Beschleunigung von Aufgaben der parallelen Bildverarbeitung durch Benutzung von NVIDIA-Grafikkarten mit der Compute Unified Device Architecture (CUDA)

Roman Glebov
roman@glebov.de

Abstract
Diese Arbeit behandelt Cuda, Tesla-Architektur und ihre Anwendung bei der Bildverarbeitung.

1 Einleitung

Grafikkarten haben sich als 3D-Beschleuniger für OpenGL-Anwendungen und Spiele etabliert.
Heutzutage findet sich in fast jedem gängigen Personal Computer eine Grafikkarte der Firmen NVIDIA oder ATI.
Die Leistungsfähigkeit der Grafikkarten lässt sich aber noch für andere Zwecke als 3D-Beschleunigung einsetzen. Insbesondere kann die Bildverarbeitung, die besonders hohe Anforderungen an die Verarbeitungsgeschwindigkeit stellt, davon profitieren.

Früher war es sehr umständlich, Grafikkarten für allgemeine Programmierzwecke zu nutzen, da z.B. ein C-Programm nicht auf einer Grafikkarte ausführbar ist, so dass man ein allgemeines Programmierproblem erst auf eine für die Grafikkarte passende Shadersprache umschreiben musste. Der Einsatz von Grafikkarten für allgemeinere Zwecke wurde erst praktikabel, als NVIDIA 2007 eine neue, allgemeiner einsetzbare Hardware-Architektur namens Tesla für ihre Grafikkarten entwickelte. Auf diesen Tesla-Grafikkarten läuft die ebenfalls von NVIDIA entwickelte Programmierentwicklung CUDA (Compute Unified Device Architecture), die man mit Hilfe eines Software Development KITs (SDK) benutzen kann. Mit Tesla und CUDA wurde es nun möglich, fast jedes beliebige C-Programm mit minimalen Anpassungen auf einer Grafikkarte ablaufen zu lassen, so dass die Grafikkarten nun auch für allgemeine EDV-Aufgaben einsetzbar sind.
In dieser Arbeit wird gezeigt, wie durch die Verwendung von CUDA auf Grafikkarten, Aufgaben aus der Bildverarbeitung beschleunigt werden können. Dazu wird zunächst in Kapitel 2 gezeigt, wie durch Verwendung der Architektur Tesla parallelisierbare Programmieraufgaben effizient berechenbar sind. Kapitel 3 erläutert das Programmiermodell der CUDA, welche eine unterliegende Tesla-Architektur benötigt.
Anschließend werden in Kapitel 4 einige Aufgaben aus der Bildverarbeitung mit Hilfe des CUDA SDK implementiert und die Messergebnisse dazu beschrieben. Abschließend erfolgt in Kapitel 5 eine Bewertung der Implementationsergebnisse.

2 Parallelisierbarkeit von Programmen auf Grafikkarten mit Tesla-Architektur

2.1 Einleitung

Die folgenden Abschnitte basieren auf [2] entnommen.
CUDA ist nur mit der neuen Tesla-Architektur der Grafikkarten einsetzbar. Die erste GPU (Graphical Processor Unit) dieser Architektur hieß GeForce 8800 und wurde von NVIDI im November 2006 vorgestellt. Die Tesla-Architektur vereinheitlicht die Fähigkeiten der Grafikkarten; sie vereint Vertex- und Pixel-Prozessoren und erweitert ihre Fähigkeiten. Das ermöglich den Einsatz der Grafikkarten für High Performance Computing (HPC).
Heutzutage implementiert eine große Familie von Grafikkarten der Quadro GeForce 8-, 9- und 280-Serie diese Architektur. Diese Grafikkarten sind in vielen Workstations, tragbaren Computern, Servern und Desktop-Computern eingebaut. Desweiteren wird die Tesla-Architektur in der Tesla-GPU-Computing-Platform für HPC verwendet, die NVIDIA 2007 vorgestellt hat. In diesem Kapitel wird auf die Tesla-Architektur und ihre Eigenschaften eingegangen.

2.2 Gründe für die Entwicklung von Tesla

Traditionell besaßen Grafikkarten eine Pipeline aus Vertex- und Pixel- Prozessoren, die strikt auf ihre Aufgaben beschränkt waren. Die steigende Komplexität der Anforderungen hat dazu geführt, diese Prozessoren programmierbar zu machen. Die Anzahl der Vertex- und

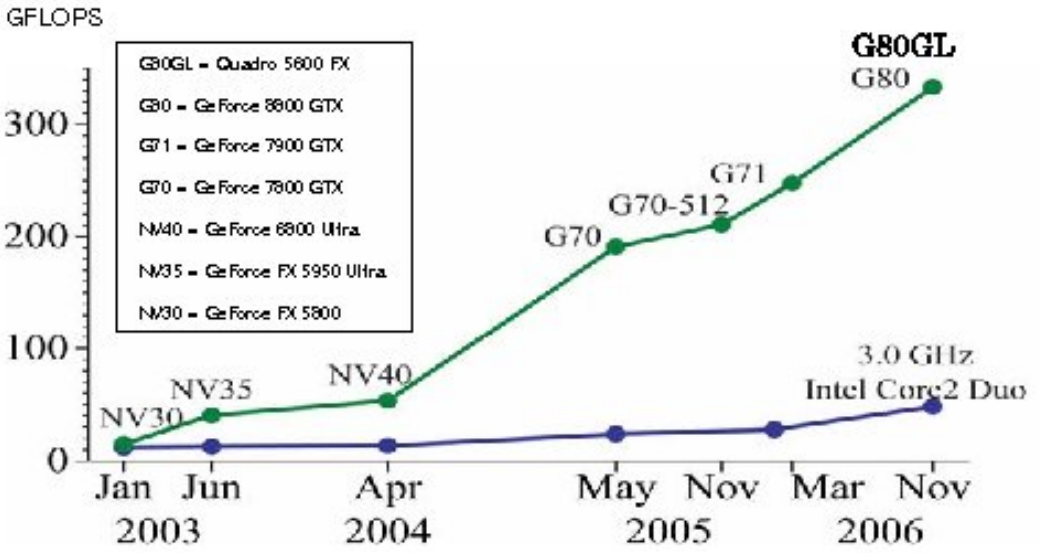

Abbildung 1: Entwicklung der Geschwindigkeit von CPUs und GPUs [1]

