



universität
wien

MASTERARBEIT

Titel der Masterarbeit

„Optimierung von Life Sciences Algorithmen für GPUs
mit CUDA/OpenCL“

Verfasser

David Dilch, BSc

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2013

Studienkennzahl lt. Studienblatt: A 066 940

Studienrichtung lt. Studienblatt: Masterstudium Scientific Computing UG2002

Betreuer: Ao. Univ.-Prof. Dipl.-Ing. Dr. Eduard Mehofer

Kurzfassung

Ein Forschungsschwerpunkt in Scientific Computing beschäftigt sich mit der Parallelisierung von Algorithmen für GPUs, da die theoretische maximale Leistung von GPUs ein Vielfaches größer ist als von CPUs. In der Masterarbeit wird zunächst auf die Nvidia-Fermi-GPU-Architektur und auf die Änderungen gegenüber der Vorgängerarchitektur GT200 näher eingegangen. Im nächsten Schritt werden die GPU-Programmiersprachen CUDA und OpenCL erklärt und die Unterschiede in der Programmierung gegenübergestellt. Das Ziel der Masterarbeit ist die Optimierung von zwei Life-Science-Algorithmen (Needleman-Wunsch/Smith-Waterman und Direct Coulomb Summation) für die Ausführung auf Grafikkarten (Nvidia-Fermi und Nvidia-GT200) bei Verwendung der Programmiersprachen CUDA und OpenCL. Der DCS-Algorithmus ist für Grafikkarten deutlich besser als der NW/SW-Algorithmus geeignet, da der Parallelisierungsgrad wesentlich größer ist. In der Masterarbeit wird ermittelt, wie wirksam einzelne Optimierungstechniken bei unterschiedlichen GPU-Architekturen (Fermi und GT200) und bei unterschiedlichen GPU-Programmiersprachen (CUDA und OpenCL) sind. Dabei wird überprüft, ob Optimierungstechniken, wie Memory Coalescing und Verwendung der kleinen on-chip-Speicher, durch die mit der Fermi-Architektur eingeführte Cache-Hierarchie über den Global Memory nicht mehr notwendig sind und dadurch eine einfachere Überführung von CPU-Code in GPU-Code ohne Einbußen bei der Performance möglich ist. Ebenfalls ein wichtiges Thema der Masterarbeit ist der faire Vergleich des GPU-Programms mit dem CPU-Programm beim Beantworten der Frage, ob bei beiden Algorithmen auf der GPU ein Speedup gegenüber der CPU erreicht werden kann. Für einen fairen Vergleich werden unter anderem bei der CPU mit der Programmiersprache OpenMP alle Kerne verwendet. In der Masterarbeit wird auch auf die limitierenden Faktoren eingegangen, die dazu führen, dass der Algorithmus nicht die Peak-Leistung der GPU erreicht.

Abstract

A research focus in scientific computing deals with the parallelisation of algorithms for GPUs because the theoretical maximum performance of GPUs is many times higher than of CPUs. The master thesis starts with the Nvidia Fermi GPU architecture and the changes compared to the predecessor architecture GT200. In the next step the GPU programming languages CUDA and OpenCL are explained and the differences in programming are compared. The aim of the master thesis is the optimization of two life science algorithms (Needleman-Wunsch/Smith-Waterman, Direct Coulomb Summation) for the execution on graphics cards (Nvidia-Fermi, Nvidia-GT200) by using the programming languages CUDA and OpenCL. The DCS algorithm is much more suitable for graphics cards than the NW/SW algorithm because the degree of parallelisation is significantly higher. In this master thesis, it is determined how effective the optimization techniques are on different GPU architectures (Fermi, GT200) and on different GPU programming languages (CUDA, OpenCL). In doing so, it is checked whether the new cache hierarchy for the global memory in the Fermi architecture replaces optimization techniques (memory coalescing, use of the small on chip memories) and therefore makes an easier transfer from CPU code to GPU code without losing performance possible. Another important issue of the master thesis is the fair comparison between GPU program and CPU program when answering the question whether the two algorithms can reach a speedup on the GPU in comparison with the CPU. For a fair comparison all cores on the CPU are used with the programming language OpenMP.

Inhaltsverzeichnis

KURZFASSUNG	1
ABSTRACT	2
INHALTSVERZEICHNIS	3
1 EINLEITUNG	9
1.1 GPU und CPU	9
1.2 GPU-Programmiersprachen	11
1.3 Ziele der Masterarbeit	11
2 GPU-ARCHITEKTUR (NVIDIA-GPUS).....	13
2.1 Multiprozessoren	13
2.1.1 Kerne	13
2.1.2 MAD- versus FMA-Floating-Point-Berechnungen	14
2.1.3 Single-Precision Peak GFLOP/s	14
2.1.4 Integer-Einheit.....	15
2.1.5 SFUs (Special Function Units)	15
2.1.6 Aktive Threads.....	16
2.1.7 Thread-Blöcke.....	16
2.1.8 Warps (=32 Threads) und Warp Scheduler	17
2.2 Unterschiedliche Speicher.....	17
2.2.1 Host Memory (=off chip memory).....	18
2.2.2 Device Memory (=off chip memory)	19
2.2.3 Global Memory (=off chip memory).....	19
2.2.3.1 CGMA.....	19
2.2.3.2 Memory Coalescing	19
2.2.4 Register (=on chip memory, Sichtbarkeit: Thread)	21
2.2.5 Local Memory (=off chip memory, Sichtbarkeit: Thread)	22
2.2.6 L2-Cache und L1-Caches	23
2.2.7 Constant Memory und Constant Memory Caches (=read-only off/on chip memory)..	23
2.2.8 Shared Memory (=on chip memory, Sichtbarkeit: Block)	24
2.2.8.1 Tiling	25
2.2.9 Latenzzeit, Bandbreite und Größe der einzelnen Speicher (Tesla C2050)	25
2.3 PCI-Anschluss.....	26
2.4 Architekturen	28
2.4.1 Compute Capability	28

2.4.2	Änderungen bei Fermi gegenüber der Vorgängerarchitektur GT200	30
2.4.2.1	40 Bit Unified Address Space statt drei GPU-Adressräume	30
2.4.2.2	Unified Virtual Addressing (64 Bit gemeinsamer virtueller CPU- und GPU-Speicheradressraum).....	30
2.4.2.3	L1-Caches und L2-Cache	31
2.4.2.4	Multiprozessoren und Kerne	32
2.4.2.5	SFUs	32
2.4.2.6	Warp Scheduler	32
2.4.2.7	Aktive Threads	32
2.4.2.8	Register Memory	32
2.4.2.9	Shared Memory	33
2.4.2.10	Rekursion / Funktionsaufrufe innerhalb eines Kernel.....	33
2.4.2.11	printf() im Kernel	34
2.4.2.12	malloc() und free() im Kernel	34
2.4.2.13	Schnelleres Application Context Switching	34
2.4.2.14	Gleichzeitiges Ausführen mehrerer Kernels auf einer GPU	36
2.4.2.15	Deutliche Erhöhung der Double-Precision-Peak-Leistung	39
2.4.2.16	Verbesserung der Genauigkeit bei Floating-Point.....	39
2.4.2.17	Ausführen von 32-Bit-Integer-Multiplikationen in einem Taktzyklus	43
2.4.2.18	ECC bei Tesla-GPUs.....	44
2.4.2.19	Debugger CUDA-GDB und CUDA-memcheck	44
2.4.2.20	Memory Coalescing / Global-Memory-Zugriff	48
2.4.2.21	Constant Cache.....	48
2.4.2.22	Concurrent Data Transfers	48
2.4.2.23	C++ Unterstützung.....	48
3	GPU-PROGRAMMIERPLATTFORMEN	48
3.1	CUDA	49
3.2	OpenCL	49
3.3	Programmierung mit CUDA und OpenCL	50
3.3.1	Include-Statements	50
3.3.2	Initiierung	50
3.3.3	API-Funktionen für den Host für das Anlegen und Freigeben von Speicher im Arbeitsspeicher am Host	52
3.3.4	API-Funktionen für den Host zum Anlegen und Freigeben von Speicher im Global Memory am Device	53

3.3.5	API-Funktion(en) für den Host für das Transferieren von Daten vom Host zum Device und zurück	54
3.3.6	Kernel (=paralleler Code ausgeführt auf Device)	54
3.3.6.1	Aufruf des Kernels	55
3.3.6.2	Kerneldeklaration	55
3.3.6.3	Thread-ID-Bestimmung im Kernel	55
3.3.6.4	Concurrent Kernels (Streams und Queues)	56
3.3.6.5	Ausführung von Kernels auf mehreren Devices	57
3.3.7	Begriffe zur Beschreibung der Device-Architektur	57
3.3.8	Shared Memory in CUDA ist Local Memory in OpenCL	58
3.3.9	Constant Memory	58
3.3.10	Synchronisationsfunktion nach Kernelaufruf	59
3.3.11	Synchronisationsfunktion im Kernel	59
3.3.12	Mathematische Funktionen im Kernel	59
3.3.12.1	Standard-Funktionen	59
3.3.12.2	Intrinsic/Native-Funktionen	61
3.3.13	Double Precision	62
4	BENÜTZTE HARDWARE UND COMPILER	63
4.1	Nvidia-GTX-Konfiguration	63
4.2	Nvidia-Tesla-Konfiguration	65
5	LAUFZEITMESSUNG DES PROGRAMMCODES	66
5.1	Messmethode	67
5.2	CPU-Programm	68
5.2.1	CPU-Gesamtlaufzeit	68
5.2.2	CPU-Funktionslaufzeit	68
5.3	GPU-Programm	69
5.3.1	GPU-Gesamtlaufzeit	70
5.3.2	GPU-Kernellaufzeit	70
5.4	Gesamtspeedup und Kernelspeedup für die Speedup-Diagramme	71
6	ÜBERPRÜFUNG DER KORREKTHEIT	71
6.1	NW-Algorithmus (Short- und Int-Datentyp)	71
6.2	DCS-Algorithmus (Float- und Double-Datentyp)	71
7	ALGORITHMUS NW/SW (=NEEDLEMAN-WUNSCH/SMITH-WATERMAN)	76
7.1	Einleitung	76
7.2	CPU-Programm mit Laufzeitmessungen	82

7.2.1	C.....	82
7.2.1.1	Schritt 1 - Initialisierung der Startmatrix.....	82
7.2.1.2	Schritt 2 - Berechnung der Ergebnismatrix	82
7.2.1.3	Schritt 3 - Traceback auf die Ergebnismatrix.....	84
7.2.1.4	Unterschiedliche Compiler und Compiler-Flags	85
7.2.1.5	Laufzeit der drei Schritte	86
7.2.1.6	GCUPS.....	87
7.2.2	OpenMP.....	87
7.3	GPU-Programm mit Laufzeitmessungen	90
7.3.1	Global-Memory- und Shared-Memory-Programm	90
7.3.2	Optimierungen am Global-Memory-Programm.....	93
7.3.2.1	Finden der optimalen Blockgröße	93
7.3.2.2	Verwendung des Short Datentyps.....	93
7.3.2.3	Memory Coalescing	94
7.3.3	Optimierungen am Shared-Memory-Programm.....	95
7.3.3.1	Wahl der optimalen Blockgröße.....	96
7.3.3.2	Verwendung von Short (2 Bytes) anstatt von Int (4 Bytes).....	97
7.3.3.3	Nur eine Inputmatrix und eine shared Teilmatrix.....	98
7.3.3.4	Initialisierung der Inputmatrix auf GPU anstatt Übertragung.....	98
7.3.3.5	Nicht erfolgreich: Pinned Memory statt Pageable Memory	101
7.3.4	Zusammenfassung der Gesamtspeedups aller Optimierungen in einem Diagramm ..	102
7.4	Diskussion der gemessenen GPU-Laufzeitdaten.....	107
7.4.1	Auswirkung der Optimierungen auf das CUDA- und OpenCL-Programm	107
7.4.1.1	Global-Memory-Programm	107
7.4.1.2	Shared-Memory-Programm	109
7.4.2	Fermi im Vergleich zu der Vorgängerarchitektur GT200.....	119
7.4.3	Wie stark konnte der Ausgangscode verbessert werden?	121
7.4.4	Wie viel wurde von der maximalen Leistung der GPU erreicht?	121
8	ALGORITHMUS DCS (=DIRECT COULOMB SUMMATION).....	122
8.1	Einleitung.....	122
8.2	CPU-Programm mit Laufzeitmessungen	123
8.2.1	C.....	123
8.2.1.1	Drei Codevarianten und deren Laufzeit	123
8.2.1.2	Unterschiedliche Compiler und Compiler-Flags.....	126

8.2.1.3	Atom-Evaluierungen pro Sekunde	127
8.2.2	OpenMP.....	127
8.3	GPU-Programm mit Laufzeitmessungen	128
8.3.1	Global-Memory- und Const-Memory-Programm	128
8.3.2	Optimierungen	135
8.3.2.1	Verwendung von rsqrtf	135
8.3.2.2	Atoms-Array als float4-Variable bzw. lokale float4-Variable	138
8.3.2.3	Vorberechnung von $dz*dz$ auf CPU	139
8.3.2.4	Berechnung mehrerer Gridpunkte in X-Richtung pro Thread.....	141
8.3.2.5	Memory Coalescing	143
8.3.2.6	Verwendung von Compiler-Optimierungsflags bei OpenCL.....	144
8.3.2.7	native_rsqrt() bei OpenCL	145
8.3.3	Zusammenfassung der Gesamtspeedups aller Optimierungen in einem Diagramm .	146
8.4	Diskussion der gemessenen GPU-Laufzeitdaten.....	151
8.4.1	Auswirkung der Optimierungen auf das CUDA- und OpenCL-Programm	151
8.4.2	Fermi im Vergleich zu der Vorgänger-Architektur GT200	155
8.4.3	Vergleich Code ohne Optimierung mit Code mit Optimierungen.....	158
8.4.4	Wie viel wurde von der maximalen Leistung der GPU erreicht?	158
8.4.5	Performance bei Double-Precision (double).....	159
9	INTERPRETATION DER GPU-LAUFZEITMESSUNGEN	160
9.1	Kann Constant Memory durch das Caching von Global Memory ersetzt werden?	160
9.2	Kann Shared Memory durch das Caching von Global Memory ersetzt werden?	160
9.3	Kann Caching von Global Memory die Optimierung Memory Coalescing überflüssig machen?	161
9.4	Auswirkung des vergrößerten Shared Memory	162
9.5	Auswirkung der höheren Double-Precision-Peak-Leistung.....	162
9.6	Ist der Algorithmus durch die Bandbreite des Global Memory limitiert?	163
9.7	Kann bei beiden Algorithmen auf der GPU ein Speedup gegenüber der CPU erreicht werden? Wie hoch ist der Speedup?	167
9.8	Wie sieht der Performance-Unterschied der Programmierplattformen CUDA und OpenCL aus?	167
9.9	Wie sieht der Performance-Unterschied zu älteren GPU-Architekturen aus? Skaliert der GPU-Code automatisch von alter Architektur zu neuer Architektur bzw. welche Änderungen müssen vorgenommen werden?.....	168
9.10	Welche Optimierungen müssen auf einen GPU-Code angewendet werden, um einen schnelleren Code zu erzeugen? Welche davon sind Algorithmus-abhängig, welche allgemein nutzvoll?	170

10	GPU-OPTIMIERUNGSSTRATEGIEN	172
10.1	Optimierung bei der Auslastung / Parallelität.....	172
10.2	Optimierung bei den Datenzugriffen / Bandbreitenverbrauch / Latenzzeit.....	173
10.2.1	Memory Coalescing	173
10.2.2	Tiling	174
10.2.3	Verwendung der kleineren Speicher (Constant Memory und Shared Memory)	174
10.3	Optimierung bei den Instruktionen / Berechnungseffizienz.....	174
10.3.1	Verwendung von Intrinsic/Native-Funktionen und Compilerflags.....	174
10.3.2	Reduzierung der Operationen und Verwendung von schnelleren Operationen	174
10.3.3	Float- statt Double-Datentyp	174
10.4	Optimierungen an der Datenübertragung zwischen CPU-Arbeitsspeicher und GPU-Arbeitsspeicher.....	174
10.4.1	Generierung der Inputmatrix durch Inputwerte auf der GPU	174
10.4.2	Verwendung kleinerer Datentypen.....	174
11	AUSBLICK.....	175
11.1	OpenCL und AMD-GPUs	175
11.2	CUDA-Toolkit 4.1, 4.2 und 5.0	176
11.3	Nvidia Visual-Profiler	176
11.4	GTX 580	178
11.5	Compute Capability 2.1 (GTX 460)	178
11.6	Neue GPU-Architektur Kepler (Compute Capability 3.0 und 3.5)	178
11.6.1	Compute Capability 3.0 (GK104)	178
11.6.2	Compute Capability 3.5 (GK110)	179
11.6.3	Compute Capability 3.0 versus 3.5 und Vergleich mit Fermi	181
11.7	GPU als Teil der CPU, APU (=Accelerated Processing Unit)	183
12	ZUSAMMENFASSUNG.....	184
	LITERATURVERZEICHNIS	185
	ANHANG A PROGRAMMCODES	193
A.1	Needleman-Wunsch CPU-Code Schritt 3 (traceback).....	193
A.2	Needleman-Wunsch GPU-Code (Shared-Memory-Programm) mit allen Optimierungen	195
A.3	DCS GPU-Code.....	199
	ANHANG B ABSTRACT	202
	ANHANG C ABSTRACT IN DEUTSCH / KURZFASSUNG	202
	ANHANG D DEUTSCHE ZUSAMMENFASSUNG.....	203
	ANHANG E LEBENS LAUF	204

1 Einleitung

1.1 GPU und CPU

Bis zum Jahr 2003 sind Anwendungen durch das Erhöhen der Taktfrequenz mit jeder neuen Generation von CPU-Prozessoren mit wenig bis gar keinen Aufwand schneller geworden [1]. Da jedoch jede weitere Erhöhung der Taktfrequenz den Energieverbrauch und die Temperatur auf den Chips stark ansteigen lässt und das Abführen der Wärme kaum noch möglich war, ist man übergegangen, mehrere Kerne bereitzustellen [1]. Damit alle Kerne des Chips ausgenutzt werden und somit im Vergleich zur Vorgänger-Generation eine Performance-Steigerung der Anwendung erreicht werden kann, muss die Software parallelisiert werden.

Da durch die immer höher werdende Anzahl von Kernen bei den CPUs der parallele Code immer wichtiger wird, ist es hilfreich, sich mit der Programmierung von GPUs, die aus sehr vielen Kernen bestehen, auseinanderzusetzen. Die aktuelle Entwicklung bei Intel- und AMD-CPUs ist, dass GPU-Kerne auf dem Chip integriert werden [2] [3]. AMD bezeichnet diese neuen Hybrid-Chips als APUs (=Accelerated Processing Unit) [2]. Intel inkludiert ab Sandy-Bridge GPU-Kerne in die CPU [3]. Die APUs sind nicht Teil der Masterarbeit. In der Masterarbeit geht es um Grafikkarten, auf denen sich eine GPU und ein eigener Arbeitsspeicher befinden. Die Grafikkarte ist über den PCI-Bus mit der CPU und dem CPU-Arbeitsspeicher verbunden.

Damit in Zukunft von Hardwaregeneration zu Hardwaregeneration ein deutliches Performance-Wachstum beim Ausführen einer Anwendung erreicht werden kann, ist es wichtig die Programme massiv zu parallelisieren [1]. Wenn bei sequentieller Ausführung des Programms 30% der Gesamtlaufzeit der parallelisierbare Code läuft und der parallelisierbare Code bei paralleler Ausführung 100 Mal schneller als bei sequentieller Ausführung ist, dann wird das gesamte Programm bei paralleler Ausführung laut dem Gesetz von Amdahl (siehe Formel 1) nur 1,42 Mal ($=1/(0,7+(0,3/100))$) schneller [1]. Wichtig ist somit, dass der Anteil des parallelisierbaren Codes an der Gesamtlaufzeit groß ist [1]. Wenn der parallelisierbare Code einen Anteil von 99% an der Gesamtlaufzeit bei der sequentiellen Ausführung hat, dann erreicht man mit einem Speedup von 100 beim parallelen Ausführen des parallelisierbaren Codes laut dem Gesetz von Amdahl, dass das gesamte Programm bei der parallelen Ausführung 50 Mal ($=1/(0,01+(0,99/100))$) schneller als bei der sequentiellen Ausführung läuft [1].

Formel 1: Gesetz von Amdahl [1] (Variablennamen adaptiert)

Speedup = $1/(\text{Sequentieller_Anteil} + (\text{Paralleler_Anteil}/\text{Speedup_des_parallelen_Anteils}))$

Meistens vergrößert sich mit dem Anstieg der Problemgröße eines Algorithmus automatisch der parallele Anteil an der Gesamtlaufzeit. Bei kleinen Problemgrößen nimmt der sequentielle Code einen großen Anteil an der gesamten Laufzeit ein und somit ist der mögliche Speedup an der Gesamtlaufzeit gering, auch wenn man viel Speedup beim parallelen Code erreicht.

Im Bereich Scientific Computing beschäftigt man sich aktuell mit der Parallelisierung von Algorithmen für GPUs, da die theoretische maximale Leistung von GPUs ein Vielfaches größer ist als von CPUs. Ein Beispiel dafür ist der momentan schnellste Supercomputer (Stand: November 2012) der Welt „Titan“ [4], dessen hohe Rechenleistung auf GPUs basiert. Mit der Einführung der Programmierplattformen CUDA und OpenCL wurde die Programmierung von GPUs für wissenschaftliche Anwendungen stark vereinfacht.

GPU-Computing unterscheidet sich in fünf wichtigen Punkten von CPU-Computing.

Der erste Punkt ist, dass GPUs mehr Kerne besitzen, dafür aber weniger Cache zur Verfügung haben. Auf CPUs befinden sich ein paar Kerne (=multi-core), eine GPU hingegen besitzt sehr viele Kerne (=many-core). Der Grund ist, dass GPUs auf parallele Anwendungen und nicht auf sequentielle Anwendungen optimiert sind [1]. Damit kann ein massiv Daten-paralleler Code mit hoher arithmetischer Dichte auf einer GPU deutlich schneller als auf einer CPU ausgeführt werden [1].

Der zweite Unterschied ist, dass GPU-Kerne zu Gruppen (=Multiprozessoren mit SIMT(=Single-Instruction, Multiple-Thread)-Architektur) zusammengefasst werden und alle Kerne eines Multiprozessors zu einem Zeitpunkt nur dieselbe Instruktion bzw. ab der Nvidia-Fermi-Architektur nur wenige Instruktionen im Vergleich zur Anzahl der Kerne ausführen können [5].

Der dritte Unterschied ist, dass das Programm nicht komplett auf der GPU ausgeführt werden kann. Die GPU benötigt immer zusätzlich als „Host“ eine CPU für das Ausführen eines CUDA- oder OpenCL-Programms. Nur die Kernels werden auf der GPU ausgeführt.

Der vierte Unterschied ist, dass beim Ausführen eines GPU-Programms nicht nur der GPU-DRAM verwendet wird, sondern auch der CPU-DRAM. Mit dem Start des Programms befinden sich die Inputdaten im Arbeitsspeicher der CPU und müssen für einen schnelleren Zugriff zusätzlich auf den GPU-DRAM übertragen werden. Nach dem Ausführen des Kernels werden durch Aufruf einer API-Funktion im Programmcode die Ergebnisse im Arbeitsspeicher der GPU in den Arbeitsspeicher der CPU übertragen.

Der fünfte Unterschied ist, dass GPUs eine deutlich kleinere Kontrolllogik im Vergleich zu CPUs besitzen [6]. Die kleine Kontrolllogik bei GPUs ist möglich, da die Instruktionen in-order ausgeführt werden und man sehr viele Threads hat, die man abwechseln kann, sobald Threads bei Datenzugriffen mit großer Latenzzeit warten müssen [1]. Bei CPUs benötigt man die große Control Logic, um Instruktionen eines Threads parallel ausführen zu können oder die Reihenfolge der Instruktionen so zu verändern (=out-of-order), dass die Wartezeiten reduziert werden können [1]. Dabei wird sichergestellt, dass sich das Ergebnis durch die Optimierungen nicht verändert [1].

Das Design von GPUs ermöglicht deutlich mehr Floating-Point-Operationen pro Sekunde (bei Double-Precision erst seit der Fermi-Architektur [7]) und eine höhere Bandbreite zwischen Chip und Arbeitsspeicher. Die höhere FLOP/s-Leistung wird durch die höhere Anzahl von Kernen ermöglicht. Die höhere Bandbreite kann durch einfachere Speichermodelle und weniger Altlasten erreicht werden [1]. In der Praxis stellt man jedoch sehr schnell fest, dass diese beiden Vorteile nur dann etwas bringen, wenn das Programm ausreichend und effizient parallelisiert wird. Sonst ist die GPU trotz der beiden höheren Maximalwerte langsamer als die CPU. Damit macht nur das Ausführen der rechenintensiven, parallelisierbaren Teile des Programms auf der GPU einen Sinn [1]. Die sequentiellen, schwer parallelisierbaren Teile müssen weiterhin auf der CPU ausgeführt werden [1].

Seit der Nvidia GPU-Architektur G80 wird der IEEE(=Institute of Electrical and Electronics Engineers) Floating-Point-Standard 754 unterstützt, wodurch GPUs nun auch bei numerischen Berechnungen eine Alternative zu CPUs sind [1]. Der Standard, der 1985 veröffentlicht wurde, ermöglicht es, dass die Resultate immer dieselben sind, egal auf welchen Prozessor die Berechnung ausgeführt wird [1]. Seit 2008 gibt es mit IEEE 754-2008 eine neuere Version des Standards [1]. Dieser aktualisierte Standard ist bereits in der Nvidia Fermi-Architektur implementiert [7]. Seit der Fermi-Vorgängerarchitektur GT200 wird Double-Precision unterstützt [7], wodurch nun numerische Berechnungen vollständig auf die GPU verlagert werden können. Mit Fermi wurde die Peak-Leistung der GPUs bei Double-Precision deutlich erhöht und der Faktor zwischen Peak-Single-Precision-Leistung und Peak-Double-Precision-Leistung wie bei CPUs auf den Faktor 2 gesenkt [7] [8].

1.2 GPU-Programmiersprachen

Von 2003 bis 2006 musste man OpenGL oder Direct3D verwenden, um Programmteile auf der GPU ausführen zu können [1]. Ab 2007 mit der Veröffentlichung von CUDA wurde die Programmierung deutlich einfacher und für rechenintensive, parallele Anwendungen interessant [1]. CUDA wurde von Nvidia vorangetrieben und der Code kann nur auf Nvidia-Grafikkarten ausgeführt werden [5]. Nach CUDA wurde OpenCL veröffentlicht [1]. Im Unterschied zu CUDA kann OpenCL-Code nicht nur auf Nvidia-Grafikkarten ausgeführt werden, sondern auch auf AMD-GPUs, Intel-CPU und AMD-CPU [1] [9] [10]. OpenCL wurde von Apple vorangetrieben und ist ein offener Standard [1].

Nvidia-GPU-CUDA unterstützt (Stand: 2011, Nvidia-CUDA-Toolkit 4.0) CUDA C, CUDA C++, OpenCL, DirectCompute, CUDA Fortran, CUDA Java und CUDA Python [5]. Bei CUDA wird in der Masterarbeit Nvidia CUDA C verwendet. Nvidia CUDA C ist eine Erweiterung der Programmiersprache C [5]. Bei OpenCL wird in der Masterarbeit Nvidia OpenCL verwendet. Nvidia OpenCL ist OpenCL C und basiert somit auf der Programmiersprache C [1] [6].

1.3 Ziele der Masterarbeit

In der Masterarbeit wird zunächst auf die Nvidia-GPU-Architektur näher eingegangen. Dabei erfolgt eine Beschreibung der Multiprozessoren (z. B. Kerne, MAD- versus FMA-Berechnungen, aktive Threads, Thread-Blöcke und Warps), der unterschiedlichen Speicher (z. B. Global Memory, Shared Memory, Constant Memory und Local Memory) und der Änderungen bei der Fermi-Architektur gegenüber der Vorgängerarchitektur GT200 (z. B. Cache-Hierarchie, Unified Virtual Addressing, printf() im Kernel, malloc() im Kernel, Debugger, 48 KB Shared Memory und gleichzeitiges Ausführen mehrerer Kernels auf einer GPU).

Im nächsten Schritt werden die GPU-Programmiersprachen CUDA und OpenCL erklärt und dabei die Unterschiede bei der Programmierung (z. B. Initiierung, API-Funktionen für Anlegen und Freigeben von Arrays, Kernelaufbau, Kernel, Speicherbezeichnungen und mathematische Funktionen) gegenübergestellt.

Das Ziel der Masterarbeit ist die Optimierung von zwei Life-Science-Algorithmen (Needleman-Wunsch/Smith-Waterman und Direct Coulomb Summation) für die Ausführung auf Grafikkarten (Nvidia-Fermi und Nvidia-GT200) bei Verwendung der Programmiersprachen CUDA und OpenCL. Es wird ermittelt, wie wirksam einzelne Optimierungstechniken (z. B. Memory Coalescing und Verwendung der kleinen on-chip-Speicher) bei unterschiedlichen GPU-Architekturen (Fermi und GT200) und bei unterschiedlichen GPU-Programmiersprachen (CUDA und OpenCL) sind.

Der Needleman-Wunsch- bzw. Smith-Waterman-Algorithmus dient zur Suche nach Ähnlichkeiten zweier Nukleotidsequenzen oder Proteinsequenzen. Der Needleman-Wunsch- bzw. Smith-Waterman-Algorithmus ist etwas schwerer für GPUs zu optimieren. Es kann nur Wavefront für Wavefront sequentiell berechnet werden, da eine Wavefront jeweils von den davor berechneten Wavefronts abhängig ist. Innerhalb der Wavefront kann parallel gerechnet werden. Da der Programmunterschied zwischen Needleman-Wunsch und Smith-Waterman aus einer if-Bedingung auf dem Ergebniswert besteht, wirken sich Optimierungen für den Needleman-Wunsch-Algorithmus gleich positiv wie für den Smith-Waterman-Algorithmus aus, und es können somit immer beide Algorithmen betrachtet werden. In der Masterarbeit soll die Variante optimiert werden, in der es um den Vergleich zweier Sequenzen geht. Die Variante, wo eine kleine Sequenz mit einer Datenbank von kleinen Sequenzen verglichen wird, ist trivial parallelisierbar und nicht Teil der Masterarbeit.

Ein für die GPU deutlich besser geeigneter Algorithmus ist der Direct-Coulomb-Summation-Algorithmus, da der Parallelisierungsgrad wesentlich größer als bei Needleman-Wunsch/Smith-Waterman ist. Der Direct-Coulomb-Summation-Algorithmus dient zum Berechnen von

elektrostatischen Potential Maps (zum Beispiel Voraussetzung für Ion Placement). Um ein Molekül wird ein Gitternetz gelegt. Nun kann das elektrostatische Potential jedes Gitterpunkts unabhängig von den anderen Gitterpunkten berechnet werden. Jeder Gitterpunkt muss in der Berechnung die Entfernung zu allen Atomen des Moleküls und die Ladung jedes Atoms berücksichtigen.

Als Grundversion der beiden Algorithmen wird zunächst eine Global-Memory-Version erstellt, die den Arbeitsspeicher der GPU benützt. Danach wird versucht die schnelleren, jedoch deutlich kleineren Speicher auf der Grafikkarte auszunützen, wie dem Shared Memory und den Const Memory. Im nächsten Schritt werden weitere Optimierungen ausprobiert.

Beim DCS-Algorithmus stammen die Ideen zu Optimierungen und die Codes fast vollständig von [1]. Beim NW-Algorithmus wurden auf Basis des Shared-Memory-Programms von [11] (Nvidia-Seite mit Verlinkung zu Paper [12] und Code [13] des Shared-Memory-Programms) alle Optimierungen selbst entwickelt und auch die Optimierungen für das Global-Memory-Programm wurden selbst erarbeitet.

Folgende Fragen sollen mit der Masterarbeit beantwortet werden:

- Kann bei beiden Algorithmen auf der GPU ein Speedup gegenüber der CPU erreicht werden? Wie hoch ist der Speedup?
- Wie sieht der Performance-Unterschied der Programmierplattformen CUDA und OpenCL aus?
- Wie sieht der Performance-Unterschied zu älteren GPU-Architekturen aus? Skaliert der GPU-Code automatisch von alter Architektur zu neuer Architektur bzw. welche Änderungen müssen vorgenommen werden?
- Welche Optimierungen müssen auf einen GPU-Code angewendet werden, um einen schnelleren Code zu erzeugen? Welche davon sind Algorithmus-abhängig, welche allgemein nutzvoll?

In der Masterarbeit wird auch ermittelt, ob Optimierungstechniken, wie Memory Coalescing und Verwendung von Shared Memory und Constant Memory, durch die mit der Fermi-Architektur eingeführte Cache-Hierarchie über den Global-Memory nicht mehr notwendig sind und dadurch eine einfachere Überführung von CPU-Code in GPU-Code ohne Einbußen bei der Performance möglich ist. Vor der Fermi-Architektur wurde der Shared Memory als Cache-Ersatz verwendet. Für die Zwischenspeicherung von Werten aus dem Arbeitsspeicher der GPU in dem auf dem Chip befindlichen Shared Memory müssen Befehle im Code eingefügt werden. Mit Fermi wurde der Shared Memory pro Multiprozessor um den Faktor 3 vergrößert und zusätzlich wurde eine Cache-Hierarchie (L1-Caches und L2-Cache), die wie bei CPUs ohne Programmieraufwand funktioniert, eingeführt [7]. Die Cache-Hierarchie tritt somit mit den anderen auf der GPU befindlichen Speichern in Konkurrenz.

Ein weiteres Thema der Masterarbeit ist der faire Vergleich des GPU-Programms mit dem CPU-Programm bei der Frage, ob bei beiden Algorithmen auf der GPU ein Speedup gegenüber der CPU erreicht werden kann. Für den fairen Vergleich werden auch beim CPU-Programm Optimierungen durchgeführt. Es wird auf die Wahl des Compilers/der Compilerflags geachtet und es werden mit der Programmiersprache OpenMP alle Kerne der CPU verwendet. Ein weiterer Schritt in Richtung fairer Vergleich wurde getan, indem nicht nur die Kernellaufzeit ermittelt wurde, sondern auch eine Gesamtlaufzeit, in der die Kernellaufzeit, die Laufzeit für das Anlegen/Freigeben der Arrays im CPU-Arbeitsspeicher und GPU-Arbeitsspeicher und die Laufzeit für die Übertragungen zwischen CPU-Arbeitsspeicher und GPU-Arbeitsspeicher inkludiert wurde.

In der Masterarbeit wird für beide Algorithmen berechnet, wie viel von der maximalen Leistung der GPU erreicht wurde. Es wird auch auf die limitierenden Faktoren (z. B. Synchronisationspunkte, große Datenübertragungen/Bandbreite und keine FMA/MAD-Operationen) eingegangen, die dazu führen, dass der Algorithmus nicht die Peak-Leistung der GPU erreicht. Für beide Algorithmen wird theoretisch ermittelt, ob die Bandbreite des Global Memory ein limitierender Faktor ist. Dabei wird

der CGMA(=Compute to Global Memory Access)-Wert, der das Verhältnis zwischen der Anzahl der Operationen und der Anzahl der Global-Memory-Zugriffe ausdrückt, verwendet und mit der Formel „Bandbreite des Global Memory/Datentypgröße*CGMA“ wird der GFLOP/s-Wert, ab dem die maximale Bandbreite des Global Memory erreicht wird, errechnet [1].

In der Masterarbeit wird auch darauf eingegangen, welche Auswirkung die höhere Double-Precision-Peak-Leistung und der vergrößerte Shared Memory bei Fermi im Vergleich zu GT200 auf die Programme haben.

Die Masterarbeit ist in folgende Kapitel unterteilt. Im Kapitel 2 wird die Architektur von Nvidia-GPUs erklärt. Im Kapitel 3 wird auf die Programmierplattformen CUDA und OpenCL näher eingegangen. Die benutzte Umgebung (Hardware, Compiler und Betriebssystem) für die Laufzeitmessungen wird im Kapitel 4 beschrieben. Kapitel 5 beschäftigt sich mit der verwendeten Laufzeitmessung für die Programme und der Verwendung der Werte in den Speedup-Diagrammen. Auf die Überprüfung der Korrektheit der Ergebnisse des GPU-Programms und des optimierten CPU-Programms wird im Kapitel 6 eingegangen. Die zwei weiteren Kapitel erklären die zwei Algorithmen und die durchgeführten Optimierungen. Zuerst wird im Kapitel 7 auf den Needleman-Wunsch-Algorithmus und im nachfolgenden Kapitel 8 auf den Direct-Coulomb-Summation-Algorithmus eingegangen. Im Kapitel 9 erfolgt eine Interpretation der GPU-Laufzeitmessungen und es werden die in der Einleitung gestellten Fragen beantwortet. Schließlich werden im Kapitel 10 Optimierungsstrategien definiert, die beim Programmieren von GPUs beachtet werden sollten, um das Programm näher an die Peak-Leistung der GPU heranführen zu können. Im Kapitel 11 gibt es einen Ausblick auf aktuelle und kommende Technologien und im Kapitel 12 erfolgt die Zusammenfassung.

Im Anhang befinden sich Programmcodes, auf die in der Masterarbeit verwiesen wird.

Die Ergebnisse der Masterarbeit wurden in einem Paper [14] veröffentlicht.

2 GPU-Architektur (Nvidia-GPUs)

2.1 Multiprozessoren

Eine GPU ist unterteilt in Multiprozessoren, auf denen jeweils sehr viele Threads parallel abgearbeitet werden [1]. Diese Multiprozessoren werden auch als SIMT(=Single-Instruction, Multiple-Thread)-Prozessoren, SMs (=Streaming Multiprocessors) oder CUs (=Compute Units) bezeichnet [1].

2.1.1 Kerne

Jeder Multiprozessor besteht aus Kernen [1]. Die Kerne werden auch als SPs (=Streaming Processors) oder PEs (=Processing Elements) bezeichnet [1]. Alle Kerne eines Multiprozessors teilen sich dieselbe Control Logic und denselben Instruction Cache, wodurch sie nicht wie bei CPU-Kerne unabhängig voneinander arbeiten und unterschiedliche Instruktionen ausführen können [1]. Jeder Kern führt leicht tausende Threads pro Anwendung aus [1].

Die Anzahl der Kerne pro GPU wurde mit jeder neuen Nvidia-GPU-Architektur in etwa verdoppelt. Die Nvidia G80-GPU besteht aus 128 Kernen (=16 Multiprozessoren*8 Kerne) [1], die GT200-GPU aus 240 Kernen (=30 Multiprozessoren*8 Kerne) [1] und die Fermi-GPU aus 512 Kernen (=32 Kerne * 16 Multiprozessoren) [15]. Die Anzahl der Kerne pro Multiprozessor wurde mit Fermi von 8 auf 32 erhöht [15].

2.1.2 MAD- versus FMA-Floating-Point-Berechnungen

Bei Single-Precision Floating-Point (=Datentyp float, 32-Bit) kann ein Kern pro Takt eine Multiplikation oder eine Addition oder eine „ $a*b+c$ “-Operation ausführen [15]. Die „ $a*b+c$ “-Operation wird als MAD(=Multiply-Add)- oder FMA(=Fused Multiply-Add)-Operation bezeichnet, je nachdem ob schon beim Zwischenergebnis ($a*b$) gerundet wird [15]. Wenn beim Zwischenergebnis ($a*b$) nicht gerundet wird, handelt es sich um eine FMA-Operation, sonst um eine MAD-Operation [15].

Im CUDA-Kernel-Code kann mit der Intrinsic-Funktion `__fmmaf_rn()` für Single-Precision Floating-Point (=Datentyp float, 32-Bit) und der Intrinsic-Funktion `__fma_rn()` für Double-Precision Floating-Point (=Datentyp double, 64-Bit) festgelegt werden, dass „ $a*b+c$ “-Operationen als FMA-Operationen ausgeführt werden [5]. Bei Single-Precision wird die Intrinsic-Funktion bei den Architekturen ab Fermi zu 1 Instruktion kompiliert, jedoch bei den Architekturen vor Fermi werden deutlich mehr Instruktionen für die Realisierung der Intrinsic-Funktion benötigt [5]. Im Gegensatz zu der Single-Precision Intrinsic-Funktion führt die Double-Precision Intrinsic-Funktion unabhängig von der Architektur immer zu einer 1 Instruktion. Der Grund ist, dass seit der Einführung von Double-Precision mit der GT200-Architektur die Hardware FMA-Operationen ausführt (siehe Tabelle 1) [15]. Wird die Intrinsic-Funktion für FMA-Operationen nicht verwendet, entscheidet der Compiler, ob Multiplikationen und Additionen in „ $a*b+c$ “-Operationen zusammengefasst werden [5]. Wenn man verhindern möchte, dass der Compiler „ $a*b+c$ “ Operationen bildet, müssen im CUDA-Kernel-Code die Intrinsic-Funktionen `__fadd_rn()` und `__fmul_rn()` bei Single-Precision und `__dadd_rn()` und `__dmul_rn()` bei Double-Precision verwendet werden [5]. Werden die Intrinsic-Funktionen nicht verwendet, wird bei Single-Precision auf der Hardware für die „ $a*b+c$ “-Operation bei den Architekturen vor Fermi eine MAD-Operation und bei den Architekturen ab Fermi eine FMA-Operation ausgeführt (siehe Tabelle 1) [5] [15].

Tabelle 1: Maximale Anzahl von MAD- und FMA-Operationen pro Takt (erstellt auf Basis einer Tabelle von [15], Informationen von [1] [15])

GPU	G80	GT200	Fermi
Multiprozessoren	16	30	16
Double-Precision-Operationen / Takt	-	30 FMA	256 FMA
Single-Precision-Operationen / Takt	128 MAD	240 MAD	512 FMA

Mit der GT200-Architektur kann pro Multiprozessor (=8 Kerne) pro Takt eine Double-Precision FMA-Operation ausgeführt werden [15]. Mit der Fermi-Architektur können pro Multiprozessor (=32 Kerne) 16 Double-Precision FMA-Operationen ausgeführt werden [15].

2.1.3 Single-Precision Peak GFLOP/s

Die Single-Precision Peak GFLOP/s einer GPU errechnen sich durch die Multiplikation der Taktfrequenz mit der Anzahl der Kerne und mit der maximalen Anzahl von Operationen pro Kern, die während eines Takts ausgeführt werden können.

Zum Beispiel die Nvidia G80-GPU besteht aus 128 Kernen (=16 Multiprozessoren*8 Kerne), wobei jeder Kern eine MAD(=Multiply-Add)-Einheit und eine MUL(=Multiply)-Einheit besitzt [1]. Die MUL-Einheit ist jedoch virtuell, da die MUL-Operationen auf den SFUs (=Special Function Units) ausgeführt werden [16]. Durch „Dual-Issue“ können parallel zu den MAD-Operationen auf den FPU's MUL-Operationen auf den SFUs ausgeführt werden [16]. Der Durchsatz beträgt mit „Dual-Issue“ 3 Operationen pro Kern pro Takt [16]. Mit einer Taktfrequenz von 1,35 GHz erreicht die GPU somit eine Floating-Point-Leistung von 518 Gigaflop/s (=16*8*3*1,35) [1] [17]. Die GT200-GPU besteht aus 240 Kernen und überschreitet 1 Teraflop/s [1]. Die vollausgebaute Fermi-GPU (GTX 580) besteht aus 512

Kernen mit je einer FMA-Einheit (=2 Operationen pro Kern pro Takt) und bietet 1,581 TFLOP/s ($=16 \cdot 32 \cdot 2 \cdot 1,544$) [15] [18].

Im Vergleich zu Single-Precision Peak GFLOP/s sind die Double-Precision Peak GFLOP/s nicht so leicht über Taktfrequenz und Anzahl der Kerne errechenbar. Außerdem haben die Tesla-GPUs seit der Fermi-Architektur deutlich mehr Double-Precision Peak GFLOP/s als die Geforce-GPUs [19]. Bei Double-Precision Peak GFLOP/s muss die exakte Zahl bzw. die Berechnungsformel auf der Hersteller-Seite gesucht werden.

2.1.4 Integer-Einheit

Jeder Kern besitzt neben der Floating-Point-Einheit eine eigene Integer-Einheit [15].

Mit Fermi können 32-Bit-Integer-Multiplikationen pro Kern mit einer Instruktion in einem Taktzyklus ausgeführt werden [15]. Da Fermi 32 Kerne pro Multiprozessor besitzt, können 32 Integer-Multiplikationen in 32-Bit pro Multiprozessor pro Takt ausgeführt werden (siehe 32-bit integer-multiply in Tabelle 2) [6] [15]. In der Vorgängerarchitektur GT200 sind für das Ausführen einer 32-Bit-Integer-Multiplikation mehrere Instruktionen und somit mehrere Taktzyklen notwendig (siehe 32-bit integer-multiply in Tabelle 2), außer es wird die Multiplikation in 24 Bit durchgeführt [15]. Da GT200 8 Kerne pro Multiprozessor besitzt, können 8 Integer-Multiplikationen in 24-bit pro Multiprozessor pro Takt ausgeführt werden (siehe 24-bit integer-multiply in Tabelle 2) [6] [15].

Tabelle 2: Maximale Anzahl von Integer-Operationen pro Multiprozessor pro Takt (erstellt auf Basis einer Tabelle von [6], Informationen von [6] [15])

	GT200	Fermi
32-bit integer multiply, multiply-add, sum of absolute difference	Emulation (Mehrere Instruktionen pro Operation)	32
24-bit integer multiply (mul24(x,y))	8	Emulation (Mehrere Instruktionen pro Operation)
32-bit integer add, logical operation, shift, compare	8	32

2.1.5 SFUs (Special Function Units)

Jeder Multiprozessor besitzt zu den Kernen noch SFUs (=Special Function Units) [1]. Die G80- und GT200-Architektur besitzen 2 SFUs pro Multiprozessor [6] [7]. Mit der Fermi-Architektur wurde die Anzahl pro Multiprozessor auf 4 verdoppelt [6] [7].

Auf den SFUs können spezielle Single-Precision-Funktionen, wie „reciprocal square root“ ($=1/\sqrt{x}$), „reciprocal“ ($=1/x$) und transzendente Funktionen (Exponentialfunktion, Logarithmus und trigonometrische Funktionen (sin, cos)), ausgeführt werden [1] [6] [15]. Bei den Architekturen G80 und GT200 können auch Single-Precision Floating-Point Multiplikationen auf den SFUs ausgeführt werden [5] [7].

Eine SFU führt eine Operation aus der SFU-Pipeline in einem Taktzyklus aus [15]. Je nachdem wie viele SFUs eine GPU-Architektur pro Multiprozessor besitzt, desto mehr SFU-Operationen können pro Takt auf einem Multiprozessor ausgeführt werden. Bei der Compute Capability 1.x (G80 und GT200) können 2 SFU-Instruktionen pro Multiprozessor pro Takt ausgeführt werden (siehe Tabelle 3) [6] [7]. Bei der Compute Capability 2.0 (Fermi) können 4 SFU-Instruktionen pro Multiprozessor pro Takt ausgeführt werden (siehe Tabelle 3) [6] [7].

Tabelle 3: Maximale Anzahl von SFU-Instruktionen pro Multiprozessor pro Takt (erstellt auf Basis einer Tabelle von [6], Informationen von [6] [15])

	Compute Capability 1.x (G80 und GT200)	Compute Capability 2.0 (Fermi)
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (native_log), base-2 exponential (native_exp), sine (native_sin), cosine (native_cos)	2	4

Wenn die SFU-Pipeline über mehrere noch auszuführenden Operationen verfügt, dann kann sich die Dispatch Unit um das Zuteilen von Additionen/Multiplikationen-Operationen an die FPU's/Kerne kümmern [15]. FPU's und SFU's können also parallel arbeiten.

2.1.6 Aktive Threads

Aktive Threads sind alle Threads, die von der GPU initialisiert wurden, aber noch nicht vollständig ausgeführt wurden [6]. Bei jedem Instruktionsausfertigungszyklus auf einem Multiprozessor wählt der Warp Scheduler einen aktiven Warp mit exekutionsbereiten Threads aus [6]. Sollten aktive Threads auf etwas warten, gibt es andere aktive Threads, die ausgeführt werden können [6]. Bei dem Wechsel zwischen den aktiven Threads entstehen keine Kosten [6].

Da zum Beispiel die G80-GPU 768 Threads pro Multiprozessor parallel aktiv halten und abwechselnd abarbeiten kann, muss eine Anwendung mindestens um die 5000-12000 Threads zur Abarbeitung anbieten, damit eine GPU einigermaßen ausgelastet ist [1]. Bei der G80-GPU kommt man auf 12288 (=768*16) Threads, die parallel aktiv gehalten werden können [1]. Die GT200-GPU kommt auf 30720 (=1024*30) Threads [1]. Bei der vollausgebauten Fermi-GPU (GTX 580) sind es 24576 Threads (=32 Threads*48 Warps*16 SM) [5] [20]. Um die maximale Anzahl von aktiven Threads erreichen zu können, darf die Registeranzahl pro Thread nicht zu hoch sein (siehe Kapitel 2.2.4).

2.1.7 Thread-Blöcke

Auf einem Multiprozessor werden Blöcke von Threads ausgeführt [1]. Diese Blöcke können in beliebiger Reihenfolge (=out of order) ausgeführt werden, da sie nicht voneinander abhängig sind [1]. Damit können noch nicht abgearbeitete Blöcke auf einem Multiprozessor ausgeführt werden, der gerade ohne Arbeit ist [1]. Es kann also zur Laufzeit entschieden werden, auf welchen Multiprozessor ein Block ausgeführt wird. Das Programm skaliert automatisch mit der Anzahl der Multiprozessoren [1] und seit der Fermi-Architektur durch die Einführung mehrerer Warp-Schedulers pro Multiprozessor auch mit der Anzahl der Warp Schedulers pro Multiprozessor [5]. Diese automatische Skalierung wird als „transparent scalability“ bezeichnet [1]. In einem Block von Threads hingegen kann durch das Setzen von Synchronisationspunkten eine Abhängigkeit zwischen den Threads erzeugt werden [1]. Aus diesem Grund müssen alle Threads eines Blocks immer auf denselben Multiprozessor ausgeführt werden. Für die Realisierung einer Synchronisation zwischen Blöcken muss ein neuer Kernel aufgerufen werden [1]. Bei Nvidia-GPUs können pro Multiprozessor maximal 8 Blöcke parallel aktiv gehalten werden [5]. Die Anzahl der gleichzeitig aktiven Blöcke pro Multiprozessor wird automatisch reduziert, sollten die Ressourcen (Shared Memory, Register) pro Multiprozessor nicht für alle Threads der 8 Blöcke ausreichen [1]. Aber auch wenn die maximale Anzahl der aktiven Threads erreicht wird, muss die Anzahl der aktiven Blöcke reduziert werden. Pro Multiprozessor können zum Beispiel in der GT200-Architektur 1024 Threads gleichzeitig aktiv gehalten werden [1]. Wenn im Programm eine Blockgröße von 128 Threads definiert ist, dann können 8 Blöcke gleichzeitig aktiv gehalten werden [1]. Wenn ein Block jedoch aus 256 Threads besteht, dann können nur mehr 4 Blöcke parallel bearbeitet werden [1]. Bei 64 Threads pro Block, wären bei Betrachten der maximalen Threadanzahl pro Multiprozessor 16 Blöcke möglich, jedoch ist die Anzahl der gleichzeitig aktiven Blöcke pro Multiprozessor auf 8 beschränkt [1]. Beim

Programmieren muss neben der maximalen Anzahl von aktiven Blöcken pro Multiprozessor beachtet werden, dass es auch für die Anzahl der Threads pro Block ein Maximum gibt [1]. Die Beschränkung kann von GPU zu GPU unterschiedlich sein [1]. Bei der GT200-Architektur sind zum Beispiel maximal 512 Threads pro Block erlaubt [1]. Bei der Fermi-Architektur sind es 1024 Threads [5] [20].

2.1.8 Warps (=32 Threads) und Warp Scheduler

Sobald ein Block einem Multiprozessor zur Ausführung zugewiesen wurde, wird er in Pakete von je 32 Threads unterteilt [1]. Diese Pakete werden als Warp bezeichnet [1]. Aus den ersten 32 Threads eines Blocks entsteht Warp 1 und aus den weiteren 32 Threads entsteht Warp 2 [1]. Sollte die Blockgröße kein Vielfaches von 32 sein, dann bekommt der letzte Warp ein paar zusätzliche Threads [1]. Da bei der GT200-Architektur 1024 Threads pro Multiprozessor aktiv gehalten werden können, hat man bei der Blockgröße von 128 Threads 8 aktive Blöcke und somit können 32 Warps ($=128 \cdot 8 / 32$) parallel bearbeitet werden [1]. Da man aber nur 8 SPs pro Multiprozessor hat, können nur 8 Threads von 1024 aktiven Threads gleichzeitig eine Berechnung durchführen. Die 8 Threads, die ausgeführt werden, sind von demselben Warp. Sobald ein Thread eines Warps auf etwas wartet, wird durch den Warp Scheduler ein anderer aktiver Warp, dessen Threads auf nichts warten, für die Ausführung ausgewählt [1]. Wenn mehrere Warps zur Auswahl stehen, wird ein Prioritätsmechanismus verwendet [1]. Durch das Wechseln zwischen Warps, sobald ein Warp auf etwas warten muss, kann ein völliger Stillstand der Recheneinheiten verhindert und somit die Latenzzeit der Operationen versteckt werden [1]. Dies wird als „latency hiding“ bezeichnet [1]. Damit auf einem Multiprozessor mehr Threads aktiv gehalten werden können, als ausgeführt werden können, müssen die Thread-, Block-IDs und der Ausführungsstatus gesichert werden [1]. Durch das Verstecken der Latenzzeit spielen Caches bei GPUs eine nicht so große Rolle wie bei CPUs und es kann mehr Chip-Fläche für die Floating-Point-Einheiten verwendet werden [1].

Sobald ein Warp zur Ausführung ausgewählt wurde, führt der Multiprozessor dieselbe Instruktion im Code für alle Threads innerhalb eines Warps aus [1]. Erst nachdem alle Threads innerhalb des Warps ihre Instruktion ausgeführt haben, wird zu der nächsten Instruktion im Kernel weitergegangen [1]. Diesen Ausführungsstil bezeichnet man als SIMT (=Single-Instruction, Multiple-Thread) [1]. Auf der CPU gibt es mit SSE (=Streaming SIMD Extensions) etwas Ähnliches, jedoch müssen hier die Datenelemente zuerst in einen gemeinsamen Register gepackt werden, damit eine Instruktion auf mehrere Datenelemente als Vektoroperation ausgeführt werden kann [1]. Im Vergleich dazu wird auf der GPU mit SIMT direkt auf die einzelnen Register, die den Threads zugewiesen sind, zugegriffen [1]. Bei einer Warpgröße von 32 Threads und 8 Kernen dauert das Ausführen einer Instruktion eines Warps 4 Zyklen [1]. Länger kann es dauern, wenn die Threads innerhalb eines Warps einen unterschiedlichen Control-Flow-Pfad (z. B. ein paar Threads den if-Zweig und die restlichen Threads den else-Zweig) verfolgen [1]. Dann müssen zunächst alle Threads des Warps, die den if-Zweig verfolgen, ausgeführt werden und danach alle Threads, die den else-Zweig ausführen, abgearbeitet werden [1]. Die Abweichung beim Control-Flow-Pfad zwischen Threads innerhalb eines Warps wird als „Divergenz“ bezeichnet [1]. Divergenz führt immer dazu, dass die zwei unterschiedlichen Pfade sequentiell nacheinander ausgeführt werden müssen [1]. Keine Divergenz hat man, wenn Threads innerhalb eines Blocks unterschiedliche Control-Flow-Pfade nehmen, aber innerhalb eines Warps der Control-Flow-Pfad der Threads gleich ist [1].

Mit der Fermi-Architektur gibt es statt einem Warp-Scheduler zwei Warp-Schedulers und es können somit zwei Warps parallel pro Multiprozessor abgearbeitet werden (siehe Kapitel 2.4.2.6).

2.2 Unterschiedliche Speicher

Die GPU bietet einen großen Global-Memory-Speicher und viele kleine Speicher. Auf die kleineren Speicher, wie Register, Shared Memory und Constant Memory, kann mit einer höheren Geschwindigkeit als auf den Global Memory zugegriffen werden [1]. Um diese kleineren Speicher

effizient verwenden zu können, sind fast immer Änderungen am straightforward-Code des Algorithmus notwendig [1].

Es ist wichtig, dass man sich beim Programmieren mit den Hardwaredaten der einzelnen Speicher auseinandersetzt, um die limitierenden Faktoren des Programms, ausgelöst durch die Speicher, zu erkennen und das Programm dementsprechend anpassen zu können [1]. Es sollte zum Beispiel die Global-Memory-Bandbreite berücksichtigt werden [1]. Gibt es zu viele gleichzeitige Zugriffe auf den Global Memory kommt es zu längeren Wartezeiten [1]. Weiters sollte die maximale Anzahl der Register pro Block nicht überschritten werden [1]. Denn beim Überschreiten reduziert sich die Anzahl der aktiven Blöcke und man hat somit weniger aktive Threads pro SM, die Berechnungen durchführen können, sobald andere aktive Threads auf Daten vom Global Memory warten [1]. Dadurch könnte die Latenzzeit nicht mehr versteckt werden. Zu einer Reduzierung der aktiven Blöcke kann es auch kommen, wenn Shared Memory im Programm benützt wird und die maximale Größe pro Block überschritten wird [1].

In der nachfolgenden Tabelle 4 ist ein kurzer Überblick über die Speicher einer GPU.

Tabelle 4: Unterschiedliche Speicher einer GPU (Informationen von [1] [5], Spalten Sichtbarkeit und Lebensdauer basieren auf einer Tabelle von [1])

	Auf dem Chip	Zugriff	Verwaltung durch Programmieren	Sichtbarkeit	Lebensdauer	Eigenschaft
Register	Ja	R/W (Device)	Teilweise (Compiler kann Änderungen durchführen)	Thread	Kernel/Thread	Speicherung der skalaren Variablen eines Threads
Local Memory	Nein	R/W (Device)	Teilweise (Compiler kann Änderungen durchführen)	Thread	Kernel/Thread	Speicherung der Array-Variablen eines Threads, wenn die Indizes nicht konstant oder das Array zu viele Register benützen würde
Shared Memory	Ja	R/W (Device)	Ja	Block	Kernel/Block	Klein, aber schnell (niedrige Latenzzeit), Austauschen von Daten zwischen Threads eines Blocks
Global Memory	Nein	R/W (Device und Host)	Ja	Grid	Application	Groß, aber langsam
Constant Memory	Nein, Ja (Cache)	R (Device), W (Host)	Ja	Grid, Blöcke eines SM (Cache)	Application	Sehr schneller, paralleler Read-only Zugriff
L1-Cache	Ja	R/W (Device)	Nein	Blöcke eines SM		
L2-Cache	Ja	R/W (Device)	Nein	Grid		

2.2.1 Host Memory (=off chip memory)

Der Host Memory ist der System-DRAM der CPU [1]. Da ein GPU-Programm auf der CPU gestartet wird und erst durch einen Kernelaufruf im Host-Code das Ausführen von Device-Code auf der GPU erfolgt (siehe Kapitel 3.3.6), wird der Host Memory und der Device Memory verwendet [1]. Über den PCI-Bus können Daten vom Host Memory in den Device Memory und zurück übertragen werden (siehe Kapitel 2.3).

2.2.2 Device Memory (=off chip memory)

Der Device Memory beinhaltet Global Memory (siehe Kapitel 2.2.3), Local Memory (siehe Kapitel 2.2.5) und Constant Memory (siehe Kapitel 2.2.7) [5].

Beim Device Memory handelt es sich um den GDDR(=Graphics Double Data Rate)-DRAM(=Dynamic Random Access Memory) auf der Grafikkarte [1]. Da es sich um einen DRAM handelt ist er ein großer günstiger Speicher (Mehrere GByte). Der Unterschied zum System-DRAM der CPU ist, dass die Bandbreite und die Latenzzeit höher ist [1]. Die höhere Latenzzeit (mehrere 100 Zyklen) kann durch das Abwechseln von vielen Threads einigermaßen versteckt werden [1]. Die Bandbreite zwischen GPU-Arbeitsspeicher/Device Memory und GPU ist mit 177 GB/s [21] bei der Grafikkarte GTX 480 deutlich größer als die Bandbreite zwischen dem System-DRAM und der CPU Intel Xeon X5550 mit 32 GB/s [22]. Da eine GPU viele Threads parallel ausführt, kann die höhere Bandbreite im Vergleich zum System-DRAM nicht ausreichend sein. Die benötigte Bandbreite kann durch Nutzung der kleineren Speicher durch Programmierung bzw. durch die Hardware gesteuerten Caches bei der Fermi-Architektur deutlich reduziert werden [1]. Dabei reduziert sich auch die Latenzzeit deutlich.

Die Bandbreite in GB/s lässt sich errechnen durch Speichertaktfrequenz in GHz*2(=Double Data Rate)*Breite der Speicherschnittstelle in Bits/8(=Umrechnen in Byte) [23]. Die Bandbreite drückt aus, wie viele Bytes maximal pro Sekunde zwischen GPU und Device-Memory/DRAM übertragen werden können [23].

2.2.3 Global Memory (=off chip memory)

Der Global Memory befindet sich im Device Memory (siehe Kapitel 2.2.2) [5]. Die Nutzung des Global Memory ist der Ausgangspunkt jedes GPU-Programms und ermöglicht mit wenigen Änderungen am CPU-Code, den Code auf der GPU auszuführen (=straightforward code). Die Startdaten, die die einzelnen Threads benötigen, werden zunächst vom Arbeitsspeicher der CPU zum Global Memory der GPU mit Hilfe einer API(=Application Programming Interface)-Funktion übertragen [1]. Jeder Thread greift auf seine zu verarbeitenden Daten im Global Memory zu, indem er seine Block-ID und seine Thread-ID benützt [1].

Auf den Global Memory sollte bei einem Kernel nur zu Beginn und am Ende zugegriffen werden, um die Inputdaten zu lesen und die Outputdaten zu schreiben [1]. Dazwischen sollten Registervariablen verwendet werden, die die Zwischenergebnisse sichern [1].

2.2.3.1 CGMA

Damit man weiß, ob die Bandbreite des Global Memory ein limitierender Faktor des Programms ist, kann der CGMA-Wert berechnet werden [1]. CGMA steht für „Compute to Global Memory Access“ und gibt die Anzahl der Floating-Point-Operationen pro einen Global-Memory-Zugriff an [1]. Wenn man in der innersten Schleife des Kernels zwei Floating-Point-Operationen und zwei Datenzugriffe auf den Global Memory hat, erhält man ein CGMA von 1 [1]. Bei einer Bandbreite des Global Memory von 177 GB/s (GTX 480) [21], einen CGMA von 1 und float-Werten (=4 Byte) können maximal 44,25 (=177/4*1) GFLOP/s ausgeführt werden. Wäre der Global Memory nicht durch die Bandbreite limitiert und der Kernel würde aus einer FMA(=Fused Multiply-Add)-Operation (=a*b+c) bestehen, dann wäre die GPU GTX 480 zu 1345 GFLOP/s [5] [21] in der Lage. Um einen höheren CGMA-Wert zu erhalten, müssen im Kernel deutlich mehr Instruktionen als Datenzugriffe auf den Global Memory durchgeführt werden [1]. Die Anzahl der Datenzugriffe kann durch Verwendung der kleineren Speicher reduziert werden [1].

2.2.3.2 Memory Coalescing

Wenn ein Warp (=32 Threads) einen Datenzugriff auf den Global Memory durchführt, werden die Datenzugriffe der Threads zu möglichst wenigen gemeinsamen Datenzugriffen zusammengefasst [5].

Dies wird als Memory Coalescing bezeichnet [5]. Das beste Memory Coalescing wird erreicht, wenn für den Warp pro Datenzugriff im Kernel nur 1 gemeinsamer Datenzugriff auf den Global Memory durchgeführt werden muss. Je nach Programmierung und Hardware kann die Anzahl deutlich darüber liegen.

Beim Zugriff auf den Global Memory sind drei Segmentgrößen möglich (32, 64 und 128 Bytes) [5]. Da die kleinste Segmentgröße mit 32 Bytes deutlich größer als der 4-Byte-Zugriff bei einem Float- oder Int-Wert ist, würde es ohne Memory Coalescing zu einer hohen Bandbreitenverschwendung kommen. Weiters muss beachtet werden, dass die Segmente nicht ab einer beliebigen Adresse ausgelesen werden können [5]. Der Speicher ist in 32 Byte Segmente, 64 Byte Segmente und 128 Byte Segmente unterteilt [5]. Zwei 32 Byte Segmente bilden ein 64 Byte Segment und zwei 64 Byte Segmente bilden ein 128 Byte Segment [5]. Erfolgt der Zugriff im Kernel exakt auf diese Segmente im Speicher wird der Zugriff als aligned bezeichnet (siehe Tabelle 5), sonst als misaligned (siehe Tabelle 6) [5]. Wenn man die Tabelle 6 betrachtet, sieht man, dass Misaligned-Zugriffe bei der älteren G80-Architektur (=Compute Capability 1.0 und 1.1) zu der Situation führen können, dass pro Thread ein eigener Datenzugriff durchgeführt wird und somit pro Warp pro Datenzugriff statt einem gemeinsamen 128 Byte Datenzugriff (=4 Byte*32 Threads) nun 1024 Byte (=32 Byte Segment*32 Threads) vom Global Memory ausgelesen werden müssen [5] [7]. Bei der GT200-Architektur (=Compute Capability 1.2 und 1.3) und bei der Fermi-Architektur (=Compute Capability 2.0) kommt es zu deutlich weniger separaten Zugriffen [5] [7].

Tabelle 5: Aligned und sequentieller Zugriff eines Warps, 32 Threads mit je 4-Byte-Zugriff (erstellt auf Basis einer Tabelle von [5], Informationen von [5])

Zugriff auf folgende Daten	128 Bytes ab Adresse 128 auslesen (Thread 1 - ersten 4 Bytes, Thread 2 - zweiten 4 Bytes, ..., Thread 32 - 32. 4 Bytes)		
Compute Capability	1.0 und 1.1	1.2 und 1.3	2.0
Mit oder ohne Caching	Ohne Caching	Ohne Caching	Mit Caching
Anzahl von Memory Transaktionen	1x 64 Bytes ab Adresse 128 1x 64 Bytes ab Adresse 192	1x 64 Bytes ab Adresse 128 1x 64 Bytes ab Adresse 192	1x 128 Bytes ab Adresse 128

Tabelle 6: Misaligned und sequentieller Zugriff eines Warps, 32 Threads mit je 4-Byte-Zugriff (erstellt auf Basis einer Tabelle von [5], Informationen von [5])

Zugriff auf folgende Daten	128 Bytes ab Adresse 129 auslesen (Thread 1 - ersten 4 Bytes, Thread 2 - zweiten 4 Bytes, ..., Thread 32 - 32. 4 Bytes)		
Compute Capability	1.0 und 1.1	1.2 und 1.3	2.0
Mit oder ohne Caching	Ohne Caching	Ohne Caching	Mit Caching
Anzahl von Memory Transaktionen	7x 32 Bytes ab Adresse 128 8x 32 Bytes ab Adresse 160 8x 32 Bytes ab Adresse 192 8x 32 Bytes ab Adresse 224 1x 32 Bytes ab Adresse 256	1x 128 Bytes ab Adresse 128 1x 64 Bytes ab Adresse 192 1x 32 Bytes ab Adresse 256	1x 128 Bytes ab Adresse 128 1x 128 Bytes ab Adresse 256

Bei der Fermi-Architektur werden im Gegensatz zu den älteren Architekturen die Daten in einer Cache-Hierarchie zwischengespeichert (siehe „Ohne Caching“ und „Mit Caching“ in der Tabelle 6). Das führt dazu, dass immer eine Cache Line (=128 Byte) aus dem Global Memory ausgelesen wird und es somit bei Fermi nur eine Segmentgröße gibt [5]. Wird nur der L2-Cache benützt und die L1-Caches deaktiviert, wird die Segmentgröße von 128 Bytes auf 32 Bytes reduziert [5]. Für die Deaktivierung der L1-Caches wird beim Kompilieren das Flag „-Xptxas -dlcm=cg“ gesetzt (siehe Kapitel 2.4.2.3). Der große Vorteil der Cache-Hierarchie bei der Fermi-Architektur ist, dass bei Vorhandensein der Daten im Cache kein Zugriff auf den Global Memory erfolgt und dadurch die Berücksichtigung von Memory Coalescing bei der Optimierung von GPU-Programmen nicht mehr so wichtig ist.

Ein weiterer Unterschied bei der Fermi-Architektur gegenüber den Vorgängerarchitekturen ist, dass das Coalescing über einen Warp (=32 Threads) und nicht über einen halben Warp (=16 Threads) stattfindet [5]. In Tabelle 5 sieht man, dass die älteren Architekturen pro Warp im besten Fall zwei coalesced-Zugriffe auf den Global Memory haben. Hingegen bei der Fermi-Architektur ist im besten Fall nur ein Zugriff notwendig.

Nicht-sequentielle Zugriffe führen ab Compute Capability 1.2 zu deutlich weniger Zugriffen (siehe Tabelle 7).

Tabelle 7: Aligned und nicht-sequentieller Zugriff eines Warps, 32 Threads mit je 4-Byte-Zugriff (erstellt auf Basis einer Tabelle von [5], Informationen von [5])

Zugriff auf folgende Daten	128 Bytes ab Adresse 128 auslesen (Thread 1 - ersten 4 Bytes, Thread 2 - zweiten 4 Bytes, ..., Thread 4 - fünften 4 Bytes, Thread 5 - vierten 4 Bytes, Thread 6 - 6. 4 Bytes, ..., Thread 26 - 27. 4 Bytes, Thread 27 - 26. 4 Bytes, Thread 28 - 28. 4 Bytes, ..., Thread 32 - 32. 4 Bytes)		
Compute Capability	1.0 und 1.1	1.2 und 1.3	2.0
Mit oder ohne Caching	Ohne Caching	Ohne Caching	Mit Caching
Anzahl von Memory Transaktionen	8x 32 Bytes ab Adresse 128 8x 32 Bytes ab Adresse 160 8x 32 Bytes ab Adresse 192 8x 32 Bytes ab Adresse 224	1x 64 Bytes ab Adresse 128 1x 64 Bytes ab Adresse 192	1x 128 Bytes ab Adresse 128

2.2.4 Register (=on chip memory, Sichtbarkeit: Thread)

Register werden Threads zugewiesen und auf die Daten im Register kann nur der Thread selbst zugreifen [1]. Im Register werden nur skalare Variablen gespeichert [1]. Array Variablen werden im Local Memory gespeichert [1], jedoch nur wenn sie zu viele Register belegen oder keine konstanten Indizes besitzen [5]. Die Variablen im Register oder Local Memory werden als automatic variables bezeichnet [1]. Sie besitzen keinen Type Qualifier. Sowohl die skalaren Argumente des Kernels als auch die skalaren Variablen im Kernel ohne Type Qualifier führen zur Belegung von Registern [1]. Für jeden Thread wird eine private Instanz der Variable in einem Register angelegt [1]. Sobald der Thread vollständig abgearbeitet ist, wird die private Instanz der Variable wieder gelöscht [1].

Der Zugriff auf die Variablenwerte im Register geschieht sehr schnell und parallel [1]. Die Register werden für häufig benutzte Variablen verwendet, die entweder durch Programmierung oder vom

Compiler erzeugt werden, um die Latenzzeit darauf zu reduzieren und die Global-Memory-Bandbreite zu schonen [1]. Jedoch kann die beschränkte Anzahl der Register auf der GPU bei der Benützung von zu vielen Registern pro Thread zu einem Problem werden [1]. Wird der Datentyp Float und die maximale Anzahl von aktiven Threads verwendet, können bei der Fermi-Architektur 21 (=128 KB Register Memory pro SM*1024/4 Bytes/1536 aktive Threads pro SM) und bei der Vorgängerarchitektur GT200 16 (=64 KB Register Memory pro SM*1024/4 Bytes/1024 aktive Threads pro SM) Register pro Thread belegt werden [5] [7]. Beim Überschreiten der maximalen Registeranzahl pro Thread wird automatisch die Anzahl der aktiven Blöcke (=Menge von aktiven Threads) reduziert [1]. Wenn man zum Beispiel 10 Register pro Thread benötigt, eine Blockgröße von 256 hat und ein Multiprozessor maximal 8192 4 Byte-Register (=32 KB) besitzt, dann können maximal 3 Blöcke (=8192/(10*256)) auf einem Multiprozessor aktiv sein [1]. Bei einer GPU-Architektur mit maximal 768 aktiven Threads pro Multiprozessor würde die maximale Threadanzahl erreicht werden [1]. Würde man jedoch ein Register mehr, also 11 Register pro Thread benötigen, könnte man nur mehr 2 Blöcke pro Multiprozessor parallel ausführen [1]. Damit würde man die Parallelität pro Multiprozessor um 1/3 reduzieren, da nun statt 3 Blöcke (768 Threads) nur mehr 2 Blöcke (512 Threads) parallel ausgeführt werden können [1]. Ein solcher starker Einschnitt in die Parallelität und somit Performance wird als „Performance cliff“ bezeichnet [1].

In der Fermi-Architektur darf ein Thread maximal 63 Register belegen [24]. Wenn die Anzahl von 63 Registern pro Thread überschritten wird, werden die zu vielen Variablen statt im Register Memory im Local Memory angelegt [5]. Dies wird als „Register Spilling“ bezeichnet [5]. Erreicht ein Kernel diese hohe Anzahl von Registern pro Thread, wäre die Anzahl der aktiven Threads pro Multiprozessor stark reduziert und es kommt zu längeren Wartezeiten, wenn Threads auf den Global Memory zugreifen, da die Menge der ausführbaren aktiven Threads deutlich kleiner wäre. Mit der Verwendung des „-maxrregcount=N“-Flags beim Kompilieren des Programms mit dem nvcc-Compiler kann die maximale Grenze von 63 Registern künstlich reduziert werden und somit Register Spilling früher ausgelöst werden [5] [23]. Damit könnte bei einem Kernel mit einer hohen Anzahl von Registern pro Thread dennoch die maximale Anzahl von aktiven Threads pro Multiprozessor erreicht werden [5]. Bei der Fermi-Architektur müsste zum Beispiel die Grenze auf 21 Register (siehe Berechnung im vorigen Absatz) herabgesetzt werden. Dabei muss jedoch beachtet werden, dass ein Zugriff auf den Local Memory deutlich länger als ein Zugriff auf den Register Memory dauert (siehe Kapitel 2.2.5) [5].

Der Compiler versucht die Anzahl der Register, Register Spilling und die Anzahl der Instruktionen auf ein Minimum zu reduzieren [5]. Um den Compiler zu unterstützen, können im Programmcode Zusatzinformationen eingefügt werden [5]. Zum Beispiel kann bei der Kerneldeklaration zwischen „__global__“ und dem Kernelnamen „void __launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)“ geschrieben werden [5]. „maxThreadsPerBlock“ wird durch die maximale Anzahl von Threads pro Block ersetzt. Optional kann man als zweites Argument die minimale Anzahl der aktiven Blöcke pro Multiprozessor angeben [5].

Für jeden Kernel im Programmcode kann die Anzahl der benützten Register pro Thread durch das Setzen des „-ptxas-options=-v“-Flags beim Kompilieren ausgegeben werden [20].

2.2.5 Local Memory (=off chip memory, Sichtbarkeit: Thread)

Der Local Memory in CUDA ist der Private Memory in OpenCL [1]. Der Local Memory befindet sich wie der Global Memory im Device Memory (siehe Kapitel 2.2.2) und hat somit dieselbe Latenzzeit und Bandbreite wie Global Memory [5]. Seit der Fermi-Architektur werden Local-Memory-Zugriffe wie Global-Memory-Zugriffe in L1 und L2 zwischengespeichert (siehe Kapitel 2.2.6) [5]. Außerdem wurde die Größe von 16 KB auf 512 KB pro Thread vergrößert [5].

Der Compiler entscheidet, ob eine Variable im Register oder im Local Memory gespeichert wird [5].

Wenn die maximal erlaubte Anzahl von Registern pro Thread (bei der Fermi-Architektur maximal 63 Register pro Thread [24]) erreicht wird, werden alle weiteren Register im Local Memory angelegt (=Register Spilling) [5]. Mit dem Compilerflag „-maxrregcount=N“ kann diese Grenze herabgesenkt werden [23]. Damit kann verhindert werden, dass die Anzahl der aktiven Blöcke (=Menge von aktiven Threads) pro Multiprozessor bei einer hohen Anzahl von Registern pro Thread (bei der Fermi-Architektur ab 22 Registern pro Thread) reduziert wird (siehe Kapitel 2.2.4) [5].

Sowohl alle Array-Argumente des Kernels als auch alle Array-Variablen im Kernel ohne Type Qualifier führen zur Belegung von Local Memory [1], wenn sie zu viele Register benützen würden oder keine konstanten Indizes besitzen [5]. Für jeden Thread wird eine private Instanz der Array-Variable im Local Memory angelegt [1]. Sobald der Thread vollständig abgearbeitet ist, wird die private Instanz der Array-Variable wieder gelöscht [1].

Der benutzte Local Memory pro Thread jedes Kernels im Programmcode kann durch das Setzen des „--ptxas-options=-v“-Flags beim Kompilieren ermittelt werden [20].

2.2.6 L2-Cache und L1-Caches

Für Constant Memory gab es schon vor der Fermi-Architektur eigene Caches auf jedem Multiprozessor [5]. Mit der Fermi-Architektur gibt es nun auch Caches für Local Memory und Global Memory [5]. Die Werte vom Global Memory und Local Memory werden in einer gemeinsamen Cache-Hierarchie mit einem L2-Cache (768 KB) pro GPU und einem L1-Cache (Konfigurierbar: 16 KB oder 48 KB) pro Multiprozessor zwischengespeichert [5]. Der Cache funktioniert wie bei CPUs ohne Programmieraufwand. Genauere Informationen zu der Cache-Hierarchie sind im Kapitel 2.4.2.3 zusammengefasst.

2.2.7 Constant Memory und Constant Memory Caches (=read-only off/on chip memory)

Der Constant Memory (64 KByte pro GPU) befindet sich wie der Global Memory im Device Memory (siehe Kapitel 2.2.2) [5]. Die Variablen werden für einen sehr effizienten lesenden Zugriff in einem Cache zwischengespeichert (8 KByte Cache pro Multiprozessor) [1]. Dieser Speichertyp ermöglicht eine sehr kleine Latenzzeit und eine sehr hohe Parallelität, wenn Threads zur selben Zeit lesend auf dieselben Daten zugreifen [1] und die Daten sich im Constant-Memory-Cache am Multiprozessor befinden [5]. Der Nachteil des Speichers ist, dass nur Daten ausgelesen werden können und er im Vergleich zu den anderen Speichern relativ klein ist.

Die ideale Zugriffsform von Constant Memory ist, wenn alle Threads innerhalb eines Warps auf dasselbe Datenelement zugreifen [1]. In diesem Fall muss statt 32 Zugriffe auf den Constant Memory nur 1 Zugriff erfolgen und die restlichen Zugriffe greifen auf den Constant-Memory-Cache zu [1]. Wenn ein Zugriff auf den Constant-Memory-Cache stattfindet, kann mit Hilfe eines Broadcast der Wert zu allen Threads gesendet werden [1].

Wenn der Constant Memory mit 64 KB zu klein ist, muss der Datensatz in Teile von 64 KB zerlegt werden [1]. Der Kernel wird dann mehrmals aufgerufen und jedes Mal befindet sich ein anderer Teil im Constant Memory [1].

Der Constant-Memory-Cache ist deutlich kleiner (8 KByte pro SM) als der Constant Memory (64 KByte pro GPU) und kopiert aus dem Constant Memory immer einen Bereich nebeneinanderstehender Bytes (=Cache Line) in einen Cache-Eintrag [1]. Der Constant-Memory-Cache besteht aus wenigen Cache-Einträgen, um die Kosten niedrig zu halten [1]. Da die Cache-

Einträge deutlich größer als einzelne Datenelemente sind, müssen möglichst alle Daten daraus verwendet werden. Alle Daten im Constant Memory, auf die ein Thread während seiner Abarbeitung zugreift, müssen somit hintereinander im Constant Memory stehen, damit ein Thread möglichst wenige Cache-Lines benötigt [1]. Da unterschiedliche Warps (=32 Threads) bei unterschiedlichen Iterationen sein können, können somit viele Cache-Einträge notwendig sein [1]. Damit ist es wichtig, dass ein Warp möglichst wenige Einträge im Cache belegt und somit ausreichend Einträge für alle aktiven Warps eines SM verfügbar sind [1].

Der Typ-Qualifier bei CUDA, der vor den Variablentyp gesetzt werden muss, lautet in CUDA „__constant__“ (zwei Unterstriche davor und zwei danach) [1]. Die Deklaration der Variable muss bei CUDA außerhalb aller Funktionen stehen [1].

2.2.8 Shared Memory (=on chip memory, Sichtbarkeit: Block)

Der Shared Memory in CUDA ist der Local Memory in OpenCL [1]. Shared Memory wird immer einem Block von Threads zugewiesen [1]. Auf den Shared Memory eines Blockes können alle Threads des Blocks sowohl lesend als auch schreibend zugreifen [1]. Damit ist es möglich, dass Threads innerhalb eines Blocks sehr effizient Inputdaten und Zwischenergebnisse austauschen können [1]. Der Typ-Qualifier, der bei CUDA vor den Variablentyp gesetzt werden muss, lautet „__shared__“ (zwei Unterstriche davor und zwei danach) [1]. Der Zugriff auf Shared Memory ist sehr schnell und parallel [1].

Im Host-Code kann ein Shared-Memory-Array dynamisch angelegt werden [1]. Dazu wird beim Kernelaufwurf die Größe des dynamischen Shared-Memory-Arrays mit dem dritten Parameter angegeben (siehe Code 1) [5]. Damit das dynamische Array im Kernel verwendet werden kann, müssen zwei Codezeilen im Kernel-Code eingefügt werden (siehe Code 2) [5].

Code 1: Angeben der Größe des dynamischen Shared-Memory-Arrays beim Kernelaufwurf [5] (adaptiert)

```
Kernelname<<<dimGrid,dimBlock,sizeDynamicSharedMemory>>>(Parameter);
```

Code 2: Dynamisches Shared-Memory-Array im Kernel-Code [5] (adaptiert)

```
extern __shared__ Datentyp dynamicSharedMemoryStartpointer[];  
Datentyp* temp=(Datentyp*)dynamicSharedMemoryStartpointer;
```

Bei Verwendung von nur zwei Parametern beim Kernelaufwurf wird der dritte Parameter automatisch auf 0 gesetzt [5]. In diesem Fall kann das Shared-Memory-Array im Kernel-Code nur statisch angelegt werden (siehe Code 3) [5].

Code 3: Statisches Shared-Memory-Array im Kernel-Code [5]

```
__shared__ float shared[32];
```

Um das Entstehen eines Data Race zu verhindern, muss beim Programmieren ein Synchronisationspunkt eingefügt oder der Code so geschrieben werden, dass es nicht dazukommt. Ein Synchronisationspunkt wird durch das Einfügen der CUDA API-Funktion „__syncthreads()“ in den Kernel-Code realisiert [1]. Alle Threads eines Blocks müssen an dieser Stelle warten, bis alle Threads des Blocks dort ankommen [1].

Der Shared Memory ist in der Hardware auf mehrere gleich große Speichermodule aufgeteilt, die als Bänke bezeichnet werden [5]. Durch den gleichzeitigen Zugriff auf alle Bänke kann eine hohe Bandbreite ermöglicht werden [5]. Die Shared-Memory-Bänke werden im Kapitel 2.4.2.9 genauer erklärt.

2.2.8.1 Tiling

Shared Memory wird im Zusammenhang mit Tiles verwendet. Dazu wird die große zu verarbeitende Datenmenge im Global Memory in Tiles/Teilmengen zerlegt, die so groß sind, dass sie im kleinen Shared Memory Platz finden [1]. Diese Tiles müssen unabhängig voneinander abgearbeitet werden können, damit nicht nur ein Multiprozessor gleichzeitig Berechnungen durchführen kann, sondern alle beschäftigt sind [1]. Nicht immer kann ein Problem in Tiles zerlegt werden [1]. Um Tiles verwenden zu können, muss der Algorithmus Lokalität aufweisen [1]. An der Abarbeitung eines Tiles ist ein Block von Threads zuständig. Bei Nvidia-GPUs können pro Multiprozessor 8 Blöcke von Threads maximal ausgeführt werden. Das bedeutet, die Größe der Tiles darf dann maximal 1/8 der Shared-Memory-Größe eines Multiprozessors entsprechen. Falls die Größe darüber liegt, wird die Anzahl der gleichzeitig aktiven Blöcke pro Multiprozessor automatisch reduziert und somit verkleinert sich auch die Anzahl der gleichzeitig aktiven Threads pro Multiprozessor. Jeder Thread eines Blocks speichert einen Teil der Inputdaten in den Shared Memory. Durch das Setzen von Synchronisationspunkten können Wavefronts abgearbeitet werden und die Zwischenergebnisse aller Threads des Blocks sind bei der nächsten Wavefront für alle Threads verfügbar, ohne auf den langsamen Global Memory zugreifen zu müssen. Erst wenn das Tile von allen Threads vollständig abgearbeitet ist, werden die Ergebnisse in den Global Memory geschrieben. Durch die Nutzung von Tiles im Shared Memory kann verhindert werden, dass die Global-Memory-Bandbreite zu einem zu großen Hindernis wird, um in die Nähe der Maximalleistung der GPU zu kommen [1].

2.2.9 Latenzzeit, Bandbreite und Größe der einzelnen Speicher (Tesla C2050)

In der nachfolgenden Tabelle 8 befindet sich die Latenzzeit, Bandbreite und Größe der einzelnen Speicher der Tesla-GPU C2050. Diese Werte wurden aus der Literatur ermittelt. Manche Werte werden nicht von Nvidia bekanntgegeben.

Tabelle 8: Latenzzeit, Bandbreite und Größe der Speicher der Tesla C2050

	Latenzzeit	Bandbreite	Größe
Register			1792 KB (alle SMs zusammen) [5] [20] [25]
Local Memory	wie Global Memory (wenn Daten in den L1-Caches bzw. dem L2-Cache zwischengespeichert sind, dann Latenzzeit von L1-Cache bzw. L2-Cache) [5]	wie Global Memory (wenn Daten in den L1-Caches bzw. dem L2-Cache zwischengespeichert sind, dann Bandbreite von L1-Cache bzw. L2-Cache) [5]	maximal 512 KB pro Thread [5] und der reservierte Local Memory aller aktiven Threads darf die Größe des Device Memory nicht überschreiten
Shared Memory	wenn keine bank conflicts, dann 100x niedriger als Global Memory [23]	1030 GB/s (alle SMs zusammen) = $14 \text{ SM} * 32 \text{ Bänke} * 2 \text{ Byte} * 1,15 \text{ GHz}$ [5] [25] [26]	224 KB oder 672 KB (alle SMs zusammen) [5] [25]
Global Memory	groß (mehrere 100 Zyklen) [1], 400-800 (wenn Daten in den L1-Caches bzw. dem L2-Cache zwischengespeichert sind, dann Latenzzeit von L1-Cache bzw. L2-Cache) [5]	144 GB/s (ohne ECC) [26], 115 GB/s (mit ECC) [26] (wenn Daten in den L1-Caches bzw. dem L2-Cache zwischengespeichert sind, dann Bandbreite von L1-Cache bzw. L2-Cache) [5]	3 GB [25]
Constant Memory	wie Global Memory (wenn Daten in den Constant-Memory-Caches zwischengespeichert sind, dann kleinere Latenzzeit als Global Memory) [1]	wie Global Memory [6] (wenn Daten in den Constant-Memory-Caches zwischengespeichert sind, dann 4 Byte/Warp/SM [26], $14 \text{ SM} * 4 \text{ Byte} * 1,15 \text{ GHz}$ [26] = 64 GB/s (alle SMs zusammen))	64 KB Constant Memory, 112 KB Constant-Memory-Cache (alle SMs zusammen) [5] [25]
L1-Cache	derselbe Speicher wie Shared Memory [27]	derselbe Speicher wie Shared Memory [27]	224 KB oder 672 KB (alle SMs zusammen) [5] [25]
L2-Cache	keine Angabe von Nvidia	230 GB/s [26]	768 KB [5]

2.3 PCI-Anschluss

Je besser der PCI-Anschluss einer Grafikkarte ist, desto schneller können Daten zwischen dem Arbeitsspeicher der CPU (=Host Memory) und dem Arbeitsspeicher der GPU (=Device Memory) übertragen werden. Der PCI-Anschluss legt die maximal mögliche Bandbreite zwischen Host und Device fest.

Bei PCI Express 2.0 x16 hat man eine Peak-Bandbreite von 8 GB/s in eine Richtung [23]. Die PCI-Bus-Geschwindigkeit ist gegenüber der Peak-Bandbreite des Device Memory der GPU mit 177 GB/s (GTX 480) [21] deutlich langsamer. Wenn statt auf den Device Memory auf den Host Memory des Mainboards zugegriffen werden muss (z. B. beim Übertragen der Inputdaten und beim Sichern des Ergebnisses) kann die Peak-Bandbreite von 8 GB/s sehr einschränkend sein. Im Vergleich dazu muss ein reines CPU-Programm keine Daten über den PCI-Bus übertragen und die Peak-Bandbreite zum Beispiel bei der CPU Intel X5550 beim Zugriff auf den Arbeitsspeicher liegt mit 32 GB/s [22] deutlich über der Peak-Bandbreite des PCI-Buses mit 8 GB/s [23].

Mit Hilfe des Bandbreiten-Test-Programms (OpenCL und CUDA) [28] aus dem Nvidia-SDK kann man die tatsächliche PCI-Bandbreite ermitteln. Bei dem Programm werden 400 MB übertragen und als Ergebnis wird die Bandbreite in MB/s ausgegeben. Der höchste Wert, der beim Testprogramm erzielt wurde, war 5744.3 MB/s (siehe Tabelle 9). Die tatsächliche Bandbreite des PCI bei Verwendung der cudaMemcpy-Methode in CUDA liegt somit deutlich unter der Peak-Bandbreite von 8 GB/s. In der Praxis erreicht man mit der GTX 480 im CUDA-Bandbreitentestprogramm von Nvidia von der Peak-Bandbreite nur 54% (Pageable Memory) bis 70% (Pinned Memory) bei Übertragungen von Host zu Device und 39% (Pageable Memory) bis 42% (Pinned Memory) bei Übertragungen von Device zu Host.

Der Unterschied zwischen Pinned/Page-locked Memory und Pageable Memory ist, dass das Betriebssystem bei Arrays, die als Pinned/Page-locked Memory angelegt werden, kein Paging (=Auslagerung der Daten vom CPU-Arbeitsspeicher auf die Festplatte) durchführt und diese Arrays somit immer im CPU-Arbeitsspeicher behält (siehe Kapitel 3.3.3) [5] [29].

Tabelle 9: Erreichte PCI-Bandbreite bei der Ausführung des CUDA-Bandbreiten-Test-Programms

GPU	GTX 480	GTX 285	Tesla C2050	Tesla C1060
Nvidia Toolkit	4.0	4.0	3.2	3.2
Bandbreite Host zu Device (MB/s)	4458.1 (Pageable), 5722.3 (Pinned)	4593.8 (Pageable), 5726.0 (Pinned)	4350.2 (Pageable), 5744.3 (Pinned)	4707.3 (Pageable), 5727.4 (Pinned)
Bandbreite Device zu Host (MB/s)	3200.7 (Pageable), 3438.3 (Pinned)	3221.6 (Pageable), 3438.4 (Pinned)	3180.4 (Pageable), 3437.9 (Pinned)	3201.4 (Pageable), 3438.0 (Pinned)

Tabelle 10: Erreichte PCI-Bandbreite bei der Ausführung des OpenCL-Bandbreiten-Test-Programms

GPU	GTX 480	GTX 285	Tesla C2050	Tesla C1060
Nvidia Toolkit	4.0	4.0	3.2	3.2
Bandbreite Host zu Device (MB/s)	3167.4 (Pageable, Direct), 5502.6 (Pinned)	3184.9 (Pageable, Direct), 5541.1 (Pinned)	4315.4 (Pageable, Direct), 5547.9 (Pinned)	4735.8 (Pageable, Direct), 5560.9 (Pinned)
Bandbreite Device zu Host (MB/s)	2377.0 (Pageable, Direct), 3438.1 (Pinned)	2355.7 (Pageable, Direct), 3436.5 (Pinned)	3353.0 (Pageable, Direct), 3437.9 (Pinned)	3206.4 (Pageable, Direct), 3435.6 (Pinned)

Die Bandbreiten-Werte bei CUDA (siehe Tabelle 9) sind über alle GPUs fast identisch. Daten können schneller vom Host zum Device als vom Device zum Host transportiert werden.

Die Bandbreiten-Werte bei OpenCL (siehe Tabelle 10) sind bei „Pageable“ zwischen den Geforce-GPUs (GTX 480 und GTX 285) und den Tesla-GPUs (Tesla C2050 und Tesla C1060) unterschiedlich.

Bei GTX 285 und GTX 480 wird das Toolkit 4.0 und bei Tesla C1060 und Tesla C2050 wird das Toolkit 3.2 verwendet. Das CUDA-Bandbreitentestprogramm wird in Linux für „Pageable“ mit dem Befehl „./bandwidthTest --mode=range --start=400000000 --end=400000000 --increment=1 --device=0 --memory=Pageable“ und das OpenCL Programm mit „./oclBandwidthTest --mode=range --start=400000000 --end=400000000 --increment=1 --device=0 --memory=pageable --access=direct“ aufgerufen. Bei CUDA muss vor dem Aufruf die lib64-Bibliothek in den Bibliothekspfad eingefügt werden, damit sie gefunden wird. Das funktioniert über den Befehl „export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:/<Pfad zu lib64>/lib64“.

