SEMINARARBEIT

zum Thema

„Grafikkartenprogrammierung mit Cuda“

vorgelegt an der
Fakultät für Mathematik und Informatik
der Friedrich Schiller Universität Jena

von:	Tom Möbert A.-S.-Makarenko-Str. 63 07546 Gera
Matrikelnr.:	167510
Studiengang:	Informatik
Gutachter der Friedrich Schiller Universität Jena:	Herr Prof. Dr.-Ing. Martin Bücker
Fachlicher Betreuer:	Herr Dipl.-Inf. Frank Taubert

Inhaltsverzeichnis

1 Notwendigkeit paralleler Programmierung

Seit Beginn der Rechentechnik, Anfang der 40er Jahre, verfolgen Informatiker das Ziel, die zur Verfügung stehende Rechenleistung effizient zu nutzen. Die parallele Programmierung stellt eine wesentliche Maßnahme dar, um dieses Ziel zu erreichen. Mit dieser Arbeit soll die Umsetzung eines gegebenen Problems auf einer Grafikkarte mittels nVidia CUDA vorgestellt werden.

Parallele Programme beschreiben potentiell gleichzeitig ablaufende Aktivitäten, die miteinander kooperieren, um eine gemeinsame Aufgabe zu lösen. Dies setzt Programme voraus, die unabhängig von der Anzahl und der Geschwindigkeit der Prozessoren die gewünschten Ergebnisse liefern. Die Softwareentwicklung für parallele Algorithmen ist deshalb im Vergleich zur sequentiellen Programmierung wesentlich komplexer.

Durch die Aufteilung eines Programms in sogenannte **Threads** („leichtgewichtige Prozesse“) lassen sich die Ressourcen heutiger Mirkoprozessoren effizient nutzen. Threads besitzen dabei folgende Eigenschaften:

- sind sequentielle Befehlsausführungen
- stellen Einheit für die Prozessorzuteilung dar
- laufen in einem Prozessadressraum ab

Mit der Einführung von Threads werden im wesentlichen zwei Ziele verfolgt:

1. Strukturierung unabhängiger Programme und Programmkomponenten
2. Leistungssteigerung durch effiziente Parallelarbeit

Die Leistungsfähigkeit von Programmen ist hauptsächlich abhängig von der Taktrate der CPU. Erhöht sich die Taktrate, wird ebenso das Programm schneller. Allerdings haben die Taktraten heutiger CPUs einen Grenzwert erreicht. Größere Taktraten würden diverse technische Probleme mit sich bringen, wie beispielsweise erhöhte Wärmeentwicklung auf dem Prozessor und damit verbundene, eine geeignete Wärmeabfuhr zu finden. Daher gehen Chiphersteller schon seit einigen Jahren den Weg, möglichst viele Recheneinheiten (Kerne) auf einer CPU unterzubringen. Grafikprozessoren sind mit einer deutlich größeren

Anzahl an Kernen ausgestattet. Programme die entsprechend implementiert wurden, um diese Technologien effizient zu nutzen, können einen deutlichen Geschwindigkeitsvorteil gegenüber einer parallelen Ausführung auf einer CPU aufweisen.

2 Aufgabenstellung

Gegeben sind Kraft-Abstandskurven eines Rasterkraftmikroskops. Rasterkraftmikroskope werden zur Untersuchung von Oberflächen genutzt, um bspw. ein Höhenprofil erstellen zu können. Hierzu wird die vorliegende Oberfläche an möglichst vielen Stellen mit einer ca. 40 nm großen Spitze abgetastet, d.h. es wird zu jeder Messhöhe die der Abtastspitze entgegengebrachte Kraft gemessen. Der Verlauf einer solchen Kraft-Abstandskurve lässt sich in drei lineare Funktionen unterteilen und ist qualitativ in Abbildung 1 dargestellt.