Pixel-Prozessoren stand in festem Verhältnis zueinander; aber dies verhinderte eine optimale Ausnutzung der Grafikkarten. Wenn beispielsweise die zu verarbeitenden Daten wenigen großen grafischen Grundelementen bestanden, hatten die Vertex-Prozessoren nichts zu tun, mussten aber fortwährend auf die Pixelprozessoren warten, da sie in einer starren Pipeline miteinander gekoppelt waren.
Die Entwicklung der DirectX-10-Spezifikation (DX10) Anfang (2005, 2006 , 2007?) brachte die Grafikkartenhersteller in ein Dilemma, da DX10 deutlich mehr auf programmierbare Hardware setzte als alle vorherige Versionen, indem es so genannte geometrische SShaderëinführte.
Shader sind einzelne Programme, die direkt auf Pixel- oder Vertex-Prozessoren ablaufen können. Pixelshader verändern direkt die Eigenschaften einzelner Pixel, Vertex-Shader manipulieren hingegen die einzelnen Vertexe. Geometrische Shader arbeiten an vollständigen geometrischen Grundelementen, wie Polygonen, Dreiecken, Linien und Quadraten.
Mit der Zeit wurden Shader-Programme immer komplexer und wurden immer häufiger eingesetzt. Die Architektur der Grafikkarten wurde aus diesem Grund immer komplexer und teurer. Es wurde immer schwieriger vorherzusehen, wie die einzelnen Elemente der Grafikkarten benutzt werden.
Eine Beschreibung der Bauweise der alten GPUs findet sich unter [3]. NVIDIA löste das Problem mit der Einführung der Tesla-Architektur (Tesla Unified Graphics and Computing Architecture) für alle ihre modernen Grafikkarten.

2.3 Die Tesla-Architektur

Die Tesla-Architektur basiert auf einem skalierbaren Prozessorverbund. In Abbildung ist ein Block-Diagramm einer GeForce 8800 GPU zu sehen. Sie besteht aus 128 Streaming-Prozessor-Kernen (SPs), die als 16 Streaming-Multi-Prozessoren in acht unabhängige verarbeitende Einheiten - Textur/Prozessor-Cluster (TPCs) genannt - organisiert sind. Alle Aufgaben der Grafikkarte werden von diesem Prozessorverbund ausgeführt. Der Prozessorverbund ist über ein spezielles Verbindungsnetzwerk an den DRAM-Speicher angebunden. Alle Tesla-Karten besitzen diese Struktur und unterscheiden sich nur durch die Anzahl der Textur/Prozessor-Cluster, die von einem bis zu mehreren Dutzend variiert.

2.4 Streaming-Multiprozessoren (SM) in der Tesla-Architektur

Jeder Streaming-Multiprozessor besteht unter anderem aus acht Streaming-Prozessor-Kernen, zwei Special Function Units (SFU), einem Instruktions-Cache, einem Konstanten-Cache und 16 KB On-Chip Shared Memory.
Jeder Streaming-Prozessor-Kern besitzt eine skalare Multiplikations- und Additions-Einheit. Jede SFU enthält vier Fließkomma-Multiplikatoren und implementiert in Hardware sämtliche trigonometrische Funktionen.
Bei einer Taktrate von 1.5 Ghz der Stream-Prozessor-Kerne einer GeForce 8800 Ultra wird eine Höchstleistung von 36 GFlops per Streaming-Multiprozessor erreicht. Für 8 SMs ergibt dies zusammen 288 GFlops.

