FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Einsatz von FPGAs für molekulardynamische Rechnungen

Annika Schiller

FZJ-ZAM-IB-2007-06

März 2007

(letzte Änderung: 27.03.2007)

Inhaltsverzeichnis

Abbildungsverzeichnis

Abstract

Einsatz von FPGAs für molekulardynamische Rechnungen

Die Simulation komplexer Vielteilchensysteme in vielen Bereichen der Wissenschaft wird immer wichtiger bei der Vorhersage und Analyse physikalischer Phänomene. Um möglichst große Systeme über einen längeren Zeitraum hinweg simulieren zu können, werden neben der Entwicklung spezieller Algorithmen verstärkt auch spezialisierte Computerarchitekturen eingesetzt. Mittels solcher Architekturen lassen sich bestimmte Operationen extrem schnell berechnen. Durch ihre fest codierte Hardware sind sie jedoch nicht für jede Problemklasse effizient einsetzbar. Eine Alternative bieten FPGA-basierte Computerarchitekturen. Diese lassen sich jederzeit umkonfigurieren und können somit auf verschiedene Problemklassen optimiert werden

In Rahmen dieser Diplomarbeit wird das FPGA-Board PROGRAPE-4 für molekulardynamische Berechnungen eingesetzt. Zur Konfigurierung des Boards wird die Software PGR verwendet. Es wird zunächst ein FPGA-Design zur Kräfteberechnung mittels des Lennard-Jones-Potentials entwickelt und auf ein offenes System angewendet. Dadurch lässt sich ein erheblicher Leistungsgewinn erzielen. Dieses Problem hat eine Komplexität von $O(N^2)$, da für die Berechnung der Kräfte alle Teilchen des Systems mit einbezogen werden. Der Linked-Cell-List-Algorithmus optimiert die Wechselwirkungsberechnung für das kurzreichweitige Lennard-Jones-Potential auf eine Komplexität von $O(N)$. Um zu untersuchen, ob sich das PROGRAPE-4-Board auch für diesen Algorithmus effizient einsetzen lässt, wird hierzu ein passendes FPGA-Design entwickelt.

Use of FPGAs for Molecular Dynamics Calculations

The simulation of complex many-body systems becomes increasingly important for the prognosis and analysis of physical phenomena in many fields of science. Further to the development of special algorithms, the use of specialized computer architectures to simulate larger systems over a longer period of time gained momentum. Using such specialized computer architectures, certain operations can be computed extremly fast. However, they are not usable for all classes of problems equally efficient, because of their hard coded logic. FPGA based computer architectures are an alternative to these highly specialized systems. They can be reconfigured at any time and optimized for different classes of problems.

In this diploma thesis, the FPGA board PROGRAPE-4 is used for molecular dynamics calculations. To configure the board, the software PGR is used. At first, an FPGA design is developed to calculate the forces in an open system via a Lennard-Jones potential. Thereby the performance can be increased significantly. Due to the fact that for the calculation of the forces all particles in the system have to be considered, this problem has a complexity of $O(N^2)$. The Linked-Cell-List algorithm optimizes the calculation of the forces for the short ranged Lennard-Jones potential to a complexity of $O(N)$. To determine if the PROGRAPE-4 board can be used for this algorithm with similar efficiency, a suitable FPGA design is developed.

Kapitel 1

Einleitung

Die Simulation komplexer Vielteilchensysteme in vielen Bereichen der Wissenschaft wird immer wichtiger bei der Vorhersage und Analyse physikalischer Phänomene. Computersimulationen benötigen jedoch einen enormen Bedarf an Rechenleistung. Um möglichst große Systeme über einen längeren Zeitraum hinweg simulieren zu können, ist die effiziente Implementierung der Programme, sowie die Entwicklung spezieller Algorithmen von entscheidender Bedeutung.

Der Einsatz leistungsfähiger Computersysteme, wie z. B. Supercomputer, trägt ebenfalls zur Beschleunigung der Computersimulation bei. In diesem Zusammenhang werden verstärkt auch spezialisierte Computerarchitekturen benutzt. Mittels solcher Architekturen lassen sich bestimmte Operationen extrem schnell berechnen. Sie rechnen zudem massiv parallel. Durch ihre fest codierte Hardware sind sie jedoch nicht auf jede Problemklasse optimiert.

Eine Alternative zu spezialisierten Computerarchitekturen bietet der Einsatz rekonfigurierbarer Hardware, wie FPGAs (***F**ield **P**rogrammable **G**ate **A**rray*). Ein FPGA besteht aus einzelnen Funktionsblöcken, die in einer Matrix angeordnet sind und über ein Verbindungsnetzwerk miteinander verbunden sind. Durch Programmierung der Funktionsblöcke lässt sich mit einem FPGA, im Rahmen der verfügbaren Logik, eine große Menge digitalelektronischer Schaltungen realisieren. FPGA-basierte Computerarchitekturen lassen sich jederzeit umkonfigurieren und können somit auf unterschiedliche Probleme optimiert werden.

Im Rahmen dieser Arbeit sollen molekulardynamische Rechnungen auf FPGAs durchgeführt werden. Als Motivation dazu dient eine Projektarbeit, in der ein FPGA-Board zur Berechnung von Gravitationskräften eingesetzt wurde. Die Berechnung konnte um einen Faktor 35 beschleunigt werden. Das Gravitationspotential ist ein langreichweitiges Potential. Es soll untersucht werden, ob sich das FPGA-Board auch für kurzreichweitige Wechselwirkungspotentiale, wie dem Lennard-Jones-Potential, einsetzen lässt. Ein besonderes Augenmerk wird dabei auf den Linked-Cell-List-Algorithmus gelegt. Mit diesem Algorithmus wird mittels Nachbarschaftslisten die Komplexität der Kräfteberechnung von $O(N^2)$ auf $O(N)$ reduziert.

Für den Linked-Cell-List-Algorithmus soll ein FPGA-Design entwickelt werden. Als Hardware steht dazu das PROGRAPE-4-Board am Astronomischen Recheninstitut (ARI) in Heidelberg zur Verfügung. Das Board wird mit der Software PGR (***P**rocessors **G**enerator for **R**econfigurable Systems*) programmiert. Zur Entwicklung eines solchen FPGA-Designs muss ein Hardwaredesign entworfen werden und die Softwareimplementierung des Linked-Cell-List-Algorithmus dementsprechend umgeschrieben werden.

Im folgenden Kapitel werden zunächst einige Grundlagen zu den Methoden der Molekulardynamik (MD) und zur Vorgehensweise einer MD-Simulation erläutert. Kapitel 3 liefert eine Einführung in die Thematik der rekonfigurierbaren Hardware. Hierzu wird auf die Architektur von FPGAs und deren Programmierung eingegangen. Als ein spezielles FPGA-Board wird das PROGRAPE-4-Board und dessen Architektur vorgestellt. In Kapitel 4 wird die Software PGR und die dazugehörige Beschreibungssprache PGDL zur Programmierung des PROGRAPE-4-Boards beschrieben.

Die Kapitel 5 und 6 beschäftigen sich mit der Implementierung des FPGA-Designs. Als Vorarbeit wird zunächst ein einfaches Lennard-Jones-Potential zur Kräfteberechnung implementiert. Ein einfaches Testprogramm untersucht die Genauigkeit und bestimmt den Speedup des $O(N^2)$-Problems. Schließlich wird das Hardwaredesign zum Linked-Cell-List-Algorithmus entwickelt und die Softwareimplementierung angepasst. Abschließend werden Zeitmessungen durchgeführt und die Ergebnisse diskutiert.

Kapitel 2

Grundlagen der Molekulardynamik

Die klassische Molekulardynamik beschäftigt sich mit der Berechnung von Wechselwirkungen zwischen verschiedenen Teilchen und mit der Integration der klassischen Bewegungsgleichungen. Diese Teilchen können unterschiedliche Formen und Eigenschaften haben. Es können beispielsweise Moleküle und Atome, aber auch Schüttgüter in Silos, wie z. B. Getreide oder Gestein, das aus einzelnen Kieseln zusammengesetzt ist, betrachtet werden. Dabei ist man an verschiedenen Faktoren interessiert, wie z. B. der Größe der Energie im simulierten System oder an den Kräften, die auf die einzelnen Teilchen wirken.

Die Methoden der Molekulardynamik sind vielseitig einsetzbar. Sie finden Anwendung in der physikalischen Chemie, Physik, Biophysik und Astrophysik. Mit Hilfe der Molekulardynamik lassen sich beispielsweise Eigenschaften von Flüssigkeiten, Rissbildung und Verformung von Festkörpern oder Reibung simulieren [24]. Aber auch bei der DNA-Analyse oder der Berechnung der Dynamik von Galaxien spielen die Methoden der Molekulardynamik eine Rolle [19].

Dieses Kapitel liefert eine kurze Einführung in die Methoden der Molekulardynamik. Es werden zunächst die wichtigsten physikalischen und mathematischen Grundlagen beschrieben und einige Teilchenmodelle vorgestellt. Anschließend folgt ein kurzer Einblick in Listen-Techniken zur effizienteren Berechnung von kurzreichweitigen Wechselwirkungen. Des Weiteren wird der Aufbau einer MD-Simulation erläutert. Neben den Vorteilen einer Simulation werden auch die Probleme diskutiert. Abschließend werden Methoden zur Beschleunigung einer MD-Simulation vorgestellt, wobei FPGAs als Hardwarebeschleuniger eine Möglichkeit sein werden.

2.1 Physikalische und mathematische Grundlagen

Die Grundlage für die Modellierung eines physikalischen Vielteilchensystems im Sinne der klassischen Molekulardynamik bilden die klassischen Bewegungsgleichungen (zumeist auch Newtonsche Bewegungsgleichungen genannt, weil sie sich zum überwiegenden Teil aus den drei Newtonschen Axiomen ableiten). Sie beschreiben die Bewegung der Teilchen unter der Einwirkung der Kräfte, die auf jedes Teilchen ausgeübt werden. Diese Kräfte entstehen beispielsweise durch Wechselwirkungen mit anderen Teilchen, durch die Wände des Systems oder durch äußere Felder, wie etwa das Gravitationsfeld oder elektromagnetische Felder. Durch die Bewegungsgleichungen sind,

bei Angabe von Anfangsbedingungen, die Bahnen der Teilchen genau bestimmt. In der Molekulardynamik ist es üblich, statt der klassischen Bewegungsgleichungen die äquivalente Hamiltonsche Formulierung der Bewegungsgleichungen zu verwenden:

$$\dot{\vec{r}}_i = \frac{d\mathcal{H}}{d\vec{p}_i} \quad , \quad \dot{\vec{p}}_i = -\frac{d\mathcal{H}}{d\vec{r}_i} \qquad i = 1, ..., N \tag{2.1}$$

$\vec{r}_i$: Koordinaten des Teilchens i im Raum
$\vec{p}_i$: Impuls des Teilchens i
N : Anzahl der Teilchen im System
$\mathcal{H}$: Hamilton-Funktion

Die Hamiltonschen Bewegungsgleichungen stammen aus der *Hamiltonschen Mechanik*. Sie bilden ein System von Differentialgleichungen erster Ordnung. Die Hamilton-Funktion $\mathcal{H}$, auch Hamiltonian genannt, beschreibt dabei die gesamte mechanische Energie im System:

$$\mathcal{H} = \underbrace{\frac{1}{2}\sum_{i=1}^{N}\frac{\vec{p}_i^{\,2}}{m_i}}_{\text{kinetische Energie}} + \underbrace{\sum_{i=1}^{N-1}\sum_{j>i}^{N} U_{ij}(r_{ij})}_{\text{potentielle Energie}} + U_{ext} \tag{2.2}$$

N : Anzahl der Teilchen im System
$\vec{p}_i$: Impuls des Teilchens i
m_i : Masse des Teilchens i
U_{ij} : Wechselwirkungspotential zwischen zwei Teilchen i und j
r_{ij} : Abstand zwischen Teilchen i und j ($\|\vec{r}_i - \vec{r}_j\|_2$)
U_{ext} : Einfluss externer Potentiale

U_{ext} bezeichnet hier den Einfluss externer Kräfte. Solche Kräfte existieren bei nichtabgeschlossenen Teilchensystemen, d. h. bei Systemen, denen während der Simulation von außen Energie zugeführt wird. Für abgeschlossene Systeme gilt: $U_{ext} = 0$.

Um nun ein physikalisches Teilchensystem simulieren zu können, muss das durch Gleichung 2.1 festgelegte Differentialgleichungssystem gelöst werden. Dies ist jedoch für ein System von drei und mehr Teilchen nicht mehr exakt lösbar. Deshalb muss die Lösung durch ein geeignetes numerisches Verfahren angenähert werden. Ein solches Verfahren wird auch als Integrator bezeichnet. Ein Integrator ist also ein numerisches, auf dem Rechner implementierbares Verfahren, das Systeme von Differentialgleichungen näherungsweise lösen kann [21]. Dabei sollte ein Integrator gewählt werden, der das System möglichst genau löst, denn die Qualität einer Simulation basiert im Wesentlichen auf der Effizienz und Genauigkeit der verwendeten Methode. In der Molekulardynamik berechnet der Integrator die Veränderung der Position und Geschwindigkeit eines Teilchens, die sich in einem Zeitschritt ergibt. Die Grundlage der Berechnung bilden die Position, Geschwindigkeit und Kraft dieses Teilchens zum aktuellen Zeitpunkt. Beispiele für Integratoren sind der Euler- und der Verlet-Algorithmus [11].

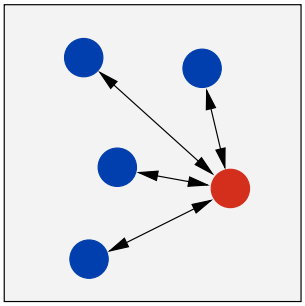


Abbildung 2.1: Wechselwirkungen in einem System mit offenen Rändern

2.2 System- und Teilchenmodelle

Durch den Einsatz der Molekulardynamik für die unterschiedlichsten Probleme ergeben sich verschiedene Modelle. Es ist leicht einzusehen, dass die Sterne einer Galaxie nicht mit dem gleichen Modell simuliert werden können, wie die Atome in einem Salzkristall. Bei der Simulation einer Galaxie sind die Wechselwirkungen mit Sternen einer anderen Galaxie meist so klein, dass man sie vernachlässigen kann. Es genügt also, nur die Sterne der eigenen Galaxie zu betrachten und weitere Einflüsse höchstens durch einen Korrekturterm zu berücksichtigen. Die Atome in einem Salzkristall dagegen sind gitterförmig angeordnet. Hierbei ist es ausreichend, lediglich einen Ausschnitt des Kristalls zu betrachten und diesen periodisch in alle Richtungen fortzusetzen [19].

2.2.1 Periodische und nicht-periodische Systeme

Aus dem oben beschriebenen Beispiel ergibt sich direkt eine erste grobe Einteilung der Modelle in periodische und nicht-periodische Systeme. Im Fall der Simulation von Sternen einer Galaxie ist es sinnvoll diese durch ein offenes, nicht-periodisches System zu simulieren. Der Salzkristall dagegen lässt sich durch ein periodisch fortgesetztes System darstellen.

Abbildung 2.1 zeigt schematisch die Wechselwirkungen in einem offenen System. Das simulierte System wird hier durch eine quadratische Box markiert und im folgenden Simulationsbox genannt. Bei einem offenen System werden zur Berechnung der Kraft, die auf ein Teilchen einwirkt, die Einflüsse aller anderen im System vorhandenen Teilchen mit einbezogen.

Bei Systemen mit periodischen Randbedingungen betrachtet man ebenfalls eine Simulationsbox, in der die Simulation stattfindet. Um Randeffekte zu vermeiden (Teilchen am Rand eines Körpers oder Stoffes verhalten sich anders als Teilchen innerhalb) und unendlich große Systeme simulieren zu können, werden Bildboxen um die eigentliche Simulationsbox gelegt. Im zweidimensionalen Fall wären es acht Bildboxen, im dreidimensionalen Fall entsprechend 26. Diese Bildboxen sind exakte Kopien der Originalbox. Dadurch bleibt die Anzahl der Teilchen im simulierten System konstant, denn tritt ein Teilchen während der Simulation aus der Originalbox aus, erscheint auf der anderen Seite der Box sofort dessen Bildteilchen (siehe Abbildung 2.2). Zur Berechnung der Kraft, die auf ein Teilchen einwirkt, werden nur die Einflüsse der nächsten Nachbarn berücksichtigt. Nächste Nachbarn eines Teilchens können in einem periodischen System aber auch Teilchen aus den Bildboxen sein. Deshalb wird eine gedachte Box mit der Kantenlänge der Simulationsbox um das Teilchen gelegt (siehe Abbildung 2.3). Nun lässt man alle Teilchen innerhalb dieser gedachten Box mit diesem Teilchen wechselwirken. Dieses Prinzip wird auch *Minimum Image Convention* genannt [24].

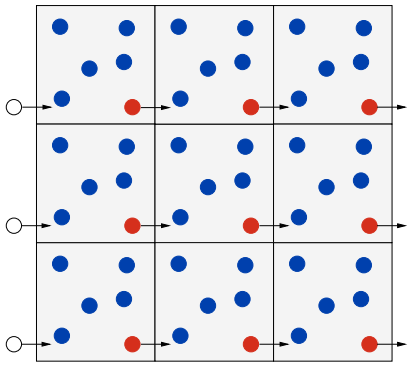


Abbildung 2.2: System mit periodischen Randbedingungen

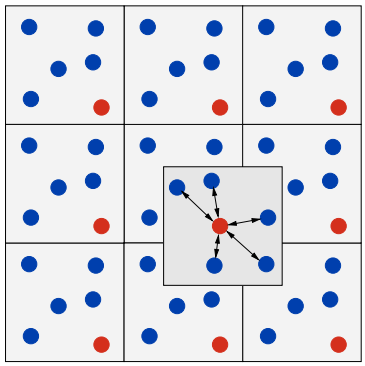


Abbildung 2.3: Wechselwirkungen in einem System mit periodischen Randbedingungen nach der *Minimum Image Convention*

Es können jedoch auch Mischformen von periodischen und offenen Systemen auftreten. Dies ist beispielsweise bei der Simulation eines Stoffes in einer Röhre sinnvoll. Dabei wird das Teilchensystem in eine Koordinatenrichtung periodisch fortgesetzt und in den anderen beiden Richtungen begrenzt.

2.2.2 Wechselwirkungspotentiale

Wie schon erwähnt werden zur Beschreibung der Interaktionen zwischen den Teilchen eines Systems so genannte Wechselwirkungspotentiale verwendet. Interagierende Teilchen können dabei verschiedene Objekte sein, wie Atome und Moleküle, aber auch Sterne oder Galaxien. In der klassischen Simulation werden sie als punktförmige Massen beschrieben. Sie können paarweise oder zu mehreren miteinander wechselwirken, oder aber aus Anteilen von mehreren Atomen bestehen (z. B. bei Winkel- oder Torsionspotentialen). In dieser Arbeit werden jedoch nur paarweise Wechselwirkungspotentiale betrachtet.

Neben der Anzahl der wechselwirkenden Teilchen unterscheidet man auch zwischen der Reichweite von Potentialen. Dabei unterteilt man diese in kurzreichweitige und langreichweitige Wechselwirkungspotentiale. Fällt das Potential mit steigendem Teilchenabstand r schneller als r^{-d} gegen Null ab, wobei d die Dimension des Systems ist, so ist das Potential kurzreichweitig, anderenfalls langreichweitig. Betrachtet man das folgende Integral, wird klar, dass diese Regel sinnvoll ist:

$$I = \int \frac{1}{r^n}\, dr^d = \begin{cases} \infty & : n \leq d \quad \text{(langreichweitig)} \\ endlich & : n > d \quad \text{(kurzreichweitig)} \end{cases} \tag{2.3}$$

r : Abstand zwischen zwei Teilchen
d : Dimension des Systems

Gilt also $n \leq d$, so berechnet sich die potentielle Energie eines Teilchens unter Berücksichtigung aller anderen Teilchen des Universums. Gilt dagegen $n > d$, sind die Einflüsse der anderen Teilchen im wesentlichen auf eine Region begrenzt. Für weiter außen liegende Teilchen kann dann ein Korrekturterm berücksichtigt werden [25].

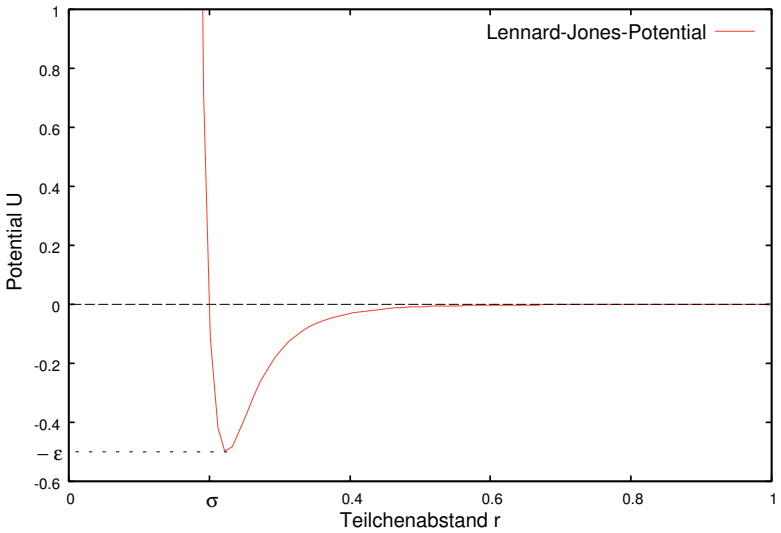


Abbildung 2.4: Lennard-Jones-Potential

Kurzreichweitige Wechselwirkungen

Kurzreichweitige Wechselwirkungen treten z. B. bei atomaren Abstoßungen auf. Sie ermöglichen es, bei der Berechnung der auf ein Teilchen einwirkenden Kraft lediglich benachbarte Teilchen bis zu einer bestimmten Entfernung zu berücksichtigen. Der Grund dafür ist, dass kurzreichweitige Potentiale mit wachsendem Teilchenabstand sehr schnell gegen Null konvergieren und deshalb weit entfernte Teilchen kaum mehr ins Gewicht fallen. Das Abschneiden des Potentials wird durch die Einführung eines so genannten Cutoff-Radius realisiert. Teilchen, die sich jenseits dieses Radius befinden, werden nicht mehr berücksichtigt, d. h. für alle Abstände $r > r_{cut}$ gilt: $U(r) = 0$. Durch das Abschneiden des Potentials entsteht jedoch eine Diskontinuität an der Kante, d. h. die Kraft wird an dieser Stelle unendlich groß. Für die MD-Simulation stellt dies ein Problem dar, da die Energieerhaltung nicht mehr erfüllt ist. Aus diesem Grund verwendet man in der Molekulardynamik das so genannte Shifted-force Potential (Gleichung 2.4), wodurch sowohl das Potential als auch die Kraft stetig über den Cutoff-Radius auf Null abfällt [24].

$$U(r) = \begin{cases} U(r) - U(r_{cut}) - \dfrac{dU}{dr}(r - r_{cut}) & , \; r \leq r_{cut} \\ 0 & , \; r > r_{cut} \end{cases} \tag{2.4}$$

Ein Beispiel für ein kurzreichweitiges Wechselwirkungspotential ist das Lennard-Jones-Potential:

$$U_{LJ}(r) = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right) \tag{2.5}$$

r : Abstand zwischen zwei Teilchen
ε : Energiekonstante
σ : Durchmesser eines Teilchens

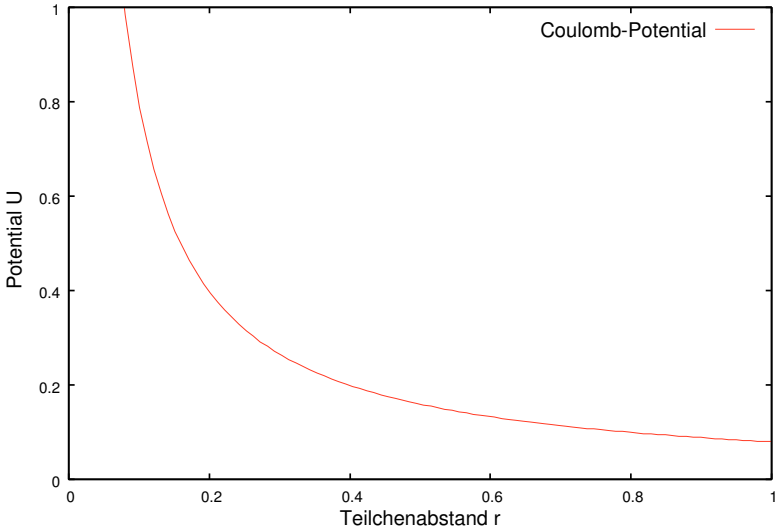


Abbildung 2.5: Coulomb-Potential für Teilchen mit gleicher Ladung

Langreichweitige Wechselwirkungen

Langreichweitige Wechselwirkungen treten beispielsweise bei elektrostatischen Interaktionen zwischen Punktladungen oder Dipolen und bei Gravitation auf. Bei der Berechnung solcher Wechselwirkungen ist es nicht mehr möglich, weit entfernte Teilchen zu vernachlässigen. Für die genaue Bestimmung der Kräfte müssen die Einwirkungen aller Teilchen im simulierten System berücksichtigt werden. Im Simulationsprogramm wird dies durch eine Doppelschleife über alle N Teilchen realisiert. Die Komplexität des Problems ist deshalb $O(N^2)$, was besonders bei großen Teilchenzahlen die Rechenzeit dramatisch verlängert. Durch geeignete Algorithmen, wie z. B. Baum-Algorithmen lässt sich die Komplexität jedoch auf $O(N\ log\ N)$ oder sogar $O(N)$ verbessern.

Ein Beispiel für ein langreichweitiges Wechselwirkungspotential ist das Coulomb-Potential:

$$U_{Coulomb}(r) = \frac{q_1\, q_2}{4\,\pi\,\varepsilon_0\, r} \tag{2.6}$$

r	:	Abstand zwischen zwei Teilchen
q_1, q_2	:	Ladungen der zwei betrachteten Teilchen
ε_0	:	Dielektrizitätskonstante des Vakuums

Durch die Festlegung des Potentials und dessen Parameter lässt sich die komplette Energie des simulierten Systems berechnen. Durch Ableiten der Potentialfunktion, also $\vec{F} = -\nabla U$, erhält man außerdem die Kraft, die auf jedes Teilchen im System einwirkt.

2.3 Nachbarschaftslisten

Von besonderem Interesse für diese Arbeit sind die kurzreichweitigen Wechselwirkungen. Wie schon erwähnt werden hierbei nur lokale Informationen benötigt. Lokale Teilchen sind dabei diejenigen, die sich innerhalb des Cutoff-Radius des betrachteten Teilchens befinden. Das Simulationsprogramm weiß nun aber nicht, welche Teilchen j lokal zu einem Teilchen i sind. Es muss für jedes

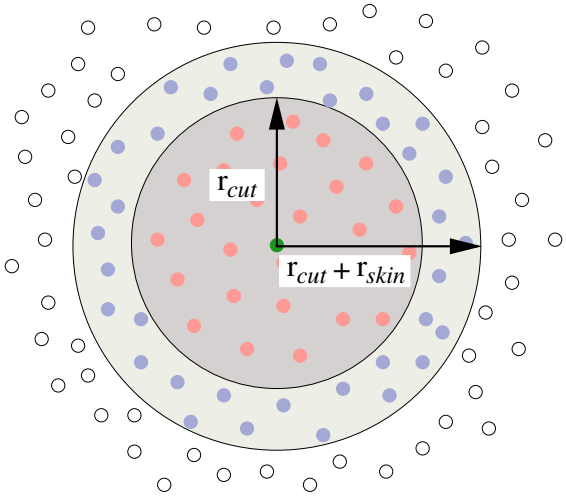


Abbildung 2.6: Prinzip der Verlet-Listen

Teilchen j überprüft werden, ob dessen Abstand zu Teilchen i kleiner oder gleich dem Cutoff-Radius ist. Dies wird durch eine Doppelschleife realisiert, womit die Komplexität der Kräfteberechnung immer noch bei $O(N^2)$ liegt.

Die Lösung für dieses Problem sind Nachbarschaftslisten. In diesen Listen werden für jedes Teilchen die Nachbarteilchen abgespeichert, die sich innerhalb des Cutoff-Radius befinden. Man unterscheidet dabei zwischen zwei Varianten: Verlet-Listen und Linked-Cell-Listen.

