TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

Bachelor-Arbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Simulation stark gekoppelter Plasmen mit OpenCL auf Grafikprozessoren

Matthias Hille
(Geboren am 20. November 1989 in Karl-Marx-Stadt)

Betreuer: Dipl.-Ing. Guido Juckeland, Dr. Michael Bussmann

Dresden, 21.11.2012

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Bachelor-Arbeit zum Thema:

Simulation stark gekoppelter Plasmen mit OpenCL auf Grafikprozessoren

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.
Dresden, den 21.11.2012

Matthias Hille

Kurzfassung

Ein stark gekoppeltes Plasma ist ein System von Ionen, welche in einer elektromagnetischen Falle gefangen sind. Die darin befindlichen Ionen beeinflussen sich gegenseitig. Die dabei entstehenden Kräfte werden durch die Lösung des N-Körper-Problems ermittelt. Die Teilchen des Systems werden mit einer externen Kühlkraft gekühlt, sodass sie im Laufe der Simulationen einen Ruhezustand erreichen. Die Simulation berechnet das Verhalten der Ionen innerhalb des Systems und hat zum Ziel, die Positionen der Teilchen zu ermitteln, welche sie während des Abkühlvorgangs in Abhängigkeit der wirkenden Kräfte einnehmen.

Die Berechnung dieser Kräfte ist sehr zeitaufwendig, da das Problem eine quadratische Komplexität besitzt. Aufgrund dieser Rechenintensität werden Grafikprozessoren als Beschleuniger eingesetzt. Es wurde ein Programm geschrieben, welches das Problem auf mehrere GPUs innerhalb eines Rechenknotens parallelisieren und mit Hilfe eines austauschbaren Integrators lösen kann. Dafür wurden unterschiedliche Parallelisierungsalgorithmen untersucht und bezüglich ihrer Leistung bewertet.

Zudem wurde ein Vergleich zwischen der Ausführung des Programms auf GPUs und Multi-Core-CPUs durchgeführt. Dabei zeigt sich, dass die massiv-parallel ausgelegte Architektur der GPUs sehr gut für die Berechnung geeignet ist und effizientere Ergebnisse erzielt als die Ausführung auf CPUs.

Abstract

A strongly coupled plasma is a system consisting of ions caught in an electromagnetic trap. Those ions influence each other. The forces, which are emerging thereby are determined by solving the N-body problem. The particles inside the system are cooled down by an extern force, so they reach an idle state throughout the simulation. The simulation calculates the behavior of the ions to determine the positions of the particles they occupy during the cool-down phase. These positions depend on the acting forces.

The calculation of these force is very time consuming, because the algorithm is of subquadratic time. Because of these intense calculations graphics processing units are used as accelerator. A program has been written, which parallelizes the problem across several GPUs in one compute node and solves it with the assistance of an exchangeable integrator. Therefore different parallelization algorithms have examined and evaluated regarding there performance.

Moreover the program has been compared between executing on GPUs and multi-core processors. It appeared, that the massively parallel architecture of the GPUs fits best for the calculation and obtains more efficient results then CPUs.

Inhaltsverzeichnis

Symbolverzeichnis

CPU	Central Processing Unit – Hauptprozessor
GPU	Graphics Processing Unit – Grafikprozessor
ID	Identifikationsnummer
LSU	Load/Store Unit
PE	Processing Element – Verarbeitungseinheit
SFU	Special Function Unit
SM	Streaming Multiprocessor – Gruppe von Rechenkernen auf einer NVIDIA GPU
SMP	Symmetrisches Multiprozessorsystem
WAW-Hazard	Write-After-Write Hazard

1 Einleitung und Motivation

Simulationsrechnungen in der Informatik haben stets das Ziel, Modelle für naturwissenschaftliche Zusammenhänge zu verifizieren und zu verbessern sowie das Verhalten von Experimenten vorherzusagen. Für N-Körper-Simulationen gibt es daher besonders viele Anwendungsgebiete. Eines der ältesten ist die Astrophysik, in der zu Beginn das Verhalten von Sternhaufen untersucht wurde, danach von Galaxien und schließlich auch das Zusammenspiel mehrerer Galaxien, zum Beispiel wenn diese kollidieren. Auch in der Biologie finden diese Rechnungen ihre Anwendung, unter anderem wenn es darum geht, die Faltung von Proteinstrukturen zu untersuchen, beispielsweise für die Synthese neuer Medikamente. Nicht zuletzt profitiert aber auch die Teilchenphysik von den N-Körper-Simulationen, im Besonderen auf dem Gebiet der Molekulardynamik. Eine Problemstellung aus diesem Feld ist der praktische Hintergrund für diese Arbeit.
N-Körper-Simulationen wurden erstmals in den 1960er Jahren mittels numerischer Integration berechnet. Damals waren die Rechenkapazitäten noch auf einem viel niedrigeren Niveau als heute, deshalb galt es zu dieser Zeit als besonders ambitioniert, ein solches Problem anzugehen ([Aar03], S. 1) (vgl. [Hoe60], S. 184-214). Simulationen dieser Generation benötigten für die Berechnung von Systemen mit weniger als zehntausend Teilchen mehrere Wochen. Im Vergleich dazu werden heutzutage im selben Zeitraum mehrere Milliarden Teilchen simuliert, wie es in der Millenium Simulation am Max-Planck-Institut in Garching bereits 2005 durchgeführt wurde. Dies ist auf den technologischen Fortschritt der Hardwarekomponenten zurückzuführen sowie auf verbesserte Algorithmen zur Berechnung von N-Körper-Problemen. So gibt es nicht mehr nur die direkten N-Körper-Simulationen, sondern auch andere Methoden wie die Fast Multipole Methode oder adaptive Zeitschritte und eine Verringerung des Rechenaufwands mit Hilfe von Nachbarschemata. Des Weiteren hat es in den letzten Jahren eine enorme Entwicklung in Richtung immer stärker parallelisierter Architekturen gegeben. Dies wirkt sich ebenfalls positiv auf die Lösung von N-Körper-Problemen aus, da diese einen hohen Parallelitätsgrad besitzen.
Das in dieser Arbeit entstandene Programm `SCPonGPUcl` simuliert mit Hilfe von Grafikprozessoren ein stark gekoppeltes Plasma. Ein solches Plasma ist eine Wolke von Ionen, welche in einer Paul-Falle oder Penning-Falle gehalten werden. Zusätzlich werden sie durch eine Laserkühlkraft abgekühlt, was schließlich dazu führt, dass sie fast völlig erkalten und sich in Abhängigkeit der zwischen den Teilchen wirkenden Coulomb-Kräfte anordnen. Ziel der Simulation ist es, diese Anordnung zu bestimmen. Anwendungsgebiete für diese Simulation sind präzise Massenbestimmungen, die Verbesserung von Kollisionsexperimenten und die Forschung für Quantencomputer.
Als Beschleuniger dieser Rechnung wurden GPUs gewählt, da sie sich in letzter Zeit im High Performance Computing in zunehmendem Maße als kosteneffiziente Hardwarebeschleuniger erwiesen haben. Ein sehr spezieller Hardwarebeschleuniger ist der GRAPE-8, welcher jedoch nicht sehr verbreitet ist.
In dieser Arbeit wird untersucht, wie sich unterschiedliche Parallelisierungsstrategien für die GPUs eines Rechenknotens umsetzen lassen und welche Effizienz diese Strategien im Vergleich zur Nutzung von nur einer GPU besitzen.

2 Algorithmen

Für die Lösung von N-Körper-Problemen gibt es verschiedene Ansätze. Deshalb ist es entscheidend zu bestimmen, von welcher Natur das zu analysierende Problem ist. So wird bei Simulationen von weiträumigen Problemen von einem sogenannten cut-off-Radius gesprochen. Dieser beschreibt, über welche Strecke ein Teilchen mit anderen korreliert. Dadurch lässt sich der Berechnungsaufwand verringern, indem nicht alle Partikel des Systems für die Kraftberechnung herangezogen werden, sondern nur die Teilchen, welche sich innerhalb der Grenzen dieses cut-off-Radius' befinden.
Da die Teilchen des stark gekoppelten Plasmas, welches in dieser Arbeit simuliert werden soll, jedoch sehr eng beieinander liegen, beeinflussen sich alle Teilchen gegenseitig, weshalb jede Interaktion berücksichtigt werden muss.

2.1 Lösen des N-Körper-Problems

Das N-Körper-Problem beschreibt das Verhalten mehrerer Körper zueinander, welche sich nach den Newtonschen Gesetzen innerhalb eines Systems beeinflussen. Im Speziellen werden hierbei die Positionen und Geschwindigkeiten der Körper ermittelt. Dabei wird davon ausgegangen, dass die Körper Punktmassen sind und die Positionen sowie Geschwindigkeiten dieser zu einem initialen Zeitpunkt t_0 bekannt sind.
Die Lösung des N-Körper-Problems für N > 2 mittels analytischer Methoden war lange Zeit unbekannt. Erst im Jahr 1912 veröffentlichte Karl Sundman einen Artikel, in dem er das N-Körper-Problem für N = 3 mit einer Potenzreihe löste. (vgl. [Sun12], S. 105-179) Im Jahr 1991 gelang es dann dem chinesischen Studenten Quidong (Don) Wang eine Potenzreihe zur Lösung des allgemeinen N-Körper-Problems zu entwickeln. (vgl. [Wan91], S. 73-88)
Das Problem dieser beiden Reihen ist jedoch, dass sie sehr langsam konvergieren. So müsste man selbst für sehr kleine Zeitintervalle Millionen von Termen addieren, um die Bewegung der Teilchen zu berechnen. Da die Rundungsfehler bei derart vielen Termen zu groß wären, ist diese Variante in der Praxis nicht anwendbar. (vgl. [Mar85], S. 49) (vgl. [Dia96], S. 66-70) Aus diesem Grund nutzt man zur Lösung des N-Körper-Problems (für $N > 2$) immer noch numerische Methoden.
Für die Simulationsrechnungen werden Systeme mit N Partikeln zugrunde gelegt. Im Folgenden wird die Masse des i-ten Partikels mit m_i, seine Position mit $\overrightarrow{x_i}$ und seine Geschwindigkeit mit $\overrightarrow{v_i}$ bezeichnet. Die Positionen und Geschwindigkeiten der Partikel sind dabei zeitabhängig, wobei diese zum Zeitpunkt t_0 bekannt sind. Um die Bewegung der Teilchen zu bestimmen, nutzt man nun das folgende Differentialgleichungssystem.

$$\begin{aligned} \dot{\overrightarrow{x_i}} &= \overrightarrow{v_i} \\ \dot{\overrightarrow{v_i}} &= \frac{\overrightarrow{F_i}}{m_i} \end{aligned} \tag{2.1}$$

Dabei steht ein Punkt für den Operator d/dt und $\overrightarrow{F_i}$ ist die Kraft, welche auf das i-te Teilchen wirkt. Die Berechnung dieser Kraft birgt den eigentlichen Rechenaufwand, da sie sich aus der Summe aller Wechselwirkungen mit den übrigen Partikeln des Systems und eventuell aus Reibungs-, sowie externen Kräften zusammensetzt. Für die Gravitationskraft innerhalb eines Systems berechnet sich die Kraft beispielsweise wie folgt.

$$\overrightarrow{F_i} = -Gm_i \sum_{\substack{j=1 \\ j \neq i}}^{N} m_j \frac{\overrightarrow{x_i} - \overrightarrow{x_j}}{r_{ij}^3} \tag{2.2}$$

Hierbei steht G für die Gravitationskonstante und r_{ij} für den Abstand zwischen den Punkten i und j. ([Mar85], S. 50)
Um dieses Differentialgleichungssystem zu lösen, können verschiedene Integratoren genutzt werden. Unterschiedliche Integratoren können sich in Genauigkeit, Stabilität oder ihrem Rechenaufwand unterscheiden (vgl. [Pap06], S. 23-24).

2.2 Sequentielle Berechnungsverfahren

Um die Lösungsschritte für die Simulation eines N-Körper-Problems darzustellen, wird zunächst ein sequentieller Ansatz erläutert. Ziel eines jeden Berechnungsschrittes ist es, die Positionen und Geschwindigkeiten der Teilchen zum nächsten Zeitpunkt zu ermitteln. Dafür wird eine Schleife über alle Partikel des Systems benötigt. Jede dieser Schleifeniterationen, hier mit dem Laufindex i bezeichnet, behandelt ein Teilchen. Innerhalb dieser Schleife muss nun noch einmal eine Schleife über alle anderen Teilchen abgearbeitet werden. Diese erhält den Laufindex j. In der zweiten Schleife wird die Wechselwirkung zwischen dem i-ten Partikel und allen anderen Teilchen bestimmt. Anschließend können, je nach Simulationsgegenstand, noch andere Kräfte hinzuaddiert werden. Diese hängen jedoch nicht von den anderen Partikeln im System ab und lassen sich deshalb für jedes Teilchen in einem Schritt bestimmen.
Nachdem alle Kräfte ermittelt worden sind, werden die daraus resultierenden Geschwindigkeiten der Teilchen berechnet sowie in Abhängigkeit des verwendeten Zeitschritts die neuen Positionen.
Wenn N die Anzahl der Körper eines Systems ist, dann wird die Schleife mit dem Laufindex j genau $N * (N-1)$-mal durchlaufen. Dies bedeutet, dass der Algorithmus eine Komplexität von $O(N^2)$ besitzt.

2.3 Parallele Berechnungsverfahren

Für die numerische Lösung des N-Körper-Problems existieren verschiedene Ansätze zur Parallelisierung. Die erste Unterteilung wird hier anhand der Verteilung der Arbeitsschritte vorgenommen, wodurch sich drei unterschiedliche Varianten ergeben.
Das erste Verfahren ist die Aufteilung der Partikel auf die vorhandenen Verarbeitungseinheiten (Processing Element - PE). Zu Beginn der Simulation werden die Partikel des Systems statisch auf die PEs verteilt. Anschließend berechnet jedes PE die Bewegung seiner lokalen Partikel. Hierbei ist zu beachten, dass bei jedem Integrationsschritt die aktuellen Daten der Teilchen auf anderen PEs benötigt werden, was eine globale Kommunikation erforderlich macht.
Als zweite Möglichkeit sei die Aufteilung der Kraftberechnung genannt. Da man sich die Kraft zwischen den Körpern in einer Matrix der Größe $N \times N$ vorstellen kann, lässt sich diese zeilenweise zerteilen. Folglich errechnet jedes PE den ihm zugewiesenen Teil der Kraftmatrix. Danach erfolgt eine globale Reduktion, welche den Vektor der Kräfte aller Teilchen zusammenführt und an alle PEs kommuniziert. Um eine weitere Reduktion zu vermeiden, bestimmt man die Positionen und Geschwindigkeiten für jedes Teilchen auf jedem PE.
Die dritte Variante unterteilt das Simulationsgebiet in Teilgebiete (Domänen). Jede dieser Domänen wird einem PE zugewiesen. Dadurch erfolgt die Verteilung der Partikel auf die PEs nach deren räumlicher Anordnung. Dieses Modell eignet sich für Simulationen, in denen die Wechselwirkungen zwischen den Teilchen in solche mit kurzer Reichweite und solche mit langer Reichweite aufgeteilt werden, oder ein sogenannter cut-off-Radius definiert wird. Dieser Radius gibt an, bis zu welcher Distanz sich die Teilchen innerhalb des Systems beeinflussen. Nun berechnet jedes PE die Bewegung seiner lokalen Teilchen unter Zuhilfenahme von Informationen seiner Nachbar-PEs.
Die genauere Erläuterung der verschiedenen Verteilungen kann in [Sut02] nachgelesen werden. Da in der Simulation dieser Arbeit, aufgrund der starken Kopplung der Teilchen, keine Kräfte mit unterschiedlichen Reichweiten oder cut-off-Radien genutzt werden, findet die erste Variante, die Aufteilung der Partikel auf die Verarbeitungseinheiten, Anwendung. Hierbei gibt es nun unterschiedliche Szenarien der Datenhaltung, welche im Folgenden erklärt werden.

2.3.1 Redundante Datenhaltung

Die redundante Datenhaltung sieht vor, dass jedes PE vollständige Informationen über alle Partikel des Systems vorhält. In Kombination mit der statischen Verteilung der Teilchen auf alle PEs führt das zu dem in Abbildung 2.1 dargestellten Ablauf.

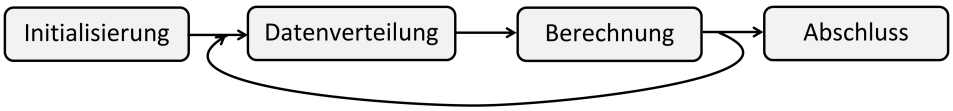

Abbildung 2.1: Programmablauf bei redundanter Datenhaltung

Dabei benötigt man zur Verteilung der Daten eine globale Kommunikation. Eine simple Implementierung benötigt dazu $N_p - 1$ Sende-/Empfangsoperationen, wobei N_p für die Anzahl der PEs steht. Nutzt man stattdessen ein baumartiges Kommunikationsmuster, lässt sich die Anzahl der notwendigen Kommunikationen auf $\log_2(N_p)$ begrenzen.
Im Berechnungsschritt bestimmt nun jedes PE die Geschwindigkeiten und Positionen seiner lokalen Teilchen. Der Vorteil hierbei ist, dass jedes PE Kenntnis über den Zustand des gesamten Systems hat und so die Bewegung der lokalen Teilchen ohne weitere Kommunikation berechnen kann. Allerdings müssen nach jedem Zeitschritt die Informationen über das gesamte System erneuert werden, was bei einer großen Teilchenanzahl zu intensiver Kommunikation großer Datenmengen führt.

