



Fakultät für Ingenieurwissenschaften
Studiengang Kommunikationsinformatik

Master Thesis

Titel der Arbeit

Max Mustermann

4. September 2012

Eidesstattliche Erklärung

Hiermit erkläre ich,

Max Mustermann
geboren am 01. Januar 1980
in Musterstadt

an Eides statt, dass die vorliegende Master Thesis mit dem Titel

Titel der Arbeit

selbständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien wie z.B. Internet – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind. Dies gilt in gleichem Maße für Bilder, Diagramme, Tabellen und sonstige Abbildungen, Programmbeschreibungen oder Quellcodes, etc.

Musterstadt, den 4. September 2012

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Realisierung dieser Arbeit unterstützt haben.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Ebenfalls möchte ich mich bei meiner Familie und meinen Freunden für die großartige Unterstützung während des gesamten Studiums bedanken.

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

Vollkommenheit entsteht offensichtlich nicht dann, wenn man nichts mehr hinzuzufügen hat, sondern wenn man nichts mehr wegnehmen kann.

Antoine de Saint-Exupéry, *Terre des Hommes*

Inhaltsverzeichnis

Einleitung	iii
1. Compute Unified Device Architecture (CUDA)	1
1.1. CUDA Architektur	2
1.2. CUDA Konzepte	3
1.2.1. Kernel	4
1.2.2. Thread-Hierarchie	5
1.2.3. Speicher-Hierarchie	5
1.2.4. Compute Capabilities	8
1.2.5. Build-Prozess	9
2. Eingesetzte Hard- und Software	13
Glossar	I
Literaturverzeichnis	V
Abbildungsverzeichnis	IX
Tabellenverzeichnis	XI
Listings	XIII
A. Anhang	XV

Einleitung

In der heutigen Welt sind Computer und ihre weltweite Vernetzung nicht mehr wegzudenken. Der Austausch von digitalen Informationen hat immer mehr an Bedeutung erlangt. Somit spielt vor allem bei der Kommunikation über unsichere Netze wie das Internet die Authentizität und Integrität der übertragenen Daten eine immer wichtigere Rolle. Aus diesem Grund werden kryptografische Hashfunktionen eingesetzt, um diese Eigenschaften auch im digitalen Zeitalter zu gewährleisten.

Das US National Institute of Standards and Technology (NIST) veranstaltet derzeit einen Wettbewerb um einen neuen Hash-Algorithmus zu finden. Dieser soll dann unter dem Namen SHA-3 als neuer Standard in die SHA-Familie aufgenommen werden und seine beiden Vorgänger¹ ablösen. In der vorliegenden Arbeit werden reduzierte Varianten von drei der fünf Finalisten dieses Wettbewerbes auf ihre Kollisionsresistenz hin untersucht.

¹SHA-1 und SHA-2

Aufbau und Gliederung

Die vorliegende Master Thesis ist in drei Bereiche gegliedert.

Teil 1 (Kapitel 1, 2):

Der erste Teil der Arbeit befasst sich mit der Einführung in CUDA sowie den Grundlagen zu Hashfunktionen.

Teil 2 (Kapitel 3, 4, 5):

In diesem Teil der Arbeit wird zunächst die verwendete Hard- und Software vorgestellt. Anschließend wird die Konzeption und Implementierung des Frameworks zur Kollisionssuche sowie die Portierung der Hashfunktionen auf CUDA beschrieben.

Teil 3 (Kapitel 6):

Im abschließenden Teil der Arbeit werden die gefundenen Kollisionen dargestellt und analysiert.

Kapitel 7:

Das letzte Kapitel beinhaltet das Fazit, das die Bearbeitung des gesamten Themas noch einmal kurz zusammenfasst.

Sonstiges

Eine CD-ROM mit dem Quellcode des Projektes sowie einer digitalen Ausführung dieser Arbeit befindet sich im Anhang.

1. Compute Unified Device Architecture (CUDA)

Die Compute Unified Device Architecture, kurz CUDA genannt, ist eine von NVIDIA entwickelte Technik, die es ermöglicht den Grafikprozessor (GPU) auf der Grafikkarte zur Beschleunigung wissenschaftlicher und technischer Berechnungen zu nutzen. Die Verwendung eines Grafikprozessors für Berechnungen über seinen ursprünglichen Aufgabenbereich hinaus wird auch als General Purpose Computation on Graphics Processing Unit (GPGPU) bezeichnet. Insbesondere bei rechenintensiven Anwendungen kann von der höheren Rechenleistung der GPU profitiert werden.

Mit der Veröffentlichung von CUDA im November 2006 hat NVIDIA eine einfache Schnittstelle geschaffen, um den ansonsten nur für Grafik-Berechnungen genutzten Grafikprozessor auch als Co-Prozessor zu verwenden [NV06]. Zwar konnten schon vorher mittels OpenGL und DirectX bestimmte Berechnungen auf der Grafikkarte durchgeführt werden, jedoch waren diese APIs primär auf Multimedia-Anwendungen ausgelegt. Die eigentliche Berechnung musste deshalb zunächst auf graphische Operationen umgesetzt werden um diese dann auf der GPU berechnen zu können.

Für die Entwicklung von CUDA-Programmen steht die Programmiersprache CUDA C zur Verfügung, die als eine Erweiterung von C um CUDA-spezifische Schlüsselwörter gesehen werden kann. Seit der Fermi-Architektur wird auch die objektorientierte Programmiersprache C++ unterstützt [NV09b]. Weiterhin existieren auch Wrapper für andere populäre Programmiersprachen wie zum Beispiel Java (jCUDA), Perl (KappaCUDA) und Python (PyCUDA). Für die Übersetzung der Quelldateien wird von NVIDIA ein eigener Compiler¹ mitgeliefert.