Verlet-Listen

Bei der Verlet-Listen-Technik werden Listen angelegt, in denen für alle N Teilchen des Systems die Teilchen gespeichert werden, deren Abstand kleiner oder gleich dem Cutoff-Radius r_{cut} ist. Außerdem wird für jedes Teilchen ein „Reservoir“ angelegt. Dieses Reservoir ist eine Kugelschale mit der Dicke r_{skin} und dem Innenradius r_{cut} (siehe Abbildung 2.6). Es enthält alle Teilchen, deren Abstand zum aktuellen Teilchen zwischen r_{cut} und $r_{cut} + r_{skin}$ liegt. Es sorgt dafür, dass keine Teilchen „unbemerkt“ Nachbarn werden können. Die im Reservoir befindlichen Teilchen können im aktuellen Simulationsschritt nicht mit dem entsprechenden Teilchen wechselwirken und werden bei der Berechnung nicht berücksichtigt. Sie können sich jedoch im nächsten Simulationsschritt innerhalb des Cutoff-Radius befinden. Die Verlet-Liste wird immer dann vollständig aktualisiert, wenn ein Teilchen eine größere Distanz als r_{skin} zurücklegt. Die Größe von r_{skin} bestimmt dabei, wie oft die Liste aktualisiert wird. Für kleines r_{skin} wird die Liste häufig aktualisiert. Es sind jedoch nur wenige Abfragen in der Kraftroutine nötig, da sich wenige Teilchen im Reservoir befinden. Bei größerem r_{skin} steigt die Anzahl der Abfragen, die Liste muss jedoch seltener aktualisiert werden. Es gilt also ein r_{skin} zu finden, das die Listen-Updates und Abfragen optimiert [27]. Die Aktualisierung der kompletten Liste ist aufwändig, denn um die Wechselwirkungspartner zu identifizieren und in die Listen für alle Teilchen einzusortieren, müssen jedes Mal $N(N-1)/2$ Abfragen gemacht werden [17]. Zudem steigt der Speicheraufwand für die Verlet-Liste beinahe wie $O(N^2)$. Die Verlet-Listen-Technik ist deshalb für große Teilchenzahlen ungeeignet.

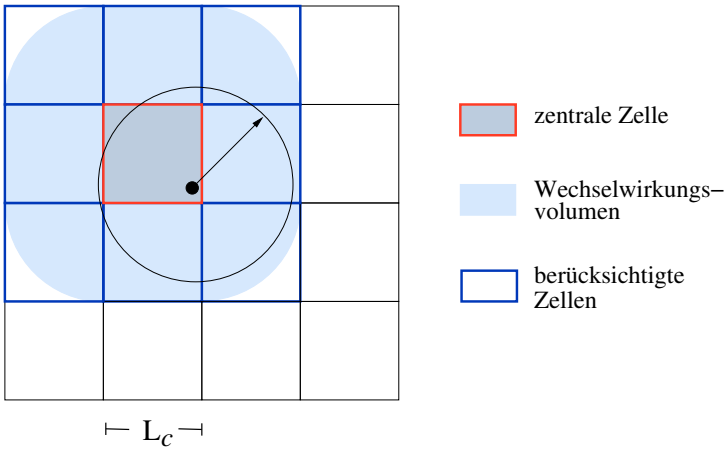


Abbildung 2.7: Prinzip der Linked-Cell-Listen

Linked-Cell-Listen

Bei der Linked-Cell-Listen-Technik wird die Simulationsbox in kubische Zellen eingeteilt. Die Teilchen werden anhand ihrer Positionen den einzelnen Zellen zugeordnet und in einer Liste je Zelle abgespeichert. In der einfachsten Art der Implementierung wird die Kantenlänge L_c der Zellen so gewählt, dass sie mindestens r_{cut} beträgt. Damit befinden sich alle Teilchen mit einem Abstand kleiner oder gleich dem Cutoff-Radius in der eigenen oder in einer der Nachbarzellen. Bei der Kraftberechnung werden dann nur die Teilchen berücksichtigt, die sich in der eigenen Zelle oder in einer der Nachbarzellen befinden. Der Vorteil dieser Methode ist, dass die Listen nicht aufwändig angelegt werden und nur lokale Operationen notwendig sind. Im Gegensatz zur Verlet-Listen-Technik brauchen hier nämlich nicht alle Teilchen des Systems betrachtet werden, sondern jeweils nur die Teilchen aus den entsprechenden Zellen. Sie kann auch für große Teilchenzahlen verwendet werden, da sie mit $O(N)$ skaliert [27].

Kombination von Verlet und Linked-Cell

Um die Effizienz der Verlet-Listen und die Lokalität der Linked-Cell-Listen auszunutzen, verwendet man häufig auch eine Kombination beider Methoden [26]. Die Berechnung der Wechselwirkungen geschieht dabei mit Hilfe der Verlet-Listen. Um das Problem der quadratischen Komplexität der Aktualisierung der Listen zu lösen, werden diese mittels Linked-Cell-Listen erneuert. Die Komplexität einer solchen Mischform ist wieder $O(N)$.

2.4 MD-Simulation

Die Computersimulation von Vielteilchensystemen hat in den letzten Jahren erheblich an Bedeutung gewonnen. Die theoretische Beschreibung komplexer Systeme und die experimentellen Techniken für detaillierte mikroskopische Informationen sind sehr weit entwickelt. Dennoch lassen sich bestimmte Aspekte solcher Systeme nur mittels Simulation in den gewünschten Einzelheiten betrachten [25].

Die traditionellen Simulationsmethoden für komplexe Vielteilchensysteme lassen sich grob in zwei Gebiete unterteilen: Molekulardynamik und „Monte Carlo“. Die Molekulardynamik basiert auf der Lösung der klassischen Bewegungsgleichungen aus der Mechanik, weshalb sie auch häufig klassische Molekulardynamik genannt wird. Bei dieser Methode werden ausschließlich Teilchensysteme betrachtet, deren Teilchen über ein gegebenes Potential miteinander wechselwirken. Dabei gewinnt man Informationen über die zeitliche Entwicklung des Systems. Die „Monte Carlo“-Methode dagegen beschäftigt sich mit der Bestimmung von Eigenschaften komplexer Systeme mit Hilfe von Zufallszahlen. Sie wird deshalb auch als Methode der statistischen Versuche bezeichnet. Die Zufallszahl-Experimente werden dabei als eine mögliche Realisierung des Systems interpretiert. Dazu müssen Kriterien definiert werden, mit denen unrealistische oder unsinnige Ergebnisse ausgeschlossen werden. Die Experimente werden hinreichend oft wiederholt und anschließend die Ergebnisse mit statistischen Methoden ausgewertet. Die „Monte Carlo“-Methode liefert keine expliziten Zeitinformationen, da sie stochastische Veränderungen der Teilchenpositionen im Konfigurationsraum erzeugt. Teilweise werden stochastisch erzeugte Trajektorien zeitlich interpretiert, wobei allerdings keine Zeitskala existiert.

Die Grundlage beider Simulationsmethoden ist immer das physikalische Modell, welches die Wechselwirkung zwischen den einzelnen Systemkomponenten beschreibt. Ein Teilchensystem, das simuliert werden soll, benötigt also eine möglichst vollständige physikalische Beschreibung. Die Computersimulation ist demnach nur insoweit korrekt, wie das physikalische Modell stimmt [21].

Dieser Arbeit liegen ausschließlich die Methoden der Molekulardynamik zugrunde. Deshalb sei die „Monte Carlo“-Methode an dieser Stelle nur der Vollständigkeit halber erwähnt. Alle weiteren Betrachtungen beziehen sich auf die Molekulardynamik.

2.4.1 Vorteile der Computersimulation

Die Computersimulation ermöglicht nicht nur eine detailliertere Betrachtung der Ergebnisse eines Experimentes. Sie bietet Möglichkeiten, die weit über die rein experimentelle Betrachtung hinaus gehen. So können mittels Computersimulation Experimente vollzogen werden, die in der Realität gar nicht oder nur sehr schwierig durchführbar sind. Dies ist beispielsweise der Fall, wenn das zu untersuchende Material mit dem Behälter reagiert oder während des Experimentes extrem hohe Temperaturen auftreten. Des Weiteren könnte es zu aufwändig oder zu teuer sein, die passende Umgebung für die Durchführung eines bestimmten Experimentes herzustellen [21]. Neben dem Kostenfaktor spielt aber auch die Sicherheit eine Rolle. So ist es sinnvoll, Experimente, deren Ausgang ungewiss ist, zunächst zu simulieren. Dadurch können mögliche Gefahrenquellen ausgemacht und Risiken abgeschätzt werden.

2.4.2 Durchführung einer MD-Simulation

Der Ablauf einer MD-Simulation ist dem eines realen Experimentes sehr ähnlich. Bei einem Experiment wird zunächst eine Probe des zu untersuchenden Stoffes vorbereitet. An diese Probe werden Messinstrumente, z. B. Thermometer, *Manometer* oder *Viskosimeter* angeschlossen und die entsprechende Größe über einen bestimmten Zeitraum hinweg gemessen. Da Messinstrumente üblicherweise nicht exakt messen, ergeben sich Messfehler. Damit diese Messfehler das Ergebnis des Experimentes nicht verfälschen, werden mehrere Messungen durchgeführt und anschließend die Mittelwerte der Resultate gebildet [10].

Nach dieser Vorgehensweise richtet sich auch ein typisches MD-Programm. Der Aufbau eines MD-Programms und damit der Ablauf einer MD-Simulation wird im Folgenden beschrieben:

1. **Parameter einlesen:** Zunächst werden die Parameter festgelegt und eingelesen, die das zu simulierende System spezifizieren. Dies sind Größen wie die Anzahl der Teilchen im System und deren Massen, Dichten, Dimensionen und Anfangstemperaturen oder auch die Anzahl und Größe der durchzuführenden Zeitschritte. Für eine MD-Simulation benötigt man darüberhinaus noch ein Wechselwirkungspotential, das beschreibt, wie die einzelnen Teilchen miteinander wechselwirken (siehe Kapitel 2.2.2), und einen Integrator, um die Bewegung der Teilchen im *Phasenraum* zu simulieren (siehe Kapitel 2.1).

2. **Initialisierung:** Um eine Simulation zu starten, benötigen die einzelnen Teilchen Anfangspositionen und -geschwindigkeiten. Die Positionen sollten entsprechend den Eigenschaften des simulierten Systems gewählt werden. Dabei ist darauf zu achten, dass die Teilchen sich nicht überschneiden. Bei der Initialisierung der Positionen gibt es verschiedene Möglichkeiten. Zum einen können Eingabedaten verwendet werden, die aus Experimenten stammen. Diese sind jedoch nicht immer vorhanden und es kann nicht ohne weiteres zu beliebigen anderen Teilchenzahlen übergegangen werden. Eine andere Möglichkeit ist, die Teilchen über einen Zufallszahlengenerator zu positionieren. Hierbei ist jedoch nicht sichergestellt, dass keine Teilchen überlappen. Die Teilchen werden deshalb häufig auf einem regelmäßigen Gitter angeordnet, z. B. einem Rechteckgitter oder einem *fcc-Gitter* (*face-centered-cube*-Gitter). Ein ähnliches Problem ergibt sich bei den Anfangsgeschwindigkeiten. Die Lösung hier sind Gauss-verteilte Zufallszahlen.
   Dieser konstruierte Anfangszustand ist zumeist nicht typisch für das simulierte System. Deshalb ist es notwendig das System zu equilibrieren. Dabei wird die Eigenschaft ausgenutzt, dass sich ein gestörtes System nach einiger Zeit wieder im thermodynamischen Gleichgewicht befindet. Im Simulationsprogramm wird die Equilibrierung realisiert, indem die zentrale Schleife zunächst einige Male durchlaufen wird, ohne dabei die physikalischen Eigenschaften des Systems zu messen. In der Regel werden einige tausend Zeitschritte ausgeführt, so dass nur noch statistische Fluktuationen um thermodynamische Mittelwerte erfolgen.

3. **Kräfteberechnung:** Die Kräfteberechnung geschieht mit Hilfe des Wechselwirkungspotentials. Sie ist der zeitintensivste Teil einer MD-Simulation, da sie in der Regel ca. 95% der gesamten Laufzeit benötigt. Im einfachsten Fall muss zur Berechnung der Einzelkräfte, die auf ein Teilchen i wirken, dieses einmal mit allen anderen Teilchen j des Systems wechselwirken. Im Programm wird dies durch eine Doppelschleife realisiert. Das bedeutet, dass die Berechnung der Kräfte für ein System mit N Teilchen mit $O(N^2)$ skaliert. Für eine sehr große Teilchenanzahl N bedeutet dies einen unrealisierbar großen Rechenaufwand. Deshalb ist die Kräfteberechnung der Ansatzpunkt für viele Algorithmen, die Laufzeit einer MD-Simulation zu verkürzen. Mittlerweile gibt es Algorithmen mit denen die Kräfteberechnung mit $O(N)$ statt mit $O(N^2)$ skaliert. Auf dieses Problem wird in Kapitel 2.4.4 noch genauer eingegangen.
   Nach der Berechnung der Einzelkräfte wird außerdem noch die Gesamtkraft und die potentielle Energie im System ermittelt.

4. **Integration der klassischen Bewegungsgleichungen:** Nachdem die Einzelkräfte berechnet wurden, können nun die klassischen Bewegungsgleichungen integriert werden. Dieses Problem ist für Drei- und Mehrteilchensysteme nicht mehr exakt lösbar und wird deshalb durch ein numerisches Verfahren angenähert, wobei die klassischen Bewegungsgleichungen numerisch integriert werden. Ausgehend von der gegebenen Position, Geschwindigkeit und Kraft

eines Teilchens zu einem bestimmten Zeitpunkt berechnet der Integrator die Veränderung der Position und Geschwindigkeit dieses Teilchens, die sich in einem Zeitschritt ergibt. Diese Berechnung wird für alle Teilchen des Systems durchgeführt. Anschließend werden mit den veränderten Positionen und Geschwindigkeiten erneut die Kräfte berechnet. Die Kräfteberechnung und die Integration der Bewegungsgleichungen bilden den Kern einer MD-Simulation. Die beiden Schritte werden solange wiederholt, bis die gewünschte Simulationslänge erreicht ist.

5. **Auswertung:** Neben der potentiellen Energie und den Positionen und Geschwindigkeiten der Teilchen, müssen auch die Größen, über die eine Simulation Aufschluss geben soll, berechnet werden. Solche Größen sind beispielsweise die Temperatur, der Druck oder verschiedene Arten von Korrelationsfunktionen. Es ist sinnvoll, solche Berechnungen oder auch Mittelungen von wichtigen Systemgrößen während der laufenden Simulation durchzuführen und auszugeben. Anderenfalls müsste die gesamte Entwicklung des $6N$-dimensionalen Phasenraums abgespeichert werden.

Wie bei einem realen Experiment können auch bei einer Simulation Fehler auftreten, die zu falschen Ergebnissen führen. Ein typischer Fehler ist, dass das System nicht korrekt vorbereitet wurde, bzw. die Parameter im Simulationsprogramm nicht korrekt sind. Des Weiteren kann der Mess- bzw. Simulationszeitraum zu kurz gewählt sein oder es wird in Wirklichkeit nicht das gemessen, was man glaubt zu messen. Bei der MD-Simulation wie bei Experimenten ist also immer Achtsamkeit und Sorgfalt geboten [10].

2.4.3 Grenzen der Computersimulation

Trotz der Vorteile die eine Computersimulation bietet, bedeutet das nicht, dass reale Experimente nicht mehr nötig sind. Bei der Beschreibung von Wechselwirkungen muss man davon ausgehen, dass jedes Objekt im Universum mit jedem anderen Objekt wechselwirkt [19]. Eine Simulation des gesamten Universums ist jedoch nicht realisierbar, da der Rechenaufwand unendlich groß werden würde. Außerdem sind meist nicht alle Faktoren, die in ein Experiment einfließen, bekannt. Eine Computersimulation kann also lediglich Ausschnitte der Realität approximieren. Daher ist es immer notwendig, die aus der Simulation gewonnenen Ergebnisse mit den theoretischen und experimentellen Ergebnissen zu vergleichen. Weichen die simulierten Ergebnisse zu stark von der Realität ab, muss das Modell entsprechend korrigiert werden. Simulation, Theorie und Realität gehen also immer Hand in Hand [21].

Durch die beschränkte Rechenleistung werden der Computersimulation weitere Grenzen gesetzt. Insbesondere wird die Anzahl der zu simulierenden Teilchen durch die Geschwindigkeit der Rechenoperationen limitiert. Im Vergleich zu den ersten molekulardynamischen Simulationen vor ca. 50 Jahren können heute aufgrund der enorm schnell gewachsenen Rechenleistung wesentlich mehr Teilchen simuliert werden. Dennoch ist man versucht, die maximal mögliche Teilchenzahl weiter zu erhöhen.

Alder und Wainwright waren die Ersten, die im Jahre 1957 die Methode der Computersimulation für ihre Studien benutzten. Sie simulierten ein System von harten Kugeln auf einer *IBM 704*. Bei einem System von harten Kugeln wird die Größe eines Zeitschritts durch die Zeit bis zur nächsten auftretenden Kollision zweier Kugeln bestimmt. Mit der IBM 704 konnten bei einer Gesamtzahl von

500 Teilchen 500 Kollisionen pro Stunde gerechnet werden. Für einen sinnvollen Simulationslauf von 200.000 Kollisionen benötigte der Rechner also mehr als zwei Wochen [25].

A. Rahman war 1964 der Erste, der ein System simulierte, bei dem die Teilchen durch ein kontinuierliches Potential miteinander wechselwirkten. Die Simulation wurde auf einer *CDC 3600* durchgeführt. Für die Wechselwirkungsberechnung der insgesamt 864 Teilchen benötigte die Maschine 45 Sekunden pro Zeitschritt (die Zeitschritte eines solchen Systems sind konstant). Eine Berechnung von 50.000 Zeitschritten dauerte also mehr als drei Wochen. Ein Standard-PC benötigt heutzutage für diese Rechnung höchstens eine Stunde [25].

Durch die Entwicklung schnellerer, größerer und auch paralleler Rechnerarchitekturen konnten die zu simulierende Zeitspanne und Systemgröße erheblich vergrößert werden. Im Jahre 1999 simulierte J. Roth auf der *Cray T3E-1200* in Jülich ein System mit $5 \cdot 10^9$ Teilchen. Dies wurde durch IMD ermöglicht, ein MD-Programm, das die 512 Knoten der Cray mit jeweils 256 MB Speicher sehr effizient ausnutzte. Allerdings würde eine Simulation von 10.000 Zeitschritten unter Ausnutzung der gesamten Ressourcen der Maschine etwa ein Vierteljahr dauern [25].

2.4.4 Methoden zur Beschleunigung einer MD-Simulation

Das Ziel ist es, die Laufzeit einer Simulation weiter zu verkürzen, um größere Systeme simulieren zu können. Dies erreicht man zum einen durch optimierte Algorithmen und zum anderen durch schnellere Rechnerarchitekturen. Mit der Optimierung setzt man dabei hauptsächlich bei der Berechnung der Wechselwirkungen an.

Durch die Optimierung der Algorithmen wurde in den letzten Jahren eine höhere Beschleunigung der Laufzeit erreicht, als durch den Einsatz schnellerer Rechner. Die Berechnung von kurzreichweitigen Wechselwirkungen lässt sich mit Hilfe von Listen-Techniken (siehe Kapitel 2.3) auf ein Problem der Komplexität $O(N)$ zurückführen. Auch langreichweitige Wechselwirkungen lassen sich entsprechend optimieren. Mittels gitterbasierter Ewald-Summationen und Baum-Algorithmen lässt sich die Komplexität auf $O(N\ log\ N)$ reduzieren. Multipol-, Mehrgitter- und Waveletverfahren erreichen sogar eine Komplexität von $O(N)$. Besonderes Augenmerk wird außerdem auf die Parallelisierung der MD-Algorithmen gelegt.

Auf der anderen Seite werden verstärkt spezialisierte Computerarchitekturen eingesetzt um größere Systeme simulieren zu können. Für die Berechnung der Wechselwirkungen wird meist nur ein begrenzter Umfang an Operationen benötigt. Deshalb verwendet man für die Kräfteberechnung Computerarchitekturen, die diese Operationen extrem schnell durchführen können. Beispiele hierfür sind APE- und GRAPE-Rechner (siehe Kapitel 3.4.1), die im Bereich der Quantenchromodynamik und der Astrophysik eingesetzt werden. Diese Rechner besitzen jedoch eine fest codierte Logik und können somit nur für eine spezielle Problemklasse eingesetzt werden. Eine flexiblere Alternative bietet der Einsatz rekonfigurierbarer Logik, wie den FPGAs. Sie können umkonfiguriert und somit für verschiedene Algorithmen verwendet werden. Mit diesem Ansatz zur Beschleunigung molekulardynamischer Berechnungen beschäftigen sich die folgenden Kapitel dieser Arbeit.

Kapitel 3

FPGA

Die Entwicklung rekonfigurierbarer Hardware, der so genannten PLDs (***P**rogrammable **L**ogic **D**evice*), ist ein wichtiger Meilenstein in der Halbleiterindustrie. Wurde zuvor die anwendungsspezifische Funktion einer Schaltung entweder durch die Auswahl geeigneter Standardbauteile und deren Verdrahtung auf einer Platine oder durch die Herstellung einer speziellen integrierten Schaltung erreicht, so kann nun der Anwender die Verschaltung selbst programmieren und jederzeit ändern.

Als eine spezielle Bauart für PLDs wurden Mitte der 80er Jahre die ersten FPGAs auf den Markt gebracht. Mit weniger als 1.000 Gatteräquivalenten waren die Möglichkeiten der FPGAs damals jedoch noch sehr beschränkt. Aufgrund ihrer geringen Logikdichte und Geschwindigkeit und ihrer hohen Anschaffungskosten boten sie zunächst keine echte Alternative zu fest codierter Logik, wie z. B. den *ASICs*. Mittlerweile jedoch sind FPGAs mit einer Million Gatteräquivalenten verfügbar. Zudem wächst die Verarbeitungsgeschwindigkeit ständig an, wogegen die Preise sinken. FPGAs bieten heute nicht nur Alternativen zu herkömmlichen Lösungen, sondern ermöglichen auch völlig neue Anwendungen. Damit stellt die programmierbare Logik einen wesentlichen Beitrag zur Technologie der Zukunft dar [29].

In diesem Kapitel werden zunächst einige Beispiele für die verschiedensten Anwendungsgebiete für FPGAs gegeben. Anschließend wird am Beispiel der Xilinx-FPGAs die grundlegende Architektur eines FPGAs beschrieben. Dies führt zur Programmierung von FPGAs. Dazu werden zunächst die Unterschiede zwischen Hardware- und Softwareprogrammierung verdeutlicht. Abschließend wird auf den Einsatz von FPGAs als Koprozessoren und die Programmierung solcher Koprozessoren eingegangen. In diesem Zusammenhang wird die Architektur des PROGRAPE-4-Boards erläutert, welches als Hardwarebeschleuniger in dieser Arbeit eingesetzt wurde.

3.1 Anwendungsgebiete

Durch ihre Rekonfigurierbarkeit bieten FPGAs weitreichende Einsatzmöglichkeiten für verschiedenste Bereiche und Anwendungen. Dazu trägt unter anderem die immer einfacher werdende Programmierung bei. Diese Vereinfachung wird nicht nur durch hochentwickelte Programmiersprachen erreicht, sondern auch durch vorgefertigte Designs, die in Form von IP-Cores zum Quelltext hinzugebunden werden können. Besonders in Bereichen, in denen Algorithmen oder Protokolle schnell

weiterentwickelt werden, ist die Verwendung rekonfigurierbarer FPGAs vorteilhaft. Produkte können so schneller auf den Markt gebracht und an neue Entwicklungen angepasst werden. Außerdem können durch die Rekonfigurierbarkeit Entwicklungsfehler nachträglich behoben werden.

Im Folgenden werden einige Beispiele für den Einsatz von FPGAs genannt [30].

- **Luft- und Raumfahrt/militärische Anwendungen:** In diesem Bereich werden hauptsächlich FPGAs mit strahlungsfestem Gehäuse eingesetzt. Sie werden vor allem für die Steuerung und Kopplung mehrerer Bildsignalquellen und für die Signalverarbeitung unter Echtzeitbedingungen verwendet.

- **Automobilindustrie:** In der Automobilindustrie werden FPGAs für die schnelle und einfache Implementierung von Designs für die Automobilelektronik verwendet. Hierbei sorgen FPGAs für den effizienten Transport von Datenströmen, der für eine Vernetzung mehrerer Peripheriefunktionen im Auto, wie Navigation, Entertainment oder Fahrerinformationssystemen, nötig ist.

- **Rundfunk und Medienbranche:** Das Senden und Empfangen von Kurz-, Mittel- und Langwellen-Signalen spielt in der Rundfunk- und Medienbranche eine große Rolle. Für die Signalverarbeitung und die Decodierung von Musik, Sprache und Datendiensten werden FPGAs eingesetzt.

- **Verbraucher:** Auch für Verbraucheranwendungen bieten FPGAs kostengünstige Lösungen. Sie werden beispielsweise für digitale Flachbildschirme, die Vernetzung daheim oder Digitalempfänger verwendet.

- **Industrie/Wirtschaft/Medizin:** In den Bereichen Industrie, Wirtschaft und Medizin werden FPGAs vor allem für die Automatisierung, zur Motorkontrolle und für die Bildverarbeitung verwendet.

- **Speicher und Server:** FPGAs eignen sich für die Realisierung sehr schneller Speicher- und Schnittstellensysteme. Sie werden unter anderem in *NAS* und *SAN*-Systemen verwendet.

- **Kommunikation:** FPGAs sind sowohl in der drahtlosen, als auch in der drahtgebundenen Kommunikation einsetzbar. Sie werden beispielsweise bei der Protokoll-Abarbeitung für verschiedene Übertragungsstandards, wie *GPRS* oder Ethernet-MAC-Layer verwendet.

- **Anwendung als Prototypen:** Bei der Herstellung von Schaltungen kann eine Simulation in vielen Fällen nicht mehr effizient durchgeführt werden. Zur Verifikation werden deshalb FPGA-basierte Prototypen eingesetzt und erst danach wird die Schaltung auf den Chip gebracht.

- **Beschleunigung von Algorithmen:** Mit dieser Art der Anwendung von FPGAs beschäftigt sich diese Diplomarbeit. Die rekonfigurierbare Hardware fungiert dabei als Koprozessor. Rechenintensive Teile eines Algorithmus werden nicht mehr auf der CPU, sondern auf einem oder mehreren FPGAs ausgeführt. In Kapitel 3.4 wird auf die Funktion von FPGAs als Koprozessoren genauer eingegangen.

Abbildung 3.1: Foto eines FPGAs

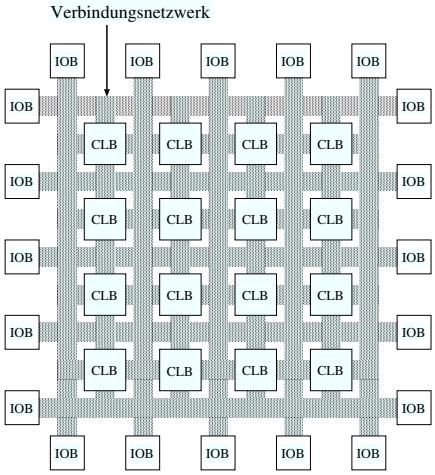


Abbildung 3.2: Architektur eines FPGAs

3.2 Allgemeine Architektur von FPGAs

Ein FPGA ist ein Siliziumchip, wie er in Abbildung 3.1 dargestellt ist. Die Abkürzung FPGA steht für ***F**ield **P**rogrammable **G**ate **A**rray*. Der Name weist auf die allen FPGAs gemeinsame Struktur und Eigenschaft hin, denn ein FPGA besteht im Wesentlichen aus einer Matrix programmierbarer Logikelemente [20].

Grundsätzlich besteht ein FPGA aus drei Komponenten: Logikzellen , I/O-Zellen und einem elektrischen Verbindungsnetzwerk. FPGAs gibt es in unterschiedlichen Bauformen und von verschiedenen Herstellern. Neben den führenden Herstellern Xilinx und Altera haben sich auch Firmen wie Actel, Lucent, Lattice und Atmel in der Herstellung von FPGAs etabliert [3, 22]. Abbildung 3.2 stellt den Aufbau eines FPGAs schematisch dar.