2.3.2 Verteilte Datenhaltung – Systolischer Ring

Bei der verteilten Datenhaltung werden auf einem PE nur die Informationen der lokalen Teilchen permanent vorgehalten. In einem Puffer werden Daten eines weiteren PEs gespeichert. Anhand dieser wird die Wechselwirkung der lokalen Teilchen mit den Partikeln einer anderen Verarbeitungseinheit berechnet. Das heißt, in einer simplen Implementierung würde der Puffer pro Zeitschritt $(N_p - 1)$-mal neu befüllt werden, wenn N_p die Anzahl der PEs darstellt.
Bei diesem Verfahren werden jedoch alle Wechselwirkungen doppelt berechnet. Angenommen es existieren zwei Verarbeitungseinheiten, auf welche die Partikel in gleichen Teilen aufgeteilt werden, dann berechnet PE_1 die Wechselwirkung seiner lokalen Teilchen mit denen von PE_2. PE_2 berechnet ebenfalls die Kräfte zwischen seinen lokalen Teilchen und denen von PE_1. Diese Dopplung lässt sich mit einer systolischen Schleife unter Nutzung des 3. Newtonschen Gesetzes vermeiden.
Nach dem 3. Newtonschen Gesetz gilt „Aktion gleich Reaktion“. Wirkt nun eine Kraft auf ein Teilchen aus PE_1 von einem anderen Teilchen aus PE_2, ist demnach die Kraft, welche auf das Teilchen aus PE_2 wirkt von gleicher Größe, jedoch entgegengesetzt gerichtet, das heißt, mit anderem Vorzeichen versehen. Für eine systolische Schleife werden die Verarbeitungseinheiten nun in Form eines Ringes angeordnet, in welchem PE_i=PE_{i+P} gilt, wobei P die Anzahl der PEs ist. Nun werden die Positionen in die erwähnten Puffer der PEs geschrieben. Dabei sendet PE_j zu PE_i mit $i < j$. Nach der Evaluation der Wechselwirkungen werden die ermittelten Kräfte von PE_i wieder zu PE_j zurückgesendet. So wird das 3. Newtonsche Gesetz angewendet und damit eine doppelte Berechnung vermieden. Dies ist in Abbildung 2.2 für vier PEs veranschaulicht.

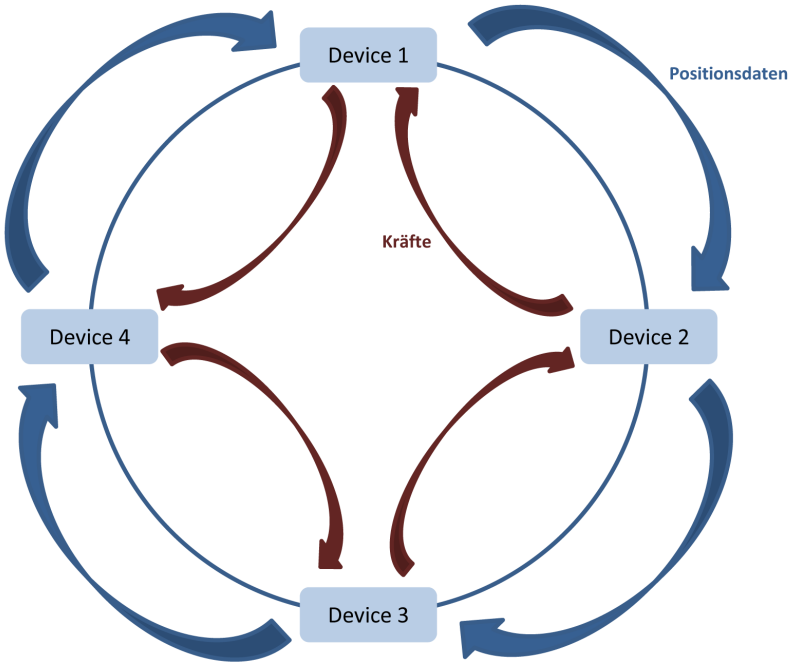

Abbildung 2.2: Schematische Darstellung des systolischen Rings für vier Geräte

Nach diesem Schema benötigt die Berechnung eines Zeitschritts $(N_P-1)/2$ Sende-/Empfangsoperationen zur Übermittlung der Positionen und $(N_P - 1)/2$ Sende-/Empfangsoperationen für den Transfer der berechneten Kräfte. ([Sut02], S. 239)

3 Implementierung

In diesem Kapitel werden zunächst einige Grundlagen zum OpenCL Standard erklärt sowie die Implementierung dessen auf GPUs und CPUs. Die Angaben bezüglich des OpenCL Standards stammen aus der OpenCL Spezifikation der Khronos Group [Khr10].
Anschließend wird der Aufbau des Programms `SCPonGPUcl` sowie die Umsetzung der Algorithmen beschrieben.

3.1 OpenCL

OpenCL (Open Compute Language) ist ein offener Industriestandard, welcher von der Khronos Group im Dezember 2008 in der Version 1.0 veröffentlicht wurde. Er wurde erschaffen, um die heute existierenden verschiedenen heterogenen Plattformen über eine einheitliche Schnittstelle nutzen zu können. OpenCL eröffnet die Möglichkeit, CPUs, GPUs oder auch Hardwarebeschleuniger auf mobilen Geräten als Recheneinheiten über diese gemeinsame Schnittstelle zu verwenden. Die Weiterentwicklung des Standards erfolgte im Juni 2010 mit der Veröffentlichung der Version 1.1, welches die derzeit am häufigsten unterstützte Version ist. Obwohl die OpenCL-Version 1.2 bereits im November 2011 bereitgestellt wurde, unterstützen bisher nicht alle Geräte diesen Standard.
Das OpenCL Framework beinhaltet eine an C99 angelehnte Programmiersprache, eine Reihe von API Funktionen sowie eine Bibliothek von nativen Routinen und ein Laufzeitsystem. Die Nutzung dieses Frameworks wird in den folgenden drei Abschnitten erläutert.

3.1.1 Plattform-Modell

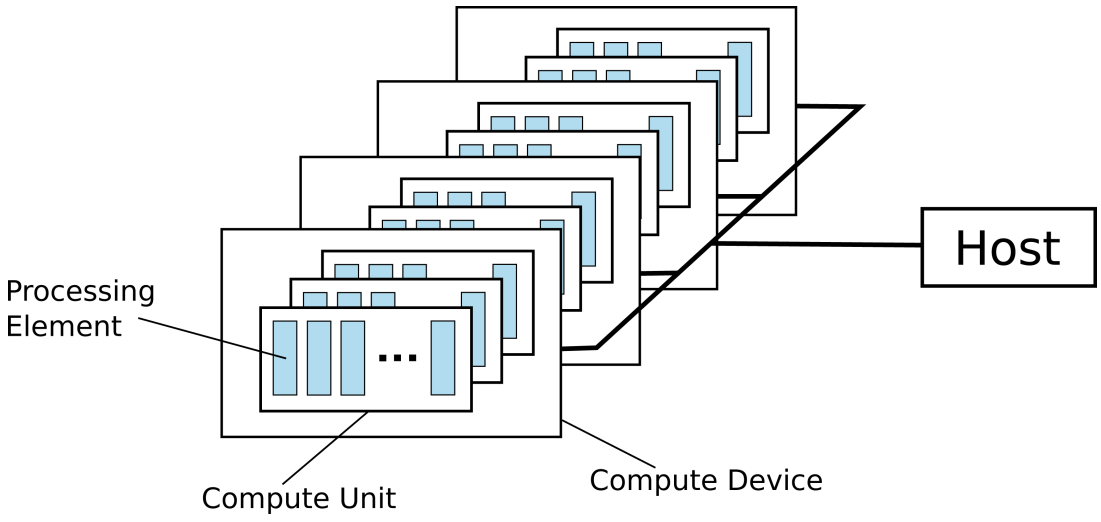

Abbildung 3.1: OpenCL Plattform-Modell

Das OpenCL Plattform-Modell, wie es in Abbildung 3.1 zu sehen ist, besteht aus einem Host, welcher mehrere OpenCL Geräte (Devices) verwaltet. Die Geräte bestehen wiederum aus einer oder mehreren Recheneinheiten (Compute Units). Innerhalb dieser befinden sich die Verarbeitungseinheiten (Processing Elements), welche die eigentliche Arbeit verrichten. Auf eine GPU abgebildet, stellt die CPU den Host dar und eine GPU ein Gerät. Die Recheneinheiten sind die Streaming Multiprozessoren (SM) innerhalb einer NVIDIA GPU, diese beinhalten die einzelnen Kerne einer GPU, welche als Verarbeitungseinheiten fungieren.

3.1.2 Ausführungsmodell

Ein OpenCL Programm besteht stets aus zwei Teilen, dem Hostprogramm und mindestens einem OpenCL Kernel, welcher auf dem OpenCL Gerät ausgeführt wird.
Zuerst muss das Hostprogramm einen Kontext erstellen, welcher folgende Ressourcen verwaltet:

- Devices - Eine Liste der Geräte, welche verwendet werden.
- Kernel - Die OpenCL Funktionen, welche auf den Devices ausgeführt werden sollen.
- Programmobjekte - Der Quellcode und die ausführbare Datei der Kernel.
- Speicherobjekte - Speicherobjekte, welche für den Host und die Devices sichtbar sind. Auf diesen Daten können die Kernel anschließend operieren.

Eine Besonderheit von OpenCL besteht darin, dass die Kernel erst zur Laufzeit übersetzt werden. Dies übernimmt der Treiber des jeweiligen OpenCL Geräts. Das Übersetzen muss vom Hostprogramm angestoßen werden, damit innerhalb des Kontexts ein Programmobjekt zur Verfügung steht.
Innerhalb des Kontexts existiert außerdem eine Befehlswarteschlange (Command Queue). In diese werden Befehle dreier unterschiedlicher Gruppen eingereiht. Dies sind die Ausführungsbefehle für die Kernel, Speicherbefehle und Synchronisationsanweisungen. Die Befehle innerhalb dieser Warteschlange werden, je nach Konfiguration der Warteschlange, nacheinander oder ohne Reihenfolge ausgeführt.

Der wichtigste Teil der Ausführung ist die Abarbeitung der Kernel. Hierfür wird ein Indexbereich angelegt, in welchem sich sogenannte Workitems befinden. Dieser NDRange genannte Indexbereich kann bis zu drei Dimensionen besitzen. Nun wird für jedes Workitem eine Instanz des Kernels ausgeführt. Alle Workitems durchlaufen somit dasselbe Programm. Der Ablauf beziehungsweise die Daten, mit welchen gearbeitet wird, lassen sich anhand des Index des Workitems beeinflussen. Jedes Workitem hat einen eindeutigen globalen Index. Weiterhin sind sie in Workgroups organisiert, sodass sie auch innerhalb ihrer Workgroup einen eindeutigen Index erhalten. Auf dem Gerät werden die Workitems einer Workgroup gleichzeitig auf einer Recheneinheit ausgeführt.

3.1.3 Speichermodell

In OpenCL existieren vier verschiedene Speicherbereiche, auf die innerhalb eines Kernels zugegriffen werden kann:

- *Global Memory* - Dieser stellt den größten Speicherbereich eines Geräts dar, den Hauptspeicher. Hierauf können alle Workitems lesend sowie schreibend zugreifen. Auf einer GPU ist dies der langsamste Speicherbereich.
- *Constant Memory* - Darin gespeicherte Daten bleiben während der Ausführung des OpenCL Kernels konstant. Auf GPUs ist dies ein extra Speicherbereich, welcher schneller als der Hauptspeicher ist.
- *Local Memory* - Diesen Speicherbereich teilen sich alle Workitems innerhalb einer Workgroup. Auf einer GPU ist dies ein on-Chip Speicher, welcher ähnliche Zugriffszeiten wie Register besitzt.
- *Private Memory* - Diesen Bereich besitzt und verwaltet jedes Workitem selbstständig. Für eine GPU gilt: werden nicht zu viele private Variablen angelegt, liegen diese in den Registern eines jeden Kerns. Im Falle eines Registerüberlaufs gibt es einen Speicherbereich im Global Memory, in dem die Variablen dann zwischengespeichert werden. Dies verlangsamt das Programm jedoch erheblich.

Auf einer CPU werden alle diese vier Speicherbereiche im Hauptspeicher abgebildet und Zugriffe nach den Möglichkeiten der jeweiligen CPU gecached. Dies ist besonders bei der Verwendung des Local Memory zu beachten, da dieser auf einer GPU oft dafür genutzt wird, häufiger benötigte Daten zu Beginn zu laden, damit alle Workitems innerhalb einer Workgroup schnellen Zugriff darauf haben. Auf einer CPU bringt dies nur Vorteile, wenn das Programm nicht zu datenintensiv ist und die anfangs geladenen Daten somit nicht aus dem Cache verdrängt werden.
Die Speicherbereiche unterscheiden sich auch hinsichtlich ihrer Konsistenz. Innerhalb eines Workitems herrscht Load/Store-Konsistenz. Für den Local Memory kann mittels einer Workgroupbarriere die Konsistenz innerhalb einer Workgroup sichergestellt werden, gleiches gilt für den Global Memory. Allerdings gibt es keine Möglichkeit, im Global Memory workgroupübergreifende Garantien zu geben.
Eine weitere Besonderheit des OpenCL Speichermodells ist die Notwendigkeit, Speicherobjekte anzulegen. Über diese Objekte erfolgt die Datenkommunikation zwischen dem Host und dem Gerät. Im Hostprogramm muss dafür gesorgt werden, dass die Eingangsdaten zuerst auf das Gerät übertragen werden und die Ergebnisse nach Abschluss der Berechnung zurück in den Hostspeicher kopiert werden.

3.1.4 Programmiermodell

OpenCL unterstützt zwei Programmiermodelle, zum einen ein taskparalleles und zum anderen ein datenparalleles Modell. Bei ersterem wird von einem Kernel nur eine Instanz ausgeführt. Parallelität muss durch Vektordatentypen oder das Ausführen mehrerer Tasks erreicht werden. Auf diesem Modell liegt jedoch nicht der Fokus von OpenCL. Dieser ist auf das datenparallele Modell gerichtet. Hierbei wird nach dem Single Program Multiple Data Prinzip vorgegangen. Wie bereits im Abschnitt 3.1.2 erläutert wurde, erhält jede Instanz des Kernels eine eindeutige ID. Anhand dieser kann nun jeder Instanz ihr eigener Datenbereich zugewiesen werden. [Khr10]

3.2 Programmaufbau

Das Programm `SCPonGPUcl` ist in C++ geschrieben und nutzt OpenCL für alle wichtigen Berechnungsaufgaben. Das Ziel der Programmstruktur ist es, dem Nutzer eine Schnittstelle zur Verfügung zu stellen, mittels derer er einen Integrator seiner Wahl verwenden kann. Dieses Prinzip ist in Abbildung 3.2 dargestellt.
Hierbei stellt die Klasse `bodySystemManager` die nötigen Methoden zur Verfügung, deshalb muss sich nicht um die Parallelisierung gekümmert werden. Für die Klasse `Integrator` erscheint das System als einzelne Instanz der Klasse `bodySystemManager`. Dieser sorgt dafür, dass die Arbeit auf die vorhandenen Geräte verteilt wird und sammelt die Ergebnisdaten, um sie dem Nutzer akkumuliert zur Verfügung zu stellen.

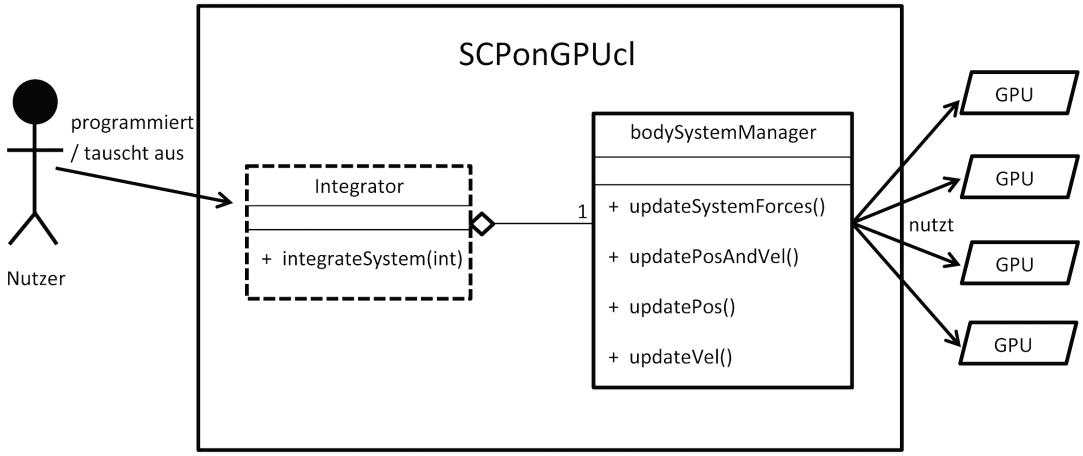

Abbildung 3.2: Sicht des Nutzers auf das Programm

Im Gegensatz zu einer anderen N-Körper-Simulation namens φGrape von K. Nitadori (welche statt OpenCL CUDA verwendet) [SBB$^+$11], nutzt das Programm einen CPU-Kern zur Verwaltung der OpenCL Geräte.
Ein weiteres Simulationsprogramm ist $NBODY6$ von S. J. Aarseth und K. Nitadori [NA12]. Dieses beschleunigt die Berechnung ebenfalls mit GPUs, verwendet allerdings ein anderes Berechnungsverfahren und zwar das Ahmad-Cohen-Nachbarschema. Mit diesem verringert sich die Kommunikation zwischen den GPUs auf Kosten der Genauigkeit der Rechnung. `SCPonGPUcl` dagegen bestimmt in jedem Zeitschritt tatsächlich alle Wechselwirkungen zwischen den Partikeln und nicht nur die der jeweils benachbarten, also naheliegenden Teilchen.

3.3 Implementierung der Algorithmen

3.3.1 Lösung auf einem Gerät

Wird das Programm auf einem einzelnen OpenCL Gerät ausgeführt, arbeitet es nach folgender Vorgehensweise.
Für jedes Teilchen wird ein Thread auf dem Gerät ausgeführt. Dieser lädt zu Beginn die Positionsdaten des Teilchens, welches ihm durch seine ID zugewiesen wird, in die Register. Die Threads werden, wie in Abschnitt 3.1.2 erläutert, in Workgroups gebündelt. Innerhalb einer solchen Workgroup wird ein Teil des Local Memory dafür genutzt, die Positionsdaten der übrigen Teilchen, zu welchen die Wechselwirkungen bestimmt werden sollen, zwischenzuspeichern. So werden sie allen Threads innerhalb der Workgroup schnell zugänglich gemacht. Die Größe des Speicherbereichs im Local Memory ist so ausgelegt, dass jeder Thread der Workgroup einen Positionsdatensatz eines weiteren Teilchens lädt. Nun iteriert jeder Thread über alle Teilchen aus dem Local Memory und berechnet die Wechselwirkungen mit dem ihm zugewiesenen Teilchen. Die resultierenden Kräfte werden sofort zu einem in den Registern liegenden Kraftvektor addiert. Sind die zwischengespeicherten Teilchen abgearbeitet und noch weitere zur Berechnung vorhanden, werden die nächsten Teilchen aus dem Global Memory in den Local Memory kopiert. So berechnet jeder Thread die Wechselwirkung zu jedem anderen Teilchen. Diese Berechnung ist, wie in Abbildung 3.3 gezeigt, in Blöcke unterteilt, sodass der Local Memory einer GPU als Zwischenspeicher für die Positionsdaten genutzt werden kann. Ein Auszug aus dem OpenCL Kernel befindet sich im Anhang im Abschnitt A.1.