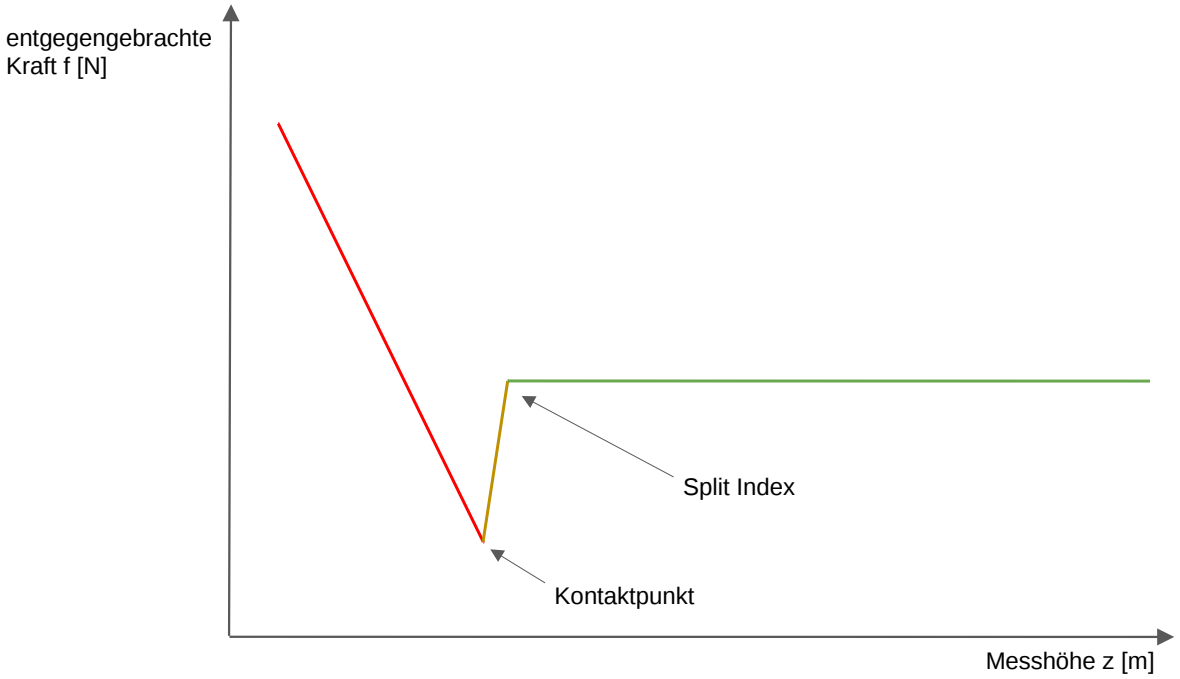


Abbildung 1: Qualitativer Verlauf einer Kraft-Abstandskurve

Der grüne Kurvenverlauf stellt die Annäherung der Abtastspitze zur Probe dar. Auffällig ist, dass die entgegengebrachte Kraft bei diesem Vorgang konstant bleibt. Die gelbe Kurve zeigt, wie die Spitze von den Adhäsionskräften zwischen ihr und der Probe erfasst wird. Die entgegengebrachte Kraft fällt daher rapide ab, bis die Spitze Kontakt mit der Probe hergestellt hat. Der rote Kurvenabschnitt zeigt, wie die auf die Spitze einwirkende Kraft nach Kontaktherstellung stark zunimmt, während weiterhin versucht wird, sich der Probe anzunähern.

Ziel ist es nun, die gegebene Punktewolke eines realen Rasterkraftmikroskops in diese drei linearen Funktionen zu zerlegen. Ein Ausschnitt einer solchen realen Kraft-Abstandskurve ist in Abbildung 2 dargestellt.