Tabelle 11: Erreichte PCI-Bandbreite in MB/s bei der Ausführung des CUDA- und OpenCL-Bandbreiten-Test-Programms von Nvidia mit einer zugewiesenen CPU bei zwei CPUs am Mainboard

CUDA	OpenCL
- Host to Device:	- Host to Device:
480 (CPU 0): 4458.1 (Pageable), 5722.3 (Pinned)	480 (CPU 0): 3167.4 (Pageable, Direct), 5502.6 (Pinned)
480 (CPU 1): 4535.9 (Pageable), 5055.5 (Pinned)	480 (CPU 1): 3090.7 (Pageable, Direct), 4794.5 (Pinned)
285 (CPU 0): 4447.4 (Pageable), 5050.7 (Pinned)	285 (CPU 0): 2992.6 (Pageable, Direct), 4841.8 (Pinned)
285 (CPU 1): 4593.8 (Pageable), 5726.0 (Pinned)	285 (CPU 1): 3184.9 (Pageable, Direct), 5541.1 (Pinned)
C2050 (CPU 0): 4350.2 (Pageable), 5744.3 (Pinned)	C2050 (CPU 0): 4315.4 (Pageable, Direct), 5547.9 (Pinned)
C2050 (CPU 1): 4301.8 (Pageable), 5012.6 (Pinned)	C2050 (CPU 1): 4296.0 (Pageable, Direct), 4799.6 (Pinned)
C1060 (CPU 0): 4707.3 (Pageable), 5727.4 (Pinned)	C1060 (CPU 0): 4735.8 (Pageable, Direct), 5560.9 (Pinned)
C1060 (CPU 1): 4555.5 (Pageable), 5012.3 (Pinned)	C1060 (CPU 1): 4536.9 (Pageable, Direct), 4827.4 (Pinned)
C2050 - device 3 (CPU 0): 4281.2 (Pageable), 5050.9 (Pinned)	C2050 - device 3 (CPU 0): 4323.5 (Pageable, Direct),
C2050 - device 3 (CPU 1): 4302.9 (Pageable), 5766.3 (Pinned)	4832.1 (Pinned)
	C2050 - device 3 (CPU 1): 4306.8 (Pageable, Direct),
	5559.8 (Pinned)
- Device to Host:	- Device to Host:
480 (CPU 0): 3200.7 (Pageable), 3438.3 (Pinned)	480 (CPU 0): 2377.0 (Pageable, Direct), 3438.1 (Pinned)
480 (CPU 1): 1968.8 (Pageable), 1991.3 (Pinned)	480 (CPU 1): 1596.8 (Pageable, Direct), 1991.9 (Pinned)
285 (CPU 0): 1967.3 (Pageable), 2002.9 (Pinned)	285 (CPU 0): 1533.5 (Pageable, Direct), 2002.3 (Pinned)
285 (CPU 1): 3221.6 (Pageable), 3438.4 (Pinned)	285 (CPU 1): 2355.7 (Pageable, Direct), 3436.5 (Pinned)
C2050 (CPU 0): 3180.4 (Pageable), 3437.9 (Pinned)	C2050 (CPU 0): 3353.0 (Pageable, Direct), 3437.9 (Pinned)
C2050 (CPU 1): 1956.9 (Pageable), 1987.7 (Pinned)	C2050 (CPU 1): 1957.4 (Pageable, Direct), 1987.9 (Pinned)
C1060 (CPU 0): 3201.4 (Pageable), 3438.0 (Pinned)	C1060 (CPU 0): 3206.4 (Pageable, Direct), 3435.6 (Pinned)
C1060 (CPU 1): 1951.6 (Pageable), 1987.9 (Pinned)	C1060 (CPU 1): 1949.3 (Pageable, Direct), 1986.9 (Pinned)
C2050 - device 3 (CPU 0): 1971.5 (Pageable), 2002.7 (Pinned)	C2050 - device 3 (CPU 0): 1971.1 (Pageable, Direct),
C2050 - device 3 (CPU 1): 3180.9 (Pageable), 3438.3 (Pinned)	2002.8 (Pinned)
	C2050 - device 3 (CPU 1): 3348.1 (Pageable, Direct),
	3438.3 (Pinned)

Wenn man sich die Bandbreitenwerte in Tabelle 11 ansieht, erkennt man, dass die Bandbreite unterschiedlich ist, je nachdem welche der zwei CPUs auf dem Mainboard dem Programm zugewiesen wird. Mit „numactl --cpunodebind=x“ (x=0 für CPU 1 und x=1 für CPU 2) vor „./program“, kann die jeweilige CPU für das Ausführen des Programms zugewiesen werden. Auf welchen Kern der ausgewählten CPU das Programm ausgeführt wird, ist dadurch nicht festgelegt. Beim Senden von Daten vom Host zum Device ist der Unterschied zwischen den zwei CPUs nur bei „Pinned“ groß. Wenn Daten vom Device zum Host gesendet werden, ist der Unterschied auch bei „Pageable“ groß. Bei zwei CPUs auf einem Mainboard ist die Wahl der richtigen CPU somit entscheidend für eine hohe Bandbreite zwischen CPU-DRAM und GPU-DRAM. Bei den Messungen hat sich herausgestellt, dass die höchste Bandbreite erreicht wird, wenn die erste CPU mit der ersten GPU und die zweite CPU mit

der zweiten GPU zusammenarbeitet. Bei drei GPUs sollte die erste CPU mit der ersten und zweiten GPU und die zweite CPU mit der dritten GPU zusammenarbeiten. Weist man keine CPU dem Programm zu, wird bei jedem Aufruf eine beliebige CPU zugewiesen und es kommt bei wiederholtem Ausführen des Programms zu Schwankungen bei der Laufzeit. Diese Schwankungen können bei Programmen, die viele Daten über den PCI-Bus transportieren, groß ausfallen. Sobald die CPU mit höherer PCI-Bandbreite zugewiesen ist, sind die Schwankungen selbst bei großen Datenübertragungen über den PCI-Bus sehr klein.

2.4 Architekturen

Bei Nvidia gibt es drei Architekturen (G80, GT200 und Fermi), die CUDA unterstützen [7]. In der Tabelle 12 werden die drei Architekturen verglichen. Seit November 2012 gibt es bei Nvidia mit Kepler eine weitere Architektur, die CUDA unterstützt [30]. Im Kapitel 11.6 befinden sich Informationen zu der Kepler-Architektur.

Bei den Architekturen G80 und GT200 können durch Dual-Issue MAD-Operationen auf den FPU's und MUL-Operationen auf den SFU's ausgeführt werden [16] [17]. Damit können 3 Operationen pro Takt pro Kern ausgeführt werden [16].

Tabelle 12: CUDA-Architekturen (erstellt auf Basis einer Tabelle von [7], Informationen von [7] [16] [17])

GPU-Architektur	G80	GT200	Fermi
Transistoren	681 Millionen	1,4 Milliarden	3 Milliarden
CUDA Kerne	128	240	512
Double-Precision FP	-	30 FMA Operationen pro Takt	256 FMA Operationen pro Takt
Single-Precision FP	128 MAD Operationen pro Takt 128 MUL Operationen pro Takt	240 MAD Operationen pro Takt 240 MUL Operationen pro Takt	512 FMA Operationen pro Takt
Warp Schedulers pro SM	1	1	2
SFU's pro SM	2	2	4
Shared Memory pro SM	16 KB	16 KB	Konfigurierbar 48 KB / 16 KB
L1-Cache pro SM	-	-	Konfigurierbar 16 KB / 48 KB
L2 Cache	-	-	768 KB
ECC	-	-	Ja
Concurrent Kernels	-	-	Bis zu 16

2.4.1 Compute Capability

Um das Maximum aus einem Kernel herausholen zu können, ist es wichtig die Limitierungen der verwendeten GPU zu kennen [1]. Diese Limitierungen werden pro Architektur standardisiert. Diese Standardisierung wird als Compute Capability bezeichnet [1]. Die Compute Capability besteht aus zwei Zahlen, die durch einen Punkt getrennt sind.

In der Tabelle 13 sieht man, dass sich GPUs nicht nur durch die Compute Capability unterscheiden. Weitere Unterschiede sind unter anderem die Anzahl der Multiprozessoren und ob es sich um eine GeForce- oder Tesla-GPU handelt.

Tabelle 13: GPU und Compute Capability [5] (Ausschnitt der Original-Tabelle, übersetzt und adaptiert)

GPU	Anzahl von Multiprozessoren	Compute Capability
GeForce 8800 GTX	16	1.0
GeForce 9800 GTX	16	1.1
GeForce GT 240	12	1.2
GeForce GTX 285	30	1.3
Tesla C1060	30	1.3
GeForce GTX 480	15	2.0
Tesla C2050	14	2.0
GeForce GTX 460	7	2.1

Tabelle 14: Compute Capability (erstellt auf Basis einer Tabelle von [5], Informationen von [1] [5] [7])

	CC 1.0	CC 1.1	CC 1.2	CC 1.3	CC 2.0	CC 2.1
Architektur	G80		GT200		Fermi	
GPUs	Tesla S870, Tesla C870, GeForce 8800 GTX, GeForce 8800 GTS	GeForce 9800 GT, GeForce 8800 GT, GeForce GT 130	GeForce GT 240, GeForce GT 220	Tesla C1060, GeForce GTX 285, GeForce GTX 260	Tesla C2050, GeForce GTX 480	GeForce GTX 460
Kerne pro SM	8	8	8	8	32	48
Maximale Anzahl von Threads pro Block	512	512	512	512	1024	1024
Maximale Grid Dimension (x,y,z)	65.535, 65.535, 1	65.535, 65.535, 1	65.535, 65.535, 1	65.535, 65.535, 1	65.535, 65.535, 65.535	65.535, 65.535, 65.535
Maximale Block Dimension (x,y,z)	512, 512, 64	512, 512, 64	512, 512, 64	512, 512, 64	1024, 1024, 64	1024, 1024, 64
Threads in einem Warp	32	32	32	32	32	32
4 Byte Register pro SM	8 K = 32 KB	8 K = 32 KB	16 K = 64 KB	16 K = 64 KB	32 K = 128 KB	32 K = 128 KB
Shared Memory pro SM	16 KB	16 KB	16 KB	16 KB	48 KB	48 KB
Bänke in Shared Memory	16	16	16	16	32	32
Constant Memory pro GPU	64 KB	64 KB	64 KB	64 KB	64 KB	64 KB
Constant Memory Cache pro SM	8 KB	8 KB	8 KB	8 KB	8 KB	8 KB
Lokaler Speicher pro Thread	16 KB	16 KB	16 KB	16 KB	512 KB	512 KB
Maximale Anzahl von aktiven Blöcken pro SM	8	8	8	8	8	8
Maximale Anzahl von aktiven Warps pro SM	24	24	32	32	48	48
Maximale Anzahl von aktiven Threads pro SM	768	768	1024	1024	1536	1536
Double-Precision Floating-Point	Nein	Nein	Nein	Ja	Ja	Ja

In der Tabelle 14 werden die Compute Capabilities dargestellt. Zum Beispiel mit Compute Capability 1.2 wurde die Anzahl der 4-Byte Register von 8192 auf 16384 und die aktive Threadanzahl pro Multiprozessor von 768 auf 1024 erhöht [1]. Seit Compute Capability 1.3 sind Double-Precision Floating-Point-Operationen möglich [1].

2.4.2 Änderungen bei Fermi gegenüber der Vorgängerarchitektur GT200

2.4.2.1 40 Bit Unified Address Space statt drei GPU-Adressräume

Bei den Vorgängerarchitekturen hat es nur einen 32 Bit physikalischen Speicheradressraum gegeben, wodurch der DRAM maximal 4 GB ($=2^{32}$) groß sein durfte [1]. Mit dem Vergrößern auf 40 Bit ist nun ein DRAM mit bis zu 1 TB ($=2^{40}$) Größe möglich.

Ein weiterer Unterschied ist, dass es sich bei dem 40-Bit physikalischen Speicheradressraum um einen Unified Address Space handelt [1]. Bei den Vorgängerarchitekturen hatte der Shared Memory, der Local Memory und der Global Memory einen eigenen Adressraum [15]. In der Fermi-Architektur gibt es für alle drei Speichertypen nur mehr einen Unified Address Space [15]. Der Vorteil ist, dass man einen Pointer einer Funktion übergeben kann und es irrelevant ist, auf welchen Speichertyp der Pointer zeigt [1]. Beim Programmieren muss nur beim Anlegen des Speichers der Speichertyp angegeben werden [1]. Der einzige Unterschied liegt nur mehr in der Verarbeitungsgeschwindigkeit [1]. Wenn der übergebene Pointer auf ein Element im Shared Memory zeigt, geht die Verarbeitung wesentlich schneller, als wenn der Pointer auf ein Element im Global Memory zeigt [1]. Durch diese Vereinfachung können Bibliotheken/Funktionen für CUDA mit weniger Aufwand produziert werden [1].

2.4.2.2 Unified Virtual Addressing (64 Bit gemeinsamer virtueller CPU- und GPU-Speicheradressraum)

Mit Fermi kann der physikalische Adressraum der CPU und der GPU in einem gemeinsamen virtuellen Speicheradressraum zusammengefasst werden [1]. Somit haben alle Variablen in der CPU/GPU-Anwendung eine eindeutige Adresse [1].

Der gemeinsame Adressraum von CPU und GPU ist in der Fermi-Architektur implementiert, wurde aber erst mit CUDA-Toolkit 4.0 verfügbar (seit Mai 2011) [5]. In [31] wurde das Feature mit dem „Direct N-Body“-Algorithmus erfolgreich getestet, indem die GPUs bei der Ausführung des Kernels auf die Inputdaten im Arbeitsspeicher der CPU zugreifen, ohne dass ein zusätzlicher Device-Pointer für das Host-Array erzeugt werden muss.

Eine weitere Vereinfachung ist, dass bei der Verwendung der cudaMemcpy-Funktion beim letzten Parameter nicht mehr die Richtung angegeben werden muss [31]. Statt cudaMemcpyHostToDevice oder cudaMemcpyDeviceToHost kann nun cudaMemcpyDefault verwendet werden [31].

Das UVA (=Unified Virtual Addressing) vereinfacht auch die Transfers zwischen GPUs. Sobald mit der Funktion cudaDeviceEnablePeerAccess() der Speicherzugriff zu einer anderen GPU aktiviert wurde, kann beim Kernelaufufruf ohne Probleme ein Pointer, der auf ein Global-Memory-Array der anderen GPU zeigt, übergeben werden (siehe Code 4) [5]. Der erste Parameter der Funktion cudaDeviceEnablePeerAccess() gibt die GPU an, auf deren Global Memory ein direkter Zugriff von der aktuell ausgewählten GPU ermöglicht werden soll [31]. Der zweite Parameter ist immer 0 [31]. Der direkte Datentransfer zwischen GPUs funktioniert nur, wenn sie mit demselben IOH-Chip auf dem Mainboard verbunden sind [32]. Mit dem Codebeispiel „simpleP2P Sample“ vom Nvidia SDK kann der direkte Datentransfer zwischen GPUs getestet werden [33].

Code 4: Zugriff einer GPU auf ein Array im Global Memory einer anderen GPU [5]

```
cudaSetDevice(0);
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);
MyKernel<<<1000,128>>>(p0);

cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0,0);
MyKernel<<<1000,128>>>(p0);
```

Seit dem Toolkit 2.2 gibt es das Feature Zero-Copy, mit dem es möglich ist, direkt von der GPU aus auf Daten im Arbeitsspeicher der CPU zuzugreifen [1]. Vor der Architektur Fermi musste dazu ein Device-Pointer über die Funktion `cudaHostGetDevicePointer()` ermittelt werden (siehe dritten Befehl im Code 5) [5]. Damit die Funktion `cudaHostGetDevicePointer()` den Device-Pointer und nicht einen Fehler zurückliefert, muss `cudaSetDeviceFlags(cudaDeviceMapHost)` vor allen anderen CUDA-Befehlen ausgeführt werden (siehe ersten Befehl im Code 5) [5]. Beim Erzeugen des Arrays im Arbeitsspeicher der CPU muss die Funktion `cudaHostAlloc()` mit dem Flag „`cudaHostAllocMapped`“ verwendet werden (siehe zweiten Befehl im Code 5) [5]. Dadurch wird ein Mapping des Page-locked-Arrays in den Adressraum des Device durchgeführt [5]. Das Array hat dann eine Host-Adresse (`a_h` im Code 5) und eine Device-Adresse (`a_d` im Code 5) [5].

Code 5: Zero-Copy [34]

```
cudaSetDeviceFlags(cudaDeviceMapHost);
cudaHostAlloc((void**)&a_h, nBytes, cudaHostAllocMapped);
cudaHostGetDevicePointer((void**)&a_d, (void *)a_h, 0);
for (i=0; i<N; i++)
    a_h[i] = i;
increment<<<grid, block>>>(a_d, N);
```

2.4.2.3 L1-Caches und L2-Cache

Vor der Fermi Architektur wurde Shared Memory benutzt, um lokale, vorhersagbare Zugriffsmuster auszunützen und damit die Performance des Programms deutlich zu erhöhen [1]. Mit Fermi gibt es zusätzlich zum Shared Memory auf jedem Multiprozessor einen L1-Cache [5]. Außerdem gibt es einen großen L2-Cache, der von allen Multiprozessoren gemeinsam genutzt wird [5]. Mit den Caches können nun auch nicht vorhersagbare Zugriffsmuster bzw. einfache auf Global Memory portierte Programmcodes, also ohne Ausnützung des Shared Memory, deutlich beschleunigt werden [1]. Shared Memory und die Caches sind von der Zugriffszeit deutlich schneller als der DRAM, da sie sich auf dem Chip befinden. Mit den neuen Caches werden nicht nur die Daten des Global Memory für einen schnelleren Zugriff zwischengespeichert, sondern auch die des Local Memory [5]. Da sich der Local Memory im DRAM befindet, bringt der Cache hier deutliche Verbesserung. Ebenfalls wurde durch die Caches der Einfluss von Memory Coalescing auf die Performance deutlich reduziert. Die Optimierung „Memory Coalescing“ hat mit der Einführung von Fermi stark an Wichtigkeit verloren [1]. Dadurch wird die Transformation von CPU-Code in einen schnelleren GPU-Code weiter erleichtert [1].

Man kann im Programmcode festlegen, ob man 16 KB oder 48 KB L1-Cache pro Multiprozessor haben möchte [5]. Je nachdem hat man dann 48 KB oder 16 KB Shared Memory pro Multiprozessor zur Verfügung [5]. Im Host-Code in die `main()`-Methode wird dazu „`cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)`“ eingefügt [5]. Damit Shared Memory auf 48 KB und der L1-Cache auf 16 KB gesetzt wird, muss „`cudaFuncCachePreferShared`“ als zweites Argument verwendet werden [5]. Für 16 KB Shared Memory und 48 KB L1-Cache wird das Argument „`cudaFuncCachePreferL1`“ benutzt [5]. Mit dem Argument „`cudaFuncCachePreferNone`“ wird die Voreinstellung genommen [5]. Die Voreinstellung ist jene Einstellung, die zuletzt am häufigsten benutzt wurde [5]. Wenn jedoch 48 KB Shared Memory pro Multiprozessor für die Ausführung des Kernels benötigt wird, dann wird

automatisch die Einstellung „cudaFuncCachePreferShared“ verwendet [5]. Das ist auch die Standard-Einstellung der Fermi-GPUs [5].

Durch Übergabe des Flags „-Xptxas -dlcm=cg“ beim Kompilieren mit dem nvcc-Kompiler kann festgelegt werden, dass die Global-Memory-Zugriffe nur im L2-Cache zwischengespeichert werden [5]. Falls dieses Flag nicht angegeben wird, werden Werte auch in den L1-Caches zwischengespeichert [5].

Eine Cache-Line ist 128 Bytes groß und enthält ein 128-byte aligned Segment aus dem Global Memory [5]. Wenn die L1-Caches aktiviert sind, werden 128 Byte Transaktionen auf die Speicher durchgeführt [5]. Wenn jedoch die L1-Caches durch das Setzen des Flags deaktiviert werden, dann werden nur 32 Byte Transaktionen ausgeführt [5]. Das hat nur dann den Vorteil, wenn der Kernel kaum coalesced-Speicherzugriffe hat, also die Threads auf Daten zugreifen, die nicht nebeneinander im Speicher liegen und es somit ständig mit den 128 Byte Zugriffen zu einem „over-fetch“ kommt [5].

Wenn die Threads eines Warps (=32 Threads) auf 8 Byte (=double) Datenelemente zugreifen, werden zwei 128 Byte Memory Request erzeugt [5]. Bei 4 Byte (=float) muss nur ein 128 Byte Memory Request ausgeführt werden [5], falls die Threads auf Daten zugreifen, die nebeneinander im Speicher liegen.

2.4.2.4 Multiprozessoren und Kerne

Die Anzahl der Kerne pro Multiprozessor wurde von 8 auf 32 erhöht [5]. Die Anzahl der Multiprozessoren wurde hingegen von 30 (=GTX 285) auf 16 (=GTX 580) reduziert [5].

2.4.2.5 SFUs

Die Anzahl der SFUs (=Special Function Units) wurde von 2 auf 4 pro Multiprozessor erhöht [5].

2.4.2.6 Warp Scheduler

Die Anzahl der Warp Scheduler pro Multiprozessor wurde von 1 auf 2 erhöht [5]. Ein Warp Scheduler kümmert sich um die Ausführung der Instruktionen der aktiven Warps mit ungerader Warp-ID (z. B. Warp 1), der andere Warp Scheduler führt die Warps mit gerader Warp-ID (z. B. Warp 2) aus [5]. Damit sind zwei unterschiedliche Instruktionen pro Takt auf einem Multiprozessor möglich. Wenn ein Warp Scheduler eine Floating-Point-Operation mit dem Datentyp double anstatt des Datentyps float ausführt, dann kann nur ein Warp Scheduler zu einem Zeitpunkt eine Instruktion ausführen [5]. Ein Warp Scheduler kann eine Instruktion nur auf der Hälfte der Kerne eines Multiprozessors ausführen [5]. Somit benötigt das Ausführen einer Integer- oder Floating-Point-Instruktion für den gesamten Warp (=32 Threads) 2 Taktzyklen [5].

2.4.2.7 Aktive Threads

Die maximale Anzahl der aktiven Threads pro Multiprozessor wurde von 1024 auf 1536 erhöht [5]. Da jedoch die Anzahl der Multiprozessoren von 30 (=GTX 285) auf 16 (=GTX 580) reduziert wurde, ist die Anzahl der aktiven Threads bei der Fermi-Architektur mit 24576 (=16*1536) kleiner als bei der GT200-Architektur mit 30720 (=30*1024) [5] [7].

2.4.2.8 Register Memory

Die Größe des Speichers für die Register wurde von 64 KB auf 128 KB pro Multiprozessor erhöht [5]. Wenn man den Float-Datentyp verwendet und die maximale Anzahl von gleichzeitig aktiven Threads von 1536 Threads (=48 Warps*32 Threads) pro Multiprozessor verwenden möchte, kann man nun 21 (=128*1024/4/1536) Register pro Thread belegen [5]. Bei der Vorgängerarchitektur GT200 können bei Nutzung von maximal 1024 aktiven Threads (=32 Warps*32 Threads) 16 (=64*1024/4/1024)

Register pro Thread verwendet werden [5] [7]. Somit wurde mit Fermi die Anzahl der aktiven Threads pro Multiprozessor und die Anzahl der pro Thread zur Verfügung stehenden Register erhöht.

2.4.2.9 Shared Memory

Mit Fermi wurde pro Multiprozessor die Größe des Shared Memory von 16 KB auf 48 KB und die Anzahl der Shared-Memory-Bänke von 16 auf 32 erhöht [5]. Die Daten werden im Shared Memory so gespeichert, dass die ersten 4 Bytes in die erste Bank gespeichert werden, die nächsten 4 Bytes in die zweite Bank und so weiter [5]. Sobald die letzte Bank erreicht ist, wird wieder mit dem Schreiben in die erste Bank begonnen [5]. Jede Bank hat eine Bandbreite von 4 Bytes pro 2 Taktzyklen [5].

Bei der Fermi-Architektur wird aufgrund der 32 Bänke ein Shared-Memory-Request für einen Warp nicht mehr in zwei Memory Requests geteilt [5]. Dadurch kann es mit Fermi zu einem Bank-Konflikt zwischen einem Thread von der ersten Hälfte des Warps mit einem Thread aus der zweiten Hälfte des Warps kommen [5]. Zu einem Bank-Konflikt kommt es, wenn mehrere Threads auf unterschiedliche 4 Byte Wörter innerhalb einer Bank zugreifen [5]. Wenn jedoch mehrere Threads auf unterschiedliche Bytes innerhalb eines 4 Byte Wortes zugreifen (siehe Code 6), gibt es bei Fermi keinen Bank-Konflikt [5]. Es wird ein Broadcast des Wortes zu den Threads durchgeführt [5]. In den Vorgängerarchitekturen hat dieser Fall noch zu einem Bank-Konflikt geführt [5].

Code 6: Zugriff mehrerer Threads auf unterschiedliche Bytes der 4-Byte-Shared-Memory-Wörter [5]

```
__shared__ char shared[32];  
char data = shared[BaseIndex + tid];
```

Wenn mehrere Threads schreibend auf die einzelnen Bytes innerhalb eines 4 Byte Wortes zugreifen, kann nur jeweils für jedes Byte ein Thread den Schreibvorgang durchführen [5]. Welcher Thread das sein wird, ist undefiniert [5].

Der ideale Shared-Memory-Zugriff erfolgt mit einem 32-Bit Strided Access (siehe Code 7) [5]. Da der Float-Datentyp (=32 Bit, 4 Byte) verwendet wird, ist der ideale stride im Code 7 die Zahl 1. Alle 16 Threads eines halben Warps (GT200) bzw. 32 Threads eines Warps (Fermi) greifen dadurch auf unterschiedliche Bänke zu und es gibt keinen Bank-Konflikt.

Code 7: Idealer Shared-Memory-Zugriff (stride s=1) [5]

```
__shared__ float shared[32];  
float data = shared[BaseIndex + s * tid];
```

2.4.2.10 Rekursion / Funktionsaufrufe innerhalb eines Kernel

Mit Fermi ist es erstmals möglich, dass zur Laufzeit Funktionen vom Kernel aus aufgerufen werden können [1]. In den Vorgängerarchitekturen konnten zwar im Source Code auch Funktionsaufrufe eingefügt werden, jedoch musste dann der Compiler den Inhalt der aufgerufenen Funktionen in den Kernel einbauen [1]. Mit Fermi sind nun virtuelle Funktionen, dynamisch gelinkte Bibliotheken und Rekursionen möglich [1]. Das ist ein wichtiger Schritt zur vollständigen Unterstützung von C++ im Kernelcode [1]. Durch die Einführung der Möglichkeit von Rekursionen wird neben den Caches eine weitere Erleichterung angeboten, um CPU-Programme in kurzer Zeit in GPU-Programme umwandeln zu können [1].

Kernels (=Funktionen mit der Deklaration `__global__` in CUDA) selbst dürfen nicht rekursiv aufgerufen werden, aber eine Device-Funktion (=Funktionen mit der Deklaration `__device__` in CUDA), die vom Kernel aus aufgerufen wird, darf sich rekursiv aufrufen [5].

Damit eine Rekursion verwendet werden kann, muss „-arch=sm_20“ beim Kompilieren angegeben werden, da eine Rekursion erst ab Compute Capability 2.0 unterstützt wird [5]. "sm_20" steht für

Compute Capability 2.0 [5]. Wird das Flag „-arch=sm_20“ nicht verwendet, versucht der nvcc-Compiler das Programm so zu kompilieren, dass es auch auf Grafikkarten mit älterer Compute Capability ausgeführt werden kann [5]. Ist eine solche Anpassung nicht möglich, gibt der Compiler eine Fehlermeldung aus.

2.4.2.11 printf() im Kernel

Mit Fermi kann mit printf() im Kernel einfaches Debugging durchgeführt und Hello-World-Programme geschrieben werden (siehe Code 8) [5]. Damit printf() verwendet werden kann, muss „-arch=sm_20“ beim Kompilieren angegeben werden, da printf() erst ab Compute Capability 2.0 unterstützt wird [5].

Code 8: printf() im Kernel [5]

```
__global__ void helloCUDA(float f) {
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

void main() {
    helloCUDA<<<1, 5>>>>(1.2345f); cudaDeviceReset();
}
```

2.4.2.12 malloc() und free() im Kernel

Mit malloc() kann man im Kernel dynamische Arrays im Heap des Global Memory erzeugen [5]. Die angelegten Arrays bleiben während der gesamten Lebenszeit des CUDA-Context erhalten [5]. Erst mit dem Aufruf der Funktion free() wird das Array freigegeben [5]. Damit malloc() und free() verwendet werden können, muss „-arch=sm_20“ beim Kompilieren angegeben werden, da malloc() und free() erst ab Compute Capability 2.0 unterstützt werden [5].

Der Default Heap ist 8 MB groß [5]. Die Größe kann mit dem Aufrufen der Funktion „cudaDeviceSetLimit(cudaLimitMallocHeapSize, size_t size)“ in der main()-Methode im Host-Code verändert werden [5].

Im Code 9 wird die Verwendung von malloc() und free() in einem Beispiel dargestellt. Im Host-Code wird in der main()-Methode die Heap-Größe von 8 MB auf 128 MB erhöht und es werden fünf Threads gestartet. Jeder Thread führt den Kernelcode aus und legt dabei ein char-Array mit 123 Werten an, gibt die Threadnummer und den Pointer aus und gibt den Speicher des char-Arrays wieder frei.

Code 9: malloc() und free() im Kernel [5]

```
#include <stdlib.h>
#include <stdio.h>

__global__ void mallocTest() {
    char* ptr = (char*)malloc(123);
    printf("Thread %d got pointer: %p\n", threadIdx.x, ptr);
    free(ptr);
}

void main() {
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);
    mallocTest<<<1, 5>>>>();
    cudaDeviceSynchronize();
}
```

2.4.2.13 Schnelleres Application Context Switching

Mit Application Context Switching wird Multitasking wie auf der CPU realisiert [15]. Die Ausführung von Programmen wird nach einer bestimmten Zeitdauer gewechselt [15]. Ein Wechsel zwischen Programmen ist mit Fermi mit einem Aufwand von unter 25 Mikrosekunden 10 Mal schneller als bei der Vorgängerarchitektur GT200 [15].

Mit den zwei folgenden Demo-Programmen wird gezeigt, dass Application Context Switching nur nach jeder Kernelausführung möglich ist. Demo-Programm 1 (siehe Code 10) zeigt, dass während des Ausführens eines Kernels (Endlos-Schleife im Kernel) kein Switching möglich ist. Mit Demo-Programm 2 (siehe Code 11) wird der Beweis erbracht, dass das Switching nach jeder Kernelausführung funktioniert (Endlos-Schleife über Kernelaufruf).

Code 10: Demo-Programm 1 (Endlosschleife im Kernel, Versuch des Application Context Switching während der Kernelausführung)

```
__global__ void Kernel(int i)
{
    printf("Anwendung %d\n",i);
}
__global__ void KernelEndlosschleife(int i)
{
    while(true){
        printf("Anwendung %d\n",i); //Diese Zeile kann auskommentiert werden,
        //Endlosschleife funktioniert dennoch
    }
}
int main(int argc, char **argv){
    int device_nr=0;
    if(grafikkarteMitCUDAUnterstuetzungAuswaehlen(device_nr)!=0)
        return EXIT_FAILURE;

    if(argc<=1){
        printf("Usage: programm1 <ID der Applikation>\n");
        return EXIT_FAILURE;
    }
    int applikationID = atoi(argv[1]);

    Kernel<<<1,1>>>>(applikationID); //1 Block mit einem Thread
    cudaThreadSynchronize();

    KernelEndlosschleife<<<1,1>>>>(applikationID); //1 Block mit einem Thread
    cudaThreadSynchronize();

    printf("Kernel wurde verlassen!\n");

    return EXIT_SUCCESS;
}
```

Terminalausgabe 1: Versuch des Application Context Switching während der Kernelausführung

Erstes Terminal und erster Programmaufruf:	Zweites Terminal und zweiter Programmaufruf:
\$./programm 1 Anwendung 1	\$./programm 2 Kernel wurde verlassen! \$

Beim ersten Aufruf des Programms wird der erste Kernel erfolgreich ausgeführt und gibt Anwendung 1 aus (siehe Terminalausgabe 1). Nachdem der erste Kernel vollständig auf der GPU ausgeführt wurde, wird der zweite Kernel im Programm auf der GPU gestartet. Da dieser Kernel eine Endlosschleife beinhaltet, dauert das Ausführen des Kernels endlos. Nun wird in einem zweiten Terminal dasselbe Programm nochmals ausgeführt. Da das Programm im ersten Terminal durch die Endlosschleife im zweiten Kernel noch immer die ganze GPU komplett blockiert, kann beim Programm im zweiten Terminal kein Kernel ausgeführt werden. Es wurde festgestellt, dass im Programmcode auf die Ausführung eines Kernels verzichtet wird, wenn ein Kernel nach 15 Sekunden noch nicht auf der GPU gestartet worden ist. Das Programm im zweiten Terminal wird automatisch beendet (siehe Terminalausgabe 1).

Code 11: Demo-Programm 2 (Endlosschleife im Host auf Kernelaufruf, Application Context Switching nach jeder Kernelausführung)

```
int main(int argc, char **argv)
{
    int device_nr=0;
    if(grafikkarteMitCUDAUnterstuetzungAuswaehlen(device_nr)!=0)
        return EXIT_FAILURE;

    if(argc<=1){
        printf("Usage: programm1 <ID der Applikation>\n");
        return EXIT_FAILURE;
    }
    int applikationID = atoi(argv[1]);

    while(true){
        Kernel<<<8*15,1>>>(applikationID);
    }
    cudaThreadSynchronize();

    return EXIT_SUCCESS;
}

__global__ void Kernel(int i)
{
    printf("Anwendung %d\n",i);
}
```

Terminalausgabe 2: Application Context Switching nach jeder Kernelausführung

Erstes Terminal und erster Programmaufruf:	Zweites Terminal und zweiter Programmaufruf:
<code>\$./programm 1</code> Anwendung 1 Anwendung 1 Anwend	<code>\$./programm 2</code> Anwendung 2 Anwendung 2 Anwend

Das Programm kann ohne Probleme gleichzeitig im Terminal 1 und im Terminal 2 ausgeführt werden (siehe Terminalausgabe 2). Nach jeder Kernelausführung wird zwischen dem Programm im Terminal 1 und dem Programm im Terminal 2 gewechselt. Dadurch wird zunächst im ersten Terminal durch Ausführen des Kernels „Anwendung 1“ ausgegeben. Nach dem Wechsel des Programms wird im Terminal 2 „Anwendung 2“ ausgegeben. Danach erfolgt wieder eine Ausgabe im Terminal 1. Die Ausgabe in den zwei Terminals wechselt sich ständig ab.

2.4.2.14 Gleichzeitiges Ausführen mehrerer Kernels auf einer GPU

Mit Fermi kann erstmals mehr als 1 Kernel gleichzeitig auf einer GPU ausgeführt werden [1]. Die Anzahl der maximal gleichzeitig laufenden Kernels wurde von 1 auf 16 erhöht [5]. Wenn beim Ausführen eines Kernels die GPU nicht vollständig ausgelastet wird, kann durch das parallele Ausführen von weiteren Kernels bereits Arbeit von diesen Kernels erledigt und dadurch Laufzeit eingespart werden.

Die gleichzeitig laufenden Kernels müssen von demselben CUDA-Context stammen [5]. Eine Anwendung belegt somit immer die ganze GPU. Mehrere Anwendungen können sich nur durch Application Context Switching die GPU teilen (siehe vorhergehendes Kapitel 2.4.2.13). Paralleles Ausführen von mehreren Anwendungen zur gleichen Zeit ist auch mit Fermi nicht möglich.

Mit Code 12 (Hostcode) und Code 13 (Kernelcode) wird ein Beispiel gezeigt, wie ein paralleles Ausführen mehrerer Kernels im Code ermöglicht werden kann. Es ist unbedingt notwendig, dass alle Kernels einen anderen Stream zugewiesen werden [5]. Dazu werden im ersten Schritt die notwendigen Streams mit der Methode „cudaStreamCreate()“ erzeugt (siehe Code 12) [5]. Danach

kann ein Kernelaufruf einem Stream zugewiesen werden. Dazu wird beim vierten Parameter der erzeugte Stream übergeben [5]. Wird der Kernelaufruf mit nur 2 oder 3 Parametern (3. Parameter für Größe des dynamischen Shared Memory pro Block) verwendet oder der Wert 0 im vierten Parameter übergeben, so wird der Kernelaufruf automatisch dem Default-Stream zugewiesen [5]. Das geschieht bei den sequentiell aufgerufenen Kernels. Die dem Default-Stream zugewiesenen Kernelaufufe werden nacheinander abgearbeitet. Mit der Funktion „cudaStreamDestroy()“ werden am Ende des Programms die erzeugten Streams gelöscht (siehe Code 12) [5].

Der Kernel wird im Code 12 immer mit nur einem Thread ausgeführt. Die Aufgabe des Kernels ist es, 10 Mal nacheinander die printf()-Funktion aufzurufen (siehe Code 13). Mit jedem Aufruf der printf()-Funktion wird die beim Kernelaufruf übergebene Zahl inklusive dem Wort „Kernel“ im Terminal ausgegeben.

Code 12: Hostcode des Concurrent-Kernel-Programms [5] (adaptiert)

```
int anzahl_paralleler_Kernels=4;
cudaStream_t stream[anzahl_paralleler_Kernels];
for(int i=0;i<anzahl_paralleler_Kernels;i++)
    cudaStreamCreate(&stream[i]);

printf("\nSequentieller Aufruf der Kernels\n");
Kernel<<<1,1>>>(1); //1 Block mit einem Thread
Kernel<<<1,1>>>(2); //1 Block mit einem Thread
Kernel<<<1,1>>>(3); //1 Block mit einem Thread
Kernel<<<1,1>>>(4); //1 Block mit einem Thread
cudaThreadSynchronize();

printf("\nConcurrent Kernels\n");
Kernel<<<1,1,0,stream[0]>>>(1); //1 Block mit einem Thread
Kernel<<<1,1,0,stream[1]>>>(2); //1 Block mit einem Thread
Kernel<<<1,1,0,stream[2]>>>(3); //1 Block mit einem Thread
Kernel<<<1,1,0,stream[3]>>>(4); //1 Block mit einem Thread
cudaThreadSynchronize();

//Streams loeschen
for(int i=0;i<anzahl_paralleler_Kernels;i++)
    cudaStreamDestroy(stream[i]);
```

Code 13: Kernel des Concurrent-Kernel-Programms

```
__global__ void Kernel(int i){
    for(int s=0;s<10;s++)
        printf("Kernel %d\n",i);
}
```

Die Ausgabe im Terminal (siehe Terminalausgabe 3) ist wie erwartet. Bei Concurrent Kernels können die Kernels parallel ausgeführt werden. Dadurch wird der Code der Kernels gleichzeitig ausgeführt und die Ausgabe der Kernels ist durcheinander gemischt. Bei den sequentiell aufgerufenen Kernels wird jeder Kernel nacheinander abgearbeitet und dadurch stehen die Zeichenketten der zehn printf()-Funktionen eines Kernels nacheinander im Terminal.

Terminalausgabe 3: Concurrent Kernels

Sequentieller Aufruf der Kernels	Concurrent Kernels
Kernel 1	Kernel 1
Kernel 1	Kernel 2
Kernel 1	Kernel 1
Kernel 1	Kernel 3
Kernel 1	Kernel 2
Kernel 1	Kernel 1
Kernel 1	Kernel 3
Kernel 1	Kernel 4
Kernel 1	Kernel 2
Kernel 1	Kernel 1
Kernel 2	Kernel 3
Kernel 2	Kernel 2
Kernel 2	Kernel 4
Kernel 2	Kernel 1
Kernel 2	Kernel 3
Kernel 2	Kernel 2
Kernel 2	Kernel 4
Kernel 2	Kernel 1
Kernel 2	Kernel 3
Kernel 2	Kernel 2
Kernel 3	Kernel 4
Kernel 3	Kernel 1
Kernel 3	Kernel 3
Kernel 3	Kernel 2
Kernel 3	Kernel 4
Kernel 3	Kernel 1
Kernel 3	Kernel 3
Kernel 3	Kernel 2
Kernel 3	Kernel 4
Kernel 3	Kernel 1
Kernel 4	Kernel 3
Kernel 4	Kernel 2
Kernel 4	Kernel 4
Kernel 4	Kernel 1
Kernel 4	Kernel 3
Kernel 4	Kernel 2
Kernel 4	Kernel 4
Kernel 4	Kernel 3
Kernel 4	Kernel 4
Kernel 4	Kernel 4

Es gibt auch ein Demo-Programm („concurrentKernels“) [28] im SDK von Nvidia, das mit Laufzeitmessung beweist, dass Concurrent Kernels funktioniert. Das Ergebnis des Programms ist (siehe Terminalausgabe 4), dass 8 parallel ausgeführte Kernels mit sehr niedriger GPU-Auslastung statt 0.08 Sekunden (=sequentielle Ausführung) nur 0.01 Sekunden benötigen.

Terminalausgabe 4: Nvidia-Programm concurrentKernels

```
$ ./concurrentKernels
[ concurrentKernels ]
> Using CUDA device [0]: GeForce GTX 480
> Detected Compute SM 2.0 hardware with 15 multi-processors
Expected time for serial execution of 8 kernels = 0.080s
Expected time for concurrent execution of 8 kernels = 0.010s
Measured time for sample = 0.010s

Press <Enter> to Quit...
-----
```

2.4.2.15 Deutliche Erhöhung der Double-Precision-Peak-Leistung

Mit Fermi wurde die Double-Precision-Peak-Leistung deutlich erhöht [1]. Bei der GT200-Architektur war die Double-Precision-Peak-Leistung 12 Mal kleiner als die Single-Precision-Peak-Leistung. Mit den Tesla-GPUs der Fermi-Architektur ist die Double-Precision-Peak-Leistung nur mehr um die Hälfte kleiner als die Single-Precision-Peak-Leistung [35]. Das entspricht demselben Verhältnis wie bei CPUs [8]. Bei den GeForce-GPUs der Fermi-Architektur ist der Unterschied mit einem Faktor von 8 [19] zwar kleiner als bei der Vorgängerarchitektur GT200 mit einem Faktor von 12 [36], jedoch deutlich schlechter im Vergleich zu den Tesla-GPUs der Fermi-Architektur mit einem Faktor von 2 [35].

2.4.2.16 Verbesserung der Genauigkeit bei Floating-Point

Mit Fermi wurde die Genauigkeit von drei Single-Precision-Funktionen (sqrtf(), x/y und 1/x) verbessert, es wird der überarbeitete IEEE 754 Floating-Point-Standard aus dem Jahr 2008 verwendet und es werden denormalisierte Zahlen unterstützt [5]. Um die Verbesserung der Genauigkeit bei Floating-Point besser verstehen zu können, wird in den folgenden Absätzen erklärt, wie eine Floating-Point-Zahl gespeichert ist.

Im Jahr 1985 wurde mit dem IEEE 754 Floating-Point-Standard eine einheitliche Repräsentation und ein einheitliches arithmetisches Verhalten von Floating-Point-Zahlen geschaffen [1]. Im Jahr 2008 wurde der Standard überarbeitet [1].

Im IEEE Standard wird eine Floating-Point-Zahl mit drei Teilen (Vorzeichen, Exponent und Mantisse) dargestellt (siehe Formel 2) [1].

Formel 2: IEEE Standard Floating-Point-Zahl (Vorzeichen S, Mantisse M und Exponent E) [1] (Wörter übersetzt)

Wert = $(-1)^S \times M \times 2^E$, wobei $1.0 \leq M < 2.0$

Das Vorzeichen S kann den Wert 0 (=positive Zahl) oder 1 (=negative Zahl) annehmen [1].

Der Exponent E gibt das Intervall an, in dem die Zahl liegt [1]. Wenn E gleich 64 ist, dann befindet sich die zu sichernde Zahl zwischen 2^{64} ($> 10^{18}$) und 2^{65} [1]. Wenn E gleich -64 ist, dann wird eine Zahl zwischen 2^{-64} ($< 10^{-18}$) und 2^{-63} dargestellt [1]. Der IEEE Standard verwendet die Excess-Repräsentation für den Exponent [1]. Die andere mögliche Repräsentation ist „Two's Complement“. Zum Beispiel entspricht die Zahl „1“ in binärer und „Two's Complement“-Schreibweise „001“. Um das negative Gegenstück (-1) in der „Two's Complement“-Repräsentation darzustellen, werden alle 1 zu 0 und alle 0 zu 1 und es wird 1 dazu addiert [1]. 001 wird zu 110 und mit 1 dazu addiert zu 111 [1]. Bei der „Excess“-Repräsentation wird $(2^{n-1}-1)$ zur „Two's Complement“-Repräsentation dazu addiert [1]. Die Zahl „-1“ entspricht „111“ in der „Two's Complement“-Repräsentation und mit der Addition von 011 erhält man die Zahl 010 (siehe Tabelle 15) [1]. Der große Vorteil der Excess-Darstellung ist, dass die niedrigste Zahl den kleinsten Code und die größte Zahl den größten Code hat (siehe Tabelle 16) [1].

Damit kann ein Vergleich ohne Berücksichtigung des Vorzeichens mit der Excess-Darstellung Zahlen mit unterschiedlichen Vorzeichen vergleichen [1]. Dieser Vorteil macht den Vergleich von Zahlen schneller [1].

Tabelle 15: Two's Complement und Excess Repräsentation (sortiert nach den Werten von Spalte Two's Complement) [1] (Spaltenüberschriften angepasst)

Two's Complement	Dezimalwert	Excess
000	0	011
001	1	100
010	2	101
011	3	110
100	Reserved pattern	111
101	-3	000
110	-2	001
111	-1	010

Tabelle 16: Two's Complement und Excess Repräsentation (sortiert nach den Werten von Spalte Dezimalwert) [1] (Spaltenüberschriften und Zelle reserved pattern angepasst)

Two's Complement	Dezimalwert	Excess
100	Reserved pattern	111
101	-3	000
110	-2	001
111	-1	010
000	0	011
001	1	100
010	2	101
011	3	110

Da alle Mantissen mit „1.“ beginnen, kann es von der Repräsentation entfernt werden und man erspart sich ein Bit [1]. Wenn man sich die Mantisse im Dezimalsystem ansieht, dann muss sie größer gleich als 1 und kleiner als 10 sein [1]. Wenn diese Einschränkung zutrifft, dann ist die Mantisse normalisiert und korrekt [1].

Die Größe der Mantisse legt die Anzahl der möglichen Zahlen innerhalb des vom Exponenten gewählten Intervalls fest [1]. Die Anzahl der Bits, die für die Mantisse verwendet werden, legt somit die Genauigkeit der Repräsentation fest [1]. Mit einer N-Bit Mantisse hat man 2^N Zahlen innerhalb des Intervalls [1]. Die Ergebniszahl einer Berechnung wird für die Speicherung auf die darstellbare Zahl gerundet, die am nächsten liegt [1]. Je größer die Mantisse ist, desto genauer kann die Zahl von der Repräsentation erfasst werden [1].

Es gibt drei unterschiedliche Formate. Die Formate heißen „No-Zero“- , „Abrupt Underflow“- und „Denormalization“-Format [1]. Diese Formate werden am Beispiel einer 5-Bit Floating-Point-Zahl in den folgenden Absätzen dargestellt und erklärt [1].

Die 5-Bit Floating-Point-Zahl hat 1-Bit S, 2-Bit E (excess-1 coded) und 2-Bit M (ohne „1.“) [1]. Die zwei Bits des Exponenten bedeuten, dass man drei Intervalle hat [1]. In der Excess-Darstellung hat man 00 (binär) für Exponent -1 (dezimal), 01 (binär) für Exponent 0 (dezimal) und 10 (binär) für Exponent 1 (dezimal) [1]. Die Intervalle befinden sich zwischen Zweierpotenzen [1]. Das erste Intervall befindet sich zwischen $0,5 (=2^{-1})$ und $1 (=2^0)$. Das zweite Intervall befindet sich zwischen $1 (=2^0)$ und $2 (=2^1)$. Und schließlich das dritte Intervall befindet sich zwischen $2 (=2^1)$ und $4 (=2^2)$. Die zwei Bits der

Mantisse bedeuten, dass man $2^N=4$ darstellbare Zahlen pro Intervall hat [1]. Das Ergebnis einer Operation wird zu der darstellbaren Zahl gerundet, die sich am nächsten zu der Ergebniszahl befindet [1]. Die 1 Bits S legt das Vorzeichen fest [1].

Wenn man die darstellbaren Zahlen von der größten zur kleinsten Zahl betrachtet, wird die Größe der Intervalle immer halbiert [1]. Somit rücken die darstellbaren Zahlen in Richtung 0 immer weiter zusammen und das Ergebnis einer Operation kann genauer erfasst werden [1]. Das ist wichtig, da bei kleinen Zahlen schon ein kleiner Rundungsfehler das Ergebnis stark verändert. Hingegen bei großen Zahlen hat ein kleiner Rundungsfehler kaum Auswirkung. Zum Beispiel ein Rundungsfehler von 1 bei der Zahl 1.000.000 wird nicht bemerkt, hingegen bei 10 ist der Rundungsfehler 1 deutlich sichtbar [1].

Beim No-Zero Format (siehe Tabelle 17) gibt es zwischen 0 und 0,5 keine darstellbaren Zahlen [1]. Auch 0 kann nicht dargestellt werden [1].

Tabelle 17: No-Zero Format (Informationen/Zahlen von [1])

Intervall 1	Wert 1	0,5
	Wert 2	0,625
	Wert 3	0,75
	Wert 4	0,875
Intervall 2	Wert 1	1
	Wert 2	1,25
	Wert 3	1,5
	Wert 4	1,75
Intervall 3	Wert 1	2
	Wert 2	2,5
	Wert 3	3
	Wert 4	3,5

Beim Abrupt Underflow Format (siehe Tabelle 18) gibt es zwischen 0 und 1 keine darstellbaren Zahlen. Hingegen kann 0 dargestellt werden [1]. Wenn der Exponent $E = 00...0$ (binär) ist, dann wird die Zahl automatisch als 0 interpretiert [1].

Tabelle 18: Abrupt Underflow Format (Informationen/Zahlen von [1])

Intervall 1	Wert 1	0
Intervall 2	Wert 1	1
	Wert 2	1,25
	Wert 3	1,5
	Wert 4	1,75
Intervall 3	Wert 1	2
	Wert 2	2,5
	Wert 3	3
	Wert 4	3,5

Beim Denormalization Format (siehe Tabelle 19) wird bei einem Exponent $E = 00...0$ (binär), die Zahl nur dann als 0 interpretiert, wenn auch die Mantisse 0 ist [1]. Sollte die Mantisse nicht 0 sein, wird die Zahl als denormalisierte Zahl bezeichnet und als solche behandelt [1].

Tabelle 19: Denormalization Format (Exponent und Mantisse) [1] (Spaltenüberschriften angepasst)

Exponent (binär)	Mantisse	Bedeutet
11 ... 1	$\neq 0$	NaN
11 ... 1	$= 0$	$(-1)^S \times \infty$
00 ... 0	$\neq 0$	Denormalized
00 ... 0	$= 0$	0

Mit dieser Methode wird die Normalisierung gelockert und es können somit Zahlen nahe 0 dargestellt werden (siehe Tabelle 20) [1]. Beim Exponent $E=00\dots 0$ (binär) hat die Mantisse nicht mehr die Form $1.x$, sondern $0.x$ [1]. Wenn der n -Bit Exponent gleich $00\dots 0$ (binär) ist, dann ist der Wert $= 0.M \times 2^{-2^{(n-1)+2}}$ [1]. „00001“ (0 für positives Vorzeichen, 00 für Exponent, 01 für Mantisse) entspricht in der denormalisierten Darstellung der Zahl 0.01×2^0 im Binärsystem und $2^{-2} = 0.25$ im Dezimalsystem [1]. 10 in der Mantisse bedeutet 0.1 im Binärsystem und $2^{-1} = 0.5$ im Dezimalsystem. 11 in der Mantisse entspricht 0.11 im Binärsystem und $2^{-1}+2^{-2} = 0.75$ im Dezimalsystem. Der Abstand zwischen den darstellbaren Zahlen in den ersten zwei Intervallen ist gleich (siehe Tabelle 20) [1].

Tabelle 20: Denormalization Format (Informationen/Zahlen von [1])

Intervall 1	Wert 1	0
	Wert 2	0,25
	Wert 3	0,5
	Wert 4	0,75
Intervall 2	Wert 1	1
	Wert 2	1,25
	Wert 3	1,5
	Wert 4	1,75
Intervall 3	Wert 1	2
	Wert 2	2,5
	Wert 3	3
	Wert 4	3,5

Ab der Fermi-Architektur wird bei Single-Precision „denormalized numbers“ unterstützt [5]. In den Architekturen vor Fermi wurden denormalisierte Zahlen einfach auf 0 gesetzt [5]. Mit dem Compilerflag „-ftz=true“ kann auch bei Fermi dieser Zustand für mehr Performance auf Kosten der Genauigkeit wieder hergestellt werden [5].

Mit Fermi wird der Full IEEE 754-2008 32-bit und 64-bit Standard verwendet [15]. Single-Precision (Float-Datentyp, 4 Byte, 32 Bit) bedeutet 1 Bit S, 8 Bit E und 23 Bit M [1]. Double-Precision (Double-Datentyp, 8 Byte, 64 Bit) bedeutet 1 Bit S, 11 Bit E und 52 Bit M [1]. Durch die um 29 Bit größere Mantisse beträgt der größte Rundungsfehler bei der Darstellung einer Zahl bei Double Precision $1 / 2^{29}$ des größten Rundungsfehlers bei der Darstellung einer Zahl bei Single-Precision [1]. Durch die größere Mantisse können sehr viele Stellen einer Zahl gesichert werden und somit die Zahlen nahe 0 sehr exakt dargestellt werden [1]. Mit den drei zusätzlichen Bits beim Exponent wird die Anzahl der Intervalle erhöht [1]. Dadurch können noch größere negative und positive Zahlen abgedeckt werden [1].

Um die Genauigkeit einer Operation darzustellen, wird ULP verwendet. ULP steht für „unit in the last place“ [1] und ist der Abstand zwischen den darstellbaren Fließkommazahlen des Intervalls, in dem sich die Ergebniszahl befindet [37]. Je größer die Fließkommazahl desto größer ist ULP, da der Abstand zwischen den darstellbaren Zahlen in den Intervallen mit größeren Exponenten immer größer wird [37]. Die Hardware führt arithmetische und Rundungs-Funktionen nur dann perfekt aus,

wenn der größte Fehler, der passieren kann, nicht über 0.5 ULP liegt [1]. 0.5 ULP wird erreicht, wenn die zu erfassende Zahl exakt in der Mitte von zwei möglichen darstellbaren Zahlen liegt. Bei GPUs wird bei der Addition- und Multiplikation-Operation kein größerer Fehler als 0.5 ULP gemacht [1]. Ein ULP von 0.5 wird vom IEEE Standard auch gefordert [5]. Bei komplexeren Operationen kann es sein, dass der ULP größer als 0.5 ist [1].

Der „Maximum ULP Error“, der in den nachfolgenden Tabellen (siehe Tabelle 21 und Tabelle 22) angegeben ist, ist der Unterschied der ULPs zwischen dem korrekt gerundeten Ergebnis und dem Ergebnis von der CUDA-Bibliothek [5]. Da beide einen ULP von 0.5 bei der Addition haben, ergibt das einen „Maximum ULP Error“ von 0 in der Tabelle. Die Zahl 0 in der Tabelle bedeutet somit keinen Unterschied zum korrekt gerundeten Ergebnis. Je höher die Zahl in der Tabelle ist, desto ungenauer das Ergebnis der Operation. Je nach Architektur/Compute Capability kann die Genauigkeit unterschiedlich sein. Mit der Architektur Fermi wurde die Genauigkeit der Single-Precision-Funktionen `sqrtf()` (Maximum ULP Error von 3 auf 0), `x/y` (Maximum ULP Error von 2 auf 0) und `1/x` (Maximum ULP Error von 1 auf 0) verbessert. Das Wort „full range“ wird von Nvidia im Dokument [5] nicht näher erklärt. In Tabelle 21 und Tabelle 22 sind nicht alle Funktionen dargestellt. In [5] kann man den ULP der in Tabelle 21 und Tabelle 22 nicht dargestellten Funktionen einsehen.

Tabelle 21: Maximaler ULP-Fehler bei den Nvidia CUDA Single-Precision-Funktionen [5] (Ausschnitt der Originaltabelle)

Function	Maximum ulp error
$x+y$	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when addition is merged into an FMAD)
$x*y$	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when multiplication is merged into an FMAD)
x/y	0 for compute capability ≥ 2 when compiled with <code>-prec-div=true</code> 2 (full range), otherwise
$1/x$	0 for compute capability ≥ 2 when compiled with <code>-prec-div=true</code> 1 (full range), otherwise
<code>rsqrtf(x)</code> <code>1/sqrtf(x)</code>	2 (full range) Applies to <code>1/sqrtf(x)</code> only when it is converted to <code>rsqrtf(x)</code> by the compiler.
<code>sqrtf(x)</code>	0 for compute capability ≥ 2 when compiled with <code>-prec-sqrt=true</code> 3 (full range), otherwise

Tabelle 22: Maximaler ULP-Fehler bei den Nvidia CUDA Double-Precision-Funktionen [5] (Ausschnitt der Originaltabelle)

Function	Maximum ulp error
$x+y$	0 (IEEE-754 round-to-nearest-even)
$x*y$	0 (IEEE-754 round-to-nearest-even)
x/y	0 (IEEE-754 round-to-nearest-even)
$1/x$	0 (IEEE-754 round-to-nearest-even)
<code>sqrt(x)</code>	0 (IEEE-754 round-to-nearest-even)
<code>rsqrt(x)</code>	1 (full range)

2.4.2.17 Ausführen von 32-Bit-Integer-Multiplikationen in einem Taktzyklus

In der Vorgängerarchitektur GT200 wurden 24-Bit-Integer-Multiplikationen in einem Taktzyklus ausgeführt [27]. Die Ausführung von 32-Bit-Integer-Multiplikationen hat mehrere Taktzyklen benötigt [27]. Mit Fermi ist es genau umgekehrt [27].

2.4.2.18 ECC bei Tesla-GPUs

Auf den Tesla-GPUs der Fermi-Architektur wurde erstmals ECC (=Error Correcting Code) integriert [15]. Durch natürliche Strahlung auf den Speicher können sich Bits im Speicher ändern und dadurch einen „soft error“ erzeugen [15]. Tesla-GPUs der Fermi-Architektur sind nun durch ECC in der Lage „single-bit soft error“ zu korrigieren, bevor es zu einem Fehler in der Berechnung kommt [15].

Mit Fermi werden SECDED (=Single-Error Correct Double-Error Detect) ECC Codes verwendet [15]. Wenn auf Datenelemente zugegriffen wird, kann die GPU Einzel-Bit-Fehler korrigieren [15]. Multi-Bit-Fehler können nicht korrigiert werden [15]. Es kann jedoch festgestellt werden, ob ein Fehler vorliegt und das Programm somit neugestartet werden muss [15].

Im Register, Shared Memory, L1-Caches, L2-Cache und DRAM kommt ECC zur Anwendung [15].

ECC ist standardmäßig aktiviert [27]. Deaktivieren lässt es sich über das „nvidia-smi“-Tool [27]. Mit ECC reduziert sich die Peak-Bandbreite des Global Memory um zirka 20% [27]. Bei verteilten Schreibzugriffen liegt der Verlust höher [27]. Ebenfalls reduziert sich die benutzbare Größe des Device Memory der GPU um 12,5% [25]. Die 12,5% werden für die ECC-Bits benötigt [25].

Da für Scientific Computing ECC sehr relevant ist, sind vor allem die Werte (Peak-Bandbreite und Größe des Global/Device Memory) mit ECC wichtig. Auf der Nvidia-Homepage zur Tesla C2050 (siehe [25]) werden diese Werte jedoch ohne ECC angegeben und nur am Ende der Tabelle befindet sich ein Hinweis, dass sich die Größe des Device Memory um 12,5% reduziert. Einen Hinweis, dass sich auch die Bandbreite des Global Memory reduziert, findet man auf der Website nicht. Diese Information wurde im Nvidia-PDF „Tuning CUDA Applications for Fermi“ [27] gefunden.

2.4.2.19 Debugger CUDA-GDB und CUDA-memcheck

Der CUDA-Debugger, abgekürzt CUDA-GDB, ist eine Erweiterung des „standard i386/AMD64 port of gdb, the GNU Project debugger, version 6.6“ [38]. Durch die Erweiterung kann auch ein Debugging des Device/CUDA-Codes durchgeführt werden [38].

Mit CUDA-GDB wird der Code beim Debugging direkt auf der GPU ausgeführt [38]. Es findet keine Simulation statt [38]. Die Benützung des CUDA-Debugger wird im Nvidia-PDF „CUDA-GDB (NVIDIA CUDA Debugger) User Manual“ [38] genau beschrieben.

Beim Debugger CUDA-GDB wird oft statt dem Begriff Thread das Wort Lane verwendet [38].

Bei Fermi-GPUs können folgende 11 Fehler durch den Debugger identifiziert werden (siehe nachfolgende Tabelle 23).

Tabelle 23: CUDA-GDB-Fehlermeldungen [38]

Exception code	Precision of the Error	Scope of the Error	Description
CUDA_EXCEPTION_0 : "Device Unknown Exception"	Not precise	Global error on the GPU	This is a global GPU error caused by the application which does not match any of the listed error codes below. This should be a rare occurrence.
CUDA_EXCEPTION_1 : "Lane Illegal Address"	Precise (Requires memcheck on)	Per lane/thread error	This occurs when a thread accesses an illegal (out of bounds) global address.
CUDA_EXCEPTION_2 : "Lane User Stack Overflow"	Precise	Per lane/thread error	This occurs when a thread exceeds its stack memory limit.
CUDA_EXCEPTION_3 : "Device Hardware Stack Overflow"	Not precise	Global error on the GPU	This occurs when the application triggers a global hardware stack overflow. The main cause of this error is large amounts of divergence in the presence of function calls.
CUDA_EXCEPTION_4 : "Warp Illegal Instruction"	Not precise	Warp error	This occurs when any thread within a warp has executed an illegal instruction.
CUDA_EXCEPTION_5 : "Warp Out-of-range Address"	Not precise	Warp error	This occurs when any thread within a warp accesses an address that is outside the valid range of local or shared memory regions.
CUDA_EXCEPTION_6 : "Warp Misaligned Address"	Not precise	Warp error	This occurs when any thread within a warp accesses an address in the local or shared memory segments that is not correctly aligned.
CUDA_EXCEPTION_7 : "Warp Invalid Address Space"	Not precise	Warp error	This occurs when any thread within a warp executes an instruction that accesses a memory space not permitted for that instruction.
CUDA_EXCEPTION_8 : "Warp Invalid PC"	Not precise	Warp error	This occurs when any thread within a warp advances its PC beyond the 40-bit address space.
CUDA_EXCEPTION_9 : "Warp Hardware Stack Overflow"	Not precise	Warp error	This occurs when any thread in a warp triggers a hardware stack overflow. This should be a rare occurrence.
CUDA_EXCEPTION_10 : "Device Illegal Address"	Not precise	Global error	This occurs when a thread accesses an illegal (out of bounds) global address. For increased precision, "set cuda memcheck on" in cuda-gdb.
CUDA_EXCEPTION_11 : "Lane Misaligned Address"	Precise (Requires memcheck on)	Per lane/thread error	This occurs when a thread accesses a global address that is not correctly aligned.

Der CUDA-Debugger enthält das Tool `cuda-memcheck` [39]. Mit diesem Tool können Speicherfehler, wie „out-of-bounds“- und „misaligned“-Global-Memory-Zugriffe, erkannt werden [39]. Ohne `CUDA-memcheck` bekommt man nur den Error Code „Unspecified Launch Failure“ zurückgeliefert [39].

Wenn `cuda-memcheck` mit der Release Version des Programms ausgeführt wird, bekommt man die Art des Speicherfehlers und den Kernel zurückgeliefert, in dem der Zugriffsfehler auftritt [39]. Die Release-Version des Programms wird durch `cuda-memcheck` langsamer und außerdem kann nur zu einem Zeitpunkt ein Kernel ausgeführt werden [39].

Will man die genaue Zeilennummer im Kernel haben, in der der Fehler auftritt, muss man das Programm in der Debug Version ausführen [39]. Dazu muss das Flag-Paar „-g -G“ beim Kompilieren angegeben werden [39]. Durch Übergabe des „--continue“-Flags wird bei einem Zugriffsfehler versucht, den Kernel weiter auszuführen, um alle Fehler auszugeben [39].

Im nachfolgenden Programm (siehe Code 14) werden 2 Fehler ausgelöst (unaligned-Zugriff und out-of-bounds-Zugriff) und es soll damit `cuda-memcheck` und der `CUDA-GDB` getestet werden.

Code 14: Demo-Programm von Nvidia zum Testen von `cuda-memcheck` und `CUDA-GDB` [39]

```
#include <stdio.h>

__device__ int x;
__global__ void unaligned_kernel(void) {
    *(int*) ((char*)&x + 1) = 42;
}

__global__ void out_of_bounds_kernel(void) {
    *(int*) 0x87654320 = 42;
}

int main() {
    printf("Running unaligned_kernel\n");
    unaligned_kernel<<<1,1>>>();
    printf("Ran unaligned_kernel: %s\n", cudaGetErrorString(cudaGetLastError()));
    printf("Sync: %s\n", cudaGetErrorString(cudaThreadSynchronize()));

    printf("Running out_of_bounds_kernel\n");
    out_of_bounds_kernel<<<1,1>>>();
    printf("Ran out_of_bounds_kernel: %s\n", cudaGetErrorString(cudaGetLastError()));
    printf("Sync: %s\n", cudaGetErrorString(cudaThreadSynchronize()));

    return 0;
}
```

Das Programm wird mit dem Befehl „`nvcc memcheck_demo.cu -o memcheck_demo -O3`“ kompiliert. Wenn das Programm ohne `memcheck` ausgeführt wird (`./memcheck_demo`), wird bei beiden Kernels als Fehler „unspecified launch failure“ ausgegeben (siehe Terminalausgabe 5). Beim ersten Kernel wird erst beim Synchronisationspunkt (siehe erstes „Sync:“ in Terminalausgabe 5) der Fehler ausgegeben, da Kernels asynchron aufgerufen werden und zu Beginn des Kernelaufrufs noch kein Fehler aufgetreten ist. Beim zweiten Kernelaufruf wird vor dem Synchronisationspunkt „unspecified launch failure“ ausgegeben, da dies der Fehler vom vorigen Kernel ist. Der Funktionsaufruf `cudaGetErrorString(cudaGetLastError())` (siehe Code 14) liest immer den letzten Fehler aus. Da der zweite Kernel aufgrund des asynchronen Aufrufs noch nicht ausgeführt wurde, wird der Fehler des ersten Kernels ausgelesen. Beim Synchronisationspunkt nach dem Aufruf des zweiten Kernels wird der Fehler des zweiten Kernels ausgegeben (siehe zweites „Sync:“ in Terminalausgabe 5).

Terminalausgabe 5: Ausführung des Demo-Programms von Nvidia ohne memcheck

```
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: unspecified launch failure
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: unspecified launch failure
Sync: unspecified launch failure
```

Wenn man das Programm mit memcheck ausführt (Befehl: `cuda-memcheck --continue ./memcheck_demo`), wird statt „unspecified launch failure“ beim ersten Kernel „misaligned“ und beim zweiten Kernel „out of bounds“ ausgegeben (siehe Terminalausgabe 6). Durch das Setzen des „--continue“-Flags beim Aufruf des Programms mit memcheck wird beim Auftreten eines Fehlers das Programm nicht sofort beendet, sondern es wird versucht, das Programm weiter laufen zu lassen, um alle Fehler ausgeben zu können [39].

Terminalausgabe 6: Ausführung des Demo-Programms von Nvidia mit memcheck

```
===== CUDA-MEMCHECK
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
===== Invalid __global__ write of size 4
===== at 0x00000020 in unaligned_kernel
===== by thread (0,0,0) in block (0,0)
===== Address 0x05200001 is misaligned
=====
===== Invalid __global__ write of size 4
===== at 0x00000010 in out_of_bounds_kernel
===== by thread (0,0,0) in block (0,0)
===== Address 0xffffffff87654320 is out of bounds
=====
===== ERROR SUMMARY: 2 errors
```

Um ein Programm mit dem in CUDA-Debugger integrierten memcheck ausführen zu können, müssen fünf Befehle nacheinander eingegeben werden. Zuerst wird das Programm mit dem Flag-Paar „-g -G“ kompiliert (Befehl: `nvcc -g -G memcheck_demo.cu -o memcheck_demo`) [39]. Im nächsten Schritt wird das Programm mit dem Debugger gestartet (Befehl: `cuda-gdb memcheck_demo`) [38]. Danach wird memcheck im Debugger mit dem Befehl „set cuda memcheck on“ aktiviert und ein Breakpoint mit „b main“ gesetzt [38]. Mit dem Starten des Debugging (Befehl: `r`) [39] wird der erste Fehler mit Typ (CUDA_EXCEPTION_1, Lane Illegal Address), der Codezeilennummer (5) und der Codezeile (5 `*(int*) ((char*) &x + 1) = 42;`) ausgegeben (siehe Terminalausgabe 7). Eigentlich würde man bei `unaligned_kernel()` nicht den „CUDA_EXCEPTION_1, Lane Illegal Address“-Fehler erwarten, da dieser für out-of-bounds-Fehler steht (siehe Tabelle 23). Die richtige Fehlermeldung für misaligned-Fehler ist „CUDA_EXCEPTION_11, Lane Misaligned Address“ (siehe Tabelle 23).

Mit dem Befehl „p &x“ kann der Ort und der Datentyp der Variable x ausgelesen werden. Mit „c“ wird die Ausführung des Programms fortgesetzt bzw. im Falle einer Fehlermeldung beendet und mit „Quit“ kann der Debugger verlassen werden [38].

Terminalausgabe 7: Ausführung des Demo-Programms von Nvidia mit memcheck im CUDA-GDB [39]

```
(cuda-gdb) r
Starting program: memcheck_demo
[Thread debugging using libthread_db enabled]
[New process 23653]
Running unaligned_kernel
[New Thread 140415864006416 (LWP 23653)]
[Launch of CUDA Kernel 0 on Device 0]

Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
[Switching to CUDA Kernel 0 (<<<(0,0),(0,0,0)>>>)]
0x0000000000992e68 in unaligned_kernel <<<(1,1),(1,1,1)>>> () at
memcheck_demo.cu:5
5      *(int*) ((char*)&x + 1) = 42;
(cuda-gdb) p &x
$1 = (@global int *) 0x42c00
(cuda-gdb) c
Continuing.

Program terminated with signal CUDA_EXCEPTION_1, Lane Illegal Address.
The program no longer exists.
(cuda-gdb)
```

2.4.2.20 Memory Coalescing / Global-Memory-Zugriff

Bei Fermi werden Global-Memory-Zugriffe von allen Threads eines Warps (=32 Threads) zusammen erzeugt [27]. Bei den Vorgängerarchitekturen wurden nur die Global-Memory-Zugriffe der Threads des halben Warps (=16 Threads) zusammengefasst [27].

2.4.2.21 Constant Cache

Mit Fermi kann der Compiler LDU(=Load Uniform)-Instruktionen einsetzen, um Werte vom Global Memory über den Constant-Memory-Cache zugreifbar zu machen [27]. Dadurch können nicht nur „__constant__“-Variablen über den Constant Cache gelesen werden [27]. Jede Variable, die read-only im Kernel ist, auf den Global Memory zeigt und unabhängig von der Thread-ID ist, kann nun über den Constant Cache gelesen werden [5].

2.4.2.22 Concurrent Data Transfers

Mit den Tesla-GPUs der Fermi-Architektur kann durch zwei DMA-Engines bidirektionale PCIe-Kommunikation durchgeführt werden [35]. Damit können gleichzeitig Daten vom Host zur GPU und Daten von der GPU zum Host transportiert werden [5]. Der Host Memory muss jedoch page-locked sein [5].

2.4.2.23 C++ Unterstützung

Mit Fermi wurde die C++ Unterstützung im Kernelcode stark erweitert, jedoch wird weiterhin keine „run time type information (RTTI)“, kein exception handling und die „C++ Standard Library“ nicht unterstützt [5].

3 GPU-Programmierplattformen

In diesem Kapitel wird auf die GPU-Programmiersprachen CUDA und OpenCL, die für die GPU-Programme in der Masterarbeit verwendet werden, näher eingegangen. Für das parallele CPU-Programm (Benützung aller Kerne der CPU) in der Masterarbeit wird OpenMP verwendet.

3.1 CUDA

CUDA sieht die Hardware als Shared-Memory-System, wodurch eine einfachere Programmierung ermöglicht wird [1]. Außerdem ist CUDA bei GPUs bestens für fine-grained Parallelisierung (=viele Threads) ausgelegt, da CUDA ein einfaches Thread-Management mit niedrigen Overhead (=lightweight threads) anbietet und es keine Cache-Kohärenz-Hardware-Anforderungen gibt [1]. Im Gegensatz dazu ist OpenMP bei CPUs aufgrund des nicht so kleinen Thread-Management-Overheads und der Cache-Kohärenz-Hardware-Anforderungen am besten für course-grained Parallelisierung (=wenige Threads) geeignet [1]. Die Erzeugung und das Scheduling von Threads kann bei GPUs durch die effiziente Hardware-Unterstützung auf wenige Zyklen reduziert werden [1]. Im Vergleich dazu bei CPUs benötigt es 1000te Taktzyklen für das Erzeugen und Scheduling von Threads [1].

CUDA C ist eine kleine Erweiterung von C um GPU-Computing zu ermöglichen [1]. Die Programmierung findet in dem SPMD(=Single Program Multiple Data)-Parallel-Programming-Modell statt, das für parallele Programmierung Standard ist [1]. SPMD bedeutet, dass alle Threads denselben Code (=Kernel im Programm) ausführen [1]. Beim Programmieren müssen zunächst die Teile im Programm identifiziert werden, die für die Parallelisierung geeignet sind [1]. Danach muss Speicher für die Inputdaten auf der GPU reserviert werden, indem eine API-Funktion benutzt wird [1]. Eine weitere API-Funktion wird verwendet, um die Daten vom System-Memory zum GPU-Memory zu transferieren [1]. Im nächsten Schritt wird der Kernel geschrieben, der parallel von sehr vielen Threads abgearbeitet wird und den parallelen Code des Programms darstellt [1]. Schließlich muss die Kernel-Funktion im Code aufgerufen werden und die Ergebnisdaten mit einer API-Funktion vom GPU-Memory in den System-Memory kopiert werden [1].

Aus Sicht eines CUDA-Programms besteht die Hardware aus einem Host und einem Device [1]. Beim Host handelt es sich um eine CPU, auf der das Programm gestartet wird und die Teile mit wenig Datenparallelität ausgeführt werden [1]. Die stark datenparallelen Teile des Programms werden auf dem Device ausgeführt [1]. Bei dem Device handelt es sich um eine GPU [1]. Unter Datenparallelität versteht man, dass eine Funktion auf viele unterschiedliche Datenbereiche gleichzeitig ausgeführt werden kann, ohne dass beim Ausführen der Funktion auf das Ergebnis eines anderen Datenbereichs gewartet werden muss. Ein ideales Beispiel für Datenparallelität ist die Matrizenmultiplikation [1]. Bei der Multiplikation von zwei 1000x1000 Matrizen müssen 1.000.0000 Skalarprodukte ausgeführt werden [1]. Da jedes Skalarprodukt bei der Berechnung nicht die Ergebnisse der anderen Skalarprodukte benötigt, können alle Skalarprodukte parallel ausgeführt werden. Der einzige Unterschied zwischen den Skalarprodukten ist, dass sich die Inputdaten unterscheiden.

Ein CUDA-Programm enthält den Host- und Device-Code [1]. Erst wenn mit nvcc (=Nvidia C Compiler) kompiliert wird, werden die zwei Codes getrennt [1]. Der Host-Code ist ANSI-C-Code und wird nach der Trennung vom Device-Code mit dem Standard-C-Compiler des Hosts kompiliert [1]. Der Device-Code ist ANSI-C-Code mit CUDA-Schlüsselwörtern für die Parallelität und wird nochmals mit dem nvcc kompiliert [1].

Die aktuelle Version (Stand: November 2012) von CUDA ist 5 und wurde im Oktober 2012 veröffentlicht [40].

3.2 OpenCL

OpenCL ist nach CUDA veröffentlicht worden und lässt sich auf ähnliche Weise wie CUDA programmieren [1]. OpenCL basiert wie CUDA ebenfalls auf der Programmiersprache C [1]. Der große Vorteil von OpenCL gegenüber CUDA ist, dass der OpenCL-Code ohne Änderungen auf allen Prozessoren (z. B. GPUs (Nvidia, AMD) und CPUs (AMD, Intel)), die die OpenCL-Spracherweiterungen und API unterstützen, ausgeführt werden kann [1] [9] [10]. Änderungen am Code sind nur dann notwendig, wenn Hardwareanforderungen im Code bei der zu benützenden Hardware nicht erfüllt

werden können. Das kann passieren, wenn die Blockgröße auf 512 Threads im Code festgelegt ist, aber bei Verwendung einer AMD-GPU maximal nur 256 Threads pro Block möglich sind.

Die Entwicklung von OpenCL wurde von Apple in Gang gesetzt und schließlich von der Khronos Group umgesetzt [1]. Die Khronos Group kümmert sich neben OpenCL auch um die Entwicklung des OpenGL-Standards [1].

Die aktuelle Version (Stand: April 2012) von OpenCL ist 1.2 und wurde am 15. November 2011 veröffentlicht [41].

3.3 Programmierung mit CUDA und OpenCL

Bei der Programmierung gibt es Unterschiede zwischen CUDA und OpenCL. Ein Unterschied ist, dass beim Kernelaufbau bei CUDA die Blockgröße und die Anzahl der Blöcke und bei OpenCL die Blockgröße und die Gridgröße übergeben werden (siehe Unterkapitel 3.3.6.1). Ein weiterer Unterschied ist, dass der Shared Memory in CUDA der Local Memory in OpenCL ist und der Local Memory in CUDA der Private Memory im OpenCL ist (siehe Unterkapitel 3.3.8). Bei der Ausführung mehrerer Kernels auf unterschiedlichen Devices oder parallel auf demselben Device (=Concurrent Kernels) gibt es ebenfalls Unterschiede zwischen CUDA und OpenCL (siehe Unterkapitel 3.3.6.4 und Unterkapitel 3.3.6.5). Bei OpenCL müssen in beiden Fällen mehrere Queues erzeugt werden [6]. Bei der Erzeugung einer Queue wird das Device festgelegt. Bei CUDA muss für die Ausführung des Kernels auf unterschiedlichen Devices das Device mit Hilfe einer API-Funktion gewechselt werden und für das parallele Ausführen mehrerer Kernels auf einem Device müssen mehrere Streams erzeugt werden [5].

3.3.1 Include-Statements

Bei CUDA und OpenCL muss eine Header-Datei („cuda.h“ bei CUDA und „opencl.h“ bei OpenCL) inkludiert werden (siehe Code 15) [28]. Bei OpenCL werden mit der „opencl.h“-Datei die Header-Dateien „cl.h“, „cl_gl.h“, „cl_gl_ext.h“ und „cl_ext.h“ inkludiert [28]. Da sich die „opencl.h“-Datei in dem Unterordner „CL“ befindet, muss dieser Unterordner beim include-Statement angegeben werden (siehe Code 15) [28].

Code 15: Include-Statement für die Header-Datei [28]

CUDA	OpenCL
<code>#include <cuda.h></code>	<code>#include <CL/opencl.h></code>

Falls der Kernel/Device-Code sich in einer anderen Datei befindet, muss dieser bei CUDA inkludiert werden. Bei OpenCL wird erst zur Laufzeit die Kerneldatei eingelesen und der Kernel kompiliert (siehe Kapitel 3.3.2) [1]. Bei OpenCL kann am Beginn des Programms der Dateiname der Kerneldatei und der Name des darin befindlichen Kernels festgelegt werden (siehe Code 16).

Code 16: Include-Statement für die Kernel-Datei

CUDA	OpenCL
<code>#include "kerneldatei.cu"</code>	<code>#define Kernel1Datei "kerneldatei.cl"</code> <code>#define Kernel1Name "kernelname"</code>

3.3.2 Initiierung

Da OpenCL sehr viele Geräte von unterschiedlichen Herstellern unterstützen muss, ist die Komplexität des Programmcodes für die Initiierung etwas größer als bei CUDA [1].

Der Unterschied zwischen CUDA und OpenCL ist, dass bei CUDA bei der Auswahl des Device nur GPUs in Frage kommen und dabei nur Nvidia-GPUs möglich sind [1]. Somit kann die Plattform automatisch

ausgewählt werden. Bei CUDA muss nur das Device festgelegt werden, falls sich mehrere Nvidia-GPUs auf dem Mainboard befinden (siehe Code 17) [1]. Durch die CUDA-runtime-API sind nicht mehr Befehle notwendig [1]. Bei OpenCL hingegen benötigt die Initialisierung deutlich mehr Codezeilen (siehe Code 18). Zunächst wird mit `clGetPlatformIDs()` eine OpenCL-Plattform ermittelt [42]. Dann wird mit `clGetDeviceIDs()` eine GPU ausgewählt [42]. Danach wird mit `clCreateContext()` ein Context, dem die GPU hinzugewiesen wird, erzeugt [42]. Um einen Kernel auf der GPU ausführen zu können, muss noch eine Command Queue mit `clCreateCommandQueue()` erzeugt werden [1]. Beim Erzeugen der Command Queue wird der Context und die ausgewählte GPU übergeben [42].

Da OpenCL ein dynamisches Kompiliermodell hat, muss im Host-Programm ein Code für das Kompilieren der Kernels eingefügt werden [1]. Dazu wird zunächst das Device-Programm aus der Kernel-Datei in einem char-Array/String gespeichert [28] [42]. Mit diesem char-Array wird das Device-Programm zum OpenCL-Runtime-System (siehe `clCreateProgramWithSource()` im Code 18) gebracht, indem das char-Array und der Context übergeben wird [1]. Im nächsten Schritt wird das Device-Programm kompiliert (siehe `clBuildProgram()` im Code 18) [1]. In diesem Schritt können Compilerflags übergeben werden [1]. Wenn Fehler beim Kompilieren auftreten, können diese abgefragt und im Terminal ausgegeben werden (siehe `clGetProgramBuildInfo()` im Code 18) [43]. Im letzten Schritt wird aus dem kompilierten Device-Programm ein Kernel erzeugt (siehe `clCreateKernel()` im Code 18) [1]. Dabei wird der Kernelname mitübergeben [1]. Sobald dieser Schritt ausgeführt wurde, kann der Kernel zum Ausführen in eine Command Queue geschickt werden [1]. Ein Vorteil des Kompilierens zur Laufzeit ist, dass Änderungen am Kernel zur Laufzeit durchgeführt werden können. Der Nachteil ist, dass das Kompilieren zur Laufzeit zusätzliche Laufzeit verursacht.

Code 17: CUDA-Initiierung [5] (adaptiert)

```
// GPU auswählen (erste GPU: device_nr = 0)
cudaSetDevice(device_nr);
```

Code 18: OpenCL-Initiierung [1] [28] [42] [43] (adaptiert)

```
// OpenCL Plattform finden
cl_platform_id platform;
err=clGetPlatformIDs(1,&platform,NULL);
if(err!=CL_SUCCESS) printf("Error: clGetPlatformIDs\n");

// GPU auswählen (erste GPU: device_nr = 0)
cl_device_id deviceGPU[2];
cl_device_id deviceGPUausgewaehlt;
err=clGetDeviceIDs(platform,CL_DEVICE_TYPE_GPU,2,deviceGPU,NULL);
if(err!=CL_SUCCESS) printf("Error: clGetDeviceIDs\n");
deviceGPUausgewaehlt=deviceGPU[device_nr];

// Context erzeugen = Sammlung von Devices
cl_context clGPUContext =
    clCreateContext(0,1,&deviceGPUausgewaehlt,NULL,NULL,&err);
if(err!=CL_SUCCESS) printf("Fehler bei clCreateContext()\n");

// Command Queue erzeugen
cl_command_queue clCommandQue =
    clCreateCommandQueue(clGPUContext,deviceGPUausgewaehlt,0,&err);
if(err!=CL_SUCCESS) printf("Error: clCreateCommandQueue\n");
```

Fortsetzung von Code 18 auf der nächsten Seite

Fortsetzung von Code 18:

```
// Kernel kompilieren
char *programSource1=LoadFile2txt(Kernel1Datei);
cl_program clProgram1 = clCreateProgramWithSource(clGPUContext,1,
    (const char**)&programSource1,NULL,&err1);

char compilerflagsStandard[4096];
sprintf(compilerflagsStandard,"-D Datentyp4=%s -D Datentyp=%s "
    "-D Anzahl_Gridwerte_in_XRichtung_Pro_Thread=%d -cl-fast-relaxed-math "
    "-cl-single-precision-constant -cl-denorms-are-zero -cl-mad-enable",
    Datentyp4_String,Datentyp_String,Anzahl_Gridwerte_in_XRichtung_Pro_Thread);

err2 = clBuildProgram(clProgram1,0,NULL,compilerflagsStandard,NULL,NULL);
if(err2!=CL_SUCCESS){ //Programmfehler ausgeben
    printf("Fehler und Warnungen (Kernel %s kompilieren):\n",Kernel1Name);
    char buf[0x10000];
    clGetProgramBuildInfo(clProgram1,deviceGPUausgewaehlt,CL_PROGRAM_BUILD_LOG,
        0x10000,buf,NULL);
    printf("%s\n", buf);
    return EXIT_FAILURE;
}

cl_kernel clKernel1 = clCreateKernel(clProgram1,Kernel1Name,&err3);
```

Da bei CUDA die Initialisierung mit dem erstem `cudaMallocHost()` oder `cudaMalloc()` ausgeführt wird, kann mit dem nachfolgenden Code 19 die Dauer für die CUDA-Initialisierung gemessen werden.

Code 19: Dauer der CUDA-Initialisierung

```
gettimeofday(&startTime_CUDA_initalisieren,(struct timezone *)0);
int *Zahl;
cudaMallocHost((void**)&Zahl,sizeof(int));
cudaFreeHost(Zahl);
gettimeofday(&endTime_CUDA_initalisieren,(struct timezone *)0);
```

Die CUDA-Initialisierung mit Toolkit 4.0 dauert bei der GTX 480-Fermi 0,08 - 0,09 Sekunden und bei der GTX 285-GT200 0,07 - 0,08 Sekunden. Im Vergleich dazu dauert die OpenCL-Initialisierung mit Toolkit 4.0 bei der GTX 480-Fermi 1,2 - 1,6 Sekunden und bei der GTX 285-GT200 1,2 - 1,3 Sekunden. Bei OpenCL muss beachtet werden, dass in der Zeitdauer zusätzlich die Kompilierdauer der Kernels enthalten ist, da Kernels bei OpenCL erst zur Laufzeit kompiliert werden.

Die CUDA-Initialisierung der Tesla C2050 benötigt mit 3,2 - 3,7 Sekunden mit Toolkit 3.2 deutlich länger als alle anderen GPUs. Da die GTX 480 vor der Umstellung auf Toolkit 4.0 ebenfalls zirka 3 Sekunden für die Initialisierung benötigt hat, ist beim Umstieg auf Toolkit 4.0 eine deutliche Reduzierung der Initialisierungsdauer bei der Tesla C2050 zu erwarten. Die Tesla C1060 mit Toolkit 3.2 ist mit 0,11 - 0,13 Sekunden fast genauso schnell wie die GTX 285 und die GTX 480 mit Toolkit 4.0. Die OpenCL-Initialisierung dauert auf der Tesla C2050 4,6 - 5,7 Sekunden und auf der Tesla C1060 3,7 - 4,8 Sekunden.

3.3.3 API-Funktionen für den Host für das Anlegen und Freigeben von Speicher im Arbeitsspeicher am Host

Beim Anlegen und Freigeben eines Arrays im Arbeitsspeicher am Host hat man bei CUDA und OpenCL zwei Möglichkeiten. Entweder man benützt wie üblich die `malloc()`-Funktion für das Anlegen und die `free()`-Funktion für das Freigeben eines Arrays im Host-Arbeitsspeicher oder man benützt CUDA bzw. OpenCL spezifische Funktionen. Die CUDA bzw. OpenCL spezifische Funktion alloziert Page-locked/Pinned-Speicher und die `malloc()`-Funktion alloziert Pageable-Speicher [5].

Pinned/Page-locked Memory bedeutet, dass das auf dem Host angelegte Array während der Ausführung der Anwendung sich immer komplett im Host-Arbeitsspeicher befindet und dessen Größe somit durch die Größe des Host-Arbeitsspeichers beschränkt ist [5] [29]. Bei Pageable Memory kann das Betriebssystem das Array auf die Festplatte auslagern (=Paging) [5] [29].

Bei CUDA heißt die Funktion für das Anlegen von Pinned Memory `cudaMallocHost()` bzw. `cudaHostAlloc()` (siehe Code 20) [5]. Mit der Funktion `cudaFreeHost()` wird der Speicher wieder freigegeben [5]. Bei OpenCL wird beim Anlegen von Page-locked/Pinned-Speicher für ein Array die Funktion `clCreateBuffer()` mit dem Flag „CL_MEM_ALLOC_HOST_PTR“ aufgerufen (siehe Code 20) [6]. Laut Nvidia OpenCL Programming Guide [6] bestimmt jedoch der Driver, ob bei OpenCL-Programmen ein Array als Page-locked/Pinned-Speicher oder Pageable-Speicher angelegt wird. Mit dem Flag „CL_MEM_ALLOC_HOST_PTR“ wird im Programmcode sichergestellt, dass die Wahrscheinlichkeit für Page-locked/Pinned-Speicher hoch ist [6]. Mit der Verwendung der `clEnqueueMapBuffer()`-Funktion und des Flags `CL_MAP_WRITE` bekommt man in OpenCL den Standard-Host-Pointer des angelegten Pinned-Arrays zurück und damit kann man im Host-Code Daten aus dem Pinned-Array auslesen und Daten in das Pinned-Array schreiben [44] [45]. Bei CUDA erhält man bereits beim Aufruf von `cudaMallocHost()` bzw. `cudaHostAlloc()` den Standard-Host-Pointer. Für das Freigeben eines Page-locked/Pinned-Speicher wird bei OpenCL die Funktion `clReleaseMemObject()` verwendet [6].

Code 20: Verwendung von Page-locked/Pinned-Speicher [5] [6] [45] (adaptiert)

CUDA	OpenCL
<pre>Datentyp *array; cudaMallocHost((void**)& array, sizeof(Datentyp)*anzahlArrayWerte); cudaFreeHost(array);</pre>	<pre>cl_mem hcl_array_Pinned = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE CL_MEM_ALLOC_HOST_PTR, anzahlArrayWerte*sizeof(Datentyp), NULL, &err1); Datentyp *h_array_Pinned = clEnqueueMapBuffer(clCommandQueue, hcl_array_Pinned, CL_TRUE, CL_MAP_WRITE, 0, anzahlArrayWerte * sizeof(Datentyp), 0, NULL, NULL, &err2); clReleaseMemObject(hcl_array_Pinned);</pre>

3.3.4 API-Funktionen für den Host zum Anlegen und Freigeben von Speicher im Global Memory am Device

Ein Speicher im Global Memory der GPU wird bei CUDA mit der API-Funktion „`cudaMalloc()`“ und bei OpenCL mit der API-Funktion „`clCreateBuffer()`“ angelegt (siehe Code 21) [1]. Am Ende des Programms wird der angelegte Speicher bei CUDA mit der API-Funktion „`cudaFree()`“ und bei OpenCL mit der API-Funktion „`clReleaseMemObject()`“ freigegeben [1].

Die zwei Methoden `cudaFree()` und `clReleaseMemObject()` zum Freigeben von Speicherbereichen unterscheiden sich etwas [1]. Bei OpenCL gibt es eine Zählervariable für jedes Speicherobjekt [1]. Mit dem Aufruf von `clRetainMemObject()` wird der Zähler um 1 erhöht und mit dem Aufruf von `clReleaseMemObject()` wird der Zähler um 1 reduziert [1]. Wenn der Zähler 0 erreicht, wird der reservierte Speicherbereich freigegeben [1]. Der Aufruf von `clCreateBuffer()` führt automatisch zum Ausführen von `clRetainMemObject()` [1].

Damit man im Programmcode sofort erkennt, dass der Pointer auf einen Ort im Speicher des Device zeigt, ist es sinnvoll ein „d“ an den Namen anzuhängen [1].

Code 21: Anlegen und Freigeben von Speicher [1] (adaptiert)

CUDA	OpenCL
<pre>Datentyp *array_d; cudaMalloc((void*)&array_d,N* sizeof(Datentyp)); cudaFree(array_d);</pre>	<pre>cl_mem array_d; array_d = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE,N*sizeof(Datentyp), NULL,NULL); clReleaseMemObject(array_d);</pre>

3.3.5 API-Funktion(en) für den Host für das Transferieren von Daten vom Host zum Device und zurück

Für das Transferieren/Kopieren von Daten vom Host zum Device und zurück wird bei CUDA eine Funktion und bei OpenCL zwei Funktionen benötigt (siehe Code 22) [1]. Bei OpenCL sind die Parameter der zwei Funktionen `clEnqueueWriteBuffer()` und `clEnqueueReadBuffer()` identisch [43]. Bei CUDA wird bei der `cudaMemcpy()`-Funktion als letzter Parameter die Richtung angegeben (`cudaMemcpyHostToDevice` oder `cudaMemcpyDeviceToHost`). Außerdem werden die zwei Parameter, die auf den angelegten Host-Speicher (siehe `array` im Code 22) und auf den angelegten Device-Speicher (siehe `array_d` im Code 22) zeigen, vertauscht.

Code 22: Kopieren von Daten vom Host zum Device und zurück [5] [6] [43] (adaptiert)

CUDA	OpenCL
<pre>cudaMemcpy(array_d,array, N*sizeof(Datentyp), cudaMemcpyHostToDevice); cudaMemcpy(array,array_d, N*sizeof(Datentyp), cudaMemcpyDeviceToHost);</pre>	<pre>clEnqueueWriteBuffer(clCommandQueue, array_d,CL_TRUE,0,N*sizeof(Datentyp), array,0,NULL,NULL); clEnqueueReadBuffer(clCommandQueue, array_d,CL_TRUE,0,N*sizeof(Datentyp), array,0,NULL,NULL);</pre>

3.3.6 Kernel (=paralleler Code ausgeführt auf Device)

Beim Kernel handelt es sich um den parallelen Code, der auf der GPU ausgeführt wird [1]. In der Kernelfunktion steht der Code, der von allen erzeugten Threads ausgeführt wird [1]. Da alle Threads denselben Code ausführen, spricht man von SPMD (=Single Program Multiple Data) [1]. Da die Hardware eine beschränkte Anzahl von Einheiten besitzt, kann gleichzeitig nur eine Teilmenge der Threads abgearbeitet werden. Threads werden zu Thread-Blöcken zusammengefasst und Thread-Blöcke werden zur Laufzeit einem beliebigen Multiprozessor zur Ausführung zugewiesen [1].

Zwischen CUDA und OpenCL gibt es Unterschiede in den Bezeichnungen. In Tabelle 24 sind drei unterschiedliche Bezeichnungen dargestellt. Gleiche Bezeichnungen zwischen CUDA und OpenCL sind unter anderem Host-Programm und Kernel [1].

Tabelle 24: Unterschiede in den Bezeichnungen zwischen CUDA und OpenCL [1] (Ausschnitt der Original-Tabelle, adaptiert)

CUDA	OpenCL
Grid	NDRange (index space)
Thread	Work item
Block	Work group

3.3.6.1 Aufruf des Kernels

Beim Aufruf eines Kernels am Host wird ein Grid von Threads auf dem Device erzeugt [1]. Die Anzahl der erzeugten Threads werden beim Aufruf des Kernels festgelegt. Im Code 23 sind alle notwendigen Programmierschritte zum Aufruf eines Kernels dargestellt. Im ersten Schritt wird die einheitliche Größe eines Blocks von Threads in X- und Y-Richtung festgelegt. Im nächsten Schritt wird die Größe des Grids fixiert, indem die Anzahl der Blöcke in X- und Y-Richtung angegeben werden. Beim Kernaufbau werden bei CUDA die Blockgröße und die Anzahl der Blöcke übergeben. Bei OpenGL werden beim Kernaufbau die Blockgröße und die Gridgröße übergeben. Bei CUDA werden zwei struct-Variablen vom Typ dim3 erzeugt und bei OpenGL werden zwei Array-Variablen vom Typ size_t definiert [1]. Bei CUDA werden die Parameter des Kernels wie bei normalen Funktionen übergeben. Bei OpenGL muss jeder Parameter mit der Funktion „clSetKernelArg()“ definiert werden. In der Klammer beim Kernaufbau werden stattdessen Parameter angegeben, die zur Initialisierung des Kernels notwendig sind. Bei CUDA werden diese Daten vor der Klammer angegeben.