CUDA kann bereits mit handelsüblichen Grafikkarten ab der GeForce-8-Serie mit mindestens 256 MB Grafikprozessorspeicher genutzt werden [NV12c]. Mit der Tesla-Serie bietet NVIDIA spezielle Grafikkarten an, die für den High-Performance-Bereich

¹NVIDIA CUDA compiler (nvcc)

zum Einsatz in Workstations und Servern vorgesehen sind. Da diese Karten ausschließlich für GPGPU Anwendungen optimiert wurden und überwiegend mit CUDA verwendet werden, verfügen sie in der Regel über keine Anschlüsse für Monitore.

1.1. CUDA Architektur

Um die Leistungsunterschiede zwischen einer CPU und GPU besser zu verstehen, betrachten wir zunächst die unterschiedlichen Ansätze, die hinter den beiden Architekturen stecken. Lange Zeit dominierten im PC-Bereich CPUs mit nur einem Prozessorkern. Eine gängige und vergleichsweise einfache Methode zur Leistungssteigerung war die Erhöhung der Taktfrequenz. Allerdings resultierte die Erhöhung der Leistung durch reine Taktsteigerung in einer nicht mehr sinnvoll handhabbaren Abwärme des Prozessors. Der Trend ging dann in die Entwicklung von CPUs mit mehreren Prozessorkernen. Aktuelle Mehrkernprozessoren verfügen über vier bis acht einzelne Prozessorkerne.

Eine derartige Architektur mit mehreren Prozessorkernen wird auch als Multicore-Architektur bezeichnet, die auf die möglichst schnelle Ausführung von sequenziellen Programmen ausgelegt ist. Analog dazu sind Grafikprozessoren (GPUs) mit ihrer Manycore-Architektur auf datenparallele Aufgaben optimiert und bestehen aus mehreren hundert Recheneinheiten. Unter datenparallelen Aufgaben versteht man dabei Programme, die dieselben Operationen auf einer großen, gleichartigen Datenmenge ausführen. Dieses Konzept wird auch als SIMD (Single Instruction, Multiple Data) bezeichnet.

In Abbildung 1.1 ist der Aufbau einer typischen CPU und GPU vereinfacht dargestellt. Man erkennt deutlich, dass bei der CPU ein Großteil der Chipfläche für Caches (orange) und Kontrolllogik (gelb) verwendet wird. Die eigentlichen Recheneinheiten - hier in grün dargestellt - beanspruchen nur einen vergleichsweise geringen Teil. Die große Kontrolllogik ist notwendig um den Ablauf von sequenziellem Code zu optimieren. Zusätzlich werden große und schnelle Cache-Speicher eingesetzt um die geringe Speicherbandbreite zu verbergen und Latenzen beim Datenzugriff zu minimieren. Bei der GPU hingegen werden Caches und Kontrolllogik nicht in diesem Umfang benötigt und sind deswegen auf das Nötigste reduziert. Der größte Teil der Chipfläche wird hier für die zahlreichen Recheneinheiten verwendet.



Abbildung 1.1.: CPU und GPU im Vergleich [NV11]

Die aktuelle Fermi-Architektur von NVIDIA verfügt über bis zu 512 solcher Recheneinheiten, die auch als CUDA-Cores bezeichnet werden [NV09a]. Jeweils 32 dieser Recheneinheiten sind zu insgesamt 16 Streaming-Multiprozessoren (SM) zusammengefasst und um einen gemeinsamen L2-Cache angeordnet. Zusätzlich verfügt jeder Streaming-Multiprozessor über einen eigenen L1-Cache, der sich in zwei Konfigurationen in einen gemeinsamen lokalen Speicher (Shared Memory) und L1-Cache aufteilen lässt.

1.2. CUDA Konzepte

Ein CUDA-Programm besteht immer aus einem Host und einem Device-Teil. Der Host-Teil ist in der Regel sequentiell und wird auf der CPU ausgeführt. Die Funktionen im Device-Teil werden durch Host-Funktionen aufgerufen und auf dem Grafikprozessor ausgeführt. Der mitgelieferte CUDA-Compiler übernimmt beim Kompilieren automatisch die Trennung in Host- und Devicecode. Der Devicecode wird dabei vom CUDA-Compiler selbst kompiliert während der Hostcode an einen entsprechenden Host-Compiler, unter Linux beispielsweise der `gcc`² oder `g++`³, weitergereicht wird.

Bei der Entwicklung von CUDA-Programmen kann die Grafikkarte mit dem Grafikprozessor als eigenständiger Co-Prozessor mit eigenem Arbeitsspeicher angesehen werden. Um eine Berechnung auf der Grafikkarte durchzuführen, muss zunächst der

²GNU C Compiler

³GNU C++ Compiler

1. Compute Unified Device Architecture (CUDA)

für die Berechnung benötigte Speicher auf dem Device allokiert werden. Anschließend werden die Daten vom Arbeitsspeicher des Hosts in den Speicher der Grafikkarte kopiert. Nachdem der Kopiervorgang abgeschlossen ist, kann die Berechnung auf der Grafikkarte ausgeführt werden. Zum Schluss wird dann das Ergebnis aus dem Speicher der Grafikkarte zurück zum Host kopiert und der allokierte Speicher auf dem Device wieder freigegeben.