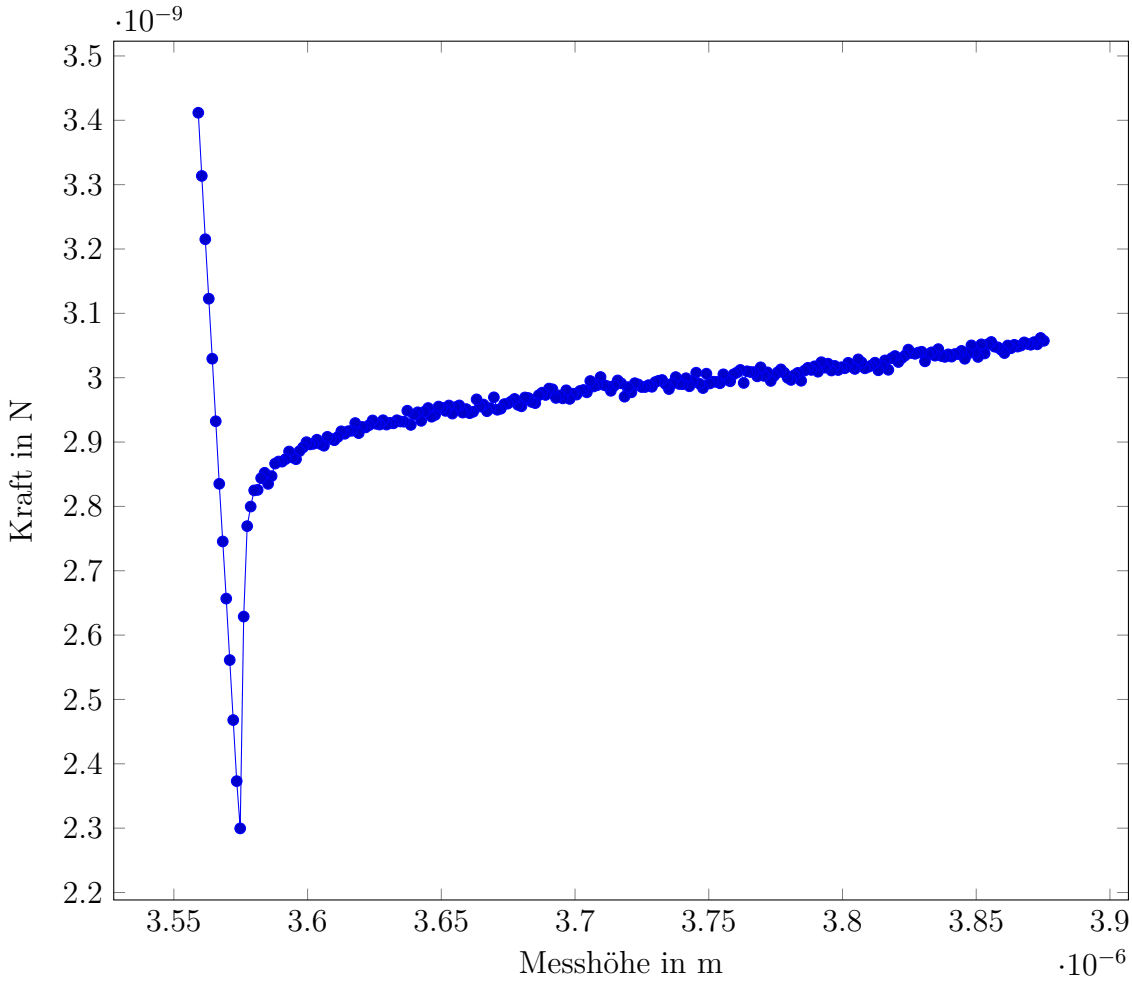


Abbildung 2: Ausschnitt einer Kraft-Abstandskurve

Aufgenommen wurden solche Abtastungen an 256x*256y Positionen, wobei sowohl eine Messung für das Anfahren auf die Probe als auch das Ablassen von ihr aufgezeichnet wurde. Es liegen also insgesamt 131072 Datensätze vor, deren je 300 Abtastwerte mittels dreifach linearer Regression auf einer GPU in die oben beschriebene Form gebracht werden sollen, indem der Kontaktpunkt mit der Probe und der Splitindex gefunden werden.

2.1 Einlesen der Datensätze

Die Datensätze liegen als Textdeateien vor, wobei jede Messposition in einer eigenen Datei gespeichert ist. Es existieren somit 65536 Textdateien, in denen die Messwerte für das An- und Abfahren von der Probe zeilenweise ähnlich wie in einer CSV Datei gespeichert sind. Das Einlesen dieser vielen 100 KiB kleinen Dateien wird durch das häufige öffnen und schließen der Dateien und dem damit verbundenen Overhead seitens des Betriebssystems verlangsamt. Zudem werden für die nötigen Berechnungen nur die ersten zwei der insgesamt 14 Datenspalten benötigt, was Caching seitens des OS zusätzlich erschwert.

Das Einlesen aller 65536 Textdateien benötigt auf einer SSD[1] ca. 2 Minuten. Dies kann auf 40 Sekunden reduziert werden, wenn das Einlesen mittels openMP Tasks parallelisiert erfolgt. Der Einlesevorgang auf einer konventionellen HDD benötigt aufgrund der trägen Mechanik hingegen mind. 10 Minuten.

Um dies zu umgehen, ist eine Vorverarbeitung nötig, die darin besteht, die Textdateien einmalig einzulesen, die Daten zu filtern, zu sortieren und sie anschließend als Binär Blob auszugeben. Der Einlesevorgang kann somit auf wenige Sekunden reduziert werden.