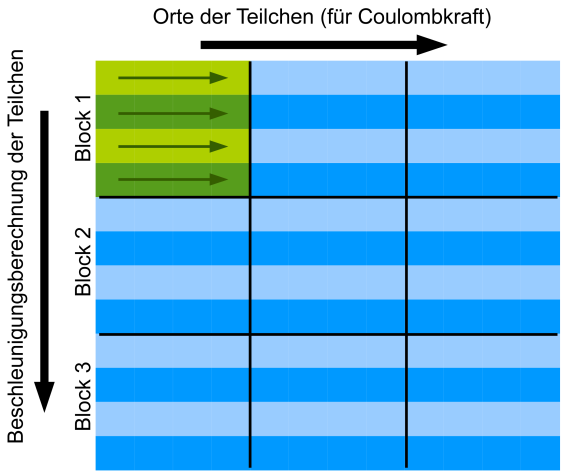

Abbildung 3.3: Berechnungsschema - Threads arbeiten sich blockweise durch die Daten

3.3.2 Lösung auf mehreren Geräten

Redundante Datenhaltung

Die Berechnung mit Hilfe des in Abschnitt 2.3.1 erklärten Vorgehens der redundanten Datenhaltung lässt sich einfach und effektiv implementieren. Dabei wird auf jedem Gerät ein Speicherobjekt für die Posi-

tionen und Geschwindigkeiten aller Teilchen des Systems angelegt. Jedes Gerät arbeitet nun den ihm zugewiesenen Teilbereich der Daten ab. Es werden die Kräfte, welche auf diese lokalen Teilchen wirken, ermittelt und anschließend deren Geschwindigkeiten und Positionen aktualisiert. Zu Beginn eines jeden weiteren Zeitschritts ist es deshalb erforderlich, die Positionsdaten der jeweiligen lokalen Teilchen eines Geräts an den Host zu übertragen. Von dieser zentralen Stelle werden sie nun an alle Geräte kommuniziert, damit die Kräfteberechnung für den nächsten Zeitschritt erneut auf der Grundlage des aktuellen und korrekten Zustands des Systems ausgeführt werden kann.

Systolischer Ring

Die Implementierung des systolischen Rings gestaltet sich komplexer. Hierbei werden zu Beginn die Positionsdaten der Geräte an den Host gesendet. Hier liegen nun alle Daten des kompletten Systems vor. Von dieser Stelle aus wird jedes Gerät mit dem entsprechenden Ausschnitt der Positionsdaten versorgt. Das entspricht der in Abschnitt 2.3.2 erklärten Kommunikation zwischen den PEs. Die so übermittelten Teilchenpositionen werden in einem dafür vorgesehenen Puffer abgelegt. Anhand dieser Daten werden dann die Wechselwirkungen der lokalen Teilchen mit denen aus dem Puffer berechnet. Da das Ergebnis der Rechnung nicht nur die Kräfte sein sollen, welche auf die lokalen Teilchen wirken, sondern auch die Kräfte für die Teilchen des anderen PEs, muss jede einzelne Wechselwirkung abgespeichert und zu zwei unterschiedlichen Kraftvektoren addiert werden. Bei der Berechnung der Wechselwirkung entsteht eine Kraftmatrix, welche in Abbildung 3.4 dargestellt ist.

P_i / P_j	P_5	P_6	P_7	P_8
P_1	F_{15}	F_{16}	F_{17}	F_{18}
P_2	F_{25}	F_{26}	F_{27}	F_{28}
P_3	F_{35}	F_{36}	F_{37}	F_{38}
P_4	F_{45}	F_{46}	F_{47}	F_{48}

Abbildung 3.4: Kraftmatrix

Die Summe aller Kräfte der Zeile i ist die Kraft, welche auf das lokale Teilchen i wirkt. Die Kraft, welche auf das externe Teilchen j wirkt, ergibt sich als Summe aller Kräfte aus der Spalte j. Nur diese über die Spalten aufsummierten Kräfte werden nach Abschluss der Berechnung auch an den Host zurückkommuniziert und gelangen dann auf das Gerät, von welchem die Positionsdaten für die externen Teilchen stammen.

Dieser Ansatz erfordert sehr viel Speicher, da die quadratische Komplexität des Problems auf die Speicheranforderung übertragen wird. Es wird nun davon ausgegangen, dass auf einer GPU ungefähr 2,5 GByte für die Kraftmatrix reserviert werden können. Die Kraft wird in einem `float4`-Vektor gespeichert und benötigt damit 16 Byte Speicherplatz. Daraus ergibt sich folgende Rechnung für die maximale

Anzahl an Teilchen, welche auf einem Gerät berechnet werden können.

$$N = \sqrt{\frac{2,5GByte * 1024^3}{16Byte}} = 12952 \tag{3.1}$$

Ungefähr 13 000 Teilchen pro Gerät sind allerdings deutlich zu wenig für die angestrebten Simulationsrechnungen.

Systolischer Ring mit Reduktion im Local Memory

Um dieses Speicherproblem zu umgehen, lässt sich der Local Memory eines Geräts nutzen. Dazu unterteilt man die Berechnung der Kraftmatrix in kleinere Blöcke, welche in den Local Memory passen. In Abbildung 3.5 ist dies schematisch dargestellt. Die dabei maximal mögliche Blockgröße für eine aktuelle Fermi-GPU beträgt 32. Dies bedeutet, dass auf einer GPU stets nur ein Block pro SM aktiv sein kann.

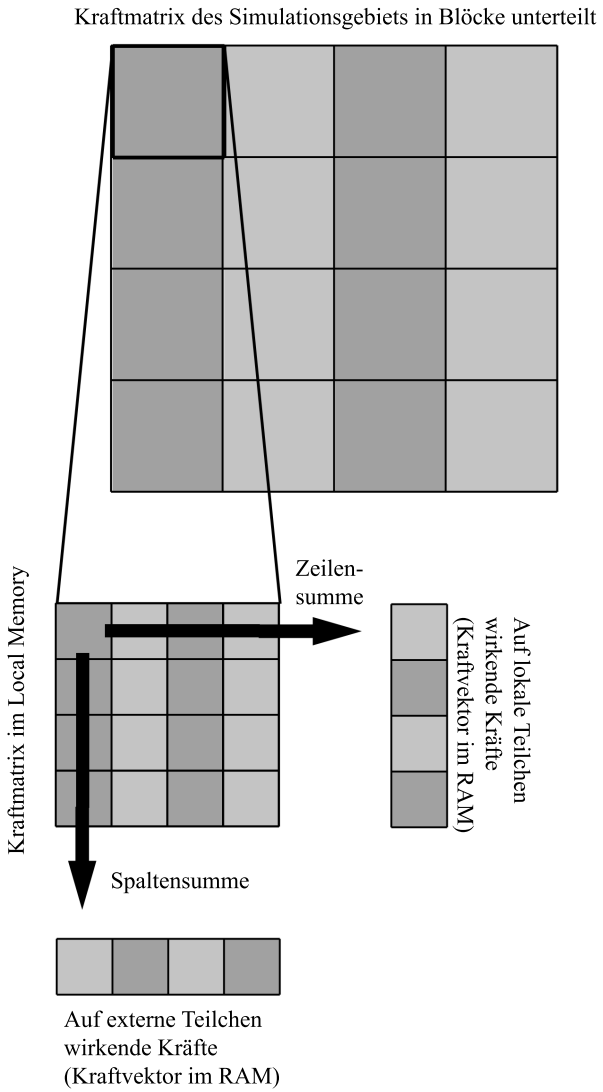

Abbildung 3.5: Schematische Darstellung der Unterteilung der Kraftmatrix für die Reduktion im Local Memory

Der Ablauf innerhalb eines Blocks gestaltet sich wie folgt. Jeder der 32 Threads innerhalb eines Blocks steht für ein lokales Teilchen, hält also die Positionsdaten dessen in den Registern. Nun lädt jeder Thread von einem externen Teilchen die Positionen in den Local Memory. Danach werden die Wechselwirkungen zwischen den lokalen und externen Teilchen berechnet, wobei jede berechnete Kraft zuerst zur Kraft auf das jeweils lokale Teilchen addiert wird und danach in die 32×32 große Kraftmatrix im Local Memory geschrieben wird. Diese Speicherzugriffe laufen wesentlich schneller ab als die erste Lösung, welche die Teilkräfte im Global Memory ablegt. Nach der Berechnung dieser Kräfte werden die Summen über die Spalten dieser Unterkraftmatrix gebildet, was in Abbildung 3.5 durch den senkrechten Pfeil veranschaulicht ist. Das Ergebnis ist ein Teil der Kraft, welche auf die externen Teilchen wirkt. Diese Zwischenergebnisse werden anschließend zu einem im Global Memory liegenden Kraftvektor hinzuaddiert.
Da nun auf einer GPU mehrere SMs existieren und damit auch mehrere Blöcke gleichzeitig ausgeführt werden, ist es möglich, dass dasselbe externe Teilchen zur gleichen Zeit in mehreren Blöcken benutzt wird. Das heißt, zwei Blöcke berechnen einen Teil der Kraft, die auf dieses externe Teilchen wirkt und versuchen anschließend das Ergebnis zum Kraftvektor im Global Memory hinzuzurechnen. Da OpenCL für den Zugriff auf den Global Memory keine Kohärenz gewährleistet, kann es passieren, dass hierbei einer der Zugriffe auf Grund eines WAW-Hazards verloren geht. Deshalb wird die Addition mit einer atomaren Funktion realisiert, was zu einer notwendigen Verlangsamung führt.
Mit der Reduktion im Local Memory benötigt man im Global Memory nur noch den Kraftvektor der Länge N, wobei N die Anzahl der simulierten Teilchen ist. Da jedes Teilchen die Eigenschaften Position, Geschwindigkeit und wirkende Kraft besitzt, werden nun pro Teilchen bei einfacher Genauigkeit 48 Byte im Global Memory belegt. Um die maximale Teilchenanzahl pro Gerät zu bestimmen, muss noch der Puffer für die Positionsdaten der externen Teilchen bedacht werden, welcher nochmals 16 Byte erfordert. Dies bedeutet, gemessen am Speicherplatz, der auf einer GPU verfügbar ist, dass circa 42 Millionen Teilchen auf einem Gerät untergebracht werden könnten. Bei einer solchen Teilchenanzahl ist jedoch die Rechenzeit auf Grund des quadratischen Aufwands zu hoch, um in vertretbarer Zeit Ergebnisse zu liefern.

3.4 Implementierung des Velocity-Verlet-Integrators

Für die Integration des Systems wurde der Velocity-Verlet-Integrator genutzt. Dieser arbeitet in drei Schritten.
Beim ersten Zeitschritt der Simulation ist, ausgehend von den initialen Positionen der Teilchen, eine vorherige Berechnung der Wechselwirkungen der Teilchen vonnöten. Jeder weitere Zeitschritt läuft wie im Folgenden beschrieben ab:

1. Ausgehend von den aktuell auf jedes Teilchen wirkenden Kräften berechnet man zunächst die neuen Positionen der Teilchen für den Zeitpunkt $t + \Delta t$. Im selben Schritt werden die Geschwindigkeiten der Teilchen bestimmt, allerdings für den Zeitpunkt $t + \frac{\Delta t}{2}$.
   (`updatePosAndVel()`)

2. Mit den neu bestimmten Positionen als Datenbasis werden nun die Wechselwirkungen zwischen den Teilchen bestimmt.
   (`updateSystemForces()`)

3. Im dritten und letzten Schritt werden die Geschwindigkeiten wiederum neu berechnet, diesmal für den Zeitpunkt $t + \Delta t$, das heißt, dass der Integrationsschritt wiederum $\frac{\Delta t}{2}$ beträgt.
   (`updateVel()`)

In jedem dieser Schritte wird die in Klammern aufgeführte Methode aufgerufen, welche von der Klasse `bodySystemManager` bereitgestellt wird. Dabei werden die mittels `updateSystemForces()` ermittelten Kräfte im Speicher der OpenCL Geräte abgelegt, damit diese bei späterer Geschwindigkeitsberechnung schnell zur Verfügung stehen.

4 Performanceanalyse

In diesem Kapitel werden die unterschiedlichen Parallelisierungsstrategien des Programms untersucht und miteinander verglichen. Die Laufzeitunterschiede werden analysiert und begründet. Durch Auswertung eines Vampir Traces des Programms wird außerdem der Verlauf des Programms sowie die Kommunikation zwischen Host und Gerät dargestellt.
Das Programm wurde mit unterschiedlichen Compilern getestet (GNU, Intel, PGI), es ergeben sich dadurch jedoch keine unterschiedlichen Laufzeiten. Die folgenden Tests wurden mit einer durch den GNU Compiler (Version 4.7.0) erstellten Version durchgeführt. Für die GPUs wurde der OpenCL Treiber, welcher mit der Cuda Version 4.2 mitgeliefert wird, verwendet. Für die Ausführung der Kernel auf CPUs wurde für die Intel CPUs der OpenCL 1.1 Treiber aus dem Intel SDK for OpenCL* Applications 2012 und für die AMD CPUs der OpenCL Treiber aus dem AMD APP SDK Version 2.7 verwendet.

4.1 Testhardware

Die Testläufe des Programms wurden auf zwei unterschiedlichen Rechenknoten durchgeführt. Die Konfiguration des ersten Knotens, auf welchem die Tests mit GPUs und CPUs gefahren wurden, ist der Tabelle 4.1 zu entnehmen. Der zweite Knoten ist in Tabelle 4.2 beschrieben und wurde nur für Tests auf der CPU genutzt.

Tabelle 4.1: Hardwarekonfiguration des GPU-Knotens

Modell CPU	XEON E5620
Anzahl CPUs	2
Taktfrequenz CPU	2.4 GHz
Arbeitsspeicher CPU	24 GB
GPU	NVIDIA Tesla S2050
Grafikchip	GF100
Arbeitsspeicher GPU	12 GB
Anzahl GPU Devices	4
GPU Typ	Fermi

Tabelle 4.2: Hardwarekonfiguration des CPU-Knotens

Modell CPU	AMD Opteraon 6276
Anzahl CPUs	4
Taktfrequenz CPU	2.3 GHz
Arbeitsspeicher CPU	256 GB

4.2 Analyse auf GPUs

Das Programm ist in der Lage, ein N-Körper-Problem mit Hilfe von vier verschiedenen Methoden zu lösen. Die Berechnungen auf dem Gerät werden dabei mit einfacher Genauigkeit ausgeführt. Die erste Lösungsmethode nutzt ein OpenCL Gerät und dient als Referenz für die Laufzeiten der anderen drei Methoden. Die nächsten zwei Verfahren sind sich sehr ähnlich, da sie beide den Algorithmus des systolischen Rings umsetzen. Der erste Ansatz zur Umsetzung des systolischen Rings verwendet, wie in Abschnitt 3.3.2 zuerst beschrieben, den Global Memory des Geräts um die Kräfte der externen Teilchen aufzusummieren. Diese Methode wird im Folgenden als Reduktion im Global Memory bezeichnet. Die zweite Variante wird entsprechend als Reduktion im Local Memory bezeichnet, da hier der Local Memory als Zwischenspeicher für die Teilkräfte verwendet wird.
Die vierte Herangehensweise nutzt die redundante Datenhaltung zur Parallelisierung über die OpenCL Geräte.
Sowohl die zwei Methoden, welche den systolischen Ring nutzen, als auch das Verfahren der redundanten Datenhaltung verwenden jeweils vier GPUs als OpenCL Geräte.

4.2.1 Architektur der verwendeten GPU

Um die Abbildung der Algorithmen auf die verwendete GPU verstehen zu können, wird zuerst auf die Funktionsweise der eingesetzten GPU eingegangen. Das in dieser Arbeit genutzte GPU Modul, eine NVIDIA Tesla S2050, beruht auf der Fermi Architektur, welche in Abbildung 4.1 schematisch dargestellt ist.