Zusammengefasst besteht der Ablauf eines typischen CUDA-Programms aus den folgenden Schritten:

- Benötigten Speicher zur Berechnung auf dem Device allokieren.
- Daten vom Host zum Device kopieren.
- Berechnung auf dem Device durchführen.
- Ergebnis vom Device zum Host kopieren.
- Allokierten Speicher auf dem Device wieder freigegeben.

1.2.1. Kernel

Im CUDA-Kontext werden Funktionen, die auf der Grafikkarte ausgeführt werden als Kernel bezeichnet. Der Code in einem Kernel wird von jedem CUDA-Thread genau einmal ausgeführt. Über die eingebaute Variable `threadIdx` kann ein Thread eindeutig identifiziert werden. Ein Kernel wird im Code mit dem CUDA-Schlüsselwort `__global__` definiert. Beim Aufrufen des Kernels muss außerdem die Anzahl der zu startenden Threads mit angegeben werden. Im Gegensatz zu Funktionen können Kernel keine Werte zurückgeben. Die Rückgabe erfolgt über das Zurückkopieren des entsprechenden Speicherbereichs vom Device zum Host.

Im folgenden Beispiel werden zwei Vektoren (A,B) mit N Elementen addiert und das Ergebnis im Vektor C gespeichert. In der main-Funktion wird mit `VecAdd<<<1, N>>>(A, B, C)` der Kernel mit N Threads aufgerufen. Die Anzahl der Threads ist hierbei gleich der Anzahl der Elemente des Vektors. [NV11, Kapitel 2.1]

```
1 // Kernel definition
2 __global__ void VecAdd(float* A, float* B, float* C) {
3     int i = threadIdx.x;
4     C[i] = A[i] + B[i];
5 }
```

```
8  int main() {  
9      ...  
10     // Kernel invocation with N threads  
11     VecAdd<<<1, N>>>(A, B, C);  
12     ...  
13 }
```

Listing 1.1: CUDA-Beispiel: Vektor Addition

1.2.2. Thread-Hierarchie

Im Vergleich zu CPU-Threads sind CUDA-Threads extrem leichtgewichtig, wodurch Kontextwechsel vom Scheduler wesentlich schneller durchgeführt werden können. Ein CUDA-Thread bildet somit die kleinste funktionale Einheit. Mehrere Threads werden in gleichgroße ein-, zwei- oder dreidimensionale Blöcke gruppiert, die auf der aktuellen Fermi-Architektur aus bis zu 1024 Threads bestehen können. Alle Blöcke zusammen bilden ein Grid, das alle Threads umfasst und somit die parallele Ausführung des Kernels repräsentiert. [NV11]

Beim Ausführen des Kernels sorgt CUDA selbstständig dafür, dass alle Blöcke gleichmäßig auf die vorhandenen Streaming-Multiprozessoren verteilt werden. Alle Threads eines Blocks werden somit auf dem gleichen Streaming-Multiprozessor ausgeführt. Eine Gruppe von bis zu 32 Threads eines Blocks bildet einen sogenannten Warp. Alle Threads in einem solchen Warp werden gleichzeitig (parallel) auf einem Streaming-Multiprozessor ausgeführt. Um eine optimale Auslastung der Hardware zu erreichen sollte die Blockgröße immer ein Vielfaches der Warp-Size betragen. [NV11]

1.2.3. Speicher-Hierarchie

Während der Ausführung können die einzelnen CUDA-Threads auf unterschiedliche Speicher zugreifen. Zunächst einmal verfügt jeder Thread über seinen eigenen lokalen Speicherbereich (Per-Thread Registers, Per-Thread Local Memory), auf den nur er zugreifen kann. Innerhalb eines Blocks teilen sich alle Threads einen gemeinsamen Speicherbereich (Per-Block Shared Memory). Darüber hinaus können alle Threads innerhalb eines Grids auf den globalen Speicher (Per-Grid Global Memory) der Grafikkarte zugreifen, über den auch der Datenaustausch zwischen Host und Device

1. Compute Unified Device Architecture (CUDA)

erfolgt. Zusätzlich gibt es noch zwei Speicher auf die von allen Threads nur lesend zugegriffen werden kann. Der Constant Memory (Per-Grid Constant Memory) dient zum Ablegen von Konstanten und der Texture Memory (Per-Grid Texture Memory) ist auf das Speichern von zweidimensional-organisierten Daten optimiert. [NV11, Kapitel 2.3]

Speicher-Hierarchie aus Programmiersicht

In folgendem Codebeispiel wird veranschaulicht, wie die einzelnen Speicherbereiche aus Programmiersicht genutzt werden können. Außerdem wird die korrekte Verwendung der Schlüsselwörter `__constant__`, `__device__` und `__shared__` erläutert.

```
1  /* Variable im Constant Memory */
2  __constant__ float c_var = 3.14159265;

4  /* Variable im Global Memory */
5  __device__ int g_var;

7  /* Variable im Shared Memory */
8  __shared__ int s_var

10 __global__ void Kernel() {
11     /* Variable in einem Register */
12     int r_var;

14     /* Array im Local Memory */
15     int l_var[20];
16 }
```

Listing 1.2: Deklaration von Variablen in CUDA

Der `__constant__` Qualifizierer deklariert eine Variable, die im Constant Memory angelegt wird. Der Constant Memory ist ein spezieller Teil des Global Memory auf den von allen Threads nur lesend zugegriffen werden kann.