Code 23: Aufruf eines Kernels [5] [6] (adaptiert)

CUDA	OpenGL
<pre>dim3 dimBlock(blockgroesse_xrichtung, blockgroesse_yrichtung) dim3 dimGrid(blockanzahl_xrichtung, blockanzahl_yrichtung) Kernelname<<<dimGrid,dimBlock>>>(Array1,Variable1,Variable2);</pre>	<pre>size_t localWorkSize2d[2]; localWorkSize2d[0]=blockgroesse_xrichtung; localWorkSize2d[1]=blockgroesse_yrichtung; size_t globalWorkSize2d[2]; globalWorkSize2d[0]= blockgroesse_xrichtung* blockanzahl_xrichtung; globalWorkSize2d[1]= blockgroesse_yrichtung* blockanzahl_yrichtung; clSetKernelArg(clKernel,0,sizeof(cl_mem), (void *)& Array1); clSetKernelArg(clKernel,1,sizeof(int), (void *)& Variable1); clSetKernelArg(clKernel,2,sizeof(int), (void *)& Variable2); clEnqueueNDRangeKernel (clCommandQue, clKernel,2,NULL,globalWorkSize2d, localWorkSize2d,0,NULL,NULL);</pre>

3.3.6.2 Kerneldeklaration

Damit der Compiler erkennen kann, dass es sich bei der Funktion um einen Kernel handelt, muss vor dem Funktionsname bei CUDA das „__global__“ - Schlüsselwort und bei OpenGL das „__kernel__“ - Schlüsselwort gesetzt werden (siehe Code 24) [1].

Code 24: Kerneldeklaration [5] [6] (adaptiert)

CUDA	OpenGL
<pre>__global__ void Kernelname(Parameter){}</pre>	<pre>__kernel void Kernelname(Parameter){}</pre>

3.3.6.3 Thread-ID-Bestimmung im Kernel

Die Kernelfunktion besteht aus ANSI-C mit hinzugefügten Schlüsselwörtern [1]. Es gibt bei CUDA acht Schlüsselwörter und bei OpenGL acht Funktionen, die im Kernel benutzt werden, damit ein Thread herausfinden kann, wo er sich im Grid befindet (siehe Tabelle 25). Dadurch kann jeder Thread unterschiedliche Inputdaten bearbeiten und die Ergebnisse der Threads können in unterschiedliche Zellen eines Arrays geschrieben werden.

Bei CUDA kann ein Thread mit „threadIdx.x“ und „threadIdx.y“ auf die x- und y-Position innerhalb des Blocks zugreifen [1]. Bei OpenCL werden die Funktionen get_local_id(0) und get_local_id(1) verwendet [1].

Damit ein Thread auf die x- und y-Position innerhalb des Grids zugreifen kann, werden bei OpenCL die Funktionen get_global_id(0) und get_global_id(1) verwendet [1]. Bei CUDA müssen die Formeln „blockIdx.x*blockDim.x+threadIdx.x“ und „blockIdx.y*blockDim.y+threadIdx.y“ verwendet werden [5]. Mit „blockIdx.x“ und „blockIdx.y“ wird bei CUDA auf die x- und y-Position des Blocks, dem der Thread angehört, zugegriffen und mit „blockDim.x“ und „blockDim.y“ wird die Größe des Blocks in x- und y-Richtung ausgelesen [1]. Die Blockgröße in x- und y-Richtung ist für alle Threads gleich [1]. Die maximale Threadanzahl pro Block ist durch die Hardware vorgegeben [5]. In OpenCL werden die Funktionen get_local_size(0) und get_local_size(1) verwendet, um die Blockgröße zu ermitteln [1].

Mit „gridDim.x“ und „gridDim.y“ wird in CUDA die Anzahl der Blöcke in x- und y-Richtung ausgelesen [1]. Durch die Multiplikation der Anzahl der Blöcke mit der Blockgröße kann in CUDA die Gridgröße in x- und y-Richtung ermittelt werden [1]. In OpenCL werden für den Zugriff auf die Gridgröße in x- und y-Richtung die Funktionen get_global_size(0) und get_global_size(1) verwendet [1].

Tabelle 25: Variablen/Funktionen zur Bestimmung der Position eines Threads (erstellt auf Basis einer Tabelle von [1], Informationen von [1])

Bedeutung	CUDA	OpenCL
Position des Threads im Block in X-Richtung	threadIdx.x	get_local_id(0)
Position des Threads im Block in Y-Richtung	threadIdx.y	get_local_id(1)
Position des Threads im Grid in X-Richtung	blockIdx.x*blockDim.x+threadIdx.x	get_global_id(0)
Position des Threads im Grid in Y-Richtung	blockIdx.y*blockDim.y+threadIdx.y	get_global_id(1)
Größe des Blocks in X-Richtung	blockDim.x	get_local_size(0)
Größe des Blocks in Y-Richtung	blockDim.y	get_local_size(1)
Größe des Grids in X-Richtung	gridDim.x*blockDim.x	get_global_size(0)
Größe des Grids in Y-Richtung	gridDim.y*blockDim.y	get_global_size(1)

Wenn man ein zweidimensionales Grid von Threads hat, fallen bei dem GPU-Kernel im Vergleich zur CPU-Funktion zwei äußere Schleifen weg [1]. Statt i und j hat man die x- und y-Position des Threads im Grid [1]. Beim eindimensionalen Grid fällt nur eine äußere Schleife weg [1].

3.3.6.4 Concurrent Kernels (Streams und Queues)

Damit mehrere voneinander unabhängige Kernels parallel ausgeführt werden können (=Concurrent Kernels), müssen sie in CUDA in unterschiedliche Streams und in OpenCL in unterschiedliche Queues geladen werden (siehe Code 25) [5] [6].

Code 25: Concurrent Kernels [5] [6] [42] (adaptiert)

CUDA	OpenCL
<pre> cudaStream_t stream[2]; for (int i = 0; i < 2; ++i) cudaStreamCreate(&stream[i]); for (int i = 0; i < 2; ++i) MyKernel<<<100,512,0,stream[i]>>>(); </pre>	<pre> cl_command_queue queue[2]; for (int i = 0; i < 2; ++i) queue[i]=clCreateCommandQueue(clGPUContext,deviceGPU[0],0,&err); for (int i = 0; i < 2; ++i) clEnqueueNDRangeKernel(queue[i], MyKernel,2,0,globalWorkSize, localWorkSize,0,0,0); </pre>

3.3.6.5 Ausführung von Kernels auf mehreren Devices

Damit Kernels auf mehreren Devices ausgeführt werden können, muss in CUDA mit der API-Funktion `cudaSetDevice()` zwischen den Devices gewechselt werden (siehe Code 26) [5]. Bei OpenCL muss eine Queue für jedes Device erzeugt werden (siehe Code 26) [6].

Code 26: Ausführung von Kernels auf mehreren Devices [5] [6] [42] (adaptiert)

CUDA	OpenCL
<pre> cudaSetDevice(0); MyKernel<<<1000, 128>>>(); cudaSetDevice(1); MyKernel<<<1000, 128>>>(); </pre>	<pre> cl_device_id deviceGPU[2]; clGetDeviceIDs(platform,CL_DEVICE_TYPE_GPU, 2,deviceGPU,NULL); cl_context clGPUContext = clCreateContext(0, 2,deviceGPU,NULL, NULL,&err); cl_command_queue queue_device1 = clCreateCommandQueue(clGPUContext, deviceGPU[0],0,&err); cl_command_queue queue_device2 = clCreateCommandQueue(clGPUContext, deviceGPU[1],0,&err); clEnqueueNDRangeKernel(queue_device1, MyKernel, 2, 0, globalWorkSize, localWorkSize, 0, 0, 0); clEnqueueNDRangeKernel(queue_device2, MyKernel, 2, 0, globalWorkSize, localWorkSize, 0, 0, 0); </pre>

3.3.7 Begriffe zur Beschreibung der Device-Architektur

Bei der Beschreibung der Device-Architektur gibt es Begriffe, die bei CUDA und OpenCL genau gleich heißen und auch dasselbe bedeuten, aber es gibt auch Begriffe wie Local Memory, die bei OpenCL und CUDA unterschiedliche Bedeutung haben (siehe Tabelle 26). Der Shared Memory wird bei OpenCL als Local Memory (siehe Kapitel 3.3.8) und der Local Memory als Private Memory bezeichnet [1].

Tabelle 26: Begriffe zur Beschreibung der Device-Architektur (erstellt auf Basis einer Tabelle von [1], Informationen von [1])

CUDA	OpenCL
SM (=Streaming Multiprocessor)	CU (=Compute Unit)
SP (=Streaming Processors)	PE (=Processing Elements)
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory

3.3.8 Shared Memory in CUDA ist Local Memory in OpenCL

Bei der Programmierung im Kernel muss beachtet werden, dass der Shared Memory in CUDA der Local Memory in OpenCL ist (siehe Code 27) [1]. Der Local Memory in CUDA bekommt in OpenCL den Namen „Private Memory“.

Code 27: Shared Memory in CUDA ist Local Memory in OpenCL [5] [6]

CUDA	OpenCL
<code>__shared__ float shared[32];</code>	<code>__local float shared[32];</code>

3.3.9 Constant Memory

Bei Constant Memory gibt es einen wesentlichen Unterschied zwischen CUDA und OpenCL. Bei OpenCL legt der Host ein Array im Constant Memory dynamisch an [1]. Bei CUDA wird das Array im Constant Memory statisch erzeugt [1].

Bei OpenCL wird ein Array im Constant Memory mit der `clCreateBuffer()`-Funktion mit dem Flag „CL_MEM_READ_ONLY“ als zweiten Parameter erzeugt und in der Kerneldeklaration wird das Schlüsselwort „__constant“ für den Pointer auf das Constant-Memory-Array verwendet (siehe Code 28) [1] [6]. Die Daten werden über die `clEnqueueWriteBuffer()`-Funktion vom Arbeitsspeicher der CPU in den Constant Memory kopiert [43]. Bei CUDA wird ein Array im Constant Memory statisch außerhalb der main-Methode durch das Schlüsselwort „__constant__“ erzeugt (siehe Code 28) [5]. Mit der `cudaMemcpyToSymbol()`-Funktion werden die Daten vom Arbeitsspeicher der CPU in den Constant Memory kopiert [5].

Das „atoms“-Array im Code 28 ist das Array, das im Constant Memory auf der GPU angelegt wird. Das „atoms_h“-Array hingegen befindet sich im Host Memory und die Daten von diesem Array werden auf das Array im Constant Memory der GPU übertragen.

Code 28: Verwendung von Constant Memory [1] [5] [6] [43] (adaptiert)

CUDA	OpenCL
Im Host-Code außerhalb der main-Methode: <pre>__constant__ Datentyp atoms[CHUNK_SIZE*4];</pre> In der main-Methode: <pre>cudaMemcpyToSymbol(atoms, atoms_h, CHUNK_SIZE*4*sizeof(Datentyp));</pre> Kernelcode: <pre>__global__ void Kernelname(Parameter) { Variable=atoms[atomid*4]; }</pre>	Im Host-Code innerhalb der main-Methode: <pre>cl_mem atoms = clCreateBuffer(clGPUContext, CL_MEM_READ_ONLY, CHUNK_SIZE*4*sizeof(Datentyp), NULL, NULL);</pre> <pre>clEnqueueWriteBuffer(clCommandQue, atoms, CL_TRUE, 0, CHUNK_SIZE*4* sizeof(Datentyp), atoms_h, 0, NULL, NULL);</pre> <pre>clSetKernelArg(clKernel, 5, sizeof(cl_mem), &atoms);</pre> Kernelcode: <pre>__kernel void Kernelname(Parameter, __constant Datentyp* atoms) { Variable=atoms[atomid*4]; }</pre>

3.3.10 Synchronisationsfunktion nach Kernelaufruf

Nach dem letzten Kernelaufruf muss eine Synchronisationsfunktion auf der Host-Seite aufgerufen werden, damit die CPU wartet, bis alle Kernels vollständig ausgeführt worden sind [5]. Im Code 29 werden die möglichen Synchronisationsfunktionen dargestellt. In CUDA wurde vor Toolkit 4.0 „cudaThreadSynchronize()“ verwendet [46]. Seit Toolkit 4.0 soll stattdessen die Funktion „cudaDeviceSynchronize()“ verwendet werden [5]. Bei OpenCL wird die Funktion „clFinish()“ benutzt [47]. Als Parameter muss bei der OpenCL-Funktion die Command Queue übergeben werden [47].

Code 29: Synchronisationsfunktion nach Kernelaufruf

CUDA	OpenCL
In Toolkit 3.2: [46] <code>cudaThreadSynchronize();</code>	Variante 1: [47] <code>clFinish(clCommandQueue);</code>
In Toolkit 4.0: [5] <code>cudaDeviceSynchronize();</code>	Variante 2: [1] <code>cl_event event; clEnqueueNDRangeKernel(clcmdq, clkernel, 2, NULL, Gsz, Bsz, 0, NULL, &event); clWaitForEvents(1, &event); clReleaseEvent(event);</code>

3.3.11 Synchronisationsfunktion im Kernel

Bei CUDA und OpenCL können Threads mit anderen Threads, die sich innerhalb desselben Thread-Blocks befinden, Daten über den Shared Memory austauschen [5] [6]. Um sicherstellen zu können, dass alle Threads eines Blocks die Befehle bis zu einer Codezeile des Kernelcodes durchgeführt haben, muss eine Synchronisationsfunktion im Kernel aufgerufen werden [5]. Im Code 30 steht die notwendige Synchronisationsfunktion für CUDA und OpenCL.

Code 30: Synchronisationsfunktion im Kernel [5] [6]

CUDA	OpenCL
<code>__syncthreads();</code>	<code>barrier(CLK_LOCAL_MEM_FENCE);</code>

3.3.12 Mathematische Funktionen im Kernel

Die verwendbaren mathematischen Funktionen im Kernel gibt es als Standard-Funktion, als Standard- und Intrinsic-Funktion oder nur als Intrinsic-Funktion (siehe Kapitel 3.3.12.1 und 3.3.12.2). Standard-Funktionen können im Kernel und im Host-Code aufgerufen werden [5]. Im Gegensatz dazu können Intrinsic-Funktionen nur im Kernel/Device-Code ausgeführt werden [5]. In OpenCL wird eine Intrinsic-Funktion als Native-Funktion bezeichnet [6]. Es muss beachtet werden, dass sich die Implementierung der Standard-Funktion von der Implementierung der Intrinsic-Funktion unterscheiden kann [5]. Bei 11 Single-Precision Intrinsic-Funktionen (`__fdividef(x,y)`, `__sinf(x)`, `__cosf(x)`, `__tanf(x)`, `__sincosf(x,sptr,cptr)`, `__logf(x)`, `__log2f(x)`, `__log10f(x)`, `__expf(x)`, `__exp10f(x)` und `__powf(x,y)`) ist die Implementierung schneller, aber ungenauer als die Implementierung der Standard-Funktion [5]. Bei diesen 11 Funktionen kann mit Hilfe des Flags „-use_fast_math“ beim Kompilieren des CUDA-Programms der Funktionsaufruf im Kernel von der Standard-Funktion auf die Intrinsic-Funktion automatisch geändert werden [5].

3.3.12.1 Standard-Funktionen

In CUDA gibt es für Single- und Double-Precision 72 mathematische Standard-Funktionen, die im Kernel ausgeführt werden können (siehe Tabelle 27 und Tabelle 28) [5]. Der Unterschied zwischen Single- und Double-Precision-Funktion ist, dass bei Single-Precision ein „f“ am Ende des Funktionsnamens steht (Ausnahmen: `x+y`, `x*y`, `x/y`, `1/x`, `signbit(x)`, `isinf(x)`, `isnan(x)` und `isfinite(x)`).

Tabelle 27: Single-Precision Standard-Funktionen in CUDA (Inhalt von [5])

x+y	expm1f(x)	acosf(x)	erfinvf(x)	remainderf(x,y)	llrintf(x)
x*y	logf(x)	atanf(x)	erfcinvf(x)	remquof(x,y,iptr)	llroundf(x)
x/y	log2f(x)	atan2f(y,x)	lgammaf(x)	modff(x,iptr)	signbit(x)
1/x	log10f(x)	sinhf(x)	tgammaf(x)	fdimf(x,y)	isinf(x)
rsqrtf(x)	log1pf(x)	coshf(x)	fmaf(x,y,z)	truncf(x)	isnan(x)
sqrtf(x)	sinf(x)	tanhf(x)	frexpf(x,exp)	roundf(x)	isfinite(x)
cbrtf(x)	cosf(x)	asinhf(x)	ldexpf(x,exp)	rintf(x)	copysignf(x,y)
rcbrtf(x)	tanf(x)	acoshf(x)	scalbnf(x,n)	nearbyintf(x)	fminf(x,y)
hypotf(x,y)	sincosf(x,sptr,cptr)	atanhf(x)	scalblnf(x,l)	ceilf(x)	fmaxf(x,y)
expf(x)	sinpif(x)	powf(x,y)	logbf(x)	floorf(x)	fabsf(x)
exp2f(x)	cospif(x)	erff(x)	ilogbf(x)	lrintf(x)	nanf(cpctr)
exp10f(x)	asinf(x)	erfcf(x)	fmodf(x,y)	lroundf(x)	nextafterf(x,y)

Tabelle 28: Double-Precision Standard-Funktionen in CUDA (Inhalt von [5])

x+y	expm1(x)	acos(x)	erfinv(x)	remainder(x,y)	llrint(x)
x*y	log(x)	atan(x)	erfcinv(x)	remquo(x,y,iptr)	llround(x)
x/y	log2(x)	atan2(y,x)	lgamma(x)	modf(x,iptr)	signbit(x)
1/x	log10(x)	sinh(x)	tgamma(x)	fdim(x,y)	isinf(x)
rsqrt(x)	log1p(x)	cosh(x)	fma(x,y,z)	trunc(x)	isnan(x)
sqrt(x)	sin(x)	tanh(x)	frexp(x,exp)	round(x)	isfinite(x)
cbrt(x)	cos(x)	asinh(x)	ldexp(x,exp)	rint(x)	copysign(x,y)
rcbrt(x)	tan(x)	acosh(x)	scalbn(x,n)	nearbyint(x)	fmin(x,y)
hypot(x,y)	sincos(x,sptr,cptr)	atanh(x)	scalbln(x,l)	ceil(x)	fmax(x,y)
exp(x)	sinpi(x)	pow(x,y)	logb(x)	floor(x)	fabs(x)
exp2(x)	cospi(x)	erf(x)	ilogb(x)	lrint(x)	nan(cpctr)
exp10(x)	asin(x)	erfc(x)	fmod(x,y)	lround(x)	nextafter(x,y)

In OpenCL gibt es für Single- und Double-Precision 57 mathematische Standard-Funktionen, die im Kernel ausgeführt werden können (siehe Tabelle 29 und Tabelle 30) [6]. Im Vergleich zu CUDA fehlen bei OpenCL 15 Funktionen (rcbrt(x), sinpi(x), cospi(x), erfinv(x), erfcinv(x), signbit(x), isinf(x), isnan(x), isfinite(x), copysign(x,y), fmin(x,y), fmax(x,y), fabs(x), nan(cpctr) und nextafter(x,y)) [5] [6]. Außerdem sind bei OpenCL die Funktionsnamen von Single-Precision- und Double-Precision-Funktionen identisch [6]. Ein weiterer Unterschied ist, dass die sincos-Funktion laut Nvidia-PDF „OpenCL Programming Guide for the CUDA Architecture“ [6] bei Single-Precision nur 2 statt 3 Parameter hat.

Tabelle 29: Single-Precision Standard-Funktionen in OpenCL (Inhalt von [6])

x+y	exp10(x)	asin(x)	pow(x,y)	logb(x)	nearbyint(x)
x*y	expm1(x)	acos(x)	erf(x)	ilogb(x)	ceil(x)
x/y	log(x)	atan(x)	erfc(x)	fmod(x,y)	floor(x)
1/x	log2(x)	atan2(y,x)	lgamma(x)	remainder(x,y)	lrint(x)
rsqrt(x)	log10(x)	sinh(x)	tgamma(x)	remquo(x,y,iptr)	lround(x)
sqrt(x)	log1p(x)	cosh(x)	fma(x,y,z)	modf(x,iptr)	llrint(x)
cbrt(x)	sin(x)	tanh(x)	frexp(x,exp)	fdim(x,y)	llround(x)
hypot(x,y)	cos(x)	asinh(x)	ldexp(x,exp)	trunc(x)	
exp(x)	tan(x)	acosh(x)	scalbn(x,n)	round(x)	
exp2(x)	sincos(x,cptr)	atanh(x)	scalbln(x,l)	rint(x)	

Tabelle 30: Double-Precision Standard-Funktionen in OpenCL (Inhalt von [6])

x+y	exp10(x)	asin(x)	pow(x,y)	logb(x)	nearbyint(x)
x*y	expm1(x)	acos(x)	erf(x)	ilogb(x)	ceil(x)
x/y	log(x)	atan(x)	erfc(x)	fmod(x,y)	floor(x)
1/x	log2(x)	atan2(y,x)	lgamma(x)	remainder(x,y)	lrint(x)
rsqrt(x)	log10(x)	sinh(x)	tgamma(x)	remquo(x,y,iptr)	lround(x)
sqrt(x)	log1p(x)	cosh(x)	fma(x,y,z)	modf(x,iptr)	llrint(x)
cbrt(x)	sin(x)	tanh(x)	frexp(x,exp)	fdim(x,y)	llround(x)
hypot(x,y)	cos(x)	asinh(x)	ldexp(x,exp)	trunc(x)	
exp(x)	tan(x)	acosh(x)	scalbn(x,n)	round(x)	
exp2(x)	sincos(x,sptr, cptr)	atanh(x)	scalbln(x,l)	rint(x)	

3.3.12.2 Intrinsic/Native-Funktionen

Für die Verwendung einer Intrinsic-Funktion werden bei CUDA zwei Unterstriche und bei OpenCL „native_“ vor dem Funktionsnamen gestellt [5] [6]. Bei CUDA hat man im Gegensatz zu OpenCL eine Unterscheidung zwischen Single-Precision (=float) und Double-Precision (=double) im Namen der Funktion [5] [6]. Der Buchstabe f wird für Single-Precision und der Buchstabe d für Double Precision verwendet [5]. Der Buchstabe befindet sich entweder am Beginn oder am Ende oder an beiden Enden des Funktionsnamens (z. B. fdividef) [5]. Die Double-Precision Intrinsic-Funktion fma ist die einzige Funktion, bei der der Buchstabe nicht angegeben wird (siehe Tabelle 32) [5].

In CUDA gibt es für Single-Precision 17 und für Double-Precision 6 mathematische Intrinsic-Funktionen, die im Kernel ausgeführt werden können (siehe Tabelle 31 und Tabelle 32) [5]. Im Vergleich dazu gibt es in OpenCL für Single-Precision 12 mathematische Intrinsic-Funktionen, die im Kernel ausgeführt werden können (siehe Tabelle 31) [6]. Mathematische Intrinsic-Funktionen für Double-Precision in OpenCL werden im Nvidia-PDF „OpenCL Programming Guide for the CUDA Architecture“ [6] nicht angegeben.

Tabelle 31: Single-Precision Intrinsic-Funktionen für CUDA und OpenCL (Inhalt von [5] [6])

CUDA	OpenCL
__fdividef(x,y)	native_divide(x,y)
__sinf(x)	native_sin(x)
__cosf(x)	native_cos(x)
__tanf(x)	native_tan(x)
__logf(x)	native_log(x)
__log2f(x)	native_log2(x)
__log10f(x)	native_log10(x)
__expf(x)	native_exp(x)
__exp10f(x)	native_exp10(x)
__powf(x,y)	native_pow(x,y)
__sincosf(x,sptr,cptr)	
__frcp_[rn,rz,ru,rd] (x)	native_recip(x)
__fsqrt_[rn,rz,ru,rd] (x)	native_sqrt(x)
__fmaf_[rn,rz,ru,rd] (x,y,z)	
__fadd_[rn,rz,ru,rd] (x,y)	
__fmul_[rn,rz,ru,rd] (x,y)	
__fdiv_[rn,rz,ru,rd] (x,y)	

Tabelle 32: Double-Precision Intrinsic-Funktionen für CUDA und OpenCL (CUDA-Funktionen von [5], Informationen von [5] [6])

CUDA	OpenCL
__dadd_[rn,rz,ru,rd] (x,y)	Im „OpenCL Programming Guide for the CUDA Architecture“ [6] sind nur die OpenCL Single-Precision Intrinsic-Funktionen angegeben.
__dmul_[rn,rz,ru,rd] (x,y)	
__fma_[rn,rz,ru,rd] (x,y,z)	
__ddiv_[rn,rz,ru,rd] (x,y) - ab Compute Capability 2	
__drcp_[rn,rz,ru,rd] (x) - ab Compute Capability 2	
__dsqrt_[rn,rz,ru,rd] (x) - ab Compute Capability 2	

3.3.13 Double Precision

Damit Double-Precision statt Single-Precision verwendet wird, sind bei CUDA und OpenCL neben der Änderung des Datentyps von float auf double weitere Schritte notwendig.

Bei CUDA muss das Programm mit dem Flag „-arch=sm_13“ (Befehl: nvcc -arch=sm_13 ./program) kompiliert werden [23]. Mit dem Flag „-arch=sm_13“ werden alle GPUs, die eine Compute Capability kleiner als 1.3 haben, ausgeschlossen [23]. Das ist notwendig, da erst mit Compute Capability 1.3 Double-Precision auf Nvidia-GPUs möglich ist [23]. Die dritte Änderung neben Datentyp double und Compilerflag „-arch=sm_13“ betrifft die Funktionsnamen von mathematischen Operationen (z. B. rsqrt()). Diese müssen angepasst werden, da es einen Unterschied beim Funktionsnamen zwischen Single- und Double-Precision gibt (siehe Kapitel 3.3.12.1) [5].

Bei OpenCL muss vor der Kernelfunktion „#pragma OPENCL EXTENSION cl_khr_fp64 : enable“ eingefügt werden [48]. Im Gegensatz zu CUDA muss der Funktionsname von mathematischen Funktionen (z. B. rsqrt()) nicht geändert werden, da der Single-Precision-Funktionsname und der Double-Precision-Funktionsname gleich ist (siehe Kapitel 3.3.12.1) [5] [6].

4 Benützte Hardware und Compiler

4.1 Nvidia-GTX-Konfiguration

Die Nvidia-GTX-Konfiguration besitzt als Betriebssystem x86_64 Linux RHEL (=Red Hat Enterprise Linux) Server mit der Version 5.6 (Tikanga). Dies wurde über den Befehl „cat /etc/issue“ ermittelt. Als Compiler wird icc Version 11.1 (Befehl: icc -V), gcc Version 4.1.2 (Befehl: gcc --version) und nvcc- bzw. OpenCL-Compiler aus CUDA-Toolkit Version 4.0 (Befehl: nvcc --version) verwendet.

In der nachfolgenden Tabelle 33 ist die Hardware der Nvidia-GTX-Konfiguration dargestellt.

Tabelle 33: Nvidia-GTX-Hardware-Konfiguration (Quellen am Ende der Tabelle)

	Intel Xeon X5550	Nvidia GTX 480	Nvidia GTX 285	ATI Radeon 5870
Verwendungszweck	CPU	Aktuelle GPU	Ältere GPU	
Architektur	Nehalem-EP	Fermi	GT200	Cypress
Art des Prozessors	CPU	GPU	GPU	GPU
Stückanzahl	2	1	1	1
Peak GFLOP/s (Single-Precision)	85,12 (=4*4 FMA*2*2,66)	1345 (=480 FMA*2*1,401)	1062,72 (=240 MAD*2+240 MUL)*1,476)	2720 (=1600 FMA*2*0,85)
Peak GFLOP/s (Double-Precision)	42,56 (=4*2 FMA*2*2,66)	Eigentlich 672,48 (=240 FMA*2* 1,401), wurde aber reduziert auf 25% der Leistung → 168,12	88,56 (=30 FMA*2*1,476)	544 (=320 FMA*2*0,85)
Anzahl von Kerne	4	480	240	1600
Taktfrequenz	2,66 GHz / Turbo Frequenz 3,06 GHz	1401 MHz	1476 MHz	850 MHz
Anzahl von Multiprozessoren/ SM		15	30	20
Maximale Anzahl von aktiven Threads	8 (=2 Threads*4 Kerne)	23040 Threads (=32 Threads*48 Warps*15 SM)	30720 Threads (=32 Threads*32 Warps*30 SM)	
Maximale Anzahl von aktiven Thread- Blöcken		120 (=8 Blöcke*15 SM)	240 (=8 Blöcke*30 SM)	
Gesamter Cache + Shared Memory	9,25 MB	1,6875 MB	480 KB	1,28 MB
L3-Cache	8 MB (shared)			
L2-Cache	1 MB (=4 Kerne*256 KB)	768 KB (shared)		512 KB (=128 KB*4 Bänke)

Fortsetzung der Tabelle 33 auf der nächsten Seite

Fortsetzung der Tabelle 33:

L1-Cache	256 KB (=4 Kerne*32 KB Instruction Cache + 4 Kerne*32 KB Data Cache)	Konfigurierbar: 240 KB (=15 SM*16 KB) oder 720 KB (=15 SM*48 KB)		160 KB (=8 KB*20 SM)
Shared Memory		Konfigurierbar: 720 KB (=15 SM*48 KB) oder 240 KB (=15 SM*16 KB)	480 KB (=30 SM*16 KB)	640 KB=(20 SM*32 KB)
Constant Memory		64 KB	64 KB	64 KB
Register Memory		1920 KB (=128 KB *15 SM)	1920 KB (=64 KB *30 SM)	640 KB (=32 KB *20 SM)
Arbeitsspeicher	24GB DDR3-1333	1,5 GDDR5	1GB GDDR3	0,5 GB GDDR5
Bandbreite zwischen Chip und Arbeitsspeicher	32 GB/s	177 GB/s (=1,848*2*384/8)	159 GB/s (=1,242*2*512/8)	154 GB/s
Bandbreite des Shared Memorys		1345 GB/s (=2 Byte pro Bank * 32 Bänke * 15 SM * 1,401)	1417 GB/s (=2 Byte pro Bank * 16 Bänke * 30 SM * 1,476)	
ECC	Ja	Nein	Nein	
PCI-Anschluss		PCIe 2.0 x16	PCIe 2.0 x16	PCIe 2.1 x16
OpenCL-Version		OpenCL 1.0 CUDA	OpenCL 1.0 CUDA	OpenCL 1.1 ATI-Stream-v2.3 (451)
OpenCL-Programm wird kompiliert in		PTX = Parallel Thread eXecution	PTX = Parallel Thread eXecution	CAL/IL = Compute Abstraction Layer/ Intermediate Language
Nvidia Compute Capability		2.0	1.3	
Maximale Blockgröße		1024	512	256
DMA Engines		1		
Special Function Units		60 (=4*15 SM)	60 (=2*30 SM)	320 (=16*20 SM)
Warp schedulers (pro SM)		2	1	
Anzahl von Concurrent Kernels		Bis zu 16		
Max TDP	95 W		204 W	188 W
Chiptakt		700 MHz	648MHz	

Quellen der Tabelle 33: [5] [7] [8] [16] [19] [20] [21] [22] [49] [50] [51] [52] [53] [54] [55]
[Hardwaredatenermittlung-OpenCL-Programm (Funktion „clGetDeviceInfo“ wurde mit unterschiedlichen Parameter aufgerufen)]

4.2 Nvidia-Tesla-Konfiguration

Die Nvidia-Tesla-Konfiguration besitzt als Betriebssystem x86_64 Linux RHEL (=Red Hat Enterprise Linux) Server Version 5.6 (Tikanga). Dies wurde über den Befehl „cat /etc/issue“ ermittelt. Als Compiler wird icc Version 11.1 (Befehl: icc -V), gcc Version 4.1.2 (Befehl: gcc --version) und nvcc- bzw. OpenCL-Compiler aus CUDA-Toolkit Version 3.2 (Befehl: nvcc --version) verwendet.

Die Nvidia-Tesla-Konfiguration besteht aus 8 Computational Nodes und einem Front End. Für die Masterarbeit wurden die Programme nur auf einem Computational Node ausgeführt.

In der nachfolgenden Tabelle 34 ist die Hardware eines Knoten der Nvidia-Tesla-Konfiguration dargestellt.

Tabelle 34: Nvidia-Tesla-Hardware-Konfiguration (Quellen am Ende der Tabelle)

	Intel Xeon X5550	Nvidia Tesla C2050	Nvidia Tesla C1060
Architektur	Nehalem-EP	Fermi	GT200
Prozessortyp	CPU	GPU	GPU
Stückanzahl	2	2	1
Peak GFLOP/s (Single-Precision)	85,12 (=4*4 FMA*2*2,66)	1027,7 (=448 FMA*2*1,147)	933,12 (=240 MAD*2+240 MUL)*1,296)
Peak GFLOP/s (Double-Precision)	42,56 (=4*2 FMA*2*2,66)	513,86 (=224 FMA*2*1,147)	77,76 (=30 FMA*2*1,296)
Anzahl von Kerne	4	448	240
Taktfrequenz	2,66 GHz / Turbo Frequenz 3,06 GHz	1147 MHz	1296 MHz
Anzahl von Multiprozessoren/SM		14	30
Maximale Anzahl von aktiven Threads	8 (=2 Threads*4 Kerne)	21504 Threads (=32 Threads*48 Warps*14 SM)	30720 Threads (=32 Threads*32 Warps*30 SM)
Maximale Anzahl von aktiven Thread- Blöcken		112 (=8 Blöcke*14 SM)	240 (=8 Blöcke*30 SM)
Gesamter Cache + Shared Memory	9,25 MB	1,625 MB	480 KB
L3-Cache	8 MB (shared)		
L2-Cache	1 MB (=4 Kerne*256 KB)	768 KB (shared)	
L1-Cache	256 KB (=4 Kerne*32 KB Instruction Cache+4 Kerne*32 KB Data Cache)	Konfigurierbar: 224 KB (=14 SM*16 KB) oder 672 KB (=14 SM*48 KB)	
Shared Memory		Konfigurierbar: 224 KB (=14 SM*16 KB) oder 672 KB (=14 SM*48 KB)	480 KB (=30 SM*16 KB)
Constant Memory		64 KB	64 KB

Fortsetzung der Tabelle 34 auf der nächsten Seite

Fortsetzung der Tabelle 34:

Register Memory		1792 KB (=128 KB *14 SM)	1920 KB (=64 KB*30 SM)
Arbeitsspeicher	24GB DDR3-1333	2,625 GB GDDR5 (ECC ist aktiviert, ohne ECC -> 3GB)	4GB GDDR3
Bandbreite zwischen Chip und Arbeitsspeicher	32 GB/s	144 GB/s (=1,5*2*384/8), 115 GB/s mit ECC	102 GB/s (=0,8*2*512/8)
Bandbreite des Shared Memorys		1028 GB/s (=2 Byte pro bank * 32 Bänke * 14 SM*1,147)	1244 GB/s (=2 Byte pro bank * 16 Bänke * 30 SM*1,296)
Bandbreite des L2-Cache		230 GB/s	
ECC	Ja	Ja	Nein
PCI-Anschluss		PCIe 2.0 x16	PCIe 2.0 x16
OpenCL-Version		OpenCL 1.0 CUDA	OpenCL 1.0 CUDA
OpenCL-Programm wird kompiliert in		PTX = Parallel Thread eXecution	PTX = Parallel Thread eXecution
Nvidia Compute Capability		2.0	1.3
Maximale Blockgröße		1024	512
DMA Engines		2	
Special Function Units		56 (=14 SM*4)	60 (=30 SM*2)
Warp schedulers		28 (=14 SM*2)	30 (=30 SM*1)
Anzahl von Concurrent Kernels		Bis zu 16	
Max TDP	95 W	238 W	188 W
Anzahl von Transistoren	731 Millionen	3 Milliarden	1,2 Milliarden

Quellen der Tabelle 34: [5] [6] [7] [8] [19] [20] [22] [25] [26] [27] [36] [49] [51] [52] [55] [56] [Hardwaredatenermittlung-OpenCL-Programm (Funktion "clGetDeviceInfo" wurde mit unterschiedlichen Parameter aufgerufen)]

5 Laufzeitmessung des Programmcodes

Für einen fairen Vergleich der Laufzeit zwischen CPU- und GPU-Programm werden bei dem CPU-Programm der optimale Compiler `icc` (statt `gcc`) und die Compileroptimierungsflags `-O3` und `-xSSE3` benützt (siehe Kapitel 7.2.1.4 und Kapitel 8.2.1.2). Auch beim Host-Code des GPU-Programms wird der Code mit dem optimalen Compiler `icc` und dem Compilerflag `-O3` kompiliert (siehe Unterkapitel 5.3). Der `icc`-Compiler und das Compilerflag `-O3` werden also durchgehend verwendet. Bei der GPU-Gesamtlaufzeit wird das Anlegen und Freigeben der Arrays im GPU-Arbeitsspeicher und das Übertragen der Daten vom CPU-Arbeitsspeicher in den GPU-Arbeitsspeicher und zurück inkludiert (siehe Unterkapitel 5.3.1), da diese Schritte bei der Ausführung eines Algorithmus auf einer Grafikkarte im Vergleich zur Ausführung auf einer CPU zusätzlich notwendig sind und somit fairerweise dazugerechnet werden müssen. Bei der Frage, ob das GPU-Programm einen Speedup gegenüber dem CPU-Programm erreichen kann, wird das GPU-Programm mit dem parallelen CPU-Programm, das alle Kerne der CPU verwendet, verglichen und somit ein fairer Vergleich bei Ausnützung aller Ressourcen der beiden Chips ermöglicht (siehe Kapitel 9.7). Da sich bei der benützten Hardware zwei CPUs auf dem Mainboard befinden, wurde beim Ausführen des

parallelen CPU-Programms nur eine CPU ausgewählt, um einen fairen Vergleich zwischen einer CPU und einer GPU zu ermöglichen (siehe Unterkapitel 5.2).

5.1 Messmethode

Zur Messung der Laufzeit wird die Funktion `gettimeofday()` verwendet. Im nachfolgenden Code 31 ist die verwendete Laufzeitmessung dargestellt.

Code 31: Messung der Laufzeit eines Programmcodes [57] (adaptiert)

```
...
#include <sys/time.h>
...
double ZeitdauerBerechnen(const struct timeval begin, const struct timeval end);
...
int main(int argc, char **argv)
{
    ...
    struct timeval startTime, endTime;
    gettimeofday(&startTime, (struct timezone *)0);
    //Hier steht der Code, dessen Laufzeit gemessen werden soll
    gettimeofday(&endTime, (struct timezone *)0);

    double dauer=ZeitdauerBerechnen(startTime, endTime);
    ...
    return EXIT_SUCCESS;
}
...
double ZeitdauerBerechnen(const struct timeval begin, const struct timeval end)
{
    double laufzeit;
    long sekunden, mikrosekunden;

    sekunden = end.tv_sec - begin.tv_sec;
    mikrosekunden = end.tv_usec - begin.tv_usec;
    if(mikrosekunden < 0){
        mikrosekunden += 1000000;
        sekunden--;
    }

    laufzeit = sekunden+(double)mikrosekunden/1000000;

    return laufzeit;
}
```

Mit Hilfe eines Shell-Skripts (siehe Code 32) wird das Programm mehrmals mit unterschiedlichen Problemgrößen ausgeführt. Dazu wird die Startgröße, Endgröße und der Anstieg der Problemgröße an das Shell-Skript übergeben. Sobald die Endgröße erreicht ist, wird wieder ein neuer Durchgang bei der Startgröße begonnen. Die Anzahl der Durchgänge wird mit dem ersten Parameter des Shell-Skripts festgelegt. In der Masterarbeit wurde der Parameter für die Anzahl der Durchgänge immer auf 10 gesetzt. Das Ergebnis des Shell-Skripts wird in der Textdatei „Laufzeiten.txt“ gespeichert. Der Inhalt der Textdatei wird nach Ersetzen von „.“ mit „.“ in ein Excel-Dokument übernommen. Dort wird von den 10 Laufzeitmessungen einer Problemgröße der Mittelwert mit der „MITTELWERT()“-Funktion ermittelt. Diese Mittelwerte werden für die Berechnung der Speedupwerte (siehe Kapitel 5.4), die in den Diagrammen dargestellt werden, benützt.

Code 32: Shell-Skript für das wiederholte Ausführen des Programms mit unterschiedlichen Problemgrößen

```
#!/bin/bash

programm=needlemanWunsch

if [ $# -lt 8 ]
then
    echo -n "Usage: $0 <Anzahl Wiederholungen> <startgroesse> <endgroesse> "
    echo -n "<anstieg> <Gap-Kosten> <Korrektheit ueberpruefen (1=ja,0=nein)> "
    echo "<device_nr (0 .. erstes Device)> <cpu id>"
    exit 0
fi

w=0
while [ $w -lt $1 ]
do
    echo "spaltenamen" >> Laufzeiten.txt
    n=$2
    i="0"
    echo "Durchgang: $w"
    while [ $n -lt $3 ]
    do
        echo "Ausfuehren des Programms $programm mit N=$n"
        numactl --cpunodebind=$8 ./$programm $n $n $5 $6 $7 >> Laufzeiten.txt
        n=$((n+$4))
    done
    echo "Ausfuehren des Programms $programm mit N=$n"
    numactl --cpunodebind=$8 ./$programm $n $n $5 $6 $7 >> Laufzeiten.txt
    echo ""
    w=$((w+1))
done

echo "Die gemessenen Laufzeiten wurden in die Datei Laufzeiten.txt gespeichert!"
```

5.2 CPU-Programm

Beim CPU-Programm werden zwei Laufzeiten gemessen (siehe Unterkapitel 5.2.1 und Unterkapitel 5.2.2). Zur korrekten Laufzeitmessung der CPU-Laufzeit wird ein eigenes CPU-C-Programm verwendet. Als Compiler wird der icc (=Intel C Compiler) mit der Version 11.1 verwendet. Beim Kompilieren wird immer das Optimierungsflag „-O3“ verwendet.

Damit für die Laufzeitmessung nur eine und nicht zwei CPUs auf dem Mainboard verwendet werden, muss mit „numactl --cpubind=x ./programm“ das Programm aufgerufen werden und statt x die Zahl 0 (=erste CPU) oder 1 (=zweite CPU) eingesetzt werden. Für die Laufzeitmessungen des parallelen CPU-Programms wurde die zweite CPU (x=1) ausgewählt. Somit kann ein fairer Vergleich zwischen einer CPU und einer GPU durchgeführt werden.

5.2.1 CPU-Gesamtlaufzeit

Für diese Laufzeit wird die Zeitdauer für das Anlegen des Speichers (malloc()), das Ausführen der Funktion (CPU-Funktionslaufzeit) und das Freigeben des Speichers (free()) inkludiert.

5.2.2 CPU-Funktionslaufzeit

Diese Laufzeit enthält nur das Ausführen der Funktion.

5.3 GPU-Programm

Beim GPU-Programm werden zwei Laufzeiten gemessen (siehe Unterkapitel 5.3.1 und Unterkapitel 5.3.2). Als CUDA- und OpenCL-Compiler wird CUDA-Toolkit 3.2 für die Tesla-GPUs und CUDA-Toolkit 4.0 für die GeForce-GPUs verwendet. Mit dem Flag „-ccbin icc --compiler-options "-O3 -isystem/userHomeVerzeichnis"“ wird der Hostteil des GPU-Programms mit -O3 und dem icc-Compiler kompiliert. Der gesamte Kompilierbefehl lautet „nvcc -o Programm ProgrammSource.cu -ccbin icc --compiler-options "-O3 -isystem/userHomeDir"“. Für das Kompilieren des Hostteils von OpenCL-Programmen mit dem icc-Compiler und -O3 muss der Befehl „icc -U __MMX__ -o Programm ProgrammSource.c -lm -O3 -I/cudaVerzeichnis/include -lOpenCL“ verwendet werden.

Das Flag „-lm“ beim OpenCL-Kompilierbefehl steht für das Einbinden der Math Library. Mit „#include <math.h>“ im Code kann zum Beispiel die Funktion sqrt() im Hostteil des Programms verwendet werden. Die Funktion sqrt() benötigt man für die CPU-Funktion des DCS-Algorithmus, um mit den Ergebnissen der CPU-Funktion die Korrektheit der GPU-Funktion überprüfen zu können. Diese Überprüfung wird nur nach Änderungen im Code durchgeführt. Sobald feststeht, dass mit der Änderung im Code das Ergebnis der Berechnung weiterhin korrekt ist, wird die CPU-Funktion nicht mehr durchgeführt. Beim Aufruf des Programms wird mit einem Parameter festgelegt, ob diese Überprüfung durchgeführt werden soll.

Bei beiden Algorithmen müssen bestimmte Parameter übergeben werden, damit das Programm startet. Bei der Laufzeitmessung wird immer der Parameter „Korrektheit überprüfen“ auf 1 gesetzt. Dadurch wird in einer eigenen CPU-Funktion nach der kompletten Berechnung auf der GPU dieselbe Berechnung nochmals ausgeführt. Da die Laufzeit immer punktuell (über GPU-Kernel, über Host-Speicher anlegen, über Host-Speicher freigeben, ...) gemessen wird und dann diese Laufzeiten zusammenaddiert werden, verfälscht die CPU-Funktion nicht die gemessene GPU-Laufzeit. Der einzige Nachteil ist, dass das Ausführen des Programms bei der Laufzeitmessung länger dauert. Der große Vorteil ist, dass man dadurch in der Laufzeitdatei ablesen kann, ob das Programm bei allen Problemgrößen und Optimierungen das korrekte Ergebnis geliefert hat. Im Kapitel 6 wird die Überprüfung der Korrektheit genauer erklärt.

Das DCS-Programm wird mit den Argumenten „<Anzahl von Atomen> <Anzahl von Gitterpunkten - X Richtung> <Anzahl von Gitterpunkten - Y Richtung> <Anzahl von Gitterpunkten - Z Richtung> <Abstand zwischen Gitterpunkten> <Blocksize in X-Richtung> <Blocksize in Y-Richtung> <Korrektheit ueberpruefen (1=ja,0=nein)> <GPU (0 ... erste GPU, 1 ... zweite GPU)>“ aufgerufen.

Das Needleman-Wunsch-Programm wird mit den Argumenten „<Laenge fuer zufaellige 1.Sequenz> <Laenge fuer zufaellige 2.Sequenz> <Gap-Kosten> <Korrektheit ueberpruefen (1=ja,0=nein)> <device_nr (0 ... erstes Device)> <Blockgroesse>“ aufgerufen.

Da sich 2 CPUs auf dem Mainboard befinden und sich die PCI-Bandbreite je nach gewählter CPU vor allem beim Senden von Daten vom Device zum Host unterscheidet (siehe Kapitel 2.3), wird immer die optimale CPU bei den Laufzeitmessungen genommen. Bei zwei GPUs wird bei der Ausführung auf der ersten GPU die erste CPU und bei der Ausführung auf der zweiten GPU die zweite CPU verwendet. Bei drei GPUs wird bei der Ausführung auf der ersten oder zweiten GPU die erste CPU und bei der Ausführung auf der dritten GPU die zweite CPU benutzt. Um die jeweilige CPU für das Programm auszuwählen, muss vor dem Programmaufrufbefehl „numactl --cpunodebind=0“ (=CPU 1) bzw. „numactl --cpunodebind=1“ (=CPU 2) eingefügt werden. Würde man das nicht machen, bekommt man bei 10 Laufzeitmessungen Werte heraus, die zwischen zwei Werten schwanken, je nachdem welche CPU für die Ausführung ausgewählt wurde. Der Mittelwert wäre somit abhängig, wie oft bei den 10 Laufzeitmessungen welche CPU gearbeitet hat.

5.3.1 GPU-Gesamtlaufzeit

In der GPU-Gesamtlaufzeit ist das Anlegen der Arrays im CPU-Arbeitsspeicher (malloc() oder cudaMallocHost()), das Anlegen der Arrays im GPU-Arbeitsspeicher (cudaMalloc()), das Übertragen der Arrays vom CPU-Arbeitsspeicher in den GPU-Arbeitsspeicher (cudaMemcpy()), das Ausführen des Kernels (GPU-Kernellaufzeit), das Übertragen der Arrays vom GPU-Arbeitsspeicher in den CPU-Arbeitsspeicher (cudaMemcpy()), das Freigeben der Arrays im GPU-Arbeitsspeicher (cudaFree()) und das Freigeben der Arrays im CPU-Arbeitsspeicher (free() oder cudaFreeHost()) inkludiert.

Bei CUDA kann es zu einer Verfälschung der Laufzeitmessung beim ersten Aufruf einer Funktion (z. B. cudaMallocHost() oder cudaMalloc()) aus der CUDA-Runtime-Bibliothek kommen. Der Grund ist, dass CUDA beim ersten Aufruf einer Funktion selbst geladen wird. Damit es nicht dazukommt, wird der bereits im Kapitel 3.3.2 gezeigte Code 19, der das Laden von CUDA auslöst, eingefügt. Dadurch sind dann alle folgenden Laufzeitmessungen korrekt.

In der GPU-Gesamtlaufzeit ist nicht die CUDA- bzw. OpenCL-Initialisierung enthalten.

5.3.2 GPU-Kernellaufzeit

Mit der GPU-Kernellaufzeit wird nur das Ausführen des Kernels gemessen.

Im Code 33 steht der CUDA-Code und im Code 34 steht der OpenCL-Code für die Laufzeitmessung des Kernels. In beiden Codes steht nach dem letzten Kernelaufruf eine Synchronisationsfunktion, da die CPU auf die Fertigstellung aller Kernels warten muss, bevor die Zeit gemessen wird. Der Grund ist, dass sofort im Host-Code weitergegangen wird, sobald der Kernel-Aufruf erfolgt ist [5]. Durch die asynchronen Kernelaufufe können auf der CPU während des Ausführens der GPU-Kernels ebenfalls Berechnungen durchgeführt werden [5]. Asynchrone Kernelaufufe können in CUDA deaktiviert werden, indem die Umgebungsvariable „CUDA_LAUNCH_BLOCKING“ auf 1 gesetzt wird [5]. Beim CUDA-Debugger und CUDA-Visual-Profiler wird dies durchgeführt und somit werden alle Kernelaufufe synchron ausgeführt [5]. In diesem Fall wäre die Synchronisationsfunktion nicht notwendig.

Das Einfügen einer Synchronisationsfunktion zwischen den Kernelaufrufen ist nicht notwendig, da immer nur 1 Kernel pro Zeitpunkt auf der GPU ausgeführt wird [5]. Es wird also eine Warteschlange abgearbeitet. Die Warteschlange steht somit für die implizite Synchronisation zwischen den Kernelaufrufen. Die Warteschlange wird als Stream bezeichnet [5]. Bei der Fermi-Architektur können zwar 16 Kernels gleichzeitig ausgeführt werden, jedoch müssen diese dann explizit unterschiedlichen Streams zugeordnet werden [5]. Wenn bei einem Befehl kein Stream angegeben wird, ist dieser automatisch dem Stream 0 zugewiesen [5].

Code 33: CUDA-Code für die Laufzeitmessung des Kernels

```
gettimeofday(&startTimeGPUKernel, (struct timezone *)0);
Kernel<<<dimGrid, dimBlock>>>();
cudaThreadSynchronize(); //ab Toolkit 4.0 sollte stattdessen
                        //cudaDeviceSynchronize() verwendet werden [5]
gettimeofday(&endTimeGPUKernel, (struct timezone *)0);
```

Code 34: OpenCL-Code für die Laufzeitmessung des Kernels

```
gettimeofday(&startTimeGPUKernel, (struct timezone *)0);
clEnqueueNDRangeKernel(clCommandQueue, clKernel, 1, NULL, globalWorkSize, localWorkSize,
0, NULL, NULL);
clFinish(clCommandQueue);
gettimeofday(&endTimeGPUKernel, (struct timezone *)0);
```

5.4 Gesamtspeedup und Kernelspeedup für die Speedup-Diagramme

Es werden immer zwei Speedups des GPU-Programms ermittelt. In Formel 3 ist die Berechnung des Gesamtspeedups und in Formel 4 ist die Berechnung des Kernelspeedups angegeben.

Formel 3: Berechnung des Gesamtspeedups

$\text{Gesamtspeedup} = \text{CPUGesamtlaufzeit} / \text{GPUGesamtlaufzeit}$

Formel 4: Berechnung des Kernelspeedups

$\text{Kernelspeedup} = \text{CPUFunktionslaufzeit} / \text{GPUKernellaufzeit}$

6 Überprüfung der Korrektheit

Damit sichergestellt ist, dass der GPU-Code korrekt ist, wird im Host-Code des GPU-Programms und somit auf der CPU dieselbe Berechnung durchgeführt und mit dem GPU-Ergebnis verglichen. Durch ein Flag beim Programmaufruf wird die Überprüfung der Korrektheit aktiviert. Beim OpenMP-CPU-Code und bei den Optimierungen am CPU-Code wurde ebenfalls die Korrektheit überprüft.

6.1 NW-Algorithmus (Short- und Int-Datentyp)

Beim NW-Algorithmus kann zum Vergleichen die Integer-Zahl ganz rechts unten in der Ergebnismatrix herangezogen werden. Diese Ergebniszahl basiert auf allen anderen Ergebniszahlen der Ergebnismatrix. Somit reicht es aus, nur diese Zahl zwischen CPU und GPU zu vergleichen. Wenn beide Zahlen identisch sind, ist das GPU-Programm korrekt. Neben diesem Vergleich werden im Programm zusätzlich noch alle GPU-Ergebniswerte mit den CPU-Ergebniswerten verglichen und die Anzahl der nicht korrekten Ergebniswerte ermittelt.

6.2 DCS-Algorithmus (Float- und Double-Datentyp)

Beim DCS-Algorithmus müssen alle Gridpunkte zwischen CPU und GPU verglichen werden, da sie unabhängig voneinander berechnet werden. Damit zum Kontrollieren der Korrektheit nur ein Wert angesehen werden muss, wird die standardisierte euklidische Residual Norm berechnet. Da bei Fließkommazahlen (Float- und Double-Datentyp) im Gegensatz zu Ganzzahlen (Short- und Int-Datentyp) das Ergebnis einer Operation nicht exakt abgespeichert werden kann und das Ergebnis auf die nächstmögliche speicherbare Zahl gerundet werden muss, gibt es bei Optimierungen sehr oft kleinere Abweichungen vom exakten Ergebnis, obwohl der Programmcode korrekt ist (siehe Kapitel 2.4.2.16). Somit wird bei Fließkommazahlen nicht überprüft, ob alle Ergebniszahlen der GPU mit den Ergebniszahlen der CPU übereinstimmen, sondern es wird die standardisierte euklidische Residual Norm berechnet (siehe Code 35). Dabei werden die Abweichungen (=Residuals) zwischen den Werten im CPU-Ergebnisarray und den Werten im GPU-Ergebnisarray berechnet. Aufgrund der euklidischen Norm (=Zweiernorm) werden zunächst alle Residuals mit sich selbst multipliziert. Dann werden diese Teilergebnisse miteinander addiert und aus diesem Ergebnis wird die Quadratwurzel gezogen. Die euklidische Residual Norm ist nun fertig berechnet. Die standardisierte euklidische Residual Norm erhält man, indem die euklidische Residual Norm durch die euklidische Norm des CPU-Ergebnisarrays dividiert wird. Für die Berechnung der euklidischen Norm des CPU-Ergebnisarrays muss zunächst jeder Wert des CPU-Ergebnisarrays mit sich selbst multipliziert werden. Dann werden diese Teilergebnisse miteinander addiert und aus dem Ergebnis wird die Quadratwurzel gezogen.

Im Gegensatz zu den Ganzzahlen gibt es bei den Fließkommazahlen Nachkommastellen. Wie viele Stellen (Vorkommastellen + Nachkommastellen) insgesamt vom Datentyp abgedeckt werden können, legt die Mantisse des Datentyps fest. Beim Float-Datentyp hat man eine 23 Bit Mantisse und somit Mantissen mit Abstand von $1 \cdot 2^{-23}$ (binär) bzw. zirka $1,19209289551 \cdot 10^{-7}$ (dezimal) [58]. Die Zahl $1,19 \cdot 10^{-7}$ bedeutet, dass eine Zahl mit dem Datentyp Float 7 Stellen genau ist [58]. Beim

Datentyp Double hat man eine 52 Bit Mantisse und somit Mantissen mit Abstand von $1 \cdot 2^{-52}$ (binär) bzw. zirka $2,22044604925 \cdot 10^{-16}$ (dezimal) [58]. Die Zahl $2,22 \cdot 10^{-16}$ bedeutet, dass eine Zahl mit dem Double-Datentyp 16 Stellen genau ist [58].

Bei der bestmöglichen standardisierten Residual Norm beim Float-Datentyp folgen nach dem Komma 7 Nullen (z. B. 0,00000003) und beim Double-Datentyp 16 Nullen (siehe Erklärung im vorigen Absatz). Wenn Fehler im Programmcode sind, ist die Anzahl der nach dem Komma folgenden Nullen deutlich geringer. Wenn keine Null nach dem Komma steht, sind fast alle oder alle Ergebnisse komplett falsch. Zum Beispiel die Änderung von „dy=y-atomsy[a]“ in „dy=y-atomsy[a+1]“ im DCS-CPU-Programmcode führt zu einer standardisierten Residual Norm von 0,01709152 (N=100, M=100, Anzahl von Atomen=1000). 704 von 10000 Gridpunkten haben durch den Fehler zu einem komplett falschen Ergebnis geführt. Wenn man im DCS-CPU-Programmcode „atomsw[a]/sqrtf“ in „atomsw[a]*sqrtf“ ändert, bekommt man eine standardisierte Residual Norm von 0,50331084. Bei allen 10000 Gridpunkten ist die erste Stelle nicht korrekt und somit sind alle Ergebnisse falsch. Bei Änderung von „dy=y-atomsy[a]“ in „dy=y+atomsy[a]“ hat man eine standardisierte Residual Norm von 0,33919296 und 9294 von 10000 Gridpunkten sind komplett falsch.

Mit der Funktion „gleiche_Stellen_zweier_Zahlen_berechnen(...)“ im Code 36 kann ermittelt werden, wie viele nacheinander folgende idente Stellen es beginnend von der ersten Ziffer beim Vergleich zweier Zahlen gibt. Beim Datentyp Float ist das ideale Ergebnis 7 Stellen und beim Datentyp Double ist das ideale Ergebnis 16 Stellen. Wenn bei den entsprechenden Zeilen in der Funktion `matrizenVergleichen(...)` im Code 35 die Kommentarzeichen entfernt werden, kann im Terminal die Genauigkeit der einzelnen Ergebniswerte zusammengefasst ausgegeben werden (siehe Terminalausgabe 8). In der Terminalausgabe 8 sieht man den Vergleich der DCS-CPU-Variante 1 (siehe Code 51 im Kapitel 8.2.1.1) mit der DCS-CPU-Variante 2 (siehe Code 52 im Kapitel 8.2.1.1). Das Programm wurde mit dem `icc`-Compiler kompiliert und die Compiler-Optimierungen mit dem Flag „-O0“ deaktiviert. Die standardisierte Residual Norm ist mit 0,00000003 ideal. Bei allen Ergebnissen sind mindestens die ersten vier Stellen korrekt und bei 9366 von 10000 Gridpunkten sind alle 7 Stellen richtig. Wenn man die Ergebniszahlen, die eine Genauigkeit unter 7 haben, betrachtet, sieht man, dass sie aus mathematischer Sicht nur bei der letzten Stelle falsch sind und der Ziffernwert fast immer nur um 1 abweicht (z. B. 4 Stellen gleich: 806.0699 versus 806.0700).

Terminalausgabe 8: Anzahl der identen Stellen bei einer idealen standardisierten Residual Norm von 0,00000003 (Vergleich DCS-CPU-Variante 1 mit der DCS-CPU-Variante 2, N=100, M=100, Anzahl von Atomen=1000, Float-Datentyp, `icc -O0`)

Die ersten 0 Stellen sind gleich: 0 Gridpunkte
Die ersten 1 Stellen sind gleich: 0 Gridpunkte
Die ersten 2 Stellen sind gleich: 0 Gridpunkte
Die ersten 3 Stellen sind gleich: 0 Gridpunkte
Die ersten 4 Stellen sind gleich: 5 Gridpunkte
Die ersten 5 Stellen sind gleich: 54 Gridpunkte
Die ersten 6 Stellen sind gleich: 575 Gridpunkte
Die ersten 7 Stellen sind gleich: 9366 Gridpunkte

Mit der Funktion `residual_zweier_Zahlen_in_Ganzzahl_ermitteln(...)` im Code 37 kann die Abweichung als Ganzzahl ermittelt werden. Dazu werden die zwei zu vergleichende Zahlen um Zehnerpotenzen erhöht oder verkleinert, damit die letzte Ziffer der Zahl (7 Ziffern beim Float-Datentyp und 16 Ziffern beim Double-Datentyp) bei der ersten Stelle vor dem Komma steht. Dann kann ein Ganzzahlvergleich durchgeführt werden und die Differenz wird zurückgegeben. Wenn bei den entsprechenden Zeilen in der Funktion `matrizenVergleichen(...)` im Code 35 die Kommentarzeichen entfernt werden, kann im Terminal das Residual in Ganzzahl für die Werte 0 bis 5

und die Anzahl der Stellen der Ganzzahl-Residuals zusammengefasst ausgegeben werden (siehe Terminalausgabe 9). In der Terminalausgabe 9 sieht man den Vergleich der DCS-CPU-Variante 1 mit der DCS-CPU-Variante 2. Im Gegensatz zur Terminalausgabe 8 sieht man in Terminalausgabe 9, dass die nicht korrekten Ergebniswerte aus mathematischer Sicht nur in der letzten Ziffer einen Fehler aufweisen und die Residuals in Ganzzahl keinen höheren Wert als 2 haben. Ein Programmfehler liegt somit nicht vor.

Terminalausgabe 9: Das Residual in Ganzzahl auf Basis der Genauigkeit bei einer idealen standardisierten Residual Norm von 0,00000003 (Vergleich DCS-CPU-Variante 1 mit der DCS-CPU-Variante 2, N=100, M=100, Anzahl von Atomen=1000, Float-Datentyp, icc -O0)

Das Residual in Ganzzahl hat den Wert 0: 9366 Gridpunkte
Das Residual in Ganzzahl hat den Wert 1: 598 Gridpunkte
Das Residual in Ganzzahl hat den Wert 2: 36 Gridpunkte
Das Residual in Ganzzahl hat den Wert 3: 0 Gridpunkte
Das Residual in Ganzzahl hat den Wert 4: 0 Gridpunkte
Das Residual in Ganzzahl hat den Wert 5: 0 Gridpunkte

Residuals in Ganzzahl mit einem Wert grösser als 0:
Das Residual in Ganzzahl hat 1 Stelle: 634 Gridpunkte
Das Residual in Ganzzahl hat 2 Stellen: 0 Gridpunkte
Das Residual in Ganzzahl hat 3 Stellen: 0 Gridpunkte
Das Residual in Ganzzahl hat 4 Stellen: 0 Gridpunkte
Das Residual in Ganzzahl hat 5 Stellen: 0 Gridpunkte
Das Residual in Ganzzahl hat 6 Stellen: 0 Gridpunkte
Das Residual in Ganzzahl hat 7 Stellen: 0 Gridpunkte

Wenn man das Programm mit icc und dem Optimierungsflag -O3 kompiliert und ausführt, erhöht sich die standardisierte Residual Norm von 0,00000003 auf 0,00000015. Bei allen Ergebnissen sind mindestens die ersten drei Stellen korrekt und bei 3537 von 10000 Gridpunkten sind alle 7 Stellen richtig (siehe Terminalausgabe 10). Wenn man die Residuals in Ganzzahl betrachtet, sieht man, dass die nicht korrekten Ergebniswerte aus mathematischer Sicht nur in der letzten Ziffer einen Fehler aufweisen und die Residuals in Ganzzahl keinen höheren Wert als 5 haben (siehe Terminalausgabe 11). Ein Programmfehler liegt somit nicht vor.

Terminalausgabe 10: Anzahl der identen Stellen bei einer standardisierten Residual Norm von 0,00000015 (Vergleich DCS-CPU-Variante 1 mit der DCS-CPU-Variante 2, N=100, M=100, Anzahl von Atomen=1000, Float-Datentyp, icc -O3)

Die ersten 0 Stellen sind gleich: 0 Gridpunkte
Die ersten 1 Stellen sind gleich: 0 Gridpunkte
Die ersten 2 Stellen sind gleich: 0 Gridpunkte
Die ersten 3 Stellen sind gleich: 13 Gridpunkte
Die ersten 4 Stellen sind gleich: 94 Gridpunkte
Die ersten 5 Stellen sind gleich: 837 Gridpunkte
Die ersten 6 Stellen sind gleich: 5519 Gridpunkte
Die ersten 7 Stellen sind gleich: 3537 Gridpunkte

Terminalausgabe 11: Das Residual in Ganzzahl auf Basis der Genauigkeit bei einer standardisierten Residual Norm von 0,00000015 (Vergleich DCS-CPU-Variante 1 mit der DCS-CPU-Variante 2, N=100, M=100, Anzahl von Atomen=1000, Float-Datentyp, icc -O3)

Das Residual in Ganzzahl hat den Wert 0: 3537 Gridpunkte
 Das Residual in Ganzzahl hat den Wert 1: 4276 Gridpunkte
 Das Residual in Ganzzahl hat den Wert 2: 1770 Gridpunkte
 Das Residual in Ganzzahl hat den Wert 3: 383 Gridpunkte
 Das Residual in Ganzzahl hat den Wert 4: 33 Gridpunkte
 Das Residual in Ganzzahl hat den Wert 5: 1 Gridpunkte

Residuals in Ganzzahl mit einem Wert groesser als 0:
 Das Residual in Ganzzahl hat 1 Stelle: 6463 Gridpunkte
 Das Residual in Ganzzahl hat 2 Stellen: 0 Gridpunkte
 Das Residual in Ganzzahl hat 3 Stellen: 0 Gridpunkte
 Das Residual in Ganzzahl hat 4 Stellen: 0 Gridpunkte
 Das Residual in Ganzzahl hat 5 Stellen: 0 Gridpunkte
 Das Residual in Ganzzahl hat 6 Stellen: 0 Gridpunkte
 Das Residual in Ganzzahl hat 7 Stellen: 0 Gridpunkte

Code 35: Berechnung der standardisierten euklidischen Residual Norm

```
//In main()-Methode
double standardizedResidualNorm;
standardizedResidualNorm=matrizenVergleichen(energygrid_cpu,energygrid_gpu_h,grid,
gridGPU);

//Berechne Euklidische Norm (p=2)
double matrizenVergleichen(const float *A,const float *B,const dim3 gridA,
const dim3 gridB)
{
double residualNorm=0,diff,norm=0;
int x,y,z;

//int gleicheStellen;
//int Genauigkeit=7; //Float-Datentyp: 7 Stellen, Double Datentyp: 16 Stellen
//int gleicheStellenArray[Genauigkeit+1];
//int i;
//for(i=0;i<=Genauigkeit;i++)
//gleicheStellenArray[i]=0;

//int residual_als_Ganzzahl_Array[6];
//for(i=0;i<=5;i++)
//residual_als_Ganzzahl_Array[i]=0;

//int residual_als_Ganzzahl_anzahlDerStellen[Genauigkeit];
//for(i=0;i<Genauigkeit;i++)
//residual_als_Ganzzahl_anzahlDerStellen[i]=0;

for(z=0;z<gridA.z;z++){
for(y=0;y<gridA.y;y++){
for(x=0;x<gridA.x;x++){
diff=A[z*gridA.y*gridA.x+y*gridA.x+x]-B[z*gridB.y*gridB.x+y*gridB.x+x];
residualNorm+=diff*diff;
norm+=A[z*gridA.y*gridA.x+y*gridA.x+x]*A[z*gridA.y*gridA.x+y*gridA.x+x];

//gleicheStellen=gleiche_Stellen_zweier_Zahlen_berechnen(Genauigkeit,
//A[z*gridy*gridx+y*gridx+x],B[z*gridy*gridx+y*gridx+x]);
//gleicheStellenArray[gleicheStellen]++;
```

Fortsetzung von Code 35 auf der nächsten Seite

Fortsetzung von Code 35:

```
//long residual_als_Ganzzahl=
//residual_zweier_Zahlen_in_Ganzzahl_ermitteln(Genauigkeit,
//A[z*gridy*gridx+y*gridx+x],B[z*gridy*gridx+y*gridx+x]);
//if(residual_als_Ganzzahl<=5)
//residual_als_Ganzzahl_Array[residual_als_Ganzzahl]++;

//for(i=0;i<Genauigkeit;i++){
//if(residual_als_Ganzzahl>=pow(10,i) && residual_als_Ganzzahl<pow(10,i+1))
//residual_als_Ganzzahl_anzahlderStellen[i]++;
//}
}
}

//printf("\n");
//for(i=0;i<=Genauigkeit;i++)
//printf("Die ersten %d Stellen sind gleich: %d Gridpunkte\n",i,
//gleicheStellenArray[i]);
//printf("\n");

//for(i=0;i<=5;i++)
//printf("Das Residual in Ganzzahl hat den Wert %d: %d Gridpunkte\n",
//i,residual_als_Ganzzahl_Array[i]);
//printf("\n");

//printf("Residuals in Ganzzahl mit einem Wert groesser als 0:\n");
//for(i=0;i<Genauigkeit;i++){
//if(i==0)
//printf("Das Residual in Ganzzahl hat %d Stelle: %d Gridpunkte\n", (i+1),
//residual_als_Ganzzahl_anzahlderStellen[i]);
//else
//printf("Das Residual in Ganzzahl hat %d Stellen: %d Gridpunkte\n", (i+1),
//residual_als_Ganzzahl_anzahlderStellen[i]);
//}
//printf("\n");

return sqrt(residualNorm)/sqrt(norm);
}
```

Code 36: Ermittlung der Anzahl der nacheinander folgenden gleichen Stellen von zwei Zahlen zur Genauigkeitsbestimmung

```
int gleiche_Stellen_zweier_Zahlen_berechnen(int Genauigkeit, double Zahl1,
double Zahl2)
{
    int i;
    int gleicheStellen=0;

    while(Zahl1>=10){
        Zahl1/=10;
        Zahl2/=10;
    }
    while(Zahl1<1){
        Zahl1*=10;
        Zahl2*=10;
    }
    if((int)Zahl1==(int)Zahl2)
        gleicheStellen++;

    for(i=0;i<Genauigkeit-1;i++){
        Zahl1*=10;
        Zahl2*=10;
        if((int)Zahl1==(int)Zahl2)
            gleicheStellen++;
    }
    return gleicheStellen;
}
```

Code 37: Ermittlung der Abweichung zweier Zahlen in Ganzzahl auf Basis der Genauigkeit

```
long residual_zweier_Zahlen_in_Ganzzahl_ermitteln(int Genauigkeit, double
    Zahl1, double Zahl2)
{
    while(Zahl1 >= pow(10, Genauigkeit)) {
        Zahl1 /= 10;
        Zahl2 /= 10;
    }
    while(Zahl1 < pow(10, Genauigkeit-1)) {
        Zahl1 *= 10;
        Zahl2 *= 10;
    }

    return abs((long) Zahl1 - (long) Zahl2);
}
```

7 Algorithmus NW/SW (=Needleman-Wunsch/Smith-Waterman)

7.1 Einleitung

Das Anwendungsgebiet des Needleman-Wunsch/Smith-Waterman-Algorithmus ist Life Sciences/Bioinformatik [11] [59]. Der Algorithmus dient zur Suche nach Ähnlichkeiten zweier Nukleotidsequenzen oder Proteinsequenzen [60]. Man kann zwei Varianten unterscheiden. Bei der ersten Variante geht es um den Vergleich zweier großen Sequenzen [12] oder einer kleinen Sequenz mit einer sehr großen Sequenz [61]. In der zweiten Variante geht es um den Vergleich einer kleinen Sequenz mit einer Datenbank von kleinen Sequenzen [59]. In der Masterarbeit soll die erste Variante optimiert werden, in der es um den Vergleich zweier großen Sequenzen geht. Die Variante, wo eine kleine Sequenz mit einer Datenbank von kleinen Sequenzen verglichen wird, ist trivial parallelisierbar (viele unabhängige Pair Alignments) und nicht Teil der Masterarbeit. Diese Variante wird im Paper „Y. Liu, B. Schmidt, D. L Maskell, “CUDASW++2.0: enhanced Smith-Waterman Protein Database Search on CUDA-Enabled GPUs Based on SIMT and Virtualized SIMD Abstractions”, BioMed Central Ltd., April 2010.“ erklärt. Auf den Code kann man über <http://cudasw.sourceforge.net/> zugreifen. Aber auch über die Nvidia-Homepage unter „CUDA-Präsentation“ unter demselben Titel wie das Paper kann man auf den Code und das Paper zugreifen. Ein weiteres Paper zu der Variante, die nicht in der Masterarbeit behandelt wird, ist „S.A. Manavski, G. Valle, “CUDA compatible GPU cards as efficient hardware accelerator for Smith-Waterman sequence alignment”, BMC Bioinformatics, March 2008.“. Hier kann der Code über <http://bioinformatics.cribi.unipd.it/cuda/downloads.html> bezogen werden.

Das Ziel von NW/SW ist es, das optimale Alignment von zwei Sequenzen mit dynamischer Programmierung zu finden [12]. Dynamische Programmierung bedeutet, dass das optimale Ergebnis durch das Wiederverwenden der Ergebnisse der Subprobleme ermittelt wird [12].

Das Ergebnis des Algorithmus ist lokales Alignment im Falle von Smith-Waterman oder globales Alignment im Falle von Needleman-Wunsch [62]. Bei globalen Alignment müssen im Gegensatz zu lokalen Alignment die zu vergleichenden Sequenzen ähnlich groß sein [62]. Globales Alignment wird verwendet, um ähnliche Sequenzen zu vergleichen [62]. Bei lokalen Alignment geht man von unterschiedlichen Sequenzen aus und man möchte durch den Vergleich ähnliche Subsequenzen herausfiltern [62]. Da der Programmunterschied zwischen Needleman-Wunsch und Smith-Waterman aus einer if-Bedingung auf dem Ergebniswert besteht (siehe Code 41 im Kapitel 7.2.1.2) [62], wirken sich Optimierungen für den Needleman-Wunsch-Algorithmus gleich positiv wie für den Smith-Waterman-Algorithmus aus. Es können somit immer beide Algorithmen betrachtet werden.

In der Abbildung 1 sind zwei mögliche Alignments der zwei Sequenzen AAGGCCUCGCUUAG und AAUGCCAUUGCCGG angeführt. Alignment b ist besser als Alignment a, da der Ergebnis-Scoring-Wert höher ist. Für das Alignment in Abbildung 1 wird eine einfache Substitutionsmatrix verwendet (siehe

rechts in Abbildung 1). Wenn beim Alignment zwei Buchstaben der beiden Sequenzen untereinander stehen, wird der Wert aus der Substitutionsmatrix entnommen. Bei zwei gleichen Buchstaben ergibt das einen Wert von +5 und bei zwei unterschiedlichen Buchstaben ergibt das einen Wert von -3. Als Gap-Penalty-Methode wird in Abbildung 1 die gapstart/gapextend-Methode verwendet. Bei einem Gap (siehe Bindestrich in Abbildung 1) wird der Wert -8 für die Berechnung des Ergebnis-Scoring-Wertes verwendet. Bei zwei hintereinander folgenden Gaps wird für das erste Gap der Wert -8 (=gapstart) und für das zweite Gap der Wert -1 (=gapextend) benützt. Alle Werte zusammenaddiert ergeben den Ergebnis-Scoring-Wert. Je höher der Ergebnis-Scoring-Wert ist, desto besser ist das Alignment. Mit dem NW/SW-Algorithmus wird das Alignment mit dem höchsten Ergebnis-Scoring-Wert ermittelt. Damit nicht der Ergebnis-Scoring-Wert jedes möglichen Alignments einzeln berechnet werden muss, wird beim NW/SW-Algorithmus das optimale Alignment auf Basis der Ergebnisse der Subprobleme/Subalignments gefunden (siehe Abbildung 3).

Alignment a:

A A G G C C - - U - C G C U U A G Sequenz 1

A A U G C C A U U G C C G - - - G Sequenz 2

+5+5 -3 +5+5+5 -8 -1+5 -8 +5-3 -3 -8-1 -1 +5 = +4 Ergebnis-Scoring-Wert

Substitutionsmatrix:

	A	C	G	U
A	+5	-3	-3	-3
C	-3	+5	-3	-3
G	-3	-3	+5	-3
U	-3	-3	-3	+5

Alignment b:

A A G G C C - U C G C U U A G Sequenz 1

A A U G C C A U U G C C G - G Sequenz 2

+5+5 -3 +5+5+5 -8+5 -3 +5+5-3 -3 -8+5 = +17 Ergebnis-Scoring-Wert

Abbildung 1: Vergleich von zwei Alignments (Beide Alignments von [61], Informationen für die Erstellung der Alignmentbeschriftung und der Substitutionsmatrix von [61])

Als Input hat man Sequenz 1, Sequenz 2, Substitutionsmatrix und gapcosts. Daraus wird eine Startmatrix erzeugt (siehe Code 39 im Kapitel 7.2.1.1). Als Output bekommt man eine Ergebnismatrix. Mit einem traceback auf die Ergebnismatrix erhält man das optimale Alignment mit den Gaps (-).

Bei einer komplett berechneten Ergebnismatrix steht der Ergebnis-Scoring-Wert des optimalen Alignment der Sequenz 1 und der Sequenz 2 ganz rechts unten in der Matrix. Die anderen Werte in der Ergebnismatrix beinhalten den Ergebnis-Scoring-Wert der optimalen Subalignments. Der Zellenwert (i,j) enthält den Ergebnis-Scoring-Wert des optimalen Subalignments zwischen der ersten Sequenz von 0 bis (i-1) und der zweiten Sequenz von 0 bis (j-1). Zum Beispiel der rot eingekreiste Ergebniszellenwert in Abbildung 2 gibt den Ergebnis-Scoring-Wert des optimalen Subalignments zwischen ACG (=Teil von Sequenz 1) und AG (=Teil von Sequenz 2) an. Sobald die Ergebnismatrix komplett berechnet wurde, kann ausgehend vom Wert ganz rechts unten in der Matrix eine Zurückverfolgung (=traceback) des Zellenergebnisses durchgeführt werden. Dabei wird von Zelle zu Zelle ermittelt, welche umgebende Zelle zu dem Ergebniswert beigetragen hat. Sobald die Zelle ganz links oben in der Matrix erreicht wird, ist der Weg durch die Ergebnismatrix bestimmt und das optimale Alignment kann ausgegeben werden. Im Kapitel 7.2.1.3 wird die Zurückverfolgung inklusive Code und Abbildung genauer erklärt.

Sequenz 2 →

	-	A	G	T	C
-	0	-6	-12	-18	-24
A	-6	1	-5	-11	-17
C	-12	-5	-1	-7	-10
G	-18	-11	-4	-3	-9
T	-24	-17	-10	-3	-5
C	-30	-23	-16	-9	-2

Sequenz 1 ↓

Abbildung 2: Bedeutung eines berechneten Zellenwerts in der Ergebnismatrix

Der Needleman-Wunsch- bzw. Smith-Waterman-Algorithmus ist nicht so einfach für GPUs zu parallelisieren. Es kann nur Wavefront für Wavefront berechnet werden, da eine Wavefront jeweils von den davor berechneten Wavefronts abhängig ist. Innerhalb der Wavefront kann parallel gerechnet werden. Nur in derselben Antidiagonale gibt es keine Datenabhängigkeit (siehe Abbildung 3). Beim Vergleich einer Sequenz 1 mit Länge N-1 mit einer Sequenz 2 mit Länge M-1 ist die Startmatrix inklusive Randbereich N*M groß und man hat $((N-1)+(M-1)-1)$ Wavefronts. Der Code zur Berechnung der Ergebnismatrix wird im Kapitel 7.2.1.2 erklärt.

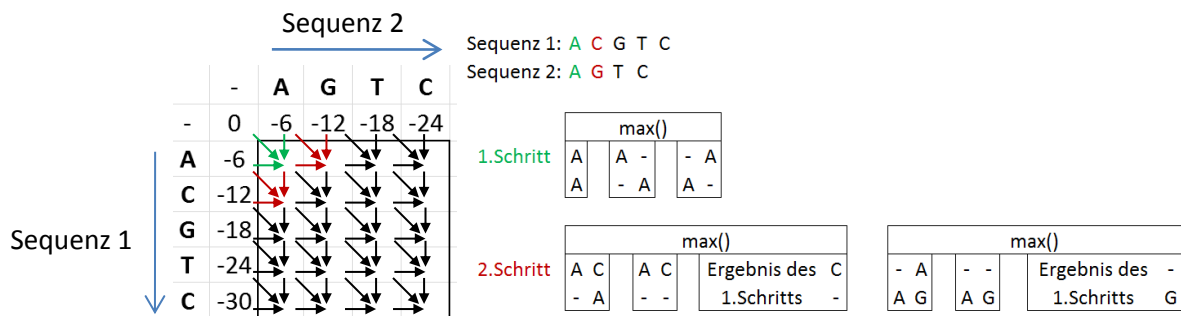


Abbildung 3: Abhängigkeiten bei der Berechnung der Ergebnismatrix

Als Substitutionsmatrix wird in der Masterarbeit die blosum62-Substitutionsmatrix verwendet (siehe Code 38). Die blosum62-Substitutionsmatrix ist eine symmetrische Matrix und die Buchstabenreihenfolge ist „ARNDCQEGHILKMFPSTWYV“ [63]. Jeder Buchstabe steht für eine Aminosäure [60]. Es gibt 20 Aminosäuren [60]. Deswegen ist die Substitutionsmatrix 20*20 groß [64]. Die Matrix kann auf 24*24 erweitert werden und hat dann zusätzlich noch B, Z, X und * [63]. X steht für alle Aminosäuren [60]. B steht für die Aminosäuren D und N [60]. Z steht für die Aminosäuren E und Q [60]. Aus der Substitutionsmatrix wird beim NW-Algorithmus der Scoring-Wert abgelesen, wenn sich zwei Aminosäuren im Alignment gegenüberstehen. Dabei wird der Typ der Aminosäuren verwendet. Wenn eine Aminosäure der Sequenz 1 den Typ x und eine Aminosäure der Sequenz 2 den Typ y hat, befindet sich der Scoring-Wert in der (x,y)-Zelle der Substitutionsmatrix. Je höher der Wert in der Substitutionsmatrix ist, desto höher ist die Wahrscheinlichkeit, dass sich die beiden Aminosäuretypen im optimalen Alignment gegenüberstehen. Wenn die Werte in der Diagonale ausgelesen werden, bedeutet das, dass sich zwei gleiche Aminosäuren im Alignment gegenüberstehen. Da sich gleiche Aminosäuren zwischen den zwei zu vergleichenden Sequenzen im Alignment gegenüber stehen sollen, sind die Werte in der Diagonale höher als die Werte außerhalb der Diagonale. Einfache Substitutionsmatrizen haben nur zwei unterschiedliche Scoring-Werte (siehe Abbildung 1). Die in Abbildung 1 dargestellte einfache Substitutionsmatrix ist nur 4x4 groß, da statt der Aminosäuren die Nukleotide der beiden Sequenzen verglichen werden und es 4 Nukleotide gibt [60]. Jeweils drei Nukleotide, die nacheinander in der Sequenz folgen, stehen für eine Aminosäure [60]. In einer einfachen Substitutionsmatrix gibt es einen Scoring-Wert für gleiche Aminosäuren/Nukleotide und einen weiteren Scoring-Wert für unterschiedliche Aminosäuren/Nukleotide im Alignment. Aufwendigere Substitutionsmatrizen, wie die blosum62-

Substitutionsmatrix, verwenden mehr als zwei Scoring-Werte. Bei der 20x20 blosum62-Substitutionsmatrix (siehe Code 38) gibt es in der Diagonale (=gleiche Aminosäuren) Werte zwischen 4 und 11 und außerhalb der Diagonale (=unterschiedliche Aminosäuren) Werte zwischen -4 und 3. In der blosum62-Substitutionsmatrix ist der Wert zwischen der Aminosäure D und der Aminosäure E mit 2 hoch [60]. Der Grund ist, dass eine Mutation zwischen D und E sehr leicht passieren kann [60]. Zwischen G und W in der blosum62-Substitutionsmatrix ist der Wert mit -2 niedrig, da eine Mutation selten passiert [60]. Die Substitutionsmatrix legt durch die unterschiedlichen Werte außerhalb der Diagonale fest, wie hoch die Wahrscheinlichkeit ist, dass die Aminosäure x durch die Aminosäure y ersetzt wurde [60] und es aus diesem Grund einen Unterschied beim Aminosäuretyp beim Vergleich von zwei Sequenzen gibt. Wie bereits erwähnt, je höher der Wert in der Substitutionsmatrix ist, desto wahrscheinlicher ist es, dass im optimalen Alignment zwei unterschiedliche Aminosäuretypen gegenüber stehen und als fast gleiche Aminosäuren behandelt werden, da eine Mutation der Aminosäure in die andere Aminosäure bei einer der zwei zu vergleichenden Sequenzen passiert sein kann. Die unterschiedlichen Werte in der Diagonale der Substitutionsmatrix stehen für die Wahrscheinlichkeit, dass es bei dem Aminosäuretyp zu keiner Mutation gekommen ist und die Aminosäure somit gleich geblieben ist [60]. Je höher der Wert in der Substitutionsmatrix, desto höher die Wahrscheinlichkeit, dass die zwei zu vergleichenden Aminosäuren nicht durch eine Mutation gleich geworden sind und dass die beiden Aminosäuren beim optimalen Alignment gegenübergestellt werden.

Code 38: Substitutionsmatrix (Daten von [63])

```
Datentyp h_Substitutionsmatrix[20][20] = {
    { 4, -1, -2, -2, 0, -1, -1, 0, -2, -1, -1, -1, -2, -1, 1, 0, -3, -2, 0},
    {-1, 5, 0, -2, -3, 1, 0, -2, 0, -3, -2, 2, -1, -3, -2, -1, -1, -3, -2, -3},
    {-2, 0, 6, 1, -3, 0, 0, 0, 1, -3, -3, 0, -2, -3, -2, 1, 0, -4, -2, -3},
    {-2, -2, 1, 6, -3, 0, 2, -1, -1, -3, -4, -1, -3, -3, -1, 0, -1, -4, -3, -3},
    { 0, -3, -3, -3, 9, -3, -4, -3, -3, -1, -1, -3, -1, -2, -3, -1, -1, -2, -1, -1},
    {-1, 1, 0, 0, -3, 5, 2, -2, 0, -3, -2, 1, 0, -3, -1, 0, -1, -2, -1, -2},
    {-1, 0, 0, 2, -4, 2, 5, -2, 0, -3, -3, 1, -2, -3, -1, 0, -1, -3, -2, -2},
    { 0, -2, 0, 0, -1, -3, -2, -2, 6, -2, -4, -4, -2, -3, -3, -2, 0, -2, -2, -3},
    {-2, 0, 1, -1, -3, 0, 0, -2, 8, -3, -3, -1, -2, -1, -2, -1, -2, -2, 2, -3},
    {-1, -3, -3, -3, -1, -3, -3, -4, -3, 4, 2, -3, 1, 0, -3, -2, -1, -3, -1, 3},
    {-1, -2, -3, -4, -1, -2, -3, -4, -3, 2, 4, -2, 2, 0, -3, -2, -1, -2, -1, 1},
    {-1, 2, 0, 0, -1, -3, 1, 1, -2, -1, -3, -2, 5, -1, -3, -1, 0, -1, -3, -2},
    {-1, -1, -2, -3, -1, 0, -2, -3, -2, 1, 2, -1, 5, 0, -2, -1, -1, -1, -1, 1},
    {-2, -3, -3, -3, -2, -3, -3, -3, -1, 0, 0, -3, 0, 6, -4, -2, -2, 1, 3, -1},
    {-1, -2, -2, -1, -3, -1, -1, -2, -2, -3, -3, -1, -2, -4, 7, -1, -1, -4, -3, -2},
    { 1, -1, 1, 0, -1, 0, 0, 0, -1, -2, -2, 0, -1, -2, -1, 4, 1, -3, -2, -2},
    { 0, -1, 0, 0, -1, -1, -1, -2, -1, -1, -1, -1, -2, -1, 1, 5, -2, -2, 0},
    {-3, -3, -4, -4, -2, -2, -3, -2, -2, -3, -2, -3, -1, 1, -4, -3, -2, 11, 2, -3},
    {-2, -2, -2, -3, -2, -1, -2, -3, 2, -1, -1, -2, -1, 3, -3, -2, -2, 2, 7, -1},
    { 0, -3, -3, -3, -1, -2, -2, -3, -3, 3, 1, -2, 1, -1, -2, -2, 0, -3, -1, 4},
};

char Substitutionsmatrix_Buchstabenreihenfolge[20]=
    {'A','R','N','D','C','Q','E','G','H','I','L','K','M','F','P','S','T','W','Y','V'};
```

Die verwendete Gap-Penalty-Methode legt fest, auf welche zuvor berechneten Ergebniswerte in der aktuellen Berechnung zugegriffen wird. Es gibt zwei Gap-Penalty-Methoden, die gapcosts- und die gapstart/gapextend-Methode. Bei der gapcosts-Methode werden für die Berechnung des Ergebniswertes (siehe rot eingekreiste Zelle in Abbildung 4) nur die unmittelbaren Nachbarwerte als Input verwendet (siehe Werte im violetten Ring in Abbildung 4) [13]. Diese Methode wird in den Codes in der Masterarbeit verwendet. Der Wert der Variable gapcosts ist der Scoring-Wert, wenn eine Aminosäure einem Gap im Alignment gegenübersteht. In der Masterarbeit wird der Wert -4 für gapcosts verwendet. Da der niedrigste Wert von der blosum62-Substitutionsmatrix bei unterschiedlichen Aminosäuren -4 ist (siehe vorigen Absatz), ist die Wahrscheinlichkeit, dass eine Aminosäure einem Gap im optimalen Alignment gegenübersteht, genauso so niedrig, wie wenn zwei unterschiedliche Aminosäuren, die sehr selten in die jeweils andere Aminosäure mutieren, im optimalen Alignment gegenüberstehen. Je kleiner der Wert von gapcosts gegenüber den Werten in der Substitutionsmatrix ist, desto geringer ist die Wahrscheinlichkeit, dass eine Aminosäure einem

Gap im Alignment gegenübersteht. In Abbildung 1 kann ein Alignment mit Gaps betrachtet werden. Ein Gap wird mit einem Bindestrich dargestellt. Neben der gapcosts-Methode gibt es die aufwendigere gapstart/gapextend-Methode, die wesentlich mehr Datenzugriffe hat. Hier müssen zusätzlich alle zuvor berechneten Ergebniswerte der Zeile und der Spalte verarbeitet werden (siehe Werte in der blauen eckigen Umrandung in Abbildung 4) [61]. Im Paper [61] kann die exakte Formel eingesehen werden. Beim Alignment in Abbildung 1, das aus dem Paper [61] stammt, wird die gapstart/gapextend-Methode verwendet, indem ein einzelnes Gap den Scoring-Wert -8 hat, jedoch bei mehreren Gaps hintereinander nur für den ersten Gap der Scoring-Wert -8 (=gapstart) beträgt und alle weiteren Gaps einen Scoring Wert von -1 (=gapextend) erhalten. Das führt dazu, dass sobald ein Gap im optimalen Alignment vorhanden ist, oft mehrere Gaps anschließen, da der Scoring-Wert von gapextend mit -1 so hoch ist, dass er die Scoring-Werte aus der Substitutionsmatrix (siehe Matrix in Abbildung 1) außerhalb der Diagonale (=unterschiedliche Aminosäuren), die einen Wert von -3 haben, übertrifft. Im Gegensatz dazu ist der gapstart-Wert mit -8 niedriger als der kleinste Wert in der Substitutionsmatrix mit -3 und damit wird die Wahrscheinlichkeit für nur ein einzelnes Gap im Alignment reduziert.

Sequenz 2
→

	-	A	G	T	C
-	0	-6	-12	-18	-24
A	-6	1	-5	-11	-17
C	-12	-5	-1	-7	-10
G	-18	-11	-4	-3	-9
T	-24	-17	10	-3	-5
C	-30	-23	-16	-9	-2

Sequenz 1
↓

Abbildung 4: Unterschiedliche Gap-Penalty-Methoden (Informationen von [13] [61])

Es gibt drei unterschiedliche Parallelisierungsmöglichkeiten für den Algorithmus. Die erste Möglichkeit ist pro Antidiagonale zur parallelisieren (siehe Abbildung 5). Diese Variante wurde selbst implementiert. Hier wird nur Global Memory verwendet und ein Speedup von 1,89 erreicht (N=8000, M=8000, GTX 480, siehe Abbildung 26 im Kapitel 7.3.2.3).

Sequenz 2
→

	-	A	G	T	C
-	0	-6	-12	-18	-24
A	-6	1	-2	-2	2
C	-12	-2	-2	-2	1
G	-18	-2	1	-2	2
T	-24	-2	-2	1	2
C	-30	-2	-2	-2	1

Sequenz 1
↓

Abbildung 5: Parallelisierung pro Antidiagonale

Sequenz 2 →

	-	A	G	T	C
-	0	-6	-12	-18	-24
A	-6				
C	-12				
G	-18				
T	-24				
C	-30				

Sequenz 1 ↓

Abbildung 6: Parallelisierung pro Antidiagonale eines Blocks und pro Antidiagonale-Blöcke (Informationen von [12] [13])

Die zweite Parallelisierungsmöglichkeit ist pro Antidiagonale eines Blocks und pro Antidiagonale-Blöcke zu parallelisieren (siehe Abbildung 6). Für diese Variante dient als Basis ein Code von der Universität von Virginia. Auf der Nvidia-Seite [11] gibt es eine Verlinkung zum Paper [12] und zum Code [13] des Programms von der Universität von Virginia. In der Masterarbeit wurden Optimierungen (z. B. Wahl der optimalen Blockgröße) für mehr Performance durchgeführt. Diese Parallelisierungsmethode verwendet Pipelining und nützt den Shared Memory der GPU. Man erreicht hier einen Speedup von 3,86 ($N=8000$, $M=8000$, GTX 480, siehe Abbildung 35 im Kapitel 7.3.3.4). Im Kapitel 7.3.1 wird die Parallelität des Shared-Memory-Programms genauer erklärt.

Bei der dritten Parallelisierungsmöglichkeit wird pro Zeile parallelisiert (siehe Abbildung 7). Diese Parallelisierungsmethode wurde in der Masterarbeit nicht berücksichtigt. Sie wurde von der Universität von Massachusetts Amherst entwickelt und vernachlässigt zunächst die linke Abhängigkeit [61]. Damit ist ein Speedup bis 45 ($N=1024$, $M=10^7$, GTX 295) möglich [61]. Die Problemgröße muss groß sein. Bei $N=1024$, $M=10^5$ und GTX 295 wird ein Speedup von 15 erreicht [61]. Bei $N=1024$, $M=10^4$ und GTX 295 liegt der Speedup bereits bei 2 [61].