[1] Testsystem: openSUSE 13.2, Linux Kernel 3.16.7 x86_64, OCZ Vertex 3

2.2 Implementierung des Kernels

Als Eingabe erhält der Kernel die aus dem Binär Blob gelesenen Datensätze. Der Kernel hat die Aufgabe die Eingangs beschriebene dreifache lineare Regression für jeden Datensatz in single-precision floating point durchzuführen.

```
1  void kernel(const point_t* pts, const int nSets)
   {
3      int myAddr = threadIdx.x+blockIdx.x*blockDim.x;

5      if(myAddr < nSets)
       {
7          const my_size_t nPoints = pts[myAddr].n;

9          contactIdx = calcContactPoint(pts[0:nPoints])
           __syncthreads();
11
           fitPoints(pts[0:contactIdx]);
13         __syncthreads();

15         my_size_t splitIdx = contactIdx+10;
           fitPoints(pts[contactIdx:splitIdx]);
17
           fitPoints(pts[splitIdx:nPoints]);
19     }
   }
```

Codefragment 1: Implementierung des Kernel in Pseudocode

Aufgrund der vergleichsweise geringen Anzahl an Messwerten pro Datensätze (=300) erfolgt eine naive Parallelisierung der Datensätze: Jeder Thread der Grafikkarte bearbeitet einen Datensatz. Wie in Codelisting 1 dargestellt wird hierfür

- die gegebene Punktwolke abgeleitet, um den Kontaktpunkt zu bestimmen (Zeile 7),
- eine lineare Regression (polyfit) der Kurve vom Ende der Messung bis zum Kontakt mit dem Medium durchgeführt (Zeile 12),
- der Split-Index erraten[2] (Zeile 15),
- ein polyfit der Kurve zwischen dem Kontaktpunkt bis Split-Index (Zeile 16) und
- ein polyfit der Kurve zwischen Split-Index und Anfang der Messung durchgeführt (Zeile 18).

[2]Die Berechnung des Split-Indexes erwies sich als zu aufwendig und instabil, daher wurde auf eine Implementierung im Rahmen dieses Seminars verzichtet.

Grafikkarten arbeiten im SIMD Verfahren. Um zu verhindern, dass die Threads divergieren, da sich die Kontaktpunkte und die Länge der zu interpolierenden Kurve zwangsläufig zwischen den Datensätzen unterscheiden, wurde in den Zeilen 10 und 13 Synchronisationsbarrieren auf Thread-Block Ebene eingeführt.

2.3 Datenmodell

Um eine effiziente Verarbeitung auf der Grafikkarte zu erreichen, müssen die Datensätze in geeigneter Weise strukturiert werden. Dazu muss entschieden werden, ob sie Daten als Array of Struct (AoS) oder als Struct of Array (SoA) übergeben werden. Nachfolgend werden diese Ansätze diskutiert.

2.3.1 Array of Struct

```
struct tuple_t { float z,f; };
tuple_t datasets[M][N];
```

Codefragment 2: Datenlayout AoS row major

Die Auslegung der Daten als AoS bedeutet, dass die Daten als zweidimensionales Array übergeben werden, deren Element ein Verbunddatentyp ist, welcher die gemessene Kraft f an der entsprechenden Messhöhe z enthält. Wird das Array wie in Listing 2 dargestellt row major im Speicher abgelegt, liegen die N Datensätze zeilenweise im Speicher, sodass die i-te Speicherzeile die M Messwerte des i-ten Datensatzes enthält. Dies ist jedoch für Grafikkarten ungeeignet, da die Messwerte für die einzelnen Threads zu weit auseinander liegen. D.h. wird eine Cacheline geladen, kann hiervon nur das erste Element (der erste Messwert) verwendet werden. Die anderen Threads benötigen ebenfalls den ersten Messwert allerdings den des ihnen zugeordneten Datensatzes. Diese befinden sich in anderen Zeilen des Speichers. Es müssen somit weitere Cachelines geladen werden, was schlimmstenfalls zu einer Ausserialisierung der Threads führt.

```
struct tuple_t { float z,f; };
tuple_t datasets[N][M];
```

Codefragment 3: Datenlayout AoS column major

Es empfiehlt sich daher, die Datensätze column major im Speicher abzulegen, wie in Listing 3 dargestellt. Die i-te Zeile des Speichers enthält somit die i-ten Messwerte aller Datensätze. Da jeder Thread einen Datensatz bearbeitet, können alle 32 Threads eines Warps durch eine Cacheline mit Daten versorgt werden.

