar im späteren Berufsleben mich damit zu beschäftigen.

6 Quellenangaben

- Nvidia: CUDA CUBLAS Library. 2007.
- Nvidia: CUDA CUFFT Library. 2007.
- Nvidia: NVIDIA Compute PTX: Parallel Thread Execution. 2007.
- Nvidia: NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 1.1 . 2007.
- Nvidia: The CUDA Compiler Driver NVCC. 2007.
- http://www.uni-giessen.de/ gd1186/F-Prak/node116.html (letzter Zugriff: 23.02.2008)
- http://forums.nvidia.com/index.php?showtopic=52573&hl=accuracy (letzter Zugriff: 23.02.2008)
- http://de.wikipedia.org/wiki/Wellengleichung (letzter Zugriff: 23.02.2008)
- http://www.nvidia.com/object/cuda_learn.html (letzter Zugriff: 23.02.2008)