Sequenz 2 →

	-	A	G	T	C
-	0	-6	-12	-18	-24
A	-6	1	-2	-2	-2
C	-12	-2	-2	-2	1
G	-18	-2	1	-2	-2
T	-24	-2	-2	1	-2
C	-30	-2	-2	-2	1

Sequenz 1 ↓

Abbildung 7: Parallelisierung pro Zeile (Informationen von [61])

Bei dem in der Masterarbeit verwendeten Code wurde Folgendes selbst entwickelt. Beim CPU-Code wurde für das parallele Ausführen auf allen CPU-Kernen ein OpenMP-Programm entwickelt. Beim GPU-Global-Memory-Programm (Parallelisierungsvariante: pro Antidiagonale) wurde der komplette CUDA- und OpenCL-Code selbst geschrieben. Beim Shared-Memory-Programm (Parallelisierungsvariante: pro Antidiagonale eines Blocks und pro Antidiagonale-Blöcke) wurden alle Optimierungen selbst entwickelt und es wurde ein OpenCL-Programm aus dem CUDA-Programm erzeugt. Außerdem wurde eine Überprüfung der Korrektheit hinzugefügt, die Sequenzgröße muss nicht mehr ein Vielfaches der Blockgröße sein und es wurde eine Kernel- und Gesamt-Laufzeitmessung hinzugefügt.

7.2 CPU-Programm mit Laufzeitmessungen

Bei den Laufzeitmessungen in den Unterkapiteln wurde beim Programmaufruf für das Argument gapcosts immer 4 gewählt. Im Programmcode wird das Argument gapcosts in der Variable penalty gespeichert und gapcosts beträgt -4, da die Variable penalty im Programmcode mit -1 multipliziert wird (siehe „i*penalty“ und „j*penalty“ im Code 39 im Unterkapitel 7.2.1.1 und „penalty*-1“ beim Funktionsaufruf im Code 41 im Unterkapitel 7.2.1.2). Die Länge der beiden zu vergleichenden Sequenzen ist identisch und ist in der X-Achse der Diagramme angegeben. Als CPU wurde die Intel Xeon X5550 verwendet.

7.2.1 C

7.2.1.1 Schritt 1 - Initialisierung der Startmatrix

Im ersten Schritt wird die Startmatrix initialisiert (siehe Code 39). Die erste Spalte und die erste Zeile der Startmatrix werden mit den gapcosts (Variable penalty im Code 39) berechnet. Die Werte der anderen Zellen werden mit Hilfe der Substitutionsmatrix und den beiden Sequenzen ermittelt.

Code 39: Schritt 1 - Initialisierung der Startmatrix [13] (adaptiert)

```
scorematrixCPU=malloc((Sequenzgroesse1+1)*(Sequenzgroesse2+1)*
    sizeof(DatentypC_CPU));
h_Sequenz1=malloc(Sequenzgroesse1*sizeof(DatentypC_CPU));
h_Sequenz2=malloc(Sequenzgroesse2*sizeof(DatentypC_CPU));

//Startsequenzen erzeugen
srand(7);
for(i=0;i<Sequenzgroesse1;i++)
    h_Sequenz1[i]=rand()%20;
for(i=0;i<Sequenzgroesse2;i++)
    h_Sequenz2[i]=rand()%20;

scorematrixCPU[0]=0; // Wert ganz links oben

for(i=1;i<Sequenzgroesse1+1;i++) // Werte der ersten Spalte
    scorematrixCPU[i*(Sequenzgroesse2+1)]=-i*penalty;

for(j=1;j<Sequenzgroesse2+1;j++) // Werte der ersten Zeile
    scorematrixCPU[j]=-j*penalty;

for(i=1;i<Sequenzgroesse1+1;i++){ // Werte der anderen Zellen
    for(j=1;j<Sequenzgroesse2+1;j++)
        scorematrixCPU[i*(Sequenzgroesse2+1)+j]=
            h_Substitutionsmatrix_Aminosaeuren[h_Sequenz1[i-1]][h_Sequenz2[j-1]];
}
```

7.2.1.2 Schritt 2 - Berechnung der Ergebnismatrix

Im zweiten Schritt wird die Ergebnismatrix berechnet (siehe Code 41). Im Code 40 wird der Code 41 kompakt zusammengefasst (ohne Funktionsdeklaration und Maximum-Implementierung), um den Code schnell erfassen zu können und eine Gegenüberstellung mit der Ergebnismatrix zu erleichtern. Beide Schleifen (i und j) beginnen mit der Zahl 1 und es wird immer die Zelle (i,j) beschrieben. Somit werden die Zellen der ersten Spalte und der ersten Zeile der Ergebnismatrix nicht beschrieben, sondern nur ausgelesen. Bei allen anderen Zellen werden drei Additionen durchgeführt, das Maximum der drei Ergebniswerte ermittelt und dieser Wert in die Ergebniszeile geschrieben. In der ersten Addition wird der Inputwert der Ergebniszeile (i,j) mit dem Wert der in der Diagonale darüber liegenden Zelle (i-1,j-1) addiert (siehe farbliche Markierung in Abbildung 8 und Code 40). In der zweiten Addition wird der Wert links von der Ergebniszeile (i,j-1) mit der Variable gapcosts addiert. In der dritten Addition wird der Wert über der Ergebniszeile (i-1,j) mit der Variable gapcosts addiert.

Code 40: Kompakte Zusammenfassung der Berechnung der Ergebnismatrix [13] (adaptiert)

```
for(i=1;i<N;i++) //Länge der Sequenz 1
{
    for(j=1;j<M;j++) //Länge der Sequenz 2
    {
        //Diagonal
        m1=matrix[(i-1)*M+(j-1)]+matrix[i*M+j];
        //Links
        m2=matrix[i*M+(j-1)]+gapcosts;
        //Oben
        m3=matrix[(i-1)*M+j]+gapcosts;

        //Bei Needleman Wunsch
        matrix[i*M+j]=max(m1,m2,m3);

        //Bei Smith Waterman
        matrix[i*M+j]=max(m1,m2,m3,0);
    }
}
```

Sequenz 2 →

	-	A	G	T	C
-	0	-6	-12	-18	-24
A	-6	1	-2	-2	-2
C	-12	-2	-2	-2	1
G	-18	-2	1	-2	-2
T	-24	-2	-2	1	-2
C	-30	-2	-2	-2	1

↓ Sequenz 1

Abbildung 8: Benötigte Zellenwerte bei der ersten Berechnung in der Ergebnismatrix (Informationen von [13])

Code 41: Schritt 2 - Berechnung der Ergebnismatrix [13] (adaptiert)

```
//In main()-Methode
needlemanWunschCPU_gapcosts(scorematrixCPU, (Sequenzgroesse1+1), (Sequenzgroesse2+1),
    penalty*-1);

//Außerhalb der main()-Methode
void needlemanWunschCPU_gapcosts(DatentypC_CPU *scorematrix, const int N,
    const int M, const int gapcosts)
{
    int i,j;
    DatentypC_CPU m1,m2,m3,tmp;

    for(i=1;i<N;i++)
    {
        for(j=1;j<M;j++)
        {
            //Diagonal
            m1=scorematrix[(i-1)*M+(j-1)]+scorematrix[i*M+j];

            //Oben
            m3=scorematrix[(i-1)*M+j]+gapcosts;

            //Links
            m2=scorematrix[i*M+(j-1)]+gapcosts;

            //Möglichkeit mit dem höchsten Wert ermitteln
            tmp=m1;
            if(m3>tmp)
                tmp=m3;
            if(m2>tmp)
                tmp=m2;
            //if(tmp<0) //Bei Smith Waterman
            //tmp=0;
            scorematrix[i*M+j]=tmp;
        }
    }
}
```

7.2.1.3 Schritt 3 - Traceback auf die Ergebnismatrix

Im dritten Schritt wird ein Traceback auf die Ergebnismatrix durchgeführt. Mit dem Java-Applet-Programm „Need.&Wunsch“ von [65] kann der Traceback leicht nachvollzogen werden (siehe Abbildung 9). Es wird ganz rechts unten in der Ergebnismatrix mit dem Zurückverfolgen/Traceback begonnen. Der Zellenwert ganz rechts unten stellt den Ergebnis-Scoring-Wert des optimalen Alignments der beiden Sequenzen dar und beträgt -2 in Abbildung 9. Das optimale Alignment kann ausgegeben werden, indem zurückverfolgt wird, wie der Ergebnis-Scoring-Wert zustande gekommen ist. Es wird ermittelt, ob der Zellenwert links (siehe Variable *w* im Code 42), der Zellenwert oben (siehe Variable *n* im Code 42) oder der Zellenwert in der Diagonale (siehe Variable *nw* im Code 42) zu dem Ergebniswert (siehe Variable *aktuelles_Element* im Code 42) geführt hat. In Abbildung 9 ist es der Zellenwert in der Diagonale. Das bedeutet für das Alignment, dass sich der letzte Buchstabe der Sequenz 1 und der letzte Buchstabe der Sequenz 2 im Alignment gegenüberstehen (siehe if-Bedingung mit *nw* im Code 42 und unten in Abbildung 9). Hätte der linke Zellenwert zu dem Ergebniswert geführt, würde im Alignment dem letzten Buchstaben von der Sequenz 2 einem gap (-) gegenüberstehen. Hätte der obere Zellenwert zu dem Ergebniswert geführt, würde im Alignment dem letzten Buchstaben von der Sequenz 1 einem gap (-) gegenüberstehen. Im nächsten Schritt wird zu der Zelle gewechselt, die zu dem Ergebniswert geführt hat. Die grün gefärbten Zellen in Abbildung 9 zeigen den zurückgefolgten Weg durch die Matrix an. Sobald die Zelle links oben in der Matrix erreicht wird, ist das optimale Alignment (siehe Ausgabe unten in Abbildung 9) ermittelt. Der komplette Code für das Traceback ist im Code 73 im Anhang A.1 dargestellt.

Code 42: Ausschnitt des Codes für Traceback [13] (adaptiert)

```
...
nw = scorematrixCPU[(i - 1) * max_cols + j - 1] +
    h_Substitutionsmatrix_Aminosaeuren[h_Sequenz1[i-1]][h_Sequenz2[j-1]];
w = scorematrixCPU[i * max_cols + j - 1]-penalty;
n = scorematrixCPU[(i - 1) * max_cols + j]-penalty;
...
if (aktuelles_Element == nw) {
    i--;
    j--;
    alignment_Sequenz1[index_alignment]=h_Sequenz1[i];
    alignment_Sequenz2[index_alignment]=h_Sequenz2[j];
    index_alignment--;
    continue;
}
else if (aktuelles_Element == w) {
    j--;
    alignment_Sequenz1[index_alignment]=-1;
    alignment_Sequenz2[index_alignment]=h_Sequenz2[j];
    index_alignment--;
    continue;
}
else if (aktuelles_Element == n) {
    i--;
    alignment_Sequenz1[index_alignment]=h_Sequenz1[i];
    alignment_Sequenz2[index_alignment]=-1;
    index_alignment--;
    continue;
}
...
```

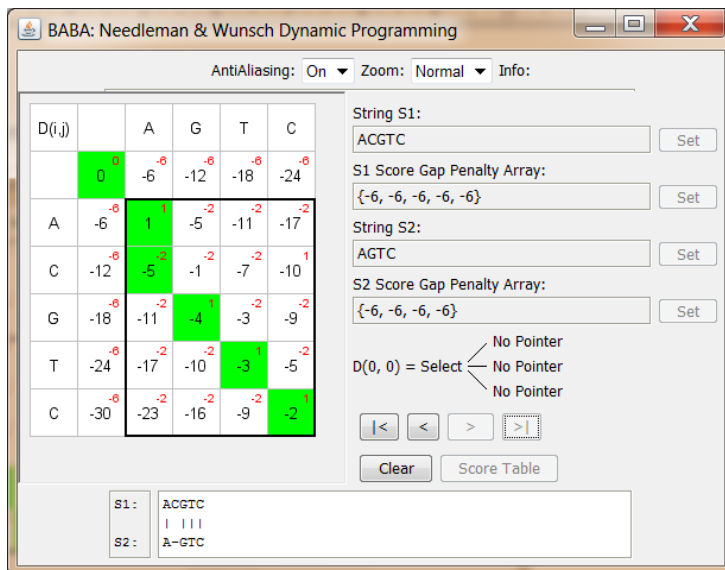


Abbildung 9: Screenshot des Java-Applet-Programms Need.&Wunsch von [65]

7.2.1.4 Unterschiedliche Compiler und Compiler-Flags

Bei der Messung der Gesamtlaufzeit des C-Programms (=Funktionslaufzeit des zweiten Schritts (Berechnung der Ergebnismatrix) + Laufzeit des Anlegens und Freigebens des Speichers für die Arrays) mit unterschiedlichen Compiler und Compiler-Flags hat sich herausgestellt, dass die richtige Wahl des Compilers und der Compiler-Flags eine große Auswirkung auf die Laufzeit hat (siehe Abbildung 10). Die niedrigste Laufzeit liefert der icc-Compiler mit dem Optimierungsflag -O3 und -xSSSE3. Die höchste Laufzeit hat das Programm, wenn es mit dem gcc-Compiler ohne Angabe von Flags kompiliert wird. Die höchste Laufzeit liegt sehr weit entfernt von der niedrigsten Laufzeit (bei $N=8000$ und $M=8000$: 1,19 versus 0,15 Sekunden). Interessant ist, dass der gcc-Compiler nicht wie beim icc-Compiler standardmäßig mit dem Flag -O2 kompiliert, sondern mit dem Flag -O0. Würde der gcc-Compiler ebenfalls standardmäßig mit dem Flag -O2 kompilieren, wäre der Abstand der niedrigsten Laufzeit zur höchsten Laufzeit deutlich geringer (bei $N=8000$ und $M=8000$: 0,29 versus 0,15 Sekunden).

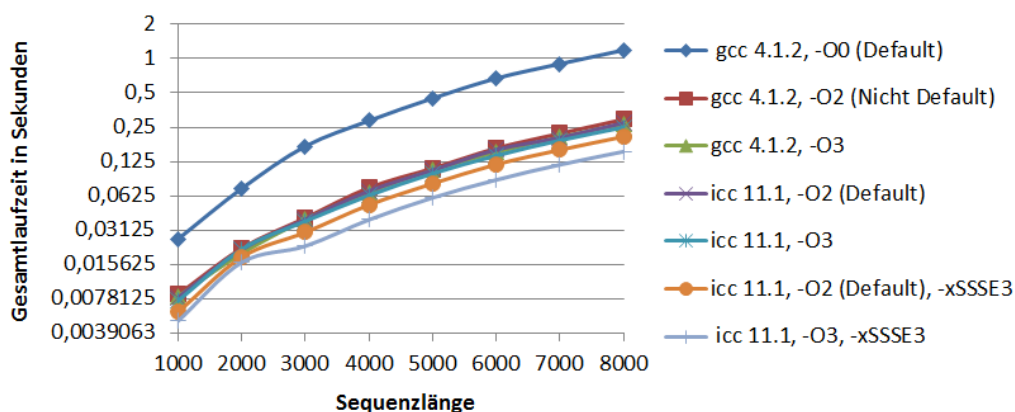


Abbildung 10: Wahl des Compilers und der Compilerflags

Mit dem Compilerflag -xSSSE3 wird die Streaming-SIMD-Extension verwendet. Da es mehrere SSE-Versionen gibt, wurde für jede SSE-Version die Laufzeit bei der Problemgröße $N=8000$ und $M=8000$ gemessen. Die Laufzeiten sind in der nachfolgenden Tabelle 35 dargestellt. Ab SSSE3 wird die Laufzeit deutlich besser. Damit ist das Erzeugen von SSSE3-Instruktionen wesentlich für den CPU Code. SSSE3 steht für „Supplemental Streaming SIMD Extensions“ [66].

Tabelle 35: Laufzeit bei unterschiedlichen SSE-Versionen (Problemgröße: N=8000, M=8000)

SSE - Flag	Gesamtlaufzeit	Funktionslaufzeit
ohne SSE	0.249442	0.242165
-xSSE	0.249488	0.242185
-xSSE2	0.249464	0.242194
-xSSE3	0.249084	0.241822
-xSSSE3	0.151692	0.144497
-xSSE4.1	0.151827	0.144549
-xSSE4.2	0.151758	0.144471

Damit das „-xSSSE3“-Flag überhaupt seine Wirkung entfalten kann, ist es wichtig die richtige Implementierung für die Ermittlung des Maximums von drei Werten zu verwenden (siehe Tabelle 36). Nur bei „2x if“ oder „2x if-else“ führt das „-xSSSE3“-Flag zu einer starken Verbesserung der Laufzeit. In der Masterarbeit wurde für den CPU-Code die „2x if“-Variante verwendet. Im Code 43 sind alle vier Varianten dargestellt.

Tabelle 36: Auswirkung der Implementierung der Ermittlung des Maximums von drei Werten auf das Flag -xSSSE3 (Problemgröße: N=8000, M=8000)

Ermittlung des Maximums von drei Werten mit:	Funktionslaufzeit bei -O3	Funktionslaufzeit bei -O3 und -xSSSE3
2x if	0,2415676	0,1441294
2x if-else	0,2415787	0,1440467
max() verschachtelt	0,2080883	0,2055322
if-else verschachtelt	0,2557453	0,2546501

Code 43: Unterschiedliche Implementierungsvarianten für die Ermittlung des Maximums von drei Werten

2x if	2x if-else [13] (adaptiert)	max() verschachtelt	if-else verschachtelt
<pre> maxWert=a; if(b>maxWert) maxWert=b; if(c>maxWert) maxWert=c; </pre>	<pre> int maximum(int a, int b,int c) { int k; if(a <= b) k = b; else k = a; if(k <= c) return(c); else return(k); } maxWert=maximum(a,b,c); </pre>	<pre> maxWert=max(a,max(b,c)); </pre>	<pre> if(a>b){ if(a>c) maxWert=a; else maxWert=c; } else{ if(b>c) maxWert=b; else maxWert=c; } </pre>

7.2.1.5 Laufzeit der drei Schritte

Wenn man die Laufzeit der drei unterschiedlichen Teile des Programms (Inputmatrix initialisieren, Funktion/Ergebnismatrix berechnen und traceback) misst, kommt man bei der Konfiguration „X5550, Standard Code, gapcosts, icc 11.1 und -O3“ zu den Werten in Abbildung 11. Die Abbildung 11 zeigt, dass die Laufzeit von traceback zu vernachlässigen ist.

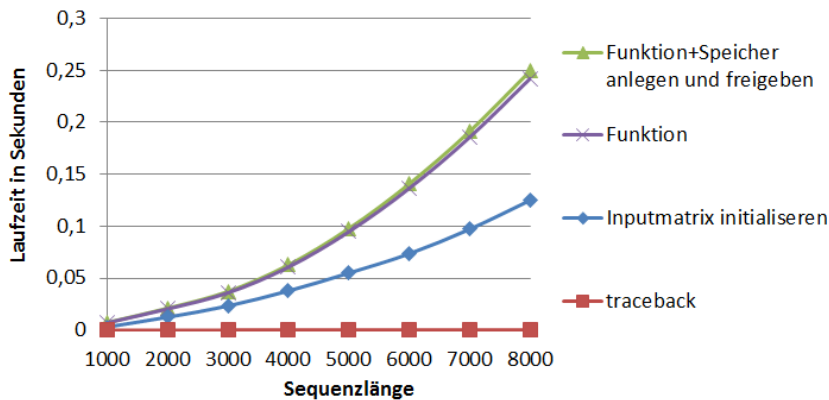


Abbildung 11: Laufzeit der einzelnen Schritte von Needleman-Wunsch

7.2.1.6 GCUPS

Oft wird beim Algorithmus Needleman-Wunsch/Smith-Waterman statt der Laufzeit die GCUPS angegeben. Ein GCUPS bedeutet, dass 10^9 Zellen der Ergebnismatrix pro Sekunde berechnet wurden. Für die Berechnung der GCUPS wurde die Kernellaufzeit benützt. In Abbildung 12 sieht man, dass ab $N=4000$ und $M=4000$ die GCUPS gleich bleiben. Die Hardware erreicht ab diesem Punkt die maximale Auslastung. Das Flag „-xSSSE3“ beim Kompilieren ermöglicht ab $N=3000$ und $M=3000$ einen deutlich höheren GCUPS-Wert (bei $N=3000$ und $M=3000$: 0,41 versus 0,25 GCUPS).

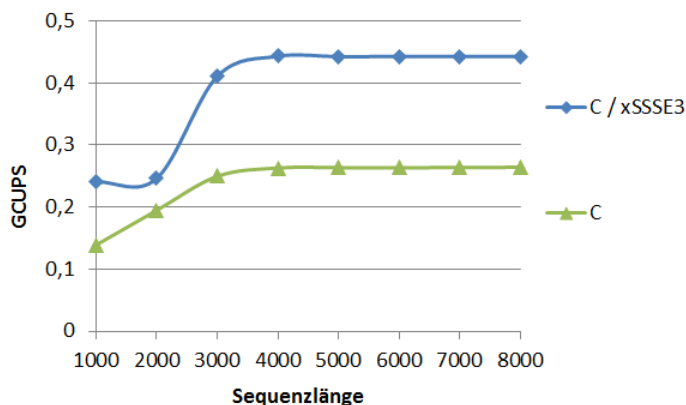


Abbildung 12: GCUPS

7.2.2 OpenMP

Beim Aufruf des Programms wird die Anzahl der Threads übergeben (Programmaufruf: `needlemanWunsch <Laenge fuer zufaellige 1.Sequenz> <Laenge fuer zufaellige 2.Sequenz> <Gap-Kosten (z. B. 4)> <Anzahl von Threads>`). Mit `"omp_set_num_threads(atoi(argv[4]));"` wird in der main-Methode die Anzahl der Threads festgelegt. Mit `"#include <omp.h>"` außerhalb der main-Methode wird der OpenMP-Header geladen. Mit dem Flag „-openmp“ kann das OpenMP-Programm kompiliert werden.

Die ersten zwei Schritte (Initialisierung der Startmatrix und Berechnen der Ergebnismatrix) des C-Programms werden mit OpenMP parallelisiert (siehe Code 44 und Code 45).

Im ersten Schritt (Initialisierung der Startmatrix) wird die Berechnung der Inputwerte für die Zellen der ersten Spalte und der ersten Zeile der Matrix jeweils separat parallelisiert. Die restlichen Zellen werden über die obere i-Schleife parallelisiert (siehe Code 44).

Code 44: Schritt 1 - Initialisierung der Startmatrix

```
#pragma omp parallel default(none) shared(h_Substitutionsmatrix,
scorematrixCPU_parallel_Rowwise,h_Sequenz1,h_Sequenz2,N,M) private(i,j)
{
#pragma omp for
for(i=1;i<N;i++){
for(j=1;j<M;j++){
scorematrixCPU_parallel_Rowwise[i*M+j]=(DatentypC)h_Substitutionsmatrix[
h_Sequenz1[i-1]][h_Sequenz2[j-1]];
}
}

scorematrixCPU_parallel_Rowwise[0]=0;

#pragma omp parallel default(none) shared(h_Substitutionsmatrix,
scorematrixCPU_parallel_Rowwise,h_Sequenz1,h_Sequenz2,N,M,penalty) private(i)
{
#pragma omp for
for(i=1;i<N;i++){
scorematrixCPU_parallel_Rowwise[i*M]=-i*penalty;
}

#pragma omp parallel default(none) shared(h_Substitutionsmatrix,
scorematrixCPU_parallel_Rowwise,h_Sequenz1,h_Sequenz2,M,penalty) private(j)
{
#pragma omp for
for(j=1;j<M;j++){
scorematrixCPU_parallel_Rowwise[j]=-j*penalty;
}
}
```

Im zweiten Schritt (Berechnen der Ergebnismatrix) wird für die Parallelisierung die Startmatrix statt Zeilenweise Antidiagonale für Antidiagonale gespeichert. Das hat den Vorteil, dass sich die benötigten Zellen für die parallele Berechnung einer Antidiagonale direkt nebeneinander befinden. Um eine Antidiagonale zu berechnen, benötigt man die Inputdaten der zu berechnenden Antidiagonale und die Zellenwerte der zwei Antidiagonalen davor (siehe Abbildung 25 im Kapitel 7.3.2.3). Im Code 45 sieht man zwei For-Schleifen. Die äußere For-Schleife iteriert über die Antidiagonalen. Die innere Schleife berechnet die Zellen einer Antidiagonale und wird parallelisiert. Die Laufzeit der Funktion `matrixRowwiseInDiagonalwise()`, die die Daten von der Zeilenweise gespeicherten Startmatrix zur Antidiagonalweise gespeicherten Startmatrix überträgt, wird nicht in die Laufzeit miteinberechnet, da sie für einen optimalen Code nicht benötigt werden sollte. Optimal wäre, wenn die Startdaten direkt in die „Antidiagonalweise“-Startmatrix gespeichert werden oder die Startdaten erst bei der Berechnung ermittelt werden.

Code 45: Schritt 2 - Berechnen der Ergebnismatrix

```
matrixRowwiseInDiagonalwise(scorematrixCPU_parallel_Diagonalwise,
scorematrixCPU_parallel_Rowwise,N,M);

needlemanWunschCPU_Parallel(scorematrixCPU_parallel_Diagonalwise,N,M,penalty*-1);

void needlemanWunschCPU_Parallel(DatentypC *scorematrix,const int N,
const int M,const int gapcosts)
{
DatentypC m1,m2,m3,tmp;

int startdiagonaleZuBerechnen=3;
int sizediagonaleZuBerechnen=3;
int startdiagonale1davor=1;
int startdiagonale2davor=0;
int anzahlDiagonalen=N+M-1;
int d,di;
int anzahlDerZuBerechnendenWerte,startdiagonaleZuBerechnen_neu;
```

Fortsetzung von Code 45 auf der nächsten Seite

Fortsetzung von Code 45:

```
int startdiagonale2davor_neu;
sizediagonaleZuBerechnen--;

for (d=2; d<anzahlDiagonalen; d++)
{
    //Aufsteigende Diagonalgroesse
    if (d<M) {
        sizediagonaleZuBerechnen++;

        anzahlDerZuBerechnendenWerte=sizediagonaleZuBerechnen-2;
        startdiagonaleZuBerechnen_neu=startdiagonaleZuBerechnen+1;
        startdiagonale2davor_neu=startdiagonale2davor;
    }
    //Gleichbleibende Diagonalgroesse
    else if (d<N) {
        anzahlDerZuBerechnendenWerte=sizediagonaleZuBerechnen-1;
        startdiagonaleZuBerechnen_neu=startdiagonaleZuBerechnen+1;
        startdiagonale2davor_neu=startdiagonale2davor;
    }
    //Diagonalindex = N
    else if (d==N) {
        sizediagonaleZuBerechnen--;
        anzahlDerZuBerechnendenWerte=sizediagonaleZuBerechnen;
        startdiagonaleZuBerechnen_neu=startdiagonaleZuBerechnen;
        startdiagonale2davor_neu=startdiagonale2davor;
    }
    //Absteigende Diagonalgroesse
    else {
        sizediagonaleZuBerechnen--;

        anzahlDerZuBerechnendenWerte=sizediagonaleZuBerechnen;
        startdiagonaleZuBerechnen_neu=startdiagonaleZuBerechnen;
        startdiagonale2davor_neu=startdiagonale2davor+1;
    }
}

#pragma omp parallel default(none) shared(scorematrix, gapcosts,
    startdiagonaleZuBerechnen_neu, startdiagonale2davor_neu, startdiagonale1davor,
    anzahlDerZuBerechnendenWerte) private(di, m1, m2, m3, tmp)
{
    #pragma omp for
    for (di=0; di<anzahlDerZuBerechnendenWerte; di++)
    {
        //Diagonal
        m1=scorematrix[startdiagonale2davor_neu+di]+
            scorematrix[startdiagonaleZuBerechnen_neu+di];
        //Links
        m2=scorematrix[startdiagonale1davor+di]+gapcosts;
        //Oben
        m3=scorematrix[startdiagonale1davor+di+1]+gapcosts;

        //Moeglichkeit mit dem hoechsten Wert ermitteln
        tmp=m1;
        if (m3>tmp)
            tmp=m3;
        if (m2>tmp)
            tmp=m2;
        scorematrix[startdiagonaleZuBerechnen_neu+di]=tmp;
    }
}

startdiagonale2davor=startdiagonale1davor;
startdiagonale1davor=startdiagonaleZuBerechnen;
startdiagonaleZuBerechnen+=sizediagonaleZuBerechnen;
}
```

In den nachfolgenden Abbildungen ist der erreichte Speedup gegenüber dem sequentiellen Code dargestellt. In Abbildung 13 ist in der Laufzeit die Zeitdauer für das Anlegen der Arrays, für das Ausführen der Funktion und für das Freigeben der Arrays enthalten. Die Laufzeit des ersten Schritts (Initialisierung der Startmatrix) und des dritten Schritts (traceback) ist nicht enthalten. Diese Laufzeiten sind in Abbildung 14 inkludiert. Die Initialisierung der Inputmatrix wird immer mit 8 Threads parallel ausgeführt. Bei der Berechnung der Speedups für die Diagramme wurden die gemessenen Gesamtlaufzeiten verwendet. In Abbildung 13 und Abbildung 14 sieht man, dass 4 Threads einen höheren Speedup als 8 Threads ermöglichen. Außerdem verbessert das Flag „-xSSSE3“ beim Kompilieren den Speedup deutlich. Da sich auf dem Mainboard zwei CPUs befinden und die Verwendung von zwei CPUs gegenüber einer GPU nicht fair wäre, wurde die Ausführung auf die zweite CPU beschränkt (siehe Kapitel 5.2).

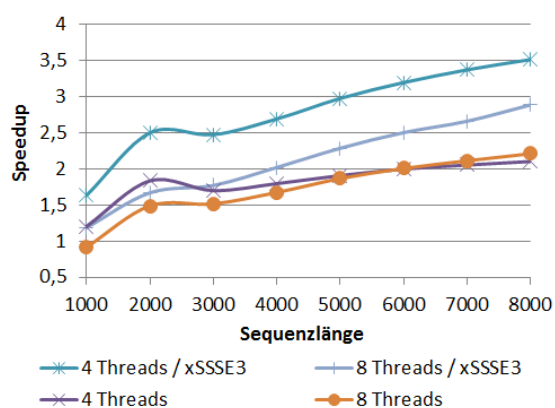


Abbildung 13: Schritt 2 (Berechnung der Ergebnismatrix) + Anlegen und Freigeben der Arrays

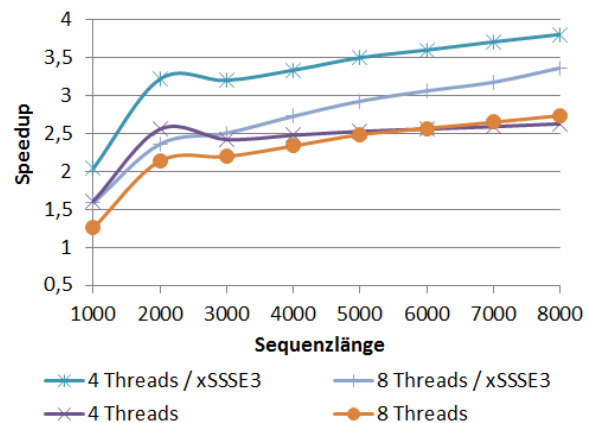


Abbildung 14: Alle drei Schritte des Algorithmus + Anlegen und Freigeben der Arrays

7.3 GPU-Programm mit Laufzeitmessungen

Als Basis für die Speedup-Diagramme wird die CPU-Konfiguration „Intel Xeon X5550, Standard Code, gapcosts, icc 11.1, -O3“ verwendet. Bei den Laufzeitmessungen wird für gapcosts immer 4 beim Aufruf des Programms übergeben. Im Programm wird dann die negative Zahl davon (-4) addiert. Die Länge der beiden zu vergleichenden Sequenzen ist identisch und ist in der X-Achse der Diagramme angegeben.

7.3.1 Global-Memory- und Shared-Memory-Programm

Das Global-Memory-Programm verwendet die Parallelisierungsmethode „pro Antidiagonale“ und wurde komplett selbst geschrieben. Da jede zu berechnende Antidiagonale von den Daten der zwei davor liegenden Antidiagonalen abhängig ist, kann nur Antidiagonale für Antidiagonale parallel berechnet werden. Dadurch entstehen Wavefronts. Bei 2 Sequenzen mit Länge N bekommt man $N+N-1$ Wavefronts. Die größte Wavefront enthält N parallele Elemente. Die kleinste Wavefront enthält 1 Element.

Der Ausgangscode des Shared-Memory-Programms (Parallelisierungsmethode: Pro Antidiagonale eines Blocks und pro Antidiagonale-Blöcke) ist ein Programm von der Universität von Virginia (USA). Die Funktion und Performance des Programms wird in [12] beschrieben. Auf das Paper [12] und auf den Code [13] wurde über die Nvidia-Seite (<http://www.nvidia.de>) in „CUDA-Präsentation“ in Kategorie „Life Sciences“ zugegriffen (siehe Abbildung 15). Über den „Code“-Link auf der Nvidia-Seite [11] wird man weitergeleitet auf [13]. Der Code [13] implementiert den Needleman-Wunsch.

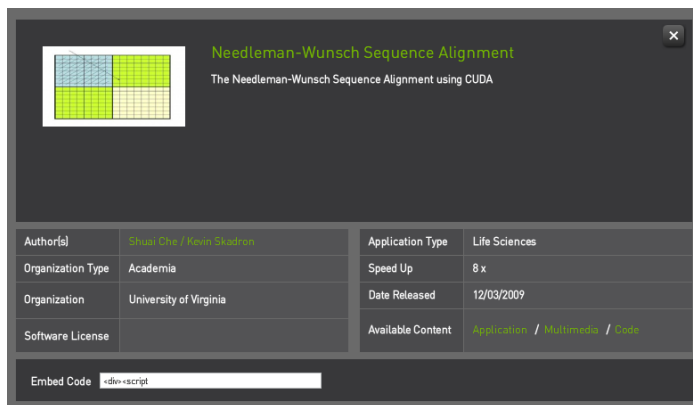


Abbildung 15: Paper und Code des Shared-Memory-Programms auf der Nvidia-Seite in CUDA-Präsentation [11]

Das Shared-Memory-Programm startet mit der Berechnung des Blocks ganz links oben in der Ergebnismatrix (siehe blauen umrandeten Bereich (ohne Randbereich, nur Bereich mit Ergebniswerte umrandet) in Abbildung 6 im Kapitel 7.1). Ein Block ist so groß, dass 8 Blöcke davon in den Shared Memory eines Multiprozessors passen. Der Grund ist, dass maximal 8 Blöcke von Threads auf einem Multiprozessor aktiv gehalten werden können. Bei einem $(n+1) \times (n+1)$ großen Block (erste Spalte und erste Zeile des Blocks ist Randbereich und wird nicht beschrieben) können n Threads durch Pipelining parallel arbeiten. Jeder Thread kümmert sich um die Berechnung einer Spalte des Blocks. Der erste Thread beginnt mit dem Berechnen der ersten Zelle ganz links oben (zweite Spalte, zweite Zeile, da erste Spalte und erste Zeile Randbereich ist). Sobald das erledigt ist, kann im zweiten Schritt der erste Thread die zweite Zelle (zweite Spalte, dritte Zeile) berechnen und der zweite Thread kann parallel mit der Berechnung der ersten Zelle (dritte Spalte, zweite Zeile) beginnen. Sobald der Block berechnet wurde und die Ergebniswerte in den Global Memory übertragen wurden, können im nächsten Schritt zwei Blöcke (siehe grüne umrandete Bereiche (jeweils ohne Randbereich, nur Bereich mit Ergebniswerte umrandet) in Abbildung 6 im Kapitel 7.1) parallel berechnet werden.

Die nachfolgenden Abbildungen stellen die Speedups der Basisprogramme dar. Die Basisprogramme haben als einzige Optimierung die Wahl der optimalen Blockgröße. Da die optimale Blockgröße bei der Shared-Memory-Version abhängig von der Größe des Shared Memory ist, haben die GPUs der Fermi- (48 KByte Shared Memory pro Multiprozessor) und GT200-Architektur (16 KByte Shared Memory pro Multiprozessor) unterschiedliche optimale Blockgrößen. Bei Fermi wird Blockgröße 27 und bei GT200 Blockgröße 15 verwendet. Beim Global-Memory-Programm wird die Blockgröße 16 für alle GPUs verwendet. Die Speedup-Diagramme zeigen die Gesamt- und Kernel-Speedups des Shared-Memory- und Global-Memory-Programms. Die Speedups der CUDA-Programme werden in Abbildung 16 (Tesla C2050), Abbildung 17 (Tesla C1060), Abbildung 18 (GTX 480) und Abbildung 19 (GTX 285) dargestellt. Die Speedups der OpenCL-Programme können in Abbildung 20 (Tesla C2050), Abbildung 21 (Tesla C1060), Abbildung 22 (GTX 480) und Abbildung 23 (GTX 285) eingesehen werden.

CUDA:

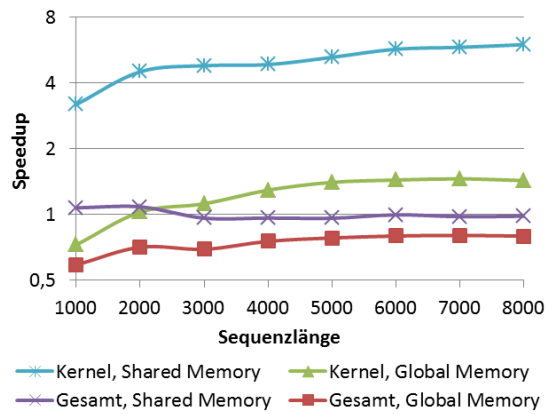


Abbildung 16: Tesla C2050

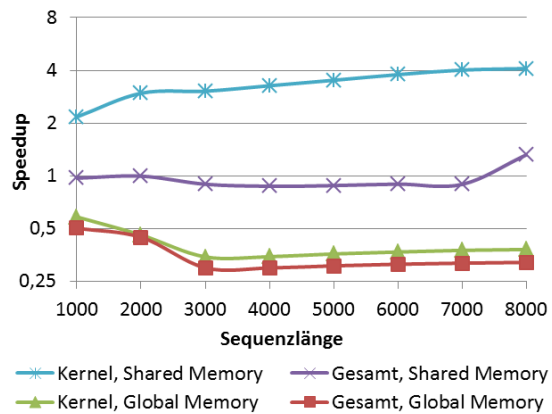


Abbildung 17: Tesla C1060

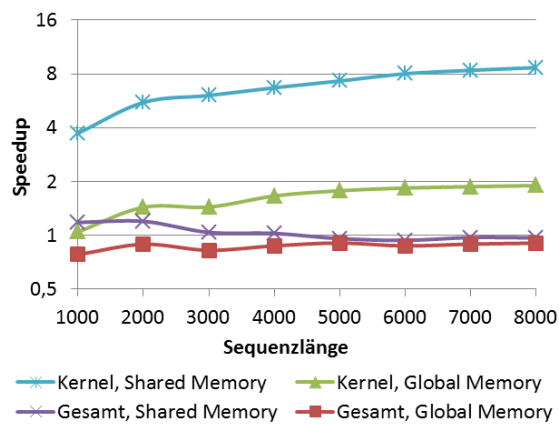


Abbildung 18: GTX 480

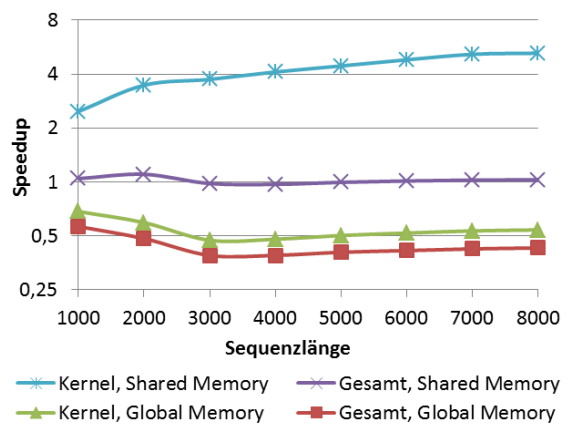


Abbildung 19: GTX 285

OpenCL:

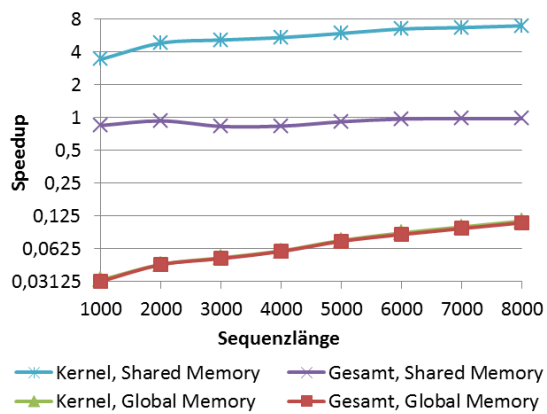


Abbildung 20: Tesla C2050

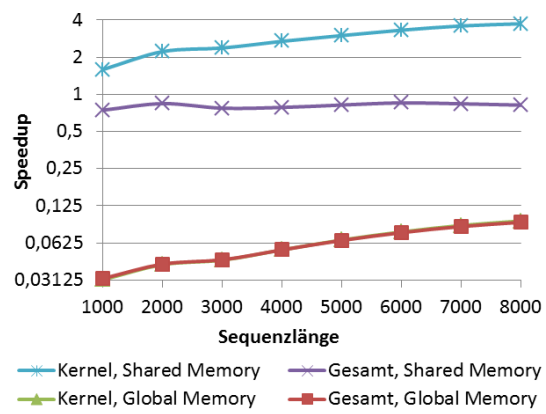


Abbildung 21: Tesla C1060

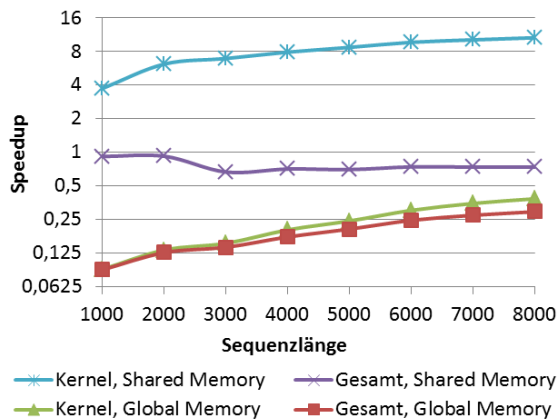


Abbildung 22: GTX 480

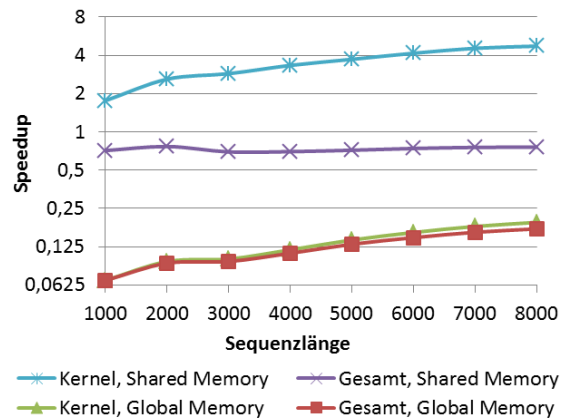


Abbildung 23: GTX 285

In den Speedup-Diagrammen zu den CUDA-Programmen (siehe Abbildung 16, Abbildung 17, Abbildung 18 und Abbildung 19) sieht man, dass der Gesamtspeedup des Shared-Memory-Programms und des Global-Memory-Programms bei der Fermi-Architektur (GTX 480, Tesla C2050) im Gegensatz zur GT200-Architektur (GTX 285, Tesla C1060) nahe beieinander liegen. Der Grund ist, dass die Fermi-Architektur für den Global Memory eine Cache-Hierarchie besitzt. Bei den Speedup-Diagrammen zu den OpenCL-Programmen (siehe Abbildung 20, Abbildung 21, Abbildung 22 und Abbildung 23) kann man bei der Fermi- und der GT200-Architektur einen großen Unterschied zwischen Global Memory und Shared Memory-Programm erkennen. Das OpenCL-Global-Memory-Programm ist deutlich schlechter als das CUDA-Global-Memory-Programm.

7.3.2 Optimierungen am Global-Memory-Programm

Die Messungen in den Unterkapiteln beziehen sich auf die GPU GTX 480 und das CUDA-Programm.

7.3.2.1 Finden der optimalen Blockgröße

Bei $N=8000$ und $M=8000$ wurde die Threadanzahl pro Block in 16er Schritten (=halbes Warp) erhöht und die Laufzeit ermittelt. Als GPU wurde die GTX 480 verwendet. Die niedrigste Laufzeit hat die Blockgröße 16 und aus diesem Grund wurde die Blockgröße als optimale Blockgröße gewählt.

7.3.2.2 Verwendung des Short Datentyps

Statt dem Int-Datentyp (4 Byte) wird der Short-Datentyp (2 Byte) verwendet, wenn die Sequenzlänge bei beiden Sequenzen 8000 nicht überschreitet. Ab Sequenzlänge 8200 bekommt man sonst falsche Ergebnisse, da der Ergebniswert den größten Wert des Datentyps Short überschreitet. Durch die Verwendung des Short-Datentyps wird die Übertragungsgröße halbiert. Dadurch kann die Start- und Ergebnismatrix zwischen CPU-Arbeitsspeicher und GPU-Arbeitsspeicher doppelt so schnell übertragen werden.

Bei $N=8000$, $M=8000$ kann der Gesamt-Speedup von 0,9 auf 1,27 und der Kernel-Speedup von 1,9 auf 1,95 gesteigert werden (siehe Abbildung 24).

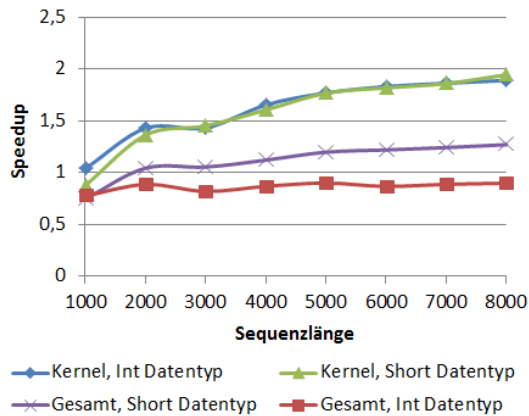


Abbildung 24: Auswirkung des Short-Datentyps auf den Speedup

7.3.2.3 Memory Coalescing

Die Idee ist es, die Matrix antidiagonalweise abzuspeichern. Dadurch ist ein coalesced-Zugriff auf die Zellenwerte bei der parallelen Berechnung möglich (siehe Abbildung 25). Die Threads eines Warps (=32 Threads) greifen somit immer auf Elemente zu, die nebeneinander im Speicher liegen. Dadurch entstehen beim Zusammenfassen der Datenzugriffe pro Warp (Fermi) bzw. halben Warp (GT200) nur ein Zugriff und es muss keine Global-Memory-Bandbreite verschwendet werden. Würde jeder Thread einen eigenen Datenzugriff auslösen und es müsste somit die kleinste mögliche Segmentgröße von 32 Bytes [5] für den Datenzugriff verwendet werden, würde sehr viel Bandbreite verschwendet werden (siehe Kapitel 2.2.3.2).

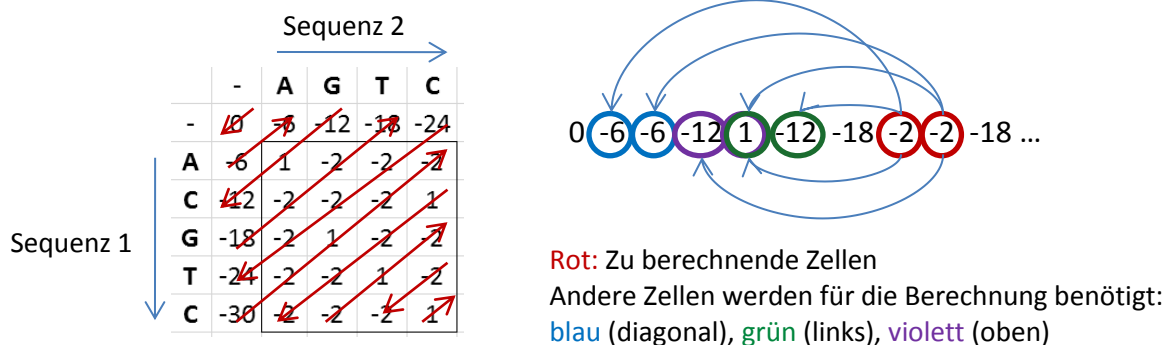


Abbildung 25: Coalesced-Zugriff auf Zellenwerte bei antidiagonaler Speicherung

Bei N=8000, M=8000 kann der Gesamt-Speedup von 1,27 auf 1,89 und der Kernel-Speedup von 1,95 auf 3,9 erhöht werden (siehe Abbildung 26).

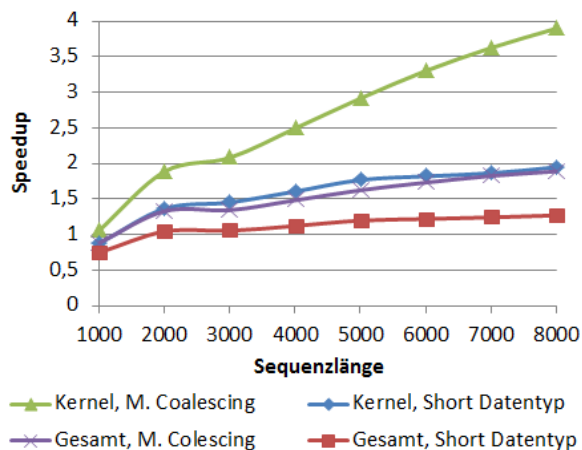


Abbildung 26: Auswirkung von Memory Coalescing auf den Speedup

7.3.3 Optimierungen am Shared-Memory-Programm

Die Messungen im Kapitel und in den Unterkapiteln beziehen sich auf die GPU GTX 480 und das CUDA-Programm.

Im Paper zur Masterarbeit [14] wurde erst nach der Wahl der optimalen Blockgröße der Ausgangscode analysiert (siehe Abbildung 30). Hier wird schon davor begonnen.

Bei der Analyse des Ausgangscodes sieht man, dass drei Teile die meiste Zeit benötigen (siehe Abbildung 27). Die Übertragung der Daten vom CPU-Arbeitsspeicher zum GPU-Arbeitsspeicher hat 38% Anteil, das Zurückübertragen hat 25% Anteil und der Kernel hat 21% Anteil.

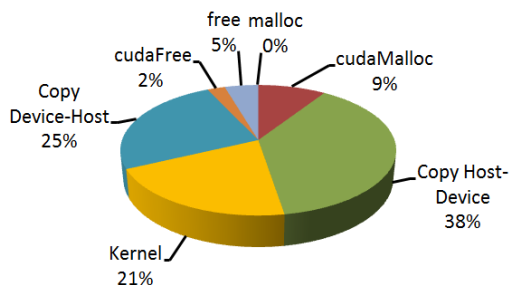


Abbildung 27: Ausgangscode bei Toolkit 4.0

Der Aufwand von cudaMalloc() wurde erst mit Toolkit 4.0 deutlich größer. Bei Toolkit 3.2 wurde 1% gemessen (siehe Abbildung 28).

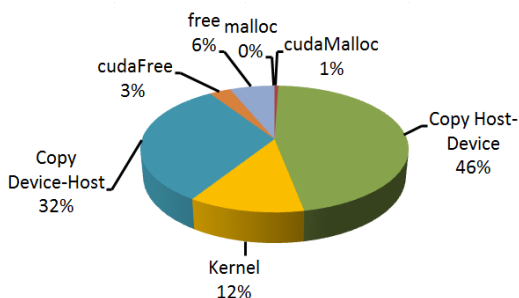


Abbildung 28: Ausgangscode bei Toolkit 3.2 mit optimaler Blockgröße 27

Der Code zum Shared-Memory-Programm mit allen Optimierungen befindet sich im Anhang A.2.

7.3.3.1 Wahl der optimalen Blockgröße

Mit der Wahl der optimalen Blockgröße führt man eine Optimierung am Kernel (21% der Laufzeit bei $N=8000$, $M=8000$) durch, indem der komplette Shared Memory ausgenutzt wird. Beim Ausgangscode ist Blockgröße 16 eingestellt. Dieser Code ist optimiert auf die GTX 260 [12], die 16 KB Shared Memory pro Multiprozessor hat [5].

Die optimale Blockgröße berechnet sich durch die Shared-Memory-Größe pro Multiprozessor (49152 Bytes bei GTX 480), durch die maximale Anzahl von aktiven Blöcken pro Multiprozessor (8 Blöcke), durch die Datentypgröße (4 Bytes bei int) und durch die Größe der zwei Shared-Memory-Matrizen pro Block im Ausgangscode. Die erste Shared-Memory-Matrix ist $(\text{Blockgröße}+1) * (\text{Blockgröße}+1)$ und die zweite Shared-Memory-Matrix ist $(\text{Blockgröße}) * (\text{Blockgröße})$ groß [13].

Zunächst wird die maximale Anzahl von Werten für jeden Block über die Formel „Shared-Memory-Größe pro Multiprozessor / maximale Anzahl von aktiven Blöcken pro Multiprozessor / Datentypgröße“ berechnet. Bei der Fermi-GPU hat man $49152/8/4=1536$ Int-Werte für jeden Block verfügbar. Der Ausgangscode hat pro Block zwei Shared-Memory-Matrizen, die fast gleich groß und quadratisch sind (siehe vorigen Absatz). Somit wird 1536 durch 2 dividiert und die Quadratwurzel daraus gezogen. Das Ergebnis ist 27,7. Somit ist die optimale Blockgröße 27. Die Anzahl der Werte pro Block entspricht dann 1513 $(=(28*28)+(27*27))$ und ist somit kleiner als 1536.

Bei $N=8000$, $M=8000$ kann der Gesamt-Speedup von 0,86 auf 0,96 und der Kernel-Speedup von 4,04 auf 8,67 gesteigert werden (siehe Abbildung 29). Die Blockgröße wurde von 16 auf 27 vergrößert.

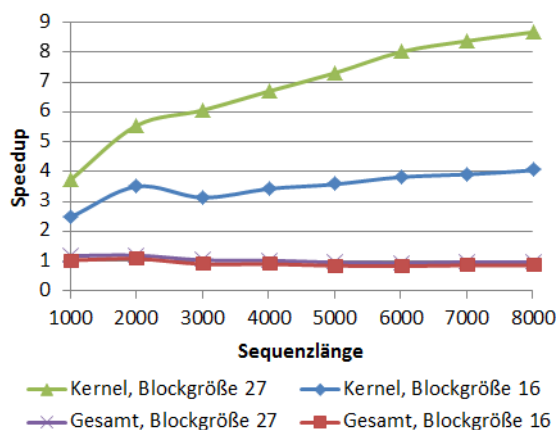


Abbildung 29: Auswirkung der optimalen Blockgröße auf den Speedup

Mit der optimalen Blockgröße 27 beträgt die Kernellaufzeit statt 21% nur mehr 11% der Gesamtlaufzeit (siehe Abbildung 30). Die Abbildung 30 befindet sich auch im Paper zur Masterarbeit [14]. Die Prozentzahlen in den darauffolgenden Optimierungen beziehen sich auf Abbildung 30.

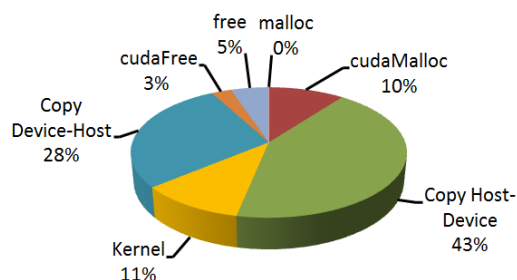


Abbildung 30: Mit optimaler Blockgröße

7.3.3.2 Verwendung von Short (2 Bytes) anstatt von Int (4 Bytes)

Mit der Verwendung des Short- anstatt des Int-Datentyps halbiert man die Datenübertragungsgröße und führt so eine Optimierung an der Übertragung der Daten vom CPU-Arbeitsspeicher zum GPU-Arbeitsspeicher (43% der Laufzeit bei N=8000, M=8000 beim Ausgangscode) und wieder zurück (28% der Laufzeit bei N=8000, M=8000 beim Ausgangscode) durch.

Außerdem kann die Anzahl der Shared-Memory-Werte pro Block von 1536 auf 3072 verdoppelt werden und die optimale Blockgröße auf 38 vergrößert werden. Damit wird der Kernel schneller und man hat eine Optimierung am Kernel (11% der Laufzeit bei N=8000, M=8000 beim Ausgangscode).

Der Nachteil der Optimierung ist, dass die Sequenzlänge bei beiden Sequenzen 8000 nicht überschreiten darf. Ab Sequenzlänge 8200 bekommt man falsche Ergebnisse, da die Ergebniswerte den maximalen Wert des Short-Datentyps überschreiten.

Bei N=8000, M=8000 kann der Gesamt-Speedup von 0,96 auf 1,86 und der Kernel-Speedup von 8,67 auf 10,98 gesteigert werden (siehe Abbildung 31). Die Blockgröße wurde von 27 auf 38 erhöht.

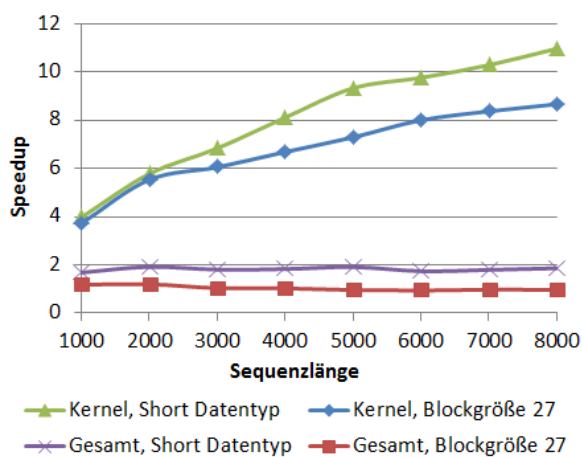


Abbildung 31: Auswirkung des Short-Datentyps auf den Speedup

In der Abbildung 32 sieht man, dass sich die Laufzeit bei N=8000 und M=8000 fast halbiert hat und dass der Kernel mit 16% (vorher: 11%) nun einen größeren Anteil an der Gesamtlaufzeit hat.

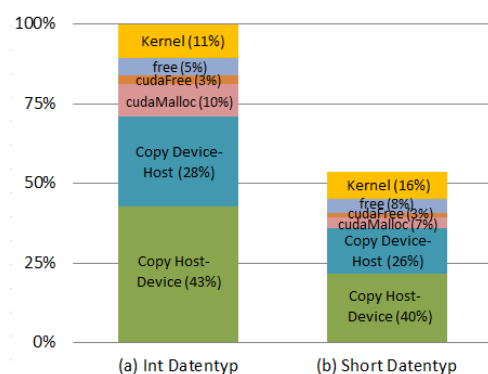


Abbildung 32: Auswirkung des Short-Datentyps bei N=8000 und M=8000

7.3.3.3 Nur eine Inputmatrix und eine shared Teilmatrix

Durch 1 statt 2 Inputmatrizen (Eine Inputmatrix mit ganz wenigen Inputdaten am linken und oberen Rand) kann eine Halbierung der zu übertragenden Inputdaten erreicht werden und somit an der Übertragung der Daten vom CPU-Arbeitsspeicher zum GPU-Arbeitsspeicher (43% der Laufzeit bei $N=8000$, $M=8000$ beim Ausgangscode) optimiert werden. Die zwei Inputmatrizen im Ausgangscode sind nicht notwendig, da ausgelesene Werte (siehe rote Werte in Inputmatrix 1 in Abbildung 33) ohne Probleme mit den Ergebniswerten überschrieben werden können (siehe blaue Werte in Inputmatrix 2 in Abbildung 33).

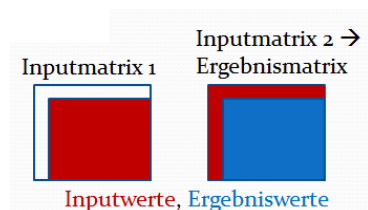


Abbildung 33: Zwei Inputmatrizen im Ausgangscode (Informationen von [13])

Die Optimierung im Global Memory kann auch bei den Teilmatrizen im Shared Memory angewendet werden. Durch die Verwendung einer gemeinsamen Input- und Outputteilmatrix kann die optimale Blockgröße von 38 auf 54 erhöht werden und eine Optimierung am Kernel (11% der Laufzeit bei $N=8000$, $M=8000$ beim Ausgangscode) erreicht werden.

Bei $N=8000$, $M=8000$ kann der Gesamt-Speedup von 1,86 auf 2,84 und der Kernel-Speedup von 11 auf 14,3 gesteigert werden (siehe Abbildung 34). Die Blockgröße wurde von 38 auf 54 erhöht.

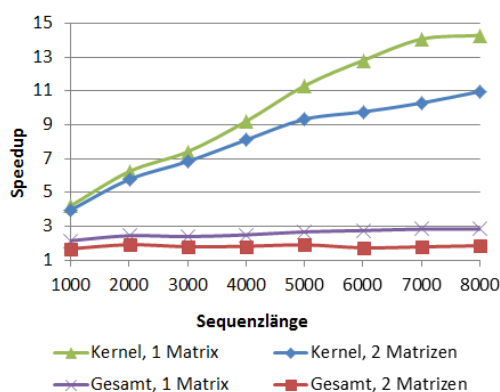


Abbildung 34: Auswirkung der Verwendung von nur einer Inputmatrix und einer Shared-Teilmatrix auf den Speedup

7.3.3.4 Initialisierung der Inputmatrix auf GPU anstatt Übertragung

Durch Initialisierung der Inputmatrix auf der GPU kann eine Optimierung an der Übertragung der Daten vom CPU-Arbeitsspeicher zum GPU-Arbeitsspeicher (43% der Laufzeit bei $N=8000$, $M=8000$ beim Ausgangscode) erzielt werden. Dadurch erspart man sich die Startmatrix mit Hilfe der CPU zu erzeugen und sie zur GPU zu übertragen. Dazu wird die Sequenz 1 (N -Werte), Sequenz 2 (M -Werte), Substitutionsmatrix (20×20 Werte) und gapcosts/penalty (1 Wert) zur GPU übertragen (siehe Code 47 und Code 49) und mit diesen Inputdaten wird mit Hilfe von drei Kernels die Startmatrix mit $(N+1) \times (M+1)$ Werten im Arbeitsspeicher der GPU erzeugt (siehe Code 48). Damit müssen nur $(N+M+20 \times 20+1) \times 2$ Bytes anstatt von $(N+1) \times (M+1) \times 2$ Bytes übertragen werden.

Die Substitutionsmatrix wird im Constant Memory und die zwei Sequenzen werden im Global Memory gespeichert (siehe Code 46).

Code 46: Anlegen der Arrays

```
// Außerhalb der main()-Funktion:
// Anlegen des Arrays für die Substitutionsmatrix im Constant Memory
__constant__ DatentypGPU Substitutionsmatrix_const[20*20];

// In der main()-Funktion:
// Anlegen der Arrays für die zwei Sequenzen im Global Memory
DatentypGPU *d_Sequenz1,*d_Sequenz2;
cudaMalloc((void**)&d_Sequenz1,max_rows*sizeof(DatentypGPU));
cudaMalloc((void**)&d_Sequenz2,max_cols*sizeof(DatentypGPU));
```

Code 47: Übertragen der Inputdaten

```
// In der main()-Funktion:
cudaMemcpy(d_Sequenz1,Sequenz1,max_rows*sizeof(DatentypGPU),cudaMemcpyHostToDevice);
cudaMemcpy(d_Sequenz2,Sequenz2,max_cols*sizeof(DatentypGPU),cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(Substitutionsmatrix_const,h_Substitutionsmatrix,20*20*
    sizeof(DatentypGPU));
```

Code 48: Kernels zur Erzeugung der Startmatrix (Blockgröße: 256(X-Richtung) x 1(Y-Richtung))

```
__global__ void reference_cuda_initialisieren_const_memory(
    DatentypGPU *reference_cuda,const DatentypGPU *d_Sequenz1,
    const DatentypGPU *d_Sequenz2,const int max_cols,const int max_rows)
{
    int bi = blockIdx.y;
    int bj = blockIdx.x;
    int ti = threadIdx.y;
    int tj = threadIdx.x;

    int i=bi*blockDim.y+ti+1;
    int j=bj*blockDim.x+tj+1;

    if(j<max_cols)
        reference_cuda[i*max_cols+j]=Substitutionsmatrix_const[
            d_Sequenz1[i-1]*20+d_Sequenz2[j-1]];
}

__global__ void matrix_cuda_erste_Zeile_initialisieren(DatentypGPU
    *matrix_cuda,const int max_cols,const DatentypGPU penalty)
{
    int bj = blockIdx.x;
    int tj = threadIdx.x;
    int j=bj*blockDim.x+tj;

    if(j<max_cols)
        matrix_cuda[j]=-j*penalty;
}

__global__ void matrix_cuda_erste_Spalte_initialisieren(DatentypGPU
    *matrix_cuda,const int max_rows,const int max_cols,const DatentypGPU penalty)
{
    int bi = blockIdx.x;
    int ti = threadIdx.x;
    int i=bi*blockDim.x+ti;

    if(i<max_rows)
        matrix_cuda[i*max_cols]=-i*penalty;
}
```

Code 49: Aufruf der Kernels und Freigeben der zwei Sequenz-Arrays im Global Memory

```
//Aufruf des ersten Kernels (Initialisierung aller Zellen bis auf erste Spalte und
//erste Zeile)
int blocksize=256;
Anzahl_von_Bloecken_in_XRichtung=(max_cols-1)/blocksize;
if(((max_cols-1)%blocksize)!=0)
    Anzahl_von_Bloecken_in_XRichtung+=1;
Anzahl_von_Bloecken_in_YRichtung=(max_rows-1);
dimGrid.x=Anzahl_von_Bloecken_in_XRichtung;
dimGrid.y=Anzahl_von_Bloecken_in_YRichtung;
dimBlock.x=blocksize;
dimBlock.y=1;
reference_cuda_initialisieren_const_memory<<<dimGrid,dimBlock>>>(matrix_cuda,
    d_Sequenz1,d_Sequenz2,max_cols,max_rows);

//Freigeben der zwei Sequenz-Arrays im Global Memory
cudaFree(d_Sequenz1);
cudaFree(d_Sequenz2);

//Aufruf des zweiten Kernels (Initialisierung der ersten Zeile der Startmatrix)
blocksize=256;
if(max_cols%blocksize!=0)
    Anzahl_von_Bloecken_in_XRichtung+=1;
dimGrid.x=Anzahl_von_Bloecken_in_XRichtung;
dimGrid.y=1;
dimBlock.x=blocksize;
dimBlock.y=1;
matrix_cuda_erste_Zeile_initialisieren<<<dimGrid,dimBlock>>>(matrix_cuda,max_cols,
    penalty);

//Aufruf des dritten Kernels (Initialisierung der ersten Spalte der Startmatrix)
blocksize=256;
Anzahl_von_Bloecken_in_YRichtung=(max_rows/blocksize);
if(max_rows%blocksize!=0)
    Anzahl_von_Bloecken_in_YRichtung+=1;
dimGrid.x=Anzahl_von_Bloecken_in_YRichtung;
dimGrid.y=1;
dimBlock.x=blocksize;
dimBlock.y=1;
matrix_cuda_erste_Spalte_initialisieren<<<dimGrid,dimBlock>>>(matrix_cuda,max_rows,
    max_cols,penalty);
```

Bei N=8000, M=8000 kann der Gesamt-Speedup von 2,84 auf 3,86 gesteigert werden (siehe Abbildung 35).

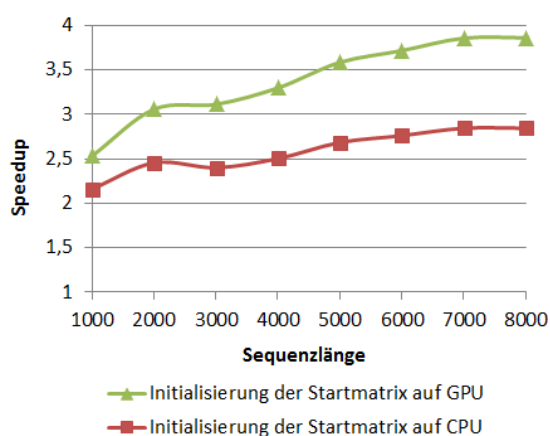


Abbildung 35: Auswirkung der Initialisierung der Startmatrix auf der GPU auf den Speedup

Der Speedup von „Initialisierung der Inputmatrix auf GPU“ fällt noch höher aus, da nun in der GPU-Laufzeit das Initialisieren der Startmatrix enthalten ist. Wenn man bei der CPU die Laufzeit der

Initialisierung dazu addiert, kann bei N=8000, M=8000 der Gesamt-Speedup von 2,84 auf 5,79 gesteigert werden (siehe Abbildung 36).

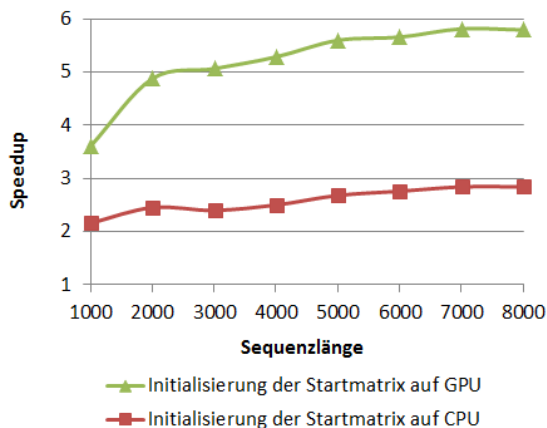


Abbildung 36: Inkludierung der Initialisierung der Startmatrix in die CPU-Laufzeit

7.3.3.5 Nicht erfolgreich: Pinned Memory statt Pageable Memory

Üblicherweise wird Speicher für Arrays am Host mit der Funktion malloc() angelegt und mit der Funktion free() wieder freigegeben. Mit malloc() wird ein Array als Pageable Memory angelegt und kann somit durch das Betriebssystem vom CPU-Arbeitsspeicher auf die Festplatte ausgelagert werden (=Paging) [5] [29]. Mit der CUDA-Funktion cudaMallocHost() wird ein Array als Page-locked/Pinned-Speicher angelegt und eine Auslagerung des Arrays auf die Festplatte wird verhindert [5] [29]. Mit der Funktion cudaFreeHost() wird der Pinned-Speicher wieder freigegeben [5]. Die OpenCL-Funktionen für das Anlegen und Freigeben von Pinned Memory stehen im Kapitel 3.3.3.

Das Verwenden von Page-locked/Pinned-Speicher wirkt sich folgendermaßen auf die Laufzeit aus. Als GPU wird die GTX 480 und als Programm der CUDA-NW-Ausgangscode mit der Optimierung „optimale Blockgröße“ (Blockgröße 27) verwendet. Die Problemgröße ist N=8000 und M=8000. Durch die Verwendung von Pinned Memory wird der Transport der Daten vom Host zum Device um 1,3 Mal und vom Device zum Host um 1,02 schneller (siehe Abbildung 37 und Abbildung 38). Hingegen ist die Funktion cudaMallocHost() und cudaFreeHost() deutlich langsamer. CudaMallocHost() ist um 12340 Mal langsamer als malloc() und cudaFreeHost() 20 Mal langsamer als free(). Wenn man die Gesamtlaufzeit betrachtet, wird das Programm um den Faktor 2,5 langsamer.

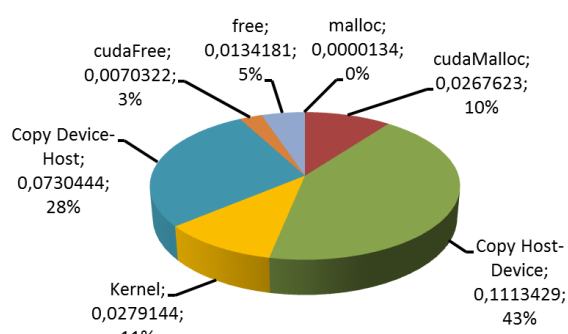


Abbildung 37: Pageable Memory (Hostmatrix mit malloc() anlegen)

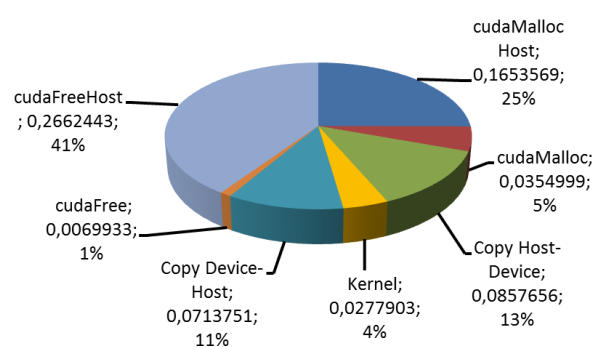


Abbildung 38: Page-locked/Pinned Memory (Hostmatrix mit cudaMallocHost() anlegen)

7.3.4 Zusammenfassung der Gesamtspeedups aller Optimierungen in einem Diagramm

In diesem Kapitel werden die Gesamtspeedups aller Optimierungen in einem Diagramm pro GPU und Programmiersprache zusammengefasst.

GPU GTX 480

Die auf der GTX 480 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 39 (CUDA-Programme) und Abbildung 40 (OpenCL-Programme).

In Abbildung 39 (CUDA-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp, „1 Inputmatrix und 1 shared Teilmatrix“ und die Initialisierung der Inputmatrix auf der GPU den Speedup deutlich verbessern. Die optimale Blockgröße für den Ausgangscode (Blockgröße 27) steigert die Performance ein wenig. Beim Global-Memory-Programm führen beide Optimierungen (Short-Datentyp und Antidiagonale-Speicherung) zu einer deutlichen Erhöhung des Speedups.

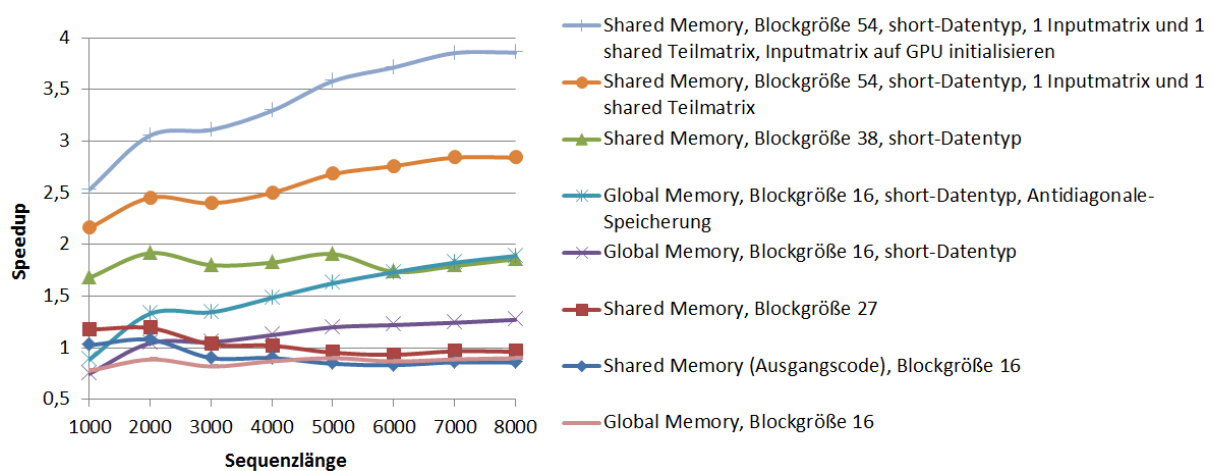


Abbildung 39: GTX 480, CUDA

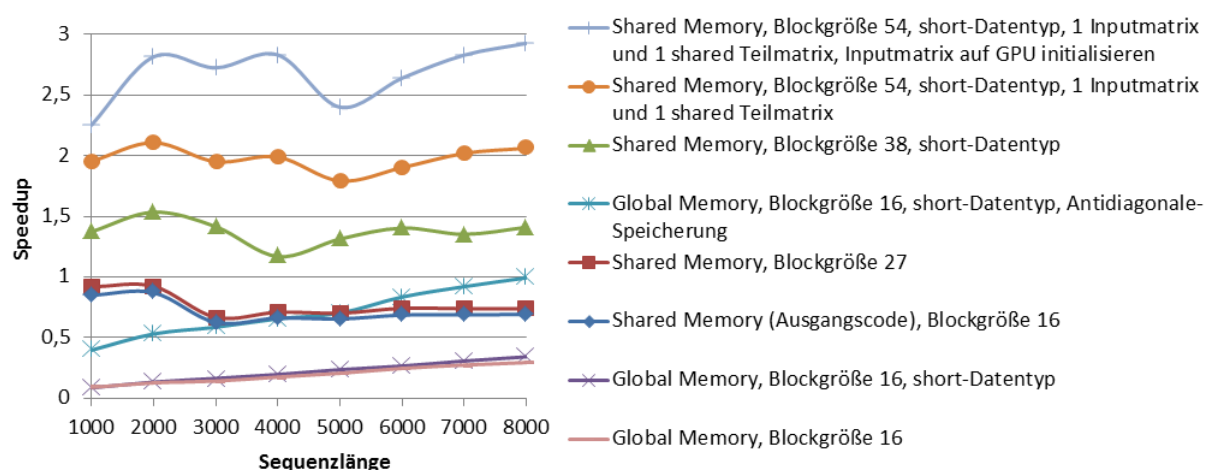


Abbildung 40: GTX 480, OpenCL

In Abbildung 40 (OpenCL-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp, „1 Inputmatrix und 1 shared Teilmatrix“ und die Initialisierung der Inputmatrix auf der GPU den Speedup deutlich verbessern. Eine sehr geringe Auswirkung auf den Speedup hat die optimale Blockgröße für den Ausgangscode (Blockgröße 27). Beim Global-Memory-Programm führt die Kombination der Optimierungen Short-Datentyp und Antidiagonale-Speicherung/Memory-Coalescing

zu einer deutlichen Erhöhung des Speedups. Die Optimierung Short-Datentyp beim Global-Memory-Programm führt zirka zu einer Halbierung der Gesamtlaufzeit ohne Kernellaufzeit und die Auswirkung dieser Optimierung wird dadurch erst sichtbar, wenn die Kernellaufzeit durch die Optimierung Antidiagonale-Speicherung/Memory-Coalescing einen geringeren Anteil an der Gesamtlaufzeit einnimmt.

GPU GTX 285

Die auf der GTX 285 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 41 (CUDA-Programme) und Abbildung 42 (OpenCL-Programme).

In Abbildung 41 (CUDA-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp, „1 Inputmatrix und 1 shared Teilmatrix“ und die Initialisierung der Inputmatrix auf der GPU den Speedup deutlich verbessern. Die optimale Blockgröße für den Ausgangscode (Blockgröße 15) steigert die Performance ein wenig. Beim Global-Memory-Programm führt die Kombination der Optimierungen Short-Datentyp und Antidiagonale-Speicherung/Memory-Coalescing zu einer deutlichen Erhöhung des Speedups. Die Optimierung Short-Datentyp beim Global-Memory-Programm führt zirka zu einer Halbierung der Gesamtlaufzeit ohne Kernellaufzeit und die Auswirkung dieser Optimierung wird dadurch erst sichtbar, wenn die Kernellaufzeit durch die Optimierung Antidiagonale-Speicherung/Memory-Coalescing einen geringeren Anteil an der Gesamtlaufzeit einnimmt.

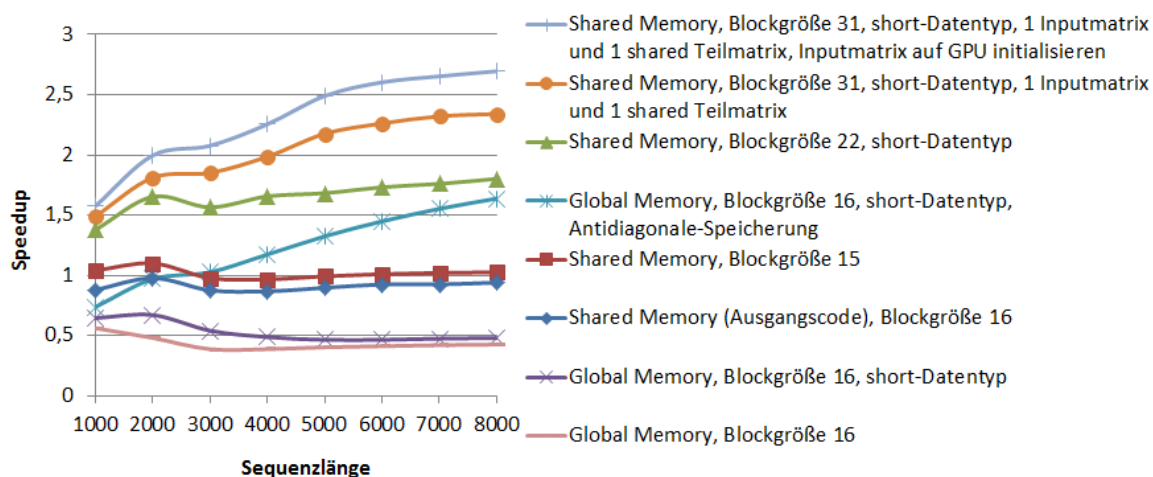


Abbildung 41: GTX 285, CUDA

In Abbildung 42 (OpenCL-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp, „1 Inputmatrix und 1 shared Teilmatrix“ und die Initialisierung der Inputmatrix auf der GPU den Speedup deutlich verbessern. Eine sehr geringe Auswirkung auf den Speedup hat die optimale Blockgröße für den Ausgangscode (Blockgröße 15). Beim Global-Memory-Programm führt die Kombination der Optimierungen Short-Datentyp und Antidiagonale-Speicherung/Memory-Coalescing zu einer deutlichen Erhöhung des Speedups. Die Optimierung Short-Datentyp beim Global-Memory-Programm führt zirka zu einer Halbierung der Gesamtlaufzeit ohne Kernellaufzeit und die Auswirkung dieser Optimierung wird dadurch erst sichtbar, wenn die Kernellaufzeit durch die Optimierung Antidiagonale-Speicherung/Memory-Coalescing einen geringeren Anteil an der Gesamtlaufzeit einnimmt.

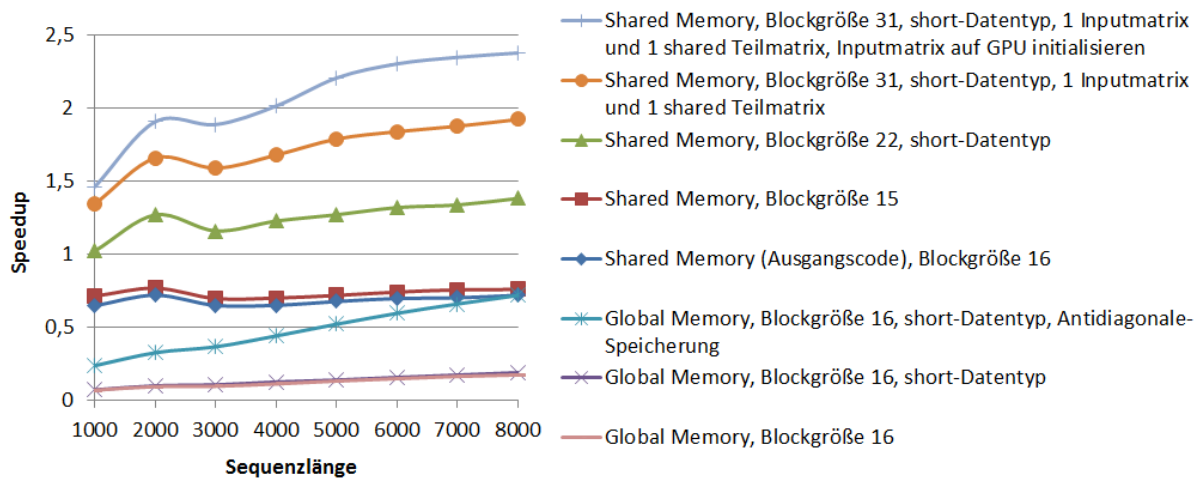


Abbildung 42: GTX 285, OpenCL

GPU Tesla C2050

Die auf der Tesla C2050 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 43 (CUDA-Programme) und Abbildung 44 (OpenCL-Programme).

In Abbildung 43 (CUDA-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp, „1 Inputmatrix und 1 shared Teilmatrix“ und die Initialisierung der Inputmatrix auf der GPU den Speedup deutlich verbessern. Die optimale Blockgröße für den Ausgangscode (Blockgröße 27) steigert die Performance ein wenig. Beim Global-Memory-Programm führen beide Optimierungen (Short-Datentyp und Antidiagonale-Speicherung) zu einer deutlichen Erhöhung des Speedups.

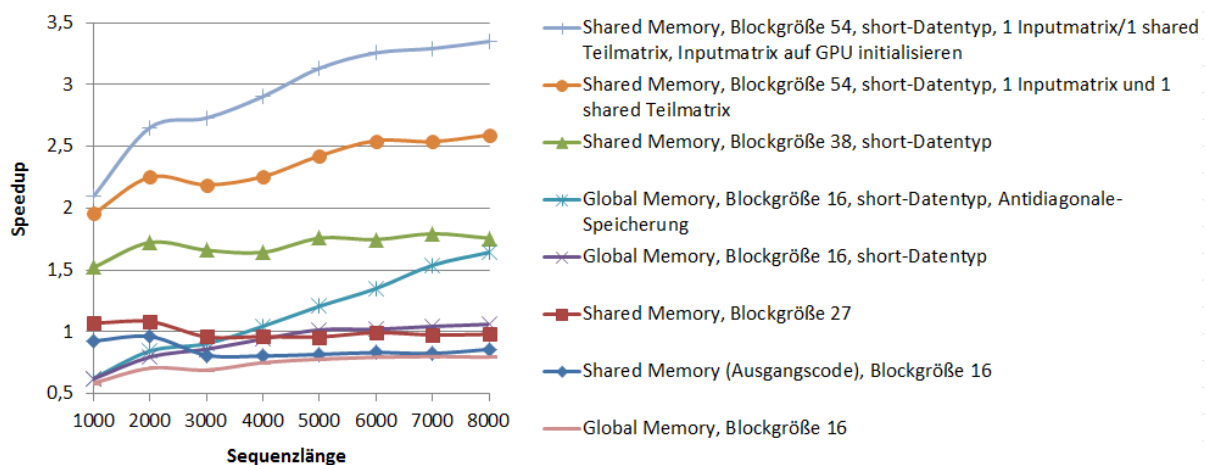


Abbildung 43: Tesla C2050, CUDA

In Abbildung 44 (OpenCL-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp, „1 Inputmatrix und 1 shared Teilmatrix“ und die Initialisierung der Inputmatrix auf der GPU den Speedup deutlich verbessern. Die optimale Blockgröße für den Ausgangscode (Blockgröße 27) steigert die Performance ein wenig. Beim Global-Memory-Programm führt die Kombination der Optimierungen Short-Datentyp und Antidiagonale-Speicherung/Memory-Coalescing zu einer deutlichen Erhöhung des Speedups. Die Optimierung Short-Datentyp beim Global-Memory-Programm führt zirka zu einer Halbierung der Gesamtlaufzeit ohne Kernellaufzeit und die Auswirkung dieser Optimierung wird dadurch erst sichtbar, wenn die Kernellaufzeit durch die Optimierung Antidiagonale-Speicherung/Memory-Coalescing einen geringeren Anteil an der Gesamtlaufzeit einnimmt.

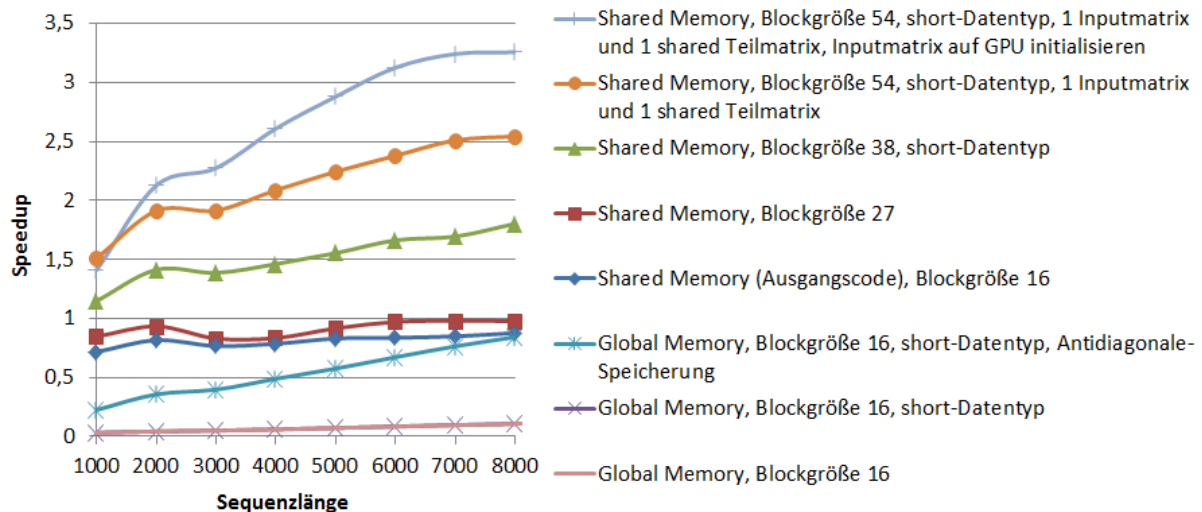


Abbildung 44: Tesla C2050, OpenCL

GPU Tesla C1060

Die auf der Tesla C1060 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 45 (CUDA-Programme) und Abbildung 46 (OpenCL-Programme).

In Abbildung 45 (CUDA-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp und „1 Inputmatrix und 1 shared Teilmatrix“ den Speedup deutlich verbessern. Die Optimierungen „Initialisierung der Inputmatrix auf der GPU“ und optimale Blockgröße für den Ausgangscode (Blockgröße 15) steigern die Performance ein wenig. Beim Global-Memory-Programm führt die Kombination der Optimierungen Short-Datentyp und Antidiagonale-Speicherung/Memory-Coalescing zu einer deutlichen Erhöhung des Speedups. Die Optimierung Short-Datentyp beim Global-Memory-Programm führt zirka zu einer Halbierung der Gesamtlaufzeit ohne Kernellaufzeit und die Auswirkung dieser Optimierung wird dadurch erst sichtbar, wenn die Kernellaufzeit durch die Optimierung Antidiagonale-Speicherung/Memory-Coalescing einen geringeren Anteil an der Gesamtlaufzeit einnimmt.

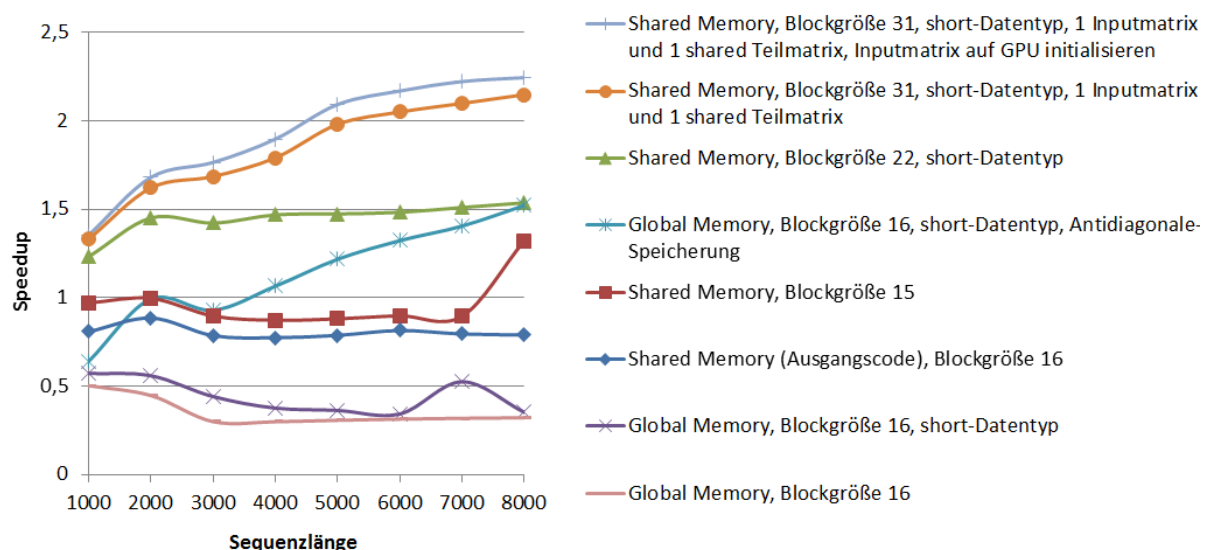


Abbildung 45: Tesla C1060, CUDA

In Abbildung 46 (OpenCL-Programme) sieht man beim Shared-Memory-Programm, dass der Short-Datentyp und „1 Inputmatrix und 1 shared Teilmatrix“ den Speedup deutlich verbessern. Die Optimierungen „Initialisierung der Inputmatrix auf der GPU“ und optimale Blockgröße für den Ausgangscode (Blockgröße 15) steigern die Performance ein wenig. Beim Global-Memory-Programm führt die Kombination der Optimierungen Short-Datentyp und Antidiagonale-Speicherung/Memory-Coalescing zu einer deutlichen Erhöhung des Speedups. Die Optimierung Short-Datentyp beim Global-Memory-Programm führt zirka zu einer Halbierung der Gesamtlaufzeit ohne Kernellaufzeit und die Auswirkung dieser Optimierung wird dadurch erst sichtbar, wenn die Kernellaufzeit durch die Optimierung Antidiagonale-Speicherung/Memory-Coalescing einen geringeren Anteil an der Gesamtlaufzeit einnimmt.

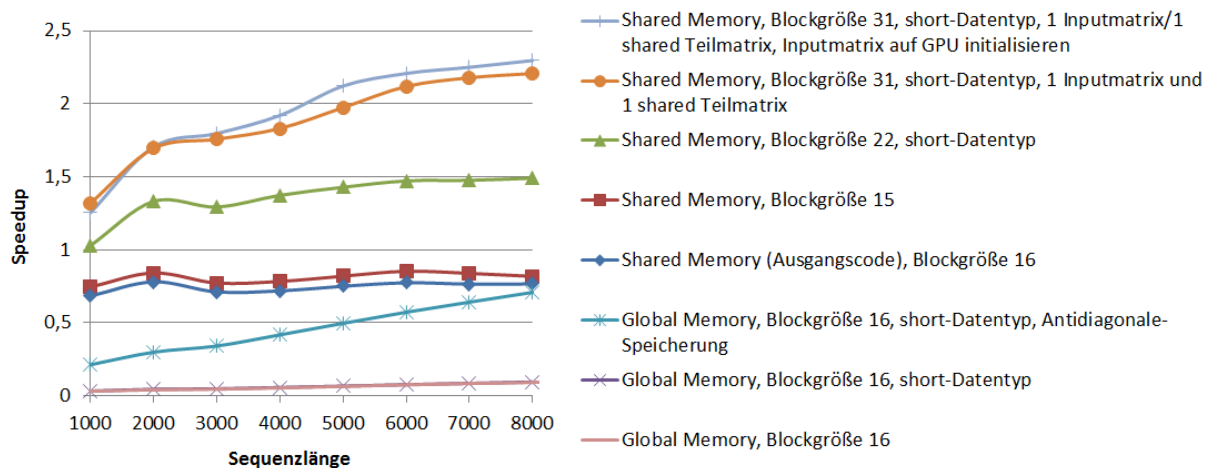


Abbildung 46: Tesla C1060, OpenCL

7.4 Diskussion der gemessenen GPU-Laufzeitdaten

7.4.1 Auswirkung der Optimierungen auf das CUDA- und OpenCL-Programm

Die CUDA-Diagramme der GTX 480 (Abbildung 48, Abbildung 50, Abbildung 58, Abbildung 66, Abbildung 70, Abbildung 74, Abbildung 78 und Abbildung 80) in den zwei Unterkapiteln sind identisch mit den CUDA-Diagrammen (Abbildung 24, Abbildung 26, Abbildung 29, Abbildung 31, Abbildung 34, Abbildung 35, Abbildung 37 und Abbildung 38) im Kapitel 7.3.2 und Kapitel 7.3.3 und werden für den direkten Vergleich mit den OpenCL-Diagrammen der GTX 480 nochmals abgebildet.