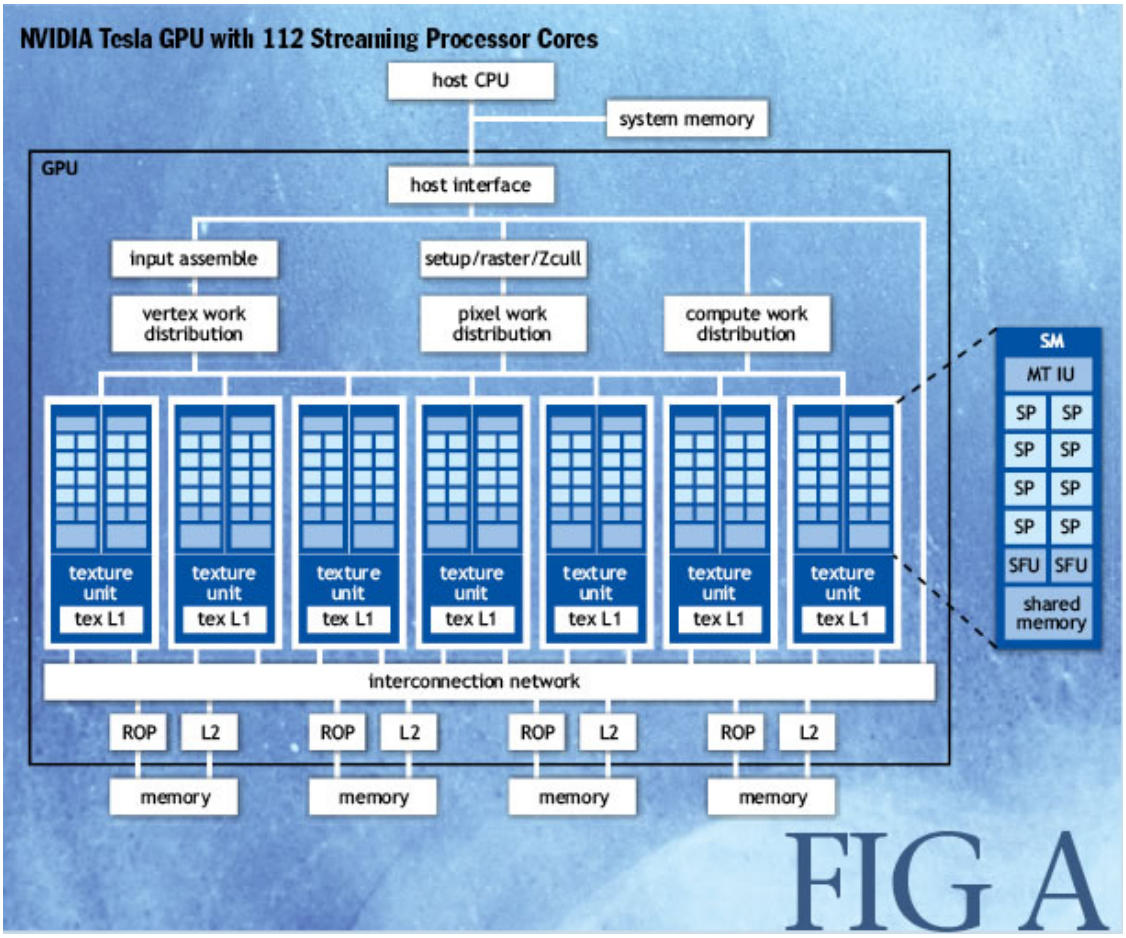

Abbildung 2: Nvidia Tesla GPU mit 112 Streaming-Prozessor-Kernen [4]

2.5 Multithreading in Streaming-Multiprozessoren

Ein graphischer Vertex- oder Pixel-Shader ist ein Programm für einen einzelnen Thread, das beschreibt, wie ein einzelner Thread oder Pixel verarbeitet werden muss. Entsprechend ist ein CUDA Kernel ein C-Programm, das beschreibt, wie für einen einzelnen Thread ein Resultat berechnet werden soll.
Viele graphische und andere Computeranwendungen starten viele Threads, um als Ergebnis gerenderte Bilder oder Arrays mit Ergebnissen zu erhalten.
Um die unterschiedliche Last der verschiedenen Programme dynamisch zu verteilen, führt ein Streaming-Multiprozessor gleichzeitig mehrere unterschiedliche Thread-Programme und gleichzeitig unterschiedliche Typen der Shader Programme aus.¡@@@ Satz umformulieren¿
Um effizient gleichzeitig verschiedene Arten von Thread-Programmen ausführen zu können, unterstützt SM in der Hardware implementierte Multithreading@@@. Jeder Multiprozessor ist in der Lage, bis zu 768 Threads gleichzeitig zu verwalten und auszuführen. Dies erledigt er mit einem in Hardware implementierten Thread Scheduler, der keine zusätzliche Berechnungszeit erfordert. Jeder Thread hat einen eigenen Zustand und kann unabhängig von anderen Threads ausgeführt werden. Eine neue Multithreading-Architektur - genannt Single Instruction Multiple Thread (SIMT)- wurde entwickelt, um dies zu ermöglichen.

2.6 Single Instruction Multiple Thread Architektur (SIMT)

SIMT steht für Single Instruction Multiple Thread. Übersetzt bedeutet das ëinzelne Machinenbefehle für mehrere Threads". Die SIMT-Architektur wird zum Thread-Management in der Tesla-Architektur eingesetzt und wurde von NVIDIA speziell dafür entwickelt. Ein Scheduler für einen Multithread-Stream-Multiprozessor verwaltet die Threads in Gruppen von 32 Stück. Diese Gruppen werden Wraps genannt. In Abbildung @@@ ist die Funktionsweise erklärt.
Jeder Streaming-Multiprozessor verwaltet bis zu 24 solcher Wraps. Alle Threads in einem Wrap gehören zum selben Programm und starten gemeinsam an derselben Programmadresse, aber ansonsten können die Threads unterschiedliche Programmzweige ausführen, d.h im Prozess der Wrap-Ausführung auseinander gehen und wieder zusammentreffen. Zum Zeitpunkt der Weitergabe des nächsten Maschinenbefehls wählt die SIMT-Instruktionseinheit

jeweils aus der Menge der wartenden Wraps einen, der zur Ausführung bereit ist, und gibt den nächsten Befehl aus dem Programm, das die betroffenen Threads gerade abarbeiten, an alle diese Threads gleichzeitig.
Wenn es zum Auseinandergehen der Threads kommt, dann werden die Threads abgeschaltet, für die die aktuelle Stelle des Programm-Codes nicht relevant ist. Im allgemeinen gilt: Jedes Mal, wenn es innerhalb des Wraps mehrere Gruppen gibt, die unterschiedliche Stellen des Programms ausführen, werden diese Gruppen nacheinander ausgeführt, statt parallel, wie es sonst innerhalb eines Wraps üblich ist. Ein Streaming-Multiprozessor verteilt die Threads eines Wraps auf die Streaming-Prozessor-Kerne, wo sie eigenständig ausgeführt werden. Jeder Thread hat dabei seinen eigenen Bereich, wo er die eigenen Register und die eigene Programm-Adresse speichert. Die maximale Leistung wird erreicht, wenn alle Threads immer denselben Programm-Code ohne Abzweigungen ausführen.

2.7 Globale Speicheranbindung in der Tesla-Architektur

Beim Einsatz von DDR3-Hauptspeicher (Double Data Rate Version 3) und einer Speichertaktrate von 1GHz erreichen die Transferraten die maximale Geschwindigkeit von 16 GB/s. ¡Weitere Erklärungen folgen¿

2.8 Architektur für paralleles Rechnen innerhalb von Tesla

Um ein Programmier-Problem effektiv auf die CUDA-Architektur abzubilden, muss man dieses Problem in eine große Anzahl kleiner Probleme aufteilen, die parallel gelöst werden können.
Für die Tesla-Architektur muss eine zweistufige Zerteilung vorgenommen werden. Das heißt, dass ein Problem zuerst in kleinere Problem-Blöcke aufgeteilt wird und jeder solche Block wieder in noch kleinere Teil-Aufgaben aufgeteilt werden kann.
Ein kleines Beispiel: Man hat einen sehr grossen Array von Zahlen, und jede solche Zahl muss mit einer bestimmten Konstante multipliziert werden. Im Sinne der Tesla-Architektur wird dann dieser Array in kleine Teilblöcke der Grösse 512 aufgeteilt. Für jeden dieser Blöcke, werden dann 512 unabhängige Multiplikationen in Auftrag gegeben.
Im Großen und Ganzen schreibt ein Programmierer ein Programm, das eine Sequenz von Resultat-Netzen berechnet. Jedes dieser Netze ist in eine Anzahl von Resultat-Teilblöcken aufgeteilt, die dann parallel berechnet werden. Jeder Resultat-Teilblock wird mit einer Reihe feinverteilter Threads ausgerechnet, die die Ergebnisse für den Block liefern.