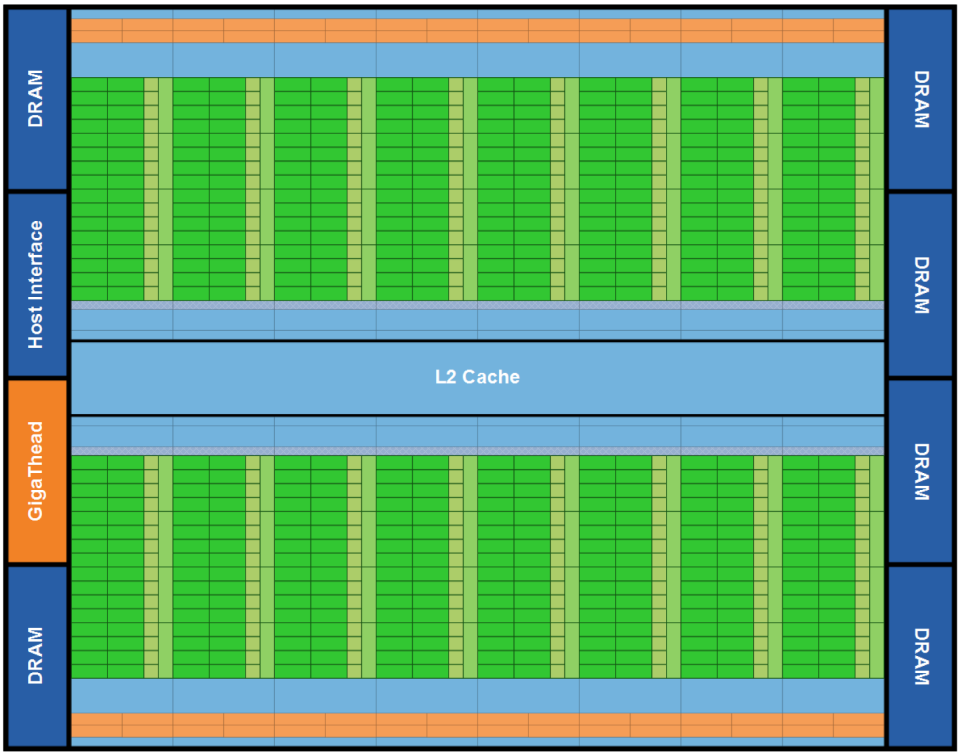

Abbildung 4.1: Schema der Fermi Architektur ([NVI09], S. 7)

Der für dieses GPU Modul verwendete Chip mit der Bezeichnung GF100 entspricht der Compute Capability 2.0. Die Compute Capability ist eine von NVIDIA eingeführte Klassifizierung der Geräte nach grundlegenden Leistungsmerkmalen, welche sich aus der Architektur ableiten.
Die in Abbildung 4.1 grün dargestellten Blöcke stellen dabei die SMs dar. Auf einem Fermi Chip können davon bis zu sechzehn Stück vorhanden sein. Die GPUs der verwendeten Tesla S2050 besitzen 14 SMs. Auf den genauen Aufbau dieser SMs wird anschließend noch eingegangen werden. Zwischen den einzelnen SMs befindet sich ein gemeinsam genutzter Level-2-Cache, in dem Zugriffe auf den Arbeitsspeicher zwischengespeichert werden. Dies ist eine bedeutende Neuerung der Fermi Architektur im Gegensatz zu ihrem Vorgänger, der GT200 Architektur. Der Arbeitsspeicher (DRAM) selbst ist am Rand des Schemas dargestellt, da sich dieser Speicher nicht auf dem Chip selbst befindet. Die SMs teilen sich zusätzlich zum DRAM zwei weitere Funktionseinheiten. Das ist zum einen das Host Interface, welches die GPU über PCI-Express mit dem Hostsystem verbindet und zum anderen der GigaThread^{TM} Thread Scheduler, der die Workgroups auf die einzelnen SMs aufteilt. ([NVI09], S. 7)
Eine genauere Darstellung der angesprochenen SMs ist in Abbildung 4.2 zu sehen.

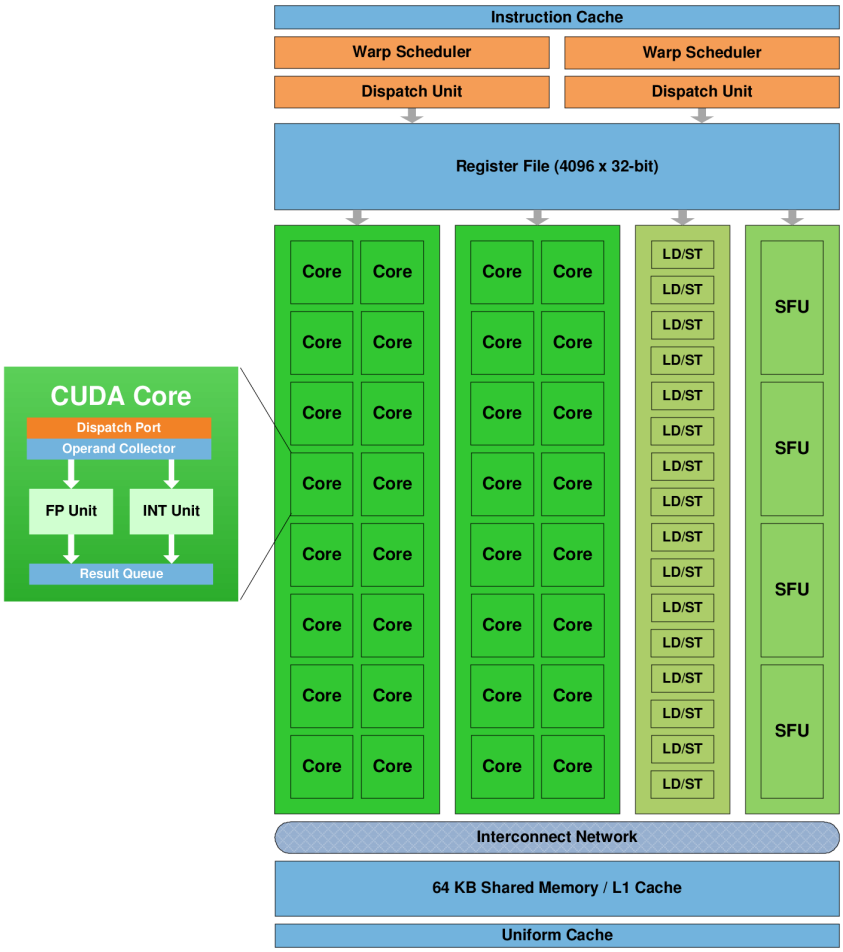

Abbildung 4.2: Detaildarstellung eines SMs ([NVI09], S. 8)

Innerhalb eines SMs befinden sich 32 CUDA Cores, die Recheneinheiten der GPU. Jeder dieser Kerne besitzt eine eigene Integer-Funktionseinheit und eine Floating Point-Einheit. Alle Kerne teilen sich

ein Register File, welches 4096 Einträge fassen kann. Folglich können auf jedem Kern zeitgleich und unabhängig voneinander 128 Register genutzt werden.
Im oberen Teil der Abbildung sind die orange eingefärbten Warp Scheduler zu sehen. Diese verteilen die Arbeit einer Workgroup, welche dem SM vom globalen GigaThread^{TM} Thread Scheduler zugeteilt wird, in sogenannte Warps. Ein Warp besteht aus 32 Threads, welche auf die einzelnen Kerne abgebildet werden. Eine Besonderheit dieser SMs besteht darin, dass pro SM zwei Warp Scheduler vorhanden sind. Somit können zwei Warps gleichzeitig eingeplant werden, um die Funktionseinheiten besser auszunutzen. Dabei wird ein Warp entweder auf sechzehn Kerne, sechzehn Load/Store-Units oder vier Special-Function-Units aufgeteilt. Das Prinzip, nach welchem die Funktionseinheiten einer GPU ausgelastet werden, basiert maßgeblich auf der Arbeitsweise der Warp Scheduler. Dabei ist es das Ziel, so viele Workgroups und damit so viele Warps wie möglich auf einem SM resident zu halten.
Dazu ist es zunächst notwendig zu erklären, was geschieht, wenn eine oder mehrere Workgroups einem SM zugeteilt werden. Zuerst werden die Ressourcen, welche die Workgroup benötigt, auf dem SM reserviert. Je weniger Register und Local Memory ein Kernel benötigt, desto mehr Workgroups können auf einem SM residieren, denn alle residenten Workgroups teilen sich die Ressourcen des SMs. Für die Compute Capability 2.0 beträgt die maximale Anzahl residenter Workgroups pro SM acht. Analog dazu gibt es auch Begrenzungen für maximal residente Warps und Threads pro SM. Konkret sind das in diesem Fall 48 Warps und 1536 Threads. Nachdem die Workgroups in Warps mit 32 Threads, also 32 Workitems, aufgeteilt wurden, können nun immer zwei Warps aktiv sein. Um beispielsweise die Latenz bei Speicheroperationen zu verdecken, kann zwischen den residenten Warps gewechselt werden, sodass in der Wartezeit andere Warps die in dieser Zeit freien Funktionseinheiten nutzen können. Ist ein Warp resident, bedeutet dies, dass die Werte der lokalen Variablen der Threads in den Registern verbleiben und der Local Memory der zugehörigen Workgroup ebenfalls allokiert bleibt. Das Wechseln eines Warps kostet keine Zeit, da zu Beginn eines jeden Taktes der Warp Scheduler aus den residenten Warps jene auswählt, deren Threads im nächsten Takt eine Operation ausführen können. (vgl. [NVI12], S. 62-63)
Zwei weitere Bestandteile eines SMs wurden bereits erwähnt: die Special Function Units (SFU) und die Load/Store Units (LSU). Die SFUs übernehmen die Berechnung transzendenter Funktionen, wozu Winkelfunktionen, die Quadratwurzel und das Ermitteln des Reziproken gehören.
Die Anzahl an LSUs pro SM beträgt sechzehn. Alle Speicherzugriffe werden standardmäßig im L1- und L2-Cache zwischengespeichert. Eine Cache-Zeile ist 128 Byte lang und repräsentiert ein an 128 Byte ausgerichtetes Speichersegment aus dem DRAM. Führt ein Warp eine Speicheroperation aus, so werden zuerst die Adressen aller Speicheranfragen ausgewertet und dann die entsprechenden Segmente aus dem Arbeitsspeicher geladen. Lädt also jeder Thread vier Byte, wird dies mit einem Zugriff auf den Arbeitsspeicher erledigt. Voraussetzung hierfür ist, dass die Daten in korrekter Ausrichtung im Hauptspeicher liegen und die Zugriffe innerhalb eines Warps dieser Ausrichtung entsprechen, also keine der 128 Byte-Grenzen überschreiten. Ein Beispiel für den Unterschied von korrekt und nicht korrekt ausgerichtetem Datenzugriff ist in Abbildung 4.3 zu sehen.

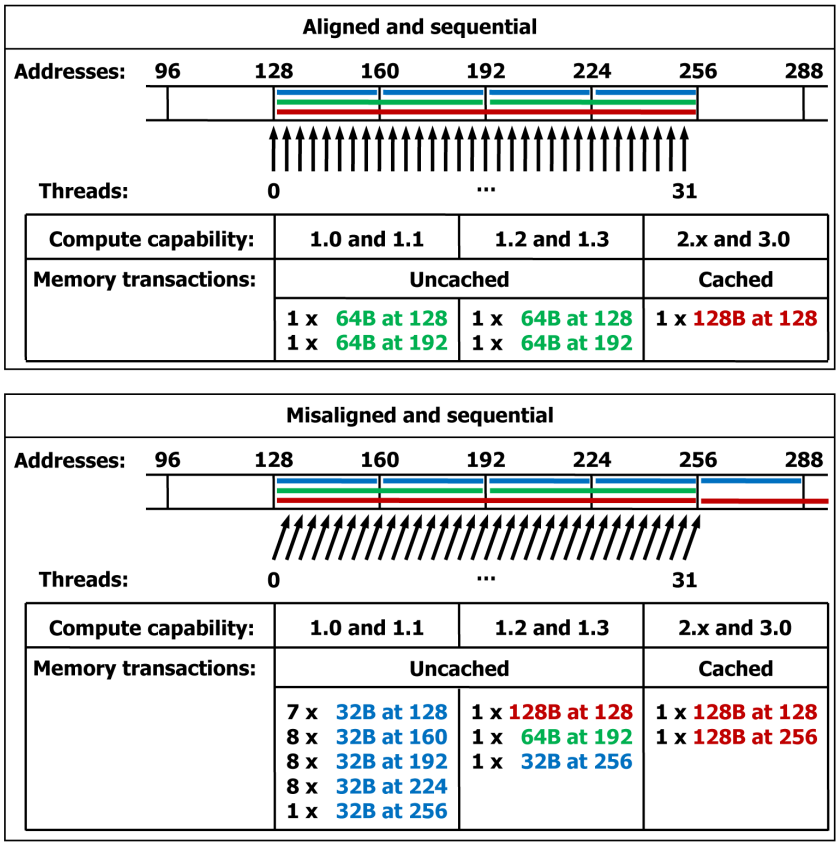

Abbildung 4.3: Beispiel für korrekt und unkorrekt ausgerichteten Datenzugriff ([NVI12], S. 151)

In der rechten Spalte ist zu erkennen, dass für die Compute Capability 2.0 im Falle ungünstiger Ausrichtung mehrere Zugriffe auf den Arbeitsspeicher nötig sind. In dem hier abgebildeten Fall werden fünfzig Prozent der übertragenen Daten nicht verwendet, was bedeutet, dass fünfzig Prozent der Speicherbandbreite verloren gehen. Die Zugriffe auf die einzelnen Teilbereiche einer Cache-Zeile sind hier sequentiell, das heißt, die Reihenfolge der Threads stimmt mit der Reihenfolge der Daten überein. Das muss jedoch nicht der Fall sein. Auch unregelmäßige Zugriffsmuster können bei korrekter Ausrichtung mit einer Speicheroperation bedient werden. (vgl. [NVI12], S. 146-147)

Das in Abbildung 4.3 gezeigte Beispiel geht davon aus, dass jeder Thread ein vier Byte großes Datenwort lädt. Wenn die angeforderten Datenwörter größer sind, müssen mehrere Speicheroperationen vollzogen werden. Beträgt die Größe acht Byte pro Datenwort, werden zwei Speicheranfragen vollzogen und im Falle von sechzehn Byte vier Anfragen. Das hier vorgestellte Programm `SCPonGPUcl` verwendet `float4` Vektoren um die Eigenschaften der Teilchen zu speichern, demzufolge beträgt die Größe eines Datenworts sechzehn Byte. Dies führt dazu, dass jeweils ein Viertel des Warps, also acht Threads mit einer Speicheranfrage bedient werden können. Eine solche Speicheroperation im DRAM benötigt 400 bis 800 Taktzyklen. ([NVI12], S. 67)

In Bezug auf das Caching der Speicherzugriffe wurde bereits der L1-Cache angesprochen. Dieser ist in der Darstellung des SMs zusammen mit dem Shared Memory abgebildet (siehe Abbildung 4.2). In der Praxis wird der 64 KB große On-Chip-Speicher zwischen dem Shared Memory, welcher im OpenCL Speichermodell den Local Memory darstellt, und dem Level-1-Cache geteilt. Dafür gibt es zwei

mögliche Aufteilungen, je nachdem wie viel Local Memory innerhalb einer Workgroup benötigt wird. Entweder werden 48 KB dem Local Memory und 16 KB dem L1-Cache zugeteilt, oder nur 16 KB dem Local Memory und die verbleibenden 48 KB dem L1-Cache. (vgl. [NVI09], S. 7-10)
Die Fermi Architektur enthält noch weitere Verbesserungen, die für das Rechnen mit GPUs und damit für dieses Programm von Vorteil sind. Zum einen werden Floating Point Operationen mit einfacher Genauigkeit nach dem IEEE 754-2008 Standard ausgeführt, zum anderen ist dies die erste GPU Architektur, welche Error Correction Code (ECC) Unterstützung bietet (vgl. [NVI09], S. 13-17). Außerdem wurde die Geschwindigkeit der atomaren Speicheroperationen durch zusätzliche Hardwareunterstützung und den neu eingeführten Level-2-Cache erhöht. Die Zugriffe sollen bis zu zwanzigmal schneller sein als auf der Vorgängerarchitektur ([NVI09], S. 17). Diese Operationen werden in der Implementierung des systolischen Rings mit Reduktion im Local Memory genutzt. Dennoch darf nicht vergessen werden, dass atomare Speicheroperationen immer noch bis zu zehnmal langsamer ablaufen als nicht atomare ([SO11], S. 5).

4.2.2 Laufzeitvergleich

In Abbildung 4.4 sind die Laufzeiten der vier Methoden in Abhängigkeit der simulierten Teilchenanzahl dargestellt.

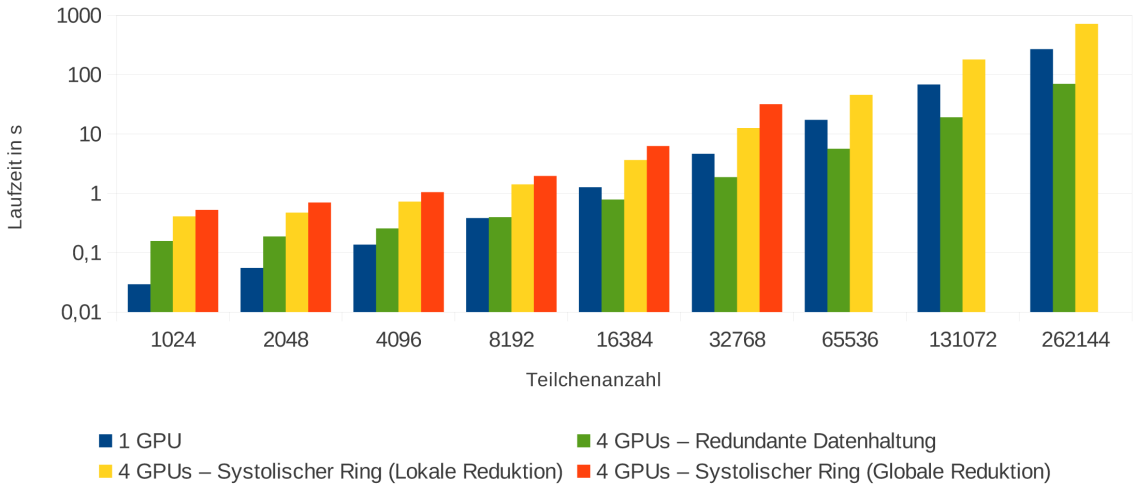

Abbildung 4.4: Laufzeiten von 100 Iterationen

Bei geringen Teilchenzahlen ist zunächst festzustellen, dass die Ausführung auf einer GPU am schnellsten ist. Hier ist der Rechenaufwand noch nicht groß genug, um die zusätzliche Kommunikation zwischen den Geräten aufzuwiegen.
Außerdem ist festzuhalten, dass bei Teilchenzahlen unter 16 000 selbst eine einzelne GPU noch nicht ausgelastet ist. Dies kommt durch die Art und Weise, wie die Workitems oder Threads innerhalb einer GPU abgearbeitet werden, zustande. Auf der genutzten Tesla-Karte wird eine Workgroup auf einem SM ausgeführt. Diese wird dafür nochmals in Gruppen von 32 Workitems unterteilt, welche dann als sogenannte Warps auf die Kerne eines SMs aufgeteilt werden. Auf einem SM können dabei je nach Auslastung der Ressourcen (Register, Local Memory) mehrere Workgroups aktiv sein. Da die Latenzen für

Speicheroperationen im Global Memory auf einer GPU sehr hoch sind, kann eine gerade aktive Gruppe von Workitems innerhalb eines Taktes durch eine andere Gruppe ersetzt werden. Dadurch können die Wartezeiten bei Speicheroperationen überdeckt werden, da die Funktionseinheiten währenddessen von anderen Workitems genutzt werden. Dieses Vorgehen ist nur möglich, wenn auf der GPU mehr Workitems als Kerne zur Ausführung bereitstehen.[NVI09]
Eine GPU der Tesla S2050 hat 14 SMs mit jeweils 32 Kernen, es existieren 448 Kerne auf einer GPU. Die optimale Größe einer Workgroup für dieses Programm beträgt 256 Workitems. Eine Ausnahme stellt dabei die Methode der Reduktion im Local Memory dar, worauf später noch eingegangen wird. Bei 3584 Teilchen beträgt die Arbeitslast der GPU erst eine Workgroup pro SM. Mit zunehmender Teilchenzahl können die SMs also besser ausgelastet werden, was in Abbildung 4.5 anhand der steigenden GFlop/s zu erkennen ist. Speziell bei der redundanten Methode ist zu sehen, dass die besten Laufzeiten erst bei sehr großen Teilchenmengen erreicht werden, da erst dann jede der vier GPUs ausgelastet werden kann.