Die Logikzellen, oder auch Logikblöcke (CLBs) genannt, sind die wichtigsten rekonfigurierbaren Elemente eines FPGAs. Sie sind in einer quadratischen Matrix angeordnet. Ihnen kann im Rahmen der verfügbaren Ressourcen ein beliebiges digitalelektronisches Verhalten zugewiesen werden. Die I/O-Zellen oder I/O-Blöcke (IOBs) bilden die Schnittstellen zwischen der FPGA-internen Schaltungslogik und der Außenwelt. Sie können für verschiedene elektrische Verbindungsstandards konfiguriert werden, so dass FPGAs in einer Vielzahl von digitalelektronischen Umgebungen eingesetzt werden können. Sämtliche Logik- und I/O-Zellen sind am elektrischen Verbindungsnetzwerk angeschlossen. Durch Programmierung können einzelne Verbindungen von Logikzellen geschaltet oder unterbrochen werden. Auf diese Weise kann eine nahezu beliebige Verbindung der Zellen untereinander erreicht werden, die jedoch durch die Anzahl der verfügbaren Leitungen begrenzt ist. Innerhalb des Verbindungsnetzwerkes gibt es meist noch speziellere Leitungen für die Verwendung eines globalen Taktsignals. Durch ihren Aufbau kommt das Signal an jeder Logikzelle nahezu zeitgleich an [22].

Je nach Hersteller und Bauform befinden sich neben den drei Grundelementen noch zusätzliche Module auf dem FPGA. Bei den Virtex-II FPGAs von Xilinx sind das Block-RAMs und Multiplizier-Elemente [20].

In Rahmen dieser Diplomarbeit wurde ausschließlich mit FPGAs von Xilinx gearbeitet, deshalb wird im Folgenden speziell auf die Architektur der Xilinx-FPGAs eingegangen. Die FPGAs anderer Hersteller können von dieser Grundarchitektur abweichen, z. B. durch die Verwendung unterschiedlicher Logikblöcke oder Speichereinheiten. Die oben beschriebene Grundstruktur ist jedoch auch bei ihnen wiederzufinden.

3.2.1 FPGAs der Virtex-Serie

Neben den oben genannten Grundelementen besitzen die FPGAs der Virtex-Serie zusätzlich noch DCMs (***D**igital **C**lock **M**anager*), Block-RAM- und Multiplizier-Elemente.

Durch die unterschiedlich langen Verbindungen erreicht ein Taktsignal die einzelnen CLBs nicht gleichzeitig. Für Pipelines z. B. ist es jedoch wichtig, dass sie synchron getaktet sind, da sonst undefinierte Werte zustande kommen können. Mit Hilfe von Verzögerungsregistern sorgen die DCMs deshalb dafür, dass der Takt alle CLBs gleichzeitig erreicht. Die Block-RAM-Elemente sind SRAM-Elemente, deren Eigenschaften, wie z. B. Bitbreite und Speichertiefe, konfiguriert werden können. Je nach Größe besitzt ein FPGA vier bis 168 solcher Block-RAM-Elemente mit jeweils 18 kBit Speicherkapazität. Sie bieten damit lokale Speicherressourcen, die weit über die Speicherfähigkeit der CLBs hinausgehen. Die Multiplizier-Elemente können 18-Bit-Integer-Zahlen multiplizieren. Sie sind durch fest verdrahtete Logik aufgebaut. Damit ist ihr Flächenbedarf auf dem Chip weitaus geringer als bei gleichartigen Multiplizierern, die durch Verwendung von CLBs implementiert werden [20].

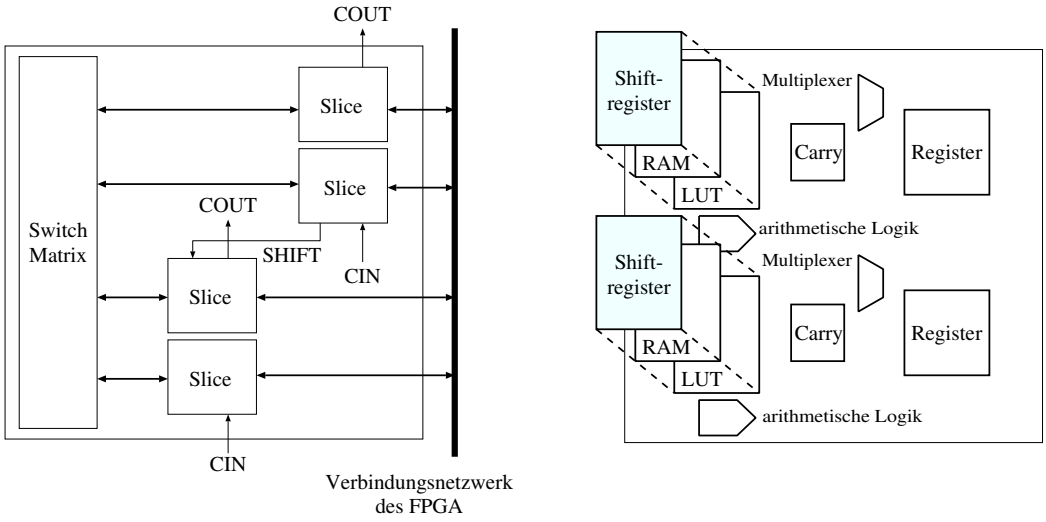
Detailliertere Informationen zu den Virtex FPGAs können den Datenblättern [31, 32] entnommen werden.

3.2.2 Konfigurierbare Logikblöcke (CLBs)

Die konfigurierbaren Logikblöcke eines FPGAs werden auch als CLBs (***C**onfigurable **L**ogic **B**locks*) bezeichnet. Die Anzahl der CLBs auf einem FPGA variiert zwischen den einzelnen Herstellern und Architekturen. Die in der *Cray XD1* verwendeten Virtex-4 FPGAs (XC4VLX140) von Xilinx besitzen beispielsweise 152.064 CLBs [8, 31]. Die Anzahl und Komplexität der CLBs ist außerdem ein Maß für die logische Größe und Leistungsfähigkeit eines FPGAs. Da jedoch die Bauform der CLBs von Hersteller zu Hersteller variiert, gibt man die Leistungsfähigkeit meist in so genannten Gatteräquivalenten an. Die Anzahl der Gatteräquivalente gibt an, wieviele NAND-Gatter zur Realisierung der Komplexität des FPGAs notwendig wären. Dieses Maß stammt aus dem Bereich der *Gate Arrays*. Da die Logikblöcke der FPGAs jedoch nicht wie Gate Arrays aus einzelnen Gattern aufgebaut sind, bietet dieses Maß für FPGAs einen großen Ermessensspielraum. Zudem gibt es für die erforderliche Umrechnung keine einheitlichen Regeln, so dass die Hersteller verschiedene Maßstäbe anlegen. Die angegebenen Gatteräquivalente sind daher nur eine grobe Richtlinie [3, 22, 29].

Ein CLB der Virtex-Serie von Xilinx besteht aus drei Grundelementen (siehe Abb. 3.3(a)): einer Switch-Matrix, Slices und einer Verdrahtungstruktur.

Die Switch-Matrix verbindet den CLB mit dem Verbindungsnetzwerk des FPGAs. Je nach Konfiguration des FPGAs lässt sie die Signale durch oder blockiert sie. Die durchgelassenen Signale werden über ein schnelles Verbindungsnetzwerk an die Slices weitergeleitet.



(a) CLB (b) Slice-Einheit

Abbildung 3.3: Struktur eines CLBs und einer Slice-Einheit der Virtex-Serie

Ein CLB besitzt vier Slices. Abbildung 3.3(b) zeigt die Struktur einer solchen Slice-Einheit. Ein Slice besteht aus zwei Funktionsgeneratoren, zwei Speicherelementen, Multiplexern, Carry-Logik und weiterer arithmetischer Logik. Die Funktionsgeneratoren, oder auch LUTs (***L**ook-**u**p **T**able*) genannt, besitzen vier Eingänge (bei Virtex-5 sind es sechs) und einen Ausgang. Eine LUT kann eine beliebige logische Funktion (NAND, XOR, AND, Multiplexer, usw.) aus den Eingangssignalen realisieren. Dies geschieht, indem in einem kleinen Speicher für jeden Zustand der Eingänge der Wert abgelegt wird, den der Ausgang annehmen soll. Für Funktionen, die mehr Eingänge erfordern, als eine einzige LUT besitzt, können mehrere LUTs direkt miteinander verschaltet werden. Darüber hinaus lassen sich LUTs noch als serielle Schieberegister oder RAMs konfigurieren. Um *sequentielle Logik* zu ermöglichen besitzen die Slices Register, die als *Flipflops* oder *Latches* verwendet werden können. Die Multiplexer-Elemente erlauben die Kombination der Funktionseinheiten sowie die Verknüpfung logischer Zwischenergebnisse mit anderen Slices des CLBs. Die zusätzliche Logik in den Slices dient der Implementierung schneller arithmetischer Operatoren, wie z. B. Integer-Addierern oder Zählern [20, 32].

3.3 Programmierung von FPGAs

FPGAs werden programmiert, indem ein so genannter Konfigurationsbitstrom auf den Chip transferiert wird. Nach der Konfiguration verhält sich der Baustein wie eine für eine Anwendung speziell konstruierte integrierte Schaltung. Im Unterschied zu ASICs kann das elektronische Verhalten eines FPGAs jederzeit durch erneute Konfiguration geändert werden. Die Konfiguration SRAM-basierter FPGAs ist jedoch flüchtig, d. h., dass der FPGA bei Wegnahme der Stromversorgung seine Programmierung verliert. Auf den meisten FPGA-Boards platziert man deshalb neben den FPGA ein wiederbeschreibbares *EEPROM*, in welchem der Konfigurationsbitstrom gespeichert wird. Beim Einsetzen der Stromzufuhr wird der Bitstrom dann aus dem EEPROM auf den FPGA geladen [22]. Ein Schaltungsdesign, auf dem der Konfigurationsbitstrom für den FPGA basiert, wird im Folgenden FPGA-Design genannt [20].

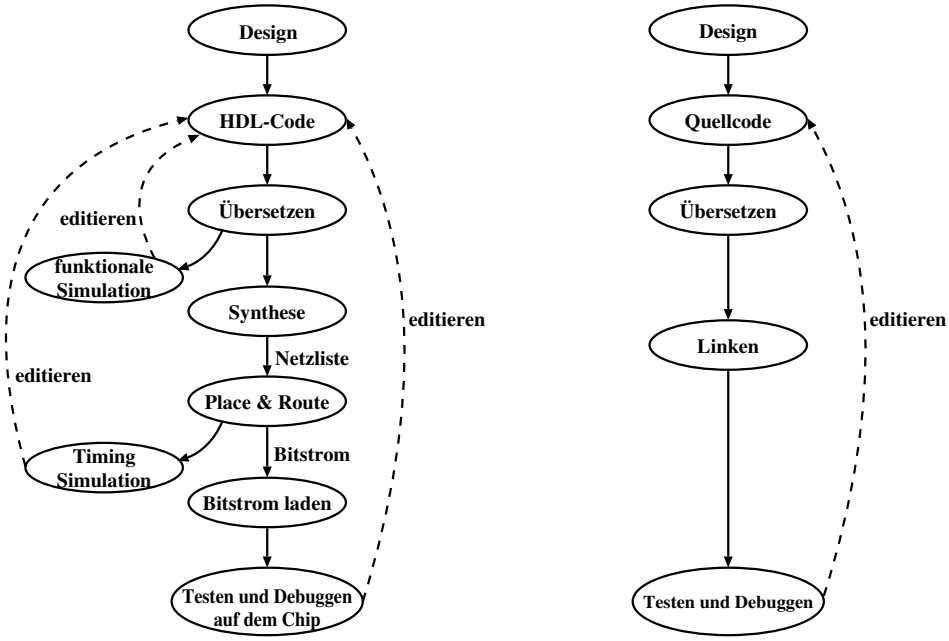


Abbildung 3.4: Entwicklungsablauf von Hardware- und Softwareentwicklung im Vergleich

3.3.1 Unterschied Hardware- und Softwareprogrammierung

Zwischen Hardware- und Softwareprogrammierung gibt es einige grundlegende Unterschiede. Ein Hardwaredesign ist die physikalische, ein Softwaredesign die logische Implementierung einer Funktionalität. Bei der Hardwareprogrammierung implementiert man für ein mikroelektronisches System einen bestimmten Umfang an Befehlen. Man programmiert also einen Prozessor, wobei die Befehle durch digitale Schaltungen realisiert werden. Zusammen mit einem Speicher und darin befindlichen Programmen als Abfolge solcher Befehle kommt man zum Begriff der Software [6].

Aufgrund dieser unterschiedlichen Definitionen ergeben sich auch Abweichungen beim Ablauf von Hardware- und Softwareentwicklungen. Abbildung 3.4 zeigt den Entwicklungsablauf von Hardware- und Softwaredesigns.

Die klassische Vorgehensweise bei der Softwareentwicklung beginnt mit der Beschreibung eines Designs, bzw. eines Algorithmus in einer höheren Programmiersprache, wie C, C++, Fortran, usw. Der Compiler übersetzt diesen Quellcode in Maschinensprache. Die vom Compiler generierten Module werden anschließend vom Linker zu einem ausführbaren Programm zusammengefügt. Um die Korrektheit des Algorithmus zu prüfen, wird das ausführbare Programm mit verschiedenen Testbeispielen getestet. Bei einer falschen Ausgabe, wird der Fehler im Quellcode gesucht und beseitigt. Danach wird erneut übersetzt, gelinkt und getestet, solange bis alle Testbeispiele fehlerfrei durchlaufen.

Der Entwicklungsablauf von Hardwaredesigns beginnt wie bei der Softwareentwicklung mit der Beschreibung eines Designs bzw. einer elektronischen Schaltung in einer Programmiersprache. Diese Sprache ist jedoch keine höhere Programmiersprache, sondern eine Hardwarebeschreibungssprache oder kurz HDL (***H**ardware **D**escription **L**anguage*). Die bekanntesten HDLs sind zur Zeit VHDL (*Very High Speed Integrated Circuit HDL*) und Verilog. Bevor das beschriebene Design auf

den FPGA gebracht wird, wird es simuliert. Treten hier Fehler auf, wird der HDL-Code entsprechend korrigiert und erneut simuliert. Ist die Simulation schließlich fehlerfrei, wird in der Synthese eine Netzliste erzeugt und diese anschließend in ein binäres Format konvertiert. Das so codierte Design wird im so genannten *Place & Route* auf dem FPGA platziert. Zum Schluss wird der Bitstrom generiert. Mit speziellen, vom Hersteller mitgelieferten Werkzeugen wird dann nochmals die Korrektheit des Designs überprüft, während es auf dem FPGA läuft. Treten dabei Fehler auf, werden wieder Änderungen am HDL-Code vorgenommen und alle Stufen erneut durchlaufen [28]. Auf die einzelnen Stufen der Hardwareentwicklung wird im Verlauf dieses Kapitels genauer eingegangen.

Bei der Entwicklung von Hardwaredesigns hat man also die Möglichkeit vor der eigentlichen Implementierung das Design zu simulieren und auf Fehler hin zu untersuchen. Während die Simulation bei der Hardwareentwicklung zwingend erforderlich ist, ist sie bei der Softwareentwicklung überflüssig. Ein Grund dafür ist die Zeit, die für eine Übersetzung des Designs benötigt wird. Die Übersetzungszeiten für Softwaredesigns sind in der Regel kurz. Um den Bitstrom eines Hardwaredesigns zu erzeugen, benötigt man dagegen wesentlich mehr Zeit. Die Generierung eines Bitstroms zum Lennard-Jones-Potential (B.2) beispielsweise dauerte 10 Minuten. Für das Linked-Cell-List-FPGA-Design (B.3) betrug die Mindestdauer bereits eine halbe Stunde. Für größere Designs sind Übersetzungszeiten von einem Tag und mehr realistisch [28]. Neben dem Zeitfaktor spielt die Simulation außerdem für den Schutz der Systemkomponenten eine große Rolle. Wird ein fehlerhaftes Design auf den FPGA geladen, ist es möglich, dass dadurch Systemkomponenten beschädigt werden [28].

3.3.2 Hardwareentwicklungsablauf

Wie schon erwähnt, wird ein FPGA programmiert, indem ein Konfigurationsbitstrom auf den FPGA transferiert wird. Um einen solchen Bitstrom zu einem FPGA-Design zu erzeugen gibt es durch hochentwickelte Designwerkzeuge eine ähnlich hohe Abstraktion der Programmierung, wie zwischen Hochsprachenprogrammierung und Maschinencode.

HDL-Code

Die Hochsprachen der Hardwareprogrammierung sind HDLs, wie VHDL oder Verilog. Mit diesen Sprachen wird nach einer vorgegebenen Syntax und Semantik die zu implementierende Schaltung formuliert. Dabei unterscheidet man zwischen der strukturellen und der verhaltensorientierten Beschreibung. Bei der strukturellen Beschreibung wird durch die Spezifikation von gegebenen Bausteinen und deren Verbindungsstruktur eine Schaltung „zusammengebaut“. Die so beschriebene Schaltung ist unabhängig von einem bestimmten Ausführungszeitpunkt. Bei der verhaltensorientierten Beschreibung steht nicht mehr die Verbindungsstruktur im Vordergrund, sondern das Verhalten der Komponenten. Hierbei werden auch Zeitabhängigkeiten beschrieben [18].

Neben den klassischen HDLs existieren seit neuestem auch hochsprachenorientierte Hardwarebeschreibungssprachen wie Handel-C oder System-C. Diese Sprachen sind von der Syntax her C oder Java sehr ähnlich. Ihr Ziel ist es, in konventionellen Programmiersprachen geschriebene Algorithmen mit speziellen Hardwarecompilern direkt in eine effiziente Hardwarestruktur zu kompilieren. Damit soll es auch Softwareprogrammierern ermöglicht werden, Hardwaredesigns zu implementieren.

Durch IP-Cores, die in den eigenen HDL-Code eingebunden werden können, wird die Entwicklung von Hardwaredesigns weiter vereinfacht. IP-Cores (IP = ***I**ntellectual **P**roperty*) sind vorgefertigte Schaltungsentwürfe, die als Block in anderen Schaltungsdesigns wiederverwendet werden können. Beispiele dafür sind Fifos, Prozessoren, serielle Schnittstellen, Ethernet-MAC-Layer, RAM-Controller, Parallel-IO, etc. [3]. Sie werden von den Herstellern der FPGAs zur Verfügung gestellt. Dabei unterscheidet man zwischen „Hard IP-Cores“, die als Netzliste für eine bestimmte FPGA-Technologie geliefert werden, und „Soft IP-Cores“, welche als synthetisierbare Schaltungsbeschreibung in einer HDL erhältlich sind.

Simulation

Die softwarebasierte Simulation eines FPGA-Designs ist notwendig, da das Testen des Designs direkt an der Hardware schwierig ist und sogar Schaden an Hardwarekomponenten anrichten kann. Die Fehlersuche im Echtzeitbetrieb ist aufgrund der hohen Schaltgeschwindigkeiten sehr aufwändig. Des Weiteren sind einige Signale der Schaltung nicht von außen zugänglich, so dass nicht alle Signalverläufe überwacht werden können. Bei der Simulation dagegen sind sämtliche Signale verfügbar. Die einzelnen zeitlichen Verläufe lassen sich leicht überwachen und aufzeichnen. Außerdem besteht keine Gefahr, Schaden anzurichten.

Im Hardwareentwicklungsablauf gibt es zwei Arten der Simulation. Zum einen gibt es die funktionale Simulation. Sie wird durchgeführt, nachdem der HDL-Code zum gewünschten FPGA-Design erzeugt wurde. Hierbei wird die funktionale Korrektheit des Designs getestet. Die andere Art ist die Timing-Simulation. Sie wird nach dem *Place & Route* durchgeführt und berücksichtigt zusätzlich zeitliche Faktoren.

Bei der Durchführung einer Simulation arbeitet man mit Testvektoren oder Testbenches. Testvektoren bestehen aus einer Reihe von Werten, wobei zu jedem Eingangssignal das erwartete Ausgangssignal angegeben ist. Unter Verwendung dieser Eingangssignale wird dann bei der Simulation überprüft, ob die tatsächlichen Ausgangssignale mit den erwarteten übereinstimmen. Für komplexere Simulationen eignen sich Testbenches besser als Testvektoren. Eine Testbench simuliert eine reale Umgebung für das zu testende FPGA-Design. Ihr Vorteil ist, dass sie Eingangssignale auch als Reaktion auf die Signale des zu testenden Designs erzeugen kann. Simulationswerkzeuge, wie beispielsweise *ModelSim* von *Mentor Graphics* [1] erleichtern das Erstellen von Testbenches und die Fehlersuche.

Synthese

Nachdem mit Hilfe der funktionalen Simulation die Korrektheit der Funktion des FPGA-Designs gewährleistet ist, wird das Design synthetisiert. Dabei werden die in der formalen Beschreibung, also dem HDL-Code, festgelegten Elemente auf die vorhandenen Ressourcen des FPGAs abgebildet. Die endgültige FPGA-Implementierung wird jedoch noch nicht festgelegt. Das Ergebnis der Synthese ist die Netzliste. Diese enthält die schaltungstechnische Struktur des FPGA-Designs, also alle verwendeten Ressourcen, ihre Parameter und Verschaltungsstruktur. Außerdem beinhaltet sie Angaben über einzuhaltende Zeitkriterien, Platzierungsinformationen für die einzelnen Elemente, Startzustände von Speicherelementen und die Zuordnung der nach außen gerichteten Signale zu den Gehäusepins [18, 20].

Auch für die Synthese existieren geeignete Werkzeuge, die die Entwicklung eines FPGA-Designs erleichtern und die Generierung der Bitströme automatisieren. Solche Synthesewerkzeuge sind beispielsweise ISE [4] und *Synplify* [2].

Place & Route

Die Netzliste dient als Eingabe für das so genannte *Place & Route*. Hier wird die endgültige FPGA-Implementierung festgelegt. Die einzelnen Logikelemente werden auf dem FPGA platziert (*Place*), d. h. es wird festgelegt, welche CLBs welche logische Funktion realisieren. Durch ein automatisches Routing-Verfahren werden dann die einzelnen Elemente mit den vorhandenen Verbindungsressourcen miteinander verbunden (*Route*), d. h. es wird festgelegt, welche Leitungen freigeschaltet werden.

Dieser Vorgang ist sehr rechenintensiv und kann mehrere Stunden benötigen. Der Zeitaufwand hängt dabei nicht nur von der Komplexität des FPGA-Designs ab, sondern auch vom Grad der Optimierung. Optimierung bedeutet dabei, dass die einzelnen Elemente so platziert werden, dass die Verbindungswege möglichst kurz sind. Dadurch werden die Signallaufzeiten gering gehalten und der Flächenverbrauch für die benutzten Leitungen minimiert. Der Grad der Optimierung ist abhängig vom verwendeten Werkzeug. *Synplify* ermöglicht beispielsweise eine höhere Optimierung als ISE.

Nach dem *Place & Route* wird das reale Zeitverhalten der Schaltung, also die Laufzeit der Signale, berechnet. In der Timing-Simulation wird dann das Design erneut unter Berücksichtigung des Zeitverhaltens simuliert. Außerdem wird noch der längste Pfad bzw. die längste Taktperiode des Designs ermittelt und daraus die maximale *Taktfrequenz* berechnet. Treten dabei Fehler auf, werden diese entsprechend beseitigt und alle Schritte erneut durchlaufen.

Das Ergebnis des *Place & Route* ist schließlich der Konfigurationsbitstrom für den FPGA. Dieser wird auf den FPGA geladen und der Chip damit konfiguriert. Abschließend werden noch Tests mit dem konfigurierten FPGA durchgeführt. Treten hier Fehler auf, müssen erneut Änderungen am HDL-Code vorgenommen werden, solange bis das Design fehlerfrei ist.

3.4 Rekonfigurierbare Koprozessoren

Ein weit verbreitetes Anwendungsgebiet für FPGAs sind rekonfigurierbare Koprozessoren. Bei solchen Koprozessoren handelt es sich um FPGA-basierte Komponenten, die eng mit einem konventionellen Mikroprozessor zusammenarbeiten. Das Ziel eines FPGA-Koprozessors ist nicht, den Prozessor durch eine eigene Hardware zu ersetzen. Die Grundidee besteht vielmehr darin, lediglich zeitintensive Teile eines Algorithmus auf dem FPGA zu rechnen und somit zu beschleunigen. Der restliche Algorithmus wird wie gewohnt vom Mikroprozessor bearbeitet. Die Beschleunigung wird dabei dadurch erreicht, dass die Hardware des Koprozessors durch die FPGAs speziell an den Algorithmus bzw. den entsprechenden Teil des Gesamtalgorithmus angepasst werden kann. Im Falle der Molekulardynamik lassen sich beispielsweise Pipelines zur Berechnung der Wechselwirkungen auf dem Koprozessor realisieren [18].

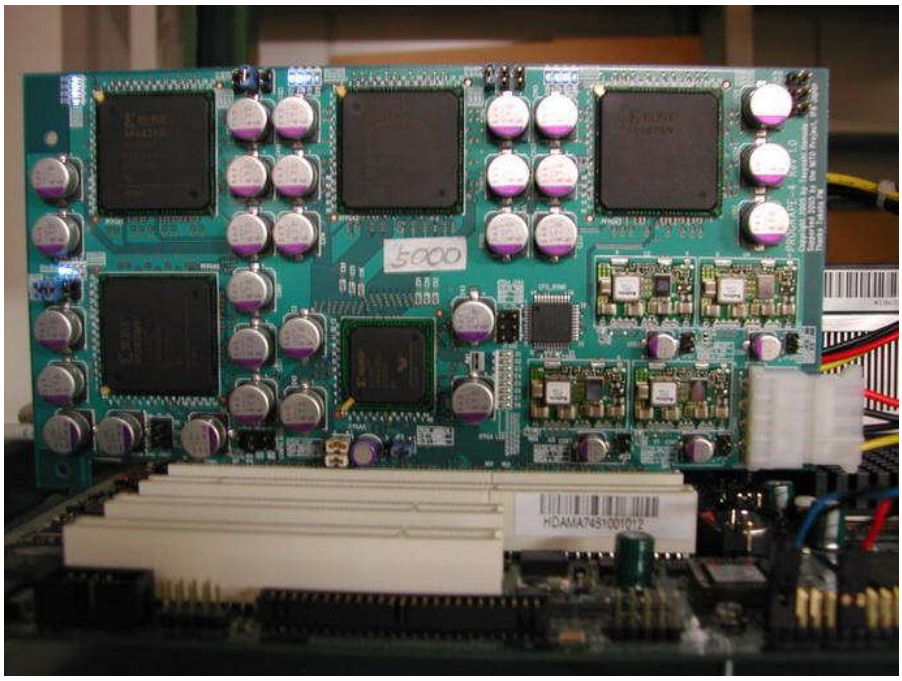


Abbildung 3.5: Foto des PROGRAPE-4-Boards

FPGA-Koprozessoren lassen sich auf unterschiedliche Weise realisieren. Eine übliche Variante für rekonfigurierbare Koprozessoren sind PCI-Einsteckkarten. Sie beinhalten in der Regel einen oder mehrere FPGAs, lokale Speicher, ein konfigurierbares Taktsystem und eine schnelle Schnittstelle für die Konfiguration der FPGAs und den Datentransfer [18]. Das PROGRAPE-4-Board ist ein Beispiel für einen FPGA-Koprozessor. Es wird im Folgenden genauer beschrieben.

3.4.1 PROGRAPE-4

Das PROGRAPE-4-Board wurde Anfang 2006 am RIKEN (Institute of Physical and Chemical Research) in Wako (Saitama, Japan) entwickelt. Das Board ist die vierte Generation der PROGRAPE-Technologie. Es wurde für die Beschleunigung der Berechnung und Simulation von Vielteilchensystemen konzipiert. Der Name PROGRAPE steht dabei für ***PRO****grammable* ***GRAPE***.