2.9 Kooperierende Arrays der Threads (CTA)

Paralleles Rechnen erfordert, dass parallel laufende Threads miteinander kommunizieren und sich miteinander synchronisieren können, damit es möglich ist, ein Resultat effizient zu berechnen.
Um das bis zu einem gewissen Grad zu ermöglichen, implementiert die Tesla-Architektur ein Konzept der kooperierenden Arrays von Threads (Cooperating Thread Arrays, CTA).
Ein CTA besteht aus einer Menge von Threads, die dasselbe Programm berechnen und miteinander kooperieren können. Es besteht aus 1 bis 512 parallelen Threads und hat eine eindeutige Identitätskennung (CTA-ID). Jedem Thread wird eine ThreadID zugewiesen, und jede solche Sammlung kann ein-, zwei- oder dreidimensional organisiert sein. (In diesen Fällen hat die ThreadID entweder ein-, zwei-, oder dreidimensionale Indizes). Threads in einem CTA arbeiten gemeinsam auf den Daten im globalen und schnellen Shared-Speicher und können sich miteinander synchronisieren.
Jeder gestartete CTA-Thread wählt anhand seiner ThreadID und CTA-ID seine Teilaufgabe bzw. seinen Teil der Eingabedaten aus.
Auf einem Streaming-Multiprozessor können gleichzeitig mehrere CTAs in Form der SIMT-Warps ausgeführt werden. Ihre Anzahl ist durch die Ressourcen-Anforderungen - wie z.B die Anzahl der benutzten Register der einzelnen CTAs - begrenzt.

2.10 Skalierbarkeit

Obwohl die Grafikkarten der Tesla Architektur über ganz unterschiedliche Ressourcen verfügen, ermöglicht die gewählte Programm-Struktur eine transparente Skalierbarkeit, da sämtliche CTAs zwischen den zur Verfügung stehenden Streaming-Multiprozessoren transparent verteilt werden können.

3 Cuda-Programmiermodell

3.1 Einleitung

In diesem Abschnitt wird das CUDA-Programmiermodell beschrieben und erklärt, wie es auf die Tesla-Architektur aufsetzt.

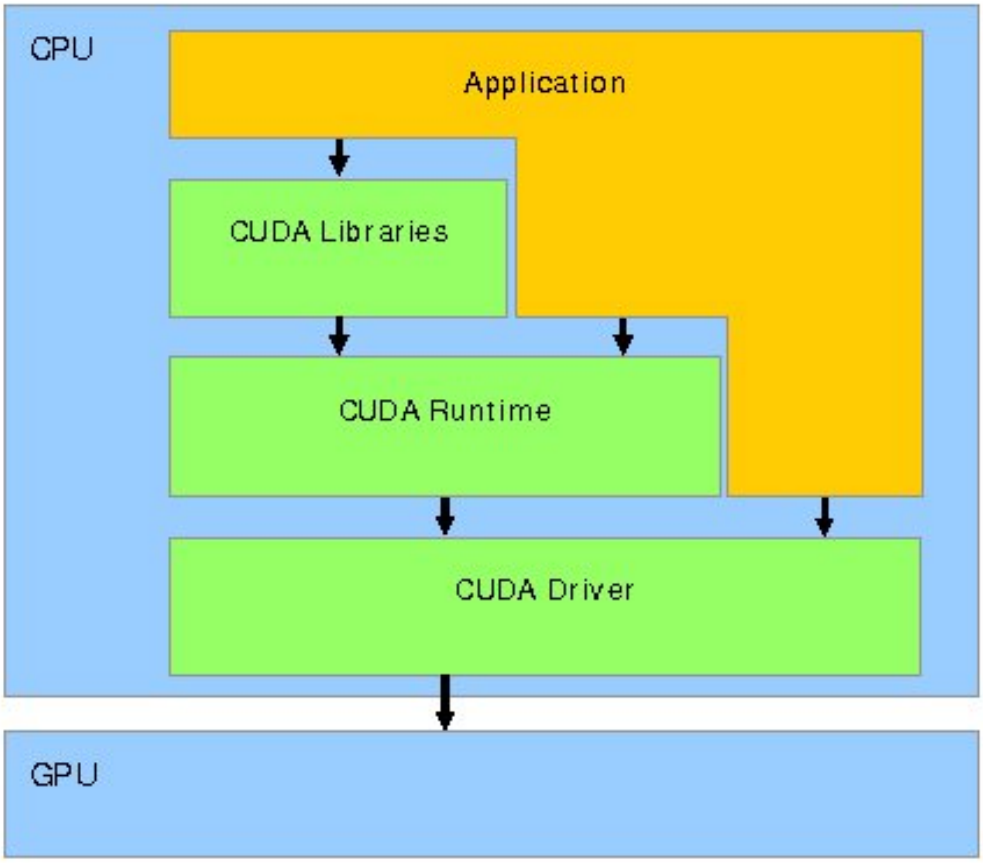

Abbildung 3: CUDA-Architektur [1]

3.2 Grundstruktur

Die Laufzeitumgebung von CUDA verwaltet die GPU als ein eigenständiges Gerät mit einem eigenen Speicher. Die CUDA-Bibliothek liefert Methoden zum Datenaustausch zwischen diesem Gerät und dem System.
Ein CUDA-Programm besteht aus zwei Teilen: Dem Host-Code, der auf dem Computer läuft, und dem Device-Code, der auf der Grafikkarte ausgeführt wird.