2.3.2 Struct of Array

Bei NVidia Grafikkarten vor der Pascal Architektur erhält jeder Thread durch einen Lesevorgang ein 32 bit Wort aus der Cacheline[3]. tuple_t hat jedoch eine Größe von 8 Bytes. Es braucht also zwei Lesevorgänge um alle Threads eines Warps mit Daten zu versorgen. Um einen Performancevergleich zu der älteren Kepler Architektur zu erhalten, wird daher auch eine SoA Variante, wie in Listing 4 dargestellt, implementiert.

```
struct tuple_t { float z[N],f[N]; };
tuple_t datasets[M];
```

Codefragment 4: Datenlayout SoA

Hier ist tuple_t ein reines SoA, welches M mal existiert und jeweils Zeiger auf Arrays enthält, welche die N Messwerte z und f des jeweiligen Datensatzes enthalten. Diese können somit von N Threads single strided bearbeitet werden, wie schematisch in Listing 5 dargestellt. Zwar hat tuple_t nun eine Größe von 16 Byte, da es zwei 8 Byte Zeiger enthält. Dieses Element muss jedoch nur einmalig gelesen werden, da die enthaltenen Zeiger von den N Threads gemeinsam verwendet werden können.

```
tuple_t datasets[M];
extern int threadId;

for (i=0; i<M; i++)
{
    float myZ = datasets[i].z[threadId];
    float myForce = datasets[i].f[threadId];

    // doWork(myZ, myForce);
}
```

Codefragment 5: Bearbeitung eines SoA

[3]http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#shared-memory-bandwidth

3 Benchmarking

In diesem Abschnitt wird die entwickelte Implementierung getestet. Die Tests erfolgen hauptsächlich auf dem Privatrechner des Autors, welcher im Folgenden als Testsystem bezeichnet wird. Es verfügt über folgende Spezifikationen:

- OS: openSUSE 13.2, Linux Kernel 3.16.7 x86_64
- CPU: Intel Core i5-3570K (3.40GHz, 4 Rechenkerne, Hyperthreading deaktiviert)
- GPU: NVidia GeForce GTX 1060 (Pascal Architektur, 1280 HW-Threads, 6GB GDDR5)

Darüberhinaus finden Tests auf gpu03.inf-ra.uni-jena.de, nachfolgend als gpu03 bezeichnet, um einen Performancevergleich zu NVidias Kepler Architektur zu erhalten.

- OS: Ubuntu 14.04.5 LTS, Linux Kernel 3.13.0 x86_64
- CPU: Intel Core i7-4770 (3.40GHz, 4 Rechenkerne, Hyperthreading aktiviert)
- GPU: NVidia GeForce GTX 780 (Kepler Architektur, 2304 HW-Threads, 3GB GDDR5)

3.1 Implementierungsvergleich bei fixer Problemgröße

Eine in Python geschriebene Referenzimplementierung diente als Vorlage für die im Rahmen dieses Seminars erstellte C++ Implementierung. Sie bearbeitet die Datensätze seriell auf der CPU und benötigt dafür auf dem Testsystem insgesamt 11 Minuten.

Die C++ Implementierung benötigt die für die in Abschnitt 2.1 beschriebene Vorverarbeitung der Datensätze 40 Sekunden. Parallel verarbeitet werden die 2^{17} Datensätze schnellstenfalls in ca. 250 Millisekunden. Der Speedup beider Implementierungen auf der CPU beträgt somit.

$$S = \frac{T_{seriell}}{T_{parallel}} = \frac{11 * 60}{40 + 0,25} = 16,4$$

Zu beachten ist, dass die C++ Implementierung verglichen zur Referenzimplementierung aus Zeitgründen unvollständig ist, weshalb die Aussagekraft des Speedups mit Vorsicht zu

genießen ist. Es wird jedoch erwartet, dass eine vollständige Implementierung nicht mehr als das vierfache der hier gemessenen Zeit benötigt.

Bei der GPU Implementierung hat sich eine Threadkonfiguration von

- 1024 Threads pro Block auf der Pascal-Architektur und
- 256 Threads pro Block auf der Kepler-Architektur

als die performanteste Variante herausgestellt. Die Bearbeitung der Datensätze erfolgt schnellstenfalls

- auf der Pascal-Architektur in 1,7 Millisekunden und
- auf der Kepler-Architektur in 3 Millisekunden.

Hinzu kommt jedoch der nicht zu vernachlässigende Overhead des Kopiervorgangs der Datensätze auf den Grafikkartenspeicher, welcher

- bei der Pascal-Architektur ca. 45 Millisekunden und
- auf der Kepler-Architektur ca. 55 Millisekunden

beträgt. Es ergibt sich somit ein Speedup vorbehaltlich der unvollständigen Implementierung von

$$S = \frac{T_{seriell}}{T_{parallel}} = \frac{11 * 60}{40 + 0,045 + 0,0017} = 16,5$$

Darüberhinaus konnten die in Listing 1 eingeführten Thread-Block-Synchronisationsbarrieren keine messbare Veränderung der Laufzeit hervorbringen, da bei nur 300 Messwerten pro Datensatz eine mögliche Divergenz der GPU-Threads zu gering ist.