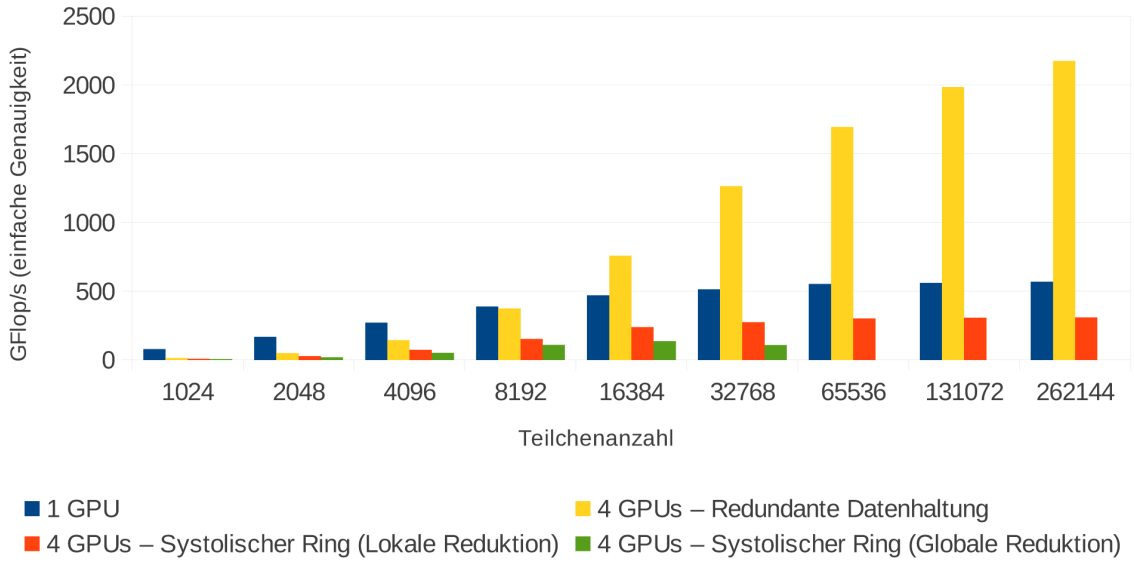

Abbildung 4.5: GFlop/s, während 100 Iterationen gemessen

Bei Teilchenzahlen größer 16 000 ist erstmals eine Verbesserung der Laufzeit bei der redundanten Datenhaltung auf 4 GPUs im Vergleich zur Ausführung auf einer GPU zu erkennen ($S_p = 1,6$; $E_p = 0,4$). Bei den beiden Implementierungen des systolischen Rings sind jedoch zu keinem Zeitpunkt Laufzeitverbesserungen festzustellen. Aufgrund der Kosten für Kommunikation und der Tatsache, dass die Algorithmen des systolischen Rings wesentlich stärker an den Speicher gebunden sind als die Implementierung mit nur einer GPU, kommt es sogar zu einer Verlängerung der Ausführungszeit. Dies wird in den nächsten Abschnitten erörtert werden.
Des Weiteren sei erwähnt, dass für den systolischen Ring mit globaler Reduktion keine Ergebnisse für Teilchenzahlen größer 32768 vorliegen, da dies die in Abschnitt 3.3.2 erläuterten quadratischen Speicheranforderungen nicht zulassen.

4.2.3 Speicherbindung der Algorithmen auf der GPU

Ein entscheidender Punkt für die Laufzeit von Programmen ist die Häufigkeit von Speicherzugriffen. Besonders auf GPUs ist der Zugriff auf den Global Memory mit hohen Latenzen verbunden. Deshalb werden die vier vorgestellten Algorithmen zuerst auf Speicherzugriffe innerhalb der Berechnung untersucht. Die einzelnen Eigenschaften der Teilchen sind in `float4` Vektoren gespeichert, damit diese im Arbeitsspeicher korrekt ausgerichtet vorliegen.
Der erste und der dritte Schritt des in Abschnitt 3.4 erläuterten Velocity-Verlet-Integrators laufen bei allen Algorithmen gleich ab, weshalb hier nur der zweite Schritt, die Bestimmung der auf die Teilchen wirkenden Kräfte, berücksichtigt wird.

Nutzung einer GPU

Im Falle der Implementierung mit nur einer GPU kommt es in jedem Thread zuerst zu einer Ladeoperation, um die Position des Teilchens, welches dem Thread zugeordnet ist, aus dem Global Memory zu holen. Dabei wird ein `float4` Vektor geladen, in dem die Positionsdaten des Teilchens abgelegt sind. Dann iteriert dieser Thread in Blockabschnitten über alle anderen Teilchen, wie in Abschnitt 3.3.1 in Abbildung 3.3 gezeigt. Hierbei werden die Positionen der anderen Teilchen geladen, was n weitere Ladeoperationen bedeutet, wobei n für die Anzahl der Teilchen im System steht. Da diese Daten in den Local Memory geschrieben werden und damit für eine ganze Workgroup zur Verfügung stehen, bedeutet dies, dass die Anzahl der Ladeoperationen durch die Größe der Workgroup geteilt wird. In diesem Fall ist die Workgroupgröße 256, also ergeben sich $\frac{n}{256}$ Speicherzugriffe.
Nachdem alle Wechselwirkungen bestimmt worden sind, wird die resultierende Kraft in einer Speicheroperation im Global Memory abgelegt.
Für eine Berechnung der Kräfte auf alle Teilchen bedeutet das in Summe $n(2 + \frac{n}{256})$ Speicherzugriffe. Dem gegenüber stehen $n(38+22n)$ Rechenoperationen. Dies sind wesentlich mehr, sodass die Speicherlatenzen bei entsprechender Auslastung der GPU, wie in Abschnitt 4.2.1 erklärt, gut verdeckt werden können.

Redundante Datenhaltung

Für die Implementierungen auf vier GPUs muss beachtet werden, dass sich die Teilchenanzahl pro Gerät viertelt. Die Teilchenanzahl pro Gerät wird im Folgenden mit n_d bezeichnet.
Bei der redundanten Datenhaltung ist der Algorithmus, welcher auf jeder GPU parallel ausgeführt wird, identisch zu dem, welcher in der Implementierung mit einer GPU verwendet wird. Der Unterschied besteht hier lediglich darin, dass jedes Gerät nur die Kräfte für die ihm zugeordneten Teilchen berechnet, also für n_d Teilchen. Es werden demnach auf jeder GPU $n_d(2+\frac{n_d}{256})$ Speicherzugriffe und $n_d(38+22n_d)$ Rechenoperationen ausgeführt.

Systolischer Ring mit globaler Reduktion

Im systolischen Ring werden die Wechselwirkungen zwischen den lokalen Teilchen wie bei der Implementierung mit einer GPU berechnet. Das bedeutet, dass dafür pro Gerät $n_d(2 + \frac{n_d}{256})$ Speicherzugriffe und $n_d(38 + 22n_d)$ Rechenoperationen ausgeführt werden.
Bei der Berechnung der Kräfte in Abhängigkeit von externen Teilchen fällt zu Beginn eine Ladeoperation für die Positionsdaten des eigenen Teilchens an sowie $\frac{n}{256}$ weitere Ladeoperationen für die Daten der externen Teilchen, welche wieder innerhalb einer Workgroup gemeinsam genutzt werden. Zusätzlich

wird jetzt das Resultat jeder einzelnen Interaktion im Global Memory abgelegt, dies bedeutet weitere n_d Speicheroperationen. Am Schluss der Berechnung wird zunächst die Kraft, welche auf das lokale Teilchen wirkt, zu dem bereits im Global Memory liegenden Wert addiert. Dies kostet eine Lade- und eine Speicheroperation. Als letzter Schritt wird die Reduktion über die im Global Memory befindliche Kraftmatrix durchgeführt. Das heißt, es werden weitere $n_d + 1$ Speicheroperationen notwendig, wobei die letzte für das Abspeichern der akkumulierten Kraft, welche auf das externe Teilchen wirkt, steht.
Nachdem diese Berechnung abgeschlossen wurde, tauschen die Geräte ihre Kraftvektoren der jeweils externen Teilchen über den Host aus. Jedes Gerät erhält nun einen Vektor mit Kräften, der auf den bereits auf dem Gerät befindlichen Kraftvektor der lokalen Teilchen addiert werden muss. Dies bedeutet, jeder Thread führt nochmals zwei Lade- und eine Speicheroperation aus. Insgesamt ergibt das für eine Berechnung von Wechselwirkungen mit externen Teilchen $n_d(7 + \frac{n_d}{256} + 2n_d)$ Speicheroperationen. Demgegenüber stehen $33n_d^2$ Rechenoperationen.
Da die Berechnung der Kräfte in einem systolischen Ring allerdings erst, wie in Abschnitt 2.3.2 erläutert, nach $\lceil (N_p - 1)/2 \rceil$ Iterationen fertiggestellt ist, müssen diese Werte noch angepasst werden. Im Falle von vier GPUs heißt dies, dass zwei Iterationen benötigt werden zuzüglich der Berechnung der lokal entstehenden Kräfte. Das bedeutet, dass in Summe pro Gerät $n_d(16 + \frac{3}{256}n_d + 4n_d)$ Speicheroperationen $n_d(38 + 88n_d)$ Rechenoperationen gegenüberstehen.

Systolischer Ring mit lokaler Reduktion

Mit der Reduktion im Local Memory soll die starke Bindung zum Global Memory vermindert werden. Dafür werden die Kräfte jeder einzelnen Interaktion innerhalb eines Berechnungsabschnitts, welche vorher im Global Memory in einer Kraftmatrix mit n_d^2 Einträgen abgegelegt wurden, im Local Memory zwischengespeichert. Die Größe des Abschnitts entspricht der Größe der Workgroup. Diese ist bei diesem Vorgehen nicht mehr 256, sondern nur noch 32 Workitems. Dies ergibt sich aus den Speicheranforderungen an den Local Memory. Eine Kraftmatrix der Größe 32×32, wobei jeder Eintrag einen `float4` Vektor fasst, benötigt $32 * 32 * 16Byte = 16KByte$ Speicher im Local Memory. Die nächstgrößere Workgroupgröße wäre 64. Dafür würden $64 * 64 * 16Byte = 64KByte$ Speicher im Local Memory benötigt werden. Das ist bei der aktuellen Hardware aber nicht umsetzbar, da der gesamte Shared Memory eines SMs nur 64 KByte fasst und innerhalb dieses Speichers noch Positionsdaten zwischengespeichert werden müssen sowie ein Teil davon (16 KByte) für den Level-1-Cache genutzt wird.
Durch die geänderte Größe der Workgroup verändern sich auch die Speicherzugriffe. Für das Laden der Positionsdaten externer Teilchen werden jetzt $\frac{n_d}{32}$ Speicheroperationen nötig, also acht mal mehr als vorher. Nachdem die Wechselwirkungen mit einer Gruppe von 32 externen Teilchen berechnet und im Local Memory zwischengespeichert wurden, werden die Spaltensummen dieser Matrix gebildet. Diese Summen stellen jeweils einen Teil der Kraft dar, welche auf das externe Teilchen wirkt. Diese muss nun zu einem Kraftvektor im Global Memory addiert werden, was mit zwei atomaren Speicheroperationen bewerkstelligt wird. Das führt pro Thread zu $\frac{2n_d}{32}$ atomaren Speicheroperationen.
Dazu wird noch eine Ladeoperation für die eigene Position und eine Speicheroperation für die eigene resultierende Kraft pro Thread fällig. Dies ergibt für eine Iteration im systolischen Ring pro Gerät $n_d(2 + \frac{3n_d}{32})$ Speicheroperationen und $33n_d^2$ Rechenoperationen.

Zusammen mit der Berechnung der lokalen Kräfte und der zweiten benötigten Iteration im systolischen Ring ergeben sich $n_d(6+\frac{49n_d}{256})$ Speicheroperationen und $n_d(38+88n_d)$ Rechenoperationen. Dabei darf nicht vergessen werden, dass ein Teil der Speicheroperationen ($\frac{n_d^2}{8}$) atomar sind und deshalb wesentlich mehr Zeit erfordern als eine normale Speicheroperation.

Verhältnis zwischen Speicher- und Rechenoperationen

Damit eine GPU ihre maximale Rechengeschwindigkeit erreichen kann, müssen ihre Funktionseinheiten immer ausgelastet sein. Dies wird nur durch Verdecken von Latenzen möglich. Mit 400-800 Takten haben Speicherzugriffe die größte Latenz. Damit sie verdeckt werden können, müssen genügend arithmetische Operationen in einem Kernel vorkommen. Nur dann kann der Thread Scheduler eines SMs bereitstehende Threads aktivieren, damit sie ihre Berechnungen durchführen, während andere Threads auf ihre Daten warten. Das Verhältnis zwischen den Speicherzugriffen und den Rechenoperationen pro Gerät ist in Abbildung 4.6 für die vier Algorithmen dargestellt. Dabei wird davon ausgegangen, dass für die parallelen Algorithmen vier Geräte verwendet werden.

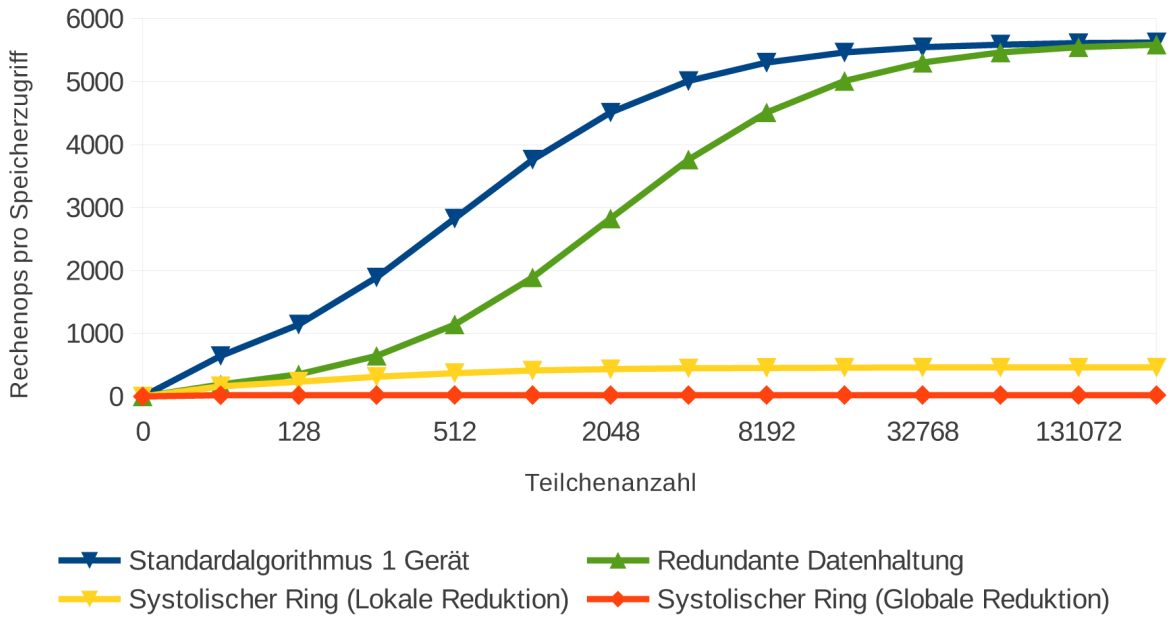

Abbildung 4.6: Anzahl der Rechenoperationen je Speicherzugriff pro Gerät

Das Verhältnis beim Vorgehen mit einer GPU bzw. der redundanten Datenhaltung beträgt bei ausreichend großer Teilchenzahl 5600 Rechenoperationen pro Speicheroperation. Das ist ein sehr gutes Verhältnis um die Speicherlatenzen zu verdecken. Bei den beiden Umsetzungen mit dem systolischen Ring ist das Verhältnis deutlich schlechter. Es kommen bei der lokalen Reduktion 460 und bei globaler Reduktion nur 22 Rechenoperationen auf einen Speicherzugriff. Damit ist zumindest für die Methode der globalen Reduktion ersichtlich, dass zu wenig arithmetische Operationen vorhanden sind, um eine Latenz von mindestens 400 Takten zu überbrücken.

Die 460 Rechenoperationen pro Speicherzugriff bei der Methode der lokalen Reduktion lassen zunächst vermuten, dass diese Anzahl, geht man von einer optimalen Latenz von 400 Takten aus, ausreichen

müsste, um die Verzögerung auszugleichen. Jedoch wurden bei dieser Untersuchung die atomaren Speicherzugriffe nicht gesondert betrachtet. Wird davon ausgegangen, dass ein atomarer Zugriff zehnmal langsamer erfolgt als ein nicht atomarer ([SO11], S. 5), müssen demzufolge die atomaren Speicherzugriffe mit dem Faktor 10 multipliziert werden. Danach ergeben sich in der oben angeführten Rechnung $n_d(6 + \frac{481n_d}{256})$ Speicherzugriffe auf die $n_d(38 + 88n_d)$ Rechenoperationen. Damit sinkt das Verhältnis auf 1 : 47. Hieraus wird ersichtlich, dass auch in dieser Implementierung die durch die Speicherzugriffe bedingten Wartezeiten nicht annäherungsweise mit Berechnungen gefüllt werden können. Die relativ großen Speicheranforderungen an den Local Memory lassen zudem nur zwei residente Workgroups pro SM zu. Dadurch hat der Thread Scheduler keine Freiheiten bei der Auswahl der Threads, was zu einer nicht optimalen Hardwareauslastung führt.