Die Abbildung 62 ist identisch mit der Abbildung 78 (CUDA-Teilzeiten der GTX 480) und die Abbildung 63 ist identisch mit der Abbildung 79 (OpenCL-Teilzeiten der GTX 480).

7.4.1.1 Global-Memory-Programm

Wenn man den erreichten Gesamtspeedup des CUDA- und OpenCL-Global-Memory-Programms mit allen Optimierungen (Short-Datentyp und Memory Coalescing) auf der Fermi-Architektur (GTX 480) mit der älteren Architektur GT200 (GTX 285) vergleicht, zeigt sich, dass bei beiden Architekturen das CUDA-Programm deutlich schneller als das OpenCL-Programm ist (siehe Abbildung 47). Bei $N=8000$, $M=8000$ ist das CUDA-Programm bei der GTX 480 um 90,5% und bei der GTX 285 um 128% schneller als das OpenCL-Programm. Da der OpenCL-Programmcode und der CUDA-Programmcode bis auf die CUDA und OpenCL spezifischen Funktionen und Variablen identisch sind, kann der Grund für den starken Unterschied nur beim Compiler liegen.

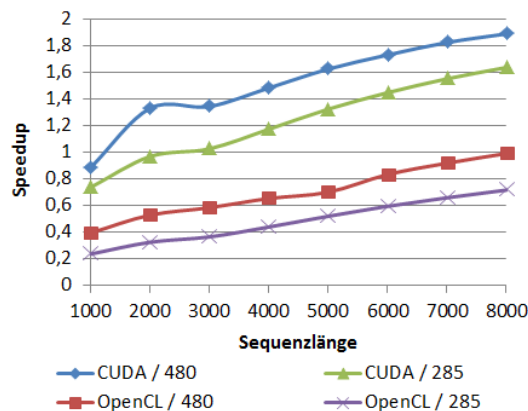


Abbildung 47: Fermi und GT200

Optimierung „Short Datentyp“

Die Optimierung „Short Datentyp“ wirkt sich bei der GTX 480 bei CUDA und OpenCL positiv auf den Gesamtspeedup aus und verändert den Kernelspeedup kaum (siehe Abbildung 48 und Abbildung 49). Bei $N=8000$, $M=8000$ und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 41,3% und bei OpenCL um 16,1%. Der Kernelspeedup verbessert sich bei CUDA um 2,9% und bei OpenCL um 1,2% (GTX 480, $N=8000$, $M=8000$). Wenn die Gesamtlaufzeit ohne Kernellaufzeit betrachtet wird, ergibt sich eine Verbesserung bei CUDA um 108% und bei OpenCL um 102% (GTX 480, $N=8000$, $M=8000$).

Der Speedup des OpenCL-Programms liegt deutlich unter dem Speedup des CUDA-Programms. Bei $N=8000$, $M=8000$, der GTX 480 und der Optimierung Short-Datentyp ist der Gesamtspeedup des CUDA-Programms um 273% höher als der Gesamtspeedup des OpenCL-Programms. Der Kernelspeedup des CUDA-Programms ist um 402% höher als der Kernelspeedup des OpenCL-Programms (GTX 480, $N=8000$, $M=8000$). Vor der Optimierung Short-Datentyp ist der Gesamtspeedup des CUDA-Programms um 207% höher als der Gesamtspeedup des OpenCL-

Programms und der Kernelspeedup des CUDA-Programms ist um 394% höher als der Kernelspeedup des OpenCL-Programms (GTX 480, N=8000, M=8000). Wie im ersten Absatz des Kapitels erwähnt, kann der Grund für den großen Laufzeitunterschied zwischen CUDA- und OpenCL-Programm nur beim Compiler liegen.

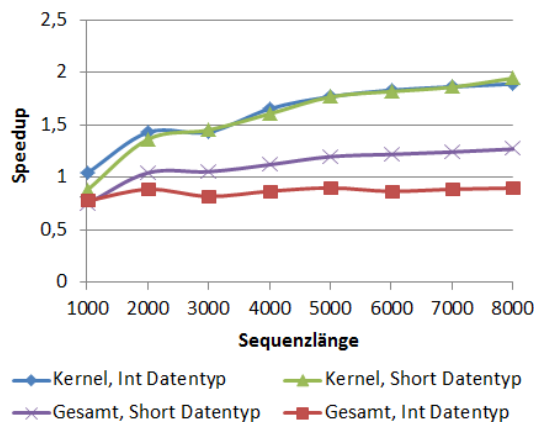


Abbildung 48: CUDA, GTX 480

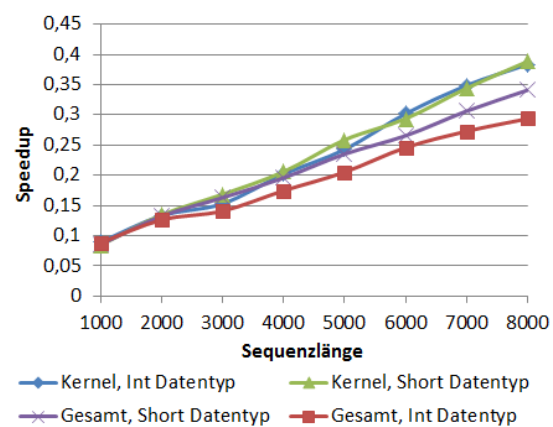


Abbildung 49: OpenCL, GTX 480

Optimierung „Memory Coalescing“

Die Optimierung „Memory Coalescing“ wirkt sich bei der GTX 480 bei CUDA und OpenCL positiv auf den Gesamtspeedup und Kernelspeedup aus (siehe Abbildung 50 und Abbildung 51). Bei N=8000, M=8000 und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 48,6% und bei OpenCL um 191%. Der Kernelspeedup verbessert sich bei CUDA um 100% und bei OpenCL um 336% (GTX 480, N=8000, M=8000).

Da die Optimierung „Memory Coalescing“ nur auf den Kernel wirkt, führt alleine die Verbesserung des Kernelspeedups zu der Verbesserung des Gesamtspeedups. Die Gesamtlaufzeit ohne Kernellaufzeit bleibt somit gleich. Bei N=8000, M=8000 und der GTX 480 beträgt die Gesamtlaufzeit ohne Kernellaufzeit vor der Optimierung 0,0718 Sekunden bei CUDA und 0,108 Sekunden bei OpenCL und nach der Optimierung 0,0698 Sekunden bei CUDA und 0,1082 Sekunden bei OpenCL. Der geringe Unterschied von 2,9% (0,0718 Sekunden vor der Optimierung versus 0,0698 Sekunden nach der Optimierung) bei CUDA kann auf die Laufzeitsschwankung (vor der Optimierung: 3,4%, nach der Optimierung: 0,14% bei 10 Laufzeitmessungen) zurückgeführt werden.

Die Optimierung „Memory Coalescing“ ermöglicht bei OpenCL eine starke Verbesserung der Performance und bringt die OpenCL-Performance etwas näher an CUDA. Jedoch ist das OpenCL-Programm bei der GTX 480 noch immer um die Hälfte langsamer. Bei N=8000, M=8000, der GTX 480 und allen Optimierungen (Short-Datentyp und Memory Coalescing) ist der Gesamtspeedup des CUDA-Programms um 90,5% höher als der Gesamtspeedup des OpenCL-Programms. Der Kernelspeedup des CUDA-Programms ist um 130% höher als der Kernelspeedup des OpenCL-Programms (GTX 480, N=8000, M=8000). Wie im ersten Absatz des Kapitels erwähnt, kann der Grund für den großen Laufzeitunterschied zwischen CUDA- und OpenCL-Programm nur beim Compiler liegen.

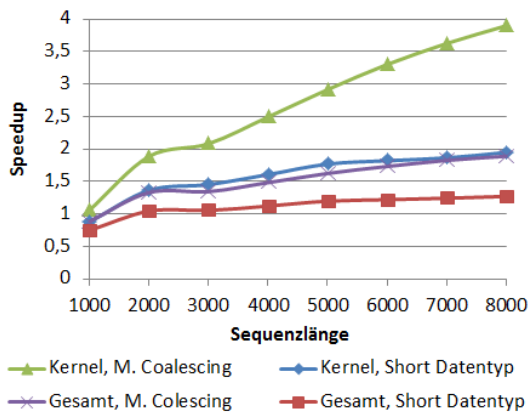


Abbildung 50: CUDA, GTX 480

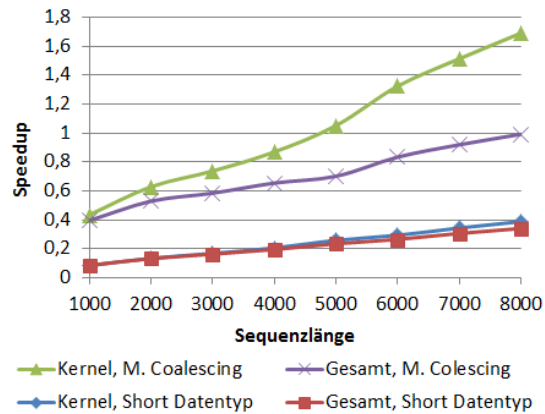


Abbildung 51: OpenCL, GTX 480

7.4.1.2 Shared-Memory-Programm

Wenn man den erreichten Gesamtspeedup des CUDA- und OpenCL-Shared-Memory-Programms mit allen Optimierungen auf der Fermi-Architektur (GTX 480) mit der älteren Architektur GT200 (GTX 285) vergleicht, zeigt sich, dass das OpenCL-Programm ab $N=5000$, $M=5000$ bei der GTX 480 deutlich schlechter als das CUDA-Programm ist (siehe Abbildung 52). Der Kernel ist dafür nicht verantwortlich, da beim Kernelspeedup der Unterschied zwischen CUDA und OpenCL über die Problemgrößen in etwa gleich bleibt (siehe Abbildung 54, der Grund für den besseren Kernelspeedup beim OpenCL-Programm im Vergleich zum CUDA-Programm wird bei der Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darunter erklärt). Der Sprung nach unten zwischen $N=4000$, $M=4000$ und $N=5000$, $M=5000$ wird durch das starke Ansteigen der Laufzeit für die Initialisierung der Startmatrix auf der GPU verursacht. Wenn diese Laufzeit von der Gesamtlaufzeit entfernt wird, gibt es diesen Sprung nicht (siehe Abbildung 56). In der Abbildung 56 sieht man, dass weiterhin ein Unterschied zwischen CUDA und OpenCL besteht. Dieser Unterschied wird durch die Laufzeit für die Übertragung der Ergebnismatrix vom Device zum Host verursacht, da bei OpenCL im Vergleich zu CUDA die Bandbreite beim Übertragen von Daten zwischen Host und Device niedriger ist (siehe Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darunter). Bei der Tesla-Fermi-GPU C2050 und dem Shared-Memory-Programm mit allen Optimierungen hat man beim Gesamtspeedup nur bei den kleineren Problemgrößen eine größere Abweichung zwischen CUDA und OpenCL (siehe Abbildung 53). Der Unterschied beim Kernelspeedup zwischen CUDA und OpenCL ist bei der Tesla C2050 über die Problemgrößen in etwa gleich groß (siehe Abbildung 55).

Bei Betrachten des Gesamtspeedups bei $N=8000$, $M=8000$ ist das CUDA-Programm bei der Tesla C2050 (Fermi) um 2,9%, bei der GTX 480 (Fermi) um 31,8% und bei der GTX 285 (GT200) um 13,2% schneller als das OpenCL-Programm. Interessant ist, dass bei der Tesla C1060 (GT200) die OpenCL-Performance fast identisch mit der CUDA-Performance ist (siehe Abbildung 53).

Für die Speedup-Kurven von Abbildung 52, Abbildung 53, Abbildung 54, Abbildung 55 und Abbildung 56 wurde das Shared-Memory-Programm mit allen Optimierungen (Optimale Blockgröße (GTX 285 und Tesla C1060: 31, GTX 480 und Tesla C2050: 54), Short Datentyp, eine Inputmatrix und eine shared Teilmatrix und Inputmatrix auf GPU initialisieren) verwendet.

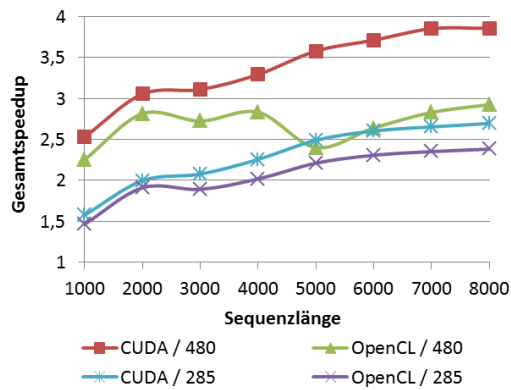


Abbildung 52: GTX 480 versus GTX 285
(Gesamtspeedup)

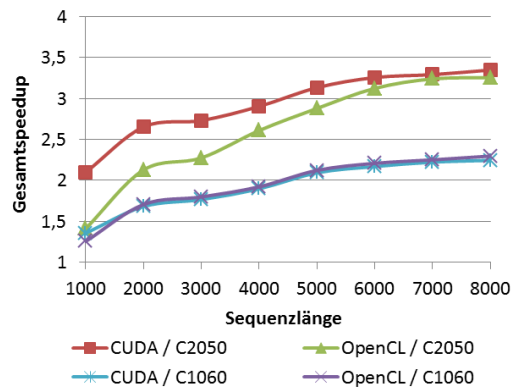


Abbildung 53: Tesla C2050 versus Tesla
C1060 (Gesamtspeedup)

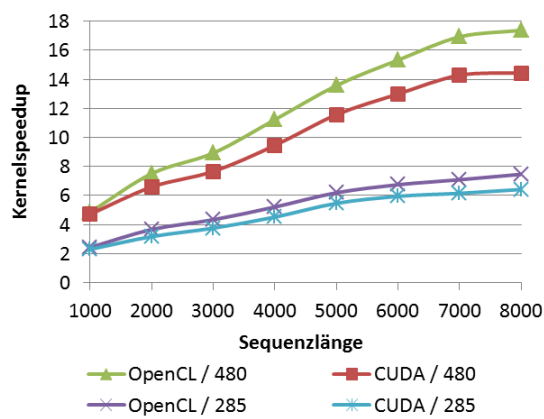


Abbildung 54: GTX 480 versus GTX 285
(Kernelspeedup)

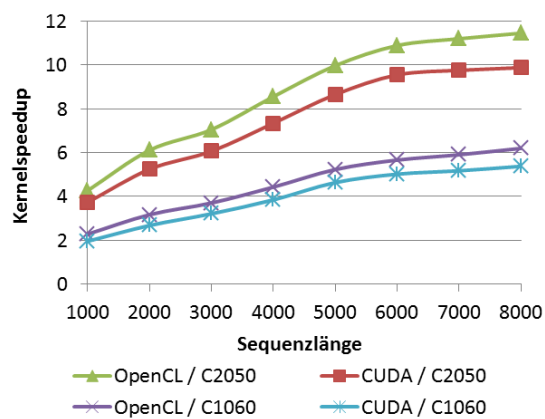


Abbildung 55: Tesla C2050 versus Tesla C1060
(Kernelspeedup)

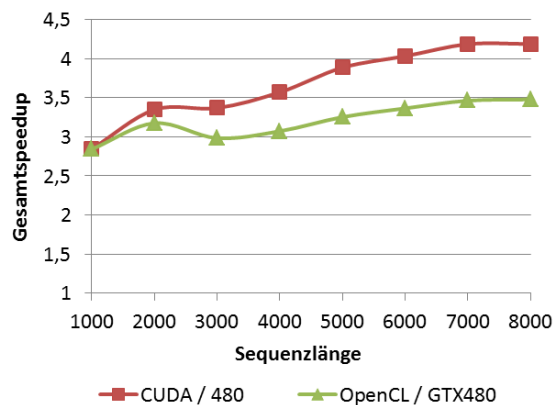


Abbildung 56: Gesamtspeedup bei GTX 480 ohne der Laufzeit für die Initialisierung der Startmatrix auf der GPU

Optimierung „Wahl der optimalen Blockgröße“

Die Optimierung „Wahl der optimalen Blockgröße“ wirkt sich bei CUDA und OpenCL positiv aus (siehe Abbildung 58, Abbildung 59, Abbildung 60 und Abbildung 61). Die Blockgröße beim Ausgangscode des Shared-Memory-Programms wird bei der GTX 480 und der Tesla C2050 von 16 auf 27 erhöht. Bei $N=8000$, $M=8000$ und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 12,2% und bei OpenCL um 6,8%. Der Kernelspeedup verbessert sich bei CUDA um 114% und bei OpenCL um 108% (GTX 480, $N=8000$, $M=8000$). Bei $N=8000$, $M=8000$ und der Tesla C2050 verbessert sich der

Gesamtspeedup bei CUDA um 14,2% und bei OpenCL um 11,5%. Der Kernelspeedup verbessert sich bei CUDA um 103% und bei OpenCL um 92,4% (Tesla C2050, N=8000, M=8000).

Bei den Diagrammen zur ersten Optimierung „Wahl der optimalen Blockgröße“ des Shared-Memory-Programms sieht man, dass der Kernelspeedup bei OpenCL im Vergleich zu CUDA höher ist (siehe Abbildung 58, Abbildung 59, Abbildung 60 und Abbildung 61). Bei N=8000, M=8000, der GTX 480 und der Optimierung „Wahl der optimalen Blockgröße“ ist der Kernelspeedup des OpenCL-Programms um 23% höher als der Kernelspeedup des CUDA-Programms. Ein Grund ist, dass die Maximum-Bestimmungsfunktion von drei Werten im Shared-Memory-CUDA-Code vom Ausgangscode [13] übernommen wurde (siehe 2x if-else im Code 50) und andere Varianten für eine schnellere Ausführung der Maximum-Bestimmungsfunktion von drei Werten beim CUDA-Shared-Memory-Programm nicht ausprobiert wurden. Beim Transformieren des Shared-Memory-CUDA-Codes in den Shared-Memory-OpenCL-Code wurde der Code für die Bestimmung des Maximums von drei Werten geändert (siehe „max() verschachtelt“ im Code 50). Im CPU-Code und im Global Memory-Programm wird eine dritte Variante verwendet (siehe 2x if im Code 50). Wenn man die Kernel-Speedups beim CUDA- und OpenCL-Shared Memory-Programm mit allen Optimierungen für alle drei Varianten misst, ist die „max() verschachtelt“-Variante bei CUDA und OpenCL am schnellsten (siehe Abbildung 57). OpenCL ist jedoch bei allen drei Varianten immer schneller als CUDA und die Ursache kann nur beim Compiler liegen, da der OpenCL-Programmcode und der CUDA-Programmcode bis auf die CUDA und OpenCL spezifischen Funktionen und Variablen identisch sind. OpenCL ist bei der Variante „2x if-else“ um 7,4%, bei der Variante „max() verschachtelt“ um 9,5% und bei der Variante „2x if“ um 5% schneller als CUDA (Kernellaufzeit bei Shared-Memory-Programm mit allen Optimierungen, GTX 480, N=8000, M=8000). Die Variante „max() verschachtelt“ ist bei CUDA um 10% schneller als die Variante „2x if-else“ (Kernellaufzeit bei Shared Memory-Programm mit allen Optimierungen, GTX 480, N=8000, M=8000). Insgesamt ergibt sich damit der zirka 20% höhere Kernelspeedup des OpenCL-Programms gegenüber dem CUDA-Programm (10% Unterschied zwischen „2x if-else“ und „max() verschachtelt“ und 9,5% Unterschied bei „max() verschachtelt“ zwischen CUDA und OpenCL).

Bei N=8000, M=8000, der Tesla C2050 und der Optimierung „Wahl der optimalen Blockgröße“ ist der Kernelspeedup des OpenCL-Programms um 16,3% höher als der Kernelspeedup des CUDA-Programms. Hier gelten dieselben Gründe wie bei der GTX 480 (siehe vorigen Absatz).

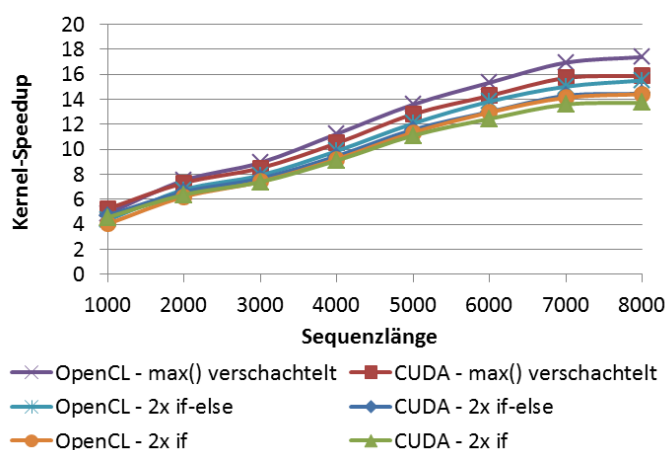


Abbildung 57: Kernel-Speedup bei unterschiedlicher Implementierung der Bestimmung des Maximums von drei Werten (Shared-Memory-Programm mit allen Optimierungen, GTX 480)

Code 50: Bestimmung des Maximums von drei Werten

CUDA-Shared-Memory-Programm [13] (adaptiert)	OpenCL-Shared-Memory-Programm	Global-Memory-Programm und CPU-Programm
2x if-else	max() verschachtelt	2x if
<pre> __device__ __host__ int maximum(int a, int b, int c) { int k; if(a <= b) k = b; else k = a; if(k <= c) return(c); else return(k); } __global__ void kernelname(...) { ... ergebniswert = maximum(a,b,c); ... } </pre>	<pre> __kernel void kernelname(...) { ... ergebniswert=max(a,max(b,c)); ... } </pre>	<pre> tmp=m1; if (m2>tmp) tmp=m2; if (m3>tmp) tmp=m3; ergebniswert=tmp; </pre>

Im Gegensatz zum Kernelspeedup ist der Gesamtspeedup bei CUDA höher. Bei N=8000, M=8000, der GTX 480 und der Optimierung „Wahl der optimalen Blockgröße“ ist der Gesamtspeedup des CUDA-Programms um 30,6% höher als der Gesamtspeedup des OpenCL-Programms. Der Grund für den schlechteren Gesamtspeedup bei OpenCL ist, dass die Übertragung der Daten über den PCI-Bus bei OpenCL länger dauert. Während bei der Dauer des Transfers zwischen Device und Host (siehe Abbildung 62, Abbildung 63, Abbildung 64 und Abbildung 65) in beide Richtungen bei der Tesla C2050 zwischen CUDA und OpenCL wenig Unterschied ist (0,07 versus 0,08 Sekunden und 0,11 versus 0,127 Sekunden), ist bei der GTX 480 der Unterschied zwischen CUDA und OpenCL groß (0,07 versus 0,1 Sekunden und 0,11 versus 0,19 Sekunden).

Bei N=8000, M=8000, der Tesla C2050 und der Optimierung „Wahl der optimalen Blockgröße“ ist der Gesamtspeedup des CUDA-Programms um 0,3% höher als der Gesamtspeedup des OpenCL-Programms. Der Unterschied ist sehr gering, da, wie bereits im vorigen Absatz erwähnt, der Unterschied zwischen CUDA und OpenCL bei der Laufzeit der Übertragungen zwischen Device und Host gering ist.

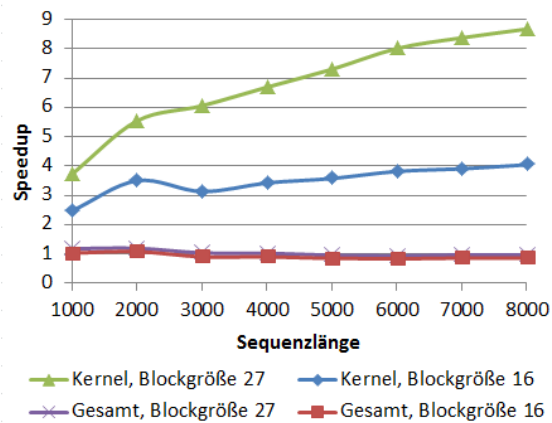


Abbildung 58: CUDA, GTX 480

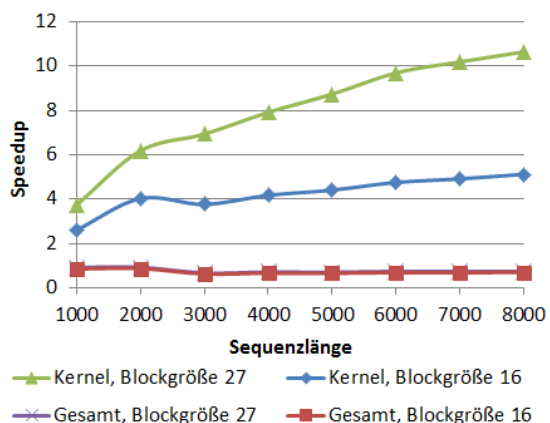


Abbildung 59: OpenCL, GTX 480

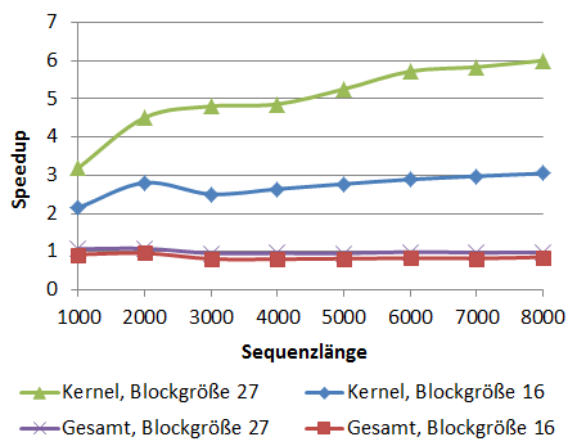


Abbildung 60: CUDA, Tesla C2050

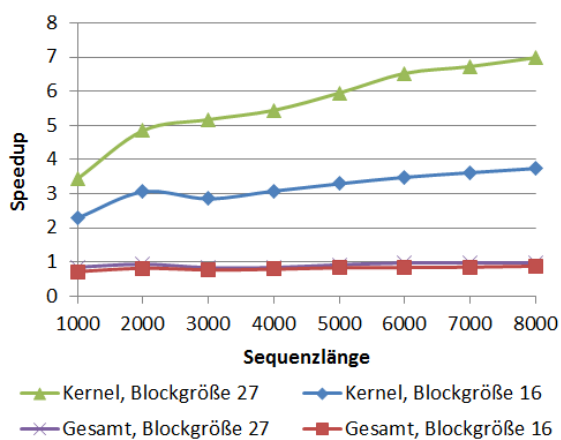


Abbildung 61: OpenCL, Tesla C2050

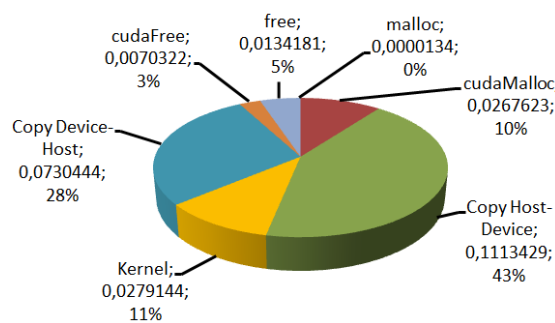


Abbildung 62: CUDA, GTX 480, N=8000, M=8000

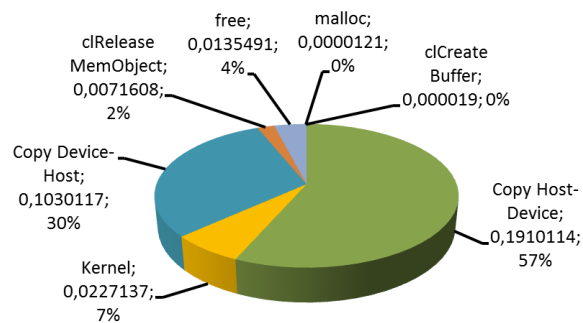


Abbildung 63: OpenCL, GTX 480, N=8000, M=8000

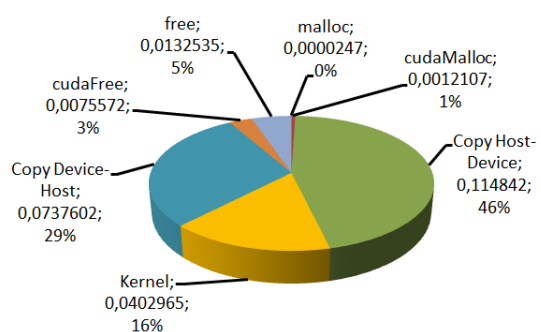


Abbildung 64: CUDA, Tesla C2050, N=8000, M=8000

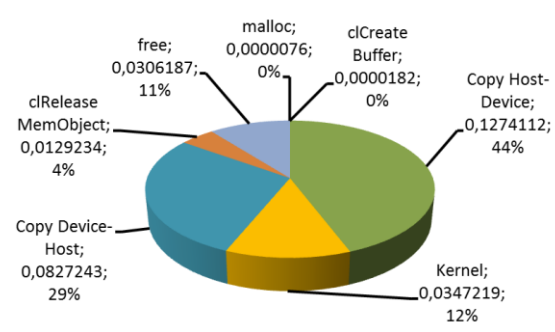


Abbildung 65: OpenCL, Tesla C2050, N=8000, M=8000

Optimierung „Verwendung von Short (2 Bytes) anstatt von Int (4 Bytes)“

Die Optimierung „Verwendung von Short (2 Bytes) anstatt von Int (4 Bytes)“ wirkt sich bei CUDA und OpenCL positiv aus (siehe Abbildung 66, Abbildung 67, Abbildung 68 und Abbildung 69). Die Blockgröße konnte durch die Optimierung bei der GTX 480 und der Tesla C2050 von 27 auf 38 erhöht werden. Bei N=8000, M=8000 und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 92,9% und bei OpenCL um 90,9%. Der Kernelspeedup verbessert sich bei CUDA um 26,6% und bei OpenCL um 24,1% (GTX 480, N=8000, M=8000). Bei N=8000, M=8000 und der Tesla C2050 verbessert sich der Gesamtspeedup bei CUDA um 79,3% und bei OpenCL um 84,3%. Der Kernelspeedup verbessert sich bei CUDA um 29,2% und bei OpenCL um 30,2% (Tesla C2050, N=8000, M=8000).

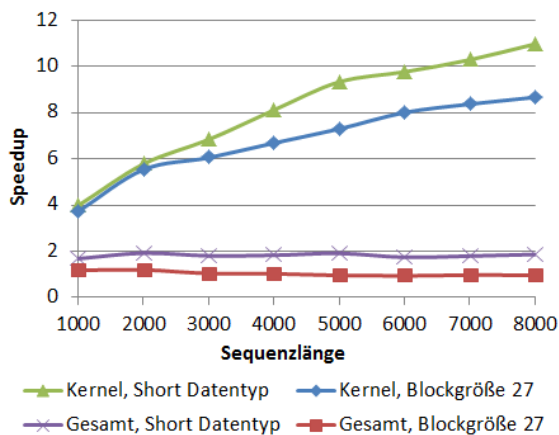


Abbildung 66: CUDA, GTX 480

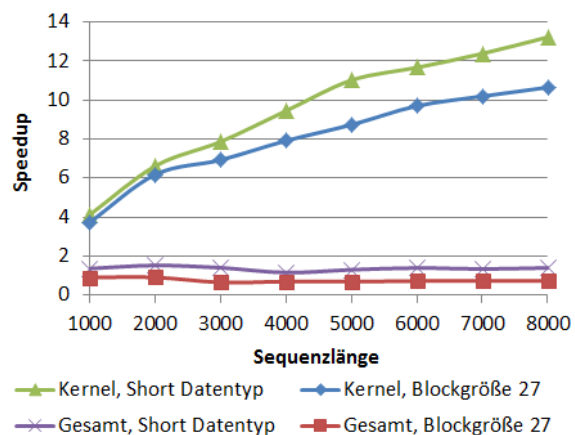


Abbildung 67: OpenCL, GTX 480

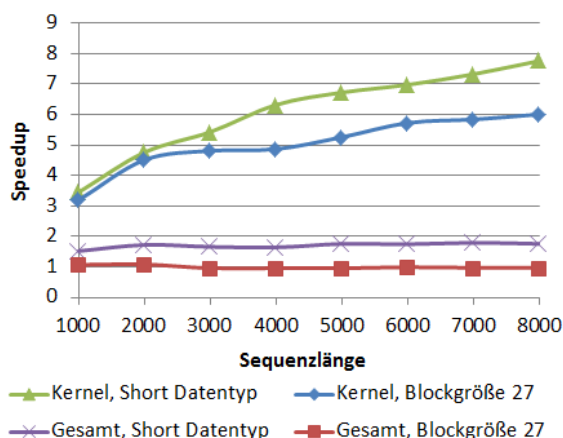


Abbildung 68: CUDA, Tesla C2050

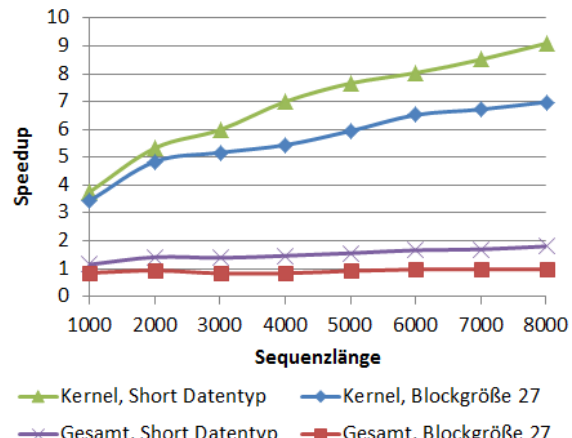


Abbildung 69: OpenCL, Tesla C2050

Bei der GTX 480 ist mit den Optimierungen „Optimale Blockgröße“ und „Verwendung von Short anstatt von Int“ der Gesamtspeedup des CUDA-Programms um 31,9% höher als der Gesamtspeedup des OpenCL-Programms (GTX 480, N=8000, M=8000). Im Gegensatz dazu ist der Kernelspeedup des OpenCL-Programms um 20,4% höher als der Kernelspeedup des CUDA-Programms (GTX 480, N=8000, M=8000). Die Gründe dafür werden bei der ersten Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darüber erklärt.

Bei der Tesla C2050 ist mit den Optimierungen „Optimale Blockgröße“ und „Verwendung von Short anstatt von Int“ der Gesamtspeedup des OpenCL-Programms um 2,5% höher als der Gesamtspeedup des CUDA-Programms (N=8000, M=8000). Der Grund für den niedrigeren Gesamtspeedup-Unterschied zwischen CUDA- und OpenCL-Programm im Vergleich zur GTX 480 wird bei der ersten Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darüber erklärt. Der Unterschied zwischen der niedrigsten und höchsten gemessenen Gesamtlaufzeit bei 10 Durchgängen liegt bei

CUDA bei 12,2 % und bei OpenCL bei 18,9% (Tesla C2050, N=8000, M=8000). Da der Gesamtspeedup-Unterschied zwischen CUDA- und OpenCL-Programm unter der Laufzeitschwankung liegt, kann das CUDA- und OpenCL-Programm als gleich schnell betrachtet werden. Der Kernelspeedup des OpenCL-Programms ist um 17,2% höher als der Kernelspeedup des CUDA-Programms (Tesla C2050, N=8000, M=8000). Der Unterschied zwischen der niedrigsten und höchsten gemessenen Kernellaufzeit bei 10 Durchgängen liegt bei CUDA bei 0,3% und bei OpenCL bei 0,8% (Tesla C2050, N=8000, M=8000).

Optimierung „Nur eine Inputmatrix und eine shared Teilmatrix“

Die Optimierung „Nur eine Inputmatrix und eine shared Teilmatrix“ wirkt sich bei CUDA und OpenCL positiv aus (siehe Abbildung 70, Abbildung 71, Abbildung 72 und Abbildung 73). Die Blockgröße konnte durch die Optimierung bei der GTX 480 und der Tesla C2050 von 38 auf 54 erhöht werden. Bei N=8000, M=8000 und der GTX 480 verbessert sich der Gesamt-Speedup bei CUDA um 52,9% und bei OpenCL um 46,3%. Der Kernel-Speedup verbessert sich bei CUDA um 30,2% und bei OpenCL um 30% (GTX 480, N=8000, M=8000). Bei N=8000, M=8000 und der Tesla C2050 verbessert sich der Gesamtspeedup bei CUDA um 47,5% und bei OpenCL um 41%. Der Kernelspeedup verbessert sich bei CUDA um 27,1% und bei OpenCL um 25,5% (Tesla C2050, N=8000, M=8000).

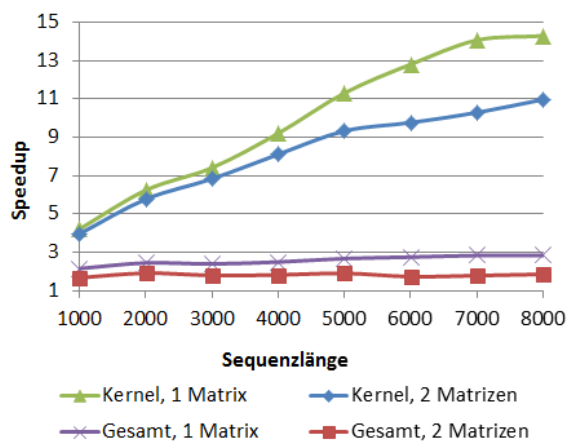


Abbildung 70: CUDA, GTX 480

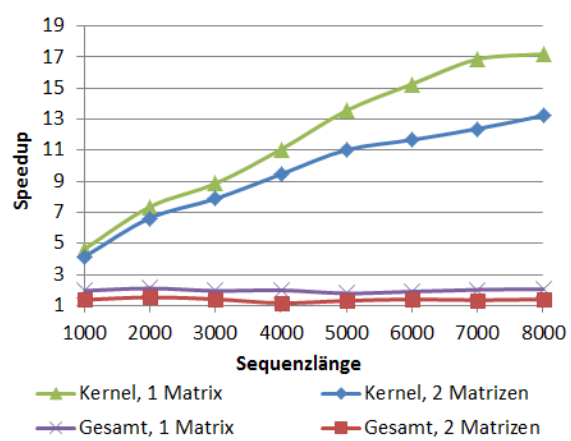


Abbildung 71: OpenCL, GTX 480

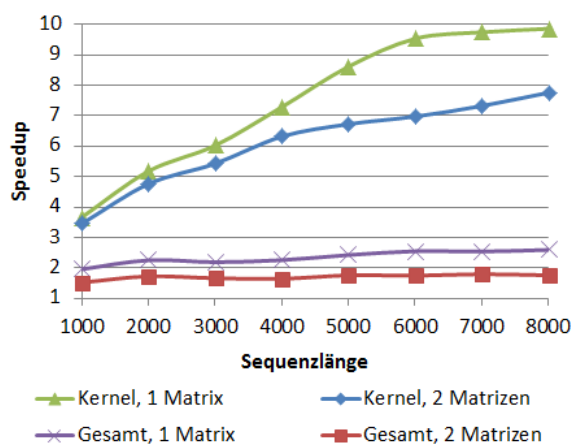


Abbildung 72: CUDA, Tesla C2050

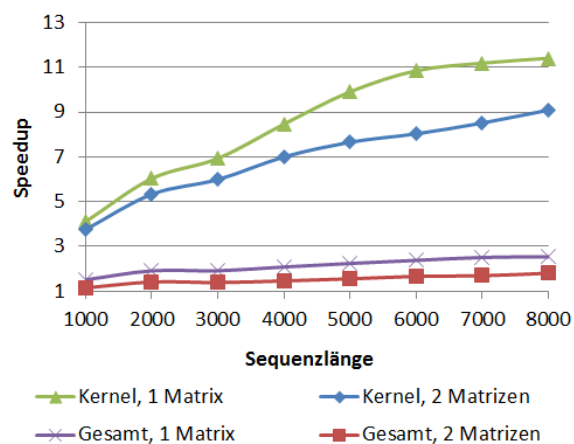


Abbildung 73: OpenCL, Tesla C2050

Bei der GTX 480 ist mit den Optimierungen „Optimale Blockgröße“, „Verwendung von Short anstatt von Int“ und „Nur eine Inputmatrix und eine shared Teilmatrix“ der Gesamtspeedup des CUDA-Programms um 37,9% höher als der Gesamtspeedup des OpenCL-Programms (GTX 480, N=8000, M=8000). Im Gegensatz dazu ist der Kernelspeedup des OpenCL-Programms um 20,3% höher als der Kernelspeedup des CUDA-Programms (GTX 480, N=8000, M=8000). Die Gründe dafür werden bei der ersten Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darüber erklärt.

Bei der Tesla C2050 ist mit den Optimierungen „Optimale Blockgröße“, „Verwendung von Short anstatt von Int“ und „Nur eine Inputmatrix und eine shared Teilmatrix“ der Gesamtspeedup des CUDA-Programms um 2% höher als der Gesamtspeedup des OpenCL-Programms (N=8000, M=8000). Im Gegensatz dazu ist der Kernelspeedup des OpenCL-Programms um 15,7% höher als der Kernelspeedup des CUDA-Programms (Tesla C2050, N=8000, M=8000). Der Grund für den niedrigeren Gesamtspeedup-Unterschied zwischen CUDA- und OpenCL-Programm im Vergleich zur GTX 480 wird bei der ersten Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darüber erklärt.

Optimierung „Initialisierung der Inputmatrix auf GPU anstatt Übertragung“

Die Optimierung „Initialisierung der Inputmatrix auf GPU anstatt Übertragung“ wirkt sich bei CUDA und OpenCL positiv aus (siehe Abbildung 74, Abbildung 75, Abbildung 76 und Abbildung 77). Bei N=8000, M=8000 und der GTX 480 verbessert sich der Gesamt-Speedup bei CUDA um 35,8% und bei OpenCL um 42%. Bei N=8000, M=8000 und der Tesla C2050 verbessert sich der Gesamt-Speedup bei CUDA um 29,2% und bei OpenCL um 28,1%. Da die Optimierung nur Einfluss auf den Gesamtspeedup hat, verändert sich der Kernel-Speedup nur gering (bei N=8000, M=8000: 1% (CUDA, GTX480), 1% (OpenCL, GTX 480), 0,2% (CUDA, Tesla C2050), 0,5% (OpenCL, Tesla C2050)). Wenn bei der GTX 480 die Gesamtlaufzeit ohne Kernellaufzeit betrachtet wird, ergibt sich eine Verbesserung bei CUDA um 47,9% und bei OpenCL um 50% (N=8000, M=8000). Wenn bei der Tesla C2050 die Gesamtlaufzeit ohne Kernellaufzeit betrachtet wird, ergibt sich eine Verbesserung bei CUDA um 43,4% und bei OpenCL um 38,6% (N=8000, M=8000). Die Schwankung der Gesamtlaufzeit bei 10 Laufzeitmessungen ist bei der Tesla C2050 deutlich größer als bei der GTX 480 (bei N=8000, M=8000: 15,3% (OpenCL, Tesla C2050), 1,1% (OpenCL, GTX 480), 5,2% (CUDA, Tesla C2050), 0,1% (CUDA, GTX 480)). Im Gegensatz zur Gesamtlaufzeit gibt es bei der Kernellaufzeit bei 10 Laufzeitmessungen eine sehr geringe Schwankung (bei N=8000, M=8000: 0,6% (OpenCL, Tesla C2050), 0,5% (OpenCL, GTX 480), 0,2% (CUDA, Tesla C2050), 0,1% (CUDA, GTX 480)).

Bei der Tesla C2050 hat man bei CUDA und OpenCL einen gleichmäßigen Anstieg des Gesamt-Speedups über die Problemgrößen (siehe Abbildung 76 und Abbildung 77). Bei der GTX 480 hat man nur bei CUDA einen gleichmäßigen Anstieg des Gesamtspeedups über die Problemgrößen (siehe Abbildung 74). Bei der GTX 480 und OpenCL (siehe Abbildung 75) gibt es einen Sprung nach unten zwischen N=4000, M=4000 und N=5000, M=5000. Dieser Sprung wird durch die Laufzeit für die Initialisierung der Inputmatrix auf der GPU und vor der Optimierung durch die Laufzeit für das Übertragen der Startmatrix von Host zu Device verursacht. Ein weiterer Grund für die geringe Steigerung des Gesamtspeedups über die Problemgrößen bei OpenCL und GTX 480 ist die Laufzeit für das Zurückübertragen der Ergebnismatrix von Device zu Host, da bei GTX 480 und OpenCL die PCI-Bus-Bandbreite deutlich niedriger ist als bei der GTX 480 und CUDA (siehe erste Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darüber). Während der Unterschied zwischen CUDA und OpenCL bei der PCI-Bus-Bandbreite bei Übertragungen von Device zu Host ab N=2000, M=2000 fast gleich bleibt, ist der Unterschied zwischen CUDA und OpenCL bei der PCI-Bus-Bandbreite bei Übertragungen von Host zu Device unterschiedlich über die Problemgrößen.

Bei der GTX 480 ist mit allen Optimierungen („Optimale Blockgröße“, „Verwendung von Short anstatt von Int“, „Nur eine Inputmatrix und eine shared Teilmatrix“ und „Initialisierung der Inputmatrix auf GPU anstatt Übertragung“) der Gesamtspeedup des CUDA-Programms um 31,8% höher als der Gesamtspeedup des OpenCL-Programms (N=8000, M=8000). Im Gegensatz dazu ist der Kernelspeedup des OpenCL-Programms um 20,3% höher als der Kernelspeedup des CUDA-Programms (GTX 480, N=8000, M=8000). Die Gründe dafür werden bei der ersten Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darüber erklärt.

Bei der Tesla C2050 ist mit allen Optimierungen („Optimale Blockgröße“, „Verwendung von Short anstatt von Int“, „Nur eine Inputmatrix und eine shared Teilmatrix“ und „Initialisierung der Inputmatrix auf GPU anstatt Übertragung“) der Gesamtspeedup des CUDA-Programms um 2,9% höher als der Gesamtspeedup des OpenCL-Programms (N=8000, M=8000). Im Gegensatz dazu ist der Kernelspeedup des OpenCL-Programms um 16% höher als der Kernelspeedup des CUDA-Programms (Tesla C2050, N=8000, M=8000). Der Grund für den deutlich niedrigeren Gesamtspeedup-Unterschied zwischen CUDA- und OpenCL-Programm im Vergleich zur GTX 480 wird bei der ersten Optimierung „Wahl der optimalen Blockgröße“ ein paar Absätze darüber erklärt.

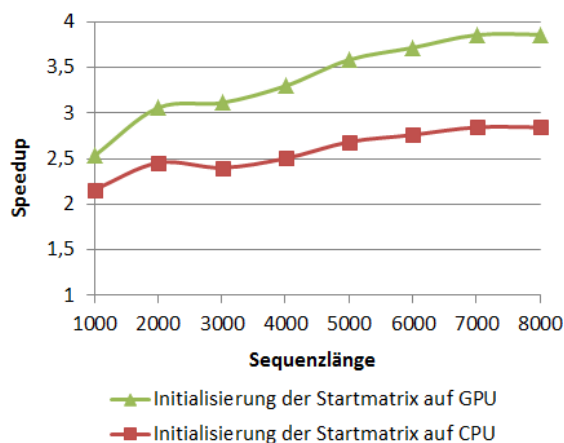


Abbildung 74: CUDA, GTX 480, Gesamtspeedup

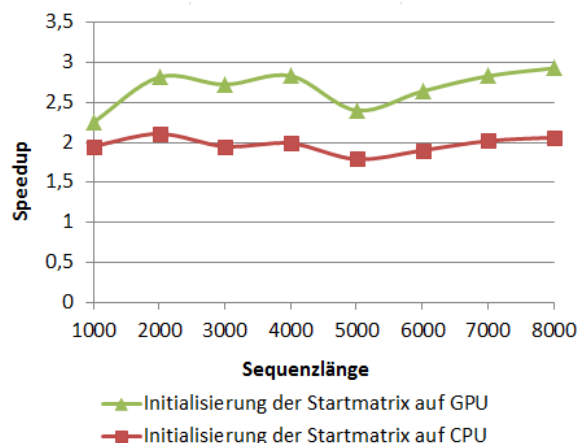


Abbildung 75: OpenCL, GTX 480, Gesamtspeedup

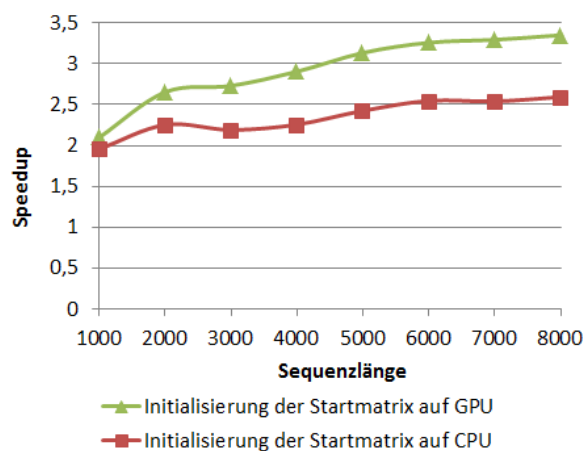


Abbildung 76: CUDA, Tesla C2050, Gesamtspeedup

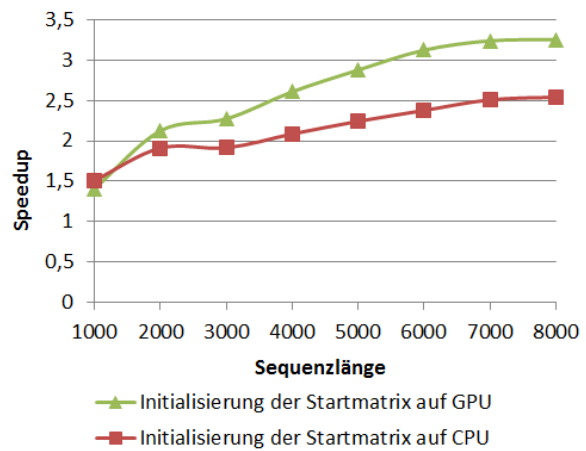


Abbildung 77: OpenCL, Tesla C2050, Gesamtspeedup

Nicht erfolgreich: „Pinned Memory statt Pageable Memory“

„Pinned Memory statt Pageable Memory“ führt bei der GTX 480 sowohl in OpenCL als auch in CUDA zu einer Erhöhung der Laufzeit und sollte deswegen nicht angewendet werden. Der Aufwand des Erzeugens und des Freigebens von Pinned Memory ist bei CUDA und bei OpenCL so aufwendig (Vergleich von Abbildung 78 und Abbildung 79 mit Abbildung 80 und Abbildung 81), dass der reduzierte Zeitaufwand für die Übertragung übertroffen wird. Wenn man alle Teilzeiten bei N=8000, M=8000 (NW-Ausgangscode mit optimaler Blockgröße 27) zusammenrechnet, hat man bei OpenCL 0,41 Sekunden (Pinned Memory) versus 0,34 Sekunden (Pageable Memory). Bei CUDA ist Pinned Memory mit 0,66 Sekunden (Pinned Memory) versus 0,26 Sekunden (Pageable Memory) deutlich schlechter.

Der große Laufzeitunterschied zwischen CUDA und OpenCL bei der Übertragung zwischen Host und Device bei der GTX 480 (siehe Abbildung 78 und Abbildung 79) kann durch Pinned Memory (siehe Abbildung 80 und Abbildung 81) deutlich reduziert werden. Bei der Ausführung des NW-Ausgangscodes mit optimaler Blockgröße 27 auf der GTX 480 kann der Laufzeitunterschied zwischen CUDA und OpenCL bei N=8000, M=8000 mit Pinned Memory im Vergleich zu Pageable Memory folgendermaßen verbessert werden. Bei der Übertragung vom Host zum Device verbessert sich der Laufzeitunterschied zwischen CUDA und OpenCL von 0,08 Sekunden (CUDA: 0,11 Sekunden, OpenCL: 0,19 Sekunden) auf 0,03 Sekunden (CUDA: 0,09 Sekunden, OpenCL: 0,12 Sekunden). Bei der Übertragung vom Device zum Host verbessert sich der Laufzeitunterschied zwischen CUDA und OpenCL von 0,03 Sekunden (CUDA: 0,07 Sekunden, OpenCL: 0,1 Sekunden) auf 0,0000652 Sekunden (CUDA: 0,0713751 Sekunden, OpenCL: 0,0714403 Sekunden).

In Abbildung 78 (CUDA) und Abbildung 79 (OpenCL) sind die Teilzeiten des Shared-Memory-Programms, das Pageable Arrays verwendet, dargestellt. Als Optimierung wird nur die Wahl der optimalen Blockgröße verwendet.

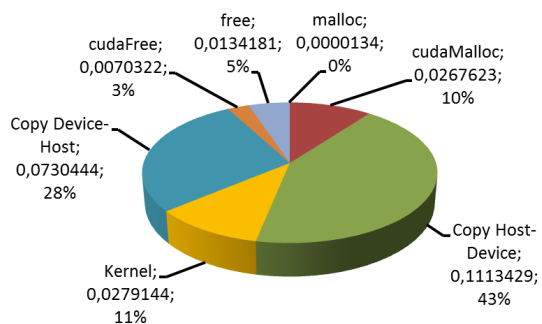


Abbildung 78: CUDA, GTX 480, N=8000, M=8000

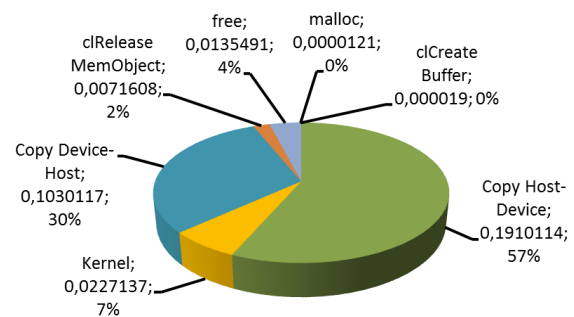


Abbildung 79: OpenCL, GTX 480, N=8000, M=8000

In Abbildung 80 (CUDA) und Abbildung 81 (OpenCL) sind die Teilzeiten des Shared-Memory-Programms, das Pinned Arrays verwendet, dargestellt. Als Optimierung wird nur die Wahl der optimalen Blockgröße verwendet. Die verwendeten Befehle bei OpenCL für das Anlegen des Speichers am Host (Malloc Host Pinned in der Abbildung 81) und für das Freigeben des Speichers am Host (Free Host Pinned in der Abbildung 81) stehen im Code 20 im Kapitel 3.3.3.

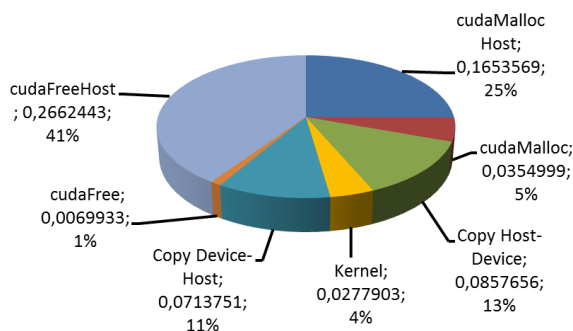


Abbildung 80: CUDA, GTX 480, N=8000, M=8000

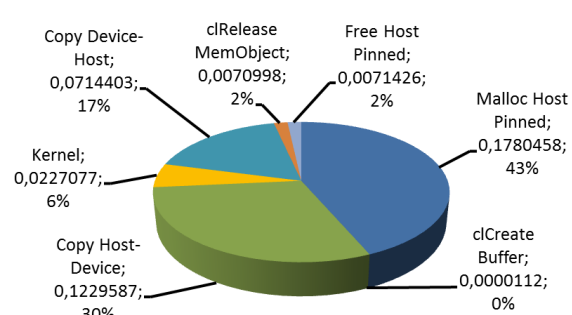


Abbildung 81: OpenCL, GTX 480, N=8000, M=8000

7.4.2 Fermi im Vergleich zu der Vorgängerarchitektur GT200

In diesem Kapitel wird für die Fermi-Architektur die GTX 480 und für die Vorgängerarchitektur GT200 die GTX 285 herangezogen. Außerdem wird das CUDA-Programm und die Problemgröße $N=8000$, $M=8000$ verwendet.

Durch 3 Mal mehr Shared Memory pro Multiprozessor bei Fermi (48 KB statt 16 KB) erreicht man beim Shared-Memory-Programm mit allen Optimierungen 43% mehr Gesamtspeedup und 125% mehr Kernelspeedup im Vergleich zu GT200 (siehe Abbildung 82 und Abbildung 83). Die Blockgröße kann bei Fermi durch den vergrößerten Shared Memory erhöht werden. Bei Fermi wird die Blockgröße 54 und bei GT200 die Blockgröße 31 benützt.

Beim Global-Memory-Programm mit allen Optimierungen führt der Global-Memory-Cache bei Fermi zu 15% mehr Gesamtspeedup und zu 36% mehr Kernelspeedup im Vergleich zu GT200 (siehe Abbildung 82 und Abbildung 83).

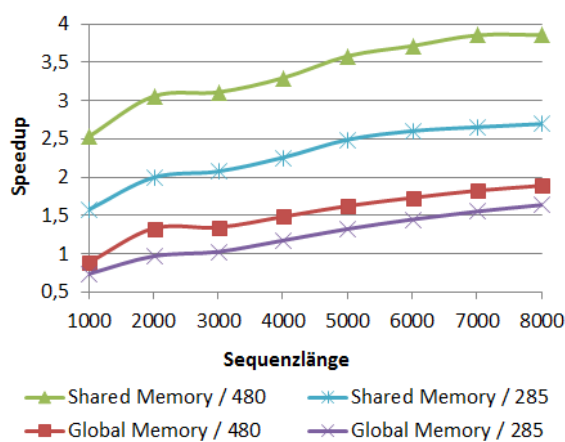


Abbildung 82: Gesamt-Speedup, CUDA

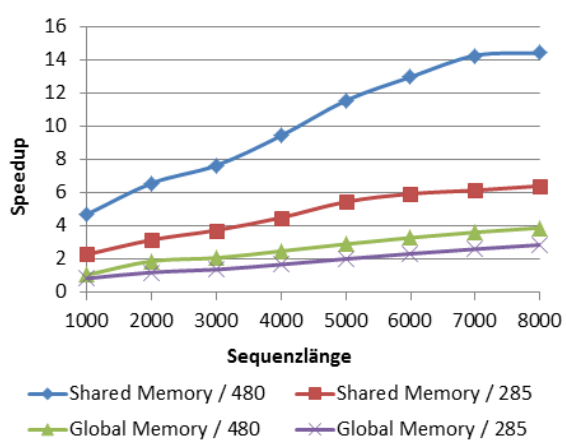


Abbildung 83: Kernel-Speedup, CUDA

Die Unterschiede zwischen Fermi und GT200 bei der Wirksamkeit der einzelnen Optimierungen kann man durch das Vergleichen der Abbildung 84 mit der Abbildung 85 erkennen. Die zwei Diagramme (Abbildung 84 und Abbildung 85) sind identisch mit den Diagrammen (Abbildung 39 und Abbildung 41) im Kapitel 7.3.4. Die Gründe für die Unterschiede sind, dass mit Fermi eine Cache-Hierarchie für den Global Memory eingeführt wurde und die Größe des Shared Memory pro Multiprozessor von 16 KB auf 48 KB erhöht wurde.

Beim Global-Memory-Programm verbessern sich die Gesamtspeedups folgendermaßen. Durch die Optimierung „Verwendung von short-Datentyp anstatt des int-Datentyps“ gibt es eine Verbesserung bei Fermi um 41,3% und bei GT200 um 12,6%. Die Optimierung Antidiagonale-Speicherung/Memory Coalescing verbessert den Speedup bei Fermi um 48,6% und bei GT200 um 241%.

Beim Shared-Memory-Programm verbessern sich die Gesamtspeedups folgendermaßen. Durch die Optimierung „Optimale Blockgröße“ gibt es eine Verbesserung bei Fermi um 12% und bei GT200 um 9%. Die Optimierung „Verwendung von short-Datentyp anstatt des int-Datentyps“ verbessert den Speedup bei Fermi um 93% und bei GT200 um 76%. Durch die Optimierung „Nur eine Inputmatrix und eine shared Teilmatrix“ gibt es eine Erhöhung des Speedups bei Fermi um 53% und bei GT200 um 30%. Die Optimierung „Initialisierung der Inputmatrix auf GPU anstatt Übertragung“ verbessert den Speedup bei Fermi um 36% und bei GT200 um 15%.

Wenn man den Gesamtspeedup des Global-Memory-Programms und des Shared-Memory-Programms nur mit der Optimierung „Optimale Blockgröße“ vergleicht, ist das Shared-Memory-Programm bei Fermi um 7% und bei GT200 um 141% schneller.

Beim Vergleichen der Gesamtspeedups zwischen Fermi und GT200 bei den einzelnen Optimierungen des Global-Memory-Programms erhält man folgende Prozentzahlen. Beim Basis-Code ohne Optimierungen ist Fermi um 111% schneller als GT200. Mit der Optimierung „Verwendung von short-Datentyp anstatt des int-Datentyps“ ist Fermi um 165% schneller als GT200. Mit der weiteren Optimierung Antidiagonale-Speicherung/Memory Coalescing ist Fermi nur mehr 15,4% schneller als GT200.

Beim Vergleichen der Gesamtspeedups zwischen Fermi und GT200 bei den einzelnen Optimierungen des Shared-Memory-Programms erhält man folgende Prozentzahlen. Beim Ausgangscode ohne Optimierungen ist GT200 um 9,5% schneller als Fermi. Mit der Optimierung „Optimale Blockgröße“ ist GT200 um 6,4% schneller als Fermi. Mit der weiteren Optimierung „Verwendung von short-Datentyp anstatt des int-Datentyps“ ist Fermi um 3,1% schneller als GT200. Bei der Optimierung „Nur eine Inputmatrix und eine shared Teilmatrix“ ist Fermi um 21,4% schneller als GT200. Bei der letzten Optimierung „Initialisierung der Inputmatrix auf GPU anstatt Übertragung“ ist Fermi um 43% schneller als GT200.

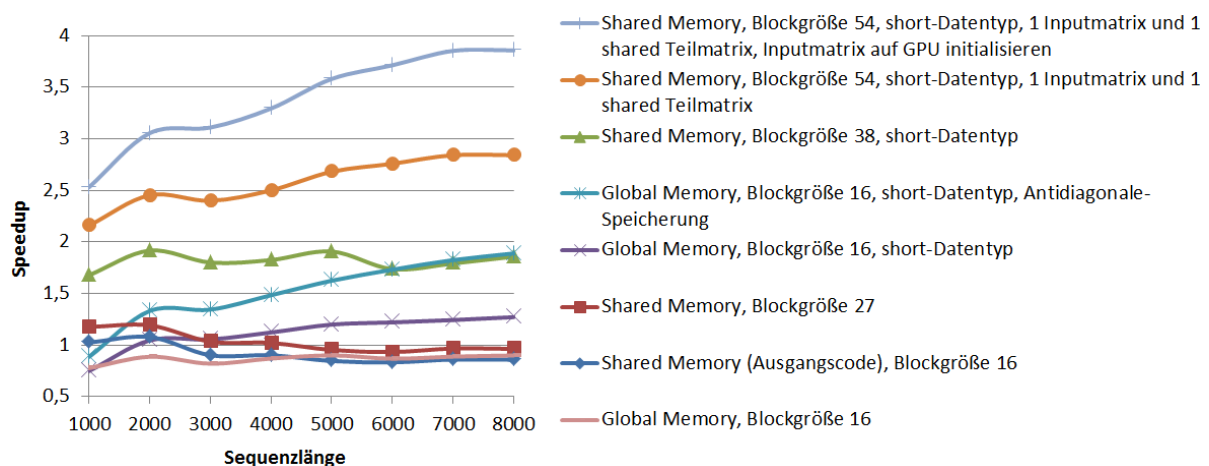


Abbildung 84: Fermi-GPU GTX 480, CUDA, Gesamtspeedup

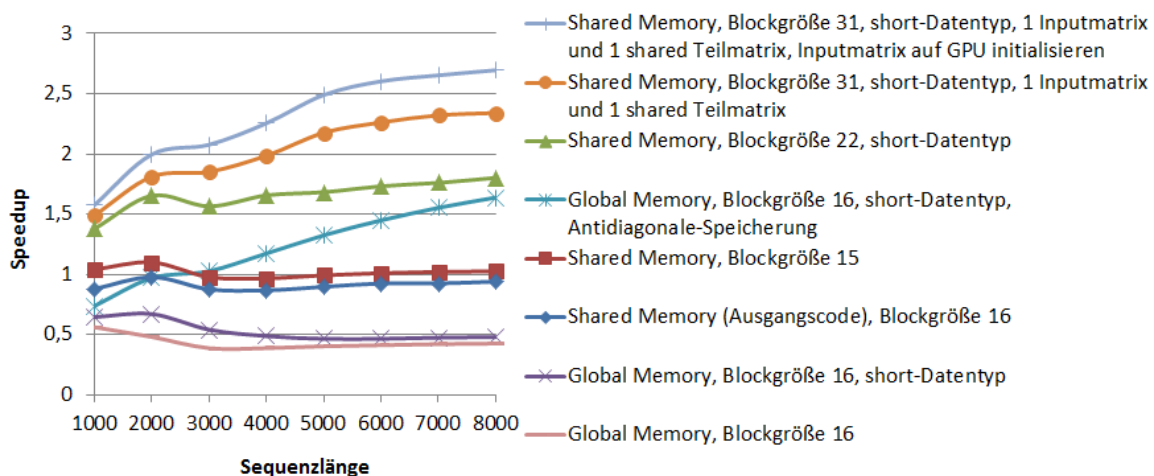


Abbildung 85: GT200-GPU GTX 285, CUDA, Gesamtspeedup

7.4.3 Wie stark konnte der Ausgangscode verbessert werden?

Wenn man die Speedup-Werte des CUDA-Shared-Memory-Programms bei der Problemgröße $N=8000$ und $M=8000$ bei der GTX 480 betrachtet, konnte der Kernel-Speedup von 4,04 auf 14,44 und der Gesamt-Speedup von 0,86 auf 3,86 erhöht werden (siehe Abbildung 86).

Wenn man die Speedup-Werte des CUDA-Global-Memory-Programms bei der Problemgröße $N=8000$ und $M=8000$ bei der GTX 480 betrachtet, konnte der Kernel-Speedup von 1,89 auf 3,9 und der Gesamt-Speedup von 0,9 auf 1,89 erhöht werden (siehe Abbildung 87).

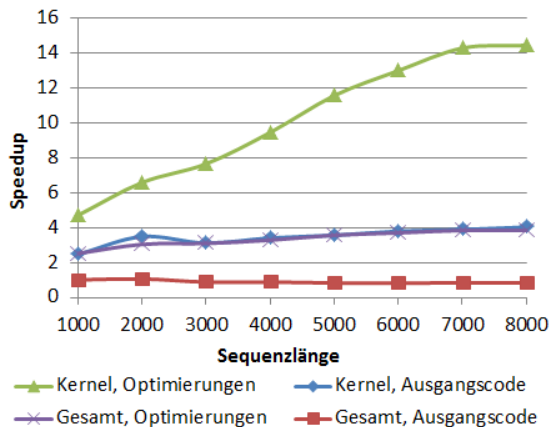


Abbildung 86: Shared-Memory-Programm, CUDA, GTX 480

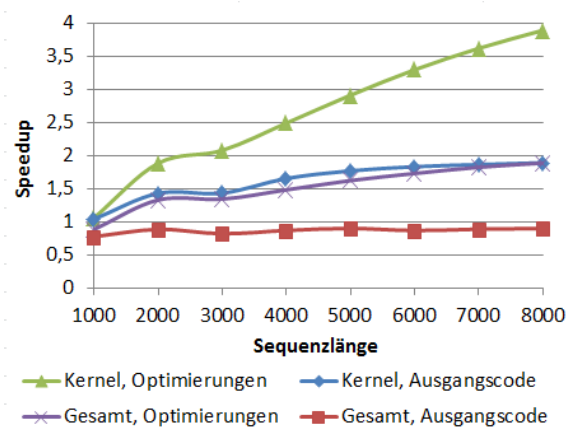


Abbildung 87: Global-Memory-Programm, CUDA, GTX 480

7.4.4 Wie viel wurde von der maximalen Leistung der GPU erreicht?

Die Peak-Leistung der GTX 480 beträgt 1345 GFLOP/s [5] [21]. Auf die 1345 GFLOP/s kommt man durch folgende Berechnung: 480 Kerne mit 1,401 GHz Taktfrequenz, 1 FMA(=1 Multiplikation und 1 Addition)-Operation pro Takt = $480 \cdot 1,401 \cdot 2 = 1345$ GFLOP/s [5] [21].

Da bei dem NW-Algorithmus (3 Additionen und 0 Multiplikationen) nur der Additionsteil der FMA-Operation verwendet wird, ergibt sich bei der GTX480 eine Peak-Leistung von 672,5 GOP/s (=Giga Operations per Second).

Wenn man das auf der GTX 480 ausgeführte Shared-Memory-Programm mit allen Optimierungen bei $N=8000$ und $M=8000$ betrachtet, erreicht man mit der GPU-Gesamtlaufzeit 0,4% und mit der GPU-Kernellaufzeit 1,7% der maximalen theoretischen Leistung der GPU (siehe Abbildung 88). Die Ursache für die geringe Leistung des Kernels beim Shared-Memory-Programm sind die Wavefronts und die geringe Anzahl von Operationen/Additionen pro Global-Memory-Zugriff. Die Ursache für die noch geringere Leistung der Gesamtlaufzeit von 0,4% ist die stark limitierende PCI-Bus-Bandbreite. Die Übertragung der Ergebnismatrix vom Arbeitsspeicher der GPU zum Arbeitsspeicher der CPU müsste deutlich schneller funktionieren.

Bei dem auf der GTX 480 ausgeführten Global-Memory-Programm mit allen Optimierungen ist die erreichte Effizienz noch geringer (siehe Abbildung 88).

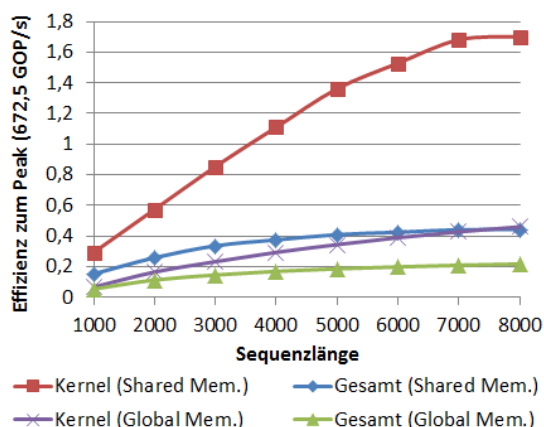


Abbildung 88: Erreichte Effizienz bei der Leistung (CUDA, GTX 480)

8 Algorithmus DCS (=Direct Coulomb Summation)

8.1 Einleitung

Ein für die GPU deutlich besser geeigneter Algorithmus ist der DCS(=Direct Coulomb Summation)-Algorithmus, da er wesentlich besser als der NW/SW (=Needleman-Wunsch/Smith-Waterman) parallelisiert werden kann. DCS verwendet Constant Memory, benützt Floating-Point-Units und der Speedup liegt zwischen 10 und 45 (siehe Abbildung 109 im Kapitel 8.3.2.5). Es gibt keine Wavefronts und es gibt im Vergleich zu Needleman-Wunsch deutlich mehr Operationen pro Speicherzugriff. Dieser Algorithmus ist so gut für GPUs geeignet, dass man der Maximalleistung der GPU sehr nahe kommt [1].

Das Anwendungsgebiet von DCS ist VMD (=Visual Molecular Dynamics) [1]. VMD ist ein Teilgebiet der „Life Sciences“-Algorithmen [67]. Mit dem DCS-Algorithmus werden elektrostatische Potential Maps berechnet [1]. Das Ergebnis der Berechnung wird oft für das Platzieren von Ionen (=Ion Placement) in eine Struktur (z. B. Proteinstruktur) verwendet, um anschließend „molecular dynamics“-Simulationen durchführen zu können [1]. Mit Hilfe der Ionen wird die Struktur stabil gemacht [1].

Für den DCS-Algorithmus wird als Literatur- und Programmcode-Quelle das Kapitel 9 (CUDA-Version) und das Kapitel 11 (OpenCL-Version) vom Buch „Programming Massively Parallel Processors“ [1] herangezogen. Für die GPU-Optimierungen wurden die Ideen und Codes aus dem Buch entnommen.

Als Input eines DCS hat man ein Molekül mit Atomen [1]. Jedes Atom des Moleküls hat eine bestimmte Ladung (float-Wert) und eine Koordinate (x, y, z) [1]. Man hat also 4 Werte pro Atom [1]. Der Kernel des Algorithmus ist sehr einfach. Um ein Molekül wird ein Gitternetz gelegt [1]. Nun kann das elektrostatische Potential jedes Gitterpunkts unabhängig von den anderen Gitterpunkten berechnet werden [1]. Jeder Gitterpunkt muss in der Berechnung die Entfernung zu allen Atomen des Moleküls und die Ladung jedes Atoms berücksichtigen [1]. Das Ergebnis sind die berechneten Gitterpunkte [1].

In der Implementierung im Buch und in der Masterarbeit werden die Laufzeitmessung und die Parallelisierung auf der Ebene eines 2D-DCS durchgeführt. Damit werden alle Gitterpunkte einer Ebene der Z-Achse berechnet. Für die Berechnung des 3D-DCS, also des kompletten dreidimensionalen Grids, muss die Anzahl der Z-Ebenen einfach von 1 auf die benötigte Anzahl gesetzt werden. Da die Berechnung der einzelnen Z-Ebenen unabhängig voneinander ist, kann die Berechnung aller Z-Ebenen parallel stattfinden, zum Beispiel auf unterschiedlichen GPUs.

Die Gridpunkte befinden sich in einem „uniform grid“ [1]. Das bedeutet, die Abstände zu den Nachbargridpunkten sind in allen drei Dimensionen immer gleich groß [1]. Dieser Abstand ist im Code in der Variable „gridspacing“ gespeichert.

Dem Programm werden die Anzahl von Atomen, die Anzahl von Gitterpunkten in X Richtung, die Anzahl von Gitterpunkten in Y Richtung, die Anzahl von Gitterpunkten in Z Richtung (=bei allen Laufzeitmessungen gleich 1) und der Abstand zwischen den Gitterpunkten (=bei allen Laufzeitmessungen gleich 0.005) übergeben. Wenn das Programm jedoch in der Praxis verwendet wird, muss aus den Atomen jeweils das Minimum und das Maximum von allen drei Richtungen (X, Y, Z) bestimmt werden. Dann kann darauf basierend berechnet werden, wie groß die gridspacing-Variable sein muss, um die Vorgaben der Anzahl der Gridpunkte in alle Richtungen zu erfüllen. Für die Messung der Laufzeit ist das aber nicht notwendig und somit wurde darauf verzichtet.

Beim DCS-Algorithmus wurde eine Implementierung, die auf die 8800 GTX optimiert ist [1], als Basis herangezogen. Beim CPU-Programm wurde die selbst entwickelte Idee umgesetzt, bei der 4 statt 1 Atomvektor (x, y, z und w) verwendet werden. Dadurch wurde für das CPU-Programm ein Speedup von 1,78 erreicht (N=800, M=800, siehe Abbildung 89 im Kapitel 8.2.1.1). Außerdem wurde auf Basis des sequentiellen Programms ein OpenMP-Code erzeugt. Beim GPU-Programm wurde fast alles (OpenCL und CUDA) vom Buch „Programming Massively Parallel Processors“ [1] übernommen. Der komplette Programmcode steht jedoch nicht im Buch. Oft ist nur der Code für den Kernel vorhanden und der dafür notwendige Hostteil muss selbst hinzugefügt werden. Es wurde eine Überprüfung der Korrektheit hinzugefügt. Alle Ideen zu den Optimierungen stammen vom Buch [1].

8.2 CPU-Programm mit Laufzeitmessungen

Bei den Laufzeitmessungen in den Unterkapiteln beträgt die Anzahl der Atome immer 1000, gridspacing immer 0.005 und die Anzahl der z-Ebenen immer 1. Als CPU wurde die Intel Xeon X5550 verwendet und für alle Diagramme wurde die gemessene Gesamtlaufzeit benützt.

8.2.1 C

8.2.1.1 Drei Codevarianten und deren Laufzeit

Die Funktion besteht aus vier for-Schleifen (siehe Code 51) [1]. Mit den ersten drei Schleifen iteriert man über die einzelnen Gridpunkte im dreidimensionalen Grid [1]. Mit der letzten for-Schleife berechnet man das elektrostatische Potential eines Gridpunkts, indem die Entfernung vom Gridpunkt zu allen Atomen und die Ladung der einzelnen Atome berücksichtigt wird [1]. Jedes Atom wird durch vier nebeneinanderstehende Werte im atoms[] - Array repräsentiert [1]. Die x-, y- und z-Koordinate des Atoms sind die ersten drei Werte [1]. Danach folgt noch die Ladung des Atoms [1].

Beim Datentyp float (=Single-Precision) wird für die Berechnung der Quadratwurzel die Funktion „sqrtf()“ verwendet. Sollte der Datentyp double (=Double-Precision) verwendet werden, muss die Funktion „sqrt()“ stattdessen benützt werden.

Damit der Speedup des GPU-Programms gegenüber dem CPU-Programm fair ist, ist es wichtig, dass auch das CPU-Programm optimiert wird. Dazu können einerseits Änderungen am Code vorgenommen werden und andererseits beim Kompilieren Optimierungsflags und Compiler für mehr Performance ausprobiert werden.

Es gibt drei Codevarianten für den DCS-CPU-Code. Ein Code ist das Standard C-Programm des DCS vom Buch [1]. Die anderen zwei Varianten zeigen, dass der Standard-Code deutlich verbessert werden kann (siehe Abbildung 89). Die zweite Variante stammt von [68] und die dritte Variante ist die Erweiterung der Variante 2 um eine Speicherzugriffsoptimierung.

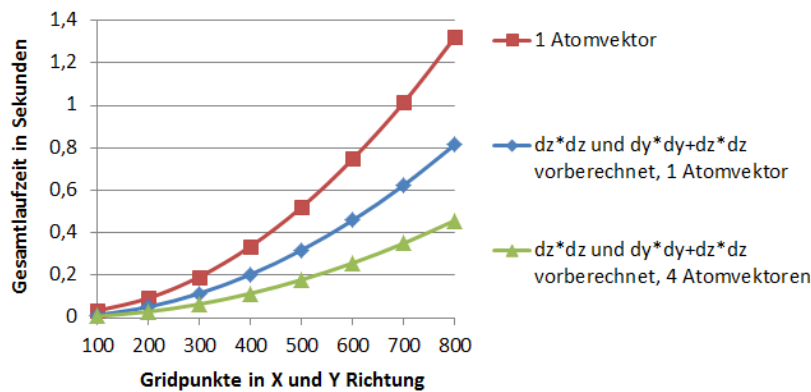


Abbildung 89: Gesamtlaufzeit der drei Codevarianten des DCS-CPU-Codes

Erste Variante

Die erste Variante (siehe Code 51 und rote Kurve in Abbildung 89) steht im Buch [1] und ist die langsamste Variante.

Code 51: Erste Variante des CPU-C-Codes [1] (adaptiert)

```
void dcs(Datentyp* energygrid,int gridx,int gridy,int gridz,Datentyp gridspacing,
const Datentyp *atoms,int numatoms)
{
    int i,j,k,n;
    int atomarrdim=numatoms*4;

    for(k=0;k<gridz;k++){
        Datentyp z=gridspacing*(Datentyp)k;

        for(j=0;j<gridy;j++){
            Datentyp y=gridspacing*(Datentyp)j;

            for(i=0;i<gridx;i++){
                Datentyp x=gridspacing*(Datentyp)i;
                Datentyp energy=0.0f;

                for(n=0;n<atomarrdim;n+=4){
                    Datentyp dx=x-atoms[n];
                    Datentyp dy=y-atoms[n+1];
                    Datentyp dz=z-atoms[n+2];

                    energy+=atoms[n+3]/sqrtf(dx*dx+dy*dy+dz*dz);
                    //energy+=atoms[n+3]/sqrt(dx*dx+dy*dy+dz*dz);
                }
                energygrid[k*gridx*gridy+j*gridx+i]=energy;
            }
        }
    }
}
```

Zweite Variante

Die zweite Variante (siehe Code 52 und blaue Kurve in Abbildung 89) ist von [68] und erreicht eine Performance-Verbesserung, indem die Berechnung „ $dy*dy+dz*dz$ “ für alle Atome von der innersten Schleife in die äußeren Schleifen gezogen wird [68]. Die Berechnung „ $dz*dz$ “ kann in der obersten Schleife durchgeführt werden, da dort bereits alle Inputdaten für die Berechnung vorhanden sind. In der darunterliegenden j-Schleife kann die Berechnung „ $dy*dy$ “ durchgeführt werden. Weiters kann in der j-Schleife das Ergebnis von „ $dy*dy$ “ mit dem Ergebnis von „ $dz*dz$ “ von der darüber liegenden Schleife addiert werden. Redundante Berechnungen werden somit verhindert, indem sie in die darüber liegenden for-Schleifen gezogen werden [68]. Der Nachteil ist, dass diese Variante durch die Speicherung der Vorberechnungen in Arrays zusätzlichen Speicherplatz benötigt [68]. Bei 1000

Atomen und dem Datentyp Float werden zusätzlich 8000 Bytes (=2 Arrays*1000 Atome*4 Bytes) an Speicherplatz benötigt.

Code 52: Zweite Variante des CPU-C-Codes [68] (adaptiert)

```
void dcs2(Datentyp* energygrid,int gridx,int gridy,int gridz,
  Datentyp gridspacing,const Datentyp *atoms,int numatoms)
{
  int i,j,k,a;
  Datentyp dx,dy,dz;
  Datentyp x,y,z;

  Datentyp *precomputedz2=malloc(numatoms*sizeof(Datentyp)) ;
  Datentyp *precomputedz2plusdy2=malloc(numatoms*sizeof(Datentyp)) ;

  for(k=0;k<gridz;k++){
    z=gridspacing*(Datentyp)k;

    for(a=0;a<numatoms;a++){
      dz=z-atoms[a*4+2];
      precomputedz2[a]=dz*dz;
    }

    for(j=0;j<gridy;j++){
      y=gridspacing*(Datentyp)j;

      for(a=0;a<numatoms;a++){
        dy=y-atoms[a*4+1];
        precomputedz2plusdy2[a]=precomputedz2[a]+dy*dy;
      }

      for(i=0;i<gridx;i++){
        x=gridspacing*(Datentyp)i;

        Datentyp energy=0.0f;
        for(a=0;a<numatoms;a++){
          dx=x-atoms[a*4];
          energy+=atoms[a*4+3]/sqrtf(precomputedz2plusdy2[a]+dx*dx);
          //energy+=atoms[a*4+3]/sqrt(precomputedz2plusdy2[a]+dx*dx);
        }
        energygrid[k*gridx*gridy+j*gridx+i]=energy;
      }
    }
  }
  free(precomputedz2);
  free(precomputedz2plusdy2);
}
```

Dritte Variante

Die dritte Variante (siehe Code 53 und grüne Kurve in Abbildung 89) ist am schnellsten. Hier wurde die Variante 2 hergenommen und statt einem gemeinsamen Atom-Array für x, y, z und w (siehe atoms im Code 52) wird je ein Vektor für x, y, z und w (siehe atomsx, atomsy, atomsz und atomsw im Code 53) verwendet. Dadurch stehen die nacheinander auszulesenden Daten nebeneinander im Speicher und die Daten können somit schneller ausgelesen werden.

Code 53: Dritte Variante des CPU-C-Codes [68] (adaptiert)

```
void dcs3(Datentyp* energygrid,int gridx,int gridy,int gridz,
  Datentyp gridspacing,const Datentyp *atomsx,const Datentyp *atomsy,
  const Datentyp *atomsz,const Datentyp *atomsw,int numatoms)
{
  int i,j,k,a;
  Datentyp dx,dy,dz;
```

Fortsetzung von Code 53 auf der nächsten Seite

Fortsetzung von Code 53:

```
Datentyp x,y,z;
Datentyp *precomputedz2=malloc(numatoms*sizeof(Datentyp));
Datentyp *precomputedz2plusdy2=malloc(numatoms*sizeof(Datentyp));

for(k=0;k<gridz;k++){
    z=gridspacing*(Datentyp)k;

    for(a=0;a<numatoms;a++){
        dz=z-atomsz[a];
        precomputedz2[a]=dz*dz;
    }

    for(j=0;j<gridy;j++){
        y=gridspacing*(Datentyp)j;

        for(a=0;a<numatoms;a++){
            dy=y-atomsy[a];
            precomputedz2plusdy2[a]=precomputedz2[a]+dy*dy;
        }

        for(i=0;i<gridx;i++){
            x=gridspacing*(Datentyp)i;

            Datentyp energy=0.0f;
            for(a=0;a<numatoms;a++){
                dx=x-atomsx[a];
                energy+=atomsx[a]/sqrtf(precomputedz2plusdy2[a]+dx*dx);
                //energy+=atomsx[a]/sqrt(precomputedz2plusdy2[a]+dx*dx);
            }
            energygrid[k*gridx*gridy+j*gridx+i]=energy;
        }
    }
}
free(precomputedz2);
free(precomputedz2plusdy2);
}
```

8.2.1.2 Unterschiedliche Compiler und Compiler-Flags

Die Wahl des Compilers spielt beim DCS eine wichtige Rolle (siehe Abbildung 90). Würde man sich für den gcc-Compiler entscheiden, wäre der GPU-Speedup beim DCS-Algorithmus deutlich höher als bei der Verwendung des icc-Compilers und der GPU-Speedup somit nicht fair. Das „-xSSE3“-Flag, das beim Needleman-Wunsch-Algorithmus zu einer deutlichen Performance-Steigerung geführt hat, spielt beim DCS-Algorithmus keine Rolle.

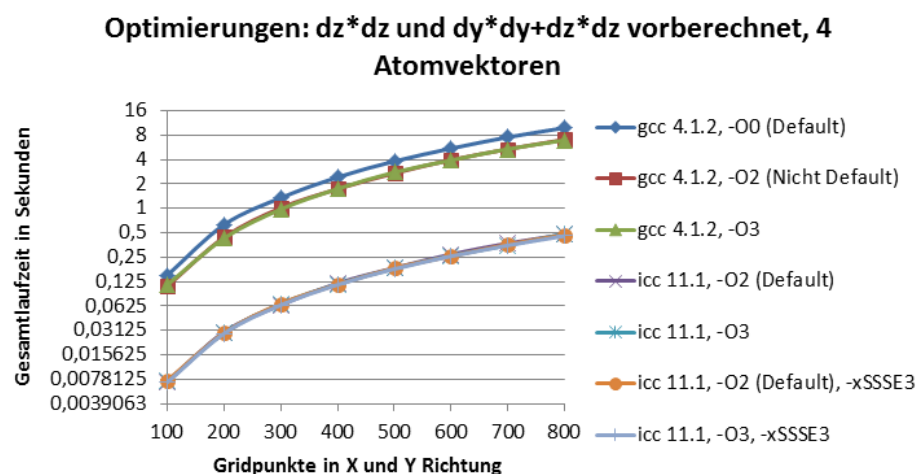


Abbildung 90: Wahl des Compilers und der Compilerflags

8.2.1.3 Atom-Evaluierungen pro Sekunde

Um die Leistung von Programmen vergleichen zu können, wird beim DCS-Algorithmus oft die Anzahl der Atom-Evaluierungen pro Sekunde angegeben. Damit hat man eine Zahl, die unabhängig von der verwendeten CPU ist, und es muss auch nicht der CPU-Code für einen fairen Vergleich optimiert werden. In Abbildung 91 sind die gemessenen Atom-Evaluierungen pro Sekunde für die drei Varianten des CPU-C-Codes dargestellt. Bei allen drei Varianten ergibt sich ab 300 Gridpunkten in X- und Y-Richtung eine horizontale Linie. Die Hardware erreicht ab diesem Punkt die maximale Auslastung.

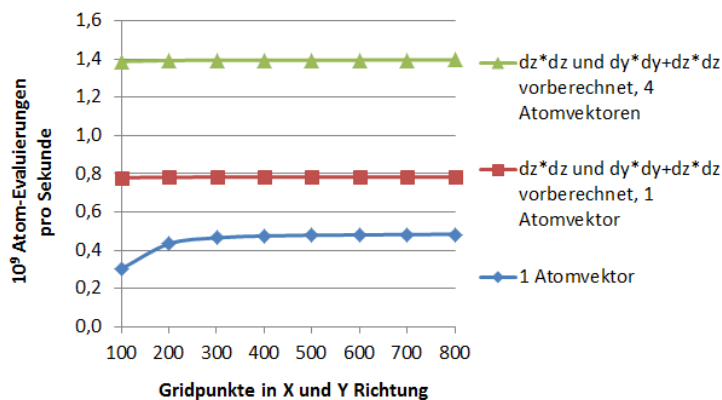


Abbildung 91: Anzahl der Atom-Evaluierungen pro Sekunde

8.2.2 OpenMP

Da jeder Gridpunkt unabhängig voneinander berechnet werden kann [1], ist die Parallelisierung mit minimalstem Aufwand möglich (siehe Code 54). Über die i-Schleife wird die OpenMP-Deklaration geschrieben. Somit wird jedem Thread immer die komplette Berechnung eines Gridpunkts zugewiesen.

Code 54: OpenMP-Code

```
...
#pragma omp parallel default(none) shared(energygrid,numatoms,atomsx,atomsw,
precomputedz2plusdy2,precomputedz2,k,j,gridspacing,gridx,gridy) private(a,i,dx,x)
{
#pragma omp for
for(i=0;i<gridx;i++){
x=gridspacing*(Datentyp)i;
Datentyp energy=0.0f;

for(a=0;a<numatoms;a++){
dx=x-atomsx[a];
energy+=atomsw[a]/sqrtf(precomputedz2plusdy2[a]+dx*dx);
//energy+=atomsw[a]/sqrt(precomputedz2plusdy2[a]+dx*dx);
}
energygrid[k*gridx*gridy+j*gridx+i]=energy;
}
}
```

Es muss beim Kompilieren das Flag "-openmp" und im Programm "#include <omp.h>" angegeben werden. Beim Aufruf des Programms wird die Anzahl der Threads übergeben ("dcs <Anzahl von Atomen> <Anzahl von Gitterpunkten - X Richtung> <Anzahl von Gitterpunkten - Y Richtung> <Anzahl von Gitterpunkten - Z Richtung> <Abstand zwischen Gitterpunkten> <Anzahl von Threads>"). Mit "omp_set_num_threads(atoi(argv[6]));" wird in der main-Methode die Anzahl der Threads festgelegt.

Da sich auf dem Mainboard zwei CPUs befinden und die Verwendung von zwei CPUs gegenüber einer GPU nicht fair wäre, wurde die Ausführung auf die zweite CPU beschränkt (siehe Kapitel 5.2). In Abbildung 92 sieht man, dass 8 Threads besser sind als 4 Threads, dass bei 800 Gridpunkten in X- und Y-Richtung der Speedup 4 erreicht wird und dass das Flag „-xSSSE3“ auch bei OpenMP keinen starken Einfluss auf die Laufzeit hat.

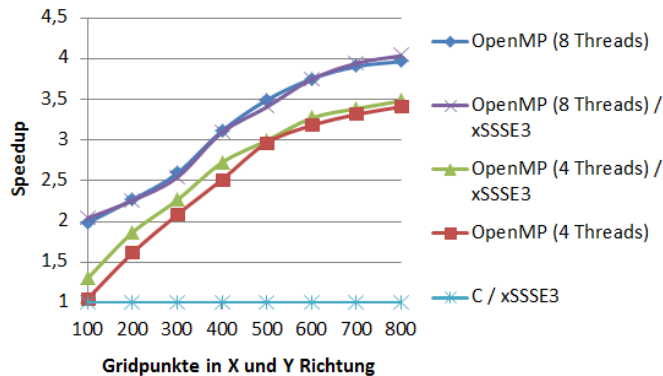


Abbildung 92: Speedup des OpenMP-Programms

8.3 GPU-Programm mit Laufzeitmessungen

Als Basis für die Speedup-Diagramme wird die CPU-Konfiguration „Intel Xeon X5550, dz*dz und dy*dy+dz*dz vorberechnet, 4 Atomvektoren, icc 11.1, -O3“ verwendet.

Die Beschreibung des Algorithmus und der möglichen GPU-Optimierungen wurde aus dem Kapitel 9 (CUDA-Version) und Kapitel 11 (OpenCL-Version) vom Buch „Programming Massively Parallel Processors“ [1] entnommen.

Bei den Laufzeitmessungen beträgt die Anzahl der Atome immer 1000, gridspacing immer 0.005, die Anzahl der z-Ebenen immer 1 und ein Block von Threads ist immer 16x16 (=256 Threads) groß.

8.3.1 Global-Memory- und Const-Memory-Programm

Für die Parallelisierung auf der GPU wird die Berechnung einer z-Ebene parallelisiert (siehe Code 55).

Code 55: Berechnung einer z-Ebene [1] (adaptiert)

```
for(j=0;j<gridy;j++){
    float y=gridspacing*(float)j;
    for(i=0;i<gridx;i++){
        float x=gridspacing*(float)i;
        float energy=0.0f;
        for(n=0;n<atomarrdim;n+=4){
            float dx=x-atoms[n];
            float dy=y-atoms[n+1];
            float dz=z-atoms[n+2];
            energy+=atoms[n+3]/sqrtf(dx*dx+dy*dy+dz*dz);
        }
        energygrid[k*gridx*gridy+j*gridx+i]=energy;
    }
}
```

Für das parallele Berechnen des 2D-Grids wird ein 2D-Thread-Grid verwendet [1]. Jeder Thread berechnet einen Gridpunkt. Um ein 2D-Thread-Grid verwenden zu können, müssen die zwei äußeren Schleifen des 2D-Grids „perfectly nested“ sein [1]. Es dürfen keine Statements dazwischen stehen. Um das zu ermöglichen, wird die Berechnung der y-Koordinate der Gridpunkte eine Schleife hinuntergestellt (siehe Code 56) [1].

Code 56: Keine Statements zwischen den zwei äußeren Schleifen als Vorbereitung für das 2D-Thread-Grid [1] (adaptiert)

```
for(j=0;j<gridy;j++){
    for(i=0;i<gridx;i++){
        float y=gridspacing*(float)j;
        float x=gridspacing*(float)i;
        float energy=0.0f;
        for(n=0;n<atomarrdim;n+=4){
            float dx=x-atoms[n];
            float dy=y-atoms[n+1];
            float dz=z-atoms[n+2];
            energy+=atoms[n+3]/sqrtf(dx*dx+dy*dy+dz*dz);
        }
        energygrid[k*gridx*gridy+j*gridx+i]=energy;
    }
}
```

Die zwei äußeren Schleifen entfallen im Kernel-Code und werden durch das Grid ersetzt (siehe Code 57) [1]. Die Größe des Grids wird beim Aufruf des Kernels festgelegt [1]. Statt den zwei Zählervariablen (i und j) für die zwei äußeren Schleifen gibt es nun die x- und y-Position des Threads im 2D-Grid.