3.3 Host-Code

Der Host Code ist dafür verantwortlich, Daten auf die Grafikkarten zu laden, CUDA-Kernels zu starten und Ergebnisse abzuholen.

Device-Code

Der Device-Code beschreibt einen oder mehrere so genannte CUDA-Kernels.
Ein Kernel ist ein Stück Code, der parallel in mehreren Threads auf der Karte ausgeführt wird. Das entspricht einem Programm-Shader wie im vorherigen Abschnitt beschrieben. Diese Kernels werden in den so genannten CUDA-Grids gestartet.
Ein CUDA-Grid besteht aus einer Menge von CUDA-Blocks. Jeder CUDA-Block besteht aus einer Reihe von Threads, die sich auf den gleichen Kernel beziehen und parallel mit normalerweise verschiedenen Teilen der Eingabedaten arbeiten. Im Grunde genommen gleicht diese Aufteilung der im vorherigen Kapitel beschriebenen Architektur für paralleles Rechnen.

3.4 C/C++ Erweiterung

CUDA liefert einen NVCC[1]-Compiler, der C/C++-CUDA-Programme compilieren kann. Dabei ist NVCC in der Lage, den Host-Code automatisch von dem Device-Code zu unterscheiden. Aus dem Device Code wird Maschinen-Code erzeugt, und aus dem Device-Code wird ein PTX[2]-Code generiert, der dann direkt auf der Grafikkarte ausgeführt werden kann.
CUDA erweitert C/C++ mit der Einführung spezieller Operatoren. __global__ für die Definition der Kernel-Startfunktionen, __device__ für die Definition der Variablen im globalen Speicher der Grafikkarte. __shared__ definiert Variablen im Shared Memory, die in einem Thread-Block gemeinsam verwendet werden.

```
// Kernelfunktion
__global__ my_Kernel(
```

[1] NVCC: NVIDIA Compiler Collection
[2] PTX: ein Maschinencode für Tesla-Grafikkarten

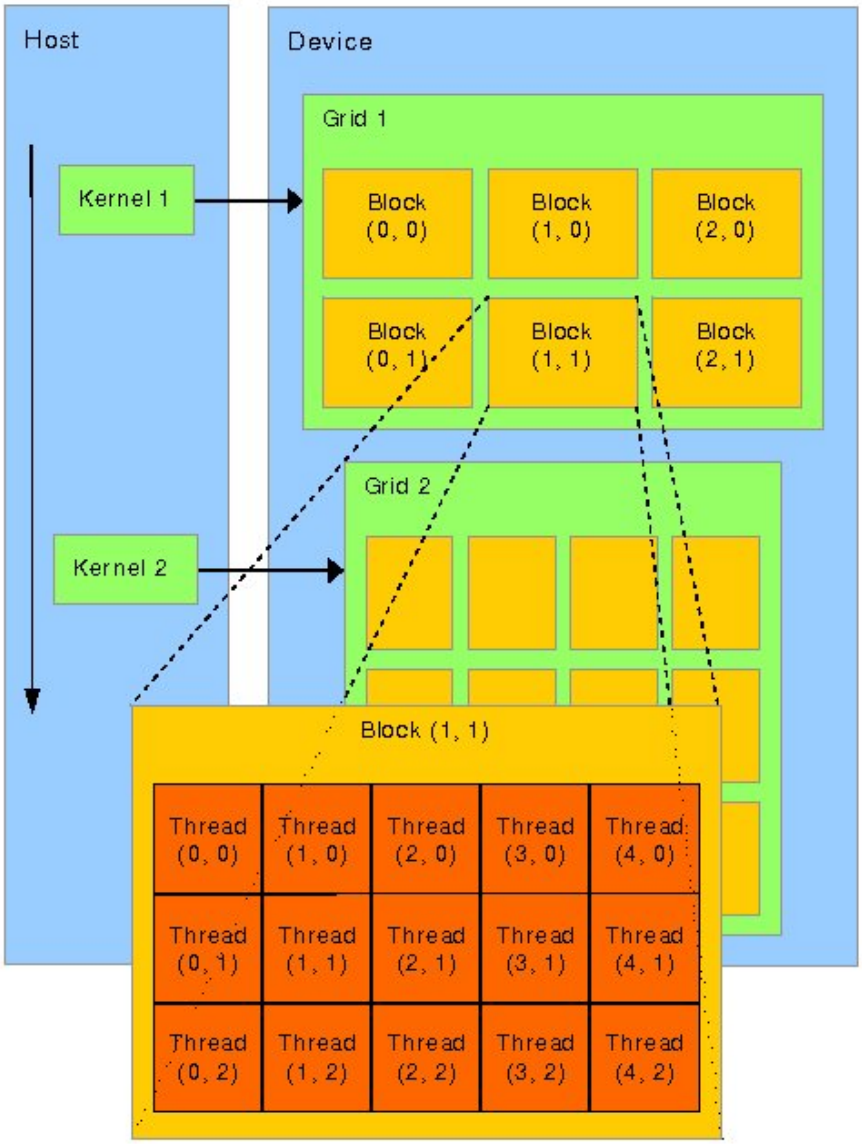

Abbildung 4: CUDA-Grid [1]

```
        char * input_data);
// ein Zeiger auf
//den Grafikkarten-Speicherbereich
__device__ char * some_datapointer;
//shared memory
__shared__ int results[1024];
```

3.5 Speicherarten

Eine Grafikkart ist mit unterschiedlichen Speichertypen ausgestattet. CUDA bietet dafür eine entsprechende Unterstützung. Es gibt folgende Speicherarten:

- **Globaler Speicher** ist in großen Mengen (Hunderte von Megabytes) vorhanden. Alle Threads können uneingeschränkt darauf zugreifen. Leider hat es den Nachteil, dass ein solcher Zugriff 200 Takte dauert. Desweiteren sind solche Speicherzugriffe nicht gecached. Aus diesen Gründen sollte versucht werden, die Zugriffe auf den globalen Speicher zu minimieren.