Der `__device__` Qualifizierer deklariert eine Variable, die im Global Memory angelegt wird. Auf diese Variable kann von allen Threads lesend und schreibend zugegriffen werden.

Mit dem `__shared__` Qualifizierer wird eine Variable deklariert, die im Shared Memory angelegt wird und auf die alle Threads innerhalb eines Blocks lesend und schreibend zugreifen können. Bei Grafikkarten auf Basis der Fermi-Architektur beträgt die maximale Größe des Shared Memory bis zu 48 kB. Die Daten werden

dabei im insgesamt 64 kB großen On-Chip Speicher des Streaming-Multiprozessors gespeichert, der sich wahlweise in 48 kB Shared Memory und 16 kB L1 Cache oder in 16 kB Shared Memory und 48 kB L1 Cache aufteilen lässt. [NV09a]

Alle anderen Variablen werden in Registern angelegt. Eine Ausnahme bilden Arrays, die aufgrund ihrer Größe im Local Memory angelegt werden.

Die einzelnen Speichertypen und ihre jeweiligen Eigenschaften sind in der folgenden Tabelle noch einmal zusammengefasst dargestellt.

Speichertyp	Gültigkeitsbereich	Lebensdauer	Zugriff	Geschwindigkeit
Register	Thread	Thread	Lesen & Schreiben	schnell (On-Chip Register)
Local	Thread	Thread	Lesen & Schreiben	langsam (Off-Chip Memory ^a)
Shared	Block	Block	Lesen & Schreiben	schnell (On-Chip Memory ^b)
Global	Grid	Anwendung	Lesen & Schreiben	langsam (Off-Chip Memory ^a)
Constant	Grid	Anwendung	nur Lesen	schnell (Off-Chip Memory ^c)
Texture	Grid	Anwendung	nur Lesen	schnell (Off-Chip Memory ^d)

Tabelle 1.1.: CUDA: Speichertypen

^a400 bis 600 Taktzyklen Latenz

^bsehr geringe Latenz, vergleichbar mit Registerzugriff

^ceigener 8 kB Cache

^deigener 6-8 kB Cache

Abschließend ist in der folgenden Abbildung zum besseren Verständnis die Thread und Speicher-Hierarchie noch einmal grafisch dargestellt.

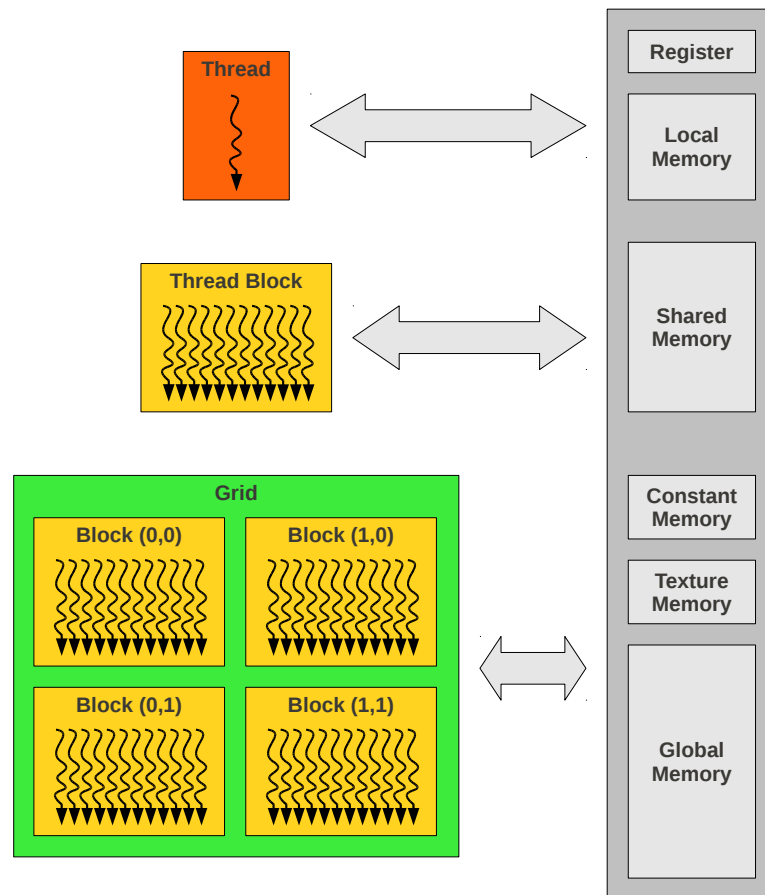


Abbildung 1.2.: CUDA: Thread und Speicher-Hierarchie [NV11]

1.2.4. Compute Capabilities

Mittels der Compute Capability lassen sich die Grafikprozessoren von NVIDIA in verschiedene Klassen einteilen. Dabei beschreibt die Compute Capability die Fähigkeiten der Hardware sowie die von ihr unterstützten Befehle. So wird beispielsweise auch die maximale Anzahl von Threads pro Block sowie die Anzahl der Register pro Streaming-Multiprozessor durch die Compute Capability spezifiziert.

Unterschieden wird die Compute Capability in eine Major und Minor-Revision. Die Major-Revision beschreibt dabei die grundlegende Architektur während die Minor-Revision für eine stufenweise Optimierung der Architektur sowie dem Hinzufügen von neuen Features steht. Die in dieser Arbeit verwendeten Tesla-Karten auf Ba-

sis der Fermi-Architektur werden durch die Compute-Capability 2.x beschrieben. Eine detaillierte Übersicht über die einzelnen Compute Capabilities ist in [NV11, Anhang F] zu finden.