Code 57: Kernel-Code [1] (adaptiert)

```
__global__ void dcs_gpu_kernel_1(float *energygrid,float *atoms,dim3 grid,
float gridspacing,int numatoms,int zindex)
{
    int xindex=blockIdx.x*blockDim.x+threadIdx.x;
    int yindex=blockIdx.y*blockDim.y+threadIdx.y;
    int outaddr=zindex*grid.x*grid.y+yindex*grid.x+xindex;

    float coorx=gridspacing*(float)xindex;
    float coory=gridspacing*(float)yindex;
    float coorz=gridspacing*(float)zindex;
    float energyval=0.0f;
    int atomid;
    float dx,dy,dz;

    for(atomid=0;atomid<numatoms;atomid++){
        dx=coorx-atoms[atomid*4+0];
        dy=coory-atoms[atomid*4+1];
        dz=coorz-atoms[atomid*4+2];
        energyval+=atoms[atomid*4+3]/sqrtf(dx*dx+dy*dy+dz*dz);
    }
    energygrid[outaddr]=energyval;
}
```

Im Buch [1] sind im ersten Kernel bereits die Optimierungen *rsqrtf*, *float4* und *Vorbereitung von dz*dz auf CPU* enthalten. In der Masterarbeit besteht der erste Kernel ohne diese Optimierungen. Der Inhalt des ersten Kernels vom Buch kann im Code 58 eingesehen werden.

Code 58: Erster Kernel-Code vom Buch [1]

```
float curenergy=energygrid[outaddr];
float coorx=gridspacing*xindex;
float coory=gridspacing*yindex;
int atomid;
float energyval=0.0f;
for(atomid=0;atomid<numatoms;atomid++){
    float dx=coorx-atominfo[atomid].x;
    float dy=coory-atominfo[atomid].y;
    energyval+=atominfo[atomid].w*rsqrtf(dx*dx+dy*dy+atominfo[atomid].z);
}
energygrid[outaddr]=curenergy+envergyval;
```

Da das 2D-Grid von Gitterpunkten oft nicht ein Vielfaches der Größe der Threadblöcke ist, muss das 2D-Grid etwas größer als nötig gewählt werden (siehe Code 59) [1]. Das wird als „Lattice padding“ bezeichnet [1]. Die Threads, die außerhalb des benötigten 2D-Grids liegen, führen zwar Berechnungen durch, deren Ergebnisse werden aber im Endergebnis einfach nicht berücksichtigt.

Code 59: Lattice padding

```
dim3 grid;
grid.x=atoi(argv[2]);
grid.y=atoi(argv[3]);
grid.z=atoi(argv[4]);
float gridspacing=(float)atof(argv[5]);

dim3 dimBlock;
dimBlock.x=atoi(argv[6]);
dimBlock.y=atoi(argv[7]);

dim3 gridGPU;
dim3 dimGrid;
if(grid.x%dimBlock.x==0){
    dimGrid.x=grid.x/dimBlock.x;
    gridGPU.x=grid.x;
}
else{
    dimGrid.x=grid.x/dimBlock.x+1;
    gridGPU.x=dimGrid.x*dimBlock.x;
}

if(grid.y%dimBlock.y==0){
    dimGrid.y=grid.y/dimBlock.y;
    gridGPU.y=grid.y;
}
else{
    dimGrid.y=grid.y/dimBlock.y+1;
    gridGPU.y=grid.y+dimBlock.y;
}

int zindex;
for(zindex=0;zindex<gridGPU.z;zindex++){
    dcs_gpu_kernel_1<<<dimGrid,dimBlock>>>(energygrid_gpu_d,atoms_gpu_d,gridGPU,
        gridspacing,numatoms,zindex);
}
```

Es muss sowohl im Arbeitsspeicher der CPU als auch im Arbeitsspeicher der GPU ein Array für die Gridpunkte und ein Array für die Atome angelegt werden (siehe Code 60). Am Ende des Programms müssen diese Arrays wieder freigegeben werden.

Code 60: Anlegen und Freigeben der Arrays für die Gridpunkte und Atome

```
//Arrays im Arbeitsspeicher der CPU anlegen
float *energygrid_gpu_h=(float*)malloc(numgridpointsGPU*sizeof(float));
float *atoms_gpu_h=(float*)malloc(numatoms*4*sizeof(float));

//Arrays im Arbeitsspeicher der GPU erzeugen
float *energygrid_gpu_d;
cudaMalloc((void**)&energygrid_gpu_d,numgridpointsGPU*sizeof(float));
float *atoms_gpu_d;
cudaMalloc((void**)&atoms_gpu_d,numatoms*4*sizeof(float));
...
//Arrays im Arbeitsspeicher der GPU freigeben
cudaFree(energygrid_gpu_d);
cudaFree(atoms_gpu_d);

//Arrays im Arbeitsspeicher der CPU freigeben
free(energygrid_gpu_h);
free(atoms_gpu_h);
```

Die Atomdaten werden vom Arbeitsspeicher der CPU zum Arbeitsspeicher der GPU übertragen und nachdem die Berechnung durchgeführt wurde, werden die Daten der Gridpunkte zurückübertragen (siehe Code 61).

Code 61: Übertragen der Atomdaten und Zurückübertragen der Ergebnisse der Gridpunkte

```
//Array von CPU-Arbeitsspeicher in GPU-Arbeitsspeicher übertragen
cudaMemcpy(atoms_gpu_d,atoms_gpu_h,numatoms*4*sizeof(float),cudaMemcpyHostToDevice);

//Ergebnis von GPU-Arbeitsspeicher in CPU-Arbeitsspeicher übertragen
cudaMemcpy(energygrid_gpu_h,energygrid_gpu_d,numgridpointsGPU*sizeof(float),
    cudaMemcpyDeviceToHost);
```

Da alle Gridpunkte/Threads auf dieselben Werte des atoms-Array zugreifen und diese Werte sich nicht ändern, ist das atoms-Array ein idealer Kandidat für den Constant Memory [1]. Da das atoms-Array größer sein kann als die Größe des Constant Memory (=64 KByte pro GPU), muss das atoms-Array in 64 KB Teile zerlegt werden und pro Kernelaufruf wird ein solcher Teil abgearbeitet (siehe Code 63) [1]. Der Constant Memory wird vor jedem Kernelaufruf neu befüllt. Am Ende des Kernels muss immer das aktuelle Zwischenergebnis im Global Memory gesichert werden [1]. Das führt dazu, dass im nächsten Kernelaufruf, in dem der nächste Teil des atoms-Array abgearbeitet wird, das Zwischenergebnis vom vorigen Kernelaufruf wieder ausgelesen und mit der aktuellen Berechnung aktualisiert werden muss [1]. Damit die Latenzzeit des zusätzlichen Global-Memory-Lesezugriffs kein Performance-Problem darstellt, wird der Gridwert zu Beginn des Kernels gelesen und erst am Ende des Kernels benutzt [1]. Damit kann die Global-Memory-Latenzzeit besser versteckt werden und man benötigt weniger Ersatzwarps pro Multiprozessor [1]. Sobald alle Teile des atoms-Array abgearbeitet worden sind, kann das 2D-Grid vom Global Memory in den CPU-Arbeitsspeicher übertragen werden [1].

Im Programmcode gibt es zwei Kernels (siehe Code 62). Der erste Kernel wird für den ersten 64KB-Teil des atom-Arrays verwendet. Sollte das Atom-Array nicht größer als 64 KB sein, wird nur dieser Kernel ausgeführt. Dieser Kernel hat den Vorteil, dass er keinen Lesezugriff auf den Global Memory durchführen muss. Wenn das atom-Array größer als 64 KB ist, kommt der zweite Kernel zum Einsatz, der den zuvor berechneten Zwischenwert für den Gridpunkt ausliest und ihn um den Ergebniswert des zweiten 64KB-Teils aktualisiert. Bei den Laufzeitmessungen für die Masterarbeit beträgt die Anzahl der Atome immer 1000. Beim Datentyp float ist somit der 64 KB Constant Memory zu einem Viertel (16 KB = 4 Byte (=Float)*4 Werte*1000) belegt. Beim Datentyp double wird der Constant Memory zur Hälfte (32 KB = 8 Byte (=Double)*4 Werte*1000) befüllt. Somit wird immer nur der erste Kernel aufgerufen.

Wenn man die innerste Schleife des Kernels betrachtet, so kommt man auf 4 Speicherzugriffe und 11 Operationen. Wären die 4 Datenzugriffe normale Global-Memory-Zugriffe, hätte man einen sehr schlechten CGMA (=Verhältnis von Floating-Point-Operationen zu Global-Memory-Zugriffe [1]) von 2,75 (=11/4). Bei allen 4 Datenzugriffen handelt es sich aber um Constant-Memory-Zugriffe. Constant-Memory-Zugriffe führen zwar auch zu Device-Memory-Zugriffen, da der Constant Memory bei Nvidia-GPUs ein Teil des Device Memory ist, jedoch besitzt der Constant Memory einen eigenen Cache (=8 KByte) pro Multiprozessor, womit bei bereits vorhandenen Daten im Cache kein Device-Memory-Zugriff notwendig ist [1]. Die Daten vom Constant-Memory-Cache werden über Broadcasts sehr schnell zu vielen Threads transportiert [1]. Wenn die Elemente im Cache sehr oft von allen Threads wiederverwendet werden (bei DCS der Fall), reduzieren sich die notwendigen Datenzugriffe auf den Device Memory deutlich und somit sind die Bandbreite und Latenzzeit des Device Memory keine limitierenden Faktoren mehr.

Code 62: Kernel-Code [1] (adaptiert)

```
//Verarbeitung des ersten 64 KB-Teils des Atom-Array
__global__ void dcs_gpu_kernel_2_Erster_Aufgerufener_Kernel(float* energygrid,
dim3 grid, float gridspacing,int zindex,int numatoms)
{
    int xindex=blockIdx.x*blockDim.x+threadIdx.x;
    int yindex=blockIdx.y*blockDim.y+threadIdx.y;
    int outaddr=zindex*grid.x*grid.y+yindex*grid.x+xindex;

    float coorx=gridspacing*xindex;
    float coory=gridspacing*yindex;
    float coorz=gridspacing*zindex;

    float energyval=0.0f; float dx,dy,dz; int atomid;
    for(atomid=0;atomid<numatoms;atomid++){
        dx=coorx-atoms[atomid*4];
        dy=coory-atoms[atomid*4+1];
        dz=coorz-atoms[atomid*4+2];
        energyval+=atoms[atomid*4+3]/sqrtf(dx*dx+dy*dy+dz*dz);
    }
    energygrid[outaddr]=energyval;
}

//Verarbeitung des zweiten 64 KB-Teils und der darauf folgenden 64 KB-Teile des
//Atom-Array
__global__ void dcs_gpu_kernel_2(Float* energygrid,dim3 grid,float gridspacing,
int zindex,int numatoms)
{
    int xindex=blockIdx.x*blockDim.x+threadIdx.x;
    int yindex=blockIdx.y*blockDim.y+threadIdx.y;
    int outaddr=zindex*grid.x*grid.y+yindex*grid.x+xindex;

    float curenergy=energygrid[outaddr];
    float coorx=gridspacing*xindex;
    float coory=gridspacing*yindex;
    float coorz=gridspacing*zindex;

    float energyval=0.0f;
    int atomid;
    float dx,dy,dz;
    for(atomid=0;atomid<numatoms;atomid++){
        dx=coorx-atoms[atomid*4];
        dy=coory-atoms[atomid*4+1];
        dz=coorz-atoms[atomid*4+2];
        energyval+=atoms[atomid*4+3]/sqrtf(dx*dx+dy*dy+dz*dz);
    }
    energygrid[outaddr]=curenergy+energyval;
}
```

Code 63: Host-Code

```

#define CHUNK_SIZE 4000 // Anzahl von moeglichen Float4-Werten (4000*4*4 Byte=64
//KByte) - Groesse des Const-Memory
__constant__ float atoms[CHUNK_SIZE*4]; // Atom-Array im Constant Memory anlegen

double dcs_gpu_2(float *energygrid_gpu_h,int numgridpointsGPU,
float *atoms_gpu_h,int numatoms,dim3 gridGPU,float gridspacing,dim3 dimBlock,
dim3 dimGrid)
{
int r,i;

//Array im GPU-Arbeitsspeicher erzeugen
float *energygrid_gpu_d;
cudaMalloc((void*)&energygrid_gpu_d,numgridpointsGPU*sizeof(Float));

//Berechnungen durchfuehren
struct timeval startTimeGPUKernel,endTimeGPUKernel;
gettimeofday(&startTimeGPUKernel,(struct timezone *)0);
int zindex;
for(zindex=0;zindex<gridGPU.z;zindex++){
if(numatoms<CHUNK_SIZE){
cudaMemcpyToSymbol(atoms,atoms_gpu_h,numatoms*4*sizeof(Float));
dcs_gpu_kernel_2_Erster_Aufgerufener_Kernel<<<dimGrid,dimBlock>>>(
energygrid_gpu_d,gridGPU,gridspacing,zindex,numatoms);
}
else{
r=numatoms%CHUNK_SIZE;
if(r==0){
for(i=0;i<numatoms/CHUNK_SIZE;i++){
cudaMemcpyToSymbol(atoms,&atoms_gpu_h[i*CHUNK_SIZE*4],CHUNK_SIZE*4*
sizeof(Float));
if(i==0)
dcs_gpu_kernel_2_Erster_Aufgerufener_Kernel<<<dimGrid,dimBlock>>>(
energygrid_gpu_d,gridGPU,gridspacing,zindex,CHUNK_SIZE);
else
dcs_gpu_kernel_2<<<dimGrid,dimBlock>>>(energygrid_gpu_d,gridGPU,gridspacing,
zindex,CHUNK_SIZE);
}
}
else{
for(i=0;i<(numatoms-r)/CHUNK_SIZE;i++){
cudaMemcpyToSymbol(atoms,&atoms_gpu_h[i*CHUNK_SIZE*4],
CHUNK_SIZE*4*sizeof(Float));
if(i==0)
dcs_gpu_kernel_2_Erster_Aufgerufener_Kernel<<<dimGrid,dimBlock>>>(
energygrid_gpu_d,gridGPU,gridspacing,zindex,CHUNK_SIZE);
else
dcs_gpu_kernel_2<<<dimGrid,dimBlock>>>(energygrid_gpu_d,gridGPU,gridspacing,
zindex,CHUNK_SIZE);
}
cudaMemcpyToSymbol(atoms,&atoms_gpu_h[i*CHUNK_SIZE*4],r*4*sizeof(Float));
dcs_gpu_kernel_2<<<dimGrid,dimBlock>>>(energygrid_gpu_d,gridGPU,gridspacing,
zindex,r);
}
}
}
cudaThreadSynchronize();
gettimeofday(&endTimeGPUKernel,(struct timezone *)0);

//Ergebnis zurueck uebertragen
cudaMemcpy(energygrid_gpu_h,energygrid_gpu_d,numgridpointsGPU*sizeof(Float),
cudaMemcpyDeviceToHost);

//Array im GPU-Arbeitsspeicher freigeben
cudaFree(energygrid_gpu_d);

return ZeitdauerBerechnen(startTimeGPUKernel,endTimeGPUKernel);
}

```

Die nachfolgenden Abbildungen stellen die Speedups der Basisprogramme dar. Die Basisprogramme sind ohne Optimierungen. Im Paper zur Masterarbeit [14] wurde bei der Basisversion bei dem Global-Memory-Programm die Optimierung rsqrt und bei dem Const-Memory-Programm die Optimierungen rsqrt und float4 verwendet. In der Masterarbeit selbst sind bei beiden Basisversionen keine Optimierungen enthalten. Die Speedup-Diagramme zeigen die Gesamt- und Kernel-Speedups des Const- und Global-Memory-Programms. Die Speedups der CUDA-Programme werden in Abbildung 93 (GTX 480), Abbildung 94 (GTX 285), Abbildung 95 (Tesla C2050) und Abbildung 96 (Tesla C1060) dargestellt. Die Speedups der OpenCL-Programme können in Abbildung 97 (GTX 480), Abbildung 98 (GTX 285), Abbildung 99 (Tesla C2050) und Abbildung 100 (Tesla C1060) eingesehen werden.

CUDA:

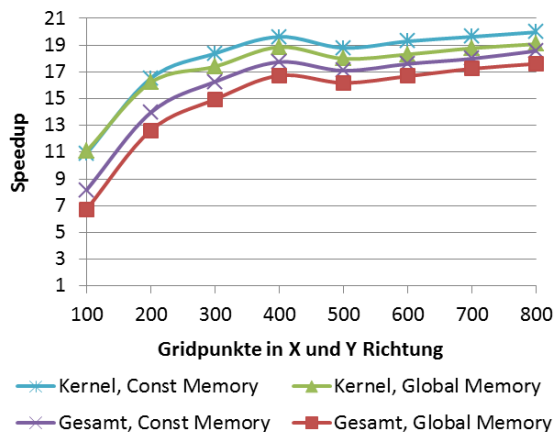


Abbildung 93: GTX 480

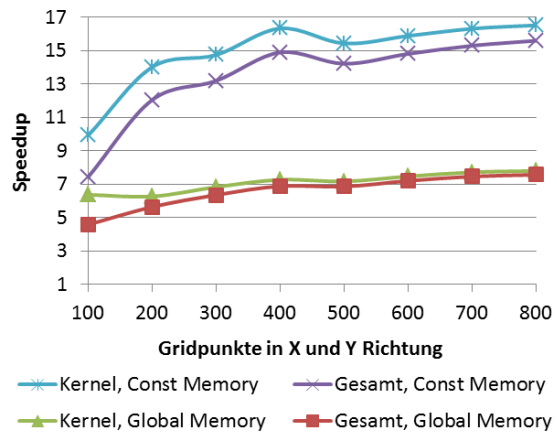


Abbildung 94: GTX 285

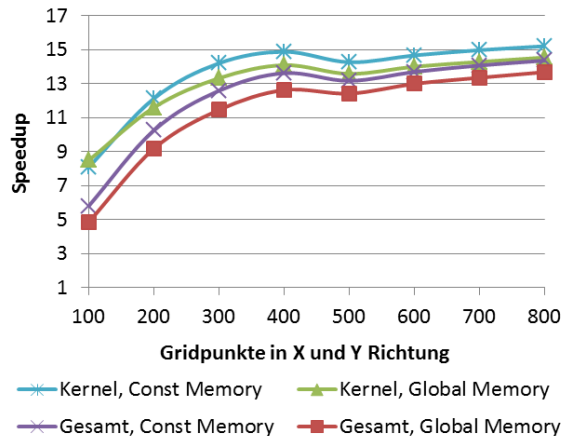


Abbildung 95: Tesla C2050

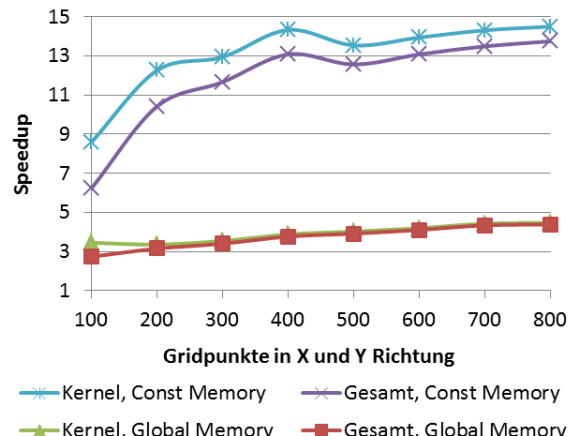


Abbildung 96: Tesla C1060

OpenCL:

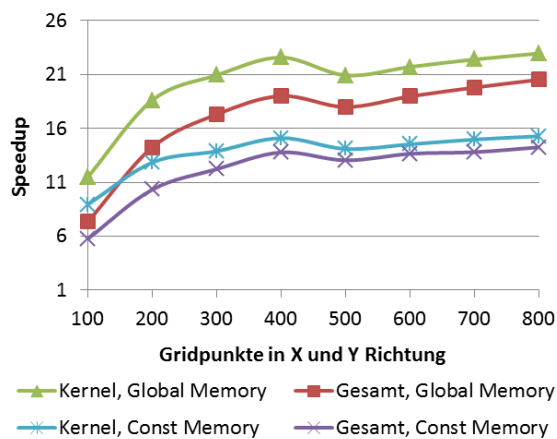


Abbildung 97: GTX 480

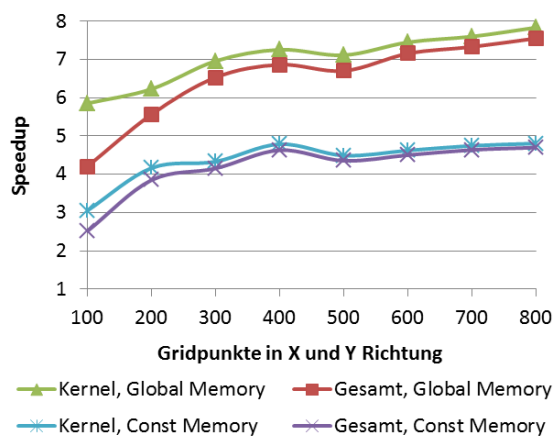


Abbildung 98: GTX 285

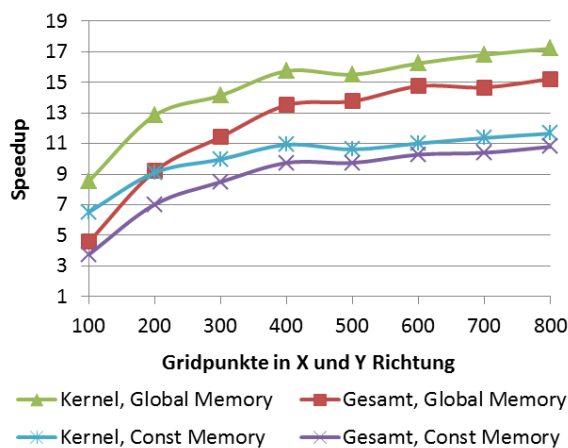


Abbildung 99: Tesla C2050

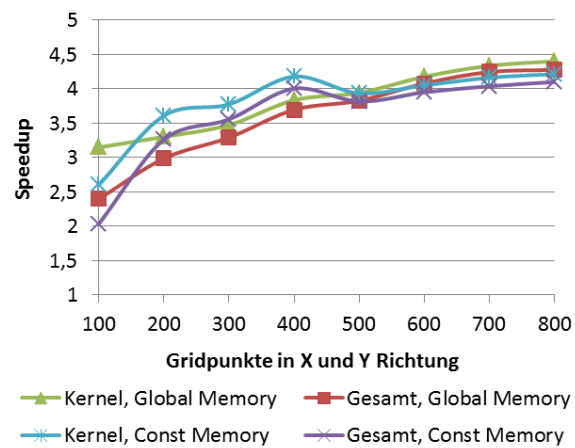


Abbildung 100: Tesla C1060

In den Speedup-Diagrammen zu den CUDA-Programmen (siehe Abbildung 93, Abbildung 94, Abbildung 95 und Abbildung 96) sieht man, dass der Speedup des Const-Memory-Programms und des Global-Memory-Programms bei der Fermi-Architektur (GTX 480, Tesla C2050) im Gegensatz zur GT200-Architektur (GTX 285, Tesla C1060) nahe beieinander liegen. Der Grund ist, dass die Fermi-Architektur für den Global Memory eine Cache-Hierarchie besitzt. Bei den Speedup-Diagrammen zu den OpenCL-Programmen (siehe Abbildung 97, Abbildung 98, Abbildung 99 und Abbildung 100) kann man bei der Fermi- und der GT200-Architektur eine Reduzierung des Speedups durch Constant Memory erkennen. Constant Memory scheint somit bei OpenCL nicht zu funktionieren.

8.3.2 Optimierungen

Die Speedup-Werte in den Unterkapiteln sind von der GPU GTX 480 und dem CUDA-Programm.

8.3.2.1 Verwendung von rsqrtf

Statt x/sqrtf wird $x*\text{rsqrtf}$ im Kernel verwendet. Da sqrtf mit $1/\text{rsqrtf}$ berechnet wird ($1/x$ und rsqrtf werden auf den SFUs ausgeführt, siehe Kapitel 2.1.5), erspart man sich eine aufwendige „reciprocal“-Operation ($=1/x$) [5]. Außerdem wird die Division durch eine Multiplikation ersetzt ($x*\text{rsqrtf}$ statt x/sqrtf). Da die Multiplikation komplett auf der FMA-Einheit ausgeführt wird, ist sie schneller als eine Division. Damit kann die Performance noch einmal gesteigert werden. Eine Division in Compute Capability 1.x wird über die "reciprocal"-Operation durchgeführt und das Ergebnis ist nicht IEEE konform [5]. Erst mit Compute Capability 2.x ist das Ergebnis einer Division IEEE konform [5].

Eine „1.0/sqrtf()“-Operation kann der Compiler nur dann in eine rsqrtf()-Operation umwandeln, wenn beim Kompilieren die Flags „-prec-div=false“ und „-prec-sqrt=false“ angegeben werden und somit 1/x und sqrtf() approximiert werden [5]. Nvidia empfiehlt statt der Verwendung dieser beiden Compiler-Flags die rsqrtf()-Operation überall dort einzusetzen, wo sie benötigt wird [5].

Den Genauigkeitsunterschied zwischen sqrtf und rsqrtf kann man sich in den Tabellen darunter ansehen (siehe Tabelle 37, Tabelle 38, Tabelle 39 und Tabelle 40). Der „maximum ulp error“ in der Tabelle gibt an, wie viele ULPs das GPU-Ergebnis vom exakt gerundeten Ergebnis maximal abweichen wird [5]. Das Ergebnis kann bei rsqrtf (Single-Precision) um 2 ULPs und bei rsqrt (Double-Precision) um 1 ULP abweichen [5]. Im Gegensatz dazu liefern die zwei Operationen „x/y“ und sqrtf (Single-Precision) bzw. sqrt (Double-Precision) mit 0 ULP-Abweichung sowohl in Single-Precision als auch in Double-Precision das exakt gerundete Ergebnis [5]. Standardmäßig werden Programme bei Fermi-GPUs (Compute Capability 2.0 und 2.1) IEEE konform (Flags: -ftz=false, -prec-div=true, -prec-sqrt=true) kompiliert und ausgeführt [5]. Würden die Programme nicht IEEE konform kompiliert und ausgeführt werden, würde es mehr als 0 ULP-Error geben [5]. Damit der Code IEEE-754 konform bleibt, ersetzt der Compiler nicht automatisch 1.0/sqrtf durch rsqrtf [6]. Deswegen muss im Programmcode die Änderung durchgeführt werden. Interessant ist, dass bei der Vorgängerarchitektur GT200 (Compute Capability < 2) bei Single-Precision bei Verwendung von sqrtf statt rsqrtf das Ergebnis mit 3 ULP sogar ungenauer werden könnte [5]. Bei der GT200-Architektur kann mit rsqrtf die Genauigkeit und die Performance gesteigert werden. Bei der Fermi-Architektur kann es hingegen mit rsqrtf zu einer Verschlechterung der Genauigkeit kommen. Bei der Ausführung der Programme auf der GTX 480-Fermi und auf der GTX 285-GT200 konnte keine Verbesserung oder Verschlechterung der Genauigkeit durch das Verwenden von rsqrtf festgestellt werden. Die standardisierte Residual Norm (siehe Berechnung im Kapitel 6.2) bleibt gleich.

Tabelle 37: CUDA-Single-Precision [5] (Ausschnitt der Originaltabelle)

Function	Maximum ulp error
x/y	0 for compute capability ≥ 2 when compiled with -prec-div=true 2 (full range), otherwise
1/x	0 for compute capability ≥ 2 when compiled with -prec-div=true 1 (full range), otherwise
rsqrtf(x)	2 (full range)
1/sqrtf(x)	Applies to 1/sqrtf(x) only when it is converted to rsqrtf(x) by the compiler.
sqrtf(x)	0 for compute capability ≥ 2 when compiled with -prec-sqrt=true 3 (full range), otherwise

Tabelle 38: CUDA-Double-Precision [5] (Ausschnitt der Originaltabelle)

Function	Maximum ulp error
x/y	0 (IEEE-754 round-to-nearest-even)
1/x	0 (IEEE-754 round-to-nearest-even)
sqrt(x)	0 (IEEE-754 round-to-nearest-even)
rsqrt(x)	1 (full range)

Tabelle 39: OpenCL-Single-Precision [6] (Ausschnitt der Originaltabelle)

Function	Maximum ulp error
x/y	0 for compute capability ≥ 2 2 (full range), otherwise
1/x	0 for compute capability ≥ 2 1 (full range), otherwise
rsqrt(x) 1/sqrt(x)	2 (full range) Applies to 1/sqrt(x) only when it is converted to rsqrt(x) by the compiler.
sqrt(x)	0 for compute capability ≥ 2 3 (full range), otherwise

Tabelle 40: OpenCL-Double-Precision [6] (Ausschnitt der Originaltabelle)

Function	Maximum ulp error
x/y	0 (IEEE-754 round-to-nearest-even)
1/x	0 (IEEE-754 round-to-nearest-even)
sqrt(x)	0 (IEEE-754 round-to-nearest-even)
rsqrt(x)	1 (full range)

Wenn man die Tabellen zwischen CUDA und OpenCL vergleicht, sieht man einen Unterschied. Bei CUDA wird beim Funktionsnamen zwischen Single- und Double-Precision unterschieden [5]. Bei OpenCL ist der Funktionsname bei Single- und Double-Precision identisch [6].

Auf der Fermi-GPU können pro Taktzyklus pro Multiprozessor zwei sqrtf-Operationen ausgeführt werden [6]. Bei der Vorgängerarchitektur GT200 ist es eine Operation [6] [15]. Bei rsqrtf können auf der Fermi-GPU vier Operationen und bei der Vorgängerarchitektur GT200 zwei Operationen pro Taktzyklus pro Multiprozessor ausgeführt werden (siehe Tabelle 41) [6] [15]. Damit kann rsqrtf doppelt so schnell wie sqrtf ausgeführt werden. Der Grund ist, dass sqrtf auf der GPU über $x = \text{rsqrtf}$ (reciprocal square root) und $y = 1/x$ (reciprocal) berechnet wird [5].

Tabelle 41: Anzahl von Single-Precision-Operationen pro Taktzyklus auf einem Multiprozessor (erstellt auf Basis einer Tabelle von [6], Informationen von [6] [15])

	GT200	Fermi
32-bit floating-point reciprocal, reciprocal square root	2	4

Die Codeänderung durch diese Optimierung ist minimal (siehe Code 64).

Code 64: Verwendung der rsqrtf()-Funktion [1] (adaptiert)

Vorher	Nachher
<code>energyval+=atoms[atomid*4+3]/ sqrtf(dx*dx+dy*dy+dz*dz);</code>	<code>energyval+=atoms[atomid*4+3]* rsqrtf(dx*dx+dy*dy+dz*dz);</code>

Bei N=800, M=800 kann der Gesamt-Speedup von 18,6 auf 27,9 und der Kernel-Speedup von 20 auf 31,4 gesteigert werden (siehe Abbildung 101).

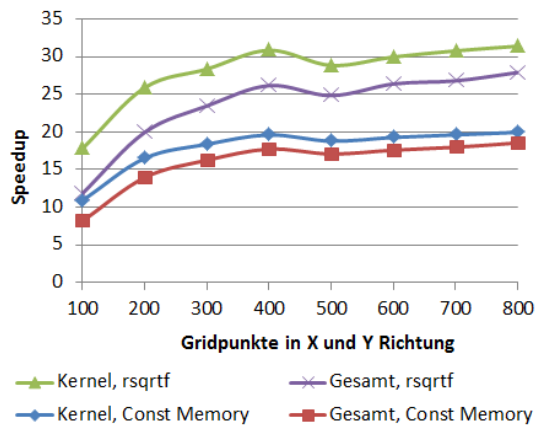


Abbildung 101: Auswirkung der rsqrtf()-Funktion auf den Speedup

8.3.2.2 Atoms-Array als float4-Variable bzw. lokale float4-Variable

Bei der Optimierung float4 wird das atoms-Array mit dem float4-Datentyp abgespeichert (siehe Code 65). Dadurch soll der Zugriff schneller funktionieren.

Die Optimierung bringt keine Verbesserung (siehe Abbildung 102 und Abbildung 103).

Code 65: Verwendung einer float4-Variable für das Atom-Array [1] (adaptiert)

Vorher	Nachher
Außerhalb des Kernels: <pre>__constant__ float atoms[CHUNK_SIZE*4];</pre> Im Kernel: <pre>dx=coorx-atoms[atomid*4]; dy=coory-atoms[atomid*4+1]; dz=coorz-atoms[atomid*4+2]; energyval+=atoms[atomid*4+3]* rsqrtf(dx*dx+dy*dy+dz*dz);</pre>	Außerhalb des Kernels: <pre>__constant__ float4 atoms[CHUNK_SIZE];</pre> Im Kernel: <pre>dx=coorx-atoms[atomid].x; dy=coory-atoms[atomid].y; dz=coorz-atoms[atomid].z; energyval+=atoms[atomid].w* rsqrtf(dx*dx+dy*dy+dz*dz);</pre>

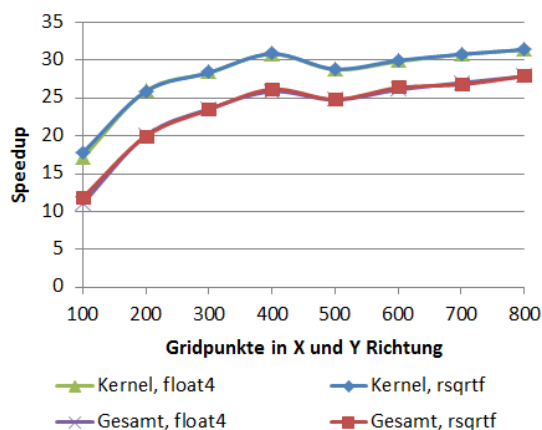


Abbildung 102: Mit Constant Memory

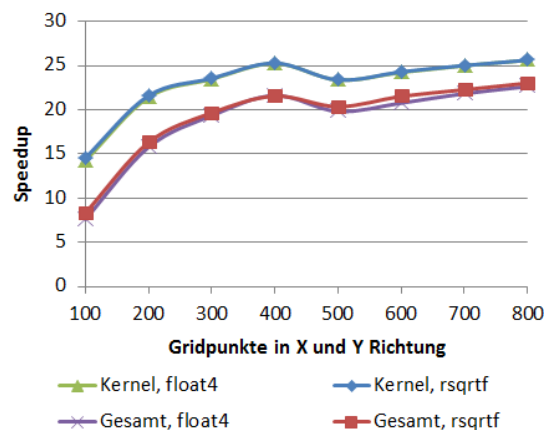


Abbildung 103: Mit Global Memory

Bei der Optimierung „lokale float4-Variable“ wird das atoms-Array mit dem float4-Datentyp abgespeichert und durch das einmalige Auslesen der float4-Variable in eine lokale float4-Variable muss nur einmal statt vier Mal geladen werden (siehe Code 66). Die Optimierung wird in allen Diagrammen in der Masterarbeit mit „float4 lokal“ abgekürzt.

Code 66: Verwendung einer lokalen float4-Variable [1] (adaptiert)

Vorher (mit float4)	Nachher (mit float4 und lokaler float4-Variable)
Außerhalb des Kernels: <pre>__constant__ float4 atoms[CHUNK_SIZE];</pre> Im Kernel: <pre>dx=coorx-atoms[atomid].x; dy=coory-atoms[atomid].y; dz=coorz-atoms[atomid].z; energyval+=atoms[atomid].w* rsqrtf(dx*dx+dy*dy+dz*dz);</pre>	Außerhalb des Kernels: <pre>__constant__ float4 atoms[CHUNK_SIZE];</pre> Im Kernel: <pre>float4 atom=atoms[atomid]; dx=coorx-atom.x; dy=coory-atom.y; dz=coorz-atom.z; energyval+=atom.w* rsqrtf(dx*dx+dy*dy+dz*dz);</pre>

Die Optimierung „lokale float4-Variable“ befindet sich nicht im Buch [1], da das Buch nur die CUDA-Version mit Const Memory analysiert und dort diese Optimierung keine Auswirkung hat. Diese Optimierung wurde bei der Suche nach der Ursache des großen Laufzeitunterschieds bei OpenCL zwischen der Fermi und der Vorgängerarchitektur GT200 gefunden. Die minimalen Änderungen haben eine sehr starke Auswirkung bei der OpenCL-Performance mit Global Memory bei der Vorgängerarchitektur GT200. Die folgenden Speedup-Vergleiche werden bei der Problemgröße $N=800$ und $M=800$ und mit dem Kernelspeedup durchgeführt. Bei OpenCL/GTX 285/Global Memory erreicht man mit float4 einen Speedup von 7,8 und mit float4-local-Variable einen Speedup von 19,2. Der Unterschied ist ein Faktor von 2,46. Bei OpenCL/Tesla C1060/Global Memory erreicht man mit float4 einen Speedup von 4,4 und mit float4-local-Variable einen Speedup von 16,5. Der Unterschied ist mit einem Faktor von 3,75 sehr hoch. Bei OpenCL/GTX480/Global Memory erreicht man mit float4 einen Speedup von 23 und mit float4-local-Variable einen Speedup von 25. Der Unterschied ist minimal. Bei OpenCL/Tesla C2050/Global Memory erreicht man mit float4 einen Speedup von 17,2 und mit float4-local-Variable einen Speedup von 18,9. Ebenfalls ein geringer Unterschied.

Bei Constant Memory bewirkt diese Optimierung keine Verbesserung (siehe Abbildung 104). Bei Global Memory (siehe Abbildung 105) kann bei $N=800$, $M=800$ der Gesamt-Speedup von 23 auf 25 und der Kernel-Speedup von 25,6 auf 28 gesteigert werden.

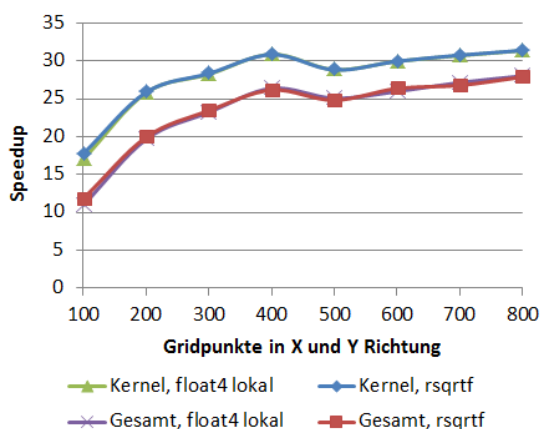


Abbildung 104: Mit Constant Memory

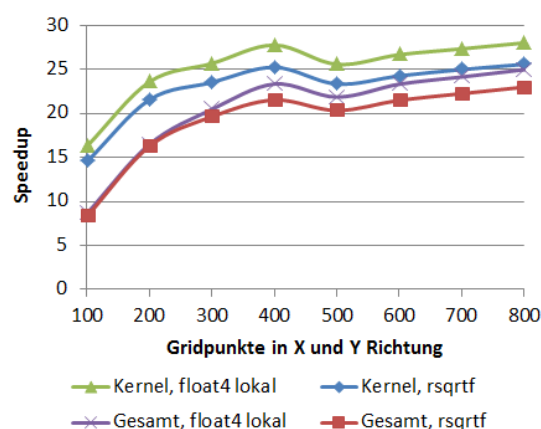


Abbildung 105: Mit Global Memory

8.3.2.3 Vorberechnung von $dz*dz$ auf CPU

Da die z-Koordinate aller Gridpunkte beim 2D-DCS (x, y) dieselbe ist, führt die Berechnung von dz bei allen Gridpunkten zum selben Ergebnis. Die Berechnung von $dz*dz$ muss somit nur einmal pro Atom für alle Gridpunkte gemacht werden (siehe Code 67). Nach der Berechnung können alle Gridpunkte

diesen Wert verwenden (siehe Code 68). Dazu berechnet man $dz*dz$ auf der CPU und speichert den Wert in das atom-Array auf dem Speicherplatz auf dem die z-Koordinate stehen würde.

Code 67: Host-Code, Vorbereitung von $dz*dz$ auf CPU

```

Vorher:
int zindex;
for(zindex=0;zindex<gridGPU.z;zindex++){...}

Nachher:
Datentyp *atoms_z=(Datentyp*)malloc(numatoms*sizeof(Datentyp));
for(i=0;i<numatoms;i++){
    atoms_z[i]=atoms_gpu_4_h[i].z;

Datentyp coorz,dz;
int zindex;
for(zindex=0;zindex<gridGPU.z;zindex++){

    //dz*dz fuer alle Gridpunkte einmal berechnen
    coorz=gridspacing*zindex;
    for(i=0;i<numatoms;i++){
        dz=coorz-atoms_z[i];
        atoms_gpu_4_h[i].z=dz*dz;
    }
    ...
}

free(atoms_z); //Von Arrays belegten CPU-Arbeitsspeicher freigeben

```

Code 68: Kernel-Code, Vorbereitung von $dz*dz$ auf CPU [1] (adaptiert)

```

Vorher:
dx=coorx-atoms[atomid].x;
dy=coory-atoms[atomid].y;
dz=coorz-atoms[atomid].z;
energyval+=atoms[atomid].w*rsqrtf(dx*dx+dy*dy+dz*dz);

Nachher:
dx=coorx-atominfo[atomid].x;
dy=coory-atominfo[atomid].y;
energyval+=atominfo[atomid].w*rsqrtf(dx*dx+dy*dy+atominfo[atomid].z);

```

Bei $N=800$, $M=800$ kann der Gesamt-Speedup von 27,9 auf 32,5 und der Kernel-Speedup von 31,4 auf 37,7 gesteigert werden (siehe Abbildung 106).

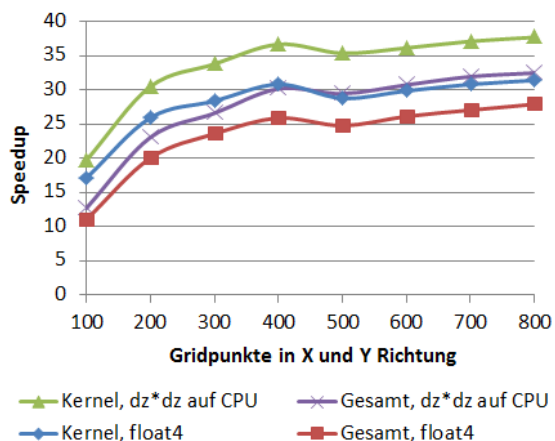


Abbildung 106: Auswirkung der Vorbereitung von $dz*dz$ auf CPU auf den Speedup

8.3.2.4 Berechnung mehrerer Gridpunkte in X-Richtung pro Thread

Um die Anzahl der Lade-Instruktionen auf den Constant Memory bzw. Constant-Memory-Cache zu reduzieren, sollte ein Thread die geladenen Atomdaten für die Berechnung mehrerer Gridpunkte verwenden [1]. Durch die Berechnung mehrerer Gridpunkte kann nicht nur die Anzahl der Ladeoperationen deutlich reduziert werden, sondern es müssen manche Operationen nur einmal durchgeführt werden [1]. Zu den Operationen zählt die Berechnung von dy und „ $dy*dy+dz*dz$ “ in der Wurzel, wenn der Thread mehrere Gridpunkte entlang der X-Achse berechnet (siehe Code 69). Der gesamte Kernelcode befindet sich im Code 78 im Anhang A.3. Der große Nachteil bei der Berechnung mehrerer Gridpunkte durch einen Thread ist, dass man dadurch die Anzahl der Threads und somit die Parallelität reduziert. Außerdem werden mehr Register pro Thread benötigt. Das kann im Falle des Erreichens der maximalen Anzahl von Registern dazu führen, dass weniger Threads pro Multiprozessor aktiv gehalten werden können (siehe Kapitel 2.2.4) [1]. Dies würde zu einer zusätzlichen Reduktion der Parallelität führen. Damit die maximale Anzahl von aktiven Threads pro Multiprozessor nicht reduziert wird, dürfen pro Thread bei der G80-Architektur maximal 10 Register (=8192 4-Byte-Register/ 768 aktive Threads), bei der GT200-Architektur maximal 16 Register (16384 4-Byte-Register / 1024 aktive Threads) und bei der Fermi-Architektur maximal 21 Register (32768 4-Byte-Register / 1536 aktive Threads) benützt werden [5]. Mit dem Angeben des Flags „--ptxas-options=-v“ beim Kompilieren des Programms kann die Anzahl der Register pro Thread für jeden Kernel ermittelt werden [20]. Beim Kompilieren mit 4 Gridpunkten pro Thread benötigt der erste Kernel (siehe `dcx_gpu_kernel_5_Erster_Aufgerufener_Kernel()` im Code 78 im Anhang A.3, für ersten 64KB-Teil des atom-Arrays) 14 Register und der zweite Kernel (siehe `dcx_gpu_kernel_5()` im Code 78 im Anhang A.3) 17 Register. Beim Kompilieren mit 8 Gridpunkten pro Thread benötigt der erste Kernel 22 Register und der zweite Kernel 29 Register. Bei der G80-Architektur, die im Buch [1] verwendet wird, wird die Registeranzahl pro Thread von 10 bei den Kernels mit 4 und 8 Gridpunkten pro Thread überschritten und es kommt somit zu einer Reduzierung der Anzahl der aktiven Threads pro Multiprozessor. Bei der GT200-Architektur (z. B. GPU GTX 285) kommt es nur beim Aufruf des ersten Kernels bei 4 Gridpunkten pro Thread zu keiner Reduktion der Anzahl der aktiven Threads. Bei der Fermi-Architektur (z. B. GPU GTX 480) hat man bei 8 Gridpunkten pro Thread eine Reduzierung der Anzahl der aktiven Threads.

Code 69: Berechnung mehrerer Gridpunkte in X-Richtung pro Thread [1] (adaptiert)

```
Vorher (Ein Gridpunkt pro Thread):
dx=coorx-atominfo[atomid].x;

dy=coory-atominfo[atomid].y;

energyval+=atominfo[atomid].w*rsqrtf(dx*dx+dy*dy+atominfo[atomid].z);

Nachher (Mehrere Gridpunkte in X-Richtung pro Thread):
x=atominfo[atomid].x;
for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    dx[i]=coorx[i]-x;

dy=coory-atominfo[atomid].y;

charge=atominfo[atomid].w;
dysqpdzsq=dy*dy+atominfo[atomid].z;
for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    energyval[i]+=charge*rsqrtf(dx[i]*dx[i]+dysqpdzsq);
```

Da die Anzahl der Gridpunkte pro Thread mit einem „#define“ am Anfang des Programmcodes festgelegt ist (siehe Code 78 im Anhang A.3), kann der Compiler die dazugehörigen for-Schleifen im Kernel aufrollen und die Arrays in Variablen umwandeln. Somit werden die Arrays im Kernel nicht im Local Memory gespeichert, sondern im Register Memory. Mit „#pragma unroll 1“ über der for-

Schleife würde der Compiler kein automatisches Aufrollen der Schleife durchführen [5]. Im Buch [1] werden die for-Schleifen (siehe nicht aufgerollte for-Schleifen im Code 70) bereits im Programmcode aufgerollt und die Arrays in Variablen transformiert (siehe aufgerollte for-Schleifen im Code 70). Das hat den Nachteil, dass man für die zwei Varianten 4 und 8 Gridpunkte in X-Richtung pro Thread entweder mehrere Kernels in den Programmcode schreiben oder den Kernel anpassen muss.

Code 70: Aufrollen von for-Schleifen [1] (adaptiert)

Nicht aufgerollte for-Schleifen (Code in der Masterarbeit):

```
float dx[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    dx[i]=coorx[i]-x;

for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    energyval[i]+=charge*rsqrtf(dx[i]*dx[i]+dysqpdzsq);
```

Aufgerollte for-Schleifen (Code im Buch [1]):

```
float dx1=coorx1-x;
float dx2=coorx2-x;
float dx3=coorx3-x;
float dx4=coorx4-x;

energyvalx1+=charge*rsqrtf(dx1*dx1+dysqpdzsq);
energyvalx2+=charge*rsqrtf(dx2*dx2+dysqpdzsq);
energyvalx3+=charge*rsqrtf(dx3*dx3+dysqpdzsq);
energyvalx4+=charge*rsqrtf(dx4*dx4+dysqpdzsq);
```

Bei 4 Gridpunkten in X-Richtung pro Thread kann bei N=800, M=800 der Gesamt-Speedup von 32,5 auf 45,3 und der Kernel-Speedup von 37,7 auf 56,5 gesteigert werden (siehe Abbildung 107). In den Diagrammen in der Masterarbeit wird „4 Gridpunkte in X-Richtung pro Thread“ in „4 Grid-X-Werte“ abgekürzt.

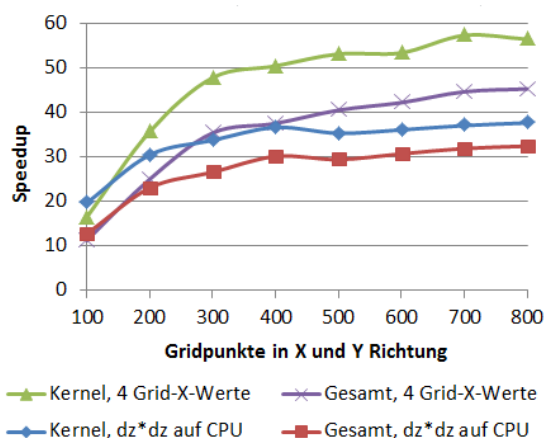


Abbildung 107: Auswirkung der Berechnung von 4 Gridpunkten in X-Richtung pro Thread auf den Speedup

Bei 8 Gridpunkten in X-Richtung pro Thread kann bei N=800, M=800 der Gesamt-Speedup von 32,5 auf 40,2 und der Kernel-Speedup von 37,7 auf 51,8 gesteigert werden (siehe Abbildung 108). In den Diagrammen in der Masterarbeit wird „8 Gridpunkte in X-Richtung pro Thread“ in „8 Grid-X-Werte“ abgekürzt.

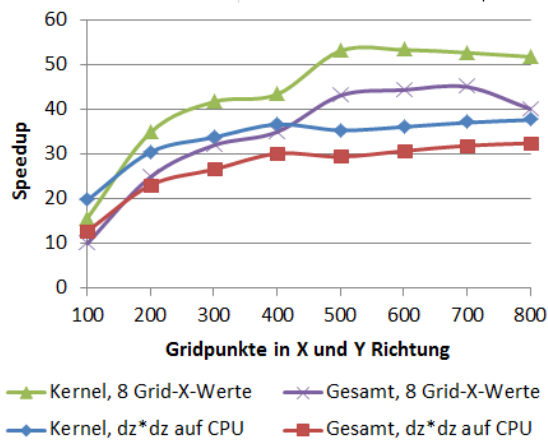


Abbildung 108: Auswirkung der Berechnung von 8 Gridpunkten in X-Richtung pro Thread auf den Speedup

8.3.2.5 Memory Coalescing

Mit der Optimierung „Memory Coalescing“ greifen Threads parallel auf Speicherstellen zu, die direkt nebeneinander liegen (siehe Kapitel 2.2.3.2). Zugriffe der Threads eines halben Warps (16 Threads) bei den älteren Architekturen bzw. eines vollen Warps (32 Threads) bei der Fermi-Architektur sollten immer auf Daten stattfinden, die direkt nebeneinander stehen, damit sie zu einem Datenzugriff zusammengefasst werden können und somit der Zugriff schneller funktioniert. Im Programm ist deswegen `blockDim.x` auf 16 gesetzt [1]. Das bedeutet, 16 Threads greifen auf Datenelemente zu, die direkt nebeneinander liegen.

Im Code 71 ist das Memory Coalescing für den Schreibvorgang der Ergebnisse dargestellt. Im Anhang A.3 befindet sich im Code 79 der gesamte Kernel-Code zu der Optimierung.

Code 71: Memory Coalescing mit `blockDim.x=16` [1] (adaptiert)

```

Vorher:
for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    energygrid[outaddr+i]=energyval[i];

Nachher:
for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    energygrid[outaddr+i*blockDim.x]=energyval[i];

```

Die Optimierung hat keine Auswirkung auf die Performance (siehe Abbildung 109). Der Grund für den gleichbleibenden Speedup ist die Festlegung der Anzahl der Atome auf 1000 bei den Laufzeit-Messungen. Somit passt das `atom-Array` komplett in den 64 KB großen Constant Memory und es ist kein Laden der Teilergebnisse der Gridpunkte aus dem Global Memory notwendig. Müsste der Kernel mehrmals aufgerufen werden, um mehrere 64KB-Teile des `atom-Arrays` zu verarbeiten, so müsste beim zweiten Mal der aktuelle Zwischenwert der Gridpunkte aus dem Global Memory geladen werden (siehe zweiten Kernel im Code 79 im Anhang A.3) und Memory Coalescing würde eine Auswirkung auf die Performance haben.

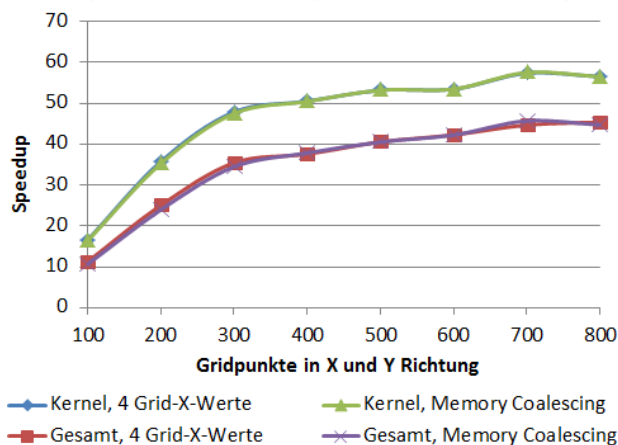


Abbildung 109: Auswirkung von Memory Coalescing auf den Speedup

8.3.2.6 Verwendung von Compiler-Optimierungsflags bei OpenCL

Durch die Verwendung von Optimierungsflags soll die Performance gesteigert werden. Der Nachteil ist, dass dadurch die Genauigkeit des Ergebnisses reduziert werden kann. Die Flags werden beim Kompiliercode des Kernels im OpenCL-Programm gesetzt (siehe Code 72).

Code 72: Verwendung von Compiler-Optimierungsflags im OpenCL-Programm [1] (adaptiert)

```
char compilerflagsStandard[4096];
sprintf(compilerflagsStandard, "-D Datentyp4=%s -D Datentyp=%s "
      "-D Anzahl_Gridwerte_in_XRichtung_Pro_Thread=%d "
      "-cl-fast-relaxed-math -cl-single-precision-constant "
      "-cl-denorms-are-zero -cl-mad-enable", Datentyp4_String,
      Datentyp_String, Anzahl_Gridwerte_in_XRichtung_Pro_Thread);
cl_program clProgram6a = clCreateProgramWithSource(clGPUContext, 1,
      (const char**) &programSource6a, NULL, &err1);
err2 = clBuildProgram(clProgram6a, 0, NULL, compilerflagsStandard, NULL, NULL);
cl_kernel clKernel6a = clCreateKernel(clProgram6a, Kernel6aName, &err3);
```

Es gibt zunächst „-cl-single-precision-constant“, „-cl-denorms-are-zero“ und „-cl-fast-relaxed-math“ [69]. Mit „-cl-single-precision-constant“ werden Double-Precision-Konstanten als Single-Precision-Konstanten behandelt [69]. Mit „-cl-denorms-are-zero“ wird festgelegt, dass denormalisierte Zahlen einfach auf 0 gesetzt werden [69]. Mit „-cl-fast-relaxed-math“ werden zwei weitere Flags „-cl-finite-math-only“ und „-cl-unsafe-math-optimizations“ gesetzt [69]. Das Compilerflag „-cl-finite-math-only“ erlaubt Floating-Point-Optimierungen, wobei angenommen wird, dass die Argumente und Ergebnisse von Berechnungen nicht NaNs oder $\pm\infty$ sind [69]. Mit „-cl-unsafe-math-optimizations“ werden zwei weitere Flags „-cl-no-signed-zeros“ und „-cl-mad-enable“ gesetzt [69]. Mit „-cl-no-signed-zeros“ wird das Vorzeichen der Zahl 0 nicht berücksichtigt [69]. Das Flag „-cl-mad-enable“ erlaubt laut der Kronos Group [69], dass „a*b+c“-Operationen als MAD-Operationen ausgeführt werden dürfen. Der Unterschied zwischen MAD- und FMA-Operation wird im Kapitel 2.1.2 erklärt. Eine MAD-Operation kann zu einer Reduktion der Genauigkeit führen, da schon beim Zwischenergebnis (a*b) auf Single- bzw. Double-Precision abgeschnitten wird [69]. Laut [70] wird das Flag „-cl-mad-enable“ bei Nvidia-GPUs nur bei den Architekturen vor Fermi benötigt. Bei Fermi wird die Operation „a*b+c“ über die Hardware als FMA-Operation ausgeführt [5]. Laut Nvidia CUDA C Programming Guide [5] und Nvidia OpenCL Programming Guide [6] bildet der Compiler bei den Architekturen vor Fermi automatisch aus Additionen und Multiplikationen MAD-Operationen und das Flag ist somit immer aktiv.

Bei den Speedups in Abbildung 110 muss beachtet werden, dass es sich hier um OpenCL-Werte und nicht um CUDA-Werte handelt. Bei N=800, M=800 kann der Gesamt-Speedup von 37,3 auf 41,6 und der Kernel-Speedup von 47,5 auf 54,8 gesteigert werden (siehe Abbildung 110). Beide Programme (mit Flags und ohne Flags) haben die Konfiguration „OpenCL, GTX 480, Global Memory, native_rsqrt,

lokale float4-Variable, $dz*dz$ auf CPU vorberechnet, 4 Gridpunkte in X-Richtung pro Thread, Memory Coalescing“.

Bei der Ausführung der Programme auf der GTX 480-Fermi und auf der GTX 285-GT200 konnte keine Verschlechterung der Genauigkeit durch das Setzen der Flags festgestellt werden. Die standardisierte Residual Norm (siehe Berechnung im Kapitel 6.2) bleibt gleich.

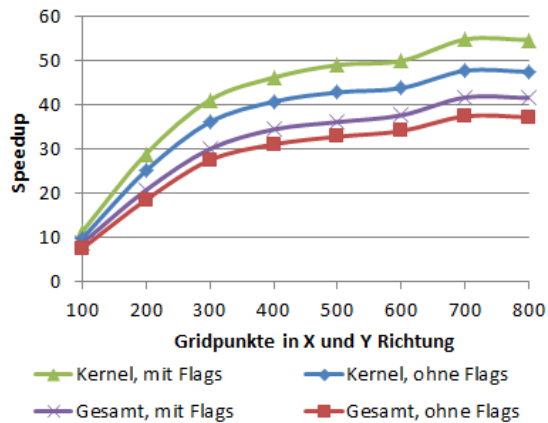


Abbildung 110: Auswirkung der Verwendung von Compilerflags bei OpenCL auf den Speedup

8.3.2.7 native_rsqrt() bei OpenCL

Im Buch [1], aus dem der GPU-Programmcod entnommen wurde, wird `native_rsqrt()` statt `rsqrt()` verwendet. Es wurde überprüft, ob das Verwenden von `native_rsqrt()` anstatt von `rsqrt()` zu einem Laufzeitvorteil führt. Das Ergebnis ist, dass es keinen Laufzeitunterschied zwischen `native_rsqrt()` und `rsqrt()` gibt (siehe Abbildung 111). Der Grund ist, dass es bei Nvidia-GPUs keine Implementierung für die Intrinsic-Funktion `native_rsqrt()` gibt (siehe Tabelle 31 im Kapitel 3.3.12.2) und somit statt der Intrinsic-Funktion `native_rsqrt()` die Standard-Funktion `rsqrt()` ausgeführt wird. Beide Programme (`native_rsqrt` und `rsqrt`) in Abbildung 111 haben die Konfiguration „OpenCL, GTX 480, Global Memory“.

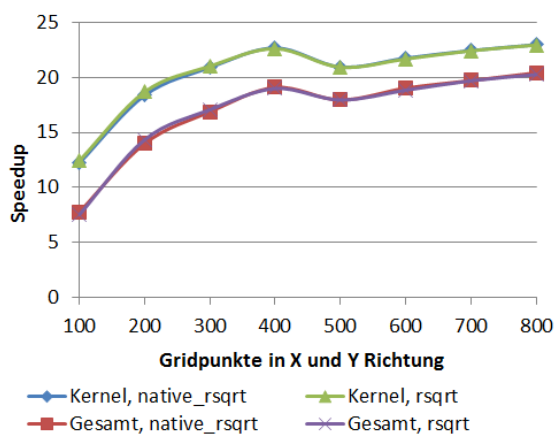


Abbildung 111: Auswirkung von `native_rsqrt()` bei OpenCL auf den Speedup

8.3.3 Zusammenfassung der Gesamtspeedups aller Optimierungen in einem Diagramm

In diesem Kapitel werden die Gesamtspeedups aller Optimierungen in einem Diagramm pro GPU und Programmiersprache zusammengefasst.

GPU GTX 480

Die auf der GTX 480 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 112 (CUDA-Programm) und Abbildung 113 (OpenCL-Programm).

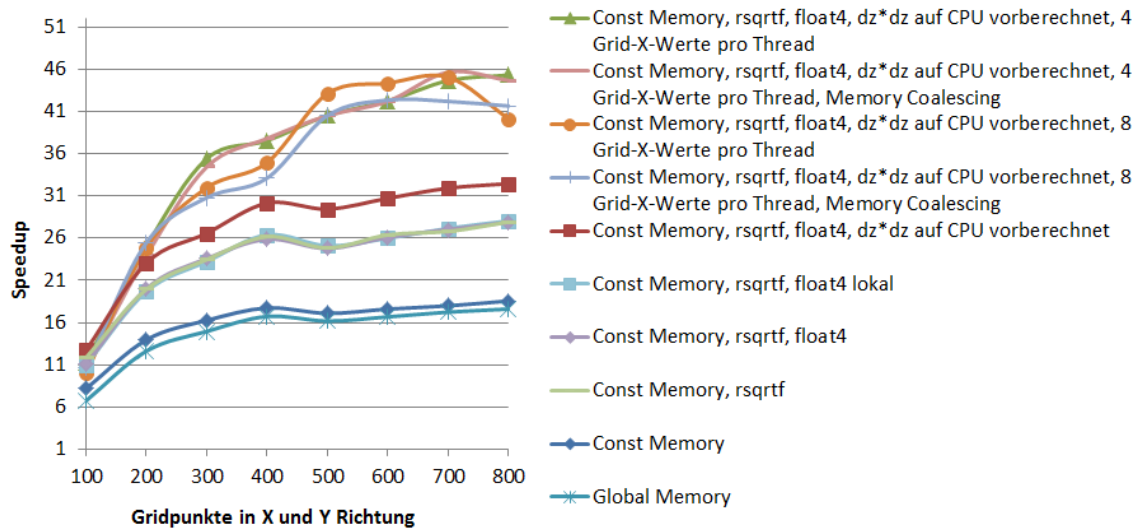


Abbildung 112: GTX 480, CUDA

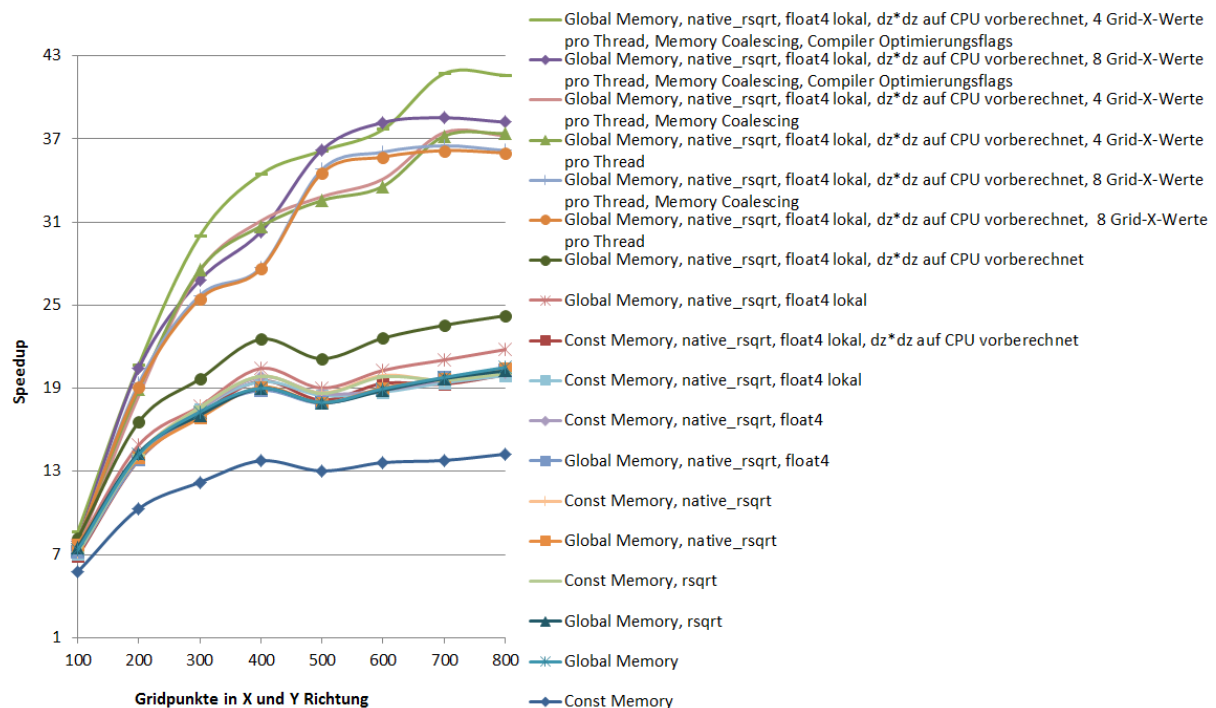


Abbildung 113: GTX 480, OpenCL

In Abbildung 112 (CUDA-Programm) sieht man, dass Constant Memory, float4 und Memory Coalescing eine sehr geringe Auswirkung auf den Speedup haben. "dz*dz auf CPU vorberechnet" steigert die Performance ein wenig. Rsqrtf und "mehrere Gridpunkte pro Thread" verbessern den Speedup deutlich.

In Abbildung 113 (OpenCL-Programm) sieht man, dass Constant Memory den Speedup verringert. Constant Memory scheint somit bei OpenCL nicht zu funktionieren. Float4 und Memory Coalescing haben eine sehr geringe Auswirkung auf den Speedup. "dz*dz auf CPU vorberechnet", lokale float4-Variable und Compiler-Optimierungsflags steigern die Performance ein wenig. Rsqrt führt bei Constant Memory im Gegensatz zu Global Memory zu einer Erhöhung des Speedups. "Mehrere Gridpunkte pro Thread" verbessern den Speedup deutlich.

GPU GTX 285

Die auf der GTX 285 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 114 (CUDA-Programm) und Abbildung 115 (OpenCL-Programm).

In Abbildung 114 (CUDA-Programm) sieht man, dass float4, Memory Coalescing und "dz*dz auf CPU vorberechnet" eine sehr geringe Auswirkung auf den Speedup haben. Constant Memory, rsqrtf und "mehrere Gridpunkte pro Thread" verbessern den Speedup deutlich.

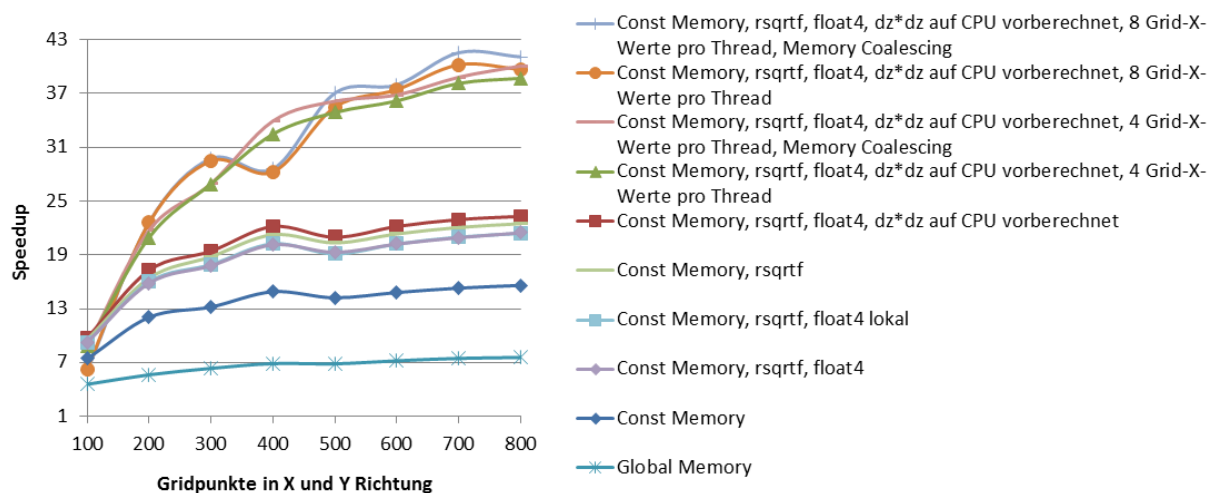


Abbildung 114: GTX 285, CUDA

In Abbildung 115 (OpenCL-Programm) sieht man, dass Constant Memory den Speedup reduziert. Constant Memory scheint somit bei OpenCL nicht zu funktionieren. Float4, "dz*dz auf CPU vorberechnet" und Memory Coalescing haben eine sehr geringe Auswirkung auf den Speedup. Rsqrt führt bei Constant Memory im Gegensatz zu Global Memory zu einer Erhöhung des Speedups. Die Optimierungen "lokale float4-Variable", "mehrere Gridpunkte pro Thread" und Compiler-Optimierungsflags verbessern den Speedup deutlich.

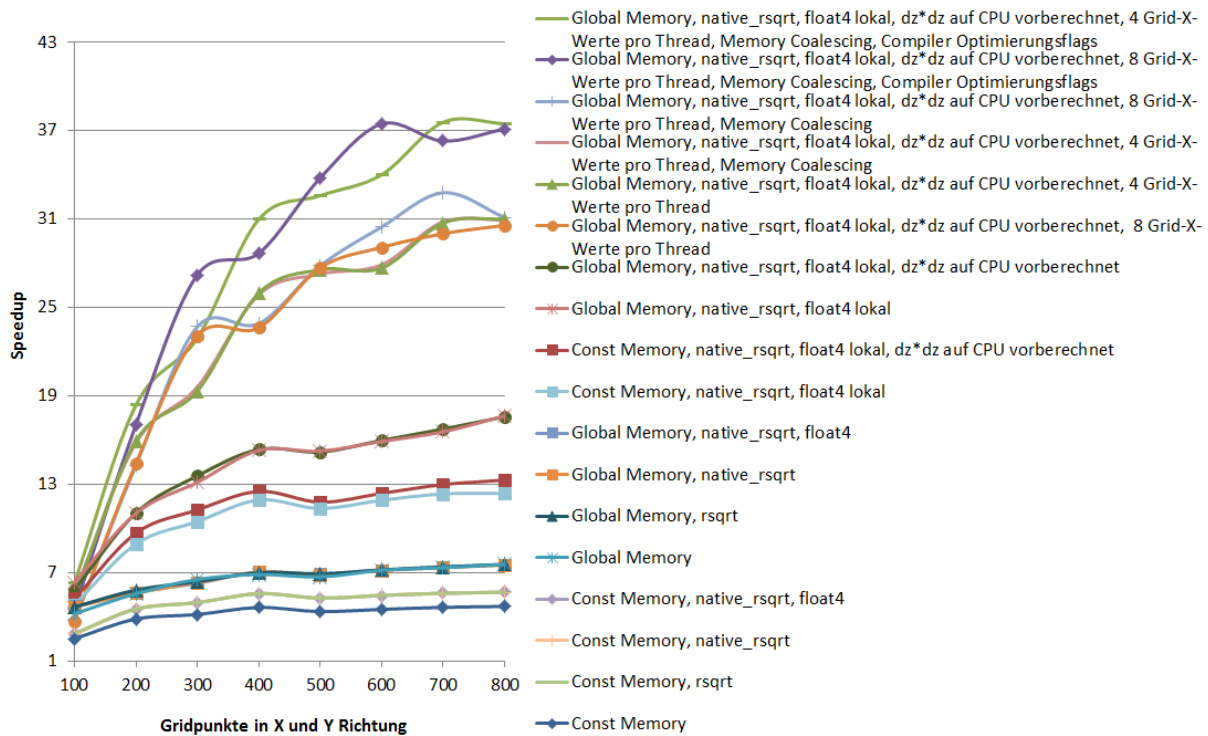


Abbildung 115: GTX 285, OpenCL

GPU Tesla C2050

Die auf der Tesla C2050 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 116 (CUDA-Programm) und Abbildung 117 (OpenCL-Programm).

In Abbildung 116 (CUDA-Programm) sieht man, dass Constant Memory, float4 und Memory Coalescing eine sehr geringe Auswirkung auf den Speedup haben. Die Optimierung "dz*dz auf CPU vorberechnet" steigert die Performance ein wenig. Rsqrtrf und "mehrere Gridpunkte pro Thread" verbessern den Speedup deutlich.

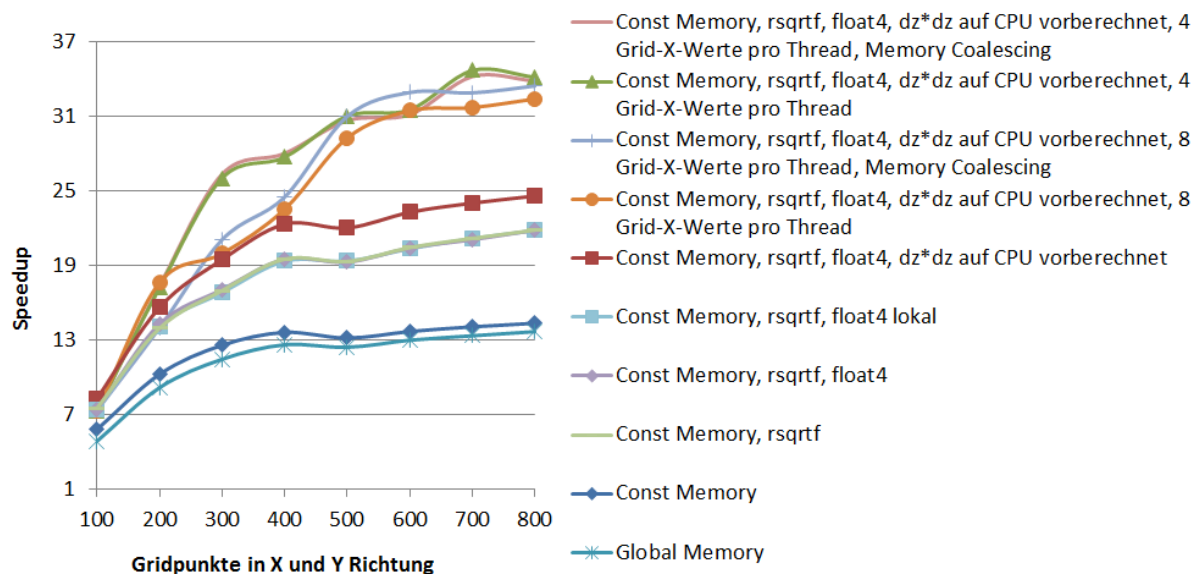


Abbildung 116: CUDA, Tesla C2050

In Abbildung 117 (OpenCL-Programm) sieht man, dass Constant Memory den Speedup verringert. Constant Memory scheint somit bei OpenCL nicht zu funktionieren. Float4 und Memory Coalescing haben eine sehr geringe Auswirkung auf den Speedup. Rsqrt führt bei Constant Memory im Gegensatz zu Global Memory zu einer Erhöhung des Speedups. Die Optimierungen "lokale float4-Variable", "dz*dz auf CPU vorberechnet" und Compiler-Optimierungsflags steigern die Performance ein wenig. "Mehrere Gridpunkte pro Thread" verbessert den Speedup deutlich.

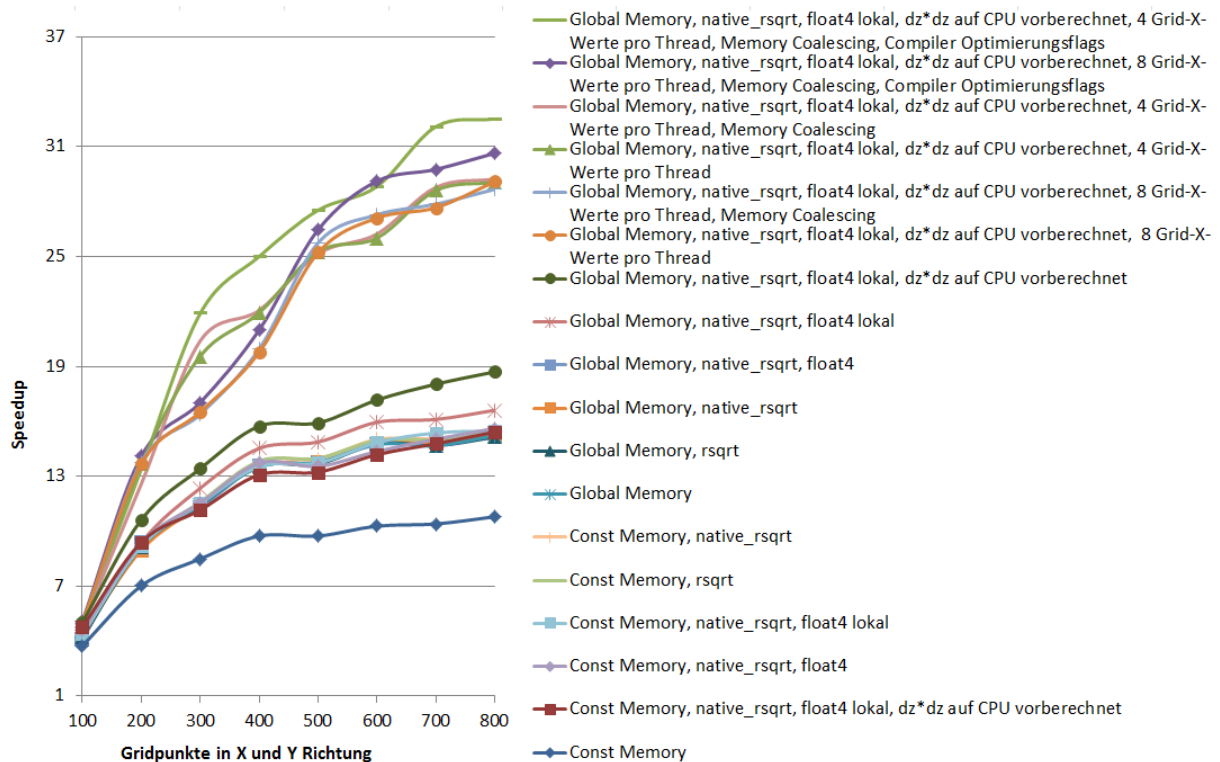


Abbildung 117: OpenCL, Tesla C2050

GPU Tesla C1060

Die auf der Tesla C1060 erreichten Gesamtspeedups aller Optimierungen befinden sich zusammengefasst in Abbildung 118 (CUDA-Programm) und Abbildung 119 (OpenCL-Programm).

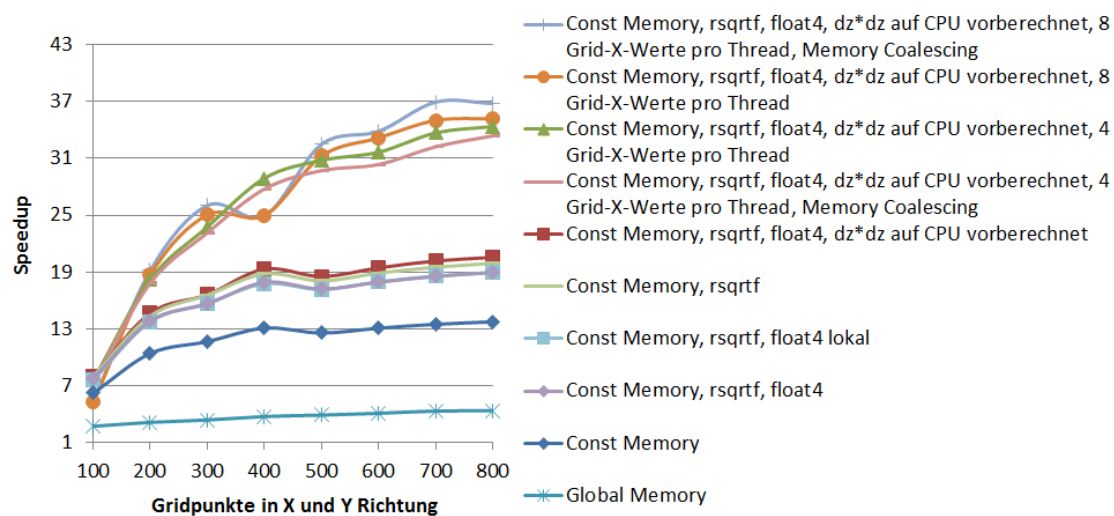


Abbildung 118: CUDA, Tesla C1060

In Abbildung 118 (CUDA-Programm) sieht man, dass float4 und "lokale float4-Variable" zu einer kleinen Verschlechterung des Speedups führen. Memory Coalescing und "dz*dz auf CPU vorberechnet" haben eine sehr geringe Auswirkung auf den Speedup. Constant Memory, rsqrtf und "mehrere Gridpunkte pro Thread" verbessern den Speedup deutlich.

In Abbildung 119 (OpenCL-Programm) sieht man, dass das Constant-Memory-Programm ab der Optimierung "lokale float4-Variable" weniger Speedup als das Global-Memory-Programm hat. Constant Memory scheint somit bei OpenCL nicht zu funktionieren. Rsqrt führt bei Constant Memory im Gegensatz zu Global Memory zu einer Erhöhung des Speedups. Float4, "dz*dz auf CPU vorberechnet" und Memory Coalescing haben eine sehr geringe Auswirkung auf den Speedup. Die Optimierungen "lokale float4-Variable", "mehrere Gridpunkte pro Thread" und Compiler-Optimierungsflags verbessern den Speedup deutlich.

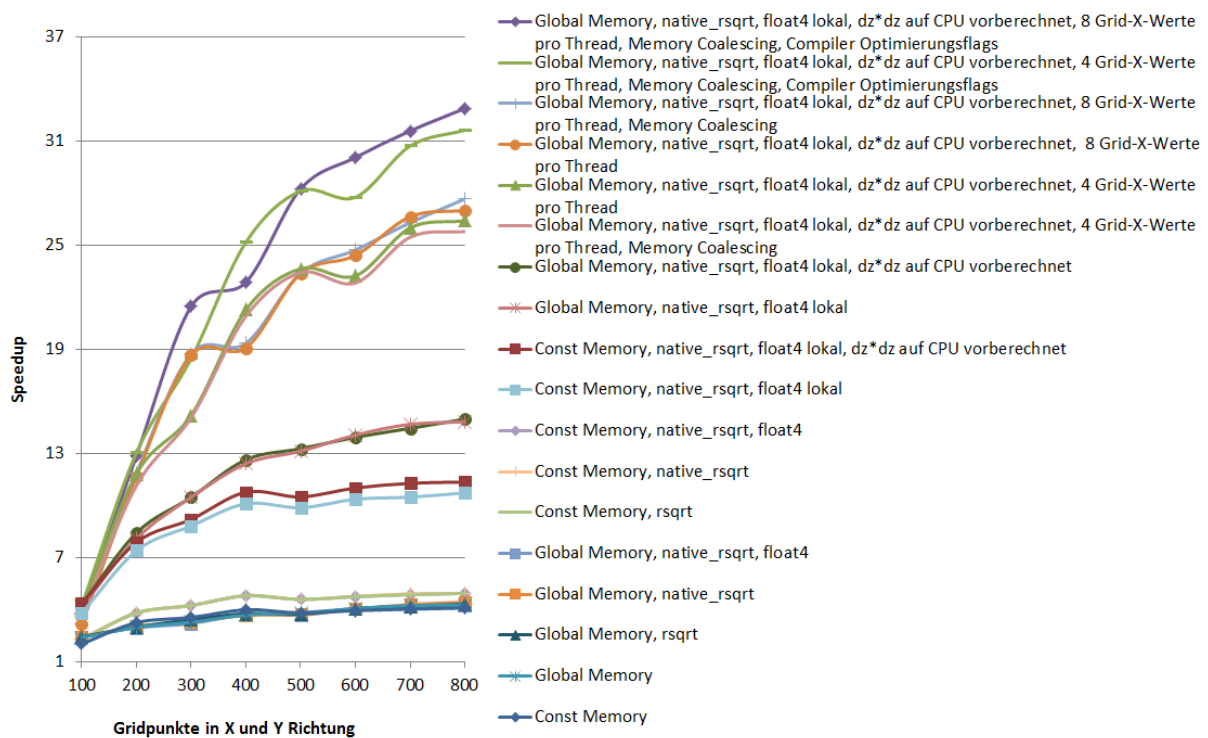


Abbildung 119: OpenCL, Tesla C1060

8.4 Diskussion der gemessenen GPU-Laufzeitdaten

8.4.1 Auswirkung der Optimierungen auf das CUDA- und OpenCL-Programm

Die CUDA-Diagramme der GTX 480 (Abbildung 122, Abbildung 125, Abbildung 127, Abbildung 129, Abbildung 130, Abbildung 132 und Abbildung 134) in diesem Kapitel sind identisch mit den CUDA-Diagrammen (Abbildung 101, Abbildung 102, Abbildung 104, Abbildung 105, Abbildung 106, Abbildung 107 und Abbildung 109) im Kapitel 8.3.2 und werden für den direkten Vergleich mit den OpenCL-Diagrammen der GTX 480 nochmals abgebildet.

Nach eigenen Messungen ist Constant Memory bei OpenCL im Gegensatz zu CUDA langsamer bzw. gleich schnell wie Global Memory. Zum Beispiel beim Vergleichen des Gesamtspeedups des Global-Memory-Programms ohne Optimierungen mit dem Gesamtspeedup des Constant-Memory-Programms ohne Optimierungen erhält man bei $N=800$, $M=800$ und bei der GTX 480 folgende Prozentwerte. Bei CUDA ist das Constant-Memory-Programm um 5,5% schneller als das Global-Memory-Programm. Im Gegensatz dazu ist bei OpenCL das Global-Memory-Programm um 44% schneller als das Constant-Memory-Programm. Aus diesem Grund wird beim Vergleichen des Speedups des CUDA-Programms mit dem Speedup des OpenCL-Programms bei allen Optimierungen bei OpenCL immer Global Memory benützt. Bei CUDA hingegen wird Constant Memory verwendet.

Bei der Angabe der Wirksamkeit der Optimierungen bei CUDA und OpenCL wird ebenfalls bei CUDA das Constant-Memory-Programm und bei OpenCL das Global-Memory-Programm benützt, jedoch bei der Optimierung `rsqrt` wird bei OpenCL zusätzlich die Wirksamkeit der Optimierung beim Constant-Memory-Programm und bei der Optimierung „Lokale float4-Variable“ wird bei CUDA zusätzlich die Wirksamkeit der Optimierung beim Global-Memory-Programm ermittelt. Nach eigenen Messungen führt die Optimierung `rsqrt` bei OpenCL bei der GTX 480 nur beim Constant-Memory-Programm zu einer Veränderung des Speedups und die Optimierung „Lokale float4-Variable“ hat bei CUDA bei der GTX 480 nur einen Einfluss beim Global-Memory-Programm.

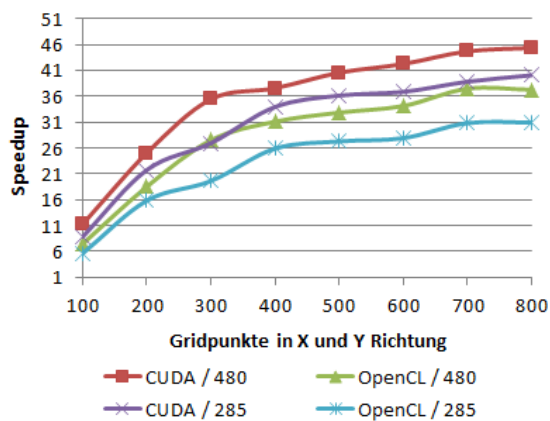


Abbildung 120: Ohne Compiler-Optimierungsflags bei CUDA und OpenCL

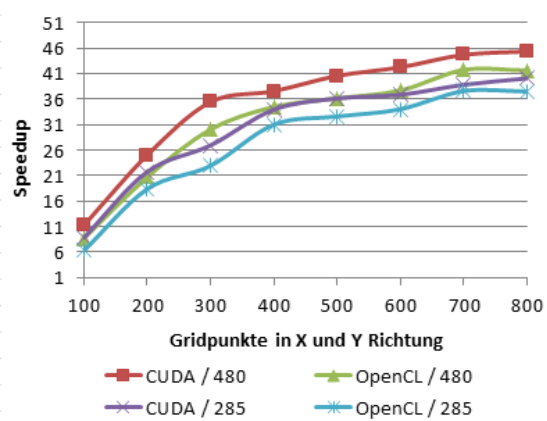


Abbildung 121: Mit Compiler-Optimierungsflags bei OpenCL

Wenn man den erreichten Gesamtspeedup auf der Fermi-GPU (GTX 480) mit der älteren GT200-GPU (GTX 285) vergleicht, erkennt man, dass der Unterschied zwischen CUDA (Constant-Memory-Programm mit allen Optimierungen: `rsqrtf`, `float4`, `dz*dz` auf CPU vorberechnet, 4 Gridpunkte in X-Richtung pro Thread, Memory Coalescing) und OpenCL (Global-Memory-Programm mit allen Optimierungen: `native_rsqrt`, lokale `float4`-Variable, `dz*dz` auf CPU vorberechnet, 4 Gridpunkte in X-Richtung pro Thread, Memory Coalescing) bei beiden Architekturen in etwa gleich groß ist (siehe Abbildung 120). Bei $N=800$, $M=800$ ist das CUDA-Programm bei der GTX 480 (Fermi) um 20% und bei GTX 285 (GT200) um 29,5% schneller als das OpenCL-Programm (ohne Compiler-Optimierungsflags).

Mit Compiler-Optimierungsflags beim OpenCL-Programm rückt OpenCL sehr nahe an CUDA heran (siehe Abbildung 121). Bei N=800, M=800 ist das CUDA-Programm bei der GTX 480 (Fermi) um 7,4% und bei der GTX 285 (GT200) um 6,9% schneller als das OpenCL-Programm mit Compiler-Optimierungsflags.

Optimierung „rsqrtf“

Die Optimierung „rsqrtf“ wirkt sich bei CUDA und OpenCL bei Verwendung von Constant Memory positiv aus (siehe Abbildung 122 und Abbildung 124). Bei N=800, M=800 und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 50,3% und bei OpenCL um 40,8%. Der Kernelspeedup verbessert sich bei CUDA um 57,3% und bei OpenCL um 50,4% (GTX 480, N=800, M=800). Bei Verwendung von Global Memory ergibt sich bei OpenCL kein Speedup (siehe Abbildung 123, bei GTX 480, N=800, M=800: Gesamtspeedup vor der Optimierung um 1,2% höher und Kernelspeedup vor der Optimierung um 0,03% höher).

Bei CUDA gibt es rsqrtf() für Single-Precision und rsqrt() für Double-Precision [5]. Hingegen gibt es bei OpenCL nur rsqrt() [6].

Mit der Optimierung rsqrtf bzw. rsqrt ist beim CUDA-Programm der Gesamtspeedup um 37,7% und der Kernelspeedup um 36,9% höher als beim OpenCL-Programm (GTX 480, N=800, M=800).

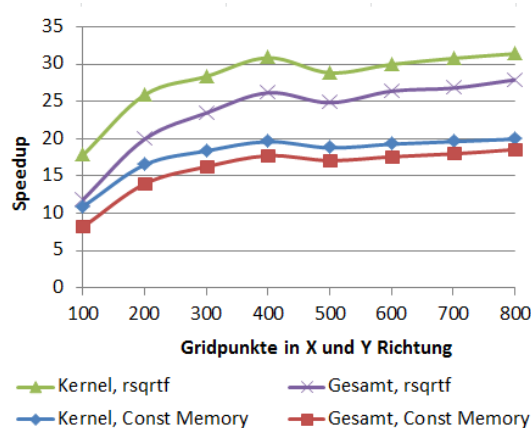


Abbildung 122: CUDA, GTX 480, Constant Memory

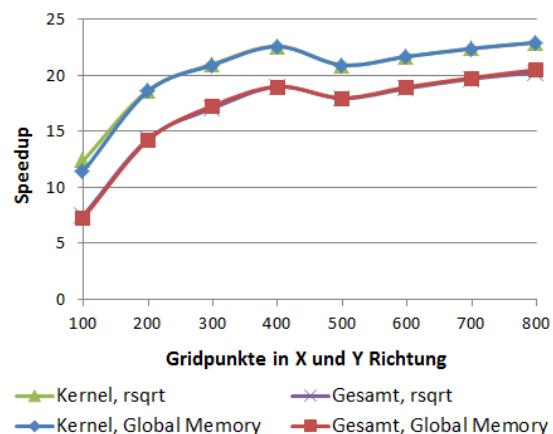


Abbildung 123: OpenCL, GTX 480, Global Memory

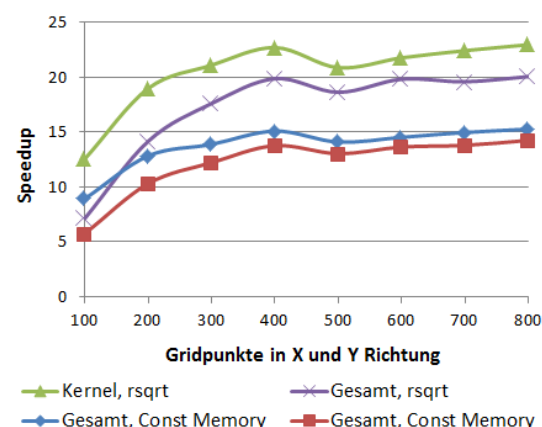


Abbildung 124: OpenCL, GTX 480, Constant Memory

Optimierung „float4“

Die Optimierung „float4“ hat bei CUDA und OpenCL keine Auswirkung (siehe Abbildung 125 und Abbildung 126, bei GTX 480, N=800, M=800: Gesamtspeedup vor der Optimierung bei CUDA um 0,08% und bei OpenCL um 0,6% höher und Kernelspeedup vor der Optimierung bei CUDA um 0,08% und bei OpenCL um 0,08% höher).