PROGRAPE ist eine Weiterentwicklung der GRAPE-Technologie. Im Gegensatz zu GRAPE sind die PROGRAPE-Boards jedoch programmierbar. Die Bezeichnung GRAPE bedeutet dabei ***GRA****vity* ***P****ipelin****E***. Wie der Name vermuten lässt, enthalten die Boards der GRAPE-Technologie eine Pipeline speziell für die Berechnung der Gravitation in astrophysikalischen Vielteilchensystemen. Die GRAPE-Boards berechnen mit Hilfe von ASICs die Gravitationskraft, die Zeitableitung der Gravitationskraft und das Gravitationspotential. Alle anderen für eine Simulation notwendigen Berechnungen werden auf dem Wirtsrechner durchgeführt. Da die Kräfteberechnung 95% der Rechenzeit einer Simulation beansprucht, ist es sinnvoll lediglich diesen Teil zu beschleunigen. Durch ihre fest codierte Hardware ist das Einsatzgebiet der GRAPE-Boards jedoch auf Gravitationsberechnungen beschränkt. Die Idee, einen solchen Beschleuniger auch für andere Probleme, z. B. für Wechselwirkungen, die auf einem Lennard-Jones-Potential basieren, benutzen zu können, führte zur Entwicklung der PROGRAPE-Boards. Dabei wurde die fest codierte Logik des GRAPE durch rekonfigurierbare FPGAs ersetzt [15].

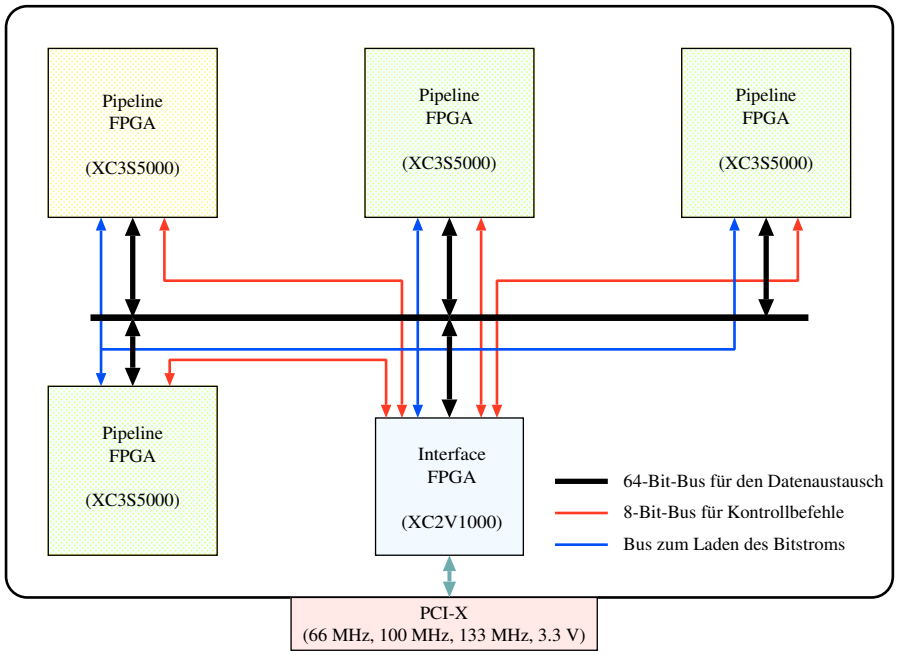


Abbildung 3.6: Architektur des PROGRAPE-4-Boards

Die PROGRAPE-Boards basieren auf dem gleichen Prinzip wie die GRAPE-Boards. Sie dienen als Koprozessoren, d. h. es wird nicht die gesamte Berechnung auf dem Board durchgeführt, sondern lediglich rechenintensive Teile, wie im Fall der Molekulardynamik die Kräfteberechnung. Die restlichen Berechnungen werden vom Wirtsrechner durchgeführt. Im Unterschied zu GRAPE, lassen sich mit PROGRAPE Pipelines für verschiedenste Probleme realisieren [15].

Das PROGRAPE-4-Board besitzt insgesamt 311.040 CLBs. Laut Angaben des Herstellers lässt sich damit eine Grenzleistung von 243,2 GFlops erreichen. Diese Angabe bezieht sich auf gravitative Vielkörperprobleme, in denen keine Kollisionen auftreten. Außerdem ist zu beachten, dass für die Berechnung gravitativer Wechselwirkungen eine geringe Genauigkeit (26 Bit zur Darstellung einer Fließkommazahl) ausreichend ist, was zusätzlich Ressourcen auf den FPGAs spart. Das Board kostet 1.543 € [12].

Abbildung 3.6 zeigt die Architektur des PROGRAPE-4-Boards. Es besteht aus insgesamt fünf FPGAs, vier Spartan-3 FPGAs (XC3S5000) und einem Virtex-II FPGA (XC2V1000) der Firma Xilinx. Die vier Spartan-3 FPGAs können vom Wirtsrechner mit einer beliebigen Konfiguration versehen werden. Sie werden im Folgenden Pipeline-FPGAs genannt. Sie können nur mit dem gleichen Pipelinedesign konfiguriert werden. Auf dem Board befinden sich also immer mindestens vier parallele Pipelines. Durch entsprechende Programmierung (siehe Kapitel 4) lassen sich auch mehrere Pipelines auf einem FPGA unterbringen, wodurch eine noch höhere Parallelität erreicht werden kann. Der Virtex-II FPGA kann nicht mit einem Pipelinedesign versehen werden. Er dient als Schnittstelle und wird im Folgenden Interface-FPGA genannt. Er erhält die Daten und Befehle vom Wirtssystem über einen PCI-X-Bus und leitet sie entsprechend an die Pipeline-FPGAs weiter. Kontrollbefehle, z. B. zum Starten einer Berechnung, werden über einen 8-Bit-Bus weitergeleitet. Die einzelnen Daten werden über einen 64-Bit-Bus zu den Pipeline-FPGAs gesendet. Der Interface-FPGA regelt dabei gleichzeitig die Verteilung der Daten auf die einzelnen FPGAs. Der Konfigurationsbitstrom wird über einen gesonderten Bus auf die vier Pipeline-FPGAs geladen. Das PROGRAPE-4-Board arbeitet mit Taktfrequenzen von 66, 100 und 133 MHz [14].

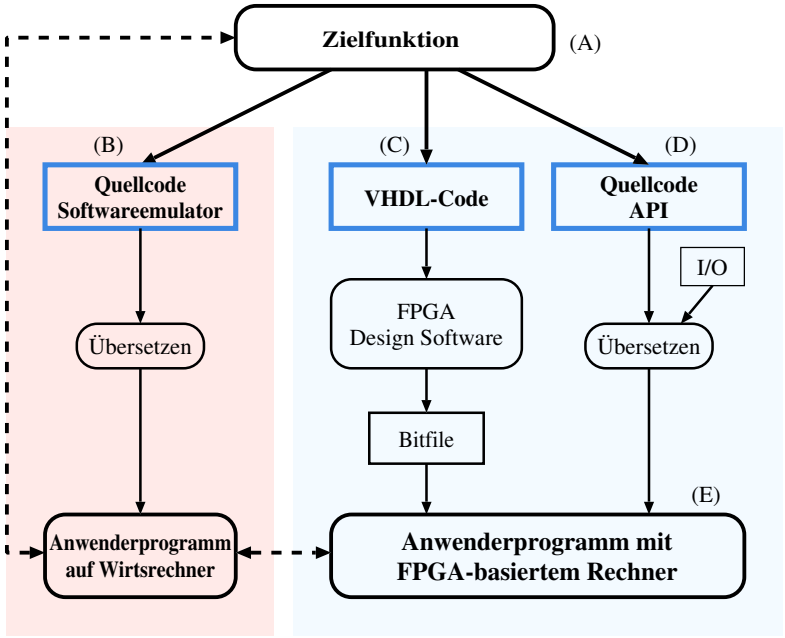


Abbildung 3.7: Entwicklungsablauf eines FPGA-basierten Rechners

Zur Programmierung eines PROGRAPE-Boards muss neben dem Pipelinedesign auch ein Schnittstellen-Design für den Interface-FPGA entwickelt werden. Dies ist sehr aufwändig. Um die Programmierung zu vereinfachen, wurde deshalb die Software PGR speziell für die PROGRAPE-Boards entwickelt.

3.4.2 Entwicklung eines FPGA-basierten Rechners

Die Entwicklung eines FPGA-Koprozessors zur Beschleunigung eines Algorithmus ist sehr aufwändig. Grundsätzlich lässt sich der Ablauf in folgende fünf Schritte zerlegen:

(A) **Zielfunktion:**
Zunächst wird die Zielfunktion spezifiziert, das Problem, das mit Hilfe des Koprozessors beschleunigt werden soll. In der Regel ist dies ein Algorithmus, dessen Softwareimplementierung bereits existiert. Aus dieser Implementierung gilt es die zeitintensiven Teilbereiche ausfindig zu machen, die sich für eine Berechnung auf dem FPGA eignen. Außerdem werden in diesem Schritt die Ein- und Ausgabedaten für den Koprozessor definiert, respektive die Festlegung des Zahlenformats, der Wortlängen für Daten und arithmetische Operationen und die Beschreibung des Datenflusses für die Berechnung der Zielfunktion.

(B) **Softwareemulator:**
In diesem Schritt wird ein Softwareemulator entwickelt. Mit Softwareemulator ist hier die Softwareimplementierung der Zielfunktion gemeint. Bei der Extrahierung von Teilbereichen zur Berechnung auf dem FPGA kann es notwendig sein, die ursprüngliche Implementierung der Zielfunktion zu modifizieren. Mit diesem Emulator wird also getestet ob der geänderte Algorithmus immer noch korrekt arbeitet, d. h. ob das in (A) spezifizierte Design die Zielfunktion korrekt berechnet. Mit dem Emulator können später außerdem Vergleichsrechnungen durchgeführt werden, um zu überprüfen, ob das FPGA-basierte Hardwaredesign korrekt

und mit der gewünschten Genauigkeit rechnet. Des Weiteren wird hier die Schnittstelle (API) zwischen Wirtsrechner und Koprozessor definiert.

(C) **Hardwaredesign:**
Als nächstes erfolgt die Implementierung des Hardwaredesigns für den Koprozessor in einer Hardwarebeschreibungssprache. Dies beinhaltet nicht nur das Design für die FPGAs, sondern auch die Definition der Kontrolllogik und der Schnittstelle zum Wirtssystem. Aus diesem Quelltext wird wie in Kapitel 3.3.2 beschrieben der Bitstrom zur Konfiguration der FPGAs bzw. des Koprozessors erzeugt.

(D) **API:**
Damit das Wirtssystem mit dem FPGA-Board kommunizieren kann, muss auf dem Wirtssystem eine Schnittstellensoftware implementiert werden. Diese regelt die Kommunikation und Datenübertragung zwischen dem Wirtssystem und dem Koprozessor. Die Schnittstellensoftware sollte außerdem eine eventuell nötige Konvertierung der Datenformate übernehmen.

(E) **FPGA-basierter Rechner:**
Schließlich erhält man durch die Verbindung von Hardware, Bitstrom für die Hardwarekonfiguration, Schnittstelle und Anwenderprogramm den FPGA-basierten Rechner.

Zu einem FPGA-basierten Rechner gehört nicht nur die Entwicklung einer Konfiguration für die Hardware, sondern auch Softwareimplementierungen in Form des Emulators und der API. Diese Software- sowie Hardwaredesigns müssen entwickelt, getestet und von Fehlern befreit werden, was sehr zeitaufwändig ist.

Sicherlich lassen sich einige Designs und Algorithmen wiederverwenden oder auch kaufen. Trotzdem ist es immer von Nöten, ein detailliertes Wissen über Hardware und HDL zu haben, um zu verstehen, wie solche Designs verwendet werden.

In Abbildung 3.7 ist erkennbar, dass alle Hardware- und Softwaredesigns auf der Definition der Zielfunktion basieren. Es sollte also für eine intelligente Software möglich sein, alle nötigen Hardware- und Softwarecodes ausgehend von der Zielfunktion, die mit einer speziell für diese Software entwickelten Beschreibungssprache beschrieben ist, zu erzeugen. Eine Software, die dies speziell für das PROGRAPE-4-Board realisiert ist PGR. Das Prinzip von PGR wird im folgenden Kapitel genauer beschrieben.

Kapitel 4

PGR

Aus Kapitel 2.4.4 geht hervor, dass der Einsatz von FPGAs zur Beschleunigung teilchenbasierter Systeme sinnvoll ist. Dabei wurde jedoch außer Acht gelassen, ob sich ein entsprechendes Design für den FPGA bzw. den FPGA-Koprozessor relativ einfach und in nicht allzu langer Zeit entwickeln lässt. Kapitel 3.4.2 beschreibt dagegen, wie aufwändig es ist, ein solches Codesign zu erzeugen.

Es ist sicherlich ein Erfolg, wenn man mit FPGAs die Effizienz eines Algorithmus steigern kann. Jedoch gehört zum Kostenfaktor nicht nur der Verbrauch an Rechenressourcen, sondern auch der Entwicklungsaufwand. Leider liegt hier genau das Problem bei der Verwendung von FPGAs. Um beispielsweise eine einfache Pipeline für gravitative Wechselwirkungen (B.1) zu entwickeln, benötigt man mehr als ein Personenjahr [13]. Für Anwendungen, die komplexere Hardwaredesigns erfordern, benötigt man dementsprechend mehr Entwicklungsarbeit. Der Aufwand und die Komplexität der Entwicklung eines FPGA-Koprozessors setzt also dem praktischen Gebrauch von FPGAs klare Grenzen [13].

Anwender arbeiten in der Regel softwareorientiert. Die Schwierigkeit der FPGA-Programmierung liegt deshalb darin, dass sie zusätzlich über sehr detaillierte Hardwarekenntnisse verfügen müssen oder aber einen Hardwarespezialisten zurate ziehen müssen, der wiederum zunächst in das Themengebiet eingearbeitet werden muss.

Um auch Softwareentwicklern die Programmierung von FPGAs zu ermöglichen, wurden Programmiersprachen wie Mitrion-C, Impulse-C, Handel-C, dime Talk/dime-C [23] und PGDL entwickelt. Diese Programmiersprachen sind von der Syntax her sehr C-ähnlich und haben zum Ziel, dass sich der Benutzer auf den eigentlichen Algorithmus und nicht auf die komplexen Details der Hardwareimplementierung konzentrieren kann.

Im Rahmen dieser Diplomarbeit wurde die Software PGR verwendet, um Hardwaredesigns für das PROGRAPE-4-Board zu erzeugen. Die Funktionsweise von PGR wird im Folgenden genauer beschrieben. Außerdem wird auf die Beschreibungssprache PGDL zur Erzeugung des von PGR übersetzbaren Quellcodes eingegangen.

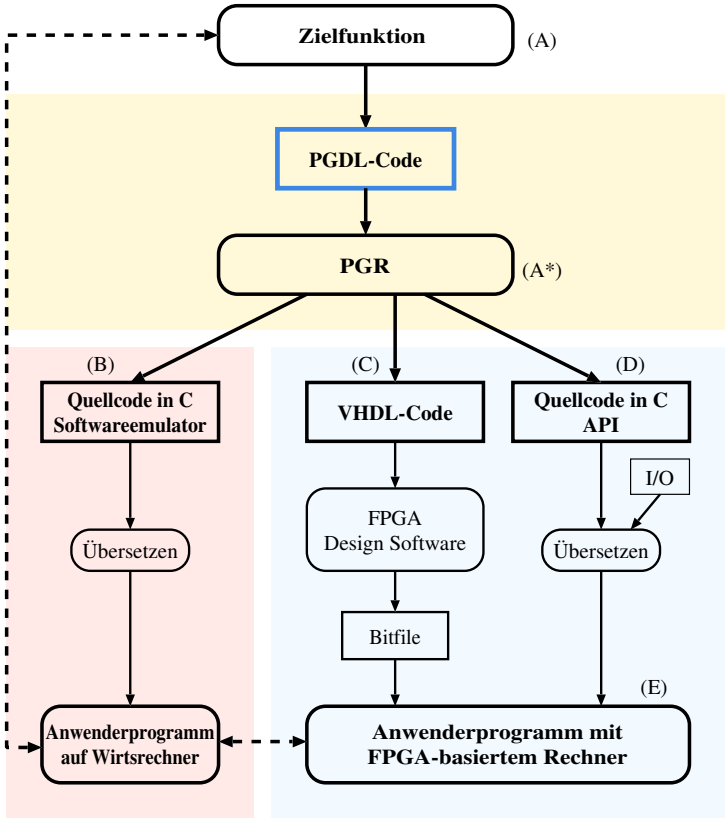


Abbildung 4.1: Entwicklungsablauf eines FPGA-basierten Rechners mit PGR

4.1 FPGA-Programmierung mit PGR

PGR (***P**rocessors **G**enerator for **R**econfigurable Systems*) ist ein Softwarepaket das von Naohito Nakasato und Tsuyoshi Hamada am RIKEN entwickelt wurde. Die Software zielt insbesondere auf die Bedürfnisse von Forschern aus der Physik, Astrophysik, Chemie und Biotechnologie ab. Sie richtet sich also nicht an Hardwarespezialisten, sondern an softwareorientierte Anwender [16].

PGR ist zur Zeit auf einem Linux-Rechner am Astronomischen Recheninstitut (ARI) in Heidelberg installiert. Dieser Rechner besitzt ein PROGRAPE-4-Board. Ursprünglich sollte die Cray XD1 des Forschungszentrums Jülich im Rahmen dieser Diplomarbeit als Zielsystem dienen. PGR ist jedoch auf der Cray XD1 noch nicht lauffähig. Deshalb liegt allen weiteren Betrachtungen in diesem Kapitel das System in Heidelberg zugrunde.

4.1.1 Entwicklungsablauf in PGR

Abbildung 4.1 zeigt den Entwicklungsablauf für einen FPGA-basierten Rechner mit PGR. Nach der Spezifikation der Zielfunktion beschreibt man diese mit der Beschreibungssprache PGDL (***PGR D**escription **L**anguage*). Ausgehend von dieser Beschreibung erzeugt PGR alle nötigen Hardware- und Softwaredesigns. Zum einen erzeugt es den Softwareemulator, der es ermöglicht, vor der Konfiguration des FPGAs die beschriebene Funktion auf ihre Korrektheit hin zu überprüfen. Zum anderen werden der VHDL-Code und eine Schnittstelle automatisch generiert. Für die Erzeugung der Designs sind folgende vier Komponenten verantwortlich:

1. **Parser:** Der Parser analysiert zunächst den PGDL-Code und erzeugt eine Datei in interner Darstellung, die für die weiteren Schritte als Eingabe dient.

2. **Modulgenerator:** Als nächstes generiert der Modulgenerator für sämtliche arithmetischen Module den entsprechenden VHDL-Code und den Quellcode für den Softwareemulator.

3. **Datenflussgenerator:** Durch Einfügen von Verzögerungsregistern wird der Datenfluss zwischen den einzelnen Operationen geregelt.

4. **Compiler:** Zuletzt erzeugt der Compiler den top-level VHDL-Code, den Quellcode für die VHDL-Bibliotheken, die Schnittstelle und den Softwareemulator.

Die Schnittstelle zwischen Wirtssystem und FPGA-Board und der Softwareemulator sind in C geschrieben. Beide besitzen eine spezielle top-level Funktion, die in beiden Fällen exakt die selben Argumente erwartet. Dies ist ein weiterer Vorteil von PGR. Es ermöglicht dem Benutzer zwischen Emulation und tatsächlicher Rechnung auf dem FPGA beliebig zu wechseln, ohne dabei Änderungen am Anwenderprogramm vornehmen zu müssen. Es müssen lediglich die entsprechenden Bibliotheken hinzugelinkt werden [16].

4.1.2 Parametrisierte arithmetische Module

Die Basis der Programmierung mit PGR bilden die parametrisierten arithmetischen Module (AM). Die aktuelle Version von PGR unterstützt etwas über 30 verschiedene AMs. Jedes AM realisiert eine arithmetische Funktion wie Addition, Multiplikation, Wurzel usw., wobei diese in drei verschiedenen Formaten implementiert sind: Festkomma, Fließkomma und logarithmisches Format. Für jedes parametrisierte AM lassen sich außerdem die Bitlänge der Daten, sowie die Anzahl der Pipelinestufen angeben. Zur Zeit sind parametrisierte AMs für die Addition, Subtraktion, Multiplikation, Division, Wurzelfunktion und If-Anweisungen verfügbar. Funktionen wie die Exponentialfunktion oder Winkelfunktionen, die für molekulardynamische Simulationen wünschenswert wären, sind leider noch nicht implementiert. Eine Liste der momentan verfügbaren Module befindet sich im Anhang (A).

Parametrisierte AMs werden mit PGDL wie folgt angesprochen:

```
pg_float_add(x, y, z, 26, 16, 1);
```

In diesem Beispiel wird die Addition $z = x + y$ realisiert, wobei x und y die Eingabe- und z der Ausgabeparameter sind. Das Präfix `pg_float` kennzeichnet die Verwendung des Fließkommaformats.

Das interne Fließkommaformat ist so definiert, dass ein Bit das Vorzeichenbit ist, ein weiteres Bit wird zur Kennzeichnung einer Zahl als gleich bzw. ungleich Null verwendet, m Bits werden für den Exponenten und n für die Mantisse reserviert. Die gesamte Bitlänge einer Fließkommazahl ist also immer $m+n+2$, wobei Mantisse und Exponent entsprechend dieser Länge variiert werden können. Die maximale Bitlänge entspricht zur Zeit 33. Wählt man also $m = 8$ und $n = 23$ entspricht dies der IEEE-Fließkommadarstellung mit einfacher Genauigkeit.

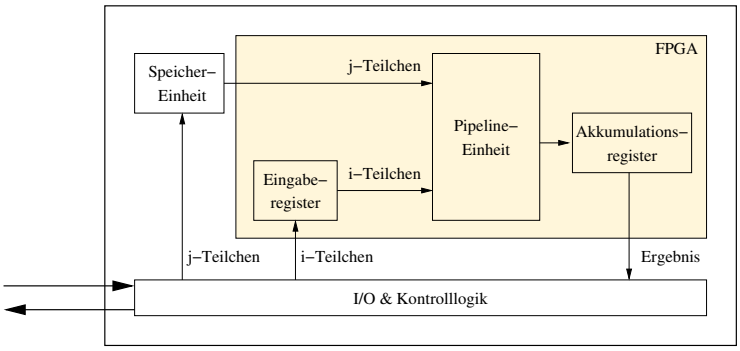


Abbildung 4.2: Blockdiagramm des Pipelineprozessors, der mit PGDL generiert wird

Im obigen Beispiel liegt ein Fließkommaformat mit einer gesamten Bitlänge von 26 Bits vor, wobei 16 Bits für die Mantisse reserviert sind. Das letzte Argument (hier die 1) gibt die Anzahl der Pipelinestufen an. Die Geschwindigkeit der Operation steigt mit der Anzahl der Pipelinestufen. Eine größere Anzahl von Pipelinestufen bedeutet aber auch eine Vergrößerung des FPGA-Designs. Die Kapazität des FPGAs setzt der Anzahl der Pipelinestufen also eine Grenze. Zudem sind AMs nur für eine begrenzte Anzahl an Pipelinestufen implementiert. Es gilt also immer, eine passende Anzahl an Pipelinestufen zu finden, so dass das Design nicht zu langsam, aber auch nicht zu groß wird.

Module, die mit dem Präfix `pg_log` beginnen rechnen mit dem logarithmischen Zahlensystem (LNS), d. h. eine reelle Zahl x wird als Festkommazahl y des Logarithmus von x zur Basis 2 ($x = 2^y$) dargestellt. Diese Darstellung hat den Vorteil, dass sich Multiplikationen und Divisionen mit Hilfe des Logarithmus auf Additionen und Subtraktionen zurückführen lassen. Das führt bei häufigem Gebrauch zu einer Reduzierung der Hardwareressourcen [16, 20]. In diesem Format wird wiederum jeweils ein Bit für das Vorzeichen und die Kennzeichnung der Null verwendet. Die gesamte Bitlänge beträgt also auch hier $m+n+2$, wobei wieder m die Anzahl der Bits für den Exponenten und n die Anzahl der Bits für die Mantisse bezeichnen.

Module, die im Festkommaformat rechnen wie `pg_fix_addsub` oder `pg_fix_accum` werden mit dem Präfix `pg_fix` gekennzeichnet. Auch hier lässt sich wieder die Bitlänge beliebig wählen [13, 16].

4.2 Die Beschreibungssprache PGDL

PGDL ist besonders darauf ausgerichtet, Pipelineprozessoren für teilchenbasierte Simulationen zu entwickeln. Allgemein lässt sich sagen, dass Probleme, die die Form

$$f_i = \sum_j G(a_i, a_j) \tag{4.1}$$

haben, mit PGDL beschrieben werden können.

Abbildung 4.2 zeigt das Blockdiagramm eines solchen Pipelineprozessors. Dieser besteht aus der Kontroll- und I/O-Logik, programmspezifischen Registern, einer Speicher- und einer Pipelineeinheit.

Das Kernstück dieses Prozessors ist die Pipelineeinheit. Sie gehört zusammen mit den Registern zu dem Teil des Prozessors, der auf den FPGAs des PROGRAPE-Boards realisiert wird. Die Pipelineeinheit berechnet die Funktion G aus Gleichung 4.1. Die Beschreibung der Pipeline geschieht mit Hilfe der parametrisierten AMs. Es gibt zwei Arten von Registern, Eingaberegister und Akkumulationsregister. In den Eingaberegistern befinden sich die Daten der i-Teilchen (a_i) aus Gleichung 4.1. Sie werden zu Beginn einer Berechnung geladen und verbleiben dort. Eine Berechnung bezeichnet dabei die Berechnung einer Komponente des Vektors f aus Gleichung 4.1. In den Akkumulationsregistern werden die Teilergebnisse $G(a_i, a_j)$ aufaddiert und gespeichert, so dass sie am Ende einer Berechnung das Ergebnis f_i enthalten.

Die Speichereinheit und die Kontroll- und I/O-Logik sind nicht Teil der FPGAs, sie befinden sich auf dem PROGRAPE-Board. In der Speichereinheit befinden sich die Daten der j-Teilchen. In jedem Takt wird dabei ein neues Datum an die Pipelineeinheit der FPGAs geschickt, solange bis die Daten aller Teilchen übermittelt wurden und eine Berechnung beendet ist. Dieser Vorgang wird für die Berechnung aller Komponenten von f wiederholt. Speichereinheit und Register können in PGDL vom Benutzer individuell zugewiesen werden. Die Kontroll- und I/O-Logik wird im Interface-FPGA des PROGRAPE-Boards realisiert. Sie wird von PGR automatisch erzeugt.

4.2.1 Das API-Modell

Wie bereits im vorherigen Kapitel erwähnt, erzeugt PGR neben dem VHDL-Code auch eine Schnittstelle zum Anwenderprogramm, die die Kommunikation mit dem FPGA-Board sowie die Datenübertragung und -konvertierung regelt. Diese Schnittstelle wird mit der `force`-Routine realisiert. Sie hat folgende Signatur:

```
void force(<arguments>)
```

Diese Funktion ist eine C-Routine, die vom Anwenderprogramm aus aufgerufen werden kann. Der Datenaustausch zwischen Anwenderprogramm und FPGA-Board geschieht über die Funktionsparameter. Das Besondere dieser Schnittstellenfunktion ist die Tatsache, dass der Funktionsprototyp für den Emulator und den FPGA identisch ist. Der Benutzer kann also bequem, ohne Änderungen an seiner Applikation vornehmen zu müssen, zwischen Emulation und tatsächlicher Rechnung auf dem FPGA-Board wechseln [13].

4.2.2 Programmstruktur

In diesem Abschnitt werden nun kurz die Struktur und die grundlegenden Elemente eines PGDL-Programms beschrieben. Dazu sei das folgende kleine Beispiel gegeben:

$$f_i = \sum_j a_i \cdot a_j \tag{4.2}$$

Listing 4.1 zeigt den zu Formel 4.2 gehörigen PGDL-Code. Wie erkennbar ist, besteht ein PGDL-Programm aus vier Teilen:

```
1  /* ------------------------------------ Makrodeklarationen */
2  #define NFLO 26   // Bitlänge einer Fließkommazahl
3  #define NMAN 16   // Bitlänge der Mantisse
4  #define NFIX 57   // Bitlänge einer Festkommazahl
5  #define NACC 64   // Bitlänge der Akkumulation
6  #define NEXAD 31
7  #define FSHIFT pow(2.0,(double)NEXAD)
8
9  /* ------------------------------ allgemeine Deklarationen */
10 /NVMP 1;
11 /NPIPE 2;
12
13 /* ------------------------------ Schnittstellen-Definition */
14 /MEM aj    <= a[]   : float(NFLO, NMAN);
15 /REG ai    <= a[]   : float(NFLO, NMAN);
16 /REG fi    => f[]   : fix(NACC);
17
18 /SCALE fi : (1.0/FSHIFT);
19
20 /* ------------------------------------ Pipelinebeschreibung */
21 pg_float_mult       (ai,aj,aij,     NFLO,NMAN, 2);
22 pg_float_expadd     (aij,aij_fix,   NEXAD,NFLO,NMAN,1);
23 pg_float_fixaccum   (aij_fix,fi,    NFLO,NMAN, NFIX, NACC, 2);
```

Listing 4.1: PGDL-Code zu Formel 4.2

1. **Makrodeklarationen:**
   Wie in einem C-Programm lassen sich auch hier Makros für häufig verwendete Konstanten und Ausdrücke definieren. Die Syntax entspricht der C-Syntax. Die Makrodeklarationen werden wie bekannt vom C-Präprozessor abgearbeitet. In dem betrachteten Code-Beispiel werden Makros für die Bitlängen der parametrisierten AMs definiert. `NEXAD` und `FSHIFT` werden für die Umwandlung von Fließkomma- in Festkommaformat und umgekehrt benötigt.