- **Shared Memory** ist sehr schnell und hat eine geringe Latenz. Es befindet sich direkt auf dem Streaming-Multi-Prozessor. Dieser Speicher ist nicht global: Jeder Thread-Block hat einen eigenen Shared-Memory-Bereich, der nur für die Threads aus demselben Block zugänglich ist. Es ist leider nicht möglich, das Threads aus zwei verschiedenen Blöcken über das Shared Memory Daten miteinander auszutauschen. Die Geschwindigkeit des Zugriffs beträgt 1 Takt.
  Die Größe des Shared Memory ist vergleichsweise gering. Es stehen pro Streaming-Multi-Prozessor nur 16 Kilobyte davon zur Verfügung.

- **Konstanter Speicher** ist ein schneller, zur Laufzeit des Programm nicht änderbarer Speicher. Die Geschwindigkeit des Zugriff beträgt 4 Takte.

- **Textur-Speicher** bietet die Möglichkeit,

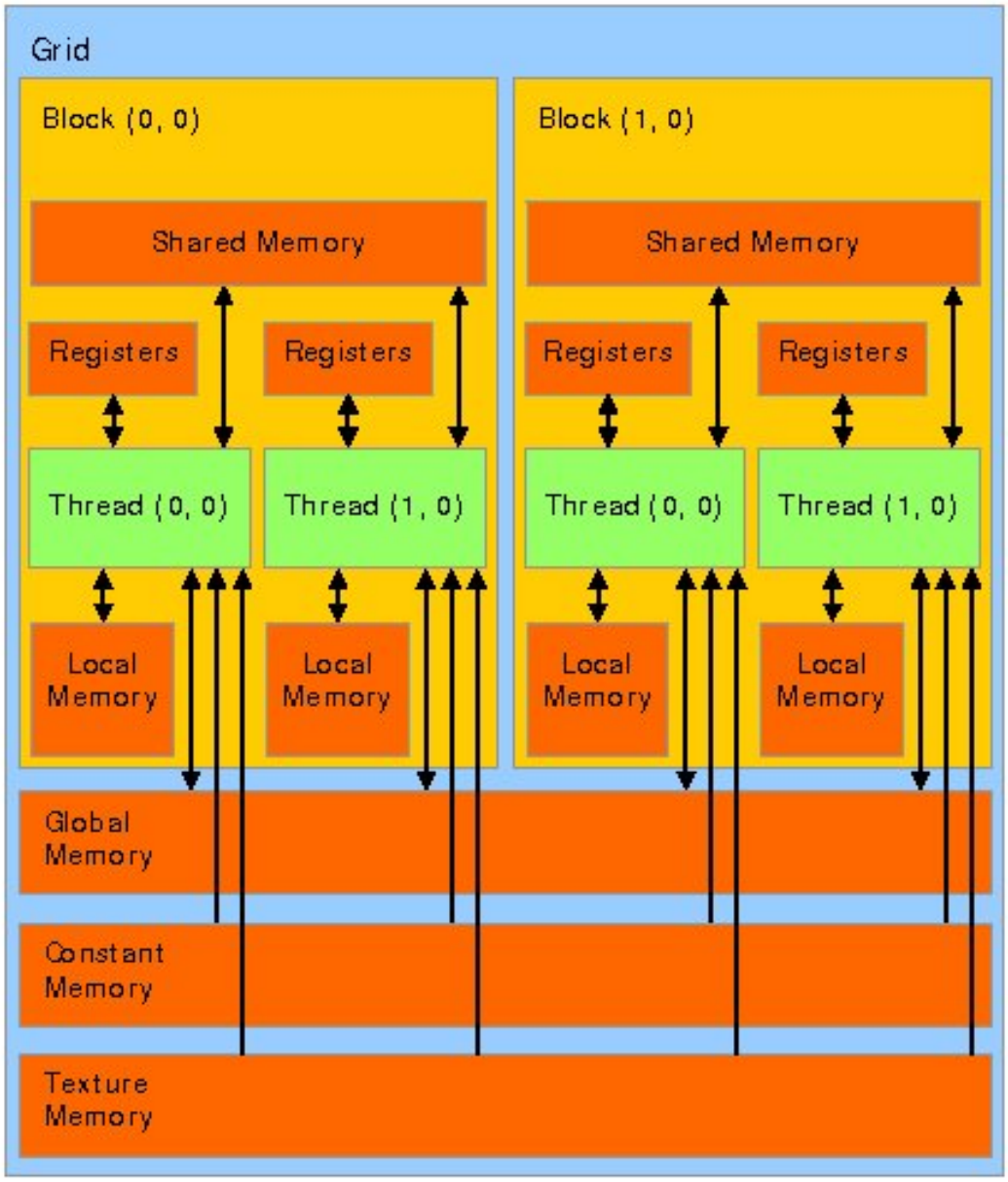

Abbildung 5: Das CUDA-Speicher-Modell [1]

die Daten im Vergleich mit globalem Speicher effektiver zu beziehen, da der Textur-Speicher mit einem Cache ausgestattet ist. Der Nachteil ist aber, dass dieser Speicher nicht direkt beschreibbar ist und vor dem jeweiligen Start des CUDA-Kernels mit Daten befüllt werden muss. Die geänderten Daten müssen dann in den langsamen globalen Speicher oder in das begrenzte Shared Memory geschrieben werden.

3.6 OpenGL-Anbindung

CUDA hat eine Schnittstelle zu OpenGL[3]. Dazu wird mit Pixel-Buffern gearbeitet. Diese stellen eine Erweiterung für OpenGL dar, die es ermöglicht, Daten für OpenGL-Texturen direkt aus einem definierten Grafikkarten-Speicherbereich zu beziehen. Die Texturen werden dann mit den Mitteln von OpenGL angezeigt.
CUDA bietet Mechanismen an, um ein Pixel-Buffer-Objekt mit Daten aus einem der CUDA-Architektur zur Verfügung stehenden globalen Speicherbereich zu befüllen. Die Daten können entweder vom Format Float oder Unsigned Byte sein. Dabei kann es sich um ein-, drei- oder vierkanalige Bildinformationen sein.