1.2.5. Build-Prozess

Um den Quellcode in ein ausführbares Programm zu übersetzen, wird der von NVIDIA mitgelieferte CUDA-Compiler mit in den Build-Prozess eingebunden. Der Hostcode kann dabei wahlweise mit dem CUDA-Compiler oder direkt mit dem entsprechenden Host-Compiler (z.B. `gcc` oder `g++`) kompiliert werden. Der Devicecode hingegen wird entweder in Binärcode (`cubin`⁴) oder Bytecode (PTX⁵) übersetzt. Letzterer wird dann beim Ausführen für die jeweilige Hardware zur Laufzeit in Maschinencode übersetzt.

Besonderheit beim Linken von Devicecode

Bei der Entwicklung von größeren Programmen ist es in der Regel üblich, den Quellcode in mehrere Übersetzungseinheiten (Dateien) aufzuteilen. Im Build-Prozess werden die einzelnen Übersetzungseinheiten getrennt kompiliert und die so erzeugten Objektdateien mit einem Linker zu einem lauffähigen Programm zusammengefügt.

Während bei CUDA der Hostcode problemlos in mehrere Übersetzungseinheiten aufgeteilt werden kann, ist dies mit dem Devicecode nicht möglich. Der Grund hierfür ist, dass mit dem aktuellen CUDA-Compiler⁶ kein Devicecode gelinkt werden kann. Zum Zeitpunkt des Kompilierens müssen alle Aufrufe innerhalb des Devicecode sowie Device-Funktionen, die aus dem Hostcode aufgerufen werden, in der gleichen Übersetzungseinheit stehen. Mit der im Herbst 2012 erscheinenden Version 5 von CUDA soll es erstmals möglich sein, auch Devicecode zu linken [NV12a].

⁴CUDA binary

⁵Parallel Thread eXecution

⁶`nvcc` 4.2

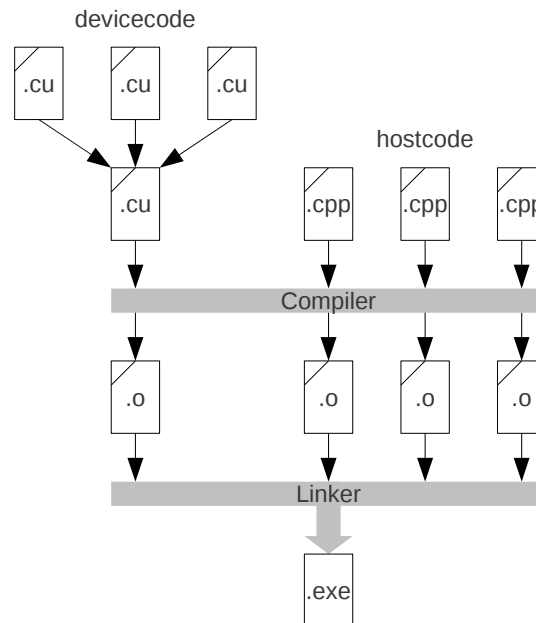


Abbildung 1.3.: CUDA Build-Prozess

Das im Listing 1.3 gezeigte Szenario soll die Problematik beim Linken von Devicecode verdeutlichen. Der Devicecode wurde in die beiden Übersetzungseinheiten `devicecode_A.cu` und `devicecode_B.cu` aufgeteilt. Der Hostcode hingegen besteht nur aus der Datei `hostcode.cpp`. Normalerweise würde man alle Übersetzungseinheiten getrennt kompilieren und die so erzeugten Objektdateien zu einer ausführbaren Datei linkern. Versucht man das Programm auf diese Weise zu erzeugen, schlägt der Build-Prozess mit folgender Fehlermeldung des CUDA-Compilers fehl:

```
Error: External calls are not supported (found non-inlined call to <Funktionsname>)
```

Um das Programm dennoch mit dem CUDA-Compiler zu erstellen, müssen die beiden Übersetzungseinheiten mit dem Devicecode zu einer einzigen Übersetzungseinheit zusammengefügt werden. Im gezeigten Beispiel lässt sich das recht einfach bewerkstelligen, indem man eine separate Datei (`devicecode.cu`) anlegt, in die der Devicecode von allen Übersetzungseinheiten inkludiert wird. Beim Kompilieren wird dann nur eine einzige Objektdatei (`devicecode.o`) für den Devicecode erzeugt, die problemlos mit den Objektdateien vom Hostcode gelinkt werden kann.

```
/* ----- devicecode_A.cu ----- */
#include "devicecode_A.h"
__global__ void kernel_A() {
...
}

/* ----- devicecode_A.h ----- */
__global__ void kernel_A()

/* ----- devicecode_B.cu ----- */
#include "devicecode_B.h"
__global__ void kernel_B() {
...
}

/* ----- devicecode_B.h ----- */
__global__ void kernel_B()

/* ----- devicecode.cu ----- */
#include "devicecode_A.cu"
#include "devicecode_B.cu"

/* ----- devicecode.h ----- */
#include "devicecode_A.h"
#include "devicecode_B.h"

/* ----- hostcode.cpp ----- */
#include "devicecode.h"
int main(void) {
    kernel_A <<<1, 1>>>();
    kernel_B <<<1, 1>>>();
}
```

Listing 1.3: Beispiel: Linken von Devicecode

2. Eingesetzte Hard- und Software

Hardware

Als Grafikkarte für CUDA wird eine NVIDIA Tesla C2075 auf Basis der Fermi-Architektur eingesetzt. Diese verfügt über insgesamt 448 CUDA-Recheneinheiten, die sich auf 14 Streaming-Multiprozessoren verteilen. Da die Grafikkarte ausschließlich als Co-Prozessor genutzt wird, stehen die vollen 6 GB GDDR5-Speicher für CUDA-Anwendungen zur Verfügung. Weitere technische Daten der Grafikkarte sind in der folgenden Tabelle aufgeführt.