2. **Allgemeine Deklarationen:**
   In diesem Teil befinden sich Deklarationen, die sich auf die Konfiguration des FPGAs beziehen. `NVMP` bezeichnet hierbei die Anzahl der virtuellen Pipelines und `NPIPE` die Anzahl der physikalischen Pipelines. Der Unterschied zwischen virtueller und physikalischer Pipeline besteht darin, dass die physikalische Pipeline real in der Hardware existiert. In Listing 4.1 ist die Anzahl der physikalischen Pipelines zwei, d. h. auf einem FPGA werden zwei Pipelines untergebracht. Wie bei den AMs gilt auch hier, je größer die Anzahl der Pipelines ist, desto schneller ist das Design. Verglichen mit einer einzelnen Pipeline, ist dasselbe Design mit zwei Pipelines doppelt so schnell, mit dreien dreimal so schnell, usw. Der Anzahl der Pipelines sind jedoch Grenzen gesetzt. Je komplexer ein Design ist, desto weniger Pipelines passen auf einen FPGA. Die maximale Anzahl der Pipelines hängt außerdem von dem verwendeten Werkzeug zur Bitstrom-Generierung ab. Verwendet man *Synplify* so lassen sich eventuell mehr parallele Pipelines realisieren als mit ISE, da *Synplify* über einen höheren Grad der Optimierung verfügt.
   Die Realisierung virtueller Pipelines ist ein Plan für die Zukunft und momentan noch nicht möglich. Im PGDL-Code wird die Anzahl deshalb immer auf eins gesetzt. Die Idee der virtuellen Pipeline ist, dass pro Takt über eine physikalische Pipeline mehrere (je nach Anzahl der virtuellen Pipelines) Berechnungen nebeneinander laufen. In einem Taktzyklus werden dann von einer physikalischen Pipeline mehrere Komponenten des Vektors f parallel berechnet.

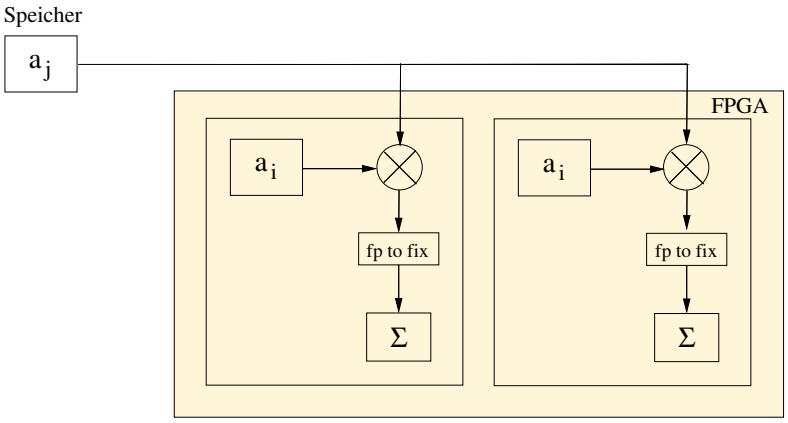


Abbildung 4.3: Blockdiagramm zu Formel 4.2

3. **Schnittstellen-Definition:**
   In den Zeilen 14 - 16 wird die Signatur der `force`-Routine definiert. Sie sieht für das betrachtete Beispiel wiefolgt aus:

   ```
   void force(double a[], double f[], int n)
   ```

   Der Vektor `a` enthält die Eingabedaten und in `f` werden die Ergebnisse gespeichert. Im PGDL-Code werden Eingabedaten durch einen Pfeil nach rechts (<=) und Ausgabedaten durch einen Pfeil nach links (=>) gekennzeichnet. Das letzte Argument `n` gibt die Länge der Vektoren `a` und `f` an. Durch `REG` und `MEM` wird festgelegt, ob die jeweiligen Daten in die Speichereinheit oder in die Register direkt auf den FPGAs geschrieben werden. In diesem Fall werden die Daten des Vektors `a` einmal komplett in den Speicher und ins Register geschrieben. Der Befehl `SCALE` (Zeile 18) dient dazu, den Dezimalpunkt einer Zahl zu verschieben. Dies ist notwendig, da die Fließkommazahlen vor der Akkumulation in Festkommazahlen umgewandelt werden. Da `f` ein Vektor mit Fließkommaelementen ist, muss das Ergebnis der Akkumulation wieder in die Fließkommadarstellung konvertiert werden.

4. **Pipelinebeschreibung:**
   Die letzten drei Zeilen von Listing 4.1 enthalten die Beschreibung der Pipeline. In Zeile 21 erfolgt zunächst die Multiplikation der Daten von Teilchen *i* und *j*. Da die Akkumulation zur Zeit nur für Festkommazahlen implementiert ist, wird das Produkt in Zeile 22 in ein Festkommaformat konvertiert. Dies geschieht, indem der Dezimalpunkt durch Multiplikation mit 2^{31} verschoben wird. In Zeile 23 wird diese Zahl dann auf das bisherige Ergebnis aufaddiert. Das Ergebnis, der Kräftevektor `f`, ist dementsprechend auch im Festkommaformat abgespeichert. Das Anwenderprogramm erwartet jedoch einen Vektor mit Elementen in Fließkommadarstellung. Aus diesem Grund wird mit dem Befehl `SCALE` das Komma wieder zurück verschoben.

In Abbildung 4.3 ist das Blockdiagramm des zu Formel 4.2 generierten Designs dargestellt. Wie in Zeile 11 definiert, werden zwei parallele Pipelines erzeugt. Die Daten der *i*-Teilchen werden in den Registern des FPGAs gespeichert und verbleiben dort während einer Berechnung unverändert. Nach Beendigung der *i*-ten Berechnung werden die Daten des nächsten Teilchens auf den FPGA geladen. Die Daten der *j*-Teilchen befinden sich in der Speichereinheit. In jedem Taktzyklus wird ein neues *j*-tes Datum an die Pipelines des FPGA übermittelt, solange bis alle *N* Teilchen durchlaufen wurden. Das Endergebnis der *i*-ten Berechnung, sowie alle Zwischenergebnisse für f_i, werden ebenfalls auf dem FPGA gespeichert und anschließend an das Wirtssystem übermittelt.

Kapitel 5

Grundlagen zum Einsatz von FPGAs

In Kapitel 2 wurde die Vorgehensweise einer molekulardynamischen Simulation beschrieben. Dabei wurde deutlich, dass 95% der Laufzeit einer MD-Simulation für die Berechnung der Wechselwirkungen benötigt wird. Würde man also die Kräfteberechnung optimieren bzw. beschleunigen, würde auch die gesamte Simulationszeit deutlich verkürzt. Eine Beschleunigung der Kräfteberechnung lässt sich unter anderem durch den Einsatz von FPGAs erreichen. In Kapitel 3 wurde dazu das FPGA-basierte PROGRAPE-4-Board vorgestellt, das speziell für die beschleunigte Berechnung von Vielteilchensystemen entwickelt wurde. Zur einfachen Entwicklung eines FPGA-Koprozessordesigns für das PROGRAPE-Board wurde schließlich in Kapitel 4 die Software PGR vorgestellt.

Aus den Kapiteln 2, 3 und 4 erhält man also:

- einen **Algorithmus**, dessen Berechnung es zu beschleunigen gilt,
- die **Hardware**, mit deren Hilfe die Berechnung des Algorithmus beschleunigt werden kann, und
- die passende **Software**, die es ermöglicht diese beiden Elemente zusammen zu führen bzw. aufeinander abzustimmen.

Dieses Kapitel ist eine Einführung in die Verwendung von FPGAs für molekulardynamische Rechnungen. Diese Arbeit beschäftigt sich dabei speziell mit den Möglichkeiten, die das PROGRAPE-4-Board in Zusammenhang mit der Software PGR bietet. Die folgenden Ergebnisse beziehen sich immer auf das PROGRAPE-4-Board und lassen sich nicht auf andere FPGA-Architekturen und -Programmiersprachen verallgemeinern.

Als Motivation werden zunächst Ergebnisse einer Projektarbeit vorgestellt, in der FPGAs zur Berechnung von Gravitationskräften eingesetzt wurden [7]. Anschließend wird als Vorarbeit ein einfaches FPGA-Design zur Kräfteberechnung mittels eines Lennard-Jones-Potentials entwickelt. Dabei wird auf einige Probleme und Besonderheiten eingegangen. Außerdem werden Performance- und Genauigkeitstests durchgeführt.

5.1 Motivation

Im Rahmen einer Projektarbeit von A. Ernst am ARI in Heidelberg wurden FPGAs für die Beschleunigung astrophysikalischer Berechnungen eingesetzt. Ausgangspunkt war der Code NBody zur Simulation von Sternen einer Galaxie. Der rechenintensivste Teil einer NBody-Simulation ist die Berechnung der Gravitationskräfte.

$$\vec{f}_i = \sum_{j \neq i} m_j \frac{\vec{r}_{ij}}{r_{ij}^3} \tag{5.1}$$

$\vec{f}_i$: Gravitationskraft auf Teilchen i (bestehend aus der Kraftkomponente in x-, y- und z-Richtung)
m_j : Masse des Teilchens j
$\vec{r}_{ij}$: Verbindungsvektor zwischen Teilchen i und j ($\vec{r}_j - \vec{r}_i$)
r_{ij} : Abstand zwischen Teilchen i und j ($\|\vec{r}_i - \vec{r}_j\|_2$)

Wie aus Gleichung 5.1 erkennbar ist, handelt es sich um ein langreichweitiges Potential, denn das System ist dreidimensional und das Potential fällt nicht schneller als r^{-3} gegen Null ab (siehe Kapitel 2.2.2). Zur Berechnung der Gesamtkraft, die auf ein Teilchen wirkt, bzw. in diesem Fall einen Stern, müssen die Einwirkungen aller anderen Sterne im System betrachtet werden. Dies führt zu einem $O(N^2)$-Problem.

Für den Einsatz der FPGAs wurde der Code so umgeschrieben, dass die Berechnung der Gravitationskräfte auf die FPGAs verlagert wurde. Dazu wurde von N. Nakasato und T. Hamada mit PGR eine Pipeline zur Berechnung der Gravitation nach Gleichung 5.1 entwickelt. Der PGDL-Code zur Beschreibung dieser Pipeline ist etwa 25 Zeilen lang (siehe Anhang B.1). Aus diesem Code wurden passend für die Virtex-II-Pro FPGAs (genauer Typ: XC2VP50) der Cray XD1 in Jülich Bitfiles erzeugt. Ein Virtex-II-Pro FPGA besitzt 5.904 CLBs. Damit passen auf einen Virtex-II-Pro FPGA maximal 10 parallele Gravitationspipelines.

Das FPGA-Design hat eine Taktrate von 120 MHz und rechnet mit einer Bitlänge von 26 Bit für Fließkommazahlen. Insgesamt besitzt es 28 arithmetische Module. Messungen mit einem Prozessor und 10 parallelen Pipelines auf einem FPGA ergaben eine Arbeitsleistung von 27 GFlops. Im Vergleich dazu lieferte die Berechnung ohne FPGA lediglich eine Leistung von 0,76 GFlops [7]. Die Arbeitsleistung ließ sich also durch den Einsatz von FPGAs um einen Faktor 35 steigern.

5.2 Lennard-Jones-FPGA-Design

Die Ergebnisse der Projektarbeit von A. Ernst motivierten dazu, FPGAs auch für molekulardynamische Berechnungen einzusetzen. In einem ersten Schritt wurde deshalb zunächst ein Pipelinedesign zur Kräfteberechnung nach dem Lennard-Jones-Potential entworfen.

5.2.1 Kraftberechnung in Fortran

Die Energie eines Teilchens i berechnet sich mit dem Lennard-Jones-Potential wie folgt:

$$U_i = 4\varepsilon \cdot \sum_{j \neq i} \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{5.2}$$

ε : Energiekonstante
σ : Durchmesser eines Teilchens
r_{ij} : Abstand zwischen Teilchen i und j ($\|\vec{r}_i - \vec{r}_j\|_2$)

Durch Ableiten von Gleichung 5.2 erhält man die Kraft, die auf Teilchen i einwirkt:

$$\vec{f}_i = -\nabla U_i = 24\varepsilon \cdot \sum_{j \neq i} \left(2\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \cdot \frac{\vec{r}_{ij}}{|r_{ij}|^2} \tag{5.3}$$

ε : Energiekonstante
σ : Durchmesser eines Teilchens
$\vec{r}_{ij}$: Verbindungsvektor zwischen Teilchen i und j ($\vec{r}_j - \vec{r}_i$)
r_{ij} : Abstand zwischen Teilchen i und j ($\|\vec{r}_i - \vec{r}_j\|_2$)

Die Kraftberechnung nach Gleichung 5.3 wird im Fortran-Programm durch eine Doppelschleife realisiert.

```
1  do i = 1, N-1
2     do j = i+1, N
3        r = 0.0
4        do k = 1, dim
5           dr(k) = x(k,i) - x(k,j)
6           r = r + dr(k)**2
7        enddo
8        r = sqrt(r)
9
10       do k = 1, dim
11          df = 4.0*epsilon * dr(k)/r**2 *&
12                ( 12*(sigma/r)**12 - 6*(sigma/r)**6 )
13          f(k,i) = f(k,i) + df
14          f(k,j) = f(k,j) - df
15       enddo
16    enddo
17 enddo
```

Listing 5.1: schematische Doppelschleife in Fortran zur Berechnung der Kraft auf alle N Teilchen eines Systems mit dem Lennard-Jones-Potential

Wie in Zeile 2 von Listing 5.1 erkennbar, läuft die innere Schleife nicht über alle N j-Teilchen. Sie läuft lediglich über die Teilchen, deren Index größer als der des momentan betrachteten i-Teilchens ist. Der Grund dafür ist die Ausnutzung des dritten Newtonschen Axioms: $F_{ij} = -F_{ji}$. Die Kraft zwischen einem Teilchenpaar i und j wird also nur einmal berechnet und der Kraftbetrag für Teilchen i aufaddiert (Zeile 13) und für Teilchen j von der bereits berechneten Kraft abgezogen (Zeile 14).

5.2.2 FPGA-Implementierung

Zur Berechnung der Kräfte auf dem PROGRAPE-4-Board sind einige Dinge zu beachten. Zum einen lässt sich das dritte Newtonsche Axiom auf dem FPGA-Board nicht ausnutzen. Der Grund dafür ist, dass PGDL keinen direkten Zugriff auf die Schleifenindizes ermöglicht. PGR generiert automatisch eine Doppelschleife, die über alle *i*- und *j*-Teilchen läuft. Zeile 2 und Zeile 14 von Listing 5.1 sind daher mit PGR nicht realisierbar.

Weiter gilt es zu beachten, dass die Anzahl der Teilchen, die gleichzeitig auf das FPGA-Board geladen werden können, begrenzt ist. Dies liegt an der beschränkten Speicherkapazität des Boards. Unter Verwendung des PROGRAPE-4-Boards und PGR beträgt die maximale Anzahl der Teilchen, die gleichzeitig auf das Board geladen werden können, 16.384. Es können also maximal 16.384 *i*-Teilchen und 16.384 *j*-Teilchen gleichzeitig geladen werden. Zu beachten ist, dass sich die Anzahl weiter verringert, je komplexer das FPGA-Design wird. In der Molekulardynamik ist man jedoch daran interessiert, Systeme mit einer größeren Anzahl Teilchen zu simulieren. In diesem Fall muss die Berechnung der Kräfte in Etappen erfolgen.

Des Weiteren ist es sinnvoll, die Pipeline auf dem FPGA möglichst einfach zu halten, um mehrere Pipelines auf einen FPGA unterbringen zu können. Je höher die Parallelität, desto schneller arbeitet das Design. Aus diesem Grund wurde Gleichung 5.3 wie folgt umgeformt:

$$\vec{f}_i = \sum_{j \neq i} \left(\left(\frac{4\varepsilon 12\, \sigma^{12}}{r_{ij}^{14}} - \frac{4\varepsilon\, 6\, \sigma^{6}}{r_{ij}^{8}} \right) \cdot \vec{r}_{ij} \right) = \sum_{j \neq i} \left(\left(\frac{a}{r_{ij}^{14}} - \frac{b}{r_{ij}^{8}} \right) \cdot \vec{r}_{ij} \right) \tag{5.4}$$

mit $a = 4\varepsilon 12\, \sigma^{12}$, $b = 4\varepsilon\, 6\, \sigma^{6}$

ε : Energiekonstante
σ : Durchmesser eines Teilchens
$\vec{r}_{ij}$: Verbindungsvektor zwischen Teilchen i und j ($\vec{r}_j - \vec{r}_i$)
r_{ij} : Abstand zwischen Teilchen i und j ($\|\vec{r}_i - \vec{r}_j\|_2$)

`a` und `b` bleiben während der gesamten Kräfteberechnung konstant. Sie können deshalb zuvor vom Programm berechnet und als Parameter an das FPGA-Board übergeben werden. Es wäre nicht effizient, `a` und `b` bei jeder FPGA-Berechnung aufs Neue zu berechnen. Sie werden, wie die Daten der *i*-Teilchen, zu Beginn der Berechnung der Gesamtkraft auf Teilchen *i* in die Register geladen.

Aus dem PGDL-Code für die Berechnung der Kräfte mit dem Lennard-Jones-Potential (siehe Anhang B.2) erhält man folgende Schnittstelle für das PROGRAPE-Board:

```
void force(double xjj[][3], double xii[][3], double a,
           double b, double f[][3], int ni, int nj)
```

Die Matrix `xjj` enthält die Positionen der *j*-Teilchen, `xii` die der *i*-Teilchen. Für jedes der *N* Teilchen besteht die Position aus der x-, y- und z-Koordinate. `a` und `b` sind Parameter mit den Werten aus Gleichung 5.4. Die Matrix `f` enthält nach der Berechnung auf dem FPGA-Board die Teilkräfte der *i*-Teilchen. Die Kraft besteht jeweils aus der Kraftkomponente in x-, y- und z-Richtung. `ni` enthält die Anzahl der *i*-Teilchen und `nj` die Anzahl der *j*-Teilchen.

Eine Besonderheit von PGR sei hier noch kurz erwähnt. PGR unterstützt implizite Vektoroperationen. `xii` und `xjj` sind in diesem Beispiel keine einfachen Vektoren, sondern jedes Vektorelement besteht aus drei Komponenten. Um die Differenz `xii(i)-xjj(i)` zu bilden, deren Ergebnis

ebenfalls ein Vektor ist, sind also drei Operationen notwendig. In PGDL wird diese Differenz durch die folgende Anweisung beschrieben:

```
pg_float_sub(xi, xj, dr, NFLO,NMAN,NST_SUB);
```

`xi` und `xj` sind im PGDL-Code als dreielementige Vektoren definiert. PGR erzeugt aus diesem Grund automatisch drei arithmetische Module. Das Ergebnis `dr` der Differenz ist wieder ein Vektor mit drei Elementen.

Listing 5.2 zeigt den Fortran-Code, der der Berechnung der Kraft auf dem FPGA-Board entspricht. Wie bereits erwähnt, generiert PGR eine Doppelschleife, die über alle *i*- und *j*-Teilchen läuft. Das dritte Newtonsche Axiom wird nicht ausgenutzt.

```
1  do i = 1, ni
2     do j = 1, nj
3        if (i /= j) then
4           r = 0.0
5           do k = 1, dim
6              dr(k) = xii(k,i) - xjj(k,j)
7              r = r + dr(k)**2
8           enddo
9           r = sqrt(r)
10
11          do k = 1, dim
12             df = ((a/r**14) - (b/r**8)) * dr(k)
13             f(k,i) = f(k,i) + df
14          enddo
15       endif
16    enddo
17 enddo
```

Listing 5.2: Doppelschleife in Fortran, die der Berechnung der Kräfte auf dem FPGA-Board entspricht

Listing 5.3 zeigt die Berechnung der Kräfte mit dem FPGA-Board. Zunächst werden die Konstanten `a` und `b` berechnet (Zeile 1 und 2). Die Funktion `force` stellt die Schnittstelle zum FPGA-Board dar. Die maximale Teilchenzahl für das FPGA-Design zum Lennard-Jones-Potential beträgt 8.192 (Zeile 3). Für Teilchenzahlen größer 8.192 können deshalb je Aufruf der Funktion `force`, also pro FPGA-Berechnung, nur Teile der Kräfte auf die *i*-Teilchen berechnet werden. Der Aufruf der Funktion `force` erfolgt deshalb in einer Doppelschleife, um auch die Kräfte größerer Systeme berechnen zu können. So werden die selben *i*-Teilchen solange erneut auf das FPGA-Board geladen, bis alle *j*-Teilchen einmal geladen wurden. Erst dann kann ein nächstes „Paket“ *i*-Teilchen geladen werden und wiederum etappenweise alle *j*-Teilchen. Die aktuellen Teilkräfte werden nach jedem `force`-Aufruf auf die bereits berechneten Teilkräfte addiert (Zeile 22-24).

Beim Lennard-Jones-FPGA-Design wird ein offenes System simuliert, d. h. die *Minimum Image Convention* muss hier nicht beachtet werden. Außerdem werden zur Berechnung der auf ein Teilchen wirkenden Kräfte die Einflüsse aller anderen Teilchen im System mit einbezogen. Dies ist nicht realistisch, da das Lennard-Jones-Potential ein kurzreichweitiges Potential ist und somit nur die Teilchen berücksichtigt werden müssen, die sich innerhalb des Cutoff-Radius befinden. Dieses Beispiel dient jedoch nur zum Testen der Funktionalität und Arbeitsleistung des FPGA-Designs. Eine realistische Implementierung mit Beachtung des Cutoff-Radius und der *Minimum Image Convention* wird in Kapitel 6 behandelt.

```
a = exp1*4.0*epsilon*(sigma)**exp1
b = exp2*4.0*epsilon*(sigma)**exp2
max_par = 8192
f = 0.0

i_begin = 1
i_end = max_par

do
   if (i_end >= n) i_end = n
   j_begin = 1
   j_end = max_par

   do
      if (j_end >= n) j_end = n

      CALL force(x(:,j_begin:j_end),x(:,i_begin:i_end),&
                 %VAL(a),%VAL(b),f_fpga(:,i_begin:i_end),&
                 %VAL(i_end-i_begin+1),%VAL(j_end-j_begin+1))

      ! Kräfte aufaddieren
      do i = i_begin, i_end
         f(:,i) = f(:,i) + f_fpga(:,i)
      enddo

      if (j_end == n) exit
      j_begin = j_begin + max_par
      j_end = j_end + max_par
   enddo

   if (i_end == n) exit
   i_begin = i_begin + max_par
   i_end = i_end + max_par
enddo
```

Listing 5.3: Berechnung der Kräfte mit dem FPGA-Board

5.2.3 Ergebnisse

Zum Testen des Lennard-Jones-FPGA-Designs wurde ein Testprogramm in Fortran geschrieben, das neben der Arbeitsleistung auch die Genauigkeit der berechneten Kräfte misst. Dazu werden die Kräfte zunächst auf der CPU berechnet. Anschließend wird dieselbe Berechnung mit dem PROGRAPE-4-Board durchgeführt. Die Tests wurden mit verschiedenen Teilchenzahlen durchgeführt.

Das System, auf dem die Tests durchgeführt wurden, ist eine Linux-Workstation am ARI in Heidelberg. Der 64-Bit-Rechner besitzt einen AMD Opteron Prozessor mit einer Taktrate von 2,0 GHz. Die reale Rechenleistung, die mit einem am ARI entwickelten Programm zur Gravitationskrafberechnung ermittelt wurde, beträgt 0,8 GFlops.

Zum PGDL-Code des Lennard-Jones-FPGA-Designs (Anhang B.2) wurden verschiedene Bitfiles erzeugt. Dazu wurde das Werkzeug ISE, Version 8.1i verwendet. ISE (***I**ntegrated **S**oftware **E**nvironment*) ist eine Entwicklungsumgebung der Firma Xilinx zum Entwurf von FPGA- und CPLD-Designs. Mit ISE können alle Entwicklungsphasen vom VHDL-Entwurf bis zum fertigen Bitstrom realisiert werden.

Mit ISE lassen sich maximal sechs Pipelines des Lennard-Jones-FPGA-Designs auf einem FPGA unterbringen. Mit den vier Pipeline-FPGAs des PROGRAPE-4-Boards lässt sich die Berechnung also mit maximal 24 parallelen Pipelines durchführen. Zu beachten ist, dass dabei mit einer Genauigkeit von 26 Bit für eine Fließkommazahl gerechnet wird. Das Design arbeitet mit einer Taktfrequenz von 66 MHz. Das ist die minimal mögliche Taktfrequenz, denn das PROGRAPE-4-Board arbeitet nur mit Taktfrequenzen von 66 MHz, 100 MHz oder 133 MHz. Designs mit niedrigeren Taktfrequenzen liefern keine korrekten Ergebnisse.

Genauigkeit

Bevor man die Leistung eines Programms analysiert, muss sichergestellt sein, dass es korrekte Ergebnisse liefert. Dazu dient in erster Linie der Softwareemulator. Jedoch liefert der Emulator keine hundertprozentige Garantie, dass das entwickelte Hardwaredesign auch auf dem FPGA richtig rechnet. Es ist immer notwendig, die Korrektheit der Ergebnisse auch unter realen Bedingungen zu überprüfen. Ein Grund dafür, dass die Berechnung auf dem FPGA fehlerhaft sein kann, obwohl der Emulator korrekte Ergebnisse liefert, kann das Zeitverhalten des Designs sein. Dieses wird nämlich beim Emulator nicht berücksichtigt, sondern nur die Funktionalität. Es wird nicht erkannt, ob einzelne Pfade zu lang sind, so dass die Taktrate des Designs unter der erforderlichen Taktrate von 66 MHz des PROGRAPE-Boards liegt. Weiterhin kann es passieren, dass das Design zu groß für den FPGA ist. Dies wird vom Emulator ebenfalls nicht erkannt.

Aus diesem Grund wurden die Ergebnisse der FPGA-Berechnung mit denen der CPU verglichen und der absolute und relative Fehler ermittelt. Absoluter und relativer Fehler berechnen sich wie folgt:

$$E_{abs} = \|f_{CPU} - f_{FPGA}\|_\infty \tag{5.5}$$

$$E_{rel} = \frac{\|f_{CPU} - f_{FPGA}\|_2}{\|f_{CPU}\|_2} \tag{5.6}$$

f_{CPU} : auf der CPU berechnete Kraft
f_{FPGA} : auf dem FPGA berechnete Kraft

$\|x\|_\infty = max_{i_1,\ldots,N} |x_i|$ (Unendlichnorm)
$\|x\|_2 = \sqrt{\sum_{i=1}^{N} x_i^2}$ (Zweinorm)

Für dieses Beispiel wurden Bitfiles mit zwei verschiedenen Bitbreiten generiert. Zum einen mit 26 Bit und zum anderen mit 32 Bit. Auf der CPU wurde mit doppelter Genauigkeit gerechnet, also mit 64 Bit. Beim 26-Bit-Fließkommaformat beträgt die Mantissenbreite 16 Bit. Daraus ergibt sich ein erwarteter relativer Fehler von $2^{-16} = 1,53 \cdot 10^{-5}$ im Vergleich zur CPU. Das 32-Bit-Fließkommaformat entspricht in etwa der einfachen Genauigkeit beim IEEE-Format. Die Mantissenbreite beträgt 23 Bit. Im Gegensatz zum IEEE-Format besitzt jedoch der Exponent statt acht, nur sieben Bit, da ein Bit zusätzlich zur Kennzeichnung der Null verwendet wird. Eigentlich sollte hier das 33-Bit-Fließkommaformat verwendet werden, was identisch mit dem IEEE-Format ist. Die Generierung eines Bitfiles für dieses Format lieferte jedoch Fehlermeldungen. Für das 32-Bit-Fließkommaformat mit einer Mantissenbreite von 23 Bit ergibt sich ein erwarteter relativer Fehler von $2^{-23} = 1,20 \cdot 10^{-7}$.