Zwar fanden alle Berechnungen, wie eingangs in Abschnitt 2.2 erwähnt, in single-precision floating point statt. Jedoch wurde im Rahmen dieses Benchmarks auch ein Test mit double-precision durchgeführt. Dies hatte zur Folge, dass sich die Laufzeiten sowohl auf CPU als auch auf GPU lediglich verdoppelten, da dies die Größe eines tuple_t verdoppelte und somit zwei Cachelines benötigt wurden, um alle Threads mit Daten zu versorgen.

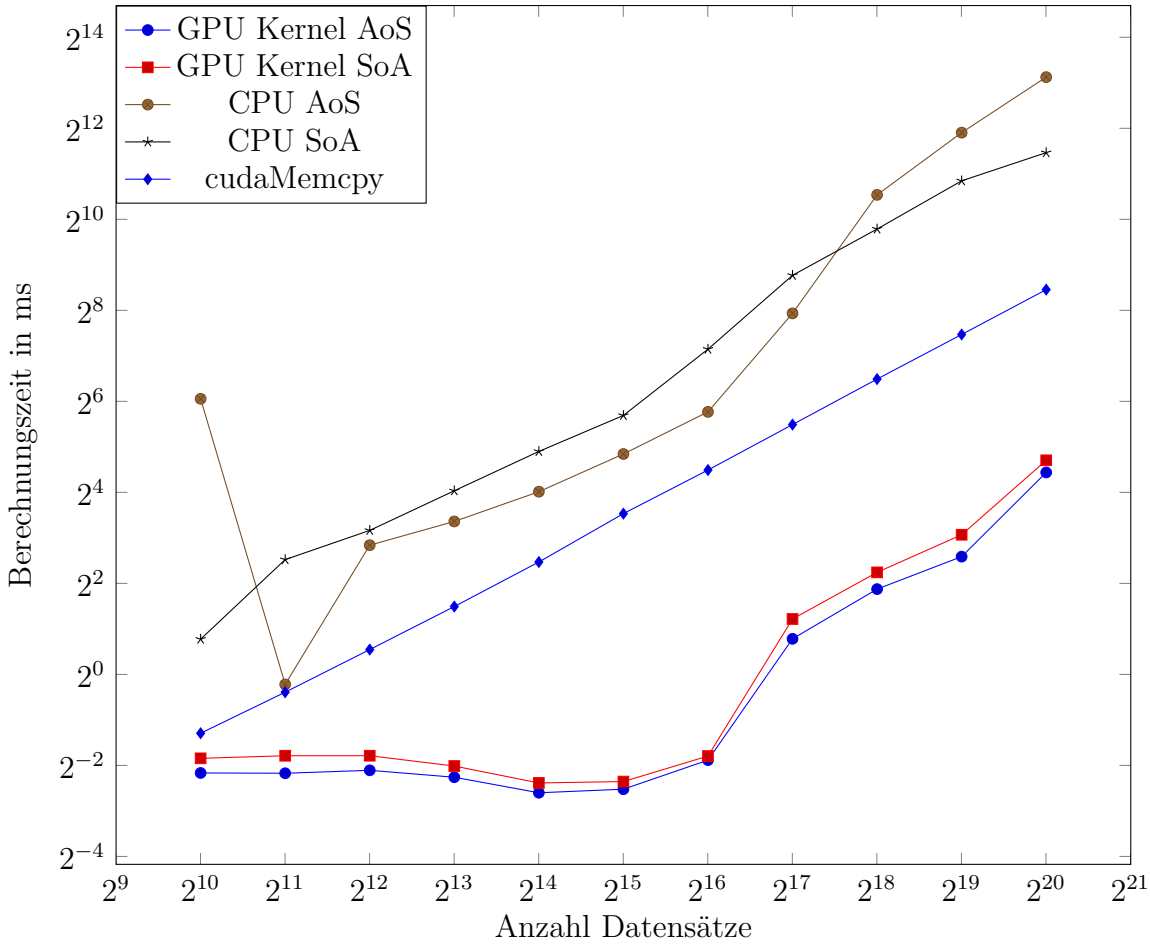
3.2 CPU vs. GPU bei variierender Problemgröße

In diesem Abschnitt soll gezeigt werden, wie performant sich die GPU Implementierung gegen die CPU schlägt wenn die Anzahl der Datensätze variiert wird. Hierzu werden jetzt bis zu 2^{20} Datensätze bearbeitet. Mehr Datensätze zu bearbeiten ist aufgrund des nur 8 GiB großen Arbeitsspeicher auf dem Testsystem mit der gegenwärtigen Implementierung nicht möglich. Der begrenzte Arbeitsspeicher ist auch ausschlaggebend, dass dieses Benchmark nur mit single-precision floats durchgeführt werden kann.