Bezeichnung	Tesla C2075
Compute Capability	2.0
CUDA-Recheneinheiten	448
Streaming-Multiprozessoren	14
Chiptakt	575 MHz
Shadertakt	1150 MHz
L1 Cache	16/48 kB ^a
L2 Cache	768 kB
Speicher	6 GB
Speichertyp	GDDR5
Speicherinterface	384-bit
Speicherbandbreite	144 GB/sec
Speichertakt	1500 MHz
Systemschnittstelle	PCIe x16 Gen2

Tabelle 2.1.: Technische Daten der Grafikkarte

^apro Streaming-Multiprozessor

2. Eingesetzte Hard- und Software

Der Host verfügt über einen Intel Core i7-960 Prozessor und 24 GB DDR3-1066 Arbeitsspeicher. Die technischen Daten sind in folgender Tabelle aufgeführt.

Prozessor	Intel Core i7-960
Prozessorkerne	4
Prozessortakt	3200 MHz
L1 Cache	64 kB ^a
L2 Cache	256 kB ^a
L3 Cache	8192 kB
Speicher	24 GB
Speichertyp	DDR3-1066
Speicherinterface	64-bit
Speicherbandbreite	8,5 GB/sec
Speichertakt	1066 MHz

Tabelle 2.2.: Technische Daten des Hosts

^apro Kern

Software

Die Hardware wird mit dem Betriebssystem Ubuntu 12.04 LTS in der 64-bit Variante betrieben. Für die CUDA-Entwicklung sind weiterhin die folgenden Softwarekomponenten installiert.

- Boost C++ Libraries 1.49.0
- GNU-C++-Compiler (g++) 4.4.7
- NVIDIA Grafikkartentreiber 295.41
- NVIDIA CUDA Toolkit 4.2
- NVIDIA CUDA compiler driver (nvcc) release 4.2, V0.2.1221

Glossar

ABI

Application Binary Interface, definiert eine Schnittstelle auf Maschinenebene zwischen einem Programm und dem Betriebssystem, bzw. zwischen einem Programm und einer Bibliothek, oder auch zwischen verschiedenen Bestandteilen des Programms.

AES

Advanced Encryption Standard, ist ein symmetrisches Kryptosystem.

API

Application Programming Interface, ist ein Programmteil, der von einem Softwaresystem anderen Programmen zur Anbindung an das System zur Verfügung gestellt wird.

Compute Capability

Beschreibt die Fähigkeiten der Hardware sowie die von ihr unterstützen Befehle. Mittels der Compute Capability lassen sich die Grafikprozessoren von NVIDIA in verschiedene Klassen einteilen.

Cubin

CUDA binary, Binärcode, der für eine bestimmte Hardwarearchitektur kompiliert wird.

CUDA

Compute Unified Device Architecture, ist eine von Nvidia entwickelte Technik, die es Programmierern erlaubt, Programmteile zu entwickeln, die durch den Grafikprozessor (GPU) auf der Grafikkarte abgearbeitet werden.

DirectX

Ist eine Sammlung von Programmierschnittstellen für multimediaintensive Anwendungen (besonders Spiele) auf der Windows-Plattform.

Distinguished Point

Ist ein Wert der Folge, der einem bestimmten Kriterium entspricht, welches sich einfach und ohne viel Rechenaufwand überprüfen lässt. Zum Beispiel, dass die ersten n Bit eines Werte alle 0 sein müssen.

Fermi

Codename für eine Grafikprozessor-Architektur von NVIDIA.

GPGPU

General Purpose Computation on Graphics Processing Unit, bezeichnet die Verwendung eines Grafikprozessors für Berechnungen über seinen ursprünglichen Aufgabenbereich hinaus.

Kernel

Bezeichnet im CUDA-Kontext Funktionen, die auf der Grafikkarte ausgeführt werden.

LLVM

Low Level Virtual Machine, ist eine modulare Compiler-Architektur mit einem virtuellen Befehlssatz und einer virtuellen Maschine, die einen Hauptprozessor virtualisiert.

Merkle-Damgård-Konstruktion

Ist eine Methode zur Konstruktion von kryptographischen Hash-Funktionen, die auf Arbeiten von Ralph Merkle und Ivan Damgård zurückgeht.

NIST

National Institute of Standards and Technology, ist eine Bundesbehörde der Vereinigten Staaten, die für Standardisierungsprozesse zuständig ist.

nvcc

NVIDIA Cuda Compiler, kompiliert den Devicecode in Binär- oder Bytecode und übergibt den Hostcode an einen entsprechenden Host-Compiler.

Open64

Ist ein Compiler für die Programmiersprachen C++, C und Fortran 77/95, der aus dem MIPSPro-Compiler des Computer-Herstellers Silicon Graphics entstanden ist.

OpenGL

Open Graphics Library, ist eine Spezifikation für eine plattform- und program-

miersprachenunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafik.

PTX

Parallel Thread eXecution, Bytecode, der auf verschiedenen Hardwarearchitekturen lauffähig ist. Wird beim Ausführen für die jeweilige Hardware zur Laufzeit in Maschinencode übersetzt.