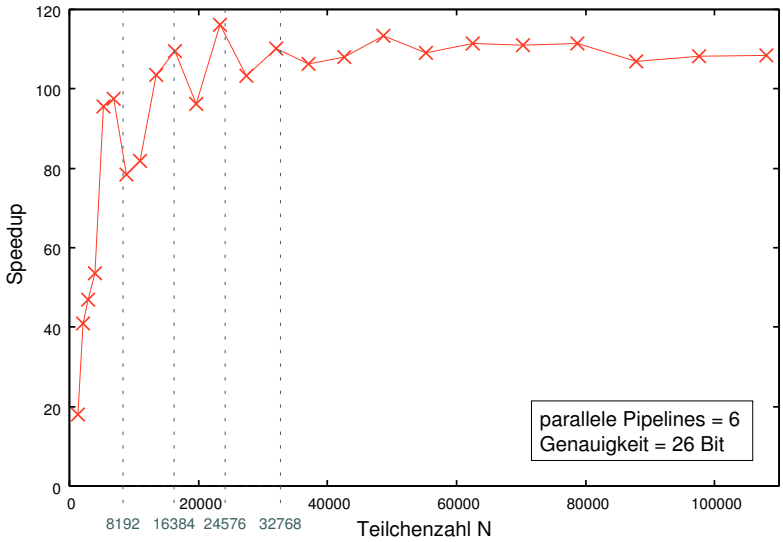


Abbildung 5.1: Speedup des Lennard-Jones-FPGA-Designs in Abhängigkeit von der Anzahl der Teilchen mit sechs parallelen Pipelines pro FPGA

Absoluter und relativer Fehler wurden für verschiedene Teilchenzahlen bis zu $N = 32000$ gemessen. Die Messungen ergaben, dass der relative Fehler bei beiden Bitbreiten deutlich unter dem erwarteten Fehler blieb. Das 26-Bit-Fließkommaformat ergab im Durchschnitt einen maximalen Fehler von etwa $0,037$ und einen relativen Fehler von ca. $7,3 \cdot 10^{-6}$. Beim 32-Bit-Fließkommaformat betrug der maximale Fehler im Durchschnitt $9 \cdot 10^{-5}$ und der relative Fehler $1,8 \cdot 10^{-8}$.

Performance

Wie schon erwähnt, passen auf einen FPGA sechs parallele Pipelines zur Kräfteberechnung nach dem Lennard-Jones-Potential. Auf dem PROGRAPE-4-Board können also maximal 24 Pipelines untergebracht werden. In Abbildung 5.1 ist der Speedup bei sechs parallelen Pipelines pro FPGA über der Anzahl der Teilchen aufgetragen. Der Speedup gibt an, um welchen Faktor ein Algorithmus oder ein Programmteil beschleunigt wurde. In diesem Fall gibt der Speedup an, um welchen Faktor die Kräfteberechnung mit dem FPGA schneller ist als die Kräfteberechnung auf der CPU. Dazu wurde die Ausführungszeit der Wechselwirkungsberechnung auf der CPU und mit Hilfe des PROGRAPE-4-Boards ermittelt. Die Ausführungszeit oder Antwortzeit ist dabei die Zeit, die zwischen Start und Fertigstellung einer bestimmten Anzahl an Befehlen gemessen wird. Sie unterscheidet sich von der CPU-Zeit, die angibt, wie lange die CPU effektiv mit der gestellten Aufgabe beschäftigt war. Der Speedup berechnet sich somit wie folgt:

$$Speedup = \frac{Zeit\ mit\ FPGA}{Zeit\ ohne\ FPGA} \tag{5.7}$$

Wie in Abbildung 5.1 erkennbar, nähert sich der Speedup für sechs parallele Pipelines pro FPGA für Teilchenzahlen größer 60.000 etwa einem Faktor von 110 an. Auffallend an dieser Kurve ist zum einen, dass der Speedup für kleine Teilchenzahlen bis etwa 20.000 sehr steil ansteigt, dann abflacht und sich schließlich bei 110 einpendelt. Zum anderen fällt auf, dass die Kurve nicht monoton steigt, sondern bis zu einer Teilchenzahl von 40.000 immer wieder leicht „einbricht“.

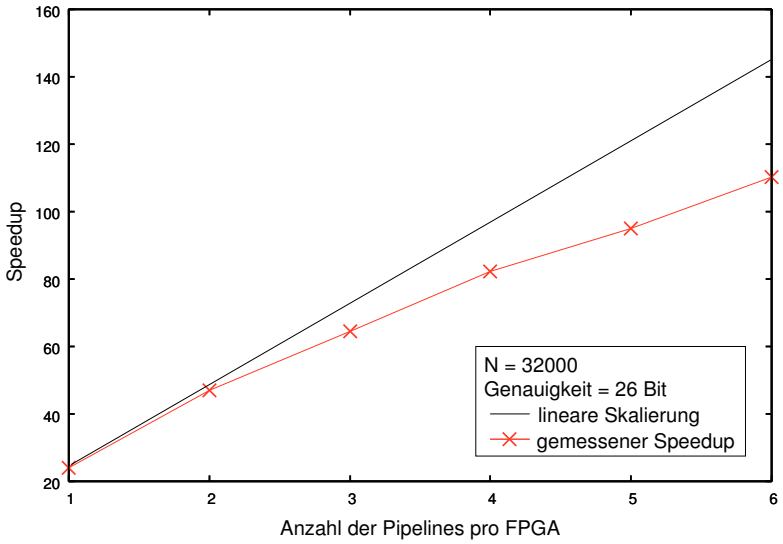


Abbildung 5.2: Speedup des Lennard-Jones-FPGA-Designs in Abhängigkeit von der Anzahl der parallelen Pipelines pro FPGA für 32.000 Teilchen

Dass der Speedup zunächst stark ansteigt und sich schließlich bei einem Wert einpendelt, lässt sich mit der Latenzzeit und der Zeit, die für die Datenübertragung benötigt wird, erklären. Die Latenzzeit einer Pipeline, ist die Zeit, die benötigt wird, um das erste Ergebnis zu produzieren. Beim Start einer Berechnung müssen zunächst alle Operationseinheiten der Pipeline durchlaufen werden, bis ein Ergebnis vorliegt. Danach wird in jedem Taktzyklus ein neues Ergebnis produziert. Für kleine Teilchenzahlen nehmen Latenz- und Datenübertragungszeit einen größeren Teil der Berechnung ein, weshalb hier der Speedup im Vergleich zur CPU nicht so groß ist. Für große Teilchenzahlen sind Latenz- und Datenübertragungszeit vernachlässigbar. Dies erklärt, warum sich der Speedup schließlich bei einem Wert einpendelt.

Die „Einbrüche“ im Speedup lassen sich anhand der Teilchenzahlen erklären. Dazu sei noch einmal in Erinnerung gerufen, dass maximal 8.192 Teilchen gleichzeitig auf das FPGA-Board geladen werden können. Der erste „Einbruch“ erfolgt genau an der Grenze der maximalen Teilchenzahl. In Abbildung 5.1 fällt der Speedup bereits bei etwa 7.000 Teilchen. Das liegt jedoch daran, dass die Intervalle zwischen den Teilchenzahlen nicht fein genug sind. Im Programm werden die Teilchen beim Start der Simulation in einem fcc-Gitter angeordnet. Sinnvolle Teilchenzahlen für dieses Gitter sind $N = 4 \cdot n^3$ Teilchen, wobei n eine ganze Zahl ist. Während bei einer Teilchenzahl von 8.192 alle Daten auf einmal auf das PROGRAPE-4-Board geladen werden, müssen bei 8.193 Teilchen zweimal Daten übertragen werden. Latenz- und Übertragungszeit fallen hier also zweimal ins Gewicht. Da diese bei kleiner Teilchenzahl noch einen relativ großen Teil der Gesamtzeit in Anspruch nehmen, fällt der Speedup deutlicher als bei großen Teilchenzahlen. Nach diesem „Einbruch“ steigt der Speedup bis zu einer Teilchenzahl von $2 \cdot 8192 = 16384$ wieder an und bricht erneut ein. Die lokalen Maxima der Speedupkurve befinden sich also immer bei Teilchenzahlen, die ein Vielfaches von 8.192 sind. Bei Teilchenzahlen größer 50.000 sind diese „Einbrüche“ nicht mehr erkennbar, da Latenz- und Übertragungszeit hier vernachlässigbar sind.

In Abbildung 5.2 ist der Speedup für 32.000 Teilchen in Abhängigkeit von der Anzahl der parallelen Pipelines pro FPGA dargestellt. Wie erkennbar ist, skaliert das Programm nicht ganz linear. Sechs parallele Pipelines pro FPGA sind also nicht sechsmal so schnell wie eine Pipeline, sondern lediglich um einen Faktor 4,5 schneller. Ein Grund dafür könnte die größere Kommunikation und

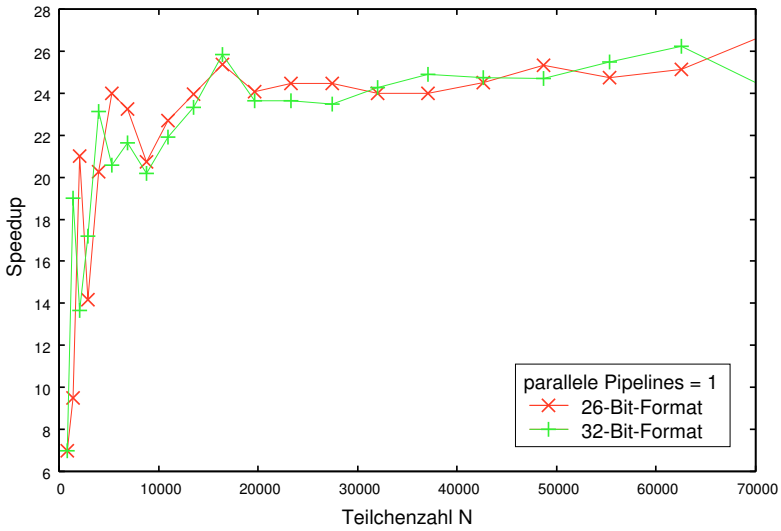


Abbildung 5.3: Speedup des Lennard-Jones-FPGA-Designs für das 26- und 32-Bit-Fließkommaformat für eine Pipeline pro FPGA

Datenübertragung zwischen Interface-FPGA und den einzelnen Pipeline-FPGAs sein. Bei sechs Pipelines pro FPGA müssen pro Taktzyklus mehr Daten an die Pipeline-FPGAs gesendet werden.

Des Weiteren ist es interessant zu untersuchen, wie sich die Rechenleistung in Abhängigkeit zur eingestellten Bitbreite verhält. Für ein 32-Bit-Fließkommaformat war es jedoch nur möglich, ein Bitfile mit einer Pipeline pro FPGA zu erzeugen. Die Generierung für zwei parallele Pipelines lieferte zwar ein Bitfile mit einer erlaubten Taktfrequenz von 66,84 MHz. Die Programmausführung mit diesem Bitfile ergab jedoch völlig falsche Werte. Deshalb wird in Abbildung 5.3 nur der Speedup von 26- und 32-Bit-Fließkommaformat für eine parallele Pipeline pro FPGA verglichen. Es ist zu erkennen, dass sich Speedup bei 26 und 32 Bit nicht wesentlich unterscheiden. Bedenkt man jedoch, dass sich mit dem 26-Bit-Fließkommaformat mehr Pipelines auf einem FPGA platzieren lassen, so lässt sich mit geringerer Genauigkeit eine höhere Beschleunigung erreichen.

Kapitel 6

Linked-Cell-List-FPGA-Design

Kapitel 5.1 und 5.2 zeigen, dass sich die Kräfteberechnung zur Simulation von Vielteilchensystemen mit FPGAs erheblich beschleunigen lässt. Jedoch sind beide betrachteten Beispiele $O(N^2)$-Probleme. In der Molekulardynamik existieren jedoch bereits Algorithmen, die die Kräfteberechnung auf eine Komplexität von $O(N\ log\ N)$ oder sogar $O(N)$ reduzieren (siehe Kapitel 2.4.4). Es stellt sich nun die Frage, ob sich das PROGRAPE-Board auch für solche Algorithmen effizient einsetzen lässt. Sicherlich ist die Beschleunigung von $O(N^2)$-Problemen ein großer Fortschritt. Algorithmen mit einer Komplexität von $O(N\ log\ N)$ oder $O(N)$ wären jedoch für sehr große Teilchenzahlen schneller als die mit FPGAs beschleunigten $O(N^2)$-Algorithmen.

Dieses Kapitel befasst sich mit der Implementierung eines FPGA-Designs auf der Basis des Linked-Cell-List-Algorithmus. Dazu wird zunächst auf Details der Softwareimplementierung des Algorithmus eingegangen. Die Vor- und Nachteile der Linked-Cell-Listen-Technik führen direkt zu einigen Problemen, die sich beim Einsatz des PROGRAPE-Boards und PGR ergeben. Zur Umgehung dieser Probleme wird ein Konzept für die Hardware- und Softwareimplementierung erarbeitet, das anschließend umgesetzt wird. Abschließend wird der gemessene Leistungsgewinn diskutiert und ein allgemeines Fazit zur Implementierung von FPGA-Designs gegeben.

6.1 Der Linked-Cell-List-Algorithmus

Wie schon in Kapitel 2.3 beschrieben, wird beim Linked-Cell-List-Algorithmus die Simulationsbox in kubische Zellen eingeteilt. Anhand ihrer Positionen werden alle N Teilchen in Zellen einsortiert (Listing 6.1, Zeile 6-8). Für jede Zelle wird eine Liste angelegt, in der die zugehörigen Teilchen gespeichert werden. Listing 6.1 enthält den Fortran-Quelltext zum Anlegen der Listen. In dieser Diplomarbeit werden ausschließlich dreidimensionale Systeme betrachtet, die sich in einer würfelförmigen Simulationsbox befinden. In einem solchen System wird die Simulationsbox in Würfel unterteilt. `Lx`, `Ly` und `Lz` in Listing 6.1 bezeichnen die Ausdehnung der Simulationsbox in die jeweilige Koordinatenrichtung. Für eine kubische Box gilt also dementsprechend: $\texttt{Lx} = \texttt{Ly} = \texttt{Lz}$. `nx` und `ny` bezeichnen die Anzahl der Zellen in der jeweiligen Ebene. Bei einer kubischen Simulationsbox sind diese ebenfalls identisch.

```
1  head = 0
2  entry = 0
3  cell_entry = -1
4
5  do i=1, N
6     cell = 1 + int(x(1,i)/Lx)
7              + int(x(2,i)/Ly) * nx
8              + int(x(3,i)/Lz) * nx * ny
9
10    entry(i) = head(cell)
11    head(cell) = i
12    cell_entry(cell) = cell_entry(cell) + 1
13 enddo
```

Listing 6.1: Fortran-Quelltext zum Anlegen der Nachbarschaftslisten beim Linked-Cell-List-Algorithmus

Die Linked-Cell-Liste besteht im Wesentlichen aus einem Vektor (`entry`) der Länge *N*. In diesem Vektor wird für jedes Teilchen der Index des nächsten Teilchens in der gleichen Zelle abgespeichert. Das letzte Teilchen einer Zelle erhält den `entry`-Eintrag 0. Um an den Anfang der Liste einer Zelle zu gelangen, wird ein Vektor (`head`) benötigt, der jeweils auf das erste Teilchen in einer Zelle zeigt. `head(1)` enthält demnach den Index des ersten Teilchens in Zelle 1, `head(2)` den des ersten Teilchens in Zelle 2, usw. Die Anzahl der Elemente von `head` entspricht somit der Anzahl der Zellen. Auf diese Weise entsteht eine „verlinkte Kette“, mit der alle Teilchen zellenweise durchlaufen werden können. Zur Erstellung der Liste wird eine Schleife benötigt, die über alle *N* Teilchen läuft. In einem weiteren Vektor (`cell_entry`) wird außerdem noch für jede Zelle die Anzahl der in ihr befindlichen Teilchen gespeichert.

Nachdem die Liste angelegt wurde, können die Kräfte berechnet werden (Listing 6.2). Bei der einfachsten Art der Implementierung wählt man die Kantenlänge der Zellen so, dass sie mindestens so groß ist wie der Cutoff-Radius. Für die vorliegende Arbeit wurde diese Realisierung gewählt. In diesem Fall, müssen für die Berechnung der Kraft auf ein Teilchen nur die Teilchen berücksichtigt werden, die sich in der aktuellen Zelle oder in einer der 26 Nachbarzellen befinden. Auf diese Weise reduziert sich die Kräfteberechnung auf ein $O(N)$-Problem, wobei jedoch der Vorfaktor sehr groß sein kann. In einem dreidimensionalen System müssen also Teilchen aus 27 Zellen berücksichtigt werden. Wendet man auch hier das dritte Newtonsche Axiom an, kann die Anzahl der zu berücksichtigenden Nachbarzellen auf 13 reduziert werden [24].

Bei der Kräfteberechnung wird zellenweise vorgegangen. Das bedeutet, dass zunächst die Kräfte auf alle Teilchen in der ersten Zelle berechnet werden, dann die Kräfte der Teilchen in der zweiten Zelle, usw. Innerhalb einer Zelle werden die Teilchen mit Hilfe der Linked-Cell-Listen nacheinander durchlaufen. Bei der Berechnung der Kraft auf ein Teilchen *i* werden dann zunächst die Einflüsse der anderen Teilchen aus der gleichen Zelle betrachtet. Anschließend werden die Beiträge der Teilchen in den 13 Nachbarzellen einbezogen. Dabei ist zu beachten, dass für jedes Teilchen überprüft werden muss, ob es sich innerhalb des Cutoff-Radius von Teilchen *i* befindet. Nur in diesem Fall wird die Kraft berechnet. Zusätzlich muss die *Minimum Image Convention* berücksichtigt werden. Das bedeutet, dass für Zellen am Rand der Simulationsbox die Bildteilchen in der entsprechenden Bildbox betrachtet werden müssen (siehe Kapitel 2.2.1). Außerdem wird das Abscheiden des Potentials beim Cutoff-Radius durch das Shifted-force Potential (siehe Kapitel 2.2.2) korrigiert.

```
1  ! Schleife über alle Zellen
2  do ic = 1, i_cell
3     i = head(ic)
4
5     ! Schleife über alle Teilchen der aktuellen Zelle
6     do le_i = 1, cell_entry(ic)
7
8        ! Schleife über alle restlichen Teilchen der aktuellen Zelle
9        j = entry(i)
10       do le_j = le_i+1, cell_entry(ic)
11          /*************************************/
12          /* Abstand rij berechnen (MIC beachten) */
13          /* überprüfen, ob rij < Cutoff-Radius   */
14          /* Kraft Fij berechnen und aufaddieren  */
15          /*************************************/
16          j = entry(j)
17       enddo
18
19       ! Schleife über alle Teilchen in den Nachbarzellen
20       do nc = 1, neigbor_cells(ic)
21          jc = cell_list(nc)
22          j = head(jc)
23          do le_j = 1, cell_entry(jc)
24             /*************************************/
25             /* Abstand rij berechnen (MIC beachten) */
26             /* überprüfen, ob rij < Cutoff-Radius   */
27             /* Kraft Fij berechnen und aufaddieren  */
28             /*************************************/
29             j = entry(j)
30          enddo
31       enddo
32
33       i = entry(i)
34    enddo
35 enddo
```

Listing 6.2: Kräfteberechnung mittels Linked-Cell-Listen

6.1.1 Vor- und Nachteile des Linked-Cell-List-Algorithmus

Ein großer Vorteil des Linked-Cell-List-Algorithmus ist die Tatsache, dass er mit $O(N)$ skaliert. Des Weiteren ist der Aufbau der Nachbarschaftslisten einfach und erfordert keine globalen Operationen. Der Speicherbedarf für die Liste wächst ebenfalls nur linear mit der Anzahl der Teilchen. Der Linked-Cell-List-Algorithmus lässt sich also ohne weiteres auch auf Systeme mit einer sehr großen Teilchenzahl anwenden. Die simulierbare Teilchenzahl wird lediglich durch den vorhandenen Speicherplatz beschränkt.

Die Nachteile des Linked-Cell-List-Algorithmus lassen sich in Listing 6.2 gut erkennen. Die Kräfteberechnung wird hier nicht mehr mit einer Doppelschleife realisiert, sondern ist in mehrere kleine Schleifen aufgeteilt. Zudem erfolgt die Adressierung der Teilchen nicht mehr direkt über einen einfachen Zähler. Durch die zellenweise Betrachtung der Teilchen werden diese nicht nacheinander, wie sie im Speicher stehen, abgearbeitet. Die Reihenfolge richtet sich dabei nach der Reihenfolge, in der sie in der Nachbarschaftsliste abgespeichert sind. Der Zugriff auf die Teilchen erfolgt über die Indizes der Liste.

6.2 Problematik

Aus den in Kapitel 6.1.1 beschriebenen Nachteilen des Linked-Cell-List-Algorithmus ergeben sich gleichzeitig die Probleme der Entwicklung eines FPGA-Designs mit PGR. Wie in Kapitel 4.2 beschrieben, wurde PGR speziell dafür konzipiert, FPGA-Pipelines für Probleme zu entwickeln, die sich wie folgt darstellen lassen:

$$f_i = \sum_j G(a_i, a_j) \tag{6.1}$$

Mit PGDL wird die Schnittstelle zu diesem Problem definiert und über die parametrisierten AMs die Funktion *G* realisiert. PGR generiert daraus für den Softwareemulator automatisch die folgende Doppelschleife:

```
for (i = 0; i < ni; i++){
    for (j = nj - 1; j >= 0; j--){
        /* Berechnung von G */
    }
}
```

Der Benutzer hat dabei nur in dem Maße Einfluss auf die Gestaltung dieser Doppelschleife, dass er die Größen `ni` und `nj` bestimmen bzw. beeinflussen kann. Diese Größen ergeben sich aus der Anzahl der *i*- und *j*-Teilchen, die auf das FPGA-Board geladen werden. Es besteht keine Möglichkeit, den Verlauf der Schleifenindizes zu beeinflussen. Zudem erlaubt PGDL nicht, gezielt auf ein Vektorelement zuzugreifen. Anweisungen wie `head(ic)` (Listing 6.2, Zeile 3) sind somit nicht realisierbar.

Macht man sich die Arbeitsweise von PGR klar, so sind diese Einschränkungen nicht verwunderlich. PGR ist darauf ausgerichtet, Pipelines zu erzeugen. Solche Pipelines sind dann sinnvoll, wenn die gleichen Operationen auf eine große Menge von Daten angewendet werden. Eine Pipeline besteht aus einer Reihe von hintereinanderliegenden Verarbeitungseinheiten. Objekte, beispielsweise die Elemente eines Vektors, werden der Pipeline nacheinander übergeben. Jedes Objekt durchläuft daraufhin alle Einheiten der Pipeline. Die Ergebnisse werden erst am Ende der Pipeline zugreifbar. Da jede Pipelineeinheit zeitgleich mit einer anderen Einheit ein Objekt in Bearbeitung haben kann, wird somit eine Parallelität in der Verarbeitung der Objekte erreicht.

PGR generiert also nur für den Softwareemulator eine Doppelschleife. Das eigentliche Hardwaredesign ist jedoch keine Doppelschleife, sondern eine Pipeline. Diese Tatsache erklärt, dass der gezielte Zugriff auf Vektorelemente nicht möglich ist, denn innerhalb einer Pipeline ist es nicht möglich, einzelne Elemente anzusprechen. Da die Elemente eines Vektors nacheinander in eine Pipeline gegeben werden, ist es ebenso einsichtig, dass der Schleifenindex nicht beeinflussbar ist.

6.3 Konzept

Mit PGR lassen sich also nur einfache Pipelinedesigns realisieren und keine komplexen Algorithmen. Um das PROGRAPE-Board trotzdem für den Linked-Cell-List-Algorithmus nutzbar zu machen, muss das FPGA-Design so einfach wie möglich gehalten werden. Dazu muss der Algorithmus dementsprechend umgeschrieben werden.

Auf dem PROGRAPE-Board soll also nur das Nötigste berechnet werden. Die dieser Arbeit zugrunde liegende Implementierung eines Linked-Cell-List-Algorithmus berechnet neben den Kräften auch in jedem Schritt die gesamte potentielle Energie des Systems. Zur Berechnung dieser Energie wird, genau wie bei der Berechnung der Kräfte, eine Doppelschleife über alle Teilchenpaare benötigt. Aus diesem Grund ist es sinnvoll, die Berechnung der potentiellen Energie ebenfalls auf dem FPGA-Board durchzuführen.

Neben Kräfte- und Energieberechnung muss auf dem FPGA-Board zudem noch die Cutoff-Bedingung, die *Minimum Image Convention* und die Korrektur durch das Shifted-force Potential beachtet werden. Die Cutoff-Bedingung kann jeweils erst dann abgefragt werden, wenn der Abstand zwischen zwei Teilchen berechnet wurde. Nur wenn die Cutoff-Bedingung erfüllt ist, also der Abstand der Teilchen kleiner als der Cutoff-Radius ist, wird die Kraft zwischen diesen beiden Teilchen berechnet. Die Abfrage der Cutoff-Bedingung ist somit fester Bestandteil der Kräfteberechnung. Da es nicht möglich ist, diese beiden Operationen voneinander zu trennen, muss die Cutoff-Bedingung auf dem FPGA-Board abgefragt werden. Anders sieht es mit der *Minimum Image Convention* aus. Diese kann bereits vorher berücksichtigt werden und muss nicht auf dem FPGA-Board durchgeführt werden. Das ist zum einen günstig, da das FPGA-Design so nicht unnötig ausgeweitet wird. Zum anderen spart es If-Anweisungen, die auf dem FPGA nicht effizient realisiert werden können und möglichst vermieden werden sollten. Die Korrekturterme für das Shifted-force Potential können ebenfalls auf dem FPGA von der jeweiligen Kraft- bzw. Energiekomponente subtrahiert werden. Auf die Einzelheiten wird in Kapitel 6.4 genauer eingegangen.

Man könnte nun die in Listing 6.2 dargestellte Schleifenstruktur des Linked-Cell-List-Algorithmus unverändert lassen und die Kräfteberechnung an den jeweiligen Stellen (Zeile 14 und Zeile 27) durchführen. Das würde bedeuten, dass man an diesen Stellen die von PGR generierte `force`-Routine aufruft. Der Nachteil ist jedoch, dass man hierbei nur jeweils ein *i*- und ein *j*-Teilchen auf den FPGA, bzw. das PROGRAPE-Board lädt. Da das Board auf dem Pipelining-Prinzip beruht und eine große Menge Daten möglichst parallel bearbeitet, ist es nicht sinnvoll. Durch den häufigen Datenaustausch zwischen PROGRAPE-Board und Wirtsrechner wäre ein solches Design wesentlich langsamer als der ursprüngliche Algorithmus. Eine bessere Lösung ist, die *i*- und *j*-Teilchen zu „sammeln“ und zu mehreren gleichzeitig auf das FPGA-Board zu laden.

Der in diesem Kapitel entwickelte Lösungsansatz wird im Folgenden weiter beschrieben.

6.4 Implementierung

In diesem Abschnitt werden die Details der Implementierung des Linked-Cell-List-FPGA-Designs beschrieben. Dazu wird insbesondere auf die Umsetzung des in Kapitel 6.3 erarbeiteten Konzepts eingegangen. Im Folgenden werden zunächst zunächst einige Besonderheiten und Probleme der Hardwareimplementierung beschrieben. Anschließend werden die Änderungen der gegebenen Linked-Cell-List-Implementierung beleuchtet, die für eine Berechnung mit dem PROGRAPE-Board notwendig waren.

6.4.1 Hardwaredesign

Wie im Konzept erarbeitet, wird auf dem FPGA-Board die Abfrage nach der Cutoff-Bedingung, die Korrektur durch das Shifted-force Potential, sowie die Berechnung der Kräfte und potentiellen Energie durchgeführt. Der PGDL-Code zum FPGA-Design befindet sich im Anhang (B.3).