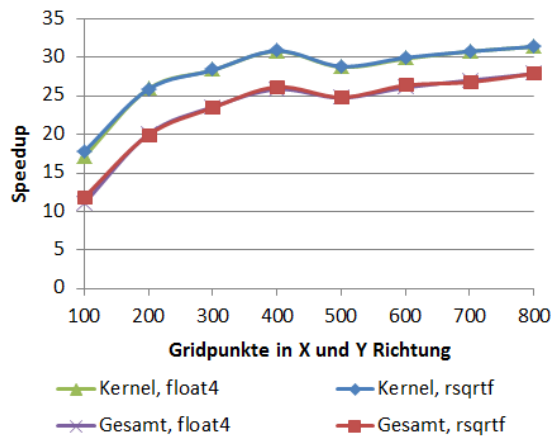


Abbildung 125: CUDA, GTX 480

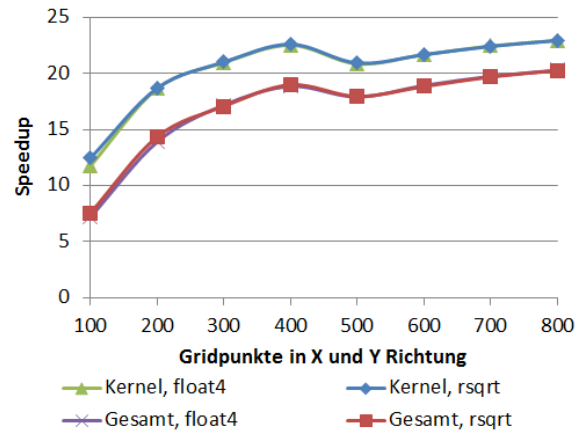


Abbildung 126: OpenCL, GTX 480

Optimierung „Lokale float4-Variable“

Die Optimierung „Lokale float4-Variable“ wirkt sich bei CUDA und OpenCL bei Verwendung von Global Memory positiv aus (siehe Abbildung 128 und Abbildung 129). Bei N=800, M=800 und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 8,6% und bei OpenCL um 6,8%. Der Kernelspeedup verbessert sich bei CUDA um 9,5% und bei OpenCL um 8,2% (GTX 480, N=800, M=800). Bei Constant Memory hat die Optimierung keine Auswirkung (siehe Abbildung 127, bei GTX 480, N=800, M=800: Verbesserung des Gesamtspeedups bei CUDA um 0,5% und Kernelspeedup vor der Optimierung bei CUDA um 0,04% höher).

Mit den Optimierungen rsqrtf bzw. rsqrt und lokale float4-Variable ist beim CUDA-Programm der Gesamtspeedup um 28,6% und der Kernelspeedup um 26,2% höher als beim OpenCL-Programm (GTX 480, N=800, M=800).

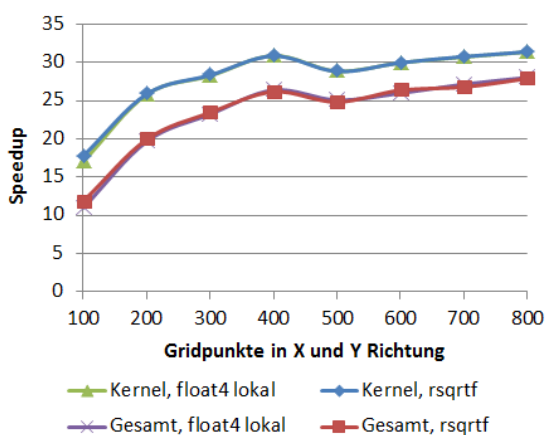


Abbildung 127: CUDA (Const Memory), GTX 480

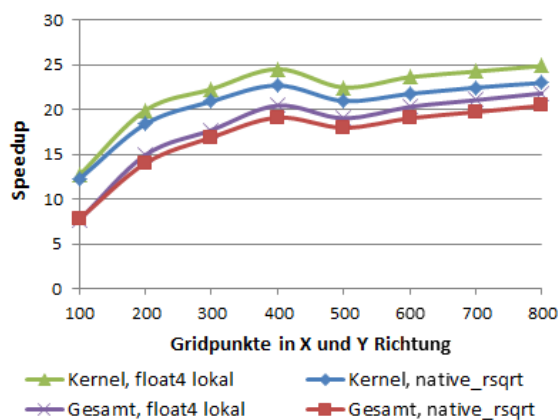


Abbildung 128: OpenCL (Global Memory), GTX 480

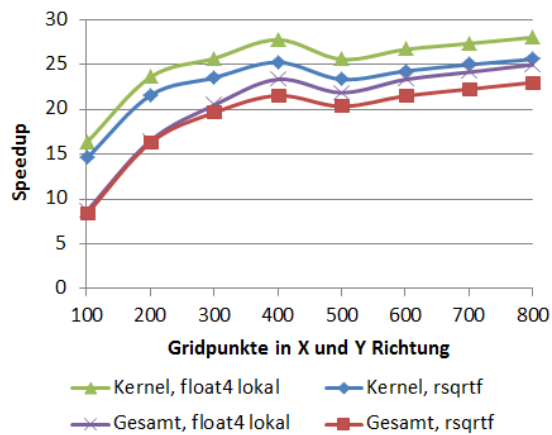


Abbildung 129: CUDA (Global Memory), GTX 480

Optimierung „Vorbereitung von $dz*dz$ auf CPU“

Die Optimierung „ $dz*dz$ auf CPU vorberechnet“ ermöglicht bei CUDA und OpenCL einen kleinen Speedup (siehe Abbildung 130 und Abbildung 131). Bei $N=800$, $M=800$ und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 16,4% und bei OpenCL um 11,1%. Der Kernelspeedup verbessert sich bei CUDA um 20% und bei OpenCL um 12,6% (GTX 480, $N=800$, $M=800$).

Mit den Optimierungen rsqrtf bzw. rsqrt, float4 bzw. lokale float4-Variable und „Vorbereitung von $dz*dz$ auf CPU“ ist beim CUDA-Programm der Gesamtspeedup um 33,9% und der Kernelspeedup um 34,6% höher als beim OpenCL-Programm (GTX 480, $N=800$, $M=800$).

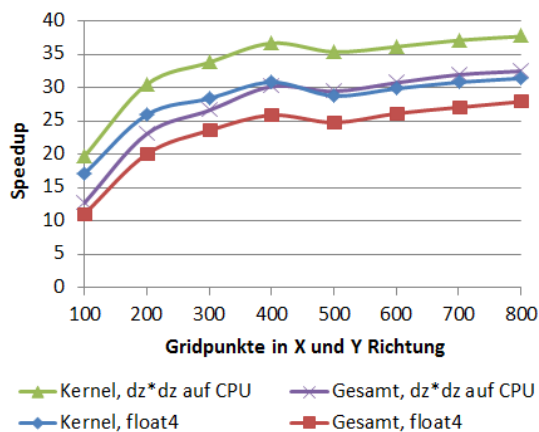


Abbildung 130: CUDA, GTX 480

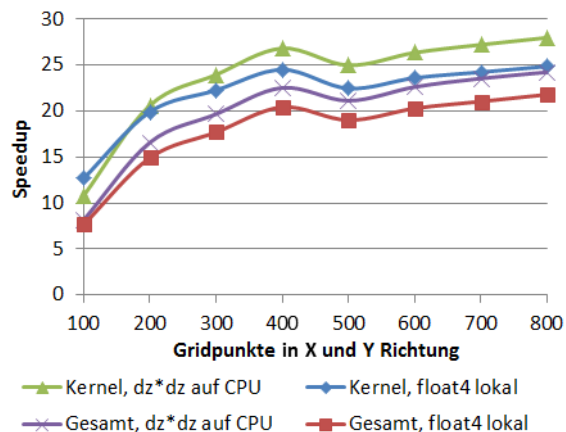


Abbildung 131: OpenCL, GTX 480

Optimierung „4 Gridpunkte in X-Richtung pro Thread“

Die Optimierung „4 Gridpunkte in X-Richtung pro Thread“ wirkt sich bei CUDA und OpenCL positiv aus (siehe Abbildung 132 und Abbildung 133). Bei $N=800$, $M=800$ und der GTX 480 verbessert sich der Gesamtspeedup bei CUDA um 39,6% und bei OpenCL um 54,3%. Der Kernelspeedup verbessert sich bei CUDA um 50% und bei OpenCL um 70,3% (GTX 480, $N=800$, $M=800$).

Mit den Optimierungen rsqrtf bzw. rsqrt, float4 bzw. lokale float4-Variable, „Vorbereitung von $dz*dz$ auf CPU“ und „4 Gridpunkte in X-Richtung pro Thread“ ist beim CUDA-Programm der Gesamtspeedup um 21,2% und der Kernelspeedup um 18,4% höher als beim OpenCL-Programm (GTX 480, $N=800$, $M=800$).

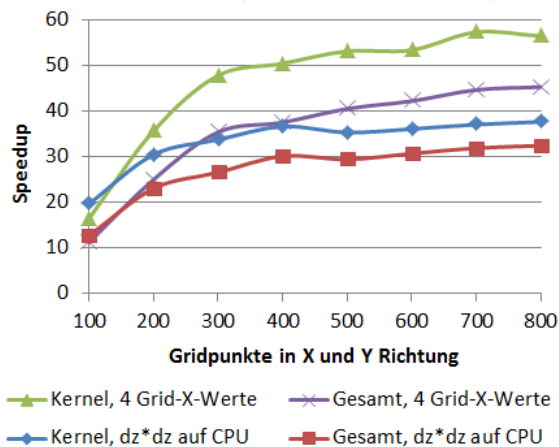


Abbildung 132: CUDA, GTX 480

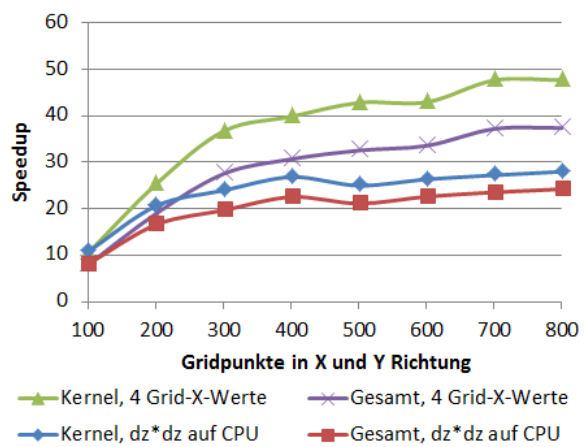


Abbildung 133: OpenCL, GTX 480

Optimierung „Memory Coalescing“

Die Optimierung „Memory Coalescing“ hat keine Auswirkung auf die Performance bei CUDA und OpenCL (siehe Abbildung 134 und Abbildung 135, bei GTX 480, N=800, M=800: Gesamtspeedup vor der Optimierung bei CUDA um 1,4% und bei OpenCL um 0,44% höher und Kernelspeedup vor der Optimierung bei CUDA um 0,05% und bei OpenCL um 0,4% höher).

Mit allen Optimierungen (rsqrtf bzw. rsqrt, float4 bzw. lokale float4-Variable, „Vorberechnung von $dz \cdot dz$ auf CPU“, „4 Gridpunkte in X-Richtung pro Thread“ und „Memory Coalescing“) ist beim CUDA-Programm der Gesamtspeedup um 20% und der Kernelspeedup um 18,8% höher als beim OpenCL-Programm (GTX 480, N=800, M=800).

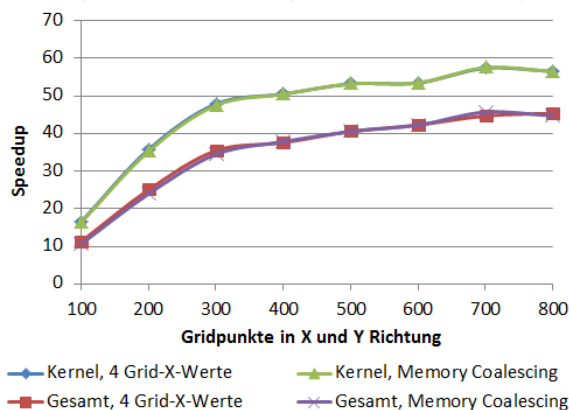


Abbildung 134: CUDA, GTX 480

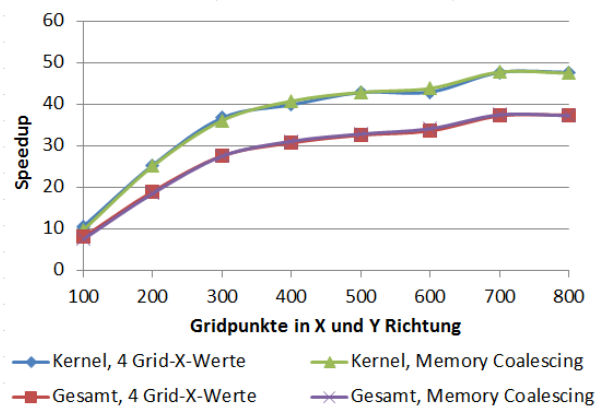


Abbildung 135: OpenCL, GTX 480

8.4.2 Fermi im Vergleich zu der Vorgänger-Architektur GT200

In diesem Kapitel wird für die Fermi-Architektur die GTX 480 und für die Vorgängerarchitektur GT200 die GTX 285 herangezogen. Außerdem wird das CUDA-Programm und die Problemgröße N=800, M=800 verwendet.

In Abbildung 136 (Gesamt-Speedup) und Abbildung 137 (Kernel-Speedup) wird der Speedup zwischen der Fermi-Architektur und der Vorgängerarchitektur GT200 bei dem CUDA-Const-Memory-Programm mit allen Optimierungen und dem CUDA-Global-Memory-Programm ohne Optimierung verglichen. Im Paper zur Masterarbeit [14] wurde in der Gesamt-Speedup-Abbildung bei dem Global-Memory-Programm die Optimierung rsqrtf verwendet (siehe Abbildung 138). In der Masterarbeit wird bei der Basisversion diese Optimierung weggelassen.

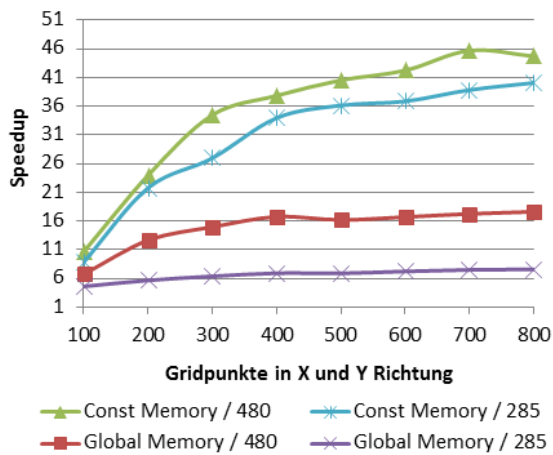


Abbildung 136: Gesamt-Speedup, CUDA (Global Memory ohne Optimierungen und Const Memory mit allen Optimierungen)

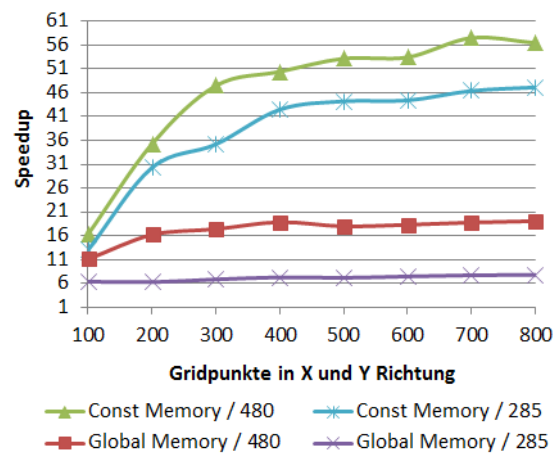


Abbildung 137: Kernel-Speedup, CUDA (Global Memory ohne Optimierungen und Const Memory mit allen Optimierungen)

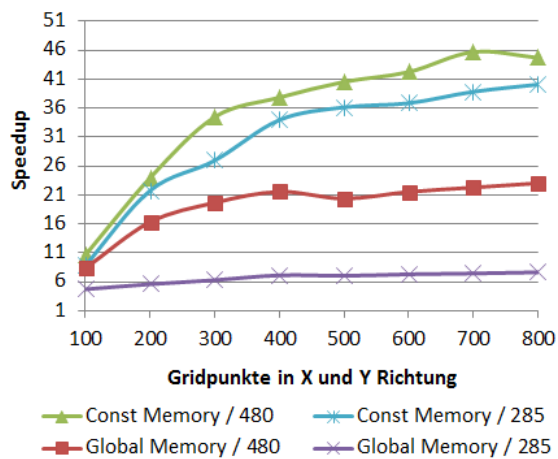


Abbildung 138: Gesamt-Speedup, CUDA (Global Memory mit Optimierung rsqrtf und Const Memory mit allen Optimierungen)

Beim Global-Memory-Programm ohne Optimierungen erreicht man mit Fermi 133% mehr Gesamtspeedup und 145% mehr Kernelspeedup im Vergleich zu GT200 (siehe Abbildung 136 und Abbildung 137). Der Grund für den starken Unterschied zwischen Fermi und GT200 ist, dass die Fermi-Architektur eine Cache-Hierarchie für den Global Memory besitzt und die konstanten Atom-Werte, sobald sie sich einmal im L2-Cache bzw. L1-Cache befinden, nicht mehr aus dem Global Memory gelesen werden müssen.

Beim Const-Memory-Programm mit allen Optimierungen erreicht man mit Fermi 12% mehr Gesamtspeedup und 20% mehr Kernelspeedup im Vergleich zu GT200 (siehe Abbildung 136 und Abbildung 137).

Die Unterschiede zwischen Fermi und GT200 bei der Wirksamkeit der einzelnen Optimierungen kann man durch das Vergleichen der Abbildung 139 mit der Abbildung 140 erkennen. Die zwei Diagramme (Abbildung 139 und Abbildung 140) sind identisch mit den Diagrammen (Abbildung 112 und Abbildung 114) im Kapitel 8.3.3.

Durch das Verwenden von Constant Memory gibt es eine Verbesserung des Gesamtspeedups bei Fermi um 5,5% und bei GT200 um 106%. Der Grund für den starken Unterschied ist, dass mit Fermi eine Cache-Hierarchie für den Global-Memory eingeführt wurde.

Durch die Optimierung rsqrtf verbessert sich der Gesamtspeedup bei Fermi um 50% und bei GT200 um 44%. Die Optimierungen float4 und „float4 lokal“ führen bei Fermi zu keiner Verbesserung. Bei GT200 gibt es bei float4 eine Verschlechterung von 4,8% und bei „float4 lokal“ eine Verschlechterung von 5%. Die Optimierung „Vorberechnung von dz*dz auf CPU“ verbessert den Gesamtspeedup bei Fermi um 16,3% und bei GT200 um 3,6%. Durch die Optimierung „Berechnung von 4 Gridpunkten in X-Richtung pro Thread“ erhöht sich der Gesamtspeedup bei Fermi um 40% und bei GT200 um 66%. Die Optimierung „Memory Coalescing“ führt bei Fermi zu keiner Verbesserung und bei GT200 zu einer Verbesserung des Gesamtspeedups von 3,5%.

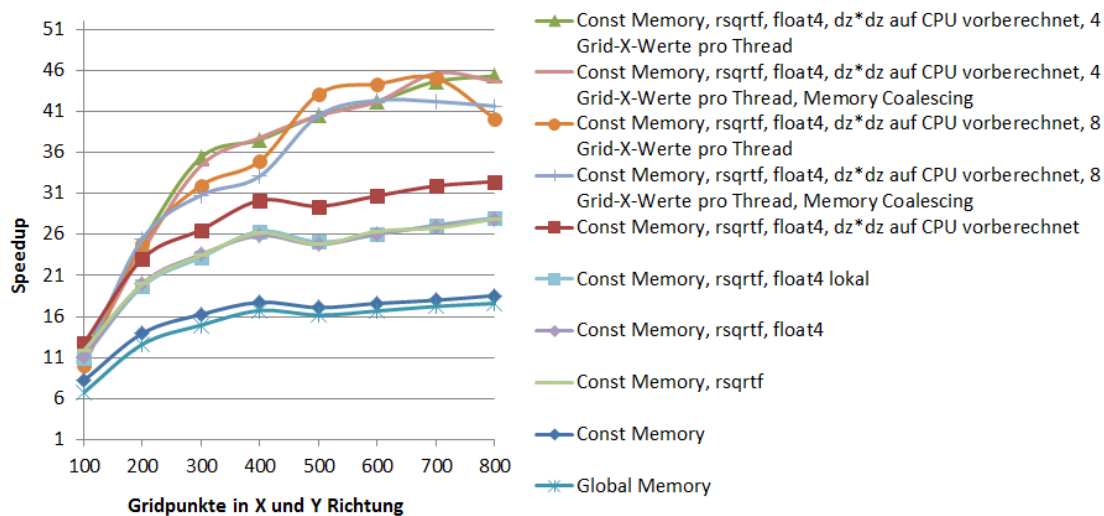


Abbildung 139: Fermi-GPU GTX 480, CUDA, Gesamtspeedup

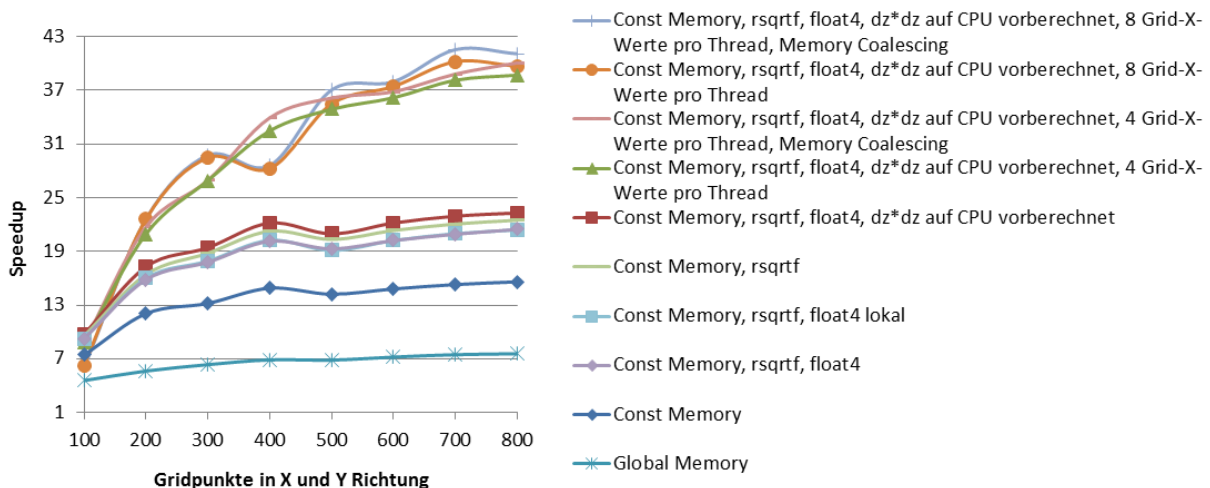


Abbildung 140: GT200-GPU GTX 285, CUDA, Gesamtspeedup

Beim Vergleichen der Gesamtspeedups zwischen Fermi und GT200 bei den einzelnen Optimierungen erhält man folgende Prozentzahlen. Beim Basis-Code ohne Optimierungen ist Fermi um 132,5% schneller als GT200. Mit der Verwendung des Constant Memory ist Fermi nur mehr um 19% schneller als GT200. Bei der Optimierung rsqrtf ist Fermi um 24% schneller als GT200. Mit „float4“ ist Fermi 30% schneller als GT200 und mit „float4 lokal“ ist Fermi 31% schneller als GT200. Bei der weiteren Optimierung „Vorberechnung von dz*dz auf CPU“ ist Fermi um 39% schneller als GT200. Mit der

Optimierung „Berechnung von 4 Gridpunkten in X-Richtung pro Thread“ ist Fermi 17% schneller als GT200. Bei der letzten Optimierung „Memory Coalescing“ ist Fermi nur mehr 11,6% schneller als GT200.

8.4.3 Vergleich Code ohne Optimierung mit Code mit Optimierungen

Bei der Berechnung von 800 Gridpunkten in X- und Y-Richtung auf der GTX 480 konnte der Kernel-Speedup von 19,1 (CUDA-Programm ohne Optimierung) auf 56,5 (CUDA-Programm mit Optimierungen) und der Gesamt-Speedup von 17,6 (CUDA-Programm ohne Optimierung) auf 45,3 (CUDA-Programm mit Optimierungen) erhöht werden (siehe Abbildung 141). Die Kernellaufzeit konnte um den Faktor 3 und die Gesamtlaufzeit um den Faktor 2,6 verbessert werden. Dieser Unterschied verursacht die Lücke zwischen den zwei oberen Kurven in Abbildung 141.

Wenn man sich die Laufzeit genau ansieht, erkennt man, dass die Optimierungen sich nur auf den Kernel auswirken. Die Gesamtlaufzeit ohne Kernellaufzeit bleibt gleich. Bei $N=800$, $M=800$, GTX 480 und CUDA wurden folgende Werte gemessen. Ohne Optimierung beträgt die Gesamtlaufzeit 0,0260604 Sekunden, die Kernellaufzeit 0,0240414 und die Differenz 0,002 Sekunden. Mit Optimierungen beträgt die Gesamtlaufzeit 0,0101191, die Kernellaufzeit 0,0081191 und die Differenz 0,002 Sekunden.

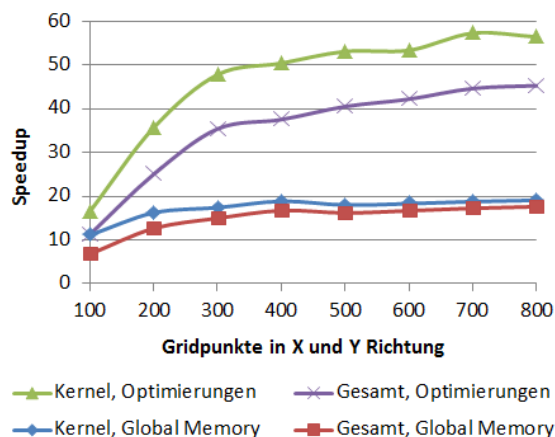


Abbildung 141: Verbesserung des Speedups bei Verwendung aller Optimierungen (CUDA, GTX 480)

8.4.4 Wie viel wurde von der maximalen Leistung der GPU erreicht?

Der GFLOP/s-Wert des DCS-Programms wird über die Formel "Problemgröße bis zur innersten Schleife*Anzahl der Operationen in der innersten Schleife/ 10^9 /Laufzeit in Sekunden" berechnet. Wenn die Gesamtlaufzeit betrachtet wird, kann bei 800 Gridpunkten in X- und Y-Richtung 32% der maximalen theoretischen Leistung der GPU ($=1345$ GFLOP/s [5] [21]) erreicht werden (siehe Abbildung 142). Wenn man nur die Kernellaufzeit heranzieht, erreicht man 40% der Peak-Leistung der GPU. Da die Operationen im Kernel nicht alle FMA-Operationen sind und die Peak-Leistung nur mit FMA-Operationen erreichbar ist, ist 40% ein sehr guter Wert. Dieser hohe Wert zeigt, dass der DCS Algorithmus sehr gut für GPUs geeignet ist.

Die Operation rsqrtf wird bei der Berechnung des GFLOP/s-Wertes als 1 Operation gezählt.

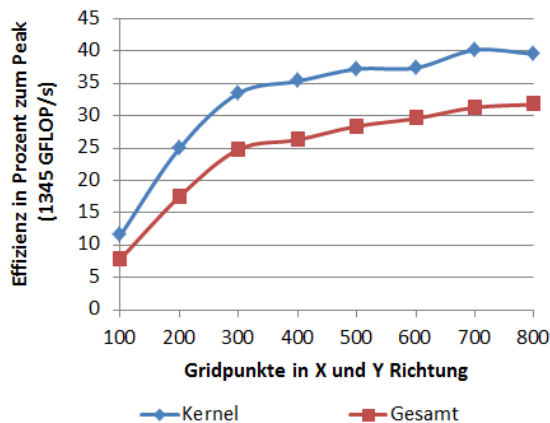


Abbildung 142: Erreichte Effizienz bei der Leistung (CUDA, GTX 480)

8.4.5 Performance bei Double-Precision (double)

Im Kapitel 3.3.13 befinden sich die notwendigen Schritte, damit Double-Precision im CUDA- und OpenCL-Programm verwendet werden kann.

In der Masterarbeit wurden drei unterschiedliche GPUs (Tesla C2050, GTX 480, GTX 285) für den Vergleich des Single-Precision(=Datentyp float)-Speedups mit dem Double-Precision(=Datentyp double)-Speedup herangezogen. Wenn man die Peak-Leistung betrachtet, ist bei der Tesla C2050 das Verhältnis zwischen Double-Precision und Single-Precision am kleinsten. Hier beträgt die Double-Precision-Peak-Leistung die Hälfte der Single-Precision-Peak-Leistung, also 515 GFLOP/s [25]. Bei der GTX 480 ist es 1/8 (=168 GFLOP/s) [19] [21] und bei der GTX 285 ist es 1/12 (=88 GFLOP/s) [16] [50] der Single-Precision-Peak-Leistung.

In Abbildung 143 werden die Gesamt-Speedups des CUDA-Programms mit allen Optimierungen bei Single-Precision (Datentyp float) und Double-Precision (Datentyp double) dargestellt. Bei N=800 und M=800 erzielt man dabei folgende Speedup-Werte. Bei der GTX 285 beträgt der Single-Precision-Speedup 40 und der Double-Precision-Speedup 2,6. Bei der GTX 480 beträgt der Single-Precision-Speedup 44,7 und der Double-Precision-Speedup 5,6. Bei der Tesla C2050 beträgt der Single-Precision-Speedup 33,9 und der Double-Precision-Speedup 6,8. Das Verhältnis zwischen Single-Precision-Speedup und Double-Precision-Speedup beträgt somit bei der GTX 285 1/15, bei der GTX 480 1/8 und bei der Tesla C2050 1/5. Bei der GTX 480 entspricht das Verhältnis der Speedupwerte genau dem Verhältnis der Peak-Leistungswerte.

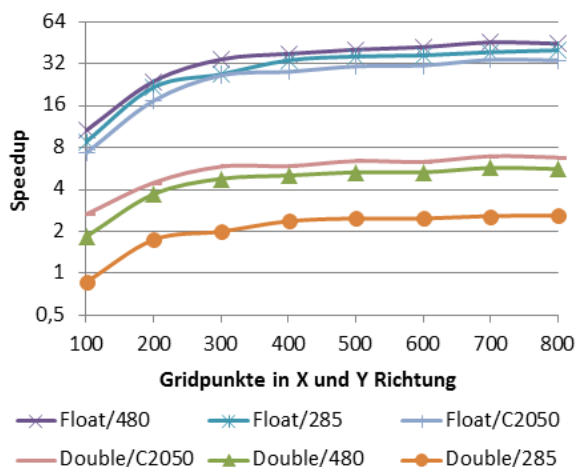


Abbildung 143: Speedup bei Double-Precision

Interessant ist, dass mit CUDA Toolkit 4.0 die Double-Precision-Performance gegenüber Toolkit 3.2 verbessert wurde. Bei eigenen Messungen beim DCS-Algorithmus mit der GTX 480 wurde festgestellt, dass sich das Verhältnis zwischen Double-Precision-Speedup und Single-Precision-Speedup bei Verwendung von Toolkit 4.0 anstatt von Toolkit 3.2 von 1/10 auf 1/8 verbessert. Auch bei der GTX 285 konnte durch eigene Messungen am DCS-Algorithmus eine Verbesserung von 1/19 auf 1/15 festgestellt werden. Bei der Tesla C2050 kann eine solche Verbesserung ebenfalls erwartet werden, sobald die Maschine, auf der sich die GPU befindet, auf Toolkit 4.0 aktualisiert wird.

9 Interpretation der GPU-Laufzeitmessungen

9.1 Kann Constant Memory durch das Caching von Global Memory ersetzt werden?

Caching von Global Memory kann Constant Memory nicht ersetzen, aber der Cache ermöglicht es, dass man mit straightforward-Kernels deutlich bessere Laufzeiten erzielt. In Abbildung 144 (Gesamtspeedups vom DCS-CUDA-Programm ohne Optimierungen) sieht man, dass der Cache für den Global Memory bei der GTX 480 die Verwendung des Constant Memory fast unnötig macht. Die Speedupkurven liegen fast übereinander. Hingegen bei der GTX 285, die keinen Cache für den Global Memory hat, kann durch Constant Memory eine deutliche Performance-Verbesserung erzielt werden. Hier ist die Verwendung des Constant Memory sehr wichtig.

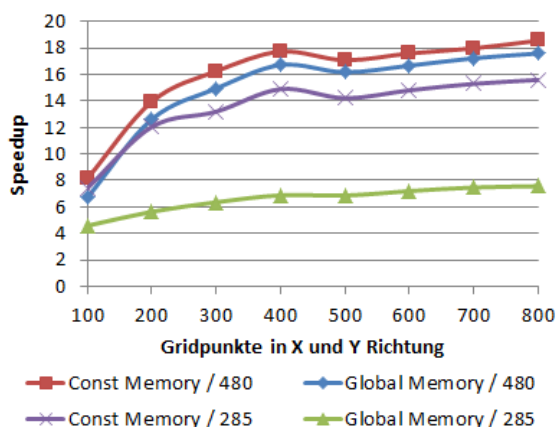


Abbildung 144: Speedup durch die Verwendung von Constant Memory im Vergleich zur Verwendung von Global Memory mit Cache bei GTX 480 und zur Verwendung von Global Memory ohne Cache bei GTX 285 (Gesamtspeedup, DCS, CUDA)

9.2 Kann Shared Memory durch das Caching von Global Memory ersetzt werden?

In Abbildung 145 (Speedups vom NW-CUDA-Programm mit optimaler Blockgröße) sieht man, dass der Cache für den Global Memory bei der GTX 480 den Speedup des Global-Memory-Programms nahe an den Speedup des Shared-Memory-Programms bringen kann. Der optimale Speedup wird jedoch weiterhin durch die Verwendung von Shared Memory erzielt. Bei der GTX 285, die keinen Cache für den Global Memory hat, kann durch Shared Memory eine deutliche Performance-Verbesserung erzielt werden. Hier ist die Verwendung des Shared Memory sehr wichtig.

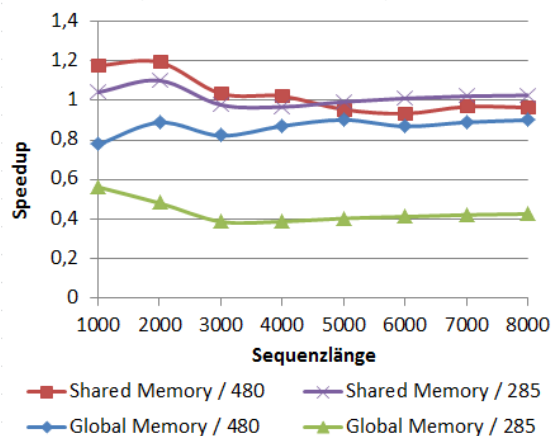


Abbildung 145: Speedup durch die Verwendung von Shared Memory im Vergleich zur Verwendung von Global Memory mit Cache bei GTX 480 und zur Verwendung von Global Memory ohne Cache bei GTX 285 (Gesamtspeedup, NW, CUDA, optimale Blockgröße)

9.3 Kann Caching von Global Memory die Optimierung Memory Coalescing überflüssig machen?

In den folgenden Abbildungen (Abbildung 146 und Abbildung 147) wird der Speedup von der Version mit Memory Coalescing zu der Version ohne Memory Coalescing dargestellt. Für DCS wird die Constant-Memory-Version und für NW die Global-Memory-Version verwendet. Bei beiden Versionen werden alle Optimierungen und CUDA verwendet.

Man sieht in Abbildung 146 (Gesamtspeedup) und Abbildung 147 (Kernelspeedup) beim Vergleichen der Kurve von GTX 480 und GTX 285 bei NW, dass der Cache der GTX 480 den Vorteil der Verwendung der Optimierung Memory Coalescing stark reduziert. Jedoch befindet sich die Kurve von GTX 480 nicht bei 1. Somit macht das Caching von Global Memory die Optimierung Memory Coalescing nicht komplett überflüssig. Um den schnellsten Code zu erreichen, ist weiterhin die Berücksichtigung von Memory Coalescing notwendig.

Beim DCS-Constant-Memory-Programm hat Memory Coalescing keinen Einfluss auf den Speedup, da die Anzahl der Atome auf 1000 festgelegt wurde und somit die Inputdaten (Atome) komplett in den Constant Memory passen (siehe Erklärung im Kapitel 8.3.2.5). Aber auch beim Global-Memory-Programm mit allen Optimierungen ist der Speedup durch Memory Coalescing mit 1,8% bei der GTX 285 (OpenCL, N=800, M=800, Anzahl von Atomen=1000) und 0,5% bei der GTX 480 (OpenCL, N=800, M=800, Anzahl von Atomen=1000) bei 8 Gridpunkten pro Thread sehr gering. Bei 4 Gridpunkten pro Thread hat man sogar einen Slowdown von 0,2% bei der GTX 285 (OpenCL, N=800, M=800, Anzahl von Atomen=1000) und 0,4% bei der GTX 480 (OpenCL, N=800, M=800, Anzahl von Atomen=1000). Der Grund bei Global Memory für den geringen Laufzeitunterschied zwischen dem Code mit Memory Coalescing und dem Code ohne Memory Coalescing ist, dass die Änderung im Code nicht den Lesezugriff auf den Global Memory betrifft, sondern nur das Schreiben der berechneten Gridpunkte in den Global Memory. Da alle Threads eines Warps (=32 Threads) bei jedem gleichzeitigen Lesezugriff auf den Global Memory dasselbe Atom aus dem Atomarray im Global Memory auslesen, also alle 32 Threads den identischen Global-Memory-Lesezugriff durchführen, muss Memory Coalescing nicht beachtet werden. Die Hardware führt in diesem Fall automatisch nur einen Global-Memory-Zugriff statt 32 Global-Memory-Zugriffe pro Warp durch.

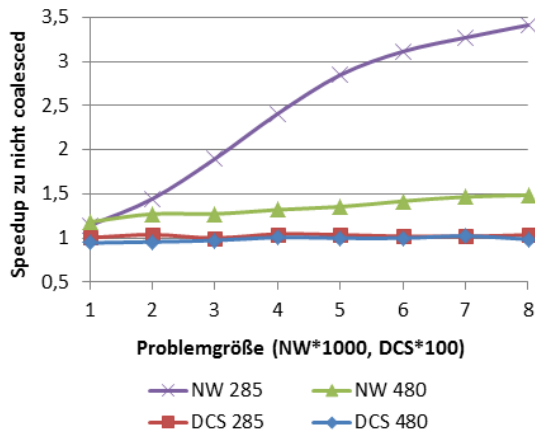


Abbildung 146: Gesamtspeedup (CUDA)

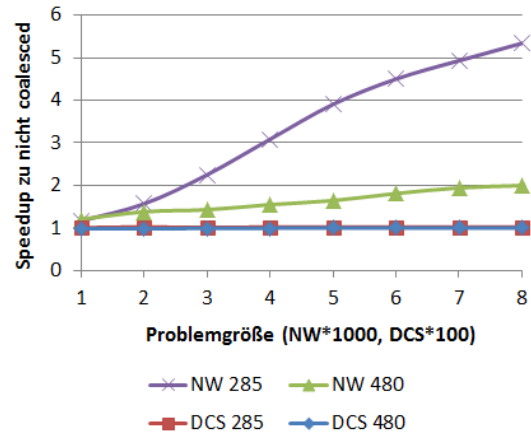


Abbildung 147: Kernelspeedup (CUDA)

Da bei der Global-Memory-Version des NW die Optimierung Memory Coalescing nach der Optimierung Short-Datentyp erfolgt, ist automatisch die Optimierung Short-Datentyp enthalten. Beim Short-Datentyp wird im Vergleich zum Int-Datentyp die Übertragungsgröße halbiert und dadurch wird die Kernellaufzeit in der Gesamtlaufzeit sehr dominant. Ohne Memory Coalescing beträgt die Kernellaufzeit 63% (GTX 480) bzw. 87% (GTX 285) der Gesamtlaufzeit (N=8000, M=8000). Memory Coalescing verbessert den Kernelspeedup bei der GTX 285 um den Faktor 5.3, wodurch sich der Gesamtspeedup um den Faktor 3.4 verbessert (siehe N=8000, M=8000 in Abbildung 146 und Abbildung 147).

9.4 Auswirkung des vergrößerten Shared Memory

Durch den 3 Mal größeren Shared Memory (48 KByte statt 16 KByte) pro Multiprozessor bei Fermi kann die Blockgröße beim Shared-Memory-Programm des NW-Algorithmus von 31 auf 54 erhöht werden. Da die Blockgröße die Anzahl der Threads festlegt, kann dadurch die Parallelität erhöht und das Programm schneller gemacht werden (siehe Abbildung 148).

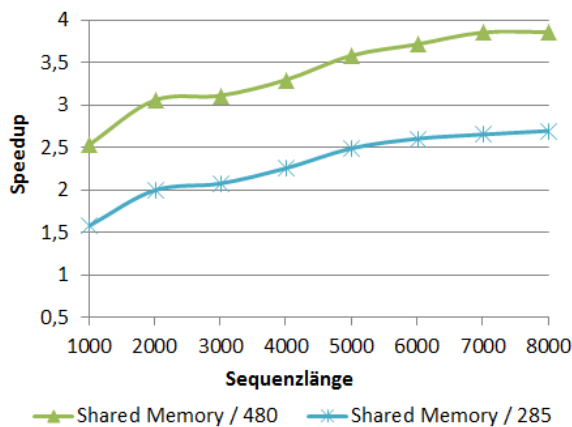


Abbildung 148: NW-Algorithmus (CUDA, alle Optimierungen)

9.5 Auswirkung der höheren Double-Precision-Peak-Leistung

Durch die zirka 2 Mal bzw. zirka 6 Mal höhere Double-Precision-Peak-Leistung (GTX 285 (GT200) mit 88 GFLOP/s [16] [50], GTX 480 (Fermi) mit 168 GFLOP/s [19] [21] und Tesla C2050 (Fermi) mit 515 GFLOP/s [25]) bei Fermi gegenüber der Vorgängerarchitektur GT200 kann bei Verwendung des Datentyps double beim DCS-Algorithmus ein deutlich höherer Speedup erreicht werden (siehe Abbildung 149).

Da die Double-Precision-Peak-Leistung stärker als die Single-Precision-Peak-Leistung erhöht wurde, ist der Unterschied zwischen Double-Precision-Peak-Leistung und Single-Precision-Peak-Leistung reduziert worden. Bei der GTX 285 (GT200) beträgt die Peak-Leistung bei Double-Precision 1/12 [16] [50] der Single-Precision-Peak-Leistung, bei der GTX 480 (Fermi) 1/8 [19] [21] und bei der Tesla C2050 (Fermi) nur mehr 1/2 [25]. Auch diese Veränderung lässt sich in der Abbildung 149 ablesen. Während die Single-Precision-Speedupwerte zwischen Fermi (GTX 480 und Tesla C2050) und GT200 (GTX 285) fast gleich sind, gibt es bei den Double-Precision-Speedupwerten einen deutlichen Unterschied zwischen Fermi und GT200.

Auf die genauen Speedupwerte der Abbildung 149 bei N=800 und M=800 wird im Kapitel 8.4.5 eingegangen. Die Abbildung 149 ist identisch mit der Abbildung 143 im Kapitel 8.4.5.

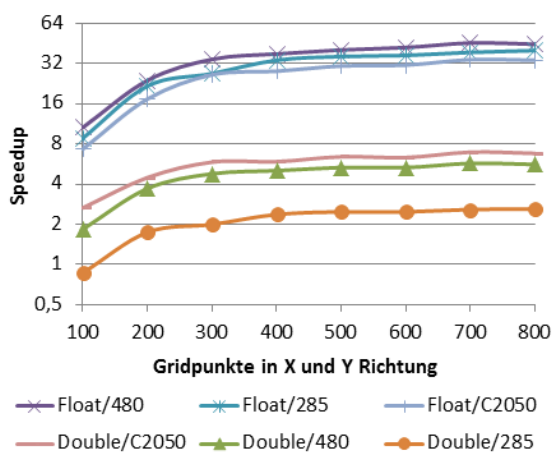


Abbildung 149: DCS-Algorithmus (Gesamtspeedup, CUDA, alle Optimierungen)

9.6 Ist der Algorithmus durch die Bandbreite des Global Memory limitiert?

Die Analyse, ob der Algorithmus durch die Bandbreite des Global Memory limitiert ist, wird mit Hilfe des CGMA-Wertes durchgeführt. CGMA steht für Compute to Global Memory Access und drückt das Verhältnis zwischen der Anzahl der Operationen und der Anzahl der Global-Memory-Zugriffe aus (siehe Kapitel 2.2.3.1) [1].

Im Folgenden werden die zwei Algorithmen NW und DCS auf eine mögliche Bandbreitenlimitierung untersucht. Dabei wird mit der Formel „Bandbreite des Global Memory/Datentypgröße*CGMA“ der GFLOP/s-Wert errechnet, ab dem die maximale Bandbreite des Global Memory erreicht wird [1]. Der mit CGMA errechnete GFLOP/s-Wert wird mit dem Peak-GFLOP/s-Wert der GPU verglichen [1]. Wenn der mit CGMA errechnete GFLOP/s-Wert kleiner als der Peak-GFLOP/s-Wert der GPU ist, dann ist der Algorithmus durch die Bandbreite des Global Memory limitiert [1].

Es wird außerdem der mit CGMA errechnete GFLOP/s-Wert mit dem über die Laufzeit errechneten GFLOP/s-Wert verglichen. Der mit der Laufzeit errechnete GFLOP/s-Wert wird über die Formel „Problemgröße bis zur innersten Schleife*Anzahl der Operationen in der innersten Schleife/10⁹/Laufzeit des Kernels“ ermittelt.

Jede Operation, egal wie aufwendig, wird als 1 Operation gezählt und dadurch wird bei dem über die Laufzeit errechneten GFLOP/s-Wert und bei der CGMA-Berechnung die rsqrt-Operation bei DCS als 1 Operation gewertet [1]. Im Paper zur Masterarbeit [14] wurde bei der GTX 480 bei dem über die Laufzeit errechneten GFLOP/s-Wert die rsqrt-Operation als 2 Operationen und bei der GTX 285 als 1 Operation gezählt. In der Masterarbeit wird in beiden Fällen rsqrt als 1 Operation gezählt. Aus

diesem Grund weichen die Werte beim DCS bei der GTX 480 bei dem über die Laufzeit errechneten GFLOP/s-Wert etwas vom Paper [14] ab.

Zuerst wird der Needleman-Wunsch auf eine mögliche Bandbreiten-Limitierung untersucht. Beim Needleman-Wunsch werden 4 Werte aus dem Global Memory ausgelesen und damit 3 Operationen durchgeführt. Somit beträgt der CGMA 0,75 ($=3/4$). 0,75 Operationen werden pro Global-Memory-Zugriff durchgeführt. Da die 3 Operationen 3 Additionen sind, also keine Multiplikationen durchgeführt werden, kann die FMA-Operation nur zur Hälfte genützt werden. Damit liegt die Peak-Leistung bei der GTX 480 bei 672,5 GOP/s ($=1345/2$) [5] [21]. Die Bandbreite des Global Memory der GTX 480 beträgt 177 GB/s [21]. Damit können $44,25 \cdot 10^9$ Int-Werte ($=177/4$ Byte) pro Sekunde ausgelesen werden. Wenn man den Wert mit dem CGMA multipliziert, bekommt man die maximal mögliche Anzahl von Operationen pro Sekunde. Es sind $33,1875 \cdot 10^9$ ($=44,25 \cdot 10^9 \cdot 0,75$) Operationen pro Sekunde möglich. Man erreicht mit 33,19 GOP/s nur 1/20 der Peak-Leistung der GPU. Der Algorithmus ist somit durch die Bandbreite limitiert. Um die Bandbreiten-Limitierung aufzuheben, müsste die Bandbreite bei 3587 GB/s ($=672,5/0,75 \cdot 4$) liegen. Das Global-Memory-Programm erreicht mit Integer-Datentyp 3,13 GOP/s (CUDA, nur Kernel, N=8000, M=8000). Das sind 9,4% von 33,19 GOP/s. Die Wavefronts (=eine Antidiagonale nach der anderen) limitieren stark. Ohne Memory Coalescing erreicht man nur 1,5 GOP/s (CUDA, nur Kernel, N=8000, M=8000). Das sind 4,5% von 33,19 GOP/s.

Mit der Verwendung des Short-Datentyps können im Vergleich zum Int-Datentyp doppelt so viele Werte pro Sekunde aus dem Global Memory ausgelesen werden. Somit sind 66,375 GOP/s ($=88,5 \cdot 10^9 \cdot 0,75$) Operationen pro Sekunde möglich. Das entspricht 1/10 der Peak-Leistung der GPU. Das Global-Memory-Programm erreicht mit Short-Datentyp 3,09 GOP/s (CUDA, nur Kernel, N=8000, M=8000). Das sind 4,7% von 66,375 GOP/s. Ohne Memory Coalescing erreicht man nur 1,55 GOP/s (CUDA, nur Kernel, N=8000, M=8000). Das sind 2,3% von 66,375 GOP/s.

Durch die Verwendung von Shared Memory kann die Anzahl der Global-Memory-Zugriffe reduziert werden. Ein Block mit der Größe $55 \cdot 55$ ($=3025$ Werte) wird vom Global Memory in den Shared Memory geladen. $54 \cdot 54$ ($=2916$) Werte werden dann in $54+54-1$ Schritten (Antidiagonale für Antidiagonale des Blockes) berechnet und kein Wert muss nochmals aus dem Global Memory geladen werden. Pro Ergebniswert hat man somit 3 Operationen und 1 Global-Memory-Zugriff. Das CGMA-Verhältnis verbessert sich von 0,75 auf 3. Dieser Wert berücksichtigt nicht die hinzugefügten Synchronisationen/Wavefronts im Kernel. Es gilt die Annahme, dass alle Elemente der Teilmatrix im Shared Memory parallel berechnet werden können. In der Realität kann nur Antidiagonale für Antidiagonale parallel berechnet werden. Mit Shared Memory und Short-Datentyp sind 265,5 ($=177/2 \cdot 3$) GOP/s maximal erreichbar. Das sind 39,5% der Peak-Leistung. Ein CGMA von 8 ($=672,5/88,5$) oder eine Bandbreite von 448 GB/s ($=672,5/3 \cdot 2$) wäre notwendig, damit der Algorithmus nicht durch die Bandbreite des Global Memory limitiert ist. Das Shared-Memory-Programm erreicht 11,45 GOP/s (CUDA, nur Kernel, N=8000, M=8000). Das sind 4,3% von 265,5 GOP/s. Die Wavefronts und damit verbundenen Synchronisationspunkte limitieren stark.

Beim DCS-Algorithmus wird zuerst das Global-Memory-Programm mit Float-Datentyp und Optimierung rsqrtf auf eine mögliche Bandbreiten-Limitierung untersucht. Bei diesem Programm werden pro Thread 4 Werte aus dem Global Memory ausgelesen und damit 11 Operationen durchgeführt. Somit beträgt der CGMA 2,75 ($=11/4$). 2,75 Operationen werden pro Global-Memory-Zugriff durchgeführt. Bei den 11 Operationen handelt es sich um 3 Subtraktionen, 3 Additionen, 4 Multiplikationen und 1 rsqrt-Operation. 6 Operationen davon sind an drei FMA-Operationen ($=a \cdot b + c$) beteiligt. Damit gilt für die Peak-Leistung die obere Grenze 1345 GFLOP/s [5] [21], die nur mit FMA-Operationen erreichbar ist. Da die Bandbreite des Global Memory der GTX 480 177 GB/s [21] beträgt, können $44,25 \cdot 10^9$ Float-Werte ($=177/4$ Byte) pro Sekunde ausgelesen werden und $121,69 \cdot 10^9$ ($=44,25 \cdot 10^9 \cdot 2,75$) Operationen sind pro Sekunde möglich. Man erreicht mit 121,69

GFLOP/s nur 9% der Peak-Leistung der GPU. Der Algorithmus wäre ohne weitere Optimierungen durch die Bandbreite limitiert.

Das DCS-Programm erreicht mit Global Memory und Optimierung rsqrtf bei der GTX 480 393 GFLOP/s (CUDA, nur Kernel, $N=800$, $M=800$, Anzahl von Atomen=1000, $800*800*1000*11$ Operationen). Dieser Wert überschreitet deutlich den mit CGMA errechneten Wert von 121,69 GFLOP/s (siehe vorhergehenden Absatz). Der Grund ist der Cache für Global Memory bei der GTX 480 und die kleine Menge der auszulesenden Werte. Alle Gridpunkte verwenden dieselben Atome und die Anzahl der Atome beträgt im Programm 1000. Das Atom-Array passt mit der Größe von 16 KB ($=1000 \text{ Atome} * 4 \text{ Werte pro Atom} * 4 \text{ Byte (Float-Datentyp)}$) in den L2-Cache (768 KB) und sogar in die L1-Caches (Konfigurierbar: 16 KB oder 48 KB) der GTX 480 [5]. Somit müssen nur die Gridpunkte, die als Erster zur Ausführung gebracht werden, auf den Global Memory zugreifen. Sobald alle Atomdaten im Cache liegen, ist kein Global-Memory-Lesezugriff mehr notwendig. Eine Bandbreiten-Limitierung ist somit bei der Ausführung des DCS-Algorithmus auf der GTX 480 durch den Global-Memory-Cache nicht vorhanden.

Bei der GTX 285 erzielt man mit dem DCS-Programm (Global Memory und Optimierung rsqrt) 121 GFLOP/s (CUDA, nur Kernel, $N=800$, $M=800$, Anzahl von Atomen=1000, $800*800*1000*11$ Operationen). Da bei der GTX 285 die Bandbreite nur 159 GB/s [50] beträgt, sollte man eigentlich nur 109,31 ($=159/4*10^9*2,75$) GFLOP/s erreichen. Der Grund ist nicht der Cache für Global Memory, da die GTX 285 keinen besitzt, sondern Memory Coalescing. Global-Memory-Zugriffe innerhalb eines halben Warps ($=16$ Threads) werden bei der GTX 285 zusammengefasst [5]. Bei der GTX 480 geschieht das innerhalb eines Warps ($=32$ Threads) [5]. Für die Berücksichtigung von Memory Coalescing im CGMA bei der GTX 285 muss die Anzahl der Operationen von 16 Threads zusammenaddiert werden und die Anzahl der Global-Memory-Zugriffe (gemeinsame und separate Zugriffe) von 16 Threads ermittelt werden. Bei einem Global-Memory-Zugriff kann ein 32 Byte, ein 64 Byte oder ein 128 Byte Segment aus dem Global Memory ausgelesen werden [5]. Da sich die 16 Threads bei der Berechnung eines Gridpunkts immer auf derselben Codestelle befinden, laden sie zu einem Zeitpunkt auch die Daten desselben Atoms. Man hat somit $16*11$ Operationen und 4 Global-Memory-Zugriffe. Das ergibt einen CGMA von 44 ($=16*11/4$). Da die kleinste Segmentgröße für einen Global-Memory-Zugriff 32 Byte beträgt und das 8 Float-Werte ($32 \text{ Byte} = 8*4 \text{ Byte}$) entspricht, reduziert sich die Anzahl der Global-Memory-Zugriffe von $39,75*10^9$ ($=159/4 \text{ Byte}$) auf $4,97*10^9$ ($=159/32 \text{ Byte}$) pro Sekunde. Damit sind $218,63*10^9$ ($=159/32*10^9*44$) Operationen pro Sekunde möglich. Im Paper zur Masterarbeit [14] wurde der 32 Byte Zugriff statt in die maximale Anzahl der Global-Memory-Zugriffe pro Sekunde in den CGMA-Wert durch Reduzierung von 44 auf 5,5 ($=16*11/(4*8)$) miteinberechnet. An der Anzahl der Operationen von $218,63*10^9$ ($=159/4*10^9*5,5$) pro Sekunde ändert sich dadurch nichts. Das Ergebnis bleibt gleich. Man erreicht mit 218,63 GFLOP/s 20,6% der Peak-Leistung der GTX 285 ($=1062,72 \text{ GFLOP/s}$ [16] [50]). Damit ist das DCS-Programm mit Global Memory und Optimierung rsqrt bei der Ausführung auf der GTX 285 durch die Bandbreite limitiert.

Mit der Optimierung „dz*dz auf CPU vorberechnet“ wird die Anzahl der Operationen pro Thread zwar von 11 auf 9 reduziert (atominfo[atomid].z statt dz*dz und dz=coor-atoms[atomid*4+2] fällt weg), jedoch mit der Optimierung „4 Gridpunkte pro Thread“ kann die Anzahl der Operationen pro Thread auf 27 erhöht werden. Durch das Zusammenfassen von 4 Gridpunkten entlang der X-Achse werden 3 Operationen (coor-atominfo[atomid].y und (dy*dy)+atominfo[atomid].z) von 9 nur einmal für alle 4 Gridpunkte ausgeführt. Dadurch hat man statt 36 ($=9*4$) nur mehr 27 ($=3*1+6*4$) Operationen. Der CGMA kann von 2,75 ($=11/4$) auf 6,75 ($=27/4$) verbessert werden. Somit sind bei der GTX 285 $268,31*10^9$ ($=159/4*10^9*6,75$) Operationen pro Sekunde möglich. Mit Berücksichtigung von Memory Coalescing beträgt der CGMA auf der GTX 285 108 ($=16*27/4$). Damit sind auf der GTX 285 $536,63*10^9$ ($=159/32*10^9*108$) Operationen pro Sekunde möglich. Mit der Optimierung „lokale float4-Variable“ wird nur 1 Zugriff statt 4 Zugriffe auf den Global Memory durchgeführt, indem der

globale float4-Vektor in eine lokale float4-Variable gesichert wird und anschließend bei der Verwendung der einzelnen Werte des float4-Vektors auf die lokale float4-Variable zugegriffen wird. Da der kleinste Datenzugriff bei 16 Threads auf den Global Memory 32 Byte entspricht, wird mit dem float4-Zugriff (=16 Byte) nur die Hälfte an Bandbreite verschwendet. Mit der Optimierung „lokale float4-Variable“ verbessert sich der CGMA auf der GTX 285 von 108 auf 432 (=16*27/1). Damit wären theoretisch auf der GTX 285 $2146,5 \cdot 10^9$ (=159/32* 10^9 *432) Operationen pro Sekunde möglich. Da dieser Wert über der Peak-Leistung der GTX 285 liegt, ist der Algorithmus nicht durch die Bandbreite limitiert.

Im OpenCL-Global-Memory-Programm inklusive aller Optimierungen und Compileroptimierungsflags erreicht man bei der GTX 285 449 GFLOP/s (OpenCL, nur Kernel, N=800, M=800, Anzahl von Atomen=1000, $800 \cdot 800 / 4 \cdot 1000 \cdot 27$ Operationen). Die Peak-Leistung der GTX 285 mit 1062,72 GFLOP/s [16] [50] wird nicht erreicht, da man die Peak-Leistung nur dann erreicht, wenn pro Thread pro Takt eine MUL-Operation (=a*b) und eine MAD-Operation (=a*b+c) parallel ausgeführt wird. Das trifft auf den DCS-Algorithmus nicht zu. Bei 4 Gridpunkten pro Thread können 18 von 27 Operationen (=67%) zu MAD- bzw. FMA-Operationen zusammengefasst werden. Man hat also 9 MAD- bzw. FMA-Operationen, 5 Subtraktion/Addition-Operationen und 4 rsqrt-Operationen. Wenn man die 4 rsqrt-Operationen unberücksichtigt lässt, hat man insgesamt 14 MAD/FMA-Operationen, wobei bei 5 MAD/FMA-Operationen nur der Additions-/Subtraktionsteil benützt wird. Da die zusätzlich mögliche MUL-Operation der GTX 285 nicht benötigt wird (Einzelne Operationen nur Subtraktion/Addition-Operationen), reduziert sich die Peak-Leistung auf 708,48 GFLOP/s ($1062,72 / 3 \cdot 2$). Die 708,48 GFLOP/s werden jedoch nur erreicht, wenn im Code MAD- bzw. FMA-Operationen stehen. Die Hälfte (354,24 GFLOP/s) von 708,48 GFLOP/s wird erreicht, wenn im Code keine Addition- und Multiplikation-Operation zu einer FMA/MAD-Operation zusammengefügt werden kann und somit nur 1 Operation pro Kern pro Takt ausgeführt wird. Beim DCS-Code mit 4 Gridpunkten pro Thread führen 9 von 14 Instruktionen (=9 FMA/MAD-Operationen + 5 Subtraktion/Addition-Operationen) zwei Operationen pro Kern pro Takt aus. Die theoretische Peak-Leistung beträgt somit 581,95 GFLOP/s (=354,25 + $227,7$ (=354,25*9/14)) bzw. 54,76%. Wenn man davon ausgeht, dass die 4 rsqrt-Operationen parallel auf den SFU-Einheiten ausgeführt werden, dann liegt die ermittelte theoretische Peak-Leistung nahe an der Realität.

Als Nächstes wird das DCS-Programm mit Constant Memory und Optimierung rsqrt auf eine mögliche Bandbreiten-Limitierung untersucht. Beim DCS-Algorithmus werden zur Berechnung der Gridpunkte die Atomdaten als Input benötigt. Die Atomdaten werden somit nur gelesen und können im Programm aufgrund der Anzahl von 1000 Atomen mit je 4 Werten ($1000 \cdot 4 \cdot 4 = 16$ KByte) komplett im Constant Memory (64 KB) bzw. teilweise im Constant-Memory-Cache (8 KB) gehalten werden [5]. Das bedeutet, nur zu Beginn muss ein Global-Memory-Zugriff durchgeführt werden. Sobald die Daten im Constant-Memory-Cache sind, können sie wiederverwendet werden und kein Global-Memory-Zugriff muss durchgeführt werden. Eine Bandbreiten-Limitierung ist somit bei Verwendung von Constant Memory nicht vorhanden. Das DCS-Programm erreicht mit Constant Memory und Optimierung rsqrt bei der GTX 480 482 GFLOP/s (CUDA, nur Kernel, N=800, M=800, Anzahl von Atomen=1000, $800 \cdot 800 \cdot 1000 \cdot 11$ Operationen). Das entspricht 36% der Peak-Leistung. Bei der GTX 285 erreicht man 377 GFLOP/s (CUDA, nur Kernel, N=800, M=800, Anzahl von Atomen=1000, $800 \cdot 800 \cdot 1000 \cdot 11$ Operationen). Das entspricht 35% der Peak-Leistung.

Das DCS-Constant-Memory-Programm inklusive aller Optimierungen erreicht bei der GTX 480 532 GFLOP/s (CUDA, nur Kernel, N=800, M=800, Anzahl von Atomen=1000, $800 \cdot 800 / 4 \cdot 1000 \cdot 27$ Operationen). Das entspricht 40% der Peak-Leistung. Theoretisch wären 82% (=1104,8 GFLOP/s = $672,5 + 672,5 \cdot 9 / 14$) möglich, da bei 4 Gridpunkten pro Thread ohne Berücksichtigung der rsqrt-Funktionen aus 23 Operationen 9 FMA-Operationen (=2 Operationen pro Takt pro Kern) und 5 Subtraktion/Addition-Operationen (=1 Operation pro Takt pro Kern) gebildet werden (siehe zwei Absätze davor). Beim Ausführen des Constant-Memory-Programms mit der GTX 285 erreicht man

444 GFLOP/s (CUDA, nur Kernel, N=800, M=800, Anzahl von Atomen=1000, $800 \cdot 800 / 4 \cdot 1000 \cdot 27$ Operationen). Das entspricht 42% der Peak-Leistung. Theoretisch wären 54,76% (=581,95 GFLOP/s) möglich (siehe zwei Absätze davor).

9.7 Kann bei beiden Algorithmen auf der GPU ein Speedup gegenüber der CPU erreicht werden? Wie hoch ist der Speedup?

Bei beiden Algorithmen kann auf der GPU (CUDA- und OpenCL-Programm) ein Speedup gegenüber dem parallelen CPU-OpenMP-C-Programm erreicht werden (siehe Abbildung 150 (NW-Programm) und Abbildung 151 (DCS-Programm)). Der Speedup des GPU-CUDA-Programms gegenüber dem parallelen CPU-Programm beträgt beim NW 1,313 (OpenMP/xSSSE3 versus CUDA, N=8000, M=8000, Tesla C2050) und beim DCS 11,077 (OpenMP/xSSSE3 versus CUDA, N=800, M=800, GTX 480). Der Speedup des GPU-OpenCL-Programms gegenüber dem parallelen CPU-Programm beträgt beim NW 1,276 (OpenMP/xSSSE3 versus OpenCL, N=8000, M=8000, Tesla C2050) und beim DCS 10,31 (OpenMP/xSSSE3 versus OpenCL, N=800, M=800, GTX 480).

Beim Speedup-Diagramm des NW-Algorithmus sind bei der CPU-Laufzeit und bei der GPU-Laufzeit alle 3 Schritte (Initialisierung der Startmatrix, Berechnung der Ergebnismatrix und traceback) enthalten. Als GPU wurde die Tesla C2050 anstatt der GTX 480 verwendet, da bei der GTX 480 die OpenCL-Performance beim NW-Algorithmus ab N=5000, M=5000 deutlich schlechter als die CUDA-Performance ist (siehe Kapitel 7.4.1.2). Beim Speedup-Diagramm des DCS-Algorithmus wurden bei OpenCL Compiler-Optimierungsflags benützt. Ohne Compiler-Optimierungsflags beträgt der Speedup gegenüber dem parallelen CPU-Programm 9,23 (OpenMP/xSSSE3 versus OpenCL ohne Compiler-Optimierungsflags, N=800, M=800, GTX 480). Im Vergleich zu demselben DCS-Speedup-Diagramm im Paper zur Masterarbeit [14] wurden zwei Änderungen durchgeführt. Bei der OpenCL-Kurve wird nun die Optimierung „float4 lokal“ verwendet und die Gestaltung der Kurven wurde an das Speedup-Diagramm des NW-Algorithmus angepasst.

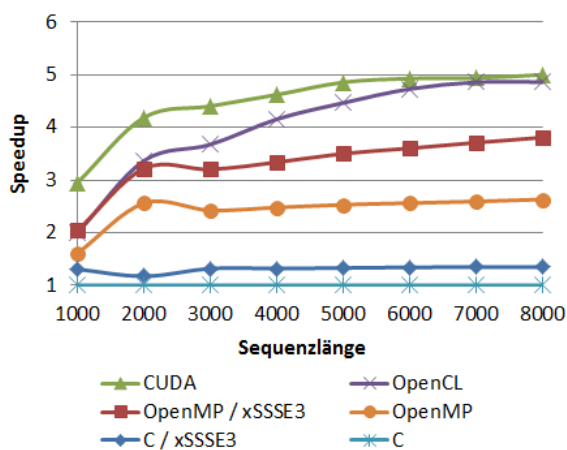


Abbildung 150: NW (alle 3 Schritte), GPU Tesla C2050

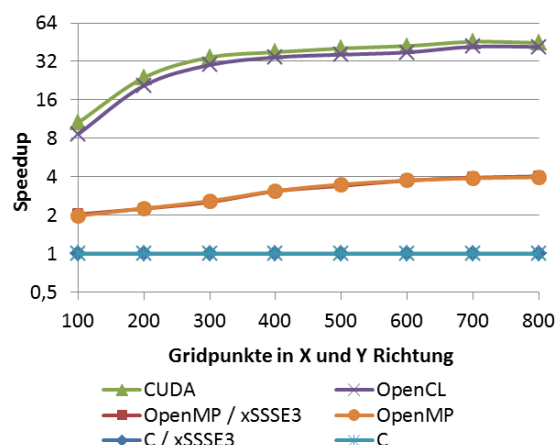


Abbildung 151: DCS, GPU GTX 480

9.8 Wie sieht der Performance-Unterschied der Programmierplattformen CUDA und OpenCL aus?

Der Performance-Unterschied zwischen CUDA und OpenCL ist nicht so groß (siehe Abbildung 152, Abbildung 153 und Abbildung 154). Beim Vergleichen der Gesamtlaufzeit sieht man, dass CUDA immer schneller als OpenCL ist. Beim DCS ohne OpenCL-Optimierungsflags beträgt der Unterschied bei N=800, M=800 20% bei Fermi (GTX 480) und 29,5% bei der Vorgängerarchitektur GT200 (GTX 285). Mit OpenCL-Optimierungsflags beträgt der Unterschied bei DCS bei N=800, M=800 7,4% bei

Fermi (GTX 480) und 6,9% bei der Vorgängerarchitektur GT200 (GTX 285). Beim NW (Shared-Memory-Programm mit allen Optimierungen) beträgt der Unterschied bei N=8000, M=8000 2,9% bei Fermi (Tesla C2050) und 13,2% bei der Vorgängerarchitektur GT200 (GTX 285).

Die Performance-Unterschiede zwischen CUDA und OpenCL bei den einzelnen Optimierungen werden im Kapitel 8.4.1 (DCS-Algorithmus) und im Kapitel 7.4.1.2 (NW-Algorithmus, Shared-Memory-Programm) dargestellt. Die nachfolgenden DCS-Abbildungen (Abbildung 153 und Abbildung 154) sind identisch mit den Abbildungen (Abbildung 120 und Abbildung 121) im Kapitel 8.4.1.

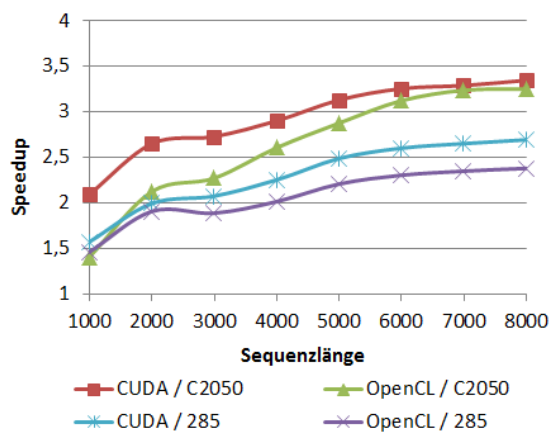


Abbildung 152: NW (Gesamtspeedup, alle Optimierungen, Shared Memory)

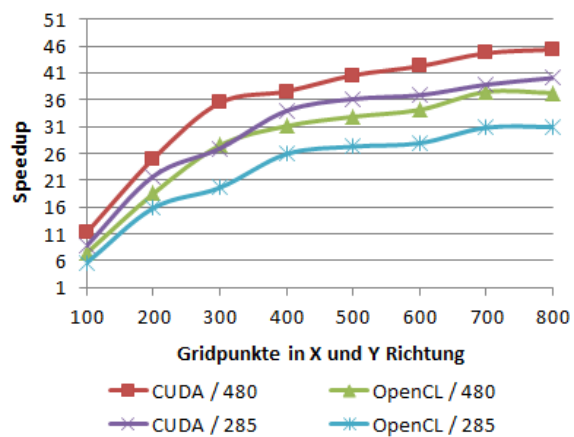


Abbildung 153: DCS, OpenCL ohne Optimierungsflags und mit Global Memory, CUDA mit Constant Memory (Gesamtspeedup, alle Optimierungen)

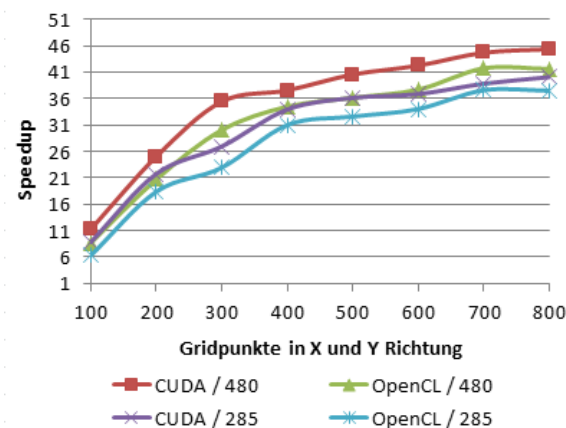


Abbildung 154: DCS, OpenCL mit Optimierungsflags und mit Global Memory, CUDA mit Constant Memory (Gesamtspeedup, alle Optimierungen)

9.9 Wie sieht der Performance-Unterschied zu älteren GPU-Architekturen aus? Skaliert der GPU-Code automatisch von alter Architektur zu neuer Architektur bzw. welche Änderungen müssen vorgenommen werden?

In diesem Kapitel wird für die Fermi-Architektur die GTX 480 und für die Vorgängerarchitektur GT200 die GTX 285 herangezogen. Außerdem wird das CUDA-Programm verwendet. Beim DCS wird die Problemgröße N=800, M=800 und beim NW die Problemgröße N=8000, M=8000 benützt.

Die GTX 480 hat 15 Multiprozessoren, 64 KB Constant Memory und eine Cache-Hierarchie über den Global Memory [5]. Ein Multiprozessor hat 32 Kerne, 4 SFUs, 2 Warp Scheduler, 48 KB Shared

Memory, 128 KB Register Memory, 8 KB Constant Memory Cache, maximal 1536 aktive Threads und maximal 8 aktive Blöcke [5]. Wenn man die Werte der 15 Multiprozessoren zusammenaddiert, erhält man 480 Kerne, 60 SFUs, 30 Warp Scheduler, 720 KB Shared Memory, 1920 KB Register Memory, 120 KB Constant Memory Cache, maximal 23040 aktive Threads und maximal 120 aktive Blöcke. Pro Kern pro Takt kann 1 FMA-Operation (=1 Addition und 1 Multiplikation, siehe Kapitel 2.1.2) ausgeführt werden [5]. Die GTX 480 hat eine Taktfrequenz von 1401 MHz und somit eine Peak-Leistung von 1345 GFLOP/s ($=1,401 \cdot 480 \cdot 2$) [5] [21].

Die GTX 285 hat 30 Multiprozessoren und 64 KB Constant Memory [5]. Ein Multiprozessor hat 8 Kerne, 2 SFUs, 1 Warp Scheduler, 16 KB Shared Memory, 64 KB Register Memory, 8 KB Constant Memory Cache, maximal 1024 aktive Threads und maximal 8 aktive Blöcke [5]. Wenn man die Werte der 30 Multiprozessoren zusammenaddiert, erhält man 240 Kerne, 60 SFUs, 30 Warp Scheduler, 480 KB Shared Memory, 1920 KB Register Memory, 240 KB Constant Memory Cache, maximal 30720 aktive Threads und maximal 240 aktive Blöcke. Pro Kern pro Takt kann 1 MAD-Operation (=1 Addition und 1 Multiplikation, siehe Kapitel 2.1.2) ausgeführt werden und zusätzlich kann pro Kern pro Takt eine Multiplikation auf den SFUs ausgeführt werden [16]. Die GTX 285 hat eine Taktfrequenz von 1476 MHz und somit eine Peak-Leistung von 1062,72 GFLOP/s ($=1,476 \cdot 240 \cdot 3$) [16] [50].

Da beim NW-Algorithmus (3 Addition-Operationen) und beim DCS-Algorithmus (9 MAD- bzw. FMA-Operationen, 5 Subtraktion/Addition-Operationen und 4 rsqrt-Operationen) die zusätzlich mögliche Multiplikation pro Kern pro Takt bei der GT200-Architektur nicht verwendet werden kann, ist die Peak-Leistung der GT200 auf 708,48 GFLOP/s reduziert und somit ist die Peak-Leistung bei Fermi zirka doppelt so hoch wie bei GT200.

Beim DCS-Algorithmus wurden keine Möglichkeiten für Änderungen am Code für eine bessere Anpassung an die Fermi-Architektur für mehr Performance gefunden. Der Gesamtspeedup ist bei Fermi automatisch um 12% höher als bei GT200 (Constant Memory mit allen Optimierungen, siehe Abbildung 155). Wenn man nur die Laufzeit des Kernels betrachtet, erreicht man bei Fermi eine Steigerung von 20% gegenüber GT200 (Constant Memory mit allen Optimierungen, siehe Abbildung 137 im Kapitel 8.4.2). Da die Anzahl der Kerne mit Fermi gegenüber GT200 verdoppelt wurden und die Taktfrequenz in etwa gleich geblieben ist, würde man eine Halbierung der Laufzeit mit Fermi erwarten. Der Grund, warum der Laufzeitunterschied zwischen Fermi und GT200 beim DCS-CUDA-Constant-Memory-Programm mit allen Optimierungen so gering ist, konnte nicht herausgefunden werden. Es wurde die Ausnutzung der maximalen Anzahl der aktiven Blöcke pro Multiprozessor von 8, die Optimierung des Codes auf die Fermi-Architektur durch den Compilerflag „-arch=sm_20“ und die Occupancy/maximale aktive Threadanzahl mit Hilfe des Nvidia CUDA Occupancy Calculator [20] (100% bei GT200 und Fermi, 14 Register werden pro Thread laut Compilerflag --ptxas-options=-v benötigt) überprüft, jedoch konnte durch Änderungen keine Verbesserung an der Laufzeit erzielt werden.

Beim NW-Algorithmus kann für mehr Performance die Blockgröße auf den vergrößerten Shared Memory bei Fermi (48 KB pro Multiprozessor) im Vergleich zu GT200 (16 KB pro Multiprozessor) angepasst werden. Mit der angepassten Blockgröße bei Fermi ist der Gesamtspeedup bei Fermi um 43% höher als bei GT200 (Shared Memory mit allen Optimierungen, siehe Abbildung 156) und damit ist die Performance-Verbesserung deutlich stärker als beim DCS-Algorithmus. Wenn man nur die Laufzeit des Kernels betrachtet, erreicht man mit dem NW-Algorithmus bei Fermi eine Steigerung von 125% gegenüber GT200 (Shared Memory mit allen Optimierungen, siehe Abbildung 83 im Kapitel 7.4.2).

Interessant ist, dass beim NW-Algorithmus beim CUDA-Shared-Memory-Programm ohne Optimierungen (Ausgangscode, Blockgröße beträgt 16 bei Fermi und GT200) bei Vergleichen der Kernellaufzeit zwischen Fermi und GT200 eine Performance-Verbesserung mit Fermi von nur 16%

erreicht wird. Dieser Wert entspricht in etwa dem Wert beim DCS-Algorithmus. Mit der ersten Optimierung „Optimale Blockgröße“ wird die Blockgröße speziell an die Fermi-Architektur und an die GT200-Architektur angepasst. Die optimale Blockgröße beträgt 15 bei GT200 und 27 bei Fermi. Durch die Vergrößerung der Blockgröße bei der Ausführung auf der Fermi-Architektur aufgrund des vergrößerten Shared Memory pro Multiprozessor (48 KB statt 16 KB) wird der Kernellaufzeitunterschied zwischen Fermi und GT200 deutlich von 16% auf 66% erhöht. Bei der nächsten Optimierung „Short-Datentyp“ beträgt der Unterschied 79% (Blockgröße: 22 bei GT200 und 38 bei Fermi). Bei der weiteren Optimierung „Nur 1 Inputmatrix und 1 shared Teilmatrix“ ist der Unterschied mit 124% (Blockgröße: 31 bei GT200 und 54 bei Fermi) sehr groß und erreicht mindestens eine Halbierung der Laufzeit. Die letzte Optimierung „Initialisierung der Inputmatrix auf der GPU anstatt Übertragung“ hat keinen Einfluss auf den Kernel und der Unterschied beträgt 125%.

Die Abbildung 155 ist identisch mit der Abbildung 136 im Kapitel 8.4.2. Die Abbildung 156 ist identisch mit der Abbildung 82 im Kapitel 7.4.2.

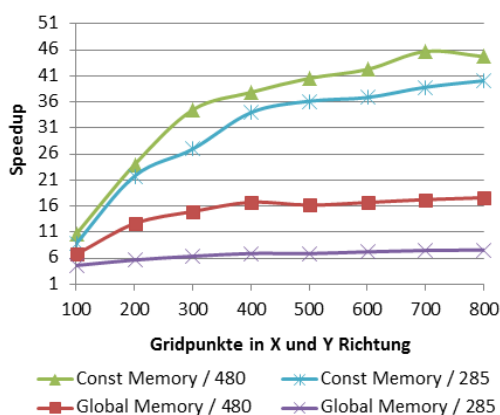


Abbildung 155: DCS, Gesamt-Speedup, CUDA (Const Memory mit allen Optimierungen und Global Memory ohne Optimierungen)

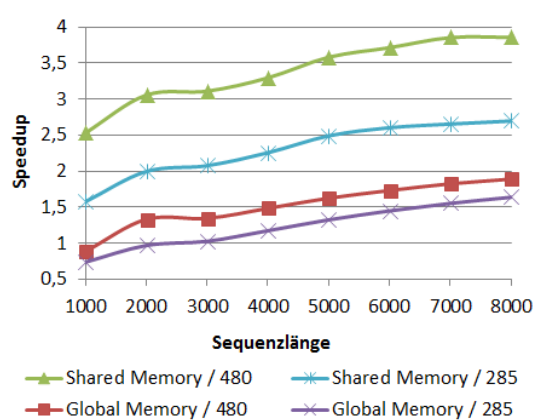


Abbildung 156: NW, Gesamt-Speedup, CUDA (Shared Memory mit allen Optimierungen und Global Memory mit allen Optimierungen)

9.10 Welche Optimierungen müssen auf einen GPU-Code angewendet werden, um einen schnelleren Code zu erzeugen? Welche davon sind Algorithmus-abhängig, welche allgemein nutzvoll?

Allgemein nutzvolle Optimierungen sind Padding, Wahl der optimalen Thread-Blockgröße, Verwendung von kleineren Datentypen und Verwendung der kleineren, schnelleren Speicher.

Padding wird benötigt, da die Anzahl der Threads immer ein Vielfaches der Blockgröße sein muss, aber in der Praxis oft eine andere Anzahl von Threads benötigt wird. Anstatt eine aufwendige if-Bedingung für alle Threads zu verwenden, ist es effizienter, die zu vielen Threads dieselben Berechnungen wie die anderen Threads durchführen zu lassen und deren Ergebnisse nicht im Endergebnis zu berücksichtigen.

Die Blockgröße von Threads sollte nicht zu klein gewählt werden, da nur 8 Blöcke pro Multiprozessor aktiv gehalten werden können. Aufgrund einer zu klein gewählten Anzahl von aktiven Threads könnte die Anzahl der exekutionsbereiten aktiven Threads zu klein werden und die große Latenzzeit, die beim Zugriff auf Daten im Global Memory entsteht, nicht mehr durch das Abwechseln von aktiven Threads versteckt werden. Im Gegensatz dazu, wenn man die Blockgröße zu groß wählt, kann es dazu kommen, dass das Programm durch die Hardwarebeschränkung „maximale Blockgröße“ auf manchen GPUs nicht funktioniert, durch das Padding zu viele unnötige Berechnungen durchgeführt werden und bei kleinen Problemgrößen wenige Blöcke entstehen, die auf die Multiprozessoren

aufgeteilt werden können. Beim Shared-Memory-Programm des NW-Algorithmus ist das Shared-Memory-Array abhängig von der Blockgröße und somit muss die Blockgröße so gewählt werden, dass der Shared Memory bei 8 Blöcken pro Multiprozessor vollständig ausgenutzt wird. Am besten ist es, wenn die Blockgröße im Programmcode dynamisch an die Hardwarebeschränkungen (z. B. maximale Blockgröße und maximale Größe des Shared Memory) der GPU angepasst wird.

Erst durch Memory Coalescing ist es möglich der maximalen Peak-Bandbreite des Global Memory näher zu kommen. Bei Fermi spielt Memory Coalescing durch die Caches eine nicht mehr so große Rolle.

Kleinere Datentypen, wie Float statt Double bei DCS und Short statt Int bei NW, führen zu einer starken Performance-Verbesserung. Float statt Double führt bei DCS zu schnelleren Berechnungen und Short statt Int führt bei NW zu schnelleren Datenübertragungen zwischen den Arbeitsspeichern der CPU und der GPU.

Mehr als einmal benützte Daten sollten in den kleineren Speichern (Constant Memory, Shared Memory) zwischengespeichert werden. Wenn Daten nur ausgelesen werden, eignet sich der 64 KByte Constant Memory und der dazugehörige 8 KByte Cache pro Multiprozessor dafür. Sollten mehrere Threads innerhalb eines Blocks von Threads auf dieselben Daten im Global Memory zugreifen, ist es sinnvoll, wenn die Threads das Laden der Daten aus dem Global Memory aufteilen und nach einem Synchronisationspunkt die geladenen Daten im Shared Memory gemeinsam nützen. Ebenfalls können Ergebnisse über den Shared Memory innerhalb eines Blocks von Threads ausgetauscht werden und auf deren Basis weitere Berechnungen durchgeführt werden. Die Latenzzeit zum Global Memory kann eingespart und die Global-Memory-Bandbreite kann entlastet werden.

Algorithmus abhängige Optimierungen sind Reduktion der Anzahl der Operationen, Reduktion der über den PCI-Bus zu übertragenden Datenmenge, Reduktion des benötigten Speichers und Verwendung von Vektordatentypen.

Die Anzahl der Operationen kann beim DCS durch Verwendung von $x \cdot \text{rsqrt}$ statt x/sqrt reduziert werden. Da sqrt mit $1/\text{rsqrt}$ berechnet wird, erspart man sich eine Operation. Zusätzlich ist die Multiplikation $x \cdot \text{rsqrt}$ weniger aufwendig als die Division x/sqrt . Der Nachteil ist, dass es dadurch zu einem ungenaueren Ergebnis kommen kann. Eine weitere Reduktion der Operationen kann durch das Zusammenlegen von partiell redundanten Berechnungen erreicht werden, indem mehrere Threads, die teilweise gleiche Operationen ausführen, zu einem Thread zusammengelegt werden. Beim DCS kann man Berechnungen in der for-Schleifenhierarchie nach oben ziehen und damit redundante Operationen verhindern bzw. reduzieren. Im Programm werden zum Beispiel 4 Threads zu einem Thread zusammengelegt. Der Nachteil ist, dass man dadurch eine Abhängigkeit erzeugt und Parallelität reduziert. Berechnungen, die nur einmal für alle GPU-Threads ausgeführt werden müssen, können bereits auf der CPU durchgeführt werden. Hier ergibt sich der Nachteil, dass die Performance zusätzlich von der CPU abhängig gemacht wird.

Da der PCI-Bus aufgrund der kleinen Bandbreite nicht für große Datenmengen geeignet ist, ist es wichtig die Übertragungsmenge zu reduzieren. Bei Algorithmen, die große Datenmengen übertragen, ist diese Optimierung wichtig, um die Gesamtlaufzeit des Programms zu reduzieren. Beim NW-Algorithmus wird zum Beispiel die aus einer kleinen Datenmenge generierte Startmatrix nicht übertragen, sondern direkt auf der GPU generiert. Durch Verwendung des Short- anstatt des Int-Datentyps ist die über den PCI-Bus zurück zu übertragende Ergebnismatrix nur halb so groß.

Der benötigte Speicher kann reduziert werden, indem zum Beispiel beim NW die Ergebnisse in die Inputmatrix anstatt in eine separate Ergebnismatrix geschrieben werden. Das gilt sowohl bei der

Teilmatrix im Shared Memory, als auch für die gesamte Matrix im Global Memory. Da beim Shared Memory die Hälfte an Speicher eingespart wird, kann die Blockgröße und damit die Parallelität (mehr Threads pro Block) deutlich vergrößert werden. Beim Global Memory hat man durch die Speicherplatzeinsparung den Vorteil, dass erst bei einer noch größeren Problemgröße die maximale Größe des DRAM-Speichers erreicht wird.

Indem man einen float4-Vektordatenzugriff anstatt vier einzelnen float-Datenzugriffen macht, können 3 Datenzugriffe eingespart werden. Da beim DRAM sowieso beim Zugriff auf ein Datenelement viele weitere danach folgende Elemente ausgelesen werden, ist der Aufwand eines float4-Zugriffs und eines float-Zugriffs fast gleich.

10 GPU-Optimierungsstrategien

Wenn man die Optimierungen bei beiden Algorithmen betrachtet, kommt man zu folgenden Strategien. Um einen Code auf eine bestimmte GPU zu optimieren und der Peak-Leistung der GPU näher zu kommen, ist es wichtig die limitierenden Faktoren des Programms ausfindig zu machen und sie durch Code-Änderungen zu reduzieren [1]. Es muss ein Kompromiss zwischen Parallelität, Berechnungseffizienz und Bandbreitenverbrauch gefunden werden [1]. Sollten viele Daten zwischen den Arbeitsspeichern der CPU und GPU übertragen werden, sollte auch bei der Übertragung Optimierungen durchgeführt werden, um nicht nur die Kernellaufzeit zu reduzieren, sondern auch die Gesamtlaufzeit des Programms.

10.1 Optimierung bei der Auslastung / Parallelität

Damit die maximale Anzahl der aktiven Thread-Blöcke pro Multiprozessor, die maximale Anzahl der aktiven Threads pro Multiprozessor, die maximale Registeranzahl und die maximale Größe des Shared Memory bestmöglich ausgenutzt wird bzw. der limitierende Faktor erkannt werden kann, gibt es von Nvidia eine Excel-Datei mit dem Namen „CUDA Occupancy Calculator“ [20] (siehe Abbildung 157) [1]. Im zweiten unteren gelb hinterlegten Bereich wird die mögliche Anzahl der aktiven Thread-Blöcke pro Multiprozessor für die im Programmcode festgelegte Anzahl der Threads pro Block, für die Anzahl der benützten Register pro Thread und für den benutzte Shared Memory pro Block berechnet. Optimal ist die Zahl 8, da in diesem Fall die maximale Anzahl von 8 aktiven Blöcken pro Multiprozessor erreicht wird. Damit die Anzahl der aktiven Thread-Blöcke berechnet werden kann, muss die Hardware (Compute Capability der GPU und Größe des Shared Memory pro Multiprozessor (16 KB oder 48 KB konfigurierbar bei Fermi bzw. 16 KB bei GT200)) im grün hinterlegten Bereich und die benützten Ressourcen bei der Ausführung des Programms (Threadanzahl pro Block, Anzahl der benützten Register pro Thread und benützter Shared Memory pro Block) im orange hinterlegten Bereich angegeben werden. Durch die rot markierte Zelle beim zweiten unteren gelb hinterlegten Bereich sieht man, dass die im Programmcode festgelegte Anzahl der Threads pro Block nur 6 aktive Blöcke ermöglicht und somit die maximale Anzahl von 8 aktiven Thread-Blöcken unterschritten wird. Mit 256 Threads pro Block können bei maximal 1536 aktiven Threads pro Multiprozessor nur 6 Thread-Blöcke aktiv gehalten werden. Die benützten 16 Register pro Thread würden die maximale Anzahl von 8 aktiven Thread-Blöcken ermöglichen. Die benützten 4 KB Shared Memory pro Block würden sogar 12 aktive Thread-Blöcke pro Multiprozessor zulassen. Im Excel-Dokument kann auch die Auslastung der GPU (=Occupancy) abgelesen werden (siehe letzte Zeile im blau hinterlegten Bereich) und somit überprüft werden, ob die maximale Anzahl der aktiven Threads erreicht wird. Im Screenshot des Excel-Dokuments ist die Auslastung bei 100% und somit wird die maximale Anzahl von aktiven Threads erreicht und es können 1536 Threads pro Multiprozessor aktiv gehalten werden (siehe zweite Zeile im blau hinterlegten Bereich). Damit auch bei der Anzahl der aktiven Blöcke pro Multiprozessor die maximalen Anzahl und somit die maximale Parallelität erreicht wird, müsste die Anzahl der Threads pro Block von 256 auf 192 reduziert werden. Die Physical Limits im grau hinterlegten Bereich werden nach der Auswahl der Compute Capability aus der zweiten Tabelle des Excel-Dokuments gelesen (siehe Abbildung 158).

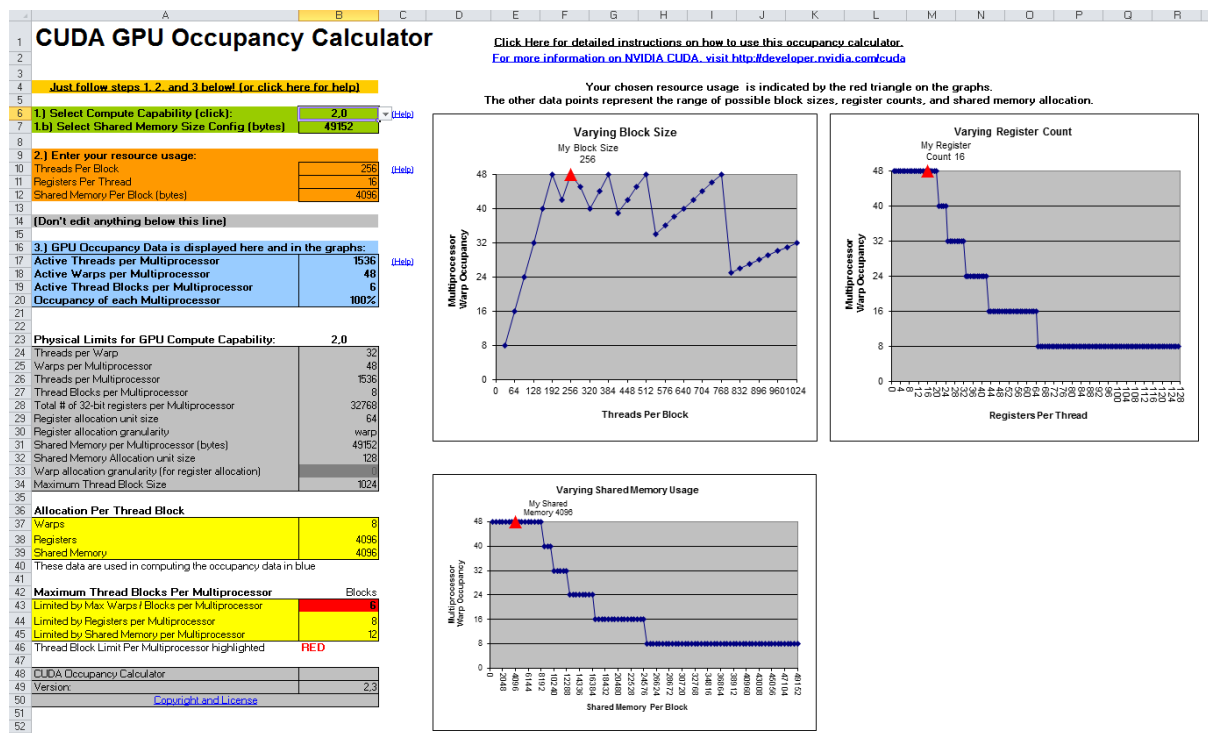


Abbildung 157: Occupancy Calculator Excel-Dokument (Tabelle: Calculator) [20]

Compute Capability	1,0	1,1	1,2	1,3	2,0	2,1
SM Version	sm_10	sm_11	sm_12	sm_13	sm_20	sm_21
Threads / Warp	32	32	32	32	32	32
Warps / Multiprocessor	24	24	32	32	48	48
Threads / Multiprocessor	768	768	1024	1024	1536	1536
Thread Blocks / Multiprocessor	8	8	8	8	8	8
Max Shared Memory / Multiprocessor (bytes)	16384	16384	16384	16384	49152	49152
Register File Size	8192	8192	16384	16384	32768	32768
Register Allocation Unit Size	256	256	512	512	64	64
Allocation Granularity	block	block	block	block	warp	warp
Shared Memory Allocation Unit Size	512	512	512	512	128	128
Warp allocation granularity (for registers)	2	2	2	2		
Max Thread Block Size	512	512	512	512	1024	1024
Shared Memory Size Configurations (bytes)	16384	16384	16384	16384	49152	49152
[note: default at top of list]					16384	16384

Abbildung 158: Occupancy Calculator Excel-Dokument (Tabelle: GPU Data) [20]

10.2 Optimierung bei den Datenzugriffen / Bandbreitenverbrauch / Latenzzeit

Damit der massive parallele Datenzugriff zum Laden der Inputdaten nicht zum limitierenden Faktor der Anwendung wird, ist Memory Coalescing, Tiling und die Verwendung der kleineren Speicher wichtig [1].