3.7 Scalable Link Interconnect (SLI)

CUDA unterstützt eine von NVIDIA entwickelte Technologie, die es ermöglicht, bis zu

[3]OpenGL: Open Graphics Language

drei ähnliche Grafikkarten miteinander zu verbinden.
In diesem Fall agieren diese drei Grafikkarten wie eine große Grafikkarte. In der Theorie liefert diese Konstruktion eine bis zu dreifache Leistungssteigerung bei Spielen oder anderen Computerpgorammen.
Bei CUDA-Programmen kann man bis zu dreimal so viele Threads parallel ausführen wie ohne SLI.

3.8 Hilfsmittel der Entwicklung

Die CUDA-Laufzeitumgebung besitzt einen Debugmodus. Das ist einer der wenigen Möglichkeiten, nach einem im Device-Code aufgetretenen Problem eine Fehlermeldung zu sehen. Außerhalb des Debug-Modus bricht das Program in einem solchen Fehlerfall einfach ohne Meldung ab. Zur genaueren Problemanalyse gibt es einen Emulation-Modus. In diesem Fall läuft der Device-Code statt auf einer Grafikkarte auf einem Computer-Prozessor. Alle Threads werden seriell abgearbeitet. Deswegen ist es nicht möglich, Fehler in der parallelen Zusammenarbeit der Threads in allen Fällen direkt festzustellen. Dafür kann man das Programm direkt in einem Debugger ausführen und den Ablauf des Programms und die Inhalte der Variablen in Echtzeit analysieren.
Zur Performanceoptimierung gibt es einen visuellen Profiler von NVIDIA. Er hilft, Engpässe bei dem Device-Code oder der Zusammenarbeit zwischen dem Host- und Device-Code festzustellen. Es sind z.B ineffiziente Speicherzugriffe oder zu lange Wartezeiten bei Speicher-Transfers. Desweiteren zeigt er an, wie effizient der Grafikchip genutzt wird. Zu diesem Zweck zeigt er viele unterschiedliche Diagrame.

3.9 Einschränkungen

Es gibt bestimmte Einschänkungen. Es ist z.B nicht möglich, rekursive Funktionen im Device-Code zu benutzen, da bei allen auf der Grafikkarte stattfindenden Funktionsaufrufen, der Code für die aufgerufene Funktion einfach an der entsprechenden Stelle eingefügt wird. Das heißt, es wird kein neuer Stack angelegt. Das ist notwendig, da man ansonsten die Funktion nicht effizient ausführen könnte, denn jede solche Stackerweiterung benötigt eine variable Anzahl von Hardwaressourcen, und man kann nicht im voraus sehen, ob die vorhandenen Ressourcen für alle gestarteten Blöcke ausreichen würden. Das macht eine präzise und schnelle Ressourcenplanung unmöglich.
Desweiteren kann man nur bedingt, wie es in C üblich ist, mit Zeigern auf Speicheradressen arbeiten, da ein Grafikkarten-Speicher eine andere Addressierung als das Host-System verwendet. Die Komplexität des Kernels ist begrenzt durch die Anzahl der zur Verfügung stehenden Register, und diese wiederum sind begrenzt durch die Größe des Shared Memory, da es die Register speichert. Außerdem bestimmt die Komplexität des Kernels die maximale Anzahl gleichzeitig parallel ausführbarer Threads, da die Verwaltungsinformationen auch im Shared Memory abgelegt werden.
Es ist nicht möglich, Arrays von CUDA-Texturen[4] zu definieren.
Es existieren keine globalen Synchronisationsmechanismen zwischen parallelen Threads. Es ist nur möglich, die Threads eines CUDA-Blocks miteinander zu synchronisieren. Es werden auch keine atomaren Speicher-Operationen unterstützt. Das schränkt die Menge der realisierbaren parallelen Algorithmen entscheidend ein.

4 Implementierung und Performance-Messung einiger Bildverarbeitungsaufgaben

4.1 Einleitung

Die Bildverarbeitung arbeitet mit großen, mehrdimensionalen Datenstrukturen, an denen Millionen von Operationen ausgeführt werden müssen. Das bringt heutige Prozessoren schnell an ihre Leistungsgrenzen, weil es enorme Größenordnungen an Rechenzeit erfordert.
Viele Bildverarbeitungsaufgaben lassen sich aber gut parallelisieren, und daher stellt sich die Frage, ob CUDA diese Aufgaben nicht deutlich schneller bewältigen kann.
Um dies zu überprüfen, wurden vom Autor ein paar solcher Aufgaben einerseits für eine normale CPU und andererseits unter Benutzung von CUDA auf einer NVIDIA-Grafikkarte implementiert und dann die Performance verglichen.
Das verwendete Testsystem war ein Intel Core Duo System mit 2.4 GHz. Es hat 2GB Arbeitspeicher und eine GeForce 9800GTX Grafikkarte. Eine GeForce 9800 GTX besitzt 16 Streaming Multiprozessoren.

4.2 Testaufgabe Bildfilter

Eine sehr verbreitete Aufgabe in der Bildverarbeitung ist die Filterung eines Bildes mit einem linearen Filter. Die einfachste Form eines solchen Filters ist die Multiplikation jedes Bildpunkts mit einem Skalar.
In einem Array der Größe 640 x 480 von Fließkommazahlen befinden sich die Bildpunkte eines Graustufenbilds.

[4] CUDA-Textur: ein Bereich im Textur-Speicher

In der Programmiersprache C sieht der Code, der die Punkte mit einer Konstante multipliziert, so aus:

```
for( int i = 0; i < 640 * 480 ; i++)
{
    h_data2[i] *= c;
}
```

Und so wurde diese Funktion in CUDA implementiert:

```
//DEVICE CODE Kernel

__global__ void
d_filter( float * d_data, float c){
  int tid = threadIdx.x + blockIdx.x
  * blockDim.x;

  d_data[tid] = d_data[tid] *  c;
}

/// HOST CODE
int main(int argc , char ** argv)
{
....
//Aufruf des Kernels mit 400 Blocks
d_filter<<<640*480 / 768, 768>>>(d_data, c);
....
}
```

Im Unterschied zum einfachen ersten Fall (C-Code) besteht die CUDA-Variante aus zwei Teilen:

1. Aus dem Device-Code, der beschreibt, wie ein einzelner Bildpunkt berechnet wird. (Dazu ist anzumerken, dass ThreadID und BlockID verwendet werden, um auszusuchen welcher Pixelpunkt zu verarbeiten ist.)