Für die Berechnung der Kräfte auf dem FPGA-Board wurde in Kapitel 5.2.2 folgende Formel hergeleitet:

$$\vec{f}_i = \sum_{j \neq i} \left(\left(\frac{a}{r_{ij}^{14}} - \frac{b}{r_{ij}^{8}} \right) \cdot \vec{r_{ij}} \right) \tag{6.2}$$

mit $a = 4\varepsilon 12\, \sigma^{12}$, $b = 4\varepsilon\, 6\, \sigma^{6}$

ε : Energiekonstante
σ : Durchmesser eines Teilchens
$\vec{r_{ij}}$: Verbindungsvektor zwischen Teilchen i und j ($\vec{r_j} - \vec{r_i}$)
r_{ij} : Abstand zwischen Teilchen i und j ($\|\vec{r_i} - \vec{r_j}\|_2$)

Mit der Korrektur durch das Shifted-force Potential ergibt sich daraus folgende Gleichung:

$$\vec{f}_i = \sum_{j \neq i} \left(\left(\frac{a}{r_{ij}^{14}} - \frac{b}{r_{ij}^{8}} \right) \cdot \vec{r_{ij}} \right) - \frac{\vec{r_{ij}}}{r_{ij}} \cdot for\_rc \tag{6.3}$$

mit $for\_rc = 4\varepsilon \frac{1}{r_{cut}} \left(12 \cdot \left(\frac{\sigma}{r_{cut}} \right)^{12} - 6 \cdot \left(\frac{\sigma}{r_{cut}} \right)^{6} \right)$

Um auf dem FPGA-Board möglichst wenige Operationen durchführen zu müssen, wird Gleichung 5.2 zur Berechnung der potentiellen Energie wie folgt umgeschrieben:

$$U_i = \sum_j 4\varepsilon\sigma^6 \cdot \left(\sigma^6 \frac{1}{r_{ij}^{12}} - \frac{1}{r_{ij}^{6}} \right) = \sum_j c \cdot \left(d \frac{1}{r_{ij}^{12}} - \frac{1}{r_{ij}^{6}} \right) \tag{6.4}$$

mit $c = 4\varepsilon\, \sigma^6$, $d = \sigma^6$

ε : Energiekonstante
σ : Durchmesser eines Teilchens
r_{ij} : Abstand zwischen Teilchen i und j ($\|\vec{r_i} - \vec{r_j}\|_2$)

Mit der Korrektur durch das Shifted-force Potential ergibt sich wiederum:

$$U_i = \sum_j c \cdot \left(d \frac{1}{r_{ij}^{12}} - \frac{1}{r_{ij}^{6}} \right) - pot\_rc + (r - r_{cut}) \cdot for\_rc \tag{6.5}$$

mit $pot\_rc = 4\varepsilon \left(\left(\frac{\sigma}{r_{cut}} \right)^{12} - \left(\frac{\sigma}{r_{cut}} \right)^{6} \right)$

$for\_rc = 4\varepsilon \frac{1}{r_{cut}} \left(12 \cdot \left(\frac{\sigma}{r_{cut}} \right)^{12} - 6 \cdot \left(\frac{\sigma}{r_{cut}} \right)^{6} \right)$

Wie bei der Kraftberechnung sind `c` und `d` Konstanten, die zuvor vom Programm berechnet und als Parameter an das FPGA-Board übergeben werden. Die Korrekturterme `pot_rc` und `for_rc` für Kraft und potentielle Energie werden ebenfalls vorher berechnet und auf das FPGA-Board geladen. Die Schnittstelle der `force`-Routine für das Design sieht damit wie folgt aus:

```
force(double xjj[][3], double xii[][3], double r_cut_sq,
      double r_cut, double pot_rc, double for_rc,
      double a, double b, double c, double d,
      double zeros[3], double f[][3], double pot[],
      int ni, int nj)
```

`xjj` enthält wie beim Lennard-Jones-Potential die Positionen der *j*-Teilchen und `xii` die Positionen der *i*-Teilchen. `r_cut_sq` enthält den quadrierten Cutoff-Radius, `r_cut` den Cutoff-Radius selbst. Die Variablen `pot_rc`, `for_rc`, `a`, `b`, `c` und `d` enthalten die bereits beschriebenen Werte. Der Vektor `zeros` enthält lediglich Nullen. Diese werden für die Cutoff-Bedingung benötigt. Darauf wird später genauer eingegangen. Die Matrix `f` enthält nach der Berechnung die Teilkräfte, der Vektor `pot` die potentielle Energie des jeweiligen Teilchens. `ni` und `nj` bezeichnen wieder die Anzahl der *i*- bzw. *j*-Teilchen.

Wie im Konzept beschrieben, ist es nicht möglich, die Abfrage der Cutoff-Bedingung von der Kräfteberechnung zu trennen. If-Abfragen sind jedoch auf dem FPGA nicht ohne Weiteres zu realisieren. Erinnert man sich wieder daran, dass aus dem PGDL-Code eine Pipeline erzeugt wird, ist dies einsichtig. Eine Pipeline besteht aus einer Reihe von Verarbeitungseinheiten. Es ist nicht möglich zu verzweigen. In PGDL besteht dennoch die Möglichkeit If-Anweisungen zu beschreiben. Eine Anweisung der Form:

```
if (a > b) then
    dx = c * d
else
    dx = e + f
endif
```

lässt sich mit PGDL wie folgt beschreiben:

```
pg_float_mult(c,d,dx1,NFLO,NMAN,4);
pg_float_add (e,f,dx2,NFLO,NMAN,4);
pg_float_compare(a,b,flag,NFLO,NMAN,1);
pg_bits_mux(flag,dx2,dx1,dx,NFLO,1);
```

Da eine Verzweigung nicht möglich ist, müssen zunächst beide Werte berechnet werden. Diese werden in `dx1` und `dx2` gespeichert. `pg_float_compare` überprüft, ob `a` größer ist als `b`. Ist dies der Fall wird die 1-Bit-Variable `flag` auf 1, andernfalls auf 0 gesetzt. Mit `pg_bits_mux` wird anschließend aufgrund des Inhalts von `flag` entschieden, welcher Wert an `dx` zugewiesen wird. Ist `flag` 0, wird `dx2`, andernfalls `dx1` gewählt. `dx` enthält schließlich das Ergebnis der If-Anweisung.

Der Umstand, dass bei einer If-Anweisung auf dem FPGA immer beide Zweige ausgeführt werden müssen, macht die Benutzung sehr ineffizient. Am Beispiel des Cutoff-Radius wird dies sehr deut-

lich. Auf dem FPGA müssen Kraft und Energie also für jedes Teilchenpaar berechnet werden, auch wenn der Abstand der Teilchen größer als der Cutoff-Radius ist. Auf der CPU werden Kraft und Energie in diesem Fall nicht berechnet. Die If-Anweisung ist auch der Grund für die Einführung des Vektors `zeros`. Ist der Teilchenabstand größer als der Cutoff-Radius wird die Kraftkomponente des Teilchenpaares auf Null gesetzt.

Ein weiteres Problem ergab sich bei der Generierung des Bitfiles. Das Design arbeitete zunächst korrekt mit dem Emulator. Bei der Berechnung auf dem FPGA-Board ergaben sich jedoch völlig falsche Werte. Dies war zunächst verwunderlich, da bei der Bitfile-Generierung keine Fehlermeldungen auftraten. Ein Blick in die Log-Dateien der Timing-Simulation lieferte die Auflösung.

Der kritische Pfad des Designs war zu lang. Eine digitale, synchrone Schaltung besteht normalerweise aus mehreren Flipflops zwischen denen sich kombinatorische Logikblöcke befinden, die aus Grundgattern (UND, ODER, NICHT, usw.) aufgebaut sind und entsprechend der gewählten Aufgabe verschaltet sind. Bevor ein Flipflop die Daten eines Logikblocks übernehmen kann muss sichergestellt sein, dass die korrekten Daten bereits am Eingang anliegen. Dabei ist zu beachten, dass die reale Hardware nicht beliebig schnell ist, sondern eine gewisse Zeit für Signale und Gatter benötigt. Die zeitintensivste Aneinanderreihung von Gattern und Flipflops einer Schaltung, die innerhalb eines Taktes ein Ergebnis liefern muss, wird als kritischer Pfad bezeichnet. Der kritische Pfad ist also der zeitlich längste Weg, der zur Ausführung einer Pipelinestufe benötigt wird. Er bestimmt die maximale Taktfrequenz einer Schaltung. Die zeitliche Verzögerung der nicht kritischen Pfade kann erhöht werden, ohne die maximal mögliche Taktrate zu beeinflussen.

In diesem Fall betrug die Zykluszeit des kritischen Pfades 22,55 ns, womit eine maximale Taktfrequenz von 44,34 MHz erreicht werden konnte. Die minimale Taktfrequenz für das PROGRAPE-4-Board ist jedoch 66 MHz. Damit das Design korrekt arbeiten kann, musste also der kritische Pfad gefunden und die Zykluszeit verkürzt werden. Ein weiterer Blick in die Log-Dateien ergab, dass die Berechnung des Kehrwertes die meiste Zeit benötigt. Um die Zykluszeit der Kehrwertberechnung zu verkürzen, wurde die Anzahl der Pipelinestufen des arithmetischen Moduls `pg_float_recipro` vergrößert. Da `pg_float_recipro` für maximal fünf Pipelinestufen implementiert ist, wurde diese auf fünf gesetzt. Die anschließende Timing-Simulation ergab nun eine Zykluszeit von 14,86 ns. Damit beträgt die maximale Taktfrequenz 67,29 MHz. Mit dieser Taktrate liefert das Design auch auf dem FPGA-Board korrekte Ergebnisse.

6.4.2 Softwaredesign

Da die Berechnung auf dem FPGA-Board im wesentlichen durch das Pipelining schneller ist, ist es nicht sinnvoll, nur jeweils ein *i*- und ein *j*-Teilchen auf das Board zu laden. Der ursprüngliche Linked-Cell-List-Algorithmus (Listing 6.2) muss also so umgeschrieben werden, dass mehrere *i*- und *j*-Teilchen gleichzeitig auf das FPGA-Board geladen werden können. Dazu werden zwei Vektoren `i_particle` und `j_particle` angelegt, die die *i*- und *j*-Teilchen „sammeln“. Der Fortran-Quelltext zur Berechnung der Wechselwirkungen mit dem Linked-Cell-List-FPGA-Design befindet sich im Anhang (B.4).

Die Grundstruktur des Linked-Cell-List-Algorithmus wird dabei beibehalten. Die Wechselwirkungen werden zellenweise berechnet, indem zunächst Kräfte und potentielle Energie aller Teilchen der ersten Zelle, dann die der Teilchen in der zweiten Zelle usw. berechnet werden. Dabei werden jedoch nicht für jedes Teilchenpaar direkt Kraft und Energie ausgerechnet. Die Teilchen werden

stattdessen „gesammelt“. Im Vektor `i_particle` werden die Positionen der Teilchen gespeichert, die als *i*-Teilchen auf das FPGA-Board geladen werden. Die *i*-Teilchen sind diejenigen Teilchen, welche in die Register auf den FPGAs geladen werden und während einer Berechnung dort bleiben. Als *i*-Teilchen werden alle Teilchen der aktuellen Zelle gespeichert. `j_particle` enthält die Positionen der Teilchen, die als *j*-Teilchen betrachtet werden. Die *j*-Teilchen werden in den Speicher des FPGA-Boards geladen. In jedem Taktzyklus wird ein neues *j*-Teilchen an die Pipelines weitergegeben. Als *j*-Teilchen werden alle Teilchen betrachtet, die als Wechselwirkungspartner für die *i*-Teilchen in Frage kommen. Das sind zum einen alle Teilchen in der aktuellen Zelle und zum anderen die Teilchen aus den 26 Nachbarzellen.

In der ursprünglichen Linked-Cell-List-Implementierung wurde das dritte Newtonsche Axiom ausgenutzt. Das bedeutet, dass hierbei nur 13 Nachbarzellen betrachtet werden mussten und die berechnete Kraftkomponente zu der Kraft des *i*-Teilchens addiert und von der des *j*-Teilchens subtrahiert wurde. Wie schon in Kapitel 5.2 beschrieben, kann das dritte Newtonsche Axiom auf dem FPGA nicht ausgenutzt werden, weshalb hier alle 26 Nachbarzellen betrachtet werden müssen.

Wurden alle *i*- und *j*-Teilchen passend zur aktuellen Zelle „gesammelt“ werden sie auf das FPGA-Board geladen. Dabei wird wieder darauf geachtet, dass die maximal mögliche Teilchenzahl nicht überschritten wird (siehe Listing 5.3). Zudem besteht die Möglichkeit, von außen die Teilchenzahl weiter zu beschränken. Dies dient zur Untersuchung des Laufzeitverhaltens des Algorithmus in Abhängigkeit von der Teilchenzahl, die auf das FPGA-Board geladen wird. Die `force`-Routine liefert einen Vektor mit der berechneten Kraft und einen mit der potentiellen Energie für die *i*-Teilchen zurück. Konnten nicht alle *j*-Teilchen auf das FPGA-Board geladen werden, sind dies dementsprechend nur Teilergebnisse. Um sich zu merken, zu welchen Teilchen Kraft und Energie berechnet wurden, wird ein zusätzlicher Vektor (`save_i`) angelegt. Dieser Vektor speichert für jedes *i*-Teilchen den dazugehörigen Index. Das ist nötig, da die Teilchen einer Zelle nicht hintereinander im Speicher liegen. Mit Hilfe dieses Vektors lassen sich die berechneten Kraftkomponenten an der richtigen Stelle im Kraftvektor `f` abspeichern, bzw. auf die richtigen Teilkräfte aufaddieren. Die Energiekomponenten werden nach jeder FPGA-Berechnung aufaddiert, so dass sie am Ende die gesamte potentielle Energie im System ergeben.

In Kapitel 6.3 wurde noch erwähnt, dass die *Minimum Image Convention* vor der Berechnung auf dem FPGA-Board berücksichtigt werden kann. Dies wird im Programm mit Hilfe der Funktion `neigbor_cells` (siehe Anhang B.5) realisiert. Diese Funktion liefert einen Vektor (`nc_list`) zurück, der passend zur aktuellen Zelle die Nummern aller 26 Nachbarzellen enthält. Zudem wird eine Matrix (`mic`) zur Berücksichtigung der *Minimum Image Convention* mit Werten belegt. `mic` enthält passend zu jeder der 26 Nachbarzellen drei Werte. Ist die aktuelle Zelle keine Randzelle, sind alle Werte der Matrix gleich 0, da in diesem Fall nur Teilchen aus der Simulationsbox miteinander wechselwirken. Ist die aktuelle Zelle jedoch eine Randzelle, müssen für einige Nachbarzellen nicht die Originalzellen aus der Simulationsbox, sondern die entsprechenden Zellen aus den Bildboxen betrachtet werden. Dazu muss auf die entsprechenden Teilchenpositionen der dazugehörigen Teilchen die Länge der Simulationsbox addiert bzw. subtrahiert werden, je nachdem an welchem Rand sich die aktuelle Zelle befindet. Die Matrix `mic` enthält dazu für die Nachbarzelle den Wert `box_length` bzw. `-box_length` für die entsprechende Positionskoordinate. Bei der Speicherung eines Teilchens im Vektor `j_particle` wird dann auf jede der drei Positionskomponenten der zugehörige Wert aus dem `mic`-Vektor aufaddiert.

6.5 Ergebnisse

Für den PGDL-Code zum Linked-Cell-List-Algorithmus wurden mit ISE verschiedene Bitfiles erzeugt. Im Gegensatz zum Lennard-Jones-Design passen hier nur drei parallele Pipelines auf einen FPGA. Auf dem PROGRAPE-Board lassen sich also insgesamt 12 Pipelines platzieren. Die eingestellte Bitbreite für Fließkommazahlen beträgt dabei 26 Bit. Für drei parallele Pipelines pro FPGA ergibt sich eine maximale Zykluszeit von 14,96 ns. Die Taktfrequenz liegt also mit 66,84 MHz im erlaubten Bereich.

Es wurde ebenfalls versucht, Bitfiles für eine Bitbreite von 32 Bit zu erzeugen. Der kritische Pfad dieses Designs war jedoch zu lang. Die Zykluszeit betrug 15,60 ns. Die maximale Taktfrequenz liegt mit 64,12 MHz unter den 66 MHz, die für das PROGRAPE-4-Board erforderlich sind.

Zunächst wurde die Zeit zur Berechnung der Wechselwirkungen mit dem FPGA-Board und ohne das Board gemessen. Dies ergab einen Speedup bei drei parallelen Pipelines pro FPGA von 1,25. Dieser Faktor ist erstaunlich niedrig, vergleicht man ihn mit dem Faktor, der mit dem Lennard-Jones-FPGA-Design erzielt werden konnte. Dass ein Faktor von 110 mit diesem Design nicht erreichbar ist, wird klar indem man sich zunächst nur die Länge des PGDL-Codes der beiden Designs ansieht. So ist der Quelltext des Linked-Cell-List-Designs mit etwa 100 Zeilen doppelt so lang wie der 50 Zeilen lange PGDL-Code zum Lennard-Jones-Design. Das Pipelinedesign zum Linked-Cell-List-Algorithmus ist somit größer, als das Pipelinedesign zum Lennard-Jones-Potential. Diese Tatsache äußert sich vor allem in der Anzahl der maximal möglichen parallelen Pipelines pro FPGA. Beim Beispiel des Lennard-Jones-Potentials wurden auf dem FPGA lediglich die Kräfte berechnet. Auf einem FPGA ließen sich deshalb für die Kräfteberechnung maximal sechs parallele Pipelines unterbringen. Beim Linked-Cell-List-Design wurde neben den Kräften zusätzlich noch die potentielle Energie berechnet. Außerdem wurden Cutoff-Bedingung, *Minimum Image Convention* und die Korrektur durch das Shifted-force Potential berücksichtigt. Folglich lassen sich von diesem Design lediglich drei parallele Pipelines auf einem FPGA unterbringen. Der Größenunterschied des Pipelinedesigns kann jedoch nicht der einzige Grund für den enorm kleinen Faktor sein.

Die Teilchenzahl, die im Durchschnitt auf das FPGA-Board geladen wird, liefert eine weitere Erklärung. Bei einem System von 62.500 Teilchen beispielsweise werden pro Kräfteberechnung im Durchschnitt sieben i-Teilchen auf das FPGA-Board geladen. Das bedeutet es entsteht jedes Mal Latenz- und Übertragungszeit, ohne dass das FPGA-Board effizient ausgenutzt wird. Von den insgesamt 12 Pipelines auf dem PROGRAPE-Board werden im Durchschnitt lediglich sieben genutzt. Am naheliegensten ist es nun, die Zellen zu vergrößern, so dass pro Kräfteberechnung mehr i-Teilchen auf das FPGA-Board geladen werden. Die Zellengröße und somit die Anzahl der i-Teilchen kann jedoch nicht willkürlich vergrößert werden. Denn eine Erhöhung der i-Teilchenzahl bringt zwingenderweise auch eine Erhöhung der j-Teilchenzahl mit sich. So lassen sich mit einer größeren Anzahl von i-Teilchen zwar die Pipelines effizienter nutzen, die Anzahl der möglichen Wechselwirkungspartner für jedes i-Teilchen steigt jedoch. Der Sinn des Linked-Cell-List-Algorithmus ist es gerade, die Anzahl der potentiellen Wechselwirkungenpartner pro Wechselwirkungsberechnung möglichst gering zu halten. Außerdem wurde bereits erwähnt, dass If-Anweisungen auf dem FPGA sehr ineffizient realisiert werden, da in jedem Fall beide Anweisungen ausgeführt werden müssen. Je mehr „überflüssige“ j-Teilchen also auf das FPGA-Board geladen werden, desto mehr überflüssige Wechselwirkungsberechnungen werden durchgeführt. Es gilt also, eine optimale Teilchenzahl zu finden, für die die Berechnung mit dem FPGA-Board die größte Beschleunigung bringt.

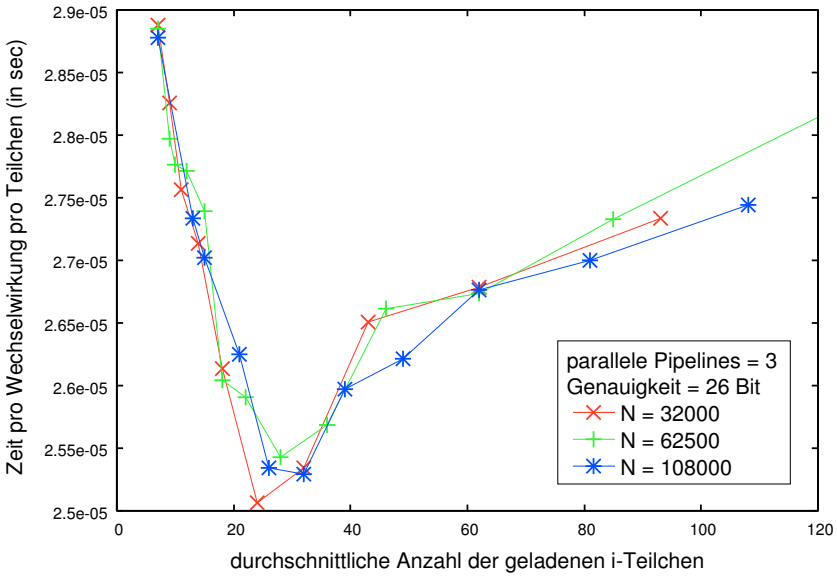


Abbildung 6.1: Zeit pro Wechselwirkungsberechnung für ein Teilchen in Abhängigkeit von der geladenen Teilchenzahl für drei parallele Pipelines

Aus diesem Grund wurde für verschiedene Zellengrößen die durchschnittliche Anzahl der *i*-Teilchen ermittelt, die jeweils gleichzeitig auf das FPGA-Board geladen werden. Für diese Teilchenzahlen wurde dann jeweils die benötigte Zeit zur Berechnung der Kräfte- und Energiekomponenten auf dem FPGA-Board gemessen. Daraus wurde die durchschnittliche Zeit ermittelt, die für eine Wechselwirkungsberechnung pro Teilchen benötigt wird. Die Zeit wurde für Systeme mit unterschiedlichen Teilchenzahlen gemessen. In Abbildung 6.1 sind die Ergebnisse für 32.000, 62.500 und 108.000 Teilchen dargestellt. Zum einen fällt auf, dass für alle drei Systemgrößen die Zeit zur Berechnung der Wechselwirkungen auf ein Teilchen bei einer Teilchenzahl zwischen 25 und 35 ihr Minimum annimmt. Mit dieser Teilchenzahl lässt sich der ursprüngliche Linked-Cell-List-Algorithmus um einen Faktor 1,31 beschleunigen.

6.5.1 Ausblick

Um den Speedup weiter zu vergrößern, könnte das FPGA-Design weiter optimiert werden. Durch Verkürzung der Signallaufzeiten könnte die Taktfrequenz des Designs auf 100 oder sogar 133 MHz erhöht werden. Diese Verkürzung könnte z. B. durch eine bessere Platzierung der Hardwarekomponenten auf dem FPGA erreicht werden. Zudem wäre noch interessant zu untersuchen, warum das 32-Bit-Format für zwei parallele Pipelines pro FPGA falsche Ergebnisse liefert. Ungeklärt ist außerdem, warum die Generierung eines Bitfiles für das 33-Bit-Fließkommaformat fehlschlug. Dies sind Probleme, deren Lösung eine detaillierte Hardwarekenntnis erfordert. Dies würde jedoch den Rahmen dieser Arbeit sprengen.

Ein weiterer Ansatz für eine höhere Beschleunigung wäre die softwareseitige Implementierung des Linked-Cell-List-Algorithmus. Dabei könnte der Algorithmus dahingehend verändert werden, dass pro FPGA-Berechnung mehr *i*-Teilchen geladen werden, die Anzahl der *j*-Teilchen jedoch so gering wie möglich gehalten wird.

6.6 Fazit

In diesem Abschnitt wird ein Fazit zur Implementierung eines Hardwaredesigns gegeben. Es werden die wichtigsten Punkte zusammengefasst. Dazu zählen neben den Ergebnissen auch die Schwierigkeiten und Probleme, die während der Implementierung auftraten. Die Punkte beziehen sich ausschließlich auf die Erfahrungen, die mit dem PROGRAPE-4-Board und PGR gesammelt wurden.

- Grundsätzlich ist die Entwicklung eines FPGA-Codesigns, bei dem bestimmte Teile eines Algorithmus auf der CPU und rechenintensive Teile auf einem FPGA-Board berechnet werden, sehr aufwändig. Zunächst müssen die zeitintensiven Teile eines Algorithmus ausgemacht und ein entsprechendes Hardwaredesign entwickelt werden. Die Softwareimplementierung muss anschließend an die Hardware angepasst werden. Wie im Falle des Linked-Cell-List-Algorithmus gilt es dann herauszufinden bei welcher Auslastung der FPGA bzw. das FPGA-Board die meiste Beschleunigung für den Algorithmus bringt.

- Trotz hochentwickelter Programmiersprachen wie etwa PGR, ist es immer ratsam über detaillierte Hardwarekenntnisse zu verfügen oder aber einen Hardwarespezialisten im Team zu haben. Meine Erfahrungen mit Hardwareprogrammierung haben gezeigt, dass die Schwierigkeit nicht darin besteht, die Funktionalität des Hardwaredesigns mit dem Emulator sicherzustellen. Die eigentlichen Schwierigkeiten treten bei der Generierung des Bitstroms auf, wenn die einzelnen Phasen Simulation, Synthese und *Place & Route* durchlaufen werden. Treten hierbei Fehler auf, wie in meinem Fall der zu lange kritische Pfad, ist der Softwareentwickler überfragt. Hier ist der Rat eines Hardwarespezialisten gefragt, wie sich die Zykluszeit verkürzen lässt. Des Weiteren kann es passieren, dass das entwickelte Hardwaredesign zu groß für den FPGA ist, weil beispielsweise zu viele LUTs oder zu viel Speicher benötigt werden. Ein Grund hierfür könnte sein, dass die einzelnen Logikbausteine nicht optimal auf dem FPGA platziert wurden. Die Lösung dieser Probleme setzt eine detaillierte Hardwarekenntnis voraus.

- Probleme können nicht nur bei der Bitstrom-Generierung auftreten. Auch mit einem korrekten Bitstrom können bei der Programmausführung Schwierigkeiten auftreten. So war bei meinen Testläufen immer wieder zu beobachten, dass die Berechnung mit dem FPGA-Board stockte, so dass die Programmausführung abgebrochen und neu gestartet werden musste. Dies geschah meist gegen Ende des Programms.

- Es ist immer nötig, die Ergebnisse, die eine Berechnung mit Hilfe von FPGAs liefert, auf ihre Korrektheit hin zu prüfen. Wie das Beispiel des Lennard-Jones-Potentials zeigt, können auch bei fehlerfrei generiertem Bitfile falsche Ergebnisse auftreten. Es ist daher sinnvoll die erzeugten Resultate mit reinen Softwareergebnissen zu vergleichen.

- Alles in allem zeigen die Beispiele in dieser Arbeit, dass sich der Einsatz von FPGAs für die Molekulardynamik durchaus lohnt. Für das einfache Lennard-Jones-Potential konnte ein Speedup von 110 erreicht werden. Für den Linked-Cell-List-Algorithmus betrug der Speedup zwar lediglich 1,3. Durch weitere Optimierung an der Software- und Hardwareimplementierung ließe sich der Speedup jedoch sicherlich noch erhöhen.

- In dieser Diplomarbeit wurden ausschließlich das PROGRAPE-4-Board und die Software PGR verwendet. Da die Möglichkeiten von PGR beschränkt sind, lassen sich unter Umständen mit anderer Software, wie beispielsweise Mitrion C oder einem anderen FPGA-Board bessere Ergebnisse erzielen.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde das FPGA-Board PROGRAPE-4 für molekulardynamische Berechnungen eingesetzt. Dazu wurde in einem ersten Schritt mit der Software PGR ein FPGA-Design zur Kräfteberechnung mittels des Lennard-Jones-Potentials entwickelt. Zur Ermittlung von Genauigkeit und Speedup der Berechnung mit dem FPGA-Board wurde ein Testprogramm geschrieben, dass die Kräfte einmal auf der CPU und einmal auf dem FPGA-Board berechnet.