Abbildungen 3 und 4 visualisieren die Testreihe und zeigen, dass die Berechnungszeiten erwartungsgemäß linear zur Problemgröße ansteigen. Die GPU ist jedoch mit 27 Millisekunden bei 2^{20} Datensätzen deutlich schneller als die CPU mit bestenfalls 2,8 Sekunden. Überraschend ist auch, dass der SoA mit dem AoS Ansatz auf der GPU nahezu gleich auf liegt. Eine auf der Pascal-Architektur erwartete Präferenz für den AoS Ansatz ist zwar leicht vorhanden, grenzt sich jedoch mit einem Unterschied von 5 Millisekunden bei 2^{20} Datensätzen nur unwesentlich vom SoA Ansatz ab, siehe Abbildung 3. Erkennbar ist, dass der Zugriff auf den Grafikkartenspeicher bei der Pacsal-Architektur optimiert wurde. Dennoch bleibt die PCI-E Schnittstelle ein Flaschenhals.

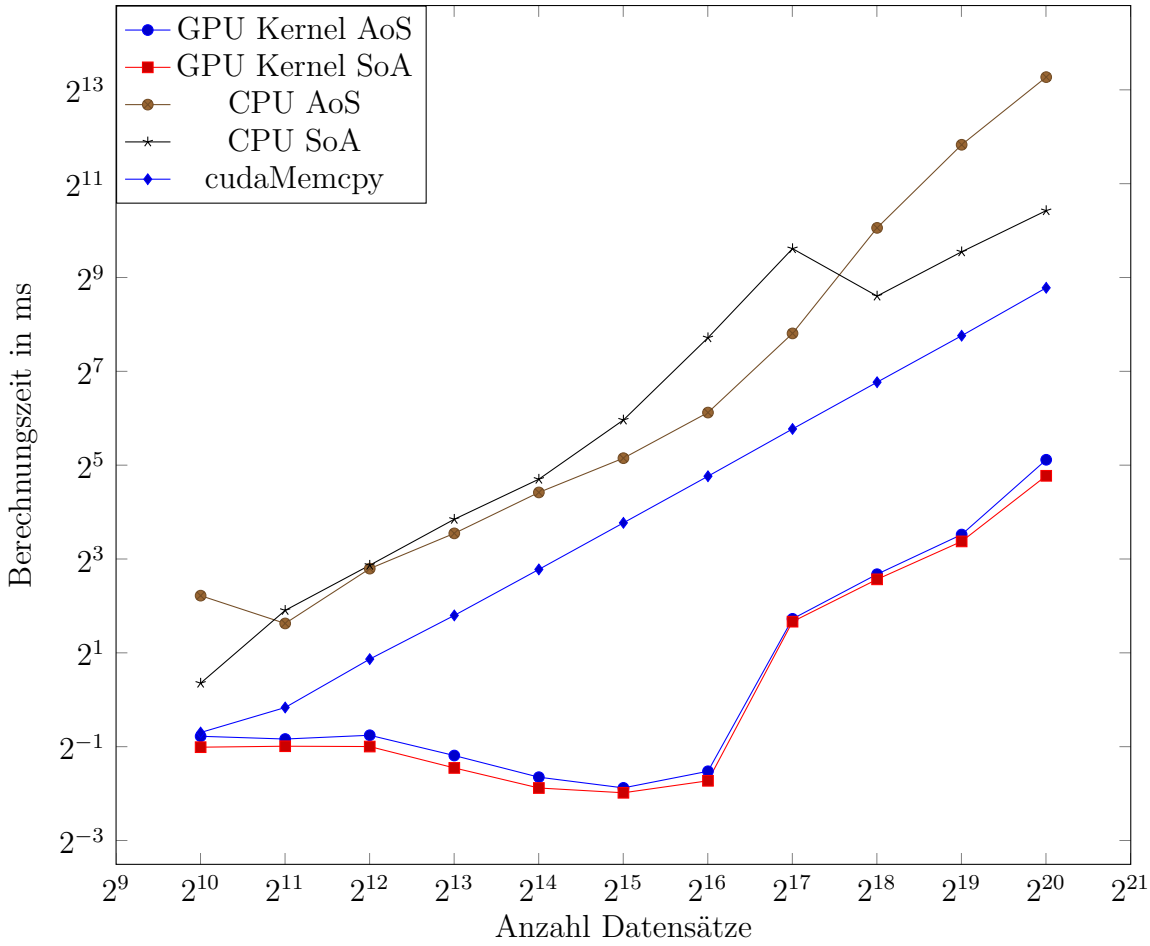
Zu beobachten ist auch, dass die SoA Variante auf der CPU erst ab 2^{18} Datensätzen besser performt als AoS. Diese Beobachtung deckt sich mit dem Benchmark auf gpu03, siehe Abbildung 4.