Datendurchsatz auf der GPU

Ausgehend von den zuvor bestimmten Speicherzugriffen kann nun ermittelt werden, wie hoch der Datendurchsatz auf der GPU ist. Hierfür wurden die Laufzeiten der Kernel mit einer Teilchenanzahl von 131072 bestimmt. Daraus errechnet sich das in Abbildung 4.7 dargestellte Ergebnis.

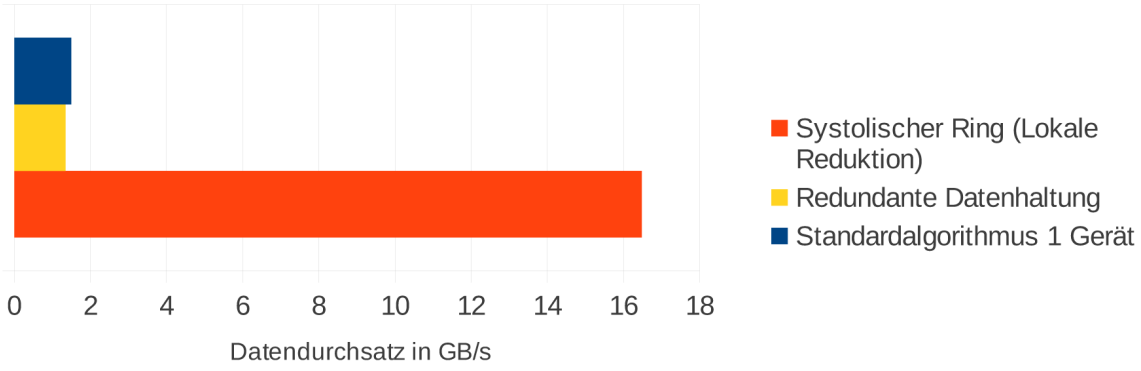

Abbildung 4.7: Datendurchsatz zum Arbeitsspeicher auf der GPU

Es ist ein Durchsatz von 1,5 Gigabyte pro Sekunde zum Hauptspeicher der GPU für die Variante mit einer GPU abzulesen. Für die redundante Datenhaltung beläuft sich dieser Wert auf 1,4 GB/s. Der speicherintensive Code des systolischen Rings erreicht dagegen einen Durchsatz von 16,5 GB/s. Die Abbildung veranschaulicht wie unterschiedlich das Verhalten der Ansätze beim Datendurchsatz ist und welche negativen Auswirkungen die hohe Speicherbindung bei der Methode des systolischen Rings besitzt.

4.2.4 Verhalten der Kernel auf dem Gerät

Die vier verschiedenen Lösungswege nutzen insgesamt drei verschiedene Kernel, welche auf dem Gerät ausgeführt werden. Der erste dieser Kernel berechnet nach dem in Abschnitt 3.3.1 erläuterten Vorgehen die Kräfte zwischen den lokal auf dem Gerät befindlichen Teilchen. Die Variante mit einer GPU sowie die Methode der redundanten Datenhaltung nutzen ausschließlich diesen Kernel, welcher in den folgenden Abbildungen mit `calculate_localCoulombForces` betitelt ist. Bei der redundanten Datenhaltung wird lediglich der Teilchenbereich eingeschränkt, für welchen die Kräfte berechnet werden, nicht aber die Anzahl der für ein Partikel zu berechnenden Wechselwirkungen. Aufgrund der Tatsache, dass

dieser Kernel für jedes Teilchen die resultierende Kraft berechnet, ohne die Symmetrie der Kraftmatrix zu nutzen, wird er Brute-Force-Kernel genannt.
Die anderen beiden Kernel werden von den systolischen Algorithmen genutzt. Es existiert ein Kernel für die globale und einer für die lokale Reduktion. Diese beiden setzen die in Abschnitt 3.3.2 erklärten Algorithmen des systolischen Rings um. Bei beiden Vorgehensweisen, der globalen und der lokalen Reduktion, werden die Kräfte, welche zwischen den lokalen Teilchen entstehen, mit dem Brute-Force-Kernel evaluiert. Die Wechselwirkungen mit den externen Teilchen werden mit Hilfe des systolischen Kernels der jeweiligen Implementierung berechnet.
Das Verhältnis der Laufzeiten dieser beiden Kernel ist in einem Trace in Abbildung 4.8 für die globale Reduktion und in Abbildung 4.9 für die lokale Reduktion dargestellt.

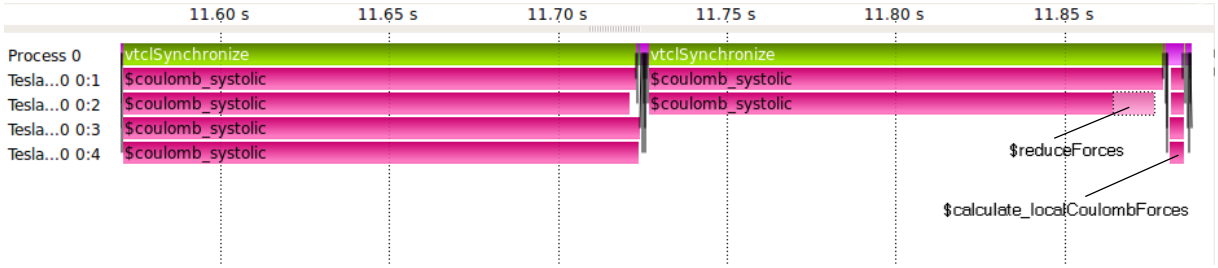

Abbildung 4.8: Ausschnitt eines Trace für den systolischen Ring mit globaler Reduktion

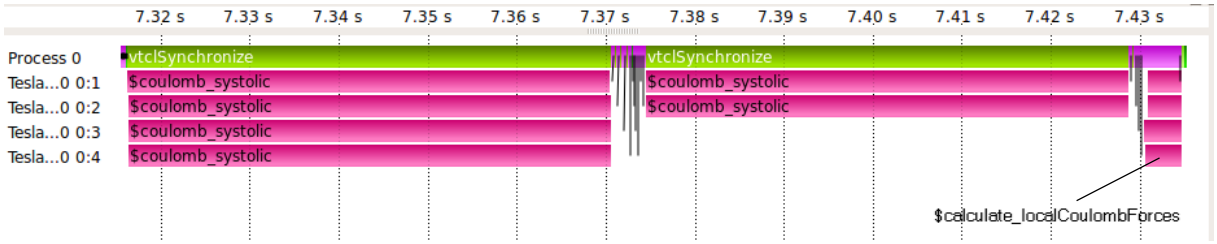

Abbildung 4.9: Ausschnitt eines Trace für den systolischen Ring mit lokaler Reduktion

Der Ausschnitt zeigt jeweils die Berechnung einer kompletten systolischen Iteration, das heißt aller Wechselwirkungen innerhalb des Systems. In diesem Beispiel hat das System 32 768 Teilchen.
Es ist ein großer Unterschied zwischen den Laufzeiten der unterschiedlichen Kernel festzustellen. Bei der globalen Reduktion benötigt der systolische Kernel im Mittel 150 Millisekunden; dem gegenüber stehen 4 Millisekunden, welche für den Brute-Force-Kernel aufgewendet werden müssen. Dies entspricht einem Verhältnis von $37,5 : 1$. Die lange Ausführungszeit des systolischen Kernels resultiert aus der hohen Speicherbindung. Bei der globalen Reduktion wird auch der Aufwand für die Reduktion der Kraftmatrix erkennbar. In Abbildung 4.8 ist dies als gesondert aufgerufener Kernel namens `reduceForces` sichtbar. Die Bearbeitungszeit einer systolischen Iteration setzt sich aus der Kraftberechnung, welche im Kernel mit dem Namen `coulomb_systolic` erfolgt, und der Reduktion der Kräfte zusammen. Erstere benötigt in diesem Fall 138 Millisekunden und die Reduktion 12 Millisekunden. Das Verhältnis von Speicheroperationen zu Rechenoperationen während der Reduktion beträgt $1 : 2$. Dies bedeutet, dass dieser Teil des Programms komplett speichergebunden ist. Damit dauert die Reduktion bereits länger als die Ausführung des Brute-Force-Kernels für die identische Anzahl an Teilchen. Daran ist deutlich zu

erkennen, dass sich speicherintensive Programme ohne Datenwiederverwendung sehr schlecht auf GPUs verhalten.
Die lokale Reduktion ist weniger speicherintensiv als die globale. Hier existiert auch keine gesonderte Reduktionsphase, was sich in der Laufzeit dieses Kernels bemerkbar macht. Der in Abbildung 4.9 gezeigte Kernel `coulomb_systolic` benötigt 54 Millisekunden. Diese stehen wiederum den 4 Millisekunden des Brute-Force-Kernels gegenüber. Das ergibt ein Verhältnis von $13,5:1$. Diese Relation ist zwar günstiger als bei der globalen Reduktion, zeigt allerdings immer noch, dass der Brute-Force-Kernel wesentlich effizienter arbeitet.
Da beim systolischen Ring jedoch die Symmetrie der Kraftmatrix ausgenutzt wird, deshalb eine systolische Iteration das Ergebnis von zwei Durchläufen des Brute-Force-Kernels liefert, muss das Verhältnis für die Effizienzbetrachtung noch angepasst werden. Es ergibt sich für die globale Reduktion ein Verhältnis von $18,75:1$ und für die lokale Reduktion eines von $6,75:1$.
Mit steigender Teilchenzahl verschlechtert sich diese Relation. Für die globale Reduktion konnten für größere Systeme aufgrund der zu hohen Speicheranforderungen keine Messungen mehr durchgeführt werden. Für die lokale Reduktion wurde für ein System mit 131 072 Teilchen ein Verhältnis von $9:1$ gemessen.

4.2.5 Kommunikation zwischen Host und Gerät

Bei Programmen, welche mit GPUs beschleunigt werden, ist oftmals die Kommunikation zwischen der GPU und dem Host, das heißt der CPU, ein Flaschenhals und somit bestimmend dafür, ob der Einsatz von GPUs sinnvoll ist. Deshalb wird im folgenden Abschnitt die Kommunikation zwischen Host und OpenCL Gerät für die unterschiedlichen Lösungsalgorithmen untersucht.

Nutzt man eine GPU zur Berechnung, werden lediglich zu Beginn die Positions- sowie Geschwindigkeitsdaten der Teilchen auf die GPU kopiert und die Kraftvektoren initialisiert. Weitere Kommunikationen fallen nur für die Protokollierung des Simulationsverlaufs an. Hierfür werden die Positionen und Geschwindigkeiten der Teilchen wieder an den Host übermittelt, welcher diese dann abspeichert. Außerdem wird die mittlere Geschwindigkeit der Teilchen berechnet, um zu bestimmen, wie weit das System bereits abgekühlt ist und ob die Simulation gegebenenfalls beendet werden kann.
Bei der redundanten Datenhaltung werden stets zu Beginn eines Zeitschritts der Simulation die Positionen der jeweiligen lokalen Teilchen der Geräte an den Host übermittelt. Danach liegen die aktuellen Positionsdaten des gesamten Systems im Speicher des Hosts. Diese Daten müssen nun an alle Geräte verteilt werden, damit sie die Berechnung des nächsten Zeitschritts auf der Grundlage aktueller Daten durchführen können. Dies bedeutet pro Iteration für jedes Gerät eine Nachricht der Größe $16 * \frac{n_d}{D}$ Byte und eine weitere mit $16 * n_d$ Byte, wobei D für die Anzahl der Geräte steht und n_d für die Anzahl der lokalen Teilchen.
Für den systolischen Ring fallen für die Berechnung eines Zeitschritts mehr Kommunikationen an. Die erste dient dazu, dem Host die aktuellen Positionsdaten des Systems zu übermitteln. Dieser Datenaustausch ist nur einmal innerhalb jedes Zeitschritts notwendig, da sich die Positionen während der einzelnen Schritte des systolischen Rings nicht verändern. Anschließend werden die Positionen unter den Geräten ausgetauscht. Nach Abschluss der ersten Berechnungsphase wird der resultierende Kraftvektor an den Host zurückübermittelt, damit dieser dann an das Gerät gesendet werden kann, welches diesen

Kraftvektor zu dem seiner lokalen Teilchen dazu addiert. Während eines Schritts im systolischen Ring fallen also drei Kommunikationen pro Gerät an. Je nachdem wie viele Geräte zur Verfügung stehen, wird dieser Schritt noch mehrmals wiederholt. Alle diese Kommunikationen haben eine Größe von $16 * \frac{n_d}{D}$ Byte.
Im hier untersuchten Fall werden vier Geräte verwendet. Dies bedeutet, es sind zwei solcher Schritte im systolischen Ring notwendig. Da die Anzahl der Geräte gerade ist, muss im letzten Schritt nur noch die Hälfte der Geräte Berechnungen durchführen, weshalb dann aus der Sicht des Hosts nur noch die Hälfte der Kommunikationen notwendig sind.
Für vier Geräte sind zu Beginn aus Sicht des Hosts also vier Kommunikationen zur Aktualisierung der Positionsdaten auf dem Host erforderlich sowie im ersten Schritt im systolischen Ring zwölf Kommunikationen. Im zweiten und gleichzeitig letzten Schritt benötigt man nur noch sechs Kommunikationen um die Berechnung abzuschließen. Insgesamt ergeben sich für diesen Algorithmus also zweiundzwanzig Kommunikationen für einen Zeitschritt.

Um die Bandbreite dieser Kommunikationen zu untersuchen, lassen sich im Anwendungstrace die einzelnen Kommunikationen sowie deren Übertragungsbandbreiten ermitteln. Diese variieren sehr stark von 1,5 bis 7 GB/s.
Eine bessere Einschätzung zum Vergleich der Kommunikationslast der Algorithmen erhält man, wenn man die Übertragungsraten über die Laufzeit der Berechnung mittelt und so die durchschnittliche Bandbreite bestimmt. Für die 1-Gerät-Variante ist diese Übertragungsrate gleich null, da lediglich zu Beginn und zum Schluss der gesamten Berechnung eine Kommunikation anfällt. Bei der redundanten Datenhaltung dagegen existieren Kommunikationen während der Berechnung, sodass hier ein mittlerer Durchsatz von 5,8 MB/s entsteht. Der wesentlich kommunikationsintensivere systolische Ring erreicht mit 56 MB/s einen etwa zehnmal so großen Wert.

4.3 Analyse auf CPUs

Da OpenCL Kernel sowohl auf GPUs als auch auf CPUs ausgeführt werden können, wird im Folgenden das Verhalten der Anwendung auf CPUs untersucht.

4.3.1 Architektur der verwendeten CPUs

Das Programm wurde auf zwei unterschiedlichen CPU Architekturen getestet. Um das Verhalten des Programms korrekt deuten zu können, werden zuerst die wichtigsten Teile der CPUs beschrieben.

Intel Xeon E5620

Der Intel Xeon E5620 ist nach der Westmere-Architektur gebaut. Dies ist der Nachfolger der Nehalem-Architektur und setzt eine Strukturverkleinerung sowie kleinere Funktionsverbesserungen dieser um.
Das Modell besitzt vier Kerne, welche auf 2,4 GHz getaktet sind. Jeder dieser Kerne besitzt einen eigenen Level-1-Cache mit jeweils 32 KB Daten- und Instruktionscache. Ebenfalls pro Kern ist ein uniformer Level-2-Cache von 256 KB vorhanden. Die letzte Cache-Stufe ist der Level-3-Cache, welcher mit einer Kapazität von 12 MB aufwartet und von den vier Kernen gemeinsam genutzt wird.
Die CPU unterstützt die Hyper-Threading-Technologie und kann somit acht Threads gleichzeitig ausführen. Der genutzte Rechenknoten ist eine Dual-Sockel-Konfiguration, kann also insgesamt sechzehn

Threads gleichzeitig abarbeiten. Außerdem beherrscht die CPU die Befehlssatzerweiterung SSE4.2, welche Vektoren mit 128 Bit Breite verarbeitet. Damit können pro Takt vier Fließkommaadditionen oder Multiplikationen mit einfacher Genauigkeit gerechnet werden. Da jeder Kern eine Vektoreinheit für die Ausführung dieser Befehle besitzt, kann das gesamte System zweiunddreißig Fließkommaoperationen pro Takt berechnen. Bei einer Taktrate von 2,4 GHz ergibt sich eine theoretische Peak Performance von 256 GFlop/s. (vgl. [Int12], S. 85-97)

AMD Opteron 6276

Der AMD Opteron 6276 entstammt der aktuellen Bulldozer-Architektur. Hier werden zum ersten Mal Module statt der klassischen Kerne verwendet. Dabei werden zwei Integerkerne zu einem Modul zusammengefasst. Diese zwei Kerne teilen sich weitere Ressourcen, wie zum Beispiel Instruktionsdekoder und die Floating-Point-Einheit, welche vor allem für diese Anwendung relevant ist.
Eine CPU des hier verwendeten Modells besitzt sechzehn Kerne, also acht Module. Jedes Modul besitzt einen 64 KB großen Level-1-Instruktionscache und zwei 16 KB große Level-1-Datencaches, also einen für jeden Kern. Der uniforme Level-2-Cache wird zwischen den beiden Kernen eines Moduls aufgeteilt und beläuft sich auf 2 MB. Die letzte Stufe in der Cache-Hierarchie, der Level-3-Cache, ist 16 MB groß. Dieser ist in zwei 8 MB große Teile gegliedert, welche sich jeweils vier Module teilen.
Diese CPU unterstützt im Gegensatz zum Intel Xeon E5620 bereits die AVX Befehlssatzerweiterung. Damit verdoppelt sich die Breite der Vektoren und damit auch die Anzahl der Floating-Point-Operationen pro Takt, demzufolge können acht dieser Operationen pro Takt fertiggestellt werden. Da das System mit vier Sockeln ausgestattet ist, und demzufolge zweiunddreißig Module (oder vierundsechzig Kerne) besitzt, ist die theoretische Peak Performance des Systems 588,8 GFlop/s. (vgl. [Adv12b], S. 29-41)

4.3.2 Ausführung von OpenCL auf CPUs

Bei der Ausführung von OpenCL Kernels auf CPUs gibt es einige Unterschiede im Vergleich zu GPUs bezüglich der Abbildung des Programms auf die Hardware.
Damit die Vektoreinheiten der CPUs genutzt werden können und somit eine hohe Rechenleistung erreicht werden kann, müssen im Programm vektorisierte Typen verwendet werden. Das Programm `SCPonGPUcl` verwendet einen solchen Typ und zwar `float4`.
Außerdem werden die Workitems einer Workgroup nicht wie auf einer GPU parallel ausgeführt. Die AMD Implementierung beispielsweise beschreibt, dass eine Workgroup auf einen Kern abgebildet wird und auf diesem bis zum Schluss ausgeführt wird (vgl. [Adv12a], S. 93-94). Dabei werden die Workitems aus der Gruppe serialisiert ausgeführt. Das kann dazu führen, dass Speicherzugriffsmuster, welche auf einer GPU performant sind, dies nicht auf einer CPU sind, da hier die Cache-Hierarchie beachtet werden muss. Benötigt das Programm zu viel Platz im Cache, kann es dazu kommen, dass jedes Workitem die Daten, welche vorher bereits im Cache lagen, erneut aus dem Arbeitsspeicher lesen muss.
Ein weiterer Aspekt bezüglich der Speichernutzung ist, dass es auf der CPU keinen extra Bereich für den Local Memory gibt. Der Einsatz des Local Memories kann sogar zur Verschlechterung der Laufzeit beitragen, weil beim Laden der Daten in den Local Memory zusätzliche Speicheroperationen ausgeführt werden müssen und sich durch die Nutzung dieses Bereichs außerdem häufiger zusätzliche Barrieren ergeben, was für die CPU einen erhöhten Aufwand aufgrund von Kontextwechseln verursacht.