10.2.1 Memory Coalescing

Um in die Nähe der Peak-Global-Memory-Bandbreite zu kommen, müssen die Datenzugriffe im Kernel so programmiert sein, dass die Threads parallel auf Daten zugreifen, die nebeneinander im Global Memory stehen (siehe Kapitel 2.2.3.2) [1]. Durch den Einsatz eines Global-Memory-Caches in der GPU-Architektur Fermi spielt Memory Coalescing nicht mehr eine so große Rolle [1].

10.2.2 Tiling

Das Tiling wird dazu benutzt, um Daten, die von mehreren Threads in einem Block benötigt werden, nur einmal laden zu müssen (siehe Kapitel 2.2.8.1). Die Threads laden parallel jeweils einen Teil der Daten aus dem Global Memory in den Shared Memory und nach einem Synchronisationspunkt können alle Threads auf diese gemeinsam geladenen Daten zugreifen, ohne auf den Global Memory zugreifen zu müssen. Dadurch kann Latenzzeit und Bandbreite des Global Memory eingespart werden.

10.2.3 Verwendung der kleineren Speicher (Constant Memory und Shared Memory)

Durch die Wiederverwendung von Daten im Shared Memory durch mehrere Threads innerhalb eines Thread-Blocks kann die Latenzzeit reduziert und auch Global-Memory-Bandbreite eingespart werden. Mit Hilfe des Constant Memory und der Constant-Memory-Caches können häufig genützte nur lesbare Daten, sobald sie sich im Cache befinden, schnell abgefragt werden. Damit wird ebenfalls Latenzzeit und Global-Memory-Bandbreite eingespart.

10.3 Optimierung bei den Instruktionen / Berechnungseffizienz

10.3.1 Verwendung von Intrinsic/Native-Funktionen und Compilerflags

Durch die Verwendung von Intrinsic/Native-Funktionen (siehe Kapitel 3.3.12) kann die Ausführung einer Funktion beschleunigt werden. Jedoch sollte dabei beachtet werden, dass es dadurch zu einer Reduktion der Genauigkeit des Ergebnisses kommen kann. Dasselbe gilt für Compilerflags.

10.3.2 Reduzierung der Operationen und Verwendung von schnelleren Operationen

Die Berechnung von $x * \text{rsqrt}()$ ist deutlich weniger aufwendiger als $x / \text{sqrt}()$. Man erspart sich eine Operation, da $\text{sqrt}()$ mit $1 / \text{rsqrt}()$ berechnet wird [5] und die aufwendigere Division ($x / \text{sqrt}()$) wird durch eine schnelle Multiplikation ($x * \text{rsqrt}()$) ersetzt (siehe Kapitel 8.3.2.1). Diese Änderung kann das Programm deutlich schneller machen. Die Anzahl der Operationen kann auch dadurch reduziert werden, indem ein Thread die Berechnung mehrerer paralleler Berechnungen übernimmt. Falls es parallele Berechnungen gibt, die teilweise auf gleichen Inputdaten basieren, kann durch die Zusammenfassung Instruktionen eingespart werden. Der Nachteil ist, dass dadurch Parallelität verloren geht.

10.3.3 Float- statt Double-Datentyp

Da eine Double-Berechnung mehr Zeit als eine Float-Berechnung auf der GPU beansprucht, ist es sinnvoll Float- statt Double-Berechnungen dort einzusetzen, wo die Genauigkeit einer Float-Berechnung ausreichend ist.

10.4 Optimierungen an der Datenübertragung zwischen CPU-Arbeitsspeicher und GPU-Arbeitsspeicher

10.4.1 Generierung der Inputmatrix durch Inputwerte auf der GPU

Sollte eine Startmatrix auf Basis einer deutlich kleineren Datenmenge generiert werden, ist es sinnvoll, diese nicht auf der CPU zu erzeugen und sie dann zu übertragen. Stattdessen sollte die deutlich kleinere Datenmenge auf die GPU übertragen werden und die Startmatrix mit Hilfe eines Kernels direkt auf der GPU generiert werden.

10.4.2 Verwendung kleinerer Datentypen

Durch Verwendung eines halb so großen Datentyps (z. B. Short- anstatt Int-Datentyp) kann die Datenmenge, die zwischen CPU und GPU transportiert werden muss, halbiert werden. Durch die

Halbierung kommt es jedoch zu einer starken Reduktion des Maximalwerts. Damit es nicht zu einem falschen Ergebnis kommt, muss ermittelt werden, wann der Maximalwert überschritten wird und das Programm mit dieser Optimierung nicht mehr zur Ausführung geeignet ist.

11 Ausblick

11.1 OpenCL und AMD-GPUs

Das OpenCL-Programm vom Needleman-Wunsch- und DCS-Algorithmus kann im Gegensatz zum CUDA-Programm auch auf AMD-GPUs ausgeführt werden. Wenn man beim DCS-OpenCL-Programm die Laufzeit der AMD-GPU ATI Radeon 5870 mit der Laufzeit der Nvidia-GPU GTX 480 vergleicht, kommt man zu der Erkenntnis, dass die AMD-GPU für große Problemgrößen und die Nvidia-GPU für kleine Problemgrößen besser geeignet ist (siehe Abbildung 159 und Abbildung 160). Beim DCS-OpenCL-Programm wurden alle Optimierungen verwendet. Die Unterschiede im Code sind, dass bei der GTX 480 vier Gridpunkte in X-Richtung pro Thread und eine lokale float4-Variable und bei der HD 5870 acht Gridpunkte in X-Richtung pro Thread und eine globale float4-Variable verwendet wurde. Als Basis für die Speedup-Diagramme wird die CPU-Konfiguration „Intel Xeon X5550, dz*dz und dy*dy+dz*dz vorberechnet, 4 Atomvektoren, icc 11.1, -O3“ verwendet.

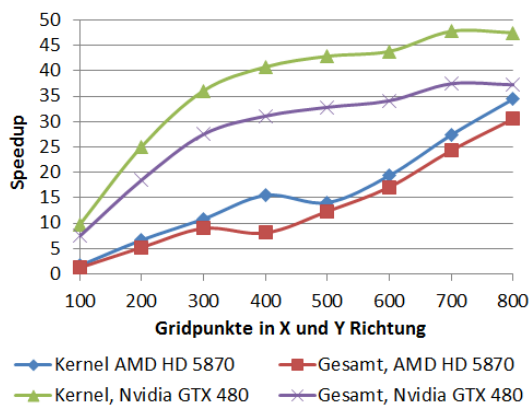


Abbildung 159: GTX 480 mit Global Memory und HD 5870 mit Constant Memory

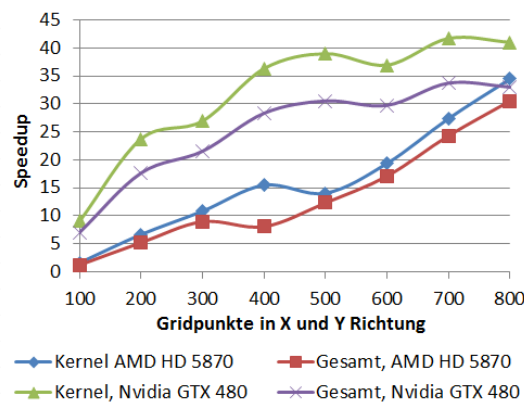


Abbildung 160: GTX 480 und HD 5870 mit Constant Memory

Während die Speedup-Kurve in Abbildung 159 bei der Nvidia-GPU mit steigender Problemgröße immer flacher wird, steigt die Kurve der AMD-GPU weiterhin an. Für eine volle Auslastung der GPU benötigt man bei AMD-GPUs eindeutig eine größere Problemgröße als bei Nvidia-GPUs. Ein Nachteil der AMD-Grafikkarte gegenüber der Nvidia-Grafikkarte ist, dass die maximale Blockgröße nur 256 Threads beträgt. Die Nvidia-Grafikkarte erlaubt bis zu 1024 Threads pro Block [20]. Dieser Unterschied kann dazu führen, dass ein OpenCL-Programm, das auf der Nvidia-GPU ohne Probleme funktioniert, auf der AMD-GPU nicht funktioniert. Damit das Programm auf der AMD- und Nvidia-GPU funktioniert, darf die Blockgröße maximal 256 Threads im Programm betragen. Weiters gibt es einen Unterschied in der Größe des Shared Memory. Die AMD-GPU hat 32 KB und die Nvidia-GPU hat 48 KB pro Multiprozessor. Die Bandbreite zwischen GPU und Device Memory/DRAM beträgt bei der AMD-GPU 154 GB/s [53] und bei der Nvidia-GPU 177 GB/s [21]. Einen deutlichen Unterschied gibt es bei der Taktfrequenz der Kerne. Die AMD-GPU hat nur 0,85 GHz [53]. Die Nvidia-GPU kommt auf 1,401 GHz [21]. Statt auf Taktfrequenz setzt die AMD-GPU auf viele Kerne und besitzt mit 1600 Kernen [53] deutlich mehr als die GTX 480 mit 480 Kernen [21]. Die Kerne sind bei der AMD-GPU auf 20 Multiprozessoren verteilt [49]. Bei der Nvidia-GPU sind es 15 Multiprozessoren [5]. Weitere Unterschiede sind, dass die AMD-GPU über keine Cache-Hierarchie verfügt und die Peak-Leistung höher ist. Die Peak-Leistung bei Double-Precision beträgt bei der AMD-GPU 544 GFLOP/s [53] und bei der GTX 480 168,12 GFLOP/s [19] [21]. Auch die Tesla C2050 mit 515 GFLOP/s [25] liegt unter der

Double-Precision-Peak-Leistung der AMD-GPU. Bei der Single-Precision-Peak-Leistung besteht ebenfalls ein starker Unterschied. Die AMD-GPU kommt auf 2720 GFLOP/s [53]. Die Nvidia-GPU kann maximal 1345 GFLOP/s [5] [21] leisten. Damit ist die AMD-GPU theoretisch doppelt so schnell. Jedoch in der Praxis ist die Nvidia-GPU bei den kleineren Problemgrößen durch die Cache-Hierarchie und die höhere Taktfrequenz deutlich schneller.

11.2 CUDA-Toolkit 4.1, 4.2 und 5.0

Im Jänner 2012 wurde CUDA-Toolkit 4.1 veröffentlicht [40]. Die Laufzeiten in der Masterarbeit wurden bei den Geforce-GPUs mit Toolkit 4.0 und bei den Tesla-GPUs mit Toolkit 3.2 ermittelt. Mit Toolkit 4.1 gibt es einen neuen LLVM-basierten Compiler, der viele Programme automatisch um 10% schneller machen soll, und einen überarbeitenden Visual-Profiler für automatische Analysen des Codes [71].

Seit April 2012 gibt es das CUDA-Toolkit 4.2 für das Ausführen von CUDA-Programmen auf GPUs der Kepler-Architektur [40] [72].

Im Oktober 2012 wurde das CUDA-Toolkit 5.0 Production Release veröffentlicht [40] [73]. Mit Toolkit 5.0 steht die Entwicklungsumgebung Nsight auch für Eclipse zur Verfügung und die neuen Möglichkeiten, wie zum Beispiel Dynamic Parallelism, der Kepler-Architektur sind nun programmierbar [73].

11.3 Nvidia Visual-Profiler

Seit 2008 gibt es einen Nvidia Visual-Profiler [74]. Dieser kann zum Analysieren von „CUDA C/C++“-Programmen verwendet werden und ist im CUDA-Toolkit enthalten [74]. Die Programme werden direkt auf der GPU ausgeführt, also nicht emuliert. Wenn man das Profiling in der GUI deaktiviert, dauert die Ausführung genauso lange wie in der Konsole.

Mit Hilfe von Daten in Tabellen und in Grafiken kann man mit dem Profiler limitierende Faktoren aufspüren und das Programm dementsprechend dort optimieren [74]. Durch das Vergleichen der Daten mehrerer Sessions kann überprüft werden, ob die Änderungen zu einer Verbesserung geführt haben [74]. Weiters kann man automatische Analysen durchführen und sich Vorschläge ausgeben lassen, die zu einer Verbesserung des Programms führen könnten [74].

Bei den Session-Einstellungen (siehe Abbildung 161) kann der Sessionname, der Programmort, die Argumente und die maximale Ausführungsdauer eingegeben werden.

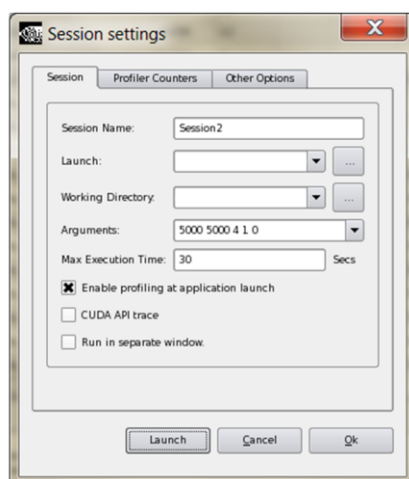


Abbildung 161: Einstellungen für Session

Bei den Profiler-Counters (siehe Abbildung 162, Abbildung 163 und Abbildung 164) werden die Profiler-Zähler, die gemessen werden sollen, ausgewählt.

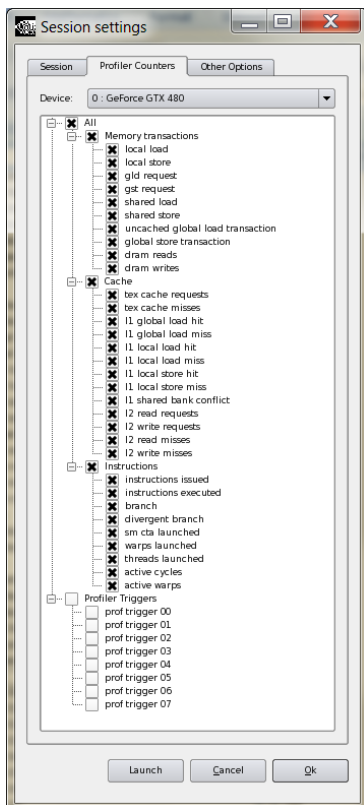


Abbildung 162:
Profiler-Counters (GTX 480)

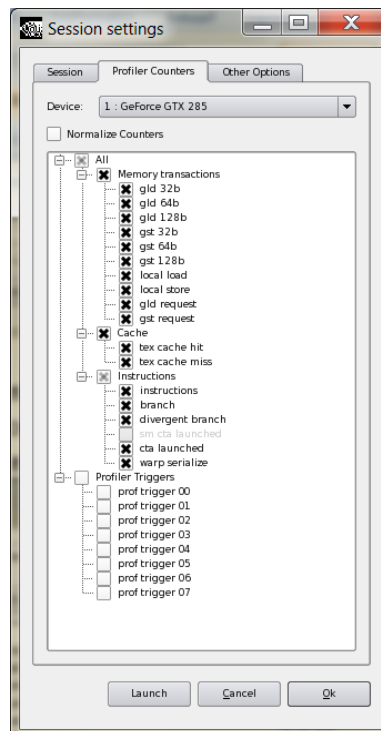


Abbildung 163:
Profiler-Counters (GTX 285)

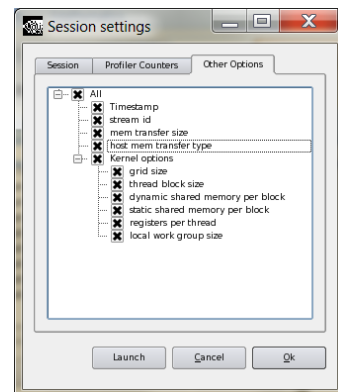


Abbildung 164:
Other Options

Nachdem der Launch-Button gedrückt wurde, öffnet sich die Tabelle „Profiler Output“ (siehe Abbildung 165) mit zahlreich ermittelten Daten. Zum Beispiel lässt sich die Occupancy (siehe Spalte Occupancy in Abbildung 165) der einzelnen aufgerufenen Kernels ablesen.

NeedlemanWunsch - Compute Visual Profiler - [Session1 - Device_0 - Context_0 [CUDA]]

Profiler Output

	GPU Timestamp	Method	GPU Time	CPU Time	grid size	block size	static shared memory per block	registers per thread	Occupancy	glid request Type: SM Run: 1	glt req: Type: SM
1	0	memcpyHtoD	4.192	20							
2	22	memcpyHtoD	3.936	12							
3	42	memcpyHtoD	1.184	5							
4	106	reference_cuda_initialisieren_3	969.088	995	[1 228]	[22 22 1]	1056	15	1	3870	54720
5	1786	matrix_cuda_ente_Zeile_initialisieren	4.384	21	[133 1]	[38 1 1]	0	6	0.333	0	20
6	2234	matrix_cuda_ente_Spalte_initialisieren	5.952	19	[133 1]	[38 1 1]	0	6	0.333	0	24
7	2684	needle_cuda_shared_1	32.864	48	[1 1]	[38 1 1]	5930	25	0.042	81	76
8	3154	needle_cuda_shared_1	32.16	44	[2 1]	[38 1 1]	5930	25	0.083	81	76
9	3626	needle_cuda_shared_1	32.16	44	[3 1]	[38 1 1]	5930	25	0.125	81	76
10	4102	needle_cuda_shared_1	32.064	45	[4 1]	[38 1 1]	5930	25	0.167	81	76
11	4570	needle_cuda_shared_1	32.096	45	[5 1]	[38 1 1]	5930	25	0.208	81	76
12	5050	needle_cuda_shared_1	32.032	45	[6 1]	[38 1 1]	5930	25	0.25	81	76
13	5518	needle_cuda_shared_1	31.904	44	[7 1]	[38 1 1]	5930	25	0.292	81	76
14	5992	needle_cuda_shared_1	32	45	[8 1]	[38 1 1]	5930	25	0.333	81	76
15	6460	needle_cuda_shared_1	31.936	44	[9 1]	[38 1 1]	5930	25	0.333	81	76
16	6936	needle_cuda_shared_1	31.904	45	[10 1]	[38 1 1]	5930	25	0.333	81	76

Read profiler output file for context #0, run #12. Number of rows=270

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'dynamic shared memory per block' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'stream id' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'local block size' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'local load Type: SM Run: 1' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'local store Type: SM Run: 1' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'l1 local load hit Type: SM Run: 2' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'l1 local load miss Type: SM Run: 3' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'l1 local store hit Type: SM Run: 3' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'divergent branch Type: SM Run: 4' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'l1 local store miss Type: SM Run: 4' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'uncached global load transaction Type: SM Run: 6' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'tex cache requests' having all zero values is hidden.

Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'tex cache misses' having all zero values is hidden.

Abbildung 165: Tabelle Profiler Output

Bei allen Tabellen kann man mit rechter Maustaste und „show All Columns“ alle Spalten anzeigen lassen. Über den Menüpunkt View kann man unterschiedliche Tabellen und Grafiken ausgeben lassen.

11.4 GTX 580

Die Fermi-Geforce-Grafikkarte GTX 580 beinhaltet die vollausgebaute Fermi-GPU. Im Vergleich zur GTX 480 wurde Folgendes verbessert. Die Anzahl der Kerne der GPU wurde von 480 auf 512, die Bandbreite zwischen GPU und Device Memory/DRAM wurde von 177,4 auf 192,4 GB/s, die Taktfrequenz der GPU von 1,401 auf 1,544 GHz und die Rechenleistung der GPU von 1,345 auf 1,581 TFLOP/s verbessert [18] [21].

11.5 Compute Capability 2.1 (GTX 460)

Bei der Fermi-Architektur wurden mit der Compute Capability 2.1 Änderungen durchgeführt. Die Anzahl der Kerne pro Multiprozessor wurde von 32 auf 48 erhöht, die Anzahl der SFUs wurde von 4 auf 8 verdoppelt und die Anzahl der Instruktionen pro Warp Scheduler wurde von 1 auf 2 erhöht [5]. Somit sind nun vier unterschiedliche Instruktionen pro Multiprozessor möglich [5].

11.6 Neue GPU-Architektur Kepler (Compute Capability 3.0 und 3.5)

11.6.1 Compute Capability 3.0 (GK104)

Mit der GTX 680 gibt es eine Gamer-Grafikkarte [75] und mit der Tesla K10 eine Computing-Grafikkarte mit der GPU GK104 [76]. Die Compute Capability der GK104-GPU hat die Zahl 3.0 [77].

Die Tesla K10-Grafikkarte besitzt zwei GK104-GPUs und ist für die Bereiche Seismic Processing, Image Processing, Signal Processing und Video Analytics geeignet [78]. Die Double-Precision-Peak-Leistung beträgt insgesamt 0,19 TFLOP/s und somit nur 95 GFLOP/s pro GPU [76]. Die Single-Precision-Peak-Leistung ist mit insgesamt 4,58 TFLOP/s und 2,29 TFLOP/s pro GPU deutlich höher [76]. Die Anzahl der Kerne beträgt $2 \cdot 1536$ [76]. Bei 2,29 TFLOP/s und 1536 Kerne ergibt sich eine Taktfrequenz von zirka 0,75 GHz $(= (2290 \text{ GFLOP/s}) / 1536 \text{ Kerne} / 2 \text{ Operationen pro Takt (=FMA)})$. Der Device

Memory/DRAM ist 2*4 GB groß und die Bandbreite zwischen den zwei GPUs und den zwei DRAMs beträgt zusammen 2*160 GB/s [78].

Die Grafikkarte GeForce GTX 680 enthält eine GK104-GPU [75]. Die Bandbreite zwischen GPU und Device Memory/DRAM ist mit 192,2 GB/s fast identisch mit der Bandbreite der Fermi-Grafikkarte GTX 580 (siehe Kapitel 11.4), dabei wurden jedoch die Bits der Speicherschnittstelle von 384-Bits auf 256-Bits reduziert und die Speichertaktfrequenz von 2 auf 3 GHz erhöht [18] [79]. Die Kepler-Grafikkarte unterstützt PCI-E 3.0 [79]. Die Speichergröße des Device Memory/DRAM wurde im Vergleich zur GTX 580 von 1,5 GByte auf 2 GByte erhöht [18] [79]. Die GTX 680 besitzt 8 Multiprozessoren (192 Kerne pro Multiprozessor) und hat somit nur halb so viele Multiprozessoren wie die GTX 580 (16 Multiprozessoren mit je 32 Kernen) [75]. Die Anzahl der Transistoren wurde nur gering von 3 Milliarden auf 3,54 Milliarden erhöht [75]. Hingegen konnte die Single-Precision-Peak-Leistung durch mehr Kerne (1536 Kerne bei der GTX 680 im Vergleich zu 512 Kerne bei der GTX 580) deutlich von 1581 GFLOP/s auf 3090 GFLOP/s erhöht werden [75]. Der TDP konnte dabei von 244 Watt auf 195 Watt reduziert werden [75]. Die Taktfrequenz der Kerne wurde durch den Wegfall der Shadertaktfrequenz deutlich von 1544 MHz auf 1006 MHz reduziert [75].

11.6.2 Compute Capability 3.5 (GK110)

Im Mai 2012 wurden Informationen zu der GK110-GPU, die für das Ausführen von CUDA-Programmen optimiert ist, veröffentlicht [77] und seit November 2012 gibt es mit Tesla K20 und Tesla K20X zwei Grafikkarten, die die GK110-GPU verwenden [30] [80]. Die Grafikkarte Tesla K20 hat 5 GB Device Memory/DRAM, 208 GB/s Bandbreite zwischen GPU und Device Memory/DRAM (wenn ECC deaktiviert ist), eine GK110-GPU mit 13 Multiprozessoren/SMX (2496 Kerne = 13 SMX * 192 Kerne pro SMX), 3,52 TFLOP/s Single-Precision-Peak-Leistung und 1,17 TFLOP/s Double-Precision-Peak-Leistung [77] [80]. Die Grafikkarte Tesla K20X hat 6 GB Device Memory/DRAM, 250 GB/s Bandbreite zwischen GPU und Device Memory/DRAM (wenn ECC deaktiviert ist), eine GK110-GPU mit 14 Multiprozessoren/SMX (2688 Kerne = 14 SMX * 192 Kerne pro SMX), 3,95 TFLOP/s Single-Precision-Peak-Leistung und 1,31 TFLOP/s Double-Precision-Peak-Leistung [77] [80]. Die voll ausgebaute Kepler-GPU GK110 wird aus 15 Multiprozessoren/SMX (2880 Kerne = 15 SMX * 192 Kerne pro SMX) bestehen [77]. Die Größe des L2-Cache beträgt 1536KB bei der GK110-GPU [77]. Im Vergleich zu Fermi wurde die Größe und die Bandbreite des L2-Cache verdoppelt [77]. Die Compute Capability der GK110-GPU hat die Zahl 3.5 [77].

Bei der Kepler-Grafikkarte Tesla K20X liegt die Double-Precision-Leistung bei der Nvidia-DGEMM-Berechnung (=Multiplikation von zwei Matrizen bei Double-Precision) bei 1,22 TFLOP/s und die Single-Precision-Leistung bei der Nvidia-SGEMM-Berechnung (=Multiplikation von zwei Matrizen bei Single-Precision) bei 2,9 TFLOP/s [81]. Im Vergleich dazu erreicht die Fermi-Grafikkarte Tesla M2090 bei der Nvidia-DGEMM 0,43 TFLOP/s und bei der Nvidia-SGEMM 0,89 TFLOP/s [81]. Damit ist die Kepler-Grafikkarte bei Single- und Double-Precision 3 Mal schneller als die Fermi-Grafikkarte [81]. Der Energieverbrauch der Kepler-Grafikkarte hat sich im Vergleich zur Fermi-Grafikkarte nur geringfügig erhöht (Kepler-Grafikkarte Tesla K20X: 235 Watt [82], Fermi-Grafikkarte Tesla M2090: 225 Watt [83]). Die drei Mal höhere Leistung bei Double-Precision wird nicht nur durch eine Erhöhung der Peak-Leistung erreicht (Kepler-Grafikkarte Tesla K20X: 1,31 TFLOP/s [80], Fermi-Grafikkarte Tesla M2090: 0,665 TFLOP/s [78]), sondern auch durch eine Erhöhung der Effizienz [77]. Bei der Ausführung der Nvidia-DGEMM-Funktion mit der Kepler-Grafikkarte beträgt die Effizienz 93% ($=1,22 \text{ TFLOP/s DGEMM} / 1,31 \text{ TFLOP/s Peak-Leistung} * 100$) [81]. Im Vergleich dazu beträgt die Effizienz bei der Fermi-Grafikkarte bei der Ausführung der Nvidia-DGEMM-Funktion 65% ($=0,43 \text{ TFLOP/s DGEMM} / 0,665 \text{ TFLOP/s Peak-Leistung} * 100$) [78] [81]. Bei Single-Precision gibt es auch eine Verbesserung der Effizienz mit Kepler. Bei der Ausführung der Nvidia-SGEMM-Funktion beträgt die Effizienz 73% ($=2,9 \text{ TFLOP/s SGEMM} / 3,95 \text{ TFLOP/s Peak-Leistung} * 100$) [81] bei der Kepler-Grafikkarte

und 67% (=0,89 TFLOP/s SGEMM/1,331 TFLOP/s Peak-Leistung*100) [78] [81] bei der Fermi-Grafikkarte.

Mit der Compute Capability 3.5 gibt es mit Dynamic Parallelism und Hyper-Q zwei völlig neue Features [77]. Mit Dynamic Parallelism kann im GPU/Kernel-Code ein weiterer Kernel gestartet werden [77]. Um Dynamic Parallelism zu ermöglichen, gibt es bei der Kepler-Architektur eine Grid-Management-Unit auf der GPU [77]. Mit Dynamic Parallelism können Kernels rekursiv aufgerufen werden [77]. Bei der Fermi-Architektur konnten nur Device-Funktionen rekursiv aufgerufen werden (siehe Kapitel 2.4.2.10) [5]. Mit Dynamic Parallelism kann die Anzahl der Kernelaufrufe, die von der CPU aus gestartet werden, reduziert werden (siehe Tabelle 42). Da ein größerer Teil des Programms direkt auf der GPU ausgeführt werden kann [77], reduziert man die Performance-Abhängigkeit von der CPU.

Tabelle 42: Reduzierung der Kernelaufrufe im CPU-Code durch Dynamic Parallelism (Inhalt von [84])

LU decomposition (Fermi)		LU decomposition (Kepler)	
CPU Code	GPU Code	CPU Code	GPU Code
<pre>dgetrf(N, N) { for j=1 to N for i=1 to 64 idamax<<<>>> memcpy dswap<<<>>> memcpy dscal<<<>>> dger<<<>>> next i memcpy dlaswap<<<>>> dtrsm<<<>>> dgemm<<<>>> next j }</pre>	<pre>idamax(); dswap(); dscal(); dger(); dlaswap(); dtrsm(); dgemm();</pre>	<pre>dgetrf(N, N) { dgetrf<<<>>> synchronize(); }</pre>	<pre>dgetrf(N, N){ for j=1 to N for i=1 to 64 idamax<<<>>> dswap<<<>>> dscal<<<>>> dger<<<>>> next i dlaswap<<<>>> dtrsm<<<>>> dgemm<<<>>> next j }</pre>

Mit Hyper-Q wurde bei der Kepler-Architektur die Anzahl der Hardware Work Queues zwischen Host und CUDA Work Distributor im Vergleich zu der Fermi-Architektur von 1 auf 32 erhöht [77]. Die 32 Hardware Work Queues können von mehreren Streams, MPI-Prozessen oder Threads innerhalb eines Prozesses verwendet werden [77]. Damit soll eine bessere Auslastung der GPU erreicht werden [77].

Die Anzahl der gleichzeitig ausführbaren Kernels auf einer GPU wurde mit der Kepler-Architektur im Vergleich zu der Fermi-Architektur von 16 auf 32 erhöht [77]. Wie bei Fermi müssen die gleichzeitig auszuführenden Kernels unterschiedlichen Streams zugeordnet sein, jedoch hat bei der Kepler-Architektur jeder Stream eine eigene Hardware Work Queue zwischen Host und CUDA Work Distributor und es muss dadurch kein Multiplexing durchgeführt werden [77]. Durch das Multiplexing bei der Fermi-Architektur kommt es bei der Ausführung mehrerer Kernels pro Stream zu Intra-Stream-Abhängigkeiten [77]. Zum Beispiel bei drei Streams mit je drei Kernels kann nur der letzte Kernel des ersten Streams mit dem ersten Kernel des zweiten Streams und der letzte Kernel des zweiten Streams mit dem ersten Kernel des dritten Streams parallel ausgeführt werden [77]. Durch die 32 Hardware Work Queues bei der Kepler-Architektur kann nun die vollständige Parallelität zwischen den Streams ermöglicht werden [77].

Ein weiteres neues Feature der Kepler-Architektur ist RDMA von Nvidia GPUDirect [77]. Mit RDMA können „3rd Party Devices“ auf den DRAM der GPU zugreifen, wodurch Daten direkt ohne CPU-Data-Buffering zwischen GPUs, die sich auf unterschiedlichen Servern/Knoten befinden, ausgetauscht werden können [77].

11.6.3 Compute Capability 3.0 versus 3.5 und Vergleich mit Fermi

Mit der Kepler-Architektur hat man nicht nur die Leistung weiter gesteigert, sondern auch die Leistung pro Watt und die Effizienz bei der Ausführung deutlich verbessert [81] [85]. Die neuen Multiprozessoren, deren Bezeichnung von SM auf SMX geändert wurde, bieten 2-3 Mal mehr Single-Precision-Peak-Leistung pro Watt, 2 Mal mehr Double-Precision-Peak-Leistung pro Watt und bei Nvidia-DGEMM (=Multiplikation von zwei Matrizen) durch die Verbesserung der Effizienz bei der Ausführung 3 Mal mehr Double-Precision-Leistung pro Watt [77] [81] [85]. Bei GK104/Compute Capability 3.0 wird 2 Mal mehr Single-Precision-Peak-Leistung pro Watt (Vergleich GTX 680 (3090 GFLOP/s, 195 Watt) mit GTX 580 (1581 GFLOP/s, 244 Watt)) erreicht [75] [77] [85]. Bei GK110/Compute Capability 3.5 wird 3 Mal mehr Single- und Double-Precision-Leistung pro Watt bei Nvidia-SGEMM/DGEMM (Vergleich Tesla K20X mit Tesla M2090, siehe zweiten Absatz im Kapitel 11.6.2), 2 Mal mehr Double-Precision-Peak-Leistung pro Watt (Vergleich Tesla K20X (1,31 TFLOP/s, 235 Watt [82]) mit Tesla M2090 (0,665 TFLOP/s [78], 225 Watt [83])) und 3 Mal mehr Single-Precision-Peak-Leistung pro Watt (Vergleich Tesla K20X (3,95 TFLOP/s, 235 Watt [82]) mit Tesla M2090 (1,331 TFLOP/s [78], 225 Watt [83])) erreicht [77] [80] [81].

Die zwei unterschiedlichen Kepler-GPUs GK104 (Compute Capability 3.0) und GK110 (Compute Capability 3.5) sind für unterschiedliche Anwendungen vorgesehen [77]. GPUs mit der Compute Capability 3.0 sind für Gaming [75] [77] (Geforce) und für Seismic Processing, Image Processing, Signal Processing und Video Analytics [77] [78] (Tesla) geeignet. GPUs mit der Compute Capability 3.5 sind für alle Gebiete von GPU Computing einsetzbar [76] [77]. Da die GK104-GPU eine deutlich niedrigere Double-Precision-Leistung im Vergleich zu der GK110-GPU hat, ist das Einsatzgebiet der GK104-GPU Single-Precision-Anwendungen [76]. Bei der GK104-GPU ist das GFLOP/s-Verhältnis von Peak-Single-Precision-Leistung zu Peak-Double-Precision-Leistung mit 24:1 (2290 GFLOP/s versus 95 GFLOP/s bei der Tesla K10) sehr groß [76]. Im Gegensatz dazu beträgt bei der GK110-GPU das GFLOP/s-Verhältnis von Peak-Single-Precision-Leistung zu Peak-Double-Precision-Leistung 3:1 (3950 GFLOP/s versus 1310 GFLOP/s bei der Tesla K20X) [80]. Damit ist die GK110-GPU sehr gut für Double-Precision-Anwendungen geeignet [80]. Beim ECC-Schutz gibt es einen weiteren Unterschied zwischen der GK104-GPU und der GK110-GPU der Tesla-Grafikkarten. Während bei der GK110-GPU der externe Device Memory/DRAM und alle internen Speicher auf der GPU mit ECC geschützt sind, bietet die GK104-GPU nur ECC-Schutz für den externen Device Memory/DRAM [76].

Mit der Kepler-Architektur wurden im Vergleich zur Fermi-Architektur folgende Änderungen durchgeführt. Mit Kepler können statt 48 nun 64 Warps pro Multiprozessor aktiv gehalten werden [77]. Damit erhöht sich die maximale Anzahl der aktiven Threads pro Multiprozessor von 1536 Threads auf 2048 Threads [77]. Da die Anzahl der Kerne pro Multiprozessor jedoch von 32 Kernen auf 192 Kerne [77] deutlich stärker als die Anzahl der aktiven Threads erhöht wurde, sind die Ersatz-Threads, die ausgeführt werden können, wenn auf Daten vom Device Memory/DRAM gewartet wird, deutlich reduziert worden. Pro Kern hat man mit Fermi 48 (=1536/32) aktive Threads und mit Kepler nur mehr 10 (=2048/192) aktive Threads. Mit Kepler wurde erstmals seit der Einführung von CUDA-kompatiblen GPUs die Anzahl der aktiven Blöcke pro Multiprozessor von 8 auf 16 erhöht [5] [77]. Bei 16 aktiven Thread-Blöcken pro Multiprozessor muss ein Block aus 128 Threads (=2048/16) bestehen. Bei der Fermi sind es mit maximal 8 aktiven Thread-Blöcken 192 (=1536/8) Threads pro Block. Die Größe des Register Memory wurde mit Kepler verdoppelt [77]. Bei maximaler Anzahl von aktiven Threads pro Multiprozessor hat man bei Kepler 32 (=65536/2048) und bei Fermi 21 (=32768/1536) 4-Byte-Register (=float, int) pro Thread zur Verfügung. Mit Compute Capability 3.5 der Kepler-

Architektur können statt maximal 63 Register pro Thread nun 255 Register pro Thread verwendet werden [77]. Bei der Auswahl der Shared-Memory-Größe und L1-Cache-Größe kann bei Kepler nun neben 16KB und 48KB auch 32 KB gewählt werden [77].

Mit Kepler wurde pro Multiprozessor die Anzahl der SFUs um den Faktor 8 (von 4 auf 32), die Anzahl der Kerne um den Faktor 6 (von 32 auf 192) und die Anzahl der LD/ST-Einheiten um den Faktor 2 (von 16 auf 32) erhöht [75]. Ebenfalls wurde mit Kepler die Anzahl der Warp Schedulers pro Multiprozessor von 2 auf 4 erhöht [75]. Jeder Warp Scheduler kann zwei unterschiedliche Instruktionen pro Warp gleichzeitig ausführen [77]. Somit sind 8 unterschiedliche Instruktionen pro Takt pro Multiprozessor möglich.

Um den Energieverbrauch der GPU zu senken, wurde bei der Kepler-Architektur der Shadertakt entfernt und die Kerne arbeiten mit der Chiptaktfrequenz [75]. Die Chiptaktfrequenz wurde mit Kepler zwar von 772 MHz [18] (GTX 580) auf 1006 MHz [79] (GTX 680) gesteigert, jedoch liegt sie deutlich unter der Shadertaktfrequenz von 1544 MHz [18] (GTX 580) der Vorgängerarchitektur Fermi. Mit der Senkung der Taktfrequenz der Kerne und der gleichzeitigen Erhöhung der Anzahl der Kerne geht man denselben Weg wie AMD-GPUs, die ebenfalls viele Kerne bei niedriger Taktfrequenz haben [53].

Bei der Kepler-Architektur gibt es eine Basistaktfrequenz (=Basis-Taktung in der Spezifikation [79]) und eine Boosttaktfrequenz (=Boost-Taktung in der Spezifikation [79]) [75]. Die Basistaktfrequenz für die GTX 680 ist 1006 MHz [75]. Wenn bei der GTX 680 der Energieverbrauch kleiner als 170 Watt ist, wird die Chiptaktfrequenz automatisch erhöht [75]. Im Durchschnitt kann dadurch die Taktfrequenz um 5% auf 1058MHz (=Boosttaktfrequenz) gesteigert werden [75].

Mit Kepler wurde das Scheduling vereinfacht, indem bereits der Compiler ermittelt, wann eine Instruktion im Code zur Ausführung bereit ist [75]. Die ermittelte Latenz-Information wird mit der Instruktion abgespeichert [75].

Die Größe des L2-Cache (768 KB bei Fermi-GPUs [5]) wurde mit der Kepler-Architektur geändert. Die Kepler-GPU GK104 besitzt 512 KB [75] L2-Cache und die Kepler-GPU GK110 hat 1536 KB L2-Cache [77].

Bei der Kepler-Architektur gibt es neben dem L1-Cache und dem Shared Memory einen 48KB Read-Only Data Cache pro Multiprozessor [77].

11.7 GPU als Teil der CPU, APU (=Accelerated Processing Unit)

Aktuelle CPUs von Intel und AMD besitzen neben CPU-Kernen auch GPU-Kerne. AMD bezeichnet diese neuen Hybrid-Chips als APUs (=Accelerated Processing Unit) [2]. Intel hat ab Sandy-Bridge begonnen, eine GPU in CPUs zu integrieren [3].

Der Vorteil der Integration einer GPU in CPUs ist, dass keine Daten zwischen dem Arbeitsspeicher der CPU und dem Arbeitsspeicher der GPU übertragen werden müssen, da CPU- und GPU-Kerne denselben Arbeitsspeicher benutzen.

Bei Intel wird der Last Level Cache / Shared L3 Cache gemeinsam von den CPU-Kernen und den GPU-Kernen benutzt [86] [87].

Auf Intel- und AMD-Prozessoren können OpenCL-Programme ausgeführt werden [9] [10].

Auf [9] findet man alle Informationen zur Ausführung von OpenCL-Programmen auf AMD-Prozessoren. Die aktuelle Version des SDKs hat den Namen „AMD APP SDK 2.7“ (Stand: November 2012) [9]. APP steht für „Accelerated Parallel Processing“ [9]. Das SDK unterstützt OpenCL 1.2 [9]. Die CPU-Komponente des SDKs läuft auf AMD-x86-CPUs, die SSE2.x oder höher unterstützen [88]. Die GPU-Komponente des SDKs läuft auf AMD-GPUs ab der 5000-Serie [88]. Bei den AMD-APUs kann neben der CPU auch die integrierte GPU mit OpenCL verwendet werden [89]. Das SDK kann auf Windows 8 (32-bit/64-bit), Windows 7 (32-bit/64-bit), Windows Vista SP2 (32-bit/64-bit), openSUSE 11.x (32-bit/64-bit), Ubuntu 11.04 (32-bit/64-bit) und Red Hat Enterprise Linux 6.x (32-bit/64-bit) installiert werden [88]. Auf [89] findet man Antworten zu häufig gestellten Fragen zu dem SDK.

Auf [10] befinden sich alle Informationen, um ein OpenCL-Programm auf Intel-Prozessoren ausführen zu können. Die aktuelle Version des SDKs hat den Namen „Intel SDK for OpenCL* Applications 2012“ (Stand: November 2012) [10]. Die nächste Version „Intel SDK for OpenCL* Applications 2013“ ist derzeit als Beta verfügbar (Stand: November 2012) [10]. Mit den neuen Intel-Prozessoren (Ivy Bridge/dritte Generation der Intel-Core-Prozessoren), dem neuen SDK und Windows 7 ist es möglich, dass ein OpenCL-Programm auf den CPU-Kernen und auf den GPU-Kernen ausgeführt werden kann [90]. Man hat somit 2 Devices zur Verfügung (CL_DEVICE_TYPE_GPU: Intel HD Graphics und CL_DEVICE_TYPE_CPU: Intel CPU) [90]. Das SDK unterstützt OpenCL 1.1 [10]. OpenCL funktioniert auf CPUs, die Intel SSE 4.1 oder höher unterstützen [90]. Das Intel-OpenCL-SDK kann auf Windows 7 (32-bit/64-bit), Windows Vista SP2 (32-bit/64-bit), Windows Server 2008 R2, Novell SUSE Linux Enterprise Server 11 SP1 (64-bit) und Red Hat Enterprise Linux 6.1 (64-bit) installiert werden [90]. Auf [91] findet man Antworten zu häufig gestellten Fragen zu dem SDK.

12 Zusammenfassung

In der Masterarbeit wurde auf die Nvidia-Fermi-GPU-Architektur und auf die Änderungen gegenüber der Vorgängerarchitektur GT200 näher eingegangen. Es wurden die GPU-Programmiersprachen CUDA und OpenCL erklärt und die Unterschiede bei der Programmierung gegenübergestellt.

In der Masterarbeit wurden zwei Life-Science-Algorithmen (Needleman-Wunsch/Smith-Waterman und Direct Coulomb Summation) für die Ausführung auf Grafikkarten (Nvidia-Fermi und Nvidia-GT200) bei Verwendung der Programmiersprachen CUDA und OpenCL optimiert. Während der DCS-Algorithmus aufgrund der deutlich höheren Parallelität bereits ohne Optimierungen auf der GPU schneller als auf der CPU ausgeführt werden konnte, mussten beim NW-Algorithmus Optimierungen angewendet werden, um dieses Ziel zu erreichen. In der Masterarbeit wurde ermittelt, wie wirksam einzelne Optimierungstechniken bei unterschiedlichen GPU-Architekturen (Fermi und GT200) und bei unterschiedlichen GPU-Programmiersprachen (CUDA und OpenCL) sind.

Bei der Frage, ob bei beiden Algorithmen bei der Ausführung auf der GPU ein Speedup gegenüber der Ausführung auf der CPU erreicht werden kann, wurde ein fairer Vergleich durchgeführt. Dazu wurden auch Optimierungen am CPU-Programm vorgenommen. Es wurde auf die bestmögliche Kompilierung geachtet (icc-Compiler, -O3-Compilerflag und -xSSE3-Compilerflag) und es wurden mit der Programmiersprache OpenMP alle Kerne der CPU verwendet. Außerdem wurde für den Vergleich die Gesamtlaufzeit und nicht die Kernellaufzeit verwendet. Die Gesamtlaufzeit inkludiert neben der Kernellaufzeit auch die Laufzeit für das Anlegen und Freigeben der Arrays im CPU- und GPU-Arbeitsspeicher und die Laufzeit für die Übertragungen zwischen CPU- und GPU-Arbeitsspeicher. Da beim NW-Algorithmus eine große Datenmenge über den PCI-Bus übertragen wird, ist die Kernellaufzeit deutlich kleiner als die Gesamtlaufzeit. Damit spielt bei Optimierungen beim NW-Algorithmus nicht nur die Kernellaufzeit eine Rolle, sondern auch die Übertragungszeit zwischen CPU- und GPU-Arbeitsspeicher.

Es wurde untersucht, ob die mit Fermi eingeführte Cache-Hierarchie über den Global Memory es möglich macht, mit sehr wenig Aufwand ein CPU-Programm in ein GPU-Programm zu transformieren und dennoch keine Einbußen bei der Performance zu haben. Das Ergebnis ist, dass Optimierungstechniken, wie Memory Coalescing und Verwendung der kleinen on-chip-Speicher (Constant Memory und Shared Memory), nach wie vor relevant sind, jedoch fällt der Performancegewinn durch diese Optimierungstechniken mit Fermi deutlich niedriger aus.

Da sich die Anzahl der Kerne bei der Fermi-Architektur gegenüber der GT200-Architektur verdoppelt hat, die Taktfrequenz in etwa gleich geblieben ist und ein Global-Memory-Cache eingeführt wurde, würde man mindestens eine Halbierung der Laufzeit erwarten. Während bei dem NW-Algorithmus bei Betrachtung der Kernellaufzeit dieses Ziel erreicht wurde, konnte beim DCS-Algorithmus dieses Ziel nicht erreicht werden. Beim NW-Algorithmus konnte durch die Vergrößerung des Shared Memory bei Fermi (48 KB pro SM) gegenüber GT200 (16 KB pro SM) die Blockgröße deutlich vergrößert werden. Im Gegensatz dazu konnten beim DCS-Algorithmus keine Verbesserungen am Code für eine niedrigere Laufzeit speziell für die Fermi-Architektur gefunden werden.

In der Masterarbeit wurde festgestellt, dass es einen Laufzeitunterschied zwischen CUDA- und OpenCL-Programmen gibt. Größere Laufzeitunterschiede wurden bei Constant Memory und bei der Übertragung von Daten zwischen einem Pageable-Host-Memory-Array vom CPU-Arbeitsspeicher und einem Array im GPU-Arbeitsspeicher festgestellt. Das zweite Problem trifft nur auf die GeForce-GPUs zu. Da das CUDA-Programm fast immer schneller als das OpenCL-Programm war, ist bei Nvidia-GPUs noch immer CUDA zu bevorzugen. Der große Vorteil eines OpenCL-Programms ist jedoch, dass man es auch auf AMD-GPUs, Intel-CPUs und AMD-CPU-GPUs ausführen kann.

Literaturverzeichnis

- [1] D. B. Kirk und W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, San Francisco, CA: Morgan Kaufmann Publishers Inc., 2010.
- [2] AMD, „Fusion Whitepaper,“ 2010. [Online]. Available: http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf. [Zugriff am 05 04 2012].
- [3] Intel, „Intel Microarchitecture Codename Sandy Bridge,“ [Online]. Available: <http://www.intel.com/content/www/de/de/architecture-and-technology/microarchitecture/intel-microarchitecture-codename-sandy-bridge.html>. [Zugriff am 05 04 2012].
- [4] J. Dongarra, H. Meuer, H. Simon und E. Strohmaier, „Top500 List - November 2012,“ 2012. [Online]. Available: <http://www.top500.org/list/2012/11/>. [Zugriff am 06 02 2013].
- [5] Nvidia, „Nvidia CUDA C Programming Guide, Version 4.0,“ 6.Mai 2011.
- [6] Nvidia, „OpenCL Programming Guide for the CUDA Architecture, Version 4.0,“ 14.Februar 2011.
- [7] Nvidia, „Next Generation GPU Architecture Code-named ‘Fermi’,“ 2009. [Online]. Available: http://www.nvidia.com/content/GTC/documents/SC09_Fermi_Overview.pdf. [Zugriff am 05 05 2012].
- [8] M. Locktyukhin, Intel, „sse execution units in core duo,“ 2008. [Online]. Available: <http://software.intel.com/en-us/forums/showpost.php?p=60696>. [Zugriff am 18 07 2012].
- [9] AMD, „Heterogeneous Computing > Accelerated Parallel Processing (APP) SDK,“ 2012. [Online]. Available: <http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/>. [Zugriff am 19 11 2012].
- [10] Intel, „Intel SDK for OpenCL* Applications 2012,“ 2012. [Online]. Available: <http://software.intel.com/en-us/vcs/source/tools/opencv-sdk>. [Zugriff am 19 11 2012].
- [11] S. Che und K. Skadron, „Needleman-Wunsch Sequence Alignment,“ 2009. [Online]. Available: http://www.nvidia.de/object/cuda_apps_flash_new_de.html#state=detailsOpen;aid=5a4a9940-b346-454f-926b-fc21e5e9995b. [Zugriff am 19 01 2010].
- [12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer und K. Skadron, „A performance study of general-purpose applications on graphics processors using CUDA,“ *Elsevier, Journal of Parallel and Distributed Computing*, Bd. 68, Nr. 10, pp. 1370-1380, Oktober 2008.

- [13] Rodinia, „Needleman-Wunsch, CUDA code,“ 2009. [Online]. Available: <https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Downloads>. [Zugriff am 19 01 2010].
- [14] D. Dilch und E. Mehofer, „Optimization Techniques and Performance Analyses of Two Life Science Algorithms for Novel GPU Architectures,“ in *20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Garching, 15-17 Feb. 2012.
- [15] Nvidia, „Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture Fermi,“ 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Zugriff am 05 03 2012].
- [16] Nvidia, „GeForce GTX 200 GPU Architectural Overview,“ 2008. [Online]. Available: http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf. [Zugriff am 25 05 2012].
- [17] Nvidia, „Technical Brief NVIDIA GeForce 8800 GPU Architecture Overview,“ 2008. [Online]. Available: http://www.ece.lsu.edu/koppel/gp/refs/GeForce_8800_GPU_Architecture_Technical_Brief.pdf. [Zugriff am 14 06 2012].
- [18] Nvidia, „GeForce GTX 580,“ [Online]. Available: <http://www.nvidia.de/object/product-geforce-gtx-580-de.html>, Tab "Technische Daten". [Zugriff am 09 06 2012].
- [19] Nvidia, „What advantages does Tesla 20-series have for GPU Computing?,“ April 2010. [Online]. Available: http://nvidia.custhelp.com/app/answers/detail/a_id/2595. [Zugriff am 05 05 2012].
- [20] Nvidia, „CUDA Occupancy Calculator 2.3,“ 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/4_0_rc2/sdk/docs/CUDA_Occupancy_Calculator.xls. [Zugriff am 04 04 2012].
- [21] Nvidia, „Geforce GTX 480,“ [Online]. Available: http://www.nvidia.de/object/product_geforce_gtx_480_de.html, Tab „Technische Daten“. [Zugriff am 05 04 2012].
- [22] Intel, „Intel Xeon Processor X5550,“ [Online]. Available: <http://ark.intel.com/products/37106/>. [Zugriff am 05 05 2012].
- [23] Nvidia, „CUDA C Best Practices Guide, Version 4.0,“ Mai 2011.
- [24] Nvidia, „Local Memory and Register Spilling,“ 2011. [Online]. Available: http://developer.download.nvidia.com/CUDA/training/register_spilling.pdf. [Zugriff am 13 06 2012].

- [25] Nvidia, „Grafikprozessor Tesla C2050/C2070,“ [Online]. Available: http://www.nvidia.de/object/product_tesla_C2050_C2070_de.html, Tab „Technische Daten“ und Tab „Merkmale“. [Zugriff am 15 04 2012].
- [26] Nvidia, „NVIDIA Update June 2010,“ 2010. [Online]. Available: http://www.di.uminho.pt/mcss2010/slides/mcss2010_lect12_cuda3_fermi.pdf. [Zugriff am 05 05 2012].
- [27] Nvidia, „Tuning CUDA Applications for Fermi, Version 1.3,“ 20.August 2010.
- [28] Nvidia, „NVIDIA CUDA Toolkit 3.2 - GPU Computing SDK code samples,“ 2010.
- [29] H. Ney, RWTH Aachen, „Betriebssysteme und Systemsoftware - Virtueller Speicher (Paging),“ 28.Jänner 2008. [Online]. Available: http://www-i6.informatik.rwth-aachen.de/web/Teaching/Lectures/SS11/BuS/script/SysPro_Slides_WS07.Kap8.pdf. [Zugriff am 24 1 2013].
- [30] Nvidia, „GPU Tesla K20,“ November 2012. [Online]. Available: <http://www.nvidia.com/object/workstation-solutions-tesla.html>. [Zugriff am 13 11 2012].
- [31] M. Zahran, „Lecture 9: Multi-GPU Systems,“ März 2012. [Online]. Available: <http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture9.pdf>. [Zugriff am 30 10 2012].
- [32] P. Micikevicius, Nvidia, „Multi-GPU Programming,“ August 2011. [Online]. Available: http://developer.download.nvidia.com/CUDA/training/cuda_webinars_multi_gpu.pdf. [Zugriff am 18 12 2012].
- [33] T. C. Schroeder, Nvidia, „Peer-to-Peer & Unified Virtual Addressing,“ September 2011. [Online]. Available: http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf. [Zugriff am 18 12 2012].
- [34] Nvidia, „Advanced CUDA Webinar Memory Optimizations,“ 2009. [Online]. Available: http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf. [Zugriff am 09 04 2012].
- [35] Nvidia, „Warum Tesla,“ [Online]. Available: <http://www.nvidia.de/page/why-choose-tesla.html>. [Zugriff am 11 04 2012].
- [36] Nvidia, „Nvidia Tesla Personal Supercomputer,“ Dezember 2008. [Online]. Available: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_PSC_US_Dec08_LowRes.pdf. [Zugriff am 11 04 2012].
- [37] E. R. Harold, IBM, „Units of last precision,“ 2009. [Online]. Available: <http://public.dhe.ibm.com/software/dw/java/j-math2-pdf.pdf>. [Zugriff am 14 06 2012].

- [38] Nvidia, „CUDA-GDB (NVIDIA CUDA Debugger) User Manual,“ 14.Oktober 2010.
- [39] Nvidia, „Cuda-memcheck User Manual,“ 10.August 2010.
- [40] Nvidia, „Cuda Toolkit Archive,“ Oktober 2012. [Online]. Available:
<http://developer.nvidia.com/cuda-toolkit-archive>. [Zugriff am 13 11 2012].
- [41] Khronos Group, „OpenCL - The open standard for parallel programming of heterogeneous systems,“ November 2011. [Online]. Available: <http://www.khronos.org/opencv/>. [Zugriff am 14 04 2012].
- [42] Nvidia, „Nvidia OpenCL JumpStart Guide, Version 1.0,“ 19.Februar 2010.
- [43] Khronos Group, „OpenCL API 1.2 Reference Card,“ November 2011. [Online]. Available:
<http://www.khronos.org/files/opencv-1-2-quick-reference-card.pdf>. [Zugriff am 16 04 2013].
- [44] Khronos Group, „clEnqueueMapBuffer,“ 2009. [Online]. Available:
<http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clEnqueueMapBuffer.html>.
[Zugriff am 5 10 2012].
- [45] Nvidia, „OpenCL Best Practices Guide,“ 27.Mai 2010.
- [46] Nvidia, „Nvidia CUDA C Programming Guide, Version 3.2,“ 9.November 2010.
- [47] Khronos Group, „clFinish,“ 2009. [Online]. Available:
<http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clFinish.html>. [Zugriff am 2012 10 2].
- [48] Khronos Group, „cl_khr_fp64 - Support for double floating-point precision,“ 2009. [Online]. Available: http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/cl_khr_fp64.html.
[Zugriff am 14 01 2013].
- [49] M. Fried, Microway, „GPGPU Architecture Comparison of ATI and NVIDIA GPUs,“ Juni 2010. [Online]. Available:
http://www.microway.com/pdfs/GPGPU_Architecture_and_Performance_Comparison.pdf.
[Zugriff am 05 04 2012].
- [50] Nvidia, „GeForce GTX 285,“ [Online]. Available:
http://www.nvidia.de/object/product_geforce_gtx_285_de.html, Tab "Technische Daten".
[Zugriff am 05 05 2012].
- [51] Intel, „Intel Xeon Processor 5500 Series Datasheet, Volume 1,“ Juni 2011. [Online]. Available:
<http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-5500-vol-1-datasheet.pdf>. [Zugriff am 09 06 2012].

- [52] P. N. Glaskowsky, „Nvidia’s Fermi: The First Complete GPU Computing Architecture,“ September 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf. [Zugriff am 05 05 2012].
- [53] Amd, „Ati Radeon HD 5870 GPU,“ [Online]. Available: <http://www.amd.com/uk/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx>. [Zugriff am 05 05 2012].
- [54] Amd, „GPU Computing: Past, Present and Future,“ März 2010. [Online]. Available: http://developer.amd.com/gpu_assets/GPU%20Computing%20-%20Past%20Present%20and%20Future%20with%20ATI%20Stream%20Technology.pdf. [Zugriff am 05 05 2012].
- [55] Universität Wien, Faculty of Computer Science, Research Group Scientific Computing, „Hardware Infrastructure,“ [Online]. Available: <http://www.par.univie.ac.at/index.php?goto=hardware>. [Zugriff am 05 05 2012].
- [56] Nvidia, „Tesla C1060 Computing-Prozessor,“ [Online]. Available: http://www.nvidia.de/object/tesla_c1060_de.html. [Zugriff am 05 05 2012].
- [57] Technische Universität Chemnitz, „Laufzeit-Messung,“ [Online]. Available: <http://www.tu-chemnitz.de/urz/kurse/unterlagen/C/kap3/zeitmess.htm>. [Zugriff am 14 06 2012].
- [58] S. Kosub, „Einführung in die Informatik 1,“ November 2008. [Online]. Available: <http://www.inf.uni-konstanz.de/algo/lehre/ws08/ei/aufbau-und-funktionsweise-www.pdf>. [Zugriff am 26 11 2012].
- [59] S. A. Manavski und G. Valle, „CUDA compatible GPU cards as efficient hardware accelerator for Smith-Waterman sequence alignment,“ *BMC Bioinformatics*, März 2008.
- [60] A. Hansen, Bioinformatik: Ein Leitfaden für Naturwissenschaftler, Basel, Schweiz: Birkhäuser Verlag, 2. überarbeitete und erweiterte Auflage, 2004.
- [61] A. Khajeh-Saeed, S. Poole und J. B. Perot, „Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors,“ *Journal of Computational Physics*, Bd. 229, Nr. 11, p. 4247–4258, Februar 2010.
- [62] U. Leser, Humboldt Universität Berlin, „Bioinformatik, Dynamische Programmierung, Globale und lokale Alignierung,“ 2005. [Online]. Available: http://www.ki.informatik.hu-berlin.de/wbi/teaching/archive/ws0506/bioinformatik-bpi/04_alignments.pdf. [Zugriff am 18 04 2012].
- [63] NCBI, „Matrix made by matblas from blosum62.iij,“ Mai 2002. [Online]. Available: <http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt>. [Zugriff am 27 04 2012].

- [64] R. Hirt, „Models of protein evolution: an introduction to amino acid exchange matrices,“ 2003. [Online]. Available: http://www.ch.embnet.org/CoursEMBnet/PHYL03/Slides/Protein_I_MDMs_4.pdf. [Zugriff am 19 06 2012].
- [65] N. Casagrande, „Basic-Algorithms-of-Bioinformatics Applet,“ Mai 2010. [Online]. Available: <http://baba.sourceforge.net/>. [Zugriff am 22 11 2012].
- [66] Intel, „Targeting IA-32 and Intel(R) 64 Architecture Processors Automatically,“ 2010. [Online]. Available: http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/optaps/common/optaps_dsp_targ.htm. [Zugriff am 18 04 2012].
- [67] Nvidia, „Life Sciences and CUDA,“ 2009. [Online]. Available: http://www.lunarc.lu.se/Documents/nvidia-workshop/files/presentation/90_Life_Sciences.pdf. [Zugriff am 25 04 2012].
- [68] W.-m. W. Hwu und D. Kirk, Berkeley Winter School, „Advanced Algorithmic Techniques for GPUs, Lecture 2: Parallelism Scalability Transformations,“ Jänner 2011. [Online]. Available: http://iccs.lbl.gov/assets/docs/2011-01-24/lecture2_parallelism_scalability_Berkeley_2011.pdf. [Zugriff am 25 04 2012].
- [69] Kronos Group, „clBuildProgram,“ 2009. [Online]. Available: <http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clBuildProgram.html>. [Zugriff am 15 03 2012].
- [70] G. Ziegler, CINECA / Bologna, Italy, „OpenCL on the GPU,“ Oktober 2011. [Online]. Available: http://corsi.cineca.it/courses/scuolaAvanzata/Ziegler-OpenCL/Part1_OpenCL_on_the_GPU.pdf. [Zugriff am 26 12 2012].
- [71] Nvidia, „CUDA 4.1,“ 2011. [Online]. Available: http://developer.download.nvidia.com/CUDA/training/CUDA_4_1_Webinar_v11-11-22.pdf. [Zugriff am 02 05 2012].
- [72] Nvidia, „CUDA Toolkit 4.2,“ 2012. [Online]. Available: <http://developer.nvidia.com/cuda-downloads>. [Zugriff am 04 06 2012].
- [73] Nvidia, „CUDA 5 - Production Release,“ 2012. [Online]. Available: <http://developer.nvidia.com/cuda-toolkit>. [Zugriff am 13 11 2012].
- [74] Nvidia, „Visual Profiler,“ [Online]. Available: <http://developer.nvidia.com/nvidia-visual-profiler>. [Zugriff am 04 06 2012].
- [75] Nvidia, „Whitepaper NVIDIA GeForce GTX 680,“ 2012. [Online]. Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf. [Zugriff am 28 03 2012].

- [76] Nvidia, „Nvidia Tesla Kepler GPU Computing Accelerators,“ 2012. [Online]. Available: http://www.nvidia.com/content/tesla/pdf/NV_DS_TeslaK_Family_May_2012_LR.pdf. [Zugriff am 14 06 2012].
- [77] Nvidia, „Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110,“ 2012. [Online]. Available: <http://www.nvidia.de/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. [Zugriff am 09 06 2012].
- [78] Nvidia, „Tesla GPU Computing Solutions for Servers,“ 2012. [Online]. Available: <http://www.nvidia.com/object/tesla-servers.html>. [Zugriff am 14 06 2012].
- [79] Nvidia, „NVIDIA GeForce GTX 680,“ 2013. [Online]. Available: <http://www.nvidia.de/object/geforce-gtx-680-de.html#pdpContent=2>. [Zugriff am 04 02 2013].
- [80] Nvidia, „Tesla Kepler GPU Accelerators,“ Oktober 2012. [Online]. Available: <http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf>. [Zugriff am 13 11 2012].
- [81] Nvidia, „NVIDIA Tesla K20-K20X GPU Accelerators Benchmarks,“ Dezember 2012. [Online]. Available: <http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf>. [Zugriff am 02 02 2013].
- [82] Nvidia, „Tesla K20X GPU Accelerator,“ November 2012. [Online]. Available: <http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf>. [Zugriff am 02 02 2013].
- [83] Nvidia, „Tesla M2090,“ Juni 2011. [Online]. Available: <http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf>. [Zugriff am 02 02 2013].
- [84] Nvidia, „Introducing CUDA 5,“ September 2012. [Online]. Available: <http://developer.download.nvidia.com/GTC/introducing-cuda-5.pdf>. [Zugriff am 16 10 2012].
- [85] Nvidia, „NVIDIA präsentiert neue Grafikkarten-Generation mit Kepler-Architektur,“ März 2012. [Online]. Available: <http://www.nvidia.de/object/nvidia-geforce-gpu-kepler-architecture-20120322-de.html>. [Zugriff am 09 06 2012].
- [86] Intel, „Intel HD Graphics Provides Next Generation Integrated Graphics,“ [Online]. Available: <http://www.intel.com/content/www/us/en/architecture-and-technology/hd-graphics/hd-graphics-consumer-general.html>. [Zugriff am 02 05 2012].
- [87] Intel, „Graphics Software Stack Evolution,“ 2011. [Online]. Available: http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS006_100F.pdf. [Zugriff am 11 06 2012].

- [88] AMD, „Accelerated Parallel Processing (APP) SDK > System Requirements & Driver Compatibility,“ 2012. [Online]. Available: <http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/system-requirements/>. [Zugriff am 19 11 2012].
- [89] AMD, „AMD APP SDK v2.7 OpenCL FAQ,“ 2012. [Online]. Available: http://developer.amd.com.php53-23.ord1-1.websitetestlink.com/wordpress/media/2012/10/AMD_APP_SDK_FAQ.pdf. [Zugriff am 19 11 2012].
- [90] Intel, „Intel SDK for OpenCL* Applications 2012 Release Notes,“ 2012. [Online]. Available: <http://software.intel.com/en-us/articles/opencl-release-notes/>. [Zugriff am 11 06 2012].
- [91] Intel, „Intel OpenCL SDK - Frequently Asked Questions,“ 2012. [Online]. Available: <http://software.intel.com/en-us/articles/opencl-sdk-frequently-asked-questions/>. [Zugriff am 11 06 2012].

Anhang A Programmcodes

In diesem Anhang befinden sich die Programmcodes, auf die in der Masterarbeit verwiesen wird.

A.1 Needleman-Wunsch CPU-Code Schritt 3 (traceback)

Mit Code 73 wird ein traceback auf die Ergebnismatrix (=scorematrixCPU im Code) auf der CPU durchgeführt. Das Ergebnis ist das Alignment (alignment_Sequenz1 und alignment_Sequenz2 im Code) der beiden Sequenzen.

Code 73: Schritt 3 (traceback) des NW-CPU-Codes [13] (adaptiert)

```
int max_rows=Sequenzgroesse1+1;
int max_cols=Sequenzgroesse2+1;
int LIMIT=-99999;
int Groesse_des_Alignments=0;

//Groesse der Alignment-Arrays ermitteln
for (i = max_rows - 1, j = max_cols - 1; i >= 0, j >= 0;) {
    int nw, n, w, traceback;
    int aktuelles_Element=scorematrixCPU[i * max_cols + j];

    if(i == 0 && j == 0)
        break;

    if (i > 0 && j > 0) {
        nw = scorematrixCPU[(i - 1) * max_cols + j - 1] +
            h_Substitutionsmatrix_Aminosaeuren[h_Sequenz1[i-1]][h_Sequenz2[j-1]];
        w = scorematrixCPU[i * max_cols + j - 1]-penalty;
        n = scorematrixCPU[(i - 1) * max_cols + j]-penalty;
    } else if (i == 0) {
        nw = n = LIMIT;
        w = aktuelles_Element;
    } else if (j == 0) {
        nw = w = LIMIT;
        n = aktuelles_Element;
    } else {}

    if (aktuelles_Element == nw) {
        i--;
        j--;
        Groesse_des_Alignments++;
        continue;
    } else if (aktuelles_Element == w) {
        j--;
        Groesse_des_Alignments++;
        continue;
    } else if (aktuelles_Element == n) {
        i--;
        Groesse_des_Alignments++;
        continue;
    } else ;
    break;
}
```

Fortsetzung von Code 73 auf der nächsten Seite

Fortsetzung von Code 73:

```
//Alignment durchfuehren
int* alignment_Sequenz1=(int*)malloc(Groesse_des_Alignments*sizeof(int));
int* alignment_Sequenz2=(int*)malloc(Groesse_des_Alignments*sizeof(int));
int index_alignment=Groesse_des_Alignments-1;

for (i = max_rows - 1, j = max_cols - 1; i >= 0, j >= 0;) {
    int nw, n, w, traceback;
    int aktuelles_Element=scorematrixCPU[i * max_cols + j];
    if (i == 0 && j == 0)
        break;
    if (i > 0 && j > 0) {
        nw = scorematrixCPU[(i - 1) * max_cols + j - 1] +
            h_Substitutionsmatrix_Aminosaeuren[h_Sequenz1[i-1]][h_Sequenz2[j-1]];
        w = scorematrixCPU[i * max_cols + j - 1]-penalty;
        n = scorematrixCPU[(i - 1) * max_cols + j]-penalty;
    } else if (i == 0) {
        nw = n = LIMIT;
        w = aktuelles_Element;
    } else if (j == 0) {
        nw = w = LIMIT;
        n = aktuelles_Element;
    } else {}

    //Maximum der drei Werte bestimmen
    traceback=nw;
    if(w>traceback)
        traceback=w;
    if(n>traceback)
        traceback=n;

    if (aktuelles_Element == nw) {
        i--;
        j--;
        alignment_Sequenz1[index_alignment]=h_Sequenz1[i];
        alignment_Sequenz2[index_alignment]=h_Sequenz2[j];
        index_alignment--;
        continue;
    }
    else if (aktuelles_Element == w) {
        j--;
        alignment_Sequenz1[index_alignment]=-1;
        alignment_Sequenz2[index_alignment]=h_Sequenz2[j];
        index_alignment--;
        continue;
    }
    else if (aktuelles_Element == n) {
        i--;
        alignment_Sequenz1[index_alignment]=h_Sequenz1[i];
        alignment_Sequenz2[index_alignment]=-1;
        index_alignment--;
        continue;
    }
    else ;
}

free(alignment_Sequenz1);
free(alignment_Sequenz2);
```

A.2 Needleman-Wunsch GPU-Code (Shared-Memory-Programm) mit allen Optimierungen

Im ersten Schritt wird die Startmatrix initialisiert. Code 74 enthält die Aufrufe der Kernels und Code 75 die Kernels.

Code 74: Kernelaufrufe für die Initialisierung der Startmatrix

```
int Anzahl_von_Bloecken_in_XRichtung,Anzahl_von_Bloecken_in_YRichtung;
DatentypGPU *d_Sequenz1,*d_Sequenz2;
cudaMalloc((void**)&d_Sequenz1,max_rows*sizeof(DatentypGPU));
cudaMalloc((void**)&d_Sequenz2,max_cols*sizeof(DatentypGPU));
cudaMemcpy(d_Sequenz1,Sequenz1,max_rows*sizeof(DatentypGPU),cudaMemcpyHostToDevice);
cudaMemcpy(d_Sequenz2,Sequenz2,max_cols*sizeof(DatentypGPU),cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(Substitutionsmatrix_const,h_Substitutionsmatrix,
    20*20*sizeof(DatentypGPU));

int blocksize=256;
Anzahl_von_Bloecken_in_XRichtung=(max_cols-1)/blocksize;
if(((max_cols-1)%blocksize)!=0)
    Anzahl_von_Bloecken_in_XRichtung+=1;
Anzahl_von_Bloecken_in_YRichtung=(max_rows-1);

dimGrid.x=Anzahl_von_Bloecken_in_XRichtung;
dimGrid.y=Anzahl_von_Bloecken_in_YRichtung;
dimBlock.x=blocksize;
dimBlock.y=1;
reference_cuda_initialisieren_const_memory<<<dimGrid,dimBlock>>>(matrix_cuda,
    d_Sequenz1,d_Sequenz2,max_cols,max_rows);

cudaFree(d_Sequenz1);
cudaFree(d_Sequenz2);

blocksize=256;
if(max_cols%blocksize!=0)
    Anzahl_von_Bloecken_in_XRichtung+=1;
dimGrid.x=Anzahl_von_Bloecken_in_XRichtung;
dimGrid.y=1;
dimBlock.x=blocksize;
dimBlock.y=1;
matrix_cuda_erste_Zeile_initialisieren<<<dimGrid,dimBlock>>>(matrix_cuda,max_cols,
    penalty);

blocksize=256;
Anzahl_von_Bloecken_in_YRichtung=(max_rows/blocksize);
if(max_rows%blocksize!=0)
    Anzahl_von_Bloecken_in_YRichtung+=1;
dimGrid.x=Anzahl_von_Bloecken_in_YRichtung;
dimGrid.y=1;
dimBlock.x=blocksize;
dimBlock.y=1;
matrix_cuda_erste_Spalte_initialisieren<<<dimGrid,dimBlock>>>(matrix_cuda,max_rows,
    max_cols,penalty);
```

Code 75: Kernels für die Initialisierung der Startmatrix

```
__global__ void reference_cuda_initialisieren_const_memory(
    DatentypGPU *reference_cuda, const DatentypGPU *d_Sequenz1,
    const DatentypGPU *d_Sequenz2, const int max_cols, const int max_rows)
{
    int bi = blockIdx.y;
    int bj = blockIdx.x;

    int ti = threadIdx.y;
    int tj = threadIdx.x;

    int i=bi*blockDim.y+ti+1;
    int j=bj*blockDim.x+tj+1;

    if(j<max_cols)
        reference_cuda[i*max_cols+j]=Substitutionsmatrix_const[
            d_Sequenz1[i-1]*20+d_Sequenz2[j-1]];
}

__global__ void matrix_cuda_erste_Zeile_initialisieren(DatentypGPU *matrix_cuda,
    const int max_cols, const DatentypGPU penalty)
{
    int bj = blockIdx.x;
    int tj = threadIdx.x;

    int j=bj*blockDim.x+tj;

    if(j<max_cols)
        matrix_cuda[j]=-j*penalty;
}

__global__ void matrix_cuda_erste_Spalte_initialisieren(DatentypGPU *matrix_cuda,
    const int max_rows, const int max_cols, const DatentypGPU penalty)
{
    int bi = blockIdx.x;
    int ti = threadIdx.x;

    int i=bi*blockDim.x+ti;

    if(i<max_rows)
        matrix_cuda[i*max_cols]=-i*penalty;
}
```

Im zweiten Schritt wird die Ergebnismatrix berechnet. Code 76 enthält die Aufrufe der Kernels und Code 77 die Kernels.

Code 76: Kernelaufrufe für die Berechnung der Ergebnismatrix [13] (adaptiert)

```
dimBlock.x=BLOCK_SIZE;
dimBlock.y=1;
int block_width = ( max_cols - 1 )/BLOCK_SIZE;

//process top-left matrix
for(int i = 1; i <= block_width; i++){
    dimGrid.x = i;
    dimGrid.y = 1;
    needle_cuda_shared_1<<<dimGrid, dimBlock>>>(matrix_cuda, max_cols, penalty, i,
        block_width);
}

//process bottom-right matrix
for(int i = block_width - 1; i >= 1; i--){
    dimGrid.x = i;
    dimGrid.y = 1;
    needle_cuda_shared_2<<<dimGrid, dimBlock>>>(matrix_cuda, max_cols, penalty, i,
        block_width);
}
```

Code 77: Kernels für die Berechnung der Ergebnismatrix [13] (adaptiert)

```
#define Datentyp short
#define SDATA( index)          CUT_BANK_CHECKER(sdata, index)

__device__ __host__ int maximum(int a, int b,int c){
    int k;
    if( a <= b )
        k = b;
    else
        k = a;

    if( k <=c )
        return(c);
    else
        return(k);
}

__global__ void needle_cuda_shared_1(Datentyp* matrix_cuda, int cols,
int penalty, int i, int block_width)
{
    int bx = blockIdx.x;
    int tx = threadIdx.x;

    int b_index_x = bx;
    int b_index_y = i - 1 - bx;
    int index=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x+tx+(cols+1);
    int index_n=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x+tx+1;
    int index_w=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x+(cols);
    int index_nw=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x;

    __shared__ Datentyp temp[BLOCK_SIZE+1][BLOCK_SIZE+1];

    if (tx == 0)
        temp[tx][0] = matrix_cuda[index_nw];
    for ( int ty = 0 ; ty < BLOCK_SIZE ; ty++)
        temp[ty+1][tx+1] = matrix_cuda[index + cols * ty];

    __syncthreads();
    temp[tx + 1][0] = matrix_cuda[index_w + cols * tx];
    __syncthreads();

    temp[0][tx + 1] = matrix_cuda[index_n];
    __syncthreads();

    for( int m = 0 ; m < BLOCK_SIZE ; m++){
        if ( tx <= m ){
            int t_index_x = tx + 1;
            int t_index_y = m - tx + 1;

            temp[t_index_y][t_index_x]=maximum(
                temp[t_index_y-1][t_index_x-1]+temp[t_index_y][t_index_x],
                temp[t_index_y][t_index_x-1]-penalty,temp[t_index_y-1][t_index_x]-penalty);
        }
        __syncthreads();
    }
    for( int m = BLOCK_SIZE - 2 ; m >=0 ; m--){
        if( tx <= m){
            int t_index_x = tx + BLOCK_SIZE - m ;
            int t_index_y = BLOCK_SIZE - tx;

            temp[t_index_y][t_index_x]=maximum(
                temp[t_index_y-1][t_index_x-1]+temp[t_index_y][t_index_x],
                temp[t_index_y][t_index_x-1]-penalty,temp[t_index_y-1][t_index_x]-penalty);
        }
        __syncthreads();
    }
}
```

Fortsetzung von Code 77 auf der nächsten Seite

Fortsetzung von Code 77:

```
for(int ty = 0;ty < BLOCK_SIZE;ty++)
    matrix_cuda[index + ty * cols] = temp[ty+1][tx+1];
}

__global__ void needle_cuda_shared_2(Datentyp* matrix_cuda, int cols, int penalty,
int i, int block_width)
{
    int bx = blockIdx.x;
    int tx = threadIdx.x;

    int b_index_x = bx + block_width - i;
    int b_index_y = block_width - bx -1;

    int index=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x+tx+(cols+1);
    int index_n=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x+tx+(1);
    int index_w=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x+(cols);
    int index_nw=cols*BLOCK_SIZE*b_index_y+BLOCK_SIZE*b_index_x;

    __shared__  Datentyp temp[BLOCK_SIZE+1][BLOCK_SIZE+1];

    for(int ty = 0;ty < BLOCK_SIZE;ty++)
        temp[ty+1][tx+1] = matrix_cuda[index + cols * ty];

    __syncthreads();

    if(tx == 0)
        temp[tx][0] = matrix_cuda[index_nw];

    temp[tx + 1][0] = matrix_cuda[index_w + cols * tx];

    __syncthreads();

    temp[0][tx + 1] = matrix_cuda[index_n];

    __syncthreads();

    for(int m = 0; m < BLOCK_SIZE; m++){
        if(tx <= m){
            int t_index_x = tx + 1;
            int t_index_y = m - tx + 1;

            temp[t_index_y][t_index_x] = maximum(
                temp[t_index_y-1][t_index_x-1]+temp[t_index_y][t_index_x],
                temp[t_index_y][t_index_x-1]-penalty,temp[t_index_y-1][t_index_x]-penalty);
        }
        __syncthreads();
    }

    for(int m = BLOCK_SIZE - 2 ; m >=0 ; m--){
        if(tx <= m){
            int t_index_x = tx + BLOCK_SIZE - m;
            int t_index_y = BLOCK_SIZE - tx;

            temp[t_index_y][t_index_x]=maximum(
                temp[t_index_y-1][t_index_x-1]+temp[t_index_y][t_index_x],
                temp[t_index_y][t_index_x-1]-penalty,temp[t_index_y-1][t_index_x]-penalty);
        }
        __syncthreads();
    }

    for( int ty = 0 ; ty < BLOCK_SIZE ; ty++)
        matrix_cuda[index + ty * cols] = temp[ty+1][tx+1];
}
```

A.3 DCS GPU-Code

Im Code 78 befindet sich der komplette Kernel-Code für die Optimierung „Berechnung mehrerer Gridpunkte in X-Richtung pro Thread“.

Code 78: Berechnung mehrerer Gridpunkte in X-Richtung pro Thread [1] (adaptiert)

```
#define Anzahl_Gridwerte_in_XRichtung_Pro_Thread 4 //4 oder 8

__global__ void dcs_gpu_kernel_5_Erster_Aufgerufener_Kernel(float* energygrid,
dim3 grid,float gridspacing,int zindex,int numatoms)
{
    int xindex=blockIdx.x*blockDim.x*
        Anzahl_Gridwerte_in_XRichtung_Pro_Thread+threadIdx.x*
        Anzahl_Gridwerte_in_XRichtung_Pro_Thread;
    int yindex=blockIdx.y*blockDim.y+threadIdx.y;
    int outaddr=zindex*grid.x*grid.y+yindex*grid.x*xindex;

    float coorx[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
    int i;
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        coorx[i]=gridspacing*(xindex+i);

    float coory=gridspacing*yindex;

    float energyval[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        energyval[i]=0.0f;

    int atomid;
    for(atomid=0;atomid<numatoms;atomid++){
        float dy=coory-atominfo[atomid].y;
        float dysqpdzsq=(dy*dy)+atominfo[atomid].z;
        float x=atominfo[atomid].x;

        float dx[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
        for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
            dx[i]=coorx[i]-x;

        float charge=atominfo[atomid].w;
        for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
            energyval[i]+=charge*rsqrtf(dx[i]*dx[i]+dysqpdzsq);
    }

    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        energygrid[outaddr+i]=energyval[i];
}
```

Fortsetzung von Code 78 auf der nächsten Seite

Fortsetzung von Code 78:

```
//Dieser Kernel wird nach dem „dcs_gpu_kernel_5_Erster_Aufgerufener_Kernel()“-
//Kernel aufgerufen, wenn das Atom-Array größer als 64 KB ist und somit nicht
//komplett in den Constant Memory passt. Mit dem
//„dcs_gpu_kernel_5_Erster_Aufgerufener_Kernel()“-Kernel wird der erste 64 KB-
//Teil des Atom-Arrays und mit dem dcs_gpu_kernel_5()-Kernel alle weiteren 64KB-
//Teile verarbeitet. Der Unterschied der zwei Kernels ist, dass im
//dcs_gpu_kernel_5() zusätzlich das Zwischenergebnis aus dem Global Memory geladen
//wird, um den Ergebniswert des aktuellen 64 KB-Teils des Atom-Arrays aufaddieren
//zu können.
__global__ void dcs_gpu_kernel_5(float* energygrid,dim3 grid,float gridspacing,
int zindex,int numatoms)
{
    int xindex=blockIdx.x*blockDim.x*
        Anzahl_Gridwerte_in_XRichtung_Pro_Thread+threadIdx.x*
        Anzahl_Gridwerte_in_XRichtung_Pro_Thread;
    int yindex=blockIdx.y*blockDim.y+threadIdx.y;
    int outaddr=zindex*grid.x*grid.y+yindex*grid.x+xindex;

    int i;
    float curenergy[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        curenergy[i]=energygrid[outaddr+i];

    float coorx[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        coorx[i]=gridspacing*(xindex+i);

    float coory=gridspacing*yindex;
    float energyval[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        energyval[i]=0.0f;

    int atomid;
    for(atomid=0;atomid<numatoms;atomid++){
        float dy=coory-atominfo[atomid].y;
        float dysqpdzsq=(dy*dy)+atominfo[atomid].z;

        float x=atominfo[atomid].x;

        float dx[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
        for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
            dx[i]=coorx[i]-x;

        float charge=atominfo[atomid].w;
        for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
            energyval[i]+=charge*sqrtf(dx[i]*dx[i]+dysqpdzsq);
    }
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        energygrid[outaddr+i]=curenergy[i]+energyval[i];
}
```

Im Code 79 befindet sich der komplette Kernel-Code für die Optimierung „Memory Coalescing“.

Code 79: Mit Memory Coalescing [1] (adaptiert)

```
__global__ void dcs_gpu_kernel_6_Erster_Aufgerufener_Kernel(float* energygrid,
dim3 grid,float gridspacing,int zindex,int numatoms)
{
    int xindex=blockIdx.x*blockDim.x*Anzahl_Gridwerte_in_XRichtung_Pro_Thread+
        threadIdx.x;
    int yindex=blockIdx.y*blockDim.y+threadIdx.y;
    int outaddr=zindex*grid.x*grid.y+yindex*grid.x+xindex;

    float coorx=gridspacing*xindex;
    float gridspacing_coalesce=gridspacing*blockDim.x;
```

Fortsetzung von Code 79 auf der nächsten Seite

Fortsetzung von Code 79:

```

float coory=gridspacing*yindex;

float energyval[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
int i;
for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    energyval[i]=0.0f;

int atomid;
for(atomid=0;atomid<numatoms;atomid++){
    float dy=coory-atominfo[atomid].y;
    float dysqpdzsq=(dy*dy)+atominfo[atomid].z;

    float dx[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
    dx[0]=coorx-atominfo[atomid].x;
    for(i=1;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        dx[i]=dx[i-1]+gridspacing_coalesce;

    float charge=atominfo[atomid].w;
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        energyval[i]+=charge*rsqrtf(dx[i]*dx[i]+dysqpdzsq);
}
for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
    energygrid[outaddr+i*blockDim.x]=energyval[i];
}

//Dieser Kernel wird nach dem „dcs_gpu_kernel_6_Erster_Aufgerufener_Kernel()“-
//Kernel aufgerufen, wenn das Atom-Array größer als 64 KB ist und somit nicht
//komplett in den Constant Memory passt. Mit dem
//„dcs_gpu_kernel_6_Erster_Aufgerufener_Kernel()“-Kernel wird der erste 64 KB-
//Teil des Atom-Arrays und mit dem dcs_gpu_kernel_6()-Kernel alle weiteren 64KB-
//Teile verarbeitet. Der Unterschied der zwei Kernels ist, dass im
//dcs_gpu_kernel_6() zusätzlich das Zwischenergebnis aus dem Global Memory geladen
//wird, um den Ergebniswert des aktuellen 64 KB-Teils des Atom-Arrays aufaddieren
//zu können.
__global__ void dcs_gpu_kernel_6(float* energygrid,dim3 grid,float gridspacing,
int zindex,int numatoms)
{
    int xindex=blockIdx.x*blockDim.x*Anzahl_Gridwerte_in_XRichtung_Pro_Thread+
        threadIdx.x;
    int yindex=blockIdx.y*blockDim.y+threadIdx.y;
    int outaddr=zindex*grid.x*grid.y+yindex*grid.x+xindex;

    float coorx=gridspacing*xindex;
    float gridspacing_coalesce=gridspacing*blockDim.x;
    float coory=gridspacing*yindex;
    float energyval[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
    int i;
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        energyval[i]=0.0f;

    int atomid;
    for(atomid=0;atomid<numatoms;atomid++){
        float dy=coory-atominfo[atomid].y;
        float dysqpdzsq=(dy*dy)+atominfo[atomid].z;

        float dx[Anzahl_Gridwerte_in_XRichtung_Pro_Thread];
        dx[0]=coorx-atominfo[atomid].x;
        for(i=1;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
            dx[i]=dx[i-1]+gridspacing_coalesce;

        float charge=atominfo[atomid].w;
        for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
            energyval[i]+=charge*rsqrtf(dx[i]*dx[i]+dysqpdzsq);
    }
    for(i=0;i<Anzahl_Gridwerte_in_XRichtung_Pro_Thread;i++)
        energygrid[outaddr+i*blockDim.x]+=energyval[i];
}

```

Anhang B Abstract

A research focus in scientific computing deals with the parallelisation of algorithms for GPUs because the theoretical maximum performance of GPUs is many times higher than of CPUs. The master thesis starts with the Nvidia Fermi GPU architecture and the changes compared to the predecessor architecture GT200. In the next step the GPU programming languages CUDA and OpenCL are explained and the differences in programming are compared. The aim of the master thesis is the optimization of two life science algorithms (Needleman-Wunsch/Smith-Waterman, Direct Coulomb Summation) for the execution on graphics cards (Nvidia-Fermi, Nvidia-GT200) by using the programming languages CUDA and OpenCL. The DCS algorithm is much more suitable for graphics cards than the NW/SW algorithm because the degree of parallelisation is significantly higher. In this master thesis, it is determined how effective the optimization techniques are on different GPU architectures (Fermi, GT200) and on different GPU programming languages (CUDA, OpenCL). In doing so, it is checked whether the new cache hierarchy for the global memory in the Fermi architecture replaces optimization techniques (memory coalescing, use of the small on chip memories) and therefore makes an easier transfer from CPU code to GPU code without losing performance possible. Another important issue of the master thesis is the fair comparison between GPU program and CPU program when answering the question whether the two algorithms can reach a speedup on the GPU in comparison with the CPU. For a fair comparison all cores on the CPU are used with the programming language OpenMP.

Anhang C Abstract in Deutsch / Kurzfassung

Ein Forschungsschwerpunkt in Scientific Computing beschäftigt sich mit der Parallelisierung von Algorithmen für GPUs, da die theoretische maximale Leistung von GPUs ein Vielfaches größer ist als von CPUs. In der Masterarbeit wird zunächst auf die Nvidia-Fermi-GPU-Architektur und auf die Änderungen gegenüber der Vorgängerarchitektur GT200 näher eingegangen. Im nächsten Schritt werden die GPU-Programmiersprachen CUDA und OpenCL erklärt und die Unterschiede in der Programmierung gegenübergestellt. Das Ziel der Masterarbeit ist die Optimierung von zwei Life-Science-Algorithmen (Needleman-Wunsch/Smith-Waterman und Direct Coulomb Summation) für die Ausführung auf Grafikkarten (Nvidia-Fermi und Nvidia-GT200) bei Verwendung der Programmiersprachen CUDA und OpenCL. Der DCS-Algorithmus ist für Grafikkarten deutlich besser als der NW/SW-Algorithmus geeignet, da der Parallelisierungsgrad wesentlich größer ist. In der Masterarbeit wird ermittelt, wie wirksam einzelne Optimierungstechniken bei unterschiedlichen GPU-Architekturen (Fermi und GT200) und bei unterschiedlichen GPU-Programmiersprachen (CUDA und OpenCL) sind. Dabei wird überprüft, ob Optimierungstechniken, wie Memory Coalescing und Verwendung der kleinen on-chip-Speicher, durch die mit der Fermi-Architektur eingeführte Cache-Hierarchie über den Global Memory nicht mehr notwendig sind und dadurch eine einfachere Überführung von CPU-Code in GPU-Code ohne Einbußen bei der Performance möglich ist. Ebenfalls ein wichtiges Thema der Masterarbeit ist der faire Vergleich des GPU-Programms mit dem CPU-Programm beim Beantworten der Frage, ob bei beiden Algorithmen auf der GPU ein Speedup gegenüber der CPU erreicht werden kann. Für einen fairen Vergleich werden unter anderem bei der CPU mit der Programmiersprache OpenMP alle Kerne verwendet. In der Masterarbeit wird auch auf die limitierenden Faktoren eingegangen, die dazu führen, dass der Algorithmus nicht die Peak-Leistung der GPU erreicht.

Anhang D Deutsche Zusammenfassung

In der Masterarbeit wurde auf die Nvidia-Fermi-GPU-Architektur und auf die Änderungen gegenüber der Vorgängerarchitektur GT200 näher eingegangen. Es wurden die GPU-Programmiersprachen CUDA und OpenCL erklärt und die Unterschiede bei der Programmierung gegenübergestellt.

In der Masterarbeit wurden zwei Life-Science-Algorithmen (Needleman-Wunsch/Smith-Waterman und Direct Coulomb Summation) für die Ausführung auf Grafikkarten (Nvidia-Fermi und Nvidia-GT200) bei Verwendung der Programmiersprachen CUDA und OpenCL optimiert. Während der DCS-Algorithmus aufgrund der deutlich höheren Parallelität bereits ohne Optimierungen auf der GPU schneller als auf der CPU ausgeführt werden konnte, mussten beim NW-Algorithmus Optimierungen angewendet werden, um dieses Ziel zu erreichen. In der Masterarbeit wurde ermittelt, wie wirksam einzelne Optimierungstechniken bei unterschiedlichen GPU-Architekturen (Fermi und GT200) und bei unterschiedlichen GPU-Programmiersprachen (CUDA und OpenCL) sind.

Bei der Frage, ob bei beiden Algorithmen bei der Ausführung auf der GPU ein Speedup gegenüber der Ausführung auf der CPU erreicht werden kann, wurde ein fairer Vergleich durchgeführt. Dazu wurden auch Optimierungen am CPU-Programm vorgenommen. Es wurde auf die bestmögliche Kompilierung geachtet (icc-Compiler, -O3-Compilerflag und -xSSE3-Compilerflag) und es wurden mit der Programmiersprache OpenMP alle Kerne der CPU verwendet. Außerdem wurde für den Vergleich die Gesamtlaufzeit und nicht die Kernellaufzeit verwendet. Die Gesamtlaufzeit inkludiert neben der Kernellaufzeit auch die Laufzeit für das Anlegen und Freigeben der Arrays im CPU- und GPU-Arbeitsspeicher und die Laufzeit für die Übertragungen zwischen CPU- und GPU-Arbeitsspeicher. Da beim NW-Algorithmus eine große Datenmenge über den PCI-Bus übertragen wird, ist die Kernellaufzeit deutlich kleiner als die Gesamtlaufzeit. Damit spielt bei Optimierungen beim NW-Algorithmus nicht nur die Kernellaufzeit eine Rolle, sondern auch die Übertragungszeit zwischen CPU- und GPU-Arbeitsspeicher.

Es wurde untersucht, ob die mit Fermi eingeführte Cache-Hierarchie über den Global Memory es möglich macht, mit sehr wenig Aufwand ein CPU-Programm in ein GPU-Programm zu transformieren und dennoch keine Einbußen bei der Performance zu haben. Das Ergebnis ist, dass Optimierungstechniken, wie Memory Coalescing und Verwendung der kleinen on-chip-Speicher (Constant Memory und Shared Memory), nach wie vor relevant sind, jedoch fällt der Performancegewinn durch diese Optimierungstechniken mit Fermi deutlich niedriger aus.

Da sich die Anzahl der Kerne bei der Fermi-Architektur gegenüber der GT200-Architektur verdoppelt hat, die Taktfrequenz in etwa gleich geblieben ist und ein Global-Memory-Cache eingeführt wurde, würde man mindestens eine Halbierung der Laufzeit erwarten. Während bei dem NW-Algorithmus bei Betrachtung der Kernellaufzeit dieses Ziel erreicht wurde, konnte beim DCS-Algorithmus dieses Ziel nicht erreicht werden. Beim NW-Algorithmus konnte durch die Vergrößerung des Shared Memory bei Fermi (48 KB pro SM) gegenüber GT200 (16 KB pro SM) die Blockgröße deutlich vergrößert werden. Im Gegensatz dazu konnten beim DCS-Algorithmus keine Verbesserungen am Code für eine niedrigere Laufzeit speziell für die Fermi-Architektur gefunden werden.

In der Masterarbeit wurde festgestellt, dass es einen Laufzeitunterschied zwischen CUDA- und OpenCL-Programm gibt. Größere Laufzeitunterschiede wurden bei Constant Memory und bei der Übertragung von Daten zwischen einem Pageable-Host-Memory-Array vom CPU-Arbeitsspeicher und einem Array im GPU-Arbeitsspeicher festgestellt. Das zweite Problem trifft nur auf die Geforce-GPUs zu. Da das CUDA-Programm fast immer schneller als das OpenCL-Programm war, ist bei Nvidia-GPUs noch immer CUDA zu bevorzugen. Der große Vorteil eines OpenCL-Programms ist jedoch, dass man es auch auf AMD-GPUs, Intel-CPUs und AMD-CPU-GPUs ausführen kann.

Anhang E Lebenslauf

David Dilch, BSc

Ausbildung

Masterstudium: seit 01.10.2009

Universität Wien

Scientific Computing UG2002 (A 066 940)

Ausprägungsfach Pharmakoinformatik

Bachelorstudium: 01.10.2006 - 30.06.2009

Universität Wien

Informatik UG2002 (A 033 521)

Ausprägungsfach Scientific Computing

Reife- und Diplomprüfungszeugnis: 13.06.2006

HTL Rennweg

Höhere Lehranstalt für Informationstechnologie

Ausbildungsschwerpunkt Internet und Medientechnik