SHA

Secure Hash Algorithm, bezeichnet eine Gruppe standardisierter kryptologischer Hashfunktionen.

SIMD

Single Instruction Multiple Data, ist eine Rechnerarchitektur, bei der viele Prozessoren ihre Befehle von einem Befehlsprozessor erhalten und gleichzeitig unterschiedliche Daten verarbeiten. Ein Befehl wird also gleichzeitig auf mehrere Datensätze angewendet und parallel abgearbeitet.

Streaming Multiprozessor (SM)

Sind wesentlicher Bestandteil der GPU und bestehen aus jeweils 32 CUDA-Recheneinheiten.

Suchrunde

Die Suche nach Distinguished Point findet in Runden statt. Eine Runde besteht dabei aus mehreren Schritten.

Tesla

Produktname für Grafikprozessoren von NVIDIA, die auf der Fermi-Architektur basieren.

Warp

Bezeichnet im CUDA-Kontext eine Gruppe von 32 Threads.

Literaturverzeichnis

- [AHMP11] JEAN-PHILIPPE AUMASSON, LUCA HENZEN, WILLI MEIER, RAPHAEL C.-W. PHAN: *SHA-3 proposal BLAKE - Version 1.4*, Januar 2011. http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Blake_FinalRnd.zip
- [AuMP08] JEAN-PHILIPPE AUMASSON, WILLI MEIER, RAPHAEL C.-W. PHAN: *The Hash Function Family LAKE*, 2008. <http://www.131002.net/data/papers/AMP08.pdf>
- [BDPA11] GUIDO BERTONI, JOAN DAEMEN, MICHAËL PEETERS, GILLES VAN ASSCHE: *The KECCAK reference- Version 3.0*, Januar 2011. <http://keccak.noekeon.org/Keccak-reference-3.0.pdf>
- [Bern08] DANIEL J. BERNSTEIN: *ChaCha, a variant of Salsa20*, 2008. <http://cr.yp.to/chacha/chacha-20080120.pdf>
- [BiDu07] ELI BIHAM, ORR DUNKELMAN: *Framework for Iterative Hash Functions — HAIFA*, 2007. <http://eprint.iacr.org/2007/278.pdf>
- [BSI08] BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK: *Leitfaden - Erstellung von Kryptokonzepten*, Juli 2008. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Broschueren/Kryptokonzept/kryptokonzept_pdf.pdf?__blob=publicationFile
- [GKMM11] PRAVEEN GAURAVARAM, LARS R. KNUDSEN, KRYSTIAN MATUSIEWICZ, FLORIAN MENDEL, CHRISTIAN RECHBERGER, MARTIN SCHLÄFFER, SØREN S. THOMSEN: *Grøstl - a SHA-3 candidate*, Januar 2011. <http://www.groestl.info/Groestl.pdf>
- [KaKi10] CHRISTIAN KARPFFINGER, HUBERT KIECHLE: *Kryptologie - Algebraische Methoden und Algorithmen*, 2010.

- [MaST12] KRYSTIAN MATUSIEWICZ, MARTIN SCHLÄFFER, SØREN S. THOMSEN: *Grøstl Implementation Guide*, März 2012.
<http://www.groestl.info/groestl-implementation-guide.pdf>
- [MeOV06] ALFRED J. MENEZES, PAUL C. VAN OORSCHOT, SCOTT A. VANSTONE: *Handbook of Applied Cryptography - Chapter 9*, 2006.
<http://cacr.uwaterloo.ca/hac/about/chap9.pdf>
- [Mici11] PAULIUS MICIKEVICIUS: *Local Memory and Register Spilling*, 2011.
http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf
- [NIST07] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY: *Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family*, November 2007.
http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf
- [NV06] NVIDIA CORPORATION: *NVIDIA präsentiert neue Computing-Technologie*, November 2006.
http://www.nvidia.de/object/IO_37510.html
- [NV09a] NVIDIA CORPORATION: *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
- [NV09b] NVIDIA CORPORATION: *NVIDIA kündigt neue CUDA-GPU-Architektur mit Codenamen „Fermi“ an*, Oktober 2009.
http://www.nvidia.de/object/io_1254418018894.html
- [NV11] NVIDIA CORPORATION: *NVIDIA CUDA C Programming Guide - Version 4.1*, November 2011.
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
- [NV12a] NVIDIA CORPORATION: *Introducing CUDA 5 - GPU Library Object Linking*, 2012. http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/GPU_Library_Object_Linking.pdf

- [NV12b] NVIDIA CORPORATION: *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>
- [NV12c] NVIDIA CORPORATION: *CUDA-fähige Produkte*, 2012. http://www.nvidia.de/object/cuda_gpus_de.html
- [OoWi96] PAUL C. VAN OORSCHOT, MICHAEL J. WIENER: *Parallel Collision Search with Cryptanalytic Applications*, September 1996. <http://people.scs.carleton.ca/~paulv/papers/JoC97.pdf>
- [WIKI12a] WIKIPEDIA: *Kryptologische Hashfunktion* — *Wikipedia, Die freie Enzyklopädie*, Juli 2012. http://de.wikipedia.org/wiki/Kryptologische_Hashfunktion
- [WIKI12b] WIKIPEDIA: *Hashfunktion* — *Wikipedia, Die freie Enzyklopädie*, Juli 2012. <http://de.wikipedia.org/wiki/Hashfunktion>