Für das Hostprogramm erscheinen alle CPUs innerhalb eines SMP-Systems als ein OpenCL Gerät. Es besteht allerdings auch die Möglichkeit, das Gerät zu unterteilen. Das geschieht entweder mit der Device Fission Erweiterung oder ab Version 1.2 von OpenCL über Standard-API-Funktionen. Damit hat man die Möglichkeit, jeden Kern als ein Gerät zu betrachten oder Untergeräte mit beliebiger Anzahl der verfügbaren Kerne zu erstellen.

4.3.3 Vergleich der Algorithmen

Um mit dem vorhanden Programm die optimale Laufzeit auf CPUs zu erreichen, wird zunächst ein Vergleich der verschiedenen Algorithmen vorgenommen. Durch die Möglichkeit, die vorhandenen Kerne des Systems in mehrere Untergeräte aufzuteilen, ergeben sich mehrere Ansätze zur Programmausführung. Eine Variante ist, alle CPU Kerne als ein Gerät agieren zu lassen und dabei den Standardalgorithmus für ein Gerät zu verwenden. Die Laufzeit dieser Methode wird als Vergleichswert für die anderen Varianten verwendet.
Da es bei einigen Anwendungen sinnvoll sein kann, einen Kern für die Ausführung des Hostprogramms freizuhalten, läuft die zweite getestete Variante ebenfalls auf einem Gerät, dieses beinhaltet jedoch genau einen CPU Kern weniger als im System vorhanden sind. Die Messungen zu diesem Ansatz haben jedoch gezeigt, dass sich die Abspaltung eines Kerns für dieses Programm nicht lohnt. Die Berechnung dauerte auf dem Xeonknoten 4% länger und benötigte auf dem Opteronknoten sogar 21% mehr Zeit als mit allen Kernen.
Um die anderen Parallelisierungsalgorithmen zu testen, werden die CPUs in Untergeräte unterteilt. Hierbei wird die Anzahl der Kerne pro Gerät so gewählt, dass ein Gerät auf eine physische CPU abgebildet werden kann. Im Falle des Opterons sind dies sechzehn Kerne und für den Xeon Prozessor acht Kerne (hierbei zählt jeder Hardwarethread als Kern). Diese Gerätekonfiguration wird nun mit dem systolischen Ring und mit dem Ansatz der redundanten Datenhaltung analysiert.
Für den systolischen Ring ergibt sich auf beiden Knoten eine Verlängerung der Rechenzeit um 18%, was durch die vorher bereits beschriebene, höhere Speicherlast und Kommunikationshäufigkeit dieses Algorithmus verursacht wird.
Organisiert man die Kommunikation zwischen den Untergeräten mit der Methode der redundanten Datenhaltung, ergibt sich für den Xeonknoten eine Verlängerung der Laufzeit um 46% und für den Opteronknoten sogar um 296%. Diese hohen Werte erklären sich durch den Mehraufwand, der betrieben werden muss, um die Teilchendaten für jedes Gerät einzeln im Speicher vorzuhalten. Die Vervierfachung der benötigten Laufzeit auf dem Opteronknoten könnte darauf zurückzuführen sein, dass die Puffer der einzelnen Geräte vom OpenCL Treiber nicht im Arbeitsspeicher der jeweiligen CPU, welche das Gerät darstellt, angelegt werden. Das führt zu langen Zugriffszeiten auf Speicher der anderen CPUs im System. Bei einem Vergleichstest, bei dem ein Untergerät mit sechzehn Kernen als Referenzwert genommen wurde und mit zwei Geräten zu jeweils acht Kernen verglichen wurde, verdoppelte sich die Laufzeit nur.

4.3.4 Skalierung der Anwendung

Im Folgenden wird untersucht, wie sich das Programm verhält, wenn entweder die für die Simulation verwendete Teilchenanzahl oder die Anzahl der eingesetzten Kerne verändert wird. Dazu wird die starke und die schwache Skalierung analysiert.

Skalierung mit der Teilchenzahl

Das N-Körper-Problem hat eine quadratische Komplexität. Um zu überprüfen, ob man diese auch im Programmverlauf erkennt oder ob es bei bestimmten Teilchenzahlen Veränderungen der Laufzeit gibt, wird zuerst ein Test durchgeführt, bei welchem ein Rechenkern mit der Berechnung immer weiter steigender Teilchenzahlen beauftragt wird.

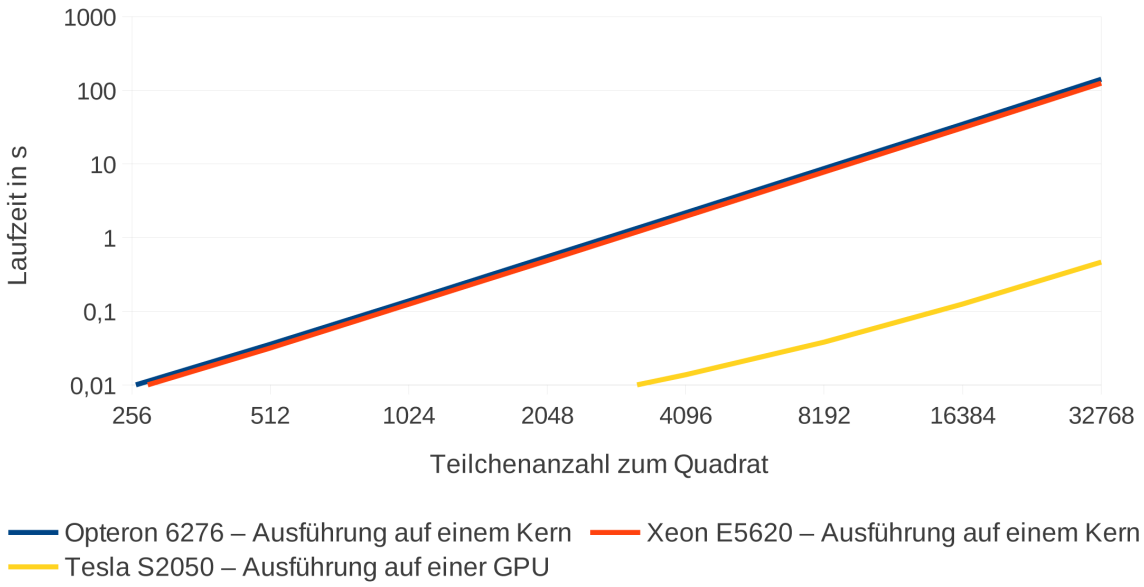

Abbildung 4.10: Skalierung der Laufzeit mit steigender Teilchenzahl

Abbildung 4.10 zeigt, dass die quadratische Zunahme der Laufzeit bereits bei geringster Teilchenanzahl zu erkennen ist. In der zum Vergleich aufgeführten Messreihe mit einer GPU verläuft die Gerade bis 8 192 Teilchen erst etwas flacher, da die GPU bei geringeren Teilchenzahlen noch nicht ausgelastet ist. Sobald die Systemgröße diese Zahl jedoch übersteigt, hat die Gerade dieselbe Steigung wie für die Messreihen mit CPUs.
Anderweitige Besonderheiten sind in den Laufzeitkurven nicht zu erkennen.

Schwache Skalierung

Bei der schwachen Skalierung wird untersucht, wie sich das Programm verhält, wenn die Problemgröße pro Prozessor konstant bleibt. Da die Komplexität des N-Körper-Problems quadratisch ist, darf in diesem Fall nicht einfach die Teilchenanzahl pro Prozessor konstant gehalten werden. Es muss stattdessen darauf geachtet werden, dass die nötigen Rechenoperationen pro Prozessor gleich bleiben. Die durchgeführte Messung startet mit einem Kern und einem System mit 65 536 Teilchen, dies bedeutet 4,3 Milliarden Operationen für einen Kern. In den nächsten Schritten wird stets die Prozessorzahl verdoppelt, dabei aber die Teilchenzahl nur mit 1,5 multipliziert. Damit bleiben die pro Kern berechneten Wechselwirkungen stets bei 4,3 Milliarden.
Das Ergebnis dieser Messung für den Opteronknoten ist in Abbildung 4.11 dargestellt.
Die Laufzeit der einzelnen Testläufe ist relativ zu der des ersten Laufs mit einem Kern angegeben. Auffällig ist der Anstieg, wenn man die Anzahl der Kerne von eins auf zwei erhöht. Danach ist ein konstantes und damit gutes Verhalten zu sehen. Dieser Anstieg erklärt sich durch die Architektur des Opteronpro-

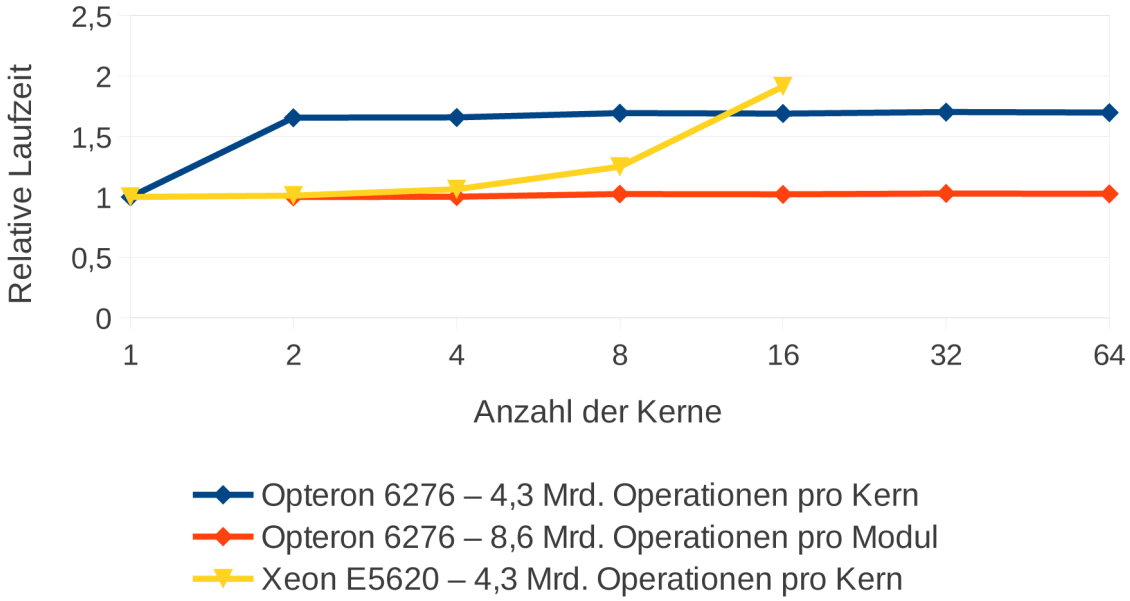

Abbildung 4.11: Schwache Skalierung auf CPUs

zessors. Da bei diesem immer zwei Kerne in einem Modul zusammengefasst sind und jedes Modul nur eine Floating-Point-Einheit besitzt, kommt bei der ersten Verdopplung der Kernanzahl keine weitere Floating-Point-Einheit hinzu. Da das Programm sehr viele Fließkommaoperationen enthält, kann der zusätzliche (Integer-)Kern keine ausreichende Leistungssteigerung bringen. Anschließend verhält sich die Kurve jedoch wieder konstant, da bei der Verdopplung der Kerne stets auch die Anzahl der Floating-Point-Einheiten verdoppelt wird. Betrachtet man für diese Messreihe die Module statt der Kerne als Recheneinheiten, dann ist der zweite Messpunkt der Reihe der Bezugspunkt für die weiteren Werte. Diese Annahme ist in der orange dargestellten Messreihe zu sehen. Hierbei entstehen 8,6 Milliarden Operationen pro Modul. Die so ermittelte schwache Skalierung verhält sich optimal.

Das Verhalten des Xeonknotens ist in Abbildung 4.11 als gelbe Messreihe dargestellt. Wie deutlich zu erkennen ist, verhält er sich nicht so optimal. Ein erster verstärkter Anstieg der Laufzeit ist bereits bei acht Kernen festzustellen. Hier spielt das Hyper-Threading der Intel CPU eine entscheidende Rolle. Da die CPU vier physische Kerne besitzt, dank Hyper-Threading aber acht Hardwarethreads verarbeiten kann, werden diese acht Threads vom OpenCL Treiber als acht Kerne wahrgenommen, sodass sich auch hier die Kerne wieder Ressourcen teilen. Im nächsten Untersuchungsschritt mit sechzehn Kernen ist zu erkennen, dass die Kommunikation über einen Sockel hinweg deutlich mehr Zeit in Anspruch nimmt, wodurch sich die relative Laufzeit verdoppelt.

Starke Skalierung

Die starke Skalierung untersucht das Verhalten des Programms bei gleichbleibender Problemgröße aber steigender Prozessoranzahl. Damit wird festgestellt, ob der Parallelisierungsaufwand ab einer bestimmten Grenze zu groß wird und sich eine weitere Parallelisierung nicht mehr lohnt.
Das Laufzeitverhalten wird mit Hilfe der parallelen Effizienz dargestellt. Dies ist der Quotient aus dem Speedup (relativ zur Ausführung mit einem Kern) und der Anzahl der verwendeten Kerne. Damit ergibt

sich ein Maß für die Güte der Parallelisierung.

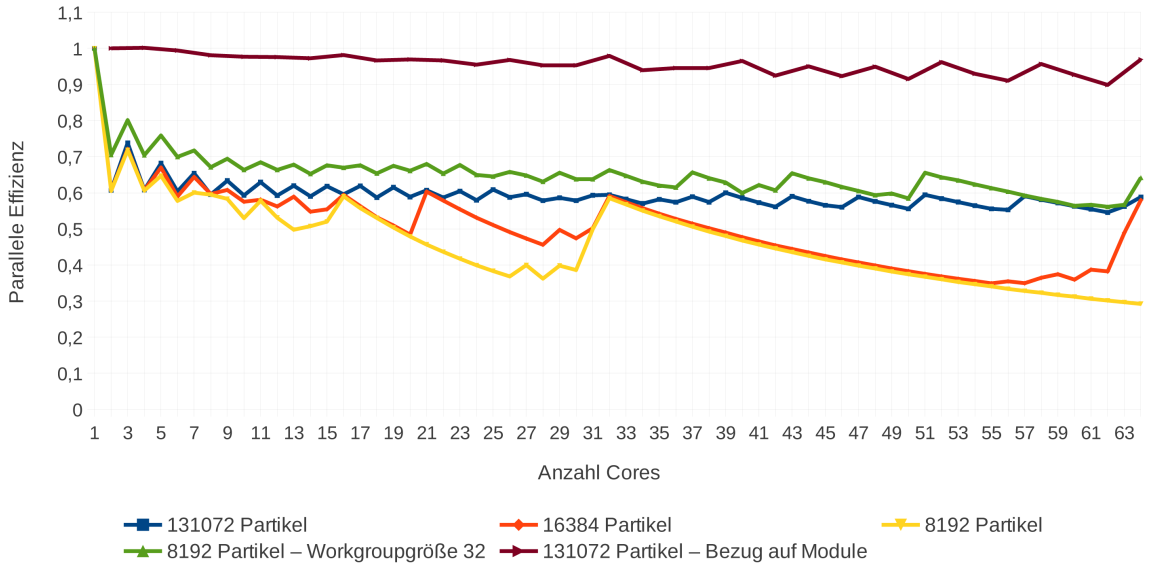

Abbildung 4.12: Starke Skalierung auf dem Opteronknoten (4x Opteron 6276)

In Abbildung 4.12 sind verschiedene Messreihen des Opteronknotens dargestellt. Bei allen Messreihen ist ein gezackter Verlauf der Werte zu sehen. Die Schwankung ist zu Beginn sehr deutlich zu erkennen. Die parallele Effizienz hat bei ungeraden Kernanzahlen einen höheren Wert als bei der vorherigen geraden Kernanzahl. Außerdem liegt die Effizienz bereits bei geringen Kernanzahlen schon weit unter dem Optimalwert eins.
Beides lässt sich mit der Architektur der Opteron CPUs erklären. Da bei der Nutzung von zwei Kernen nur ein Modul, also auch nur eine Floating-Point-Einheit genutzt wird, kann die parallele Effizienz keinen optimalen Wert erreichen. Das Verhalten verbessert sich, wenn man mit drei Kernen zwei Module anspricht. Normiert man die Skalierung auf Module, wie es in der Datenreihe mit dem Namen *131072 Partikel - Bezug auf Module* (weinrot) getan wurde, erhält man sehr gute Werte für die Skalierung.
Um zu überprüfen, wie sich die zur Parallelisierung notwendige Kommunikation auf die Effizienz auswirkt, wurden Messreihen mit geringeren Teilchenzahlen durchgeführt. Bei 8 192 Teilchen (gelb) sieht man ab der Nutzung von zweiunddreißig Kernen einen starken Abfall der Effizienz. Hier wurde keine Laufzeitverbesserung durch Erhöhung der Kernanzahl mehr erreicht. Bei 16 384 Teilchen (orange) hingegen findet sich am Schluss der Messreihe noch einmal ein starker Anstieg, der wieder auf das Niveau der 131 072er-Reihe führt. Beide Erscheinungen sind durch die Lastverteilung auf die vorhandenen Kerne zu begründen.
Das Programm verwendet eine Workgroupgröße von 256. Hat ein System nur 8 192 Partikel, dann werden diese in 32 Workgroups gegliedert. Da die OpenCL Implementierung von AMD eine Workgroup auf genau einen Kern abbildet, ist also nur Arbeit für zweiunddreißig Kerne vorhanden. Eine zweite Messung mit 8 192 Partikeln und einer Workgroupgröße von 32 (grün) zeigt ein ausgeglicheneres Verhalten.
Der Anstieg zum Schluss der 16 384er-Reihe erklärt sich durch die Abarbeitung der Workgroups. Bei dieser Teilchenanzahl und einer Workgroupgröße von 256 ergeben sich 64 Workgroups. Stehen nun nur 62

Kerne zur Verfügung und wird davon ausgegangen, dass alle Kerne in etwa gleich schnell arbeiten, gibt es einen Durchlauf mit 62 Workgroups und danach einen zweiten Durchlauf mit den zwei verbleibenden Workgroups. Dieser Durchlauf nutzt nun nur zwei Kerne aus, was die Effizienz stark beeinträchtigt. Werden dagegen 64 Kerne verwendet, erreicht das Programm eine wesentlich bessere Auslastung der Kerne und damit auch eine bessere parallele Effizienz.