2. Und dem Host-Code, der ein CUDA-Grid startet, das aus 400 CUDA-Blöcken besteht. Jeder dieser Blöcke führt 768 Kernels aus. In dieser Konfiguration laufen auf der Grafikkarte $16 * 768 = 12288$ Threads gleichzeitig. Insgesamt wird der d_filter-Kernel $640 * 480 = 307200$ mal für jeden einzelnen Bildpunkt aufgerufen.

Die Messung der jeweils benötigten Zeiten zeigten, dass ein reines CPU-Programm für diese Aufgabe 0.574 Millisekunden benötigt. Dies entspricht einer Verarbeitungsgeschwindigkeit von 535.19 Millionen Bildpunkten je Sekunde.
Das CUDA-Programm brauchte dagegen nur 0.007 Millisekunden, was einer Geschwindigkeit von 43885.71 Millionen Pixeln je Sekunde entspricht.
Daraus ergibt sich, dass das CUDA-Programm die Aufgabe 82 mal so schnell gelöst hat.

4.3 Testaufgabe Histogramm

¡Code und Ergebnisse¿

4.4 Testaufgabe Effiziente Maximum-Suche

¡Code und Ergebnisse¿

5 Zusammenfassung und Fazit

In dieser Arbeit wurden die CUDA-Architektur und ihr -Programmiermodell beschrieben und anhand von einigen Bildverarbeitungsaufgaben erläutert.
Anhand der Messungen wurde bei manchen Aufgaben eine 82-fache Beschleunigung im Vergleich mit einer Standard-Implementation erreicht.
Des weiteren wurde gezeigt, dass nicht alle Aufgaben die Leistung der CUDA-Hardware im vollen Umfang nutzen können.
Die Einfachheit der Handhabung und Geschwindigkeitsvorteile legen nahe, den Einsatz von CUDA für das eine oder andere Bildverarbeitungsproblem zu erwägen.

Literatur

[1] NVIDIA, “Cuda programming guide 1.0,” 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montryma, “Nvidia tesla: A unified graphics and computeing architecture,” IEEE MICRO March-April 2008.

[3] J. Montrym and H. Moreton, “The geforce 6800,” IEEE Micro, vol. 25, no. 2, Mar./Apr. 2005.

[4] J. Nickolls, I. Buck, M. Garland, and K.Skadron, “Scalable parallel programming with cuda,” March/April 2008 ACM QUEUE.

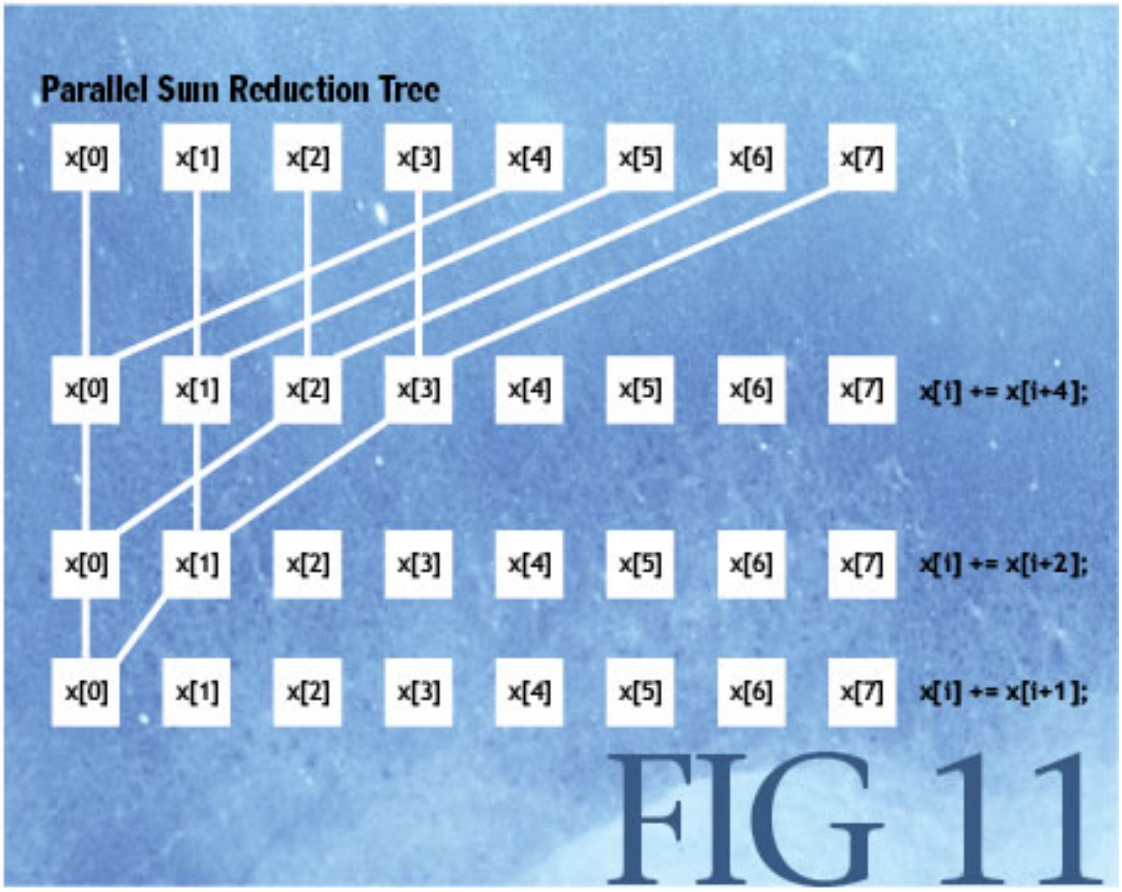

Abbildung 6: parallele Suche [4]