Abbildungsverzeichnis

1.1. CPU und GPU im Vergleich [NV11]	3
1.2. CUDA: Thread und Speicher-Hierarchie [NV11]	8
1.3. CUDA Build-Prozess	10
A.1. CPU und GPU im Vergleich [NV11]	XVI

Tabellenverzeichnis

1.1. CUDA: Speichertypen	7
2.1. Technische Daten der Grafikkarte	13
2.2. Technische Daten des Hosts	14
A.1. Zusammenfassung aller durchgeführten Kollisionssuchen	XVII

Listings

1.1. CUDA-Beispiel: Vektor Addition	4
1.2. Deklaration von Variablen in CUDA	6
1.3. Beispiel: Linken von Devicecode	11

A. Anhang

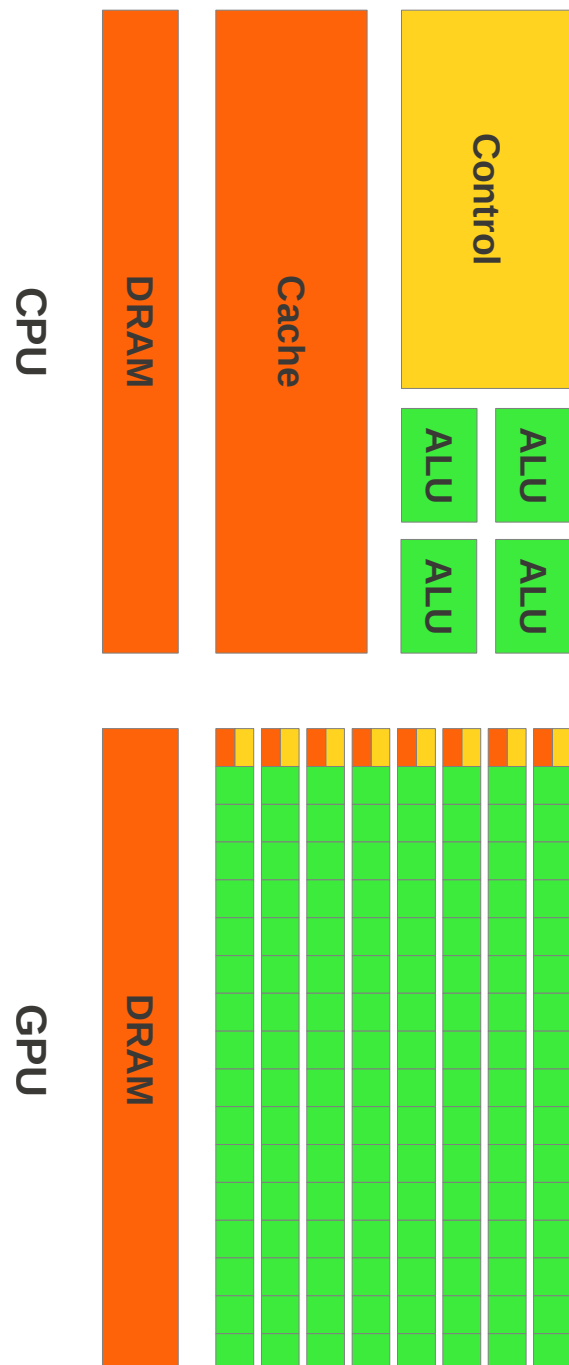


Abbildung A.1.: CPU und GPU im Vergleich [NV11]

Hash-funktion	reduzierte Hash-Länge	DP 0-Bits	CUDA Blks,Thrds	benötigte GPU Zeit	Anzahl Hashes/ms	Anzahl gef. DP's	Anzahl Runden	benötigter RAM	Operationen bis Kollision
BLAKE	80	16	448,256	2:59h	87.997	14.429.854	890	822.468 kB	$\approx 1,72 \times 2^{39}$
	80	20	448,256	2:50h	95.377	928.613	855	57.749 kB	$\approx 1,78 \times 2^{39}$
	80	24	448,256	7:48h	95.486	159.883	2.340	14.208 kB	$\approx 1,22 \times 2^{41}$
	88	24	448,256	52:31h	95.563	1.075.745	15.760	67.245 kB	$\approx 1,03 \times 2^{44}$
	96	24	448,256	872:52h	99.648	18.662.672	273.108	1.098.892 kB	$\approx 1,11 \times 2^{48}$
Grøstl	80	24	224,512	29:50h	19.714	126.810	1.847	12.334 kB	$\approx 1,93 \times 2^{40}$
	80	26	224,512	42:50h	19.715	45.371	2.651	7.722 kB	$\approx 1,38 \times 2^{41}$
	88	24	224,512	463:30h	19.704	1.959.526	28.677	118.166 kB	$\approx 1,87 \times 2^{44}$
Keccak	80	20	448,256	13:45h	29.654	1.399.961	1.286	84.446 kB	$\approx 1,34 \times 2^{40}$
	80	22	448,256	16:06h	29.842	410.926	1.510	28.427 kB	$\approx 1,57 \times 2^{40}$
	80	24	448,256	23:15h	29.891	149.341	2.183	13.611 kB	$\approx 1,14 \times 2^{41}$
	80	26	448,256	28:54h	29.896	46.277	2.712	7.773 kB	$\approx 1,41 \times 2^{41}$
	88	24	448,256	207:28h	29.866	1.329.335	19.455	81.856 kB	$\approx 1,27 \times 2^{44}$

Tabelle A.1.: Zusammenfassung aller durchgeführten Kollisionssuchen