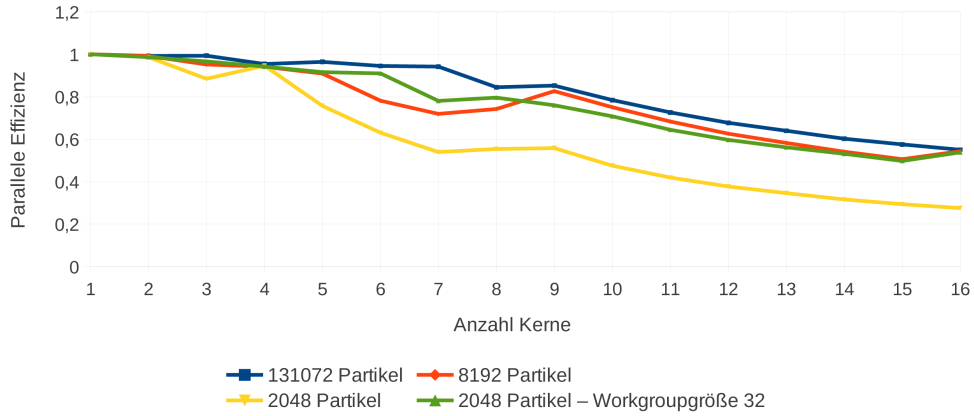

Abbildung 4.13: Starke Skalierung auf dem Xeonknoten (2x Xeon E5620)

In Abbildung 4.13 sind die Messergebnisse für den Xeonknoten dargestellt. Hier zeigen sich im anfänglichen Bereich gute Werte. Bei 131 072 Teilchen sinkt die Effizienz, sobald mehr als sieben Kerne verwendet werden, obwohl das System acht physische CPUs besitzt. Doch zusätzlich zu den OpenCL Threads besitzt das Programm auch noch einen Thread für das Hostprogramm. Wenn für die OpenCL Kernel bereits acht Kerne genutzt werden, dann muss für den Hostthread jedes Mal ein Kontextwechsel vollzogen werden. Bei Kernanzahlen größer acht sinkt die parallele Effizienz noch weiter ab. Dieses Verhalten erklärt sich durch die Hyper-Threading-Technologie. Die ersten acht OpenCL Threads können auf einen physischen CPU-Kern abgebildet werden. Weitere Threads werden dann auf den durch Hyper-Threading bereitgestellten logischen Kernen ausgeführt. Dabei teilen sich dann zwei Threads die Ressourcen eines physischen Kerns, sodass der Speedup nicht mehr im gleichen Maß wie vorher ansteigen kann und damit auch die parallele Effizienz sinkt.
Bei den Versuchsreihen mit 8 192 Teilchen sowie den beiden Reihen mit 2 048 Partikeln sieht man wieder das beim Opteron bereits erwähnte Lastverteilungsproblem. Bei 2 048 Teilchen entstehen acht Workgroups, weshalb die Effizienz bei höheren Kernanzahlen sinkt. Wird jedoch eine Workgroupgröße von 32 verwendet, skaliert diese Messung genauso wie die anderen.

4.4 Vergleich CPU und GPU

Um die geeignetste Hardware auswählen zu können, sind in Tabelle 4.3 einige Vergleichspunkte der untersuchten Architekturen aufgeführt. Die einzelnen Werte wurden jeweils unter Verwendung der maximal möglichen Anzahl an Kernen beziehungsweise Geräten ermittelt. Dies entspricht vierundsechzig Kernen für das Opteronsystem, sechzehn Kernen beim Xeonknoten und vier Tesla GPUs für die GPU-Variante.

Tabelle 4.3: Vergleich CPU und GPU

	AMD Opteron 6276	Intel Xeon E5620	NVIDIA Tesla S2050
Parallele Effizienz bei Teilchenzahlen zwischen 8 000 und 16 000	0,3 – 0,6	0,5	0,2 – 0,4
Parallele Effizienz bei Teilchenzahlen ab 130 000	0,6	0,5	0,9
Laufzeit für 131072 Teilchen (10 Zeitschritte)	59 s	226 s	5 s
GFlop/s bei einfacher Genauigkeit	64	17	1 980
Anteil der theoretischen Peak Performance	10,9 %	6,6 %	47,9 %
Maximale Teilchenanzahl für 31 Tage Rechenzeit	650 000	350 000	4 000 000

Beim Vergleich der parallelen Effizienz ist zu erkennen, dass die CPUs bei geringen Teilchenzahlen deutlich besser skalieren als GPUs. Hier machen sich die Unterschiede der beiden Architekturen bemerkbar. CPUs haben aufgrund ihrer höheren Taktrate geringere Latenzen für Rechenoperationen sowie schnelleren Zugriff auf Daten aus dem Arbeitsspeicher. Zudem besitzen CPUs die Fähigkeit der Out-of-Order Ausführung, um die Wartezeiten verschiedener Operationen zu verdecken. Da die GPU eine durchsatzorientierte Architektur besitzt, kommt ihr Potential erst bei größeren Teilchensystemen zum Tragen. Erst wenn durch genügend Arbeitslast in Form entsprechend vieler Workgroups die Fähigkeit der schnellen Kontextwechsel auf der GPU ausgenutzt wird, kann die Hardware ihre optimale Leistung erreichen.
Die Effizienzwerte der CPUs müssen unter dem Gesichtspunkt betrachtet werden, dass sie in Bezug auf die logischen Kerne des Systems bestimmt wurden. Werden beim Opteron die Module und beim Xeon Prozessor die physischen Kerne als Bezugsgröße verwendet, ergeben sich Effizienzen von 0,85 – 0,95.
Da die Relevanz einer Simulation mit großer Teilchenzahl wesentlich höher ist als die kleiner Systeme, ist ein Laufzeitvergleich für eine Partikelzahl von 131 072 Teilchen angegeben. Hier können die GPUs ihre massive Rechenleistung ausspielen und sind um ein Zehnfaches schneller als die Lösung auf dem Opteronsystem und sogar 45-mal schneller als die Xeon CPUs. Die aufgeführten GFlop/s zeigen, dass der OpenCL Kernel auf der GPU wesentlich effizienter läuft als auf einer CPU. Diese Werte wurden in einem Simulationslauf mit 131 072 Teilchen gemessen und stellen für die CPUs bereits die obere Schranke dar. Die GPUs können bei noch größeren Teilchensystemen (> 500 000 Partikel) sogar 2250 GFlop/s erreichen, was 54 % der theoretischen Peak Performance der Tesla S2050 entspricht, welche 4130 GFlop/s beträgt. ([NVI10], S. 1)
Es ist festzustellen, dass der Kommunikationsanteil bei größeren Partikelzahlen aufgrund des quadratischen Rechenaufwands immer geringer wird und so der Datenaustausch zwischen den GPUs die Gesamtleistung des Programms weniger beeinflusst.
Die genannten Teilchenzahlen, welche innerhalb von 31 Tagen simuliert werden können, beruhen auf der Annahme, dass eine Simulation 15 000 Zeitschritte benötigt, bis sich die Teilchen in Abhängigkeit ihrer gegenseitigen Beeinflussung positioniert haben und in einem Ruhezustand befinden. Hier wird ersichtlich, dass auch mit CPUs durchaus große Systeme simuliert werden können.

5 Zusammenfassung und Ausblick

Das Ziel der Arbeit war es, ein Programm zur Simulation stark gekoppelter Plasmen zu entwickeln und dabei die Berechnung auf mehreren GPUs parallel auszuführen. Dafür wurden verschiedene Parallelisierungsalgorithmen untersucht sowie deren Implementierung erläutert.
Das Programm ist so erstellt, dass die Integrationsmethode angepasst werden kann. Dabei bleibt die Parallelisierung für den Integrator transparent. Das Programm stellt Schnittstellen für diesen Integrator zur Verfügung und verteilt die Arbeit im Hintergrund auf die vorhandenen Geräte. Für diese Anwendung wurde der Velocity-Verlet-Integrator implementiert.
Bei der Untersuchung der verschiedenen Parallelisierungstrategien hat sich herausgestellt, dass die Methode des systolischen Rings nicht für die Kommunikation von GPUs innerhalb eines Knotens geeignet ist. Dies liegt vor allem an den hohen Speicheranforderungen dieses Algorithmus. Durch das Verfahren der lokalen Reduktion konnten diese Anforderungen zwar vermindert werden, jedoch enthält auch diese Implementierung eine hohe Speicherbindung. Außerdem zeigte sich, dass die Strategie des systolischen Rings einen höhreren Kommunikationsanteil im Vergleich zum Verfahren der redundanten Datenhaltung besitzt. Mit letzterem Ansatz erreicht man bei der Parallelisierung über vier GPUs eine parallele Effizienz von 95 %.
Abschließend ist festzustellen, dass sich speicherintensive Berechnungen auf GPUs nicht mit ausreichender Effizienz durchführen lassen. Deshalb ist es beim Einsatz von GPUs wichtiger als auf jeder anderen Architektur, doppelte Berechnungen in Betracht zu ziehen, selbst wenn diese komplex sind.

Da das Programm für die Berechnungen OpenCL nutzt, können diese sowohl auf GPUs als auch auf CPUs ausgeführt werden. Die Analyse des Programmverhaltens auf CPUs hat gezeigt, dass das Problem unter Berücksichtigung der hardwarespezifischen Besonderheiten der getesteten CPUs gut skaliert. Allerdings wird mit dem existierenden Kernel die mögliche Leistung der CPUs nicht so gut ausgeschöpft wie die der GPUs. Durch die Implementierung eines für die CPU optimierten Kernels ließe sich der rechenintensive Code noch beschleunigen.
Der Einsatz von GPUs als Hardwarebeschleuniger hat sich als sehr positiv erwiesen. Aufgrund der quadratischen Komplexität des Problems ist das Programm sehr rechenintensiv, was der durchsatzorientierten Architektur einer GPU entgegenkommt.

Ein möglicher Ansatzpunkt zur weiteren Verbesserung des Programms wäre die Ausnutzung der heterogenen Hardwareumgebung. Bisher werden entweder GPUs oder CPUs zur Berechnung genutzt. Da heterogene Rechensysteme heutzutage weit verbreitet sind und OpenCL auch dafür ausgelegt wurde, Anwendungen für solche heterogenen Umgebungen zu erstellen, wäre eine gemeinsame Nutzung von GPUs und CPUs denkbar. Dafür müsste die bisherige statische Lastverteilung überdacht werden, weil die verwendeten OpenCL Geräte dann unterschiedliche Rechenkapazitäten besitzen würden und sich deshalb stark in der Ausführungsgeschwindigkeit unterscheiden würden.

Da bei N-Körper-Simulationen oftmals noch wesentlich größere Systeme als in dieser Arbeit betrachtet werden sollen, wäre die Erweiterung des Programms zu einer Multi-Node-Lösung ebenfalls eine interessante Option. Bisherige Codes, die sich mit N-Körper-Simulationen beschäftigen, nutzen beispielsweise Nachbarschemata und verringern so die Menge der zu übermittelnden Daten zwischen den Rechenknoten. Da `SCPonGPUcl` keine solchen Approximationen vornimmt, wäre es interessant zu untersuchen, ob das Programm über viele Knoten hinweg auch noch gut skaliert. Dafür müsste in einer zukünftigen Arbeit das Programm beispielsweise mit Hilfe von MPI erweitert werden.

Literaturverzeichnis

[Aar03] AARSETH, S.J.: *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge University Press, 2003 (Cambridge Monographs on Mathematical Physics). – ISBN 9780521432726

[Adv12a] ADVANCED MICRO DEVICES: *Accelerated Processing OpenCL Programming Guide*. 2012

[Adv12b] ADVANCED MICRO DEVICES: *Software Optimization Guide for AMD Family 15h Processors*. 2012 (47414)

[Dia96] DIACU, Florin: The solution of the n-body problem. In: *The Mathematical Intelligencer* 18 (1996), S. 66–70. – 10.1007/BF03024313. – ISSN 0343–6993

[Hoe60] VON HOERNER, S.: *Die numerische Integration des n-Körper-Problems für Sternhaufen*. 1960

[Int12] INTEL CORPORATION: *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2012 (248966-026)

[Khr10] KHRONOS: *OpenCL 1.1 Specification*. 2010

[Mar85] MARCINIAK, A.: *Numerical solutions of the N-body problem*. D. Reidel, 1985 (Mathematics and its applications (D. Reidel Publishing Company).: East European series). – ISBN 9789027720580

[NA12] NITADORI, Keigo ; AARSETH, Sverre J.: Accelerating NBODY6 with Graphics Processing Units. (2012), Mai

[NVI09] NVIDIA: *Fermi Compute Architecture Whitepaper*. 2009

[NVI10] NVIDIA CORPORATION: Tesla S2050 GPU Computing System. (2010), October

[NVI12] NVIDIA: *NVIDIA CUDA C Programming Guide*. 2012

[Pap06] PAPKE, T.: *Integrationsverfahren für molekulardynamische Simulationen: Untersuchungen zur Effizienz und Genauigkeit*. FZJ-ZAM, 2006 (Technical Report - Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich)

[SBB+11] SPURZEM, R. ; BERCZIK, P. ; BERENTZEN, I. ; NITADORI, K. ; HAMADA, T. ; MARCUS, G. ; KUGEL, A. ; MÄNNER, R. ; FIESTAS, J. ; BANERJEE, R. ; KLESSEN, R.: Astrophysical particle simulations with large custom GPU clusters on three continents. In: *Computer Science - Research and Development* 26 (2011), S. 145–151. – ISSN 1865–2034

[SO11] STUART, Jeff A. ; OWENS, John D.: Efficient Synchronization Primitives for GPUs. In: *CoRR* abs/1110.4623 (2011)

[Sun12] SUNDMAN, Karl: Mémoire sur le probléme des trois corps. In: *Acta Mathematica* 36 (1912), S. 105–179

[Sut02] SUTMANN, Godehard: Classical Molecular Dynamics. In: *NIC Series. Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms* 10 (2002), S. 211–254

[Wan91] WANG, Q.-D.: The global solution of the n-body problem. In: *Celestial Mechanics and Dynamical Astronomy* 50 (1991), S. 73–88

Abbildungsverzeichnis

Tabellenverzeichnis

A Programmauszüge

A.1 Blockberechnung aus OpenCL Kernel

```
#define REAL3 float3
#define REAL4 float4

// Mod without divide, works on values from 0 up to 2m
#define WRAP(x,m) (((x)<m)?(x):(x-m))

// computes coulomb force for particle 'bodyPos'
REAL3 computeCoulomb(REAL4 bodyPos,
                           __global REAL4* positions,
                           int numBodies,
                           __local REAL4* sharedPos)
{

    REAL3 acc = ZERO3;

    unsigned int threadIdxx = get_local_id(0);
    unsigned int threadIdxy = get_local_id(1);
    unsigned int blockIdxx = get_group_id(0);
    unsigned int blockIdxy = get_group_id(1);
    unsigned int gridDimx = get_num_groups(0);
    unsigned int blockDimx = get_local_size(0);
    unsigned int blockDimy = get_local_size(1);
    unsigned int numTiles = numBodies / (blockDimx * blockDimy);

    for (unsigned int tile = blockIdxy; tile < numTiles + blockIdxy; tile++)
    {
        sharedPos[threadIdxx + (blockDimx * threadIdxy)] =
            positions[WRAP(blockIdxx + tile, numTiles) * blockDimx + threadIdxx];

        barrier(CLK_LOCAL_MEM_FENCE);// __syncthreads();

        // walks through the block of sharedPos and accumulates the forces
        acc =  coulombForceTile(bodyPos, acc, sharedPos);

        barrier(CLK_LOCAL_MEM_FENCE);// __syncthreads();
    }

    return acc;
}
```

Danksagung

Ein großes Dankeschön geht an meine beiden Betreuer Dipl.-Ing. Guido Juckeland und Dr. Michael Bussmann, welche stets bereit waren, mir geduldig zu helfen.
Bei Herrn Prof. Nagel möchte ich mich dafür bedanken, dass er die Betreuung dieser Arbeit übernommen hat.
Außerdem möchte ich Richard Pausch danken, da die Voraussetzungen für diese Arbeit aus einer vorangegangenen Zusammenarbeit mit ihm entstanden sind.
Ebenso möchte ich meinen Eltern danken, die mich während der Arbeit immer unterstützt haben und mir als Korrekturleser eine große Hilfe waren.
Anne Schumacher danke ich herzlich für ihre Bereitschaft, meine Arbeit ebenfalls Korrektur zu lesen, wodurch viele weitere Verbesserungen möglich wurden.

Erklärungen zum Urheberrecht

Das erstellte Programm SCPonGPUcl enthält Source Code, welcher durch die NVIDIA Corporation bereitgestellt wurde.