Die beiden Peaks bei 2^{10} Datensätzen auf dem Testsystem und bei 2^{17} auf gpu03 lassen sich durch Cache-Assoziativitäts-Effekte erklären. Normalerweise entscheidet die Speicheradresse eines Datums in welchen Teil des Caches es abgelegt wird. Bei den erwähnten Peaks scheinen die Arrays unter den CPU Threads so ungünstig aufgeteilt zu sein, dass die Daten der einzelnen Threads aufgrund ähnlicher Speicheradressen immer an die gleiche Stelle im Cache geschrieben werden. Der Cache wird somit künstlich verkleinert und es kommt vermehrt zu Cachemisses.



N	GPU Kernel AoS	GPU Kernel SoA	CPU AoS	CPU SoA	cudaMemcpy
1024	0,22	0,28	66,48	1,71	0,41
2048	0,22	0,29	0,86	5,75	0,76
4096	0,23	0,29	7,16	8,96	1,46
8192	0,21	0,25	10,27	16,39	2,81
16384	0,16	0,19	16,15	29,83	5,54
32768	0,17	0,2	28,69	51,62	11,57
65536	0,27	0,29	54,5	141,56	22,48
131072	1,72	2,33	244,25	435,92	44,91
262144	3,67	4,73	1484,54	882,46	89,81
524288	6,01	8,39	3831,04	1838,72	177,05
1048576	21,66	26,07	8920,1	2824,4	350,63

Abbildung 3: Benchmark CPU vs. GPU auf Testsystem



N	GPU Kernel AoS	GPU Kernel SoA	CPU AoS	CPU SoA	cudaMemcpy
1024	0,58	0,5	4,65	1,28	0,62
2048	0,56	0,5	3,09	3,75	0,89
4096	0,59	0,5	6,94	7,3	1,82
8192	0,44	0,37	11,68	14,38	3,48
16384	0,32	0,27	21,38	26	6,86
32768	0,27	0,25	35,47	62,34	13,64
65536	0,35	0,3	69,57	210,11	27,17
131072	3,3	3,17	223,98	783,7	54,62
262144	6,39	5,92	1064,83	388,98	108,95
524288	11,48	10,37	3634,88	747,86	216,51
1048576	34,64	27,34	9872,38	1373,8	439,33

Abbildung 4: Benchmark CPU vs. GPU auf gpu03

4 Fazit

Die durchgeführten Tests haben gezeigt, dass sich die von Rasterelektronenmikroskopen aufgenommenen Kraft-Abstands-Messungen performant auf modernen Grafikkarten verarbeiten lassen. Ein nicht zu vernachlässigender Overhead stellt jedoch der Kopiervorgang der Datensätze auf den Grafikkartenspeicher dar. Eine tatsächliche Implementierung auf einer Grafikkarte würde sich daher erst dann bezahlt machen, wenn mit den vorhandenen Daten noch mehr Berechnungen durchgeführt werden würden, als die hier beispielhaft implementierte dreifach lineare Regression. Oder aber wenn, wie in den Benchmarks gezeigt, die Anzahl der Datensätze statt der vorhandenen 2^{17} beträchtlich (d.h. gegen 2^{20} und mehr) gesteigert werden würde. Anderenfalls wird der zusätzlich benötigte Entwicklungsaufwand auf einer Grafikkarte nicht gerechtfertigt, da auch ältere CPUs mit 4 Rechenkernen 2^{20} Datensätze in weniger als 3 Sekunden bearbeitet haben.

Abbildungsverzeichnis

Listings

A Anhang

A.1 Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Seminararbeit mit dem Thema

   „Grafikkartenprogrammierung mit Cuda“

   ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Seminararbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Gera, 3. August 2017	.....
Ort, Datum	Unterschrift