Bei der Entwicklung des FPGA-Designs zeigte sich eine erste Einschränkung von PGR. Die Software ist darauf ausgerichtet, Pipelines zu erzeugen und erlaubt keinen Zugriff auf beliebige Elemente der geladenen Teilchenvektoren. Das dritte Newtonsche Axiom ließ sich auf dem FPGA-Board nicht ausnutzen. Es zeigte sich jedoch auch ein Vorteil von PGR. So unterstützt die Software implizite Vektoroperationen, indem für Vektoren automatisch die erforderliche Anzahl arithmetischer Module erzeugt wird. Weiter galt es zu beachten, dass die Anzahl der Teilchen begrenzt war, die gleichzeitig auf das FPGA-Board geladen werden konnte. So musste die Kräfteberechnung für größere Teilchenzahlen in mehreren Schritten erfolgen. Vom Lennard-Jones-Design ließen sich sechs parallele Pipelines auf einem FPGA unterbringen. Damit konnte ein Speedup von 110 erreicht werden.

Als nächstes sollte das FPGA-Board für den Linked-Cell-List-Algorithmus eingesetzt werden. Ein Problem hierbei war, dass die Kraftberechnung bei diesem Algorithmus nicht mehr in einer einfachen Doppelschleife erfolgt, sondern in mehrere kleinere Schleifen aufgeteilt ist. Die Lösung für dieses Problem war, die *i*- und *j*-Teilchen zu sammeln und zu mehreren auf das FPGA-Board zu laden. Dabei stellte sich die Frage nach der optimalen Anzahl von *i*-Teilchen, für die die Beschleunigung der Wechselwirkungsberechnung maximal ist. Der Linked-Cell-List-Algorithmus wurde passend zum Hardwaredesign umgeschrieben. Beim Hardwaredesign war zu beachten, dass neben der Kraft auch noch die potentielle Energie berechnet werden musste. Außerdem musste die Cutoff-Bedingung berücksichtigt werden. Da If-Anweisungen auf dem FPGA-Board sehr ineffizient realisiert werden, wurde die Abfrage der *Minimum Image Convention* nicht auf dem FPGA durchgeführt.

Von dem Linked-Cell-List-Design ließen sich drei parallele Pipelines auf einem FPGA unterbringen. Zunächst konnte damit ein Speedup von 1,25 erreicht werden. Um den Speedup zu verbessern, wurde die Zeit gemessen, die bei einer bestimmten Anzahl geladener Teilchen für die Wechselwirkungsberechnung pro Teilchen benötigt wird. Für ein System von 62.500 Teilchen ergab sich,

dass bei einer Anzahl von durchschnittlich 40 geladenen Teilchen die Zeit minimal ist. Mit dieser Teilchenzahl konnte der Speedup auf 1,31 verbessert werden.

Grundsätzlich lässt sich sagen, dass sich der Einsatz von FPGAs für molekulardynamische Berechnungen durchaus lohnt. Jedoch sind auch die Probleme, die bei der Entwicklung eines FPGA-Codesigns auftreten können, nicht zu vernachlässigen. In meinem Fall waren es Probleme, wie die zu geringe Taktfrequenz durch einen zu langen kritischen Pfad oder die Berechnung falscher Werte trotz eines fehlerfrei generierten Bitfiles. Deshalb ist es immer zu empfehlen, trotz hochentwickelter Programmiersprachen über Hardwarekenntnisse oder aber einen Hardwarespezialisten zu verfügen. Zudem sollten die mit dem FPGA berechneten Werte immer auf ihre Korrektheit hin überprüft werden.

In dieser Diplomarbeit wurde ausschließlich das PROGRAPE-4-Board und die dazugehörige Software PGR benutzt. Die damit erzielten Ergebnisse lassen sich nicht auf andere FPGA-Boards übertragen. Da das PROGRAPE-4-Board zur Berechnung langreichweitiger Wechselwirkungen entwickelt wurde und zudem nur für Berechnungen mit einfacher Genauigkeit oder weniger geeignet ist, wäre es interessant zu untersuchen, ob sich mit anderen FPGA-Boards und Programmiersprachen bessere Ergebnisse erzielen lassen. Des Weiteren ließe sich das Linked-Cell-List-FPGA-Design weiter optimieren. So könnte versucht werden, das Design auf Taktfrequenzen von 100 oder sogar 133 MHz zu beschleunigen.

Glossar

Die hier aufgeführten Begriffe sind bei ihrem ersten Erscheinen im Text *kursiv* gedruckt.

A

ASIC Ein ASIC (***A**pplication **S**pecific **I**ntegrated **C**ircuit*) oder eine anwendungsspezifische integrierte Schaltung ist eine elektronische Schaltung, die auf einem Siliziumchip platziert wird. ASICs werden von verschiedenen Herstellern speziell nach den jeweiligen Kundenanforderungen gefertigt. Aufgrund dieser Spezialisierung werden sie meist nur für eine einzige Modellreihe eingesetzt. Im Unterschied zu PLDs und FPGAs kann die interne Schaltung der ASICs vom Anwender nicht mehr geändert werden. ASICs finden Verwendung in nahezu allen elektronischen Geräten, vom Radiowecker bis zum Hochleistungsrechner.

C

CDC 3600 Die CDC 3600 besaß eine Taktrate von 0,7 MHz und eine Grenzleistung von 0,0013 GFlops. Der Großrechner arbeitete mit einer Wortlänge von 48 Bit und besaß 32 kByte Speicher. Seymour Cray entwickelte den Schaltkreis. Im Juni 1963 wurde die CDC 3600 zum ersten Mal ausgeliefert. 1966 wurde ihre Produktion eingestellt.

Cray T3E-1200 Die Cray T3E-1200 löste 1995 die Cray T3D ab. Sie war ein massiv paralleler Mehrprozessor-Rechner mit verteiltem Speicher. Die 512 Prozessoren wurden mit einer Taktrate von 600 MHz getaktet und besaßen jeweils 256 MByte Hauptspeicher.

Cray XD1 Die Cray XD1 im Zentralinstitut für angewandte Mathematik (ZAM) des Forschungszentrums Jülich ist ein Multiprozessorsystem. Es besteht aus 60 SMP-Knoten, die in 10 Chassis angeordnet sind. Jeder Knoten besitzt zwei AMD Opteron Prozessoren. Die Prozessoren sind mit 2.2 GHz getaktet und besitzen eine Höchstleistung von 4.4 GFlops. Insgesamt besitzt die Cray XD1 eine Speicherkapazität von 240 GB und erreicht eine Höchstleistung von 528 GFlops. Außerdem besitzt sie sechs Virtex-4 FPGAs (Typ: XC4VLX160) und sechs Virtex-II-Pro FPGAs (Typ: XC2VP50) [5].

E

EEPROM Ein EEPROM (***E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory*) ist ein nichtflüchtiger, elektronischer Speicherbaustein, der mit beliebigen Daten gefüllt und wieder gelöscht werden kann. EEPROMs werden verwendet, wenn Daten netzausfallsicher gespeichert werden müssen und selten geändert werden.

F

fcc-Gitter Mit Hilfe eines fcc-Gitters (fcc = ***f**ace **c**entered **c**ube*) lässt sich die dichteste Kugelpackung erreichen. Deshalb eignet sich ein solches Gitter besonders gut als Startkonfiguration für besonders dichte Systeme, wie z. B. Flüssigkeiten. Ein fcc-Gitter ist dreidimensional. Es wird erstellt, indem zunächst ein Einheitswürfel mit vier Teilchen konstruiert wird. Die Koordinaten der Teilchen sind $r_1 = (0,0,0)$, $r_2 = (0,a/2,a/2)$, $r_3 = (a/2,0,a/2)$ und $r_3 = (a/2,a/2,0)$, wobei a die Kantenlänge des Würfels ist. Dieser Einheitswürfel wird periodisch über die gesamte Simulationsbox in alle Richtungen fortgesetzt.

Flipflop Ein Flipflop ist eine elektronische Schaltung, die zwei stabile Zustände einnehmen und diese speichern kann. Flipflops lassen sich in zustandsgesteuerte und flankengesteuerte Flipflops einteilen, wobei zustandsgesteuerte Flipflops auch Latches genannt werden.

G

Gate Arrays Gate Arrays sind Logikschaltungen, die kundenspezifisch angefertigt werden. Auf einem solchen Chip befinden sich vorkonfigurierte Logik-Gatter (Gates) oder Logik-Blöcke. Sie sind in einer regelmäßigen Struktur (Array) angeordnet. Durch die Verschaltung dieser Gatter wird die anwendungsspezifische Funktion realisiert. Diese Verschaltung geschieht beim Halbleiterhersteller und kann nachträglich nicht geändert werden.

GPRS GPRS (***G**eneral **P**acket **R**adio **S**ervice*) (engl.: Allgemeiner paketorientierter Funkdienst) ist ein paketorientierter Übertragungsdienst, der im Bereich des Mobilfunks eingesetzt wird.

H

Hamiltonsche Mechanik Die Hamiltonsche Mechanik, die auch Hamiltonformalismus genannt wird, ist eine alternative Formulierung der klassischen Mechanik. Der Hamiltonformalismus beschäftigt sich mit der Herleitung von Bewegungsgleichungen für Probleme der klassischen Mechanik. Dabei werden Orte und Geschwindigkeiten als vollkommen unabhängige Variablen betrachtet.

I

IBM 704 Die IBM 704 war der erste in Massenfertigung hergestellte Großrechner mit eingebauten Schaltkreisen, der Gleitkommaarithmetik beherrschte. Eine weitere Neuheit war die Ersetzung der Williamsröhren durch magnetische Kernspeicher. Die IBM 704 wurde im April 1957 der Öffentlichkeit vorgestellt. Zu den Entwicklern gehörte unter anderem Gene Amdahl. Laut Angaben des Herstellers konnte die IBM 704 bis zu 40.000 Befehle pro Sekunde ausführen.

L

Latch Ein Latch ist ein zustandsgesteuertes Flipflop.

M

Manometer Ein Druckmessgerät oder auch Manometer (griech. manos = wacklig) ist die Bezeichnung für eine Messeinrichtung zum Erfassen und Anzeigen des physikalischen Druckes eines Mediums (Flüssigkeit, Gas).

N

NAS Ein NAS (***N**etwork **A**ttached **S**torage*) bezeichnet an das lokale Netzwerk angeschlossene Massenspeichereinheiten zur Erweiterung der Speicherkapazität.

P

Phasenraum Der Phasenraum ist ein $6N$-dimensionaler Raum, wobei N die Anzahl der Teilchen ist. Er wird von den Teilchenkoordinaten (je 3) und den Teilchenimpulsen (je 3) aufgespannt und enthält somit die komplette Beschreibung eines Mikrozustandes.

S

SAN Ein SAN (***S**torage **A**rea **N**etwork*) bezeichnet ein Netzwerk zur Anbindung von Festplattensubsystemen an Server-Systeme.

sequentielle Logik Als sequentielle Logik oder Schaltwerke werden digitale Schaltungen bezeichnet, die im Gegensatz zu kombinatorischer Logik oder Schaltnetzen Rückführungen von Ausgangssignalen zu Eingängen enthalten [9]. Im Gegensatz zur kombinatorischen Logik, haben die Gatter in Schaltwerken ein Laufzeitverhalten, so dass Ergebnisse nicht sofort beim Anlegen von Werten an den Eingängen sichtbar sind.

T

Taktfrequenz Die Taktfrequenz oder Taktrate (engl. *clock rate*) von Prozessoren bezeichnet den Rhythmus in dem Daten in Computern verarbeitet werden. Sie wird in Hertz (Hz) angegeben und berechnet sich, indem man den Kehrwert der Taktperiode bildet (Taktfrequenz = 1/Periodendauer).

V

Viskosimeter Ein Viskosimeter ist ein Messgerät zur Bestimmung der Zähigkeit (Viskosität) einer Flüssigkeit.

Literaturverzeichnis

[1] *ModelSim Homepage*. URL: http://www.model.com/

[2] *Synplicity Homepage*. URL: http://www.synplicity.com/

[3] *Wikipedia, die freie Enzyklopädie*. URL: http://de.wikipedia.org/wiki/Field_Programmable_Gate_Array

[4] *Xilinx Design Tools*. URL: http://www.xilinx.com/products/design_resources/design_tool/index.htm

[5] DETERT, U. *Cray XD1 Cluster Configuration*. URL: http://www.fz-juelich.de/zam/cray-xd1/configuration/

[6] ECKARDT, A. ; RÖSER, H. *VHDL, eine Möglichkeit der Algorithmenimplemetation in Hardware*. Vortragsfolien. Juni 2005

[7] ERNST, A. *N-Body Simulation - an Example for High-Performance-Computing and Visualization*. Vortragsfolien

[8] ERNST, A. ; SCHILLER, A. *Using FPGAs in the Cray XD1*. URL: http://www.fz-juelich.de/zam/cray-xd1/usage/fpga/

[9] FH-BIELEFELD. *Schaltungs-Synthese mit VHDL*. URL: http://wwwlrh.fh-bielefeld.de/vhdl_vor/VHDL_V_B.htm

[10] FRENKEL, D. ; SMIT, B.: *Understanding Molecular Simulation*. Bd. 1. Academic Press, 1996

[11] HAIRER, E. ; LUBICH, Ch. ; WANNER, G.: *Geometric Numerical Integration - Structure-Preserving Algorithms for ODEs*. Heidelberg : Springer, 2002

[12] HAMADA, T. *PGR: Processors Generator for Reconfigurable Systems*. URL: http://progrape.jp/pgr/

[13] HAMADA, T. *Introduction of the PGR software*. Dokumentation. Oktober 2005

[14] HAMADA, T. *PROGRAPE-4*. Vortragsfolien. Dezember 2006

[15] HAMADA, T. ; FUKUSHIGE, T. ; KAWAI, A. ; MAKINO, J.: PROGRAPE-1: A Programmable, Multi-Purpose Computer for Many-Body Simulations. 52 (2000), Oktober, S. 943–954

[16] HAMADA, T. ; NAKASATO, N.: PGR: A Software Package for Reconfigurable Super-Computing. In: *Field Programmable Logic and Applications, 2005. International Conference on*, 2005, S. 366 – 373

[17] HOLTSCHNEIDER, T.: *Modellanalyse der Nachbarschaftslisten-Technik für parallele Molekulardynamik-Simulationen*, Fachhochschule Aachen, Standort Jülich, Diplomarbeit, März 2005

[18] KORNMESSER, K.: *Das FPGA-Entwicklungssystem CHDL: Eine vollständige, C++-basierte Entwicklungsumgebung für FPGA-Koprozessoren*. Mannheim, Universität Mannheim, Diss., Oktober 2004

[19] KRAUTWURST, J.: *Entwicklung einer Software Bibliothek für schnelle Coulomb Löser*, Fachhochschule Aachen, Standort Jülich, Diplomarbeit, Februar 2006

[20] LIENHART, G.: *Beschleunigung Hydrodynamischer Astrophysikalischer Simulationen mit FPGA-Basierten Rekonfigurierbaren Koprozessoren*. Heidelberg, Ruprecht-Karls-Universität, Diss., Juli 2004

[21] PAPKE, T.: *Integrationsverfahren für molekulardynamische Simulationen: Untersuchungen zur Effizienz und Genauigkeit*, Fachhochschule Aachen, Standort Jülich, Diplomarbeit, Mai 2006

[22] PHILIPP, S.: *Entwicklung einer PCI-Karte zur Steuerung und Überwachung nicht-lokaler Rechner für den Einsatz in Computerclustern*, Ruprecht-Karls-Universität Heidelberg, Diplomarbeit, 2001

[23] SCHUMACHER, T. *FPGA accelerated High-Performance-Computing*. Vortragsfolien. Oktober 2006

[24] SUTMANN, G. *Computer Simulationsmethoden in den Naturwissenschaften: Molekulardynamische Simulationen*. URL: `http://www.fz-juelich.de/zam/ausbildung/tm/informationen/comp_sim_md`

[25] SUTMANN, G.: Classical Molecular Dynamics. In: GROTENDORST, J. (Hrsg.) ; MARX, D. (Hrsg.) ; MURAMATSU, A. (Hrsg.): *Quantum Simulations of Many-Body Systems: From Theory to Algorithms* Bd. 10. Jülich, 2002, S. 211 – 254

[26] SUTMANN, G.: Molecular Dynamics - Vision and Realitx. In: GROTENDORST, J. (Hrsg.) ; BLÜGEL, S. (Hrsg.) ; MARX, D. (Hrsg.): *Computational Nanoscience - Do it yourself* Bd. 31. Jülich, 2006, S. 159 – 194

[27] SUTMANN, G. ; STEGAILOV, V.: Optimization of neighbor list techniques in liquid matter simulations . In: *Journal of Molecular Liquids* (2006)

[28] WAIN, R. ; BUSH, I. ; GUEST, M. ; DEEGAN, M. ; KOZIN, I. ; KITCHEN, C.: An overview of FPGAs and FPGA programming; Initial experiences at Daresbury / Council for the Central Laboratory of the Research Councils. 2006. – Forschungsbericht

[29] WANNEMACHER, M.: *Das FPGA-Kochbuch*. MITP-Verlag, 1998

[30] XILINX. *Getting Started with FPGAs*. URL: `http://www.xilinx.com/company/`

gettingstarted/index.htm

[31] XILINX: *Virtex-4 Family Overview*. URL: http://direct.xilinx.com/bvdocs/publications/ds112.pdf. Oktober 2006. – Datenblatt

[32] XILINX: *Virtex-4 User Guide*. URL: http://direct.xilinx.com/bvdocs/userguides/ug070.pdf. Oktober 2006. – User Guide

A Parametrisierte Arithmetische Module

Liste der Module für Gleitkomma-Operationen:

Funktion	Modulname	Beschreibung in C
Addition	pg_float_add	$z = x + y$
Addition (ohne Vorzeichen)	pg_float_unsigned_add	$z = abs(x) + abs(y)$
Subtraktion	pg_float_sub	$z = x - y$
Subtraktion (ohne Vorzeichen)	pg_float_unsigned_sub	$z = abs(x) - abs(y)$
Multiplikation	pg_float_mult	$z = x * y$
Division	pg_float_div	$z = x/y$
Quadrat	pg_float_square	$z = x * x$
Wurzel	pg_float_sqrt	$z = sqrt(x)$
Kehrwert	pg_float_recipro	$z = 1/x$
Negation	pg_float_negate	$z = -x$
Multiplikation mit 2er-Potenzen	pg_float_expadd	$z = x * pow(2.0, y)$
Vergleich	pg_float_compare	$flag = (x > y)\ ?\ 1 : 0$
Vergleich (ohne Vorzeichen)	pg_float_compare	$flag = (abs(x) > abs(y))\ ?\ 1 : 0$
Vergleich mit 0	pg_float_compare	$flag = (x > 0)\ ?\ 1 : 0$
Vergleich mit 0 (ohne Vorzeichen)	pg_float_compare	$flag = (abs(x) > 0)\ ?\ 1 : 0$
SPH-Kern	pg_float_fixwtable	$z = w(x)$

Liste der Module für logarithmische Operationen:

Funktion	Modulname	Beschreibung in C
Addition	pg_log_add	$z = x + y$
Addition (ohne Vorzeichen)	pg_log_unsigned_add	$z = abs(x) + abs(y)$
Addition (ohne Vorzeichen, Interpolation)	pg_log_unsigned_add_itp	$z = abs(x) + abs(y)$
Multiplikation/Division	pg_log_muldiv	$z = x * y/z = x/y$
Potenzieren	pg_log_shift	$z = pow(x, 1 << y)$
Multiplikation mit 2er-Potenzen	pg_log_expadd	$z = x * pow(2.0, y)$

Liste der Module für Festkomma-Operationen:

Funktion	Modulname	Beschreibung in C
Addition/Subtraktion	pg_fix_addsub	$z = x + y/z = x - y$
Multiplikation	pg_fix_mult	$z = x * y$
Multiplikation (ohne Vorzeichen)	pg_fix_unsigned_mult	$z = abs(x) * abs(y)$
Aufsummierung	pg_fix_accum	$z+ = x$

Liste der Module für Bit-Operationen:

Funktion	Modulname	Beschreibung in C
AND	pg_bits_and	$z = x \,\&\, y$
OR	pg_bits_or	$z = x \; or \; y$
XOR	pg_bits_xor	$z = x^y$
NOT	pg_bits_inv	$z = -x$
Verbindung	pg_bits_join	$z = (x << NBITY) \; or \; y$
Extrahieren	pg_bits_part	$z = x(NH - 1 \;\; downto \;\; NL)$
Rotieren	pg_bits_rotate	
Verzögerung	pg_bits_delay	fügt Verzögerungsregister zwischen x und y ein
Multiplexen	pg_bits_mux	$z = (flag == 0) \; ? \, x : y$

Liste der Module für Konvertierungs-Operationen:

Funktion	Modulname
von Fließkomma nach Festkomma	pg_conv_floattofix
von Festkomma nach logarithmisch	pg_conv_ftol
von Festkomma nach logarithmisch (Interpolation)	pg_conv_ftol_itp
von logarithmisch nach Festkomma	pg_conv_ltof
von logarithmisch nach Festkomma (Interpolation)	pg_conv_ltof_itp

B Quelltexte

B.1 PGDL-Code zum Gravitationspotential

```
1  #define NFLO 26
2  #define NMAN 16
3  #define NFIX 57
4  #define NACC 64
5  #define NEXAD 31
6  #define FSHIFT pow(2.0,(double)NEXAD)
7
8  /MEM xj[3] <= x[][] : float(NFLO, NMAN);
9  /MEM mj    <= m[]   : float(NFLO, NMAN);
10 /REG xi[3] <= x[][] : float(NFLO, NMAN);
11 /REG sx[3] => a[][] : fix(NACC);
12
13 /SCALE sx[3] : (1.0/FSHIFT);
14
15 /NVMP 1;
16 /NPIPE 1;
17
18 pg_float_sub           (xj,xi, dx,              NFLO,NMAN, 4);
19 pg_float_mult          (dx,dx, dx2,             NFLO,NMAN, 2);
20 pg_float_unsigned_add  (dx2[0],dx2[1], x2y2,    NFLO,NMAN, 4);
21 pg_float_unsigned_add  (dx2[2],x2y2,   r2,      NFLO,NMAN, 4);
22 pg_float_sqrt          (r2,r1,                  NFLO,NMAN, 3);
23 pg_float_mult          (r2,r1,r3,               NFLO,NMAN, 2);
24 pg_float_recipro       (r3,r3i,                 NFLO,NMAN, 2);
25 pg_float_mult          (r3i,mj,mf,              NFLO,NMAN, 2);
26 pg_float_expadd        (mf,fs,                  NEXAD,NFLO,NMAN,1);
27 pg_float_mult          (fs,dx,fx,               NFLO,NMAN, 2);
28 pg_float_fixaccum      (fx,sx,                  NFLO,NMAN, NFIX, NACC, 2);
```

B.2 PGDL-Code zum Lennard-Jones-Potential

```
1  /* ------------------------------------------------ Macrodeclarations */
2  #define NFLO 26
3  #define NMAN 16
4  #define NFIX 57
5  #define NACC 64
6
7  #define NST_ONE  1
8  #define NST_ADD  4
9  #define NST_SUB  4
10 #define NST_MULT 2
11 #define NST_RECI 5
12 #define NST_ACCM 2
13
14 #define NEXAD 31
15 #define FSHIFT pow(2.0,(double)NEXAD)
16
17 /* ------------------------------------------------ General Definitions */
18 /NVMP 1;
19 /NPIPE 1;
20
21 /* ------------------------------------------------------ API Definition */
22 /MEM xj[3]    <= xjj[][] : float(NFLO, NMAN);
23 /REG xi[3]    <= xii[][] : float(NFLO, NMAN);
24 /REG ia       <= a       : float(NFLO, NMAN);
25 /REG ib       <= b       : float(NFLO, NMAN);
26 /REG fi[3]    => f[][]   : fix(NACC);
27
28 /SCALE fi[3] : (1.0/FSHIFT);
29
30 /* ------------------------------------------------ Pipelinedescription */
31 // r2 = |xj-xi|**2
32 pg_float_sub          (xi,xj, dr,           NFLO,NMAN,NST_SUB);
33 pg_float_mult         (dr,dr, dr2,          NFLO,NMAN,NST_MULT);
34 pg_float_unsigned_add (dr2[0],dr2[1], x2y2, NFLO,NMAN,NST_ADD);
35 pg_float_unsigned_add (dr2[2],x2y2,   r2,   NFLO,NMAN,NST_ADD);
36
37 pg_float_mult         (r2,r2,r4,            NFLO,NMAN,NST_MULT);
38 pg_float_mult         (r4,r4,r8,            NFLO,NMAN,NST_MULT);
39 pg_float_recipro      (r8,r8i,              NFLO,NMAN,NST_RECI);
40 pg_float_mult         (r8i,ib,mf1,          NFLO,NMAN,NST_MULT);
41 pg_float_mult         (r4,r8,r12,           NFLO,NMAN,NST_MULT);
42 pg_float_mult         (r2,r12,r14,          NFLO,NMAN,NST_MULT);
43 pg_float_recipro      (r14,r14i,            NFLO,NMAN,NST_RECI);
44 pg_float_mult         (r14i,ia,mf2,         NFLO,NMAN,NST_MULT);
45
46 pg_float_sub          (mf2,mf1,mf,          NFLO,NMAN,NST_SUB);
47
48 pg_float_expadd       (mf,fs,               NEXAD,NFLO,NMAN,NST_ONE);
49 pg_float_mult         (fs,dr,fr,            NFLO,NMAN,NST_MULT);
50 pg_float_fixaccum     (fr,fi,               NFLO,NMAN,NFIX,NACC,NST_ACCM);
```

B.3 PGDL-Code zum Linked-Cell-List-Algorithmus

```
1  /* ------------------------------------------------ Macrodeclarations */
2  #define NFLO 26
3  #define NMAN 16
4  #define NFIX 57
5  #define NACC 64
6
7  #define NST_ONE  1
8  #define NST_ADD  4
9  #define NST_SUB  4
10 #define NST_MULT 4
11 #define NST_RECI 5
12 #define NST_SQRT 4
13 #define NST_ACCM 5
14
15 #define NEXAD 31
16 #define FSHIFT pow(2.0,(double)NEXAD)
17
18 /* ------------------------------------------------ General Definitions */
19 /NVMP 1;
20 /NPIPE 3;
21
22 /CONST izero <= 0.000000    : float(NFLO, NMAN);
23
24 /* ------------------------------------------------------ API Definition */
25 /MEM xj[3]     <= xjj[][]  : float(NFLO, NMAN);
26 /REG xi[3]     <= xii[][]  : float(NFLO, NMAN);
27 /REG ir_cut_sq <= r_cut_sq : float(NFLO, NMAN);
28 /REG ir_cut    <= r_cut    : float(NFLO, NMAN);
29 /REG ipot_rc   <= pot_rc   : float(NFLO, NMAN);
30 /REG ifor_rc   <= for_rc   : float(NFLO, NMAN);
31 /REG ia        <= a        : float(NFLO, NMAN);
32 /REG ib        <= b        : float(NFLO, NMAN);
33 /REG ic        <= c        : float(NFLO, NMAN);
34 /REG id        <= d        : float(NFLO, NMAN);
35 /REG izeros    <= zeros[3] : float(NFLO, NMAN);
36 /REG fi[3]     => f[][]    : fix(NACC);
37 /REG poti      => pot[]    : fix(NACC);
38
39 /SCALE fi[3] : (1.0/FSHIFT);
40 /SCALE poti  : (1.0/FSHIFT);
41
42 /* ------------------------------------------------ Pipelinedescription */
43 // r2 = |xj-xi|**2
44 pg_float_sub         (xi,xj, dr,             NFLO,NMAN,NST_SUB);
45 pg_float_mult        (dr,dr, dr2,            NFLO,NMAN,NST_MULT);
46 pg_float_unsigned_add(dr2[0],dr2[1], x2y2,   NFLO,NMAN,NST_ADD);
47 pg_float_unsigned_add(dr2[2],x2y2,   r2,     NFLO,NMAN,NST_ADD);
48
49 pg_float_sqrt        (r2,r,                  NFLO,NMAN,NST_SQRT);
50
51 // ----------
52 //    Force
53 // ----------
54
55 pg_float_mult        (r2,r2,r4,              NFLO,NMAN,NST_MULT);
56 pg_float_mult        (r4,r4,r8,              NFLO,NMAN,NST_MULT);
57 pg_float_recipro     (r8,r8i,                NFLO,NMAN,NST_RECI);
58 pg_float_mult        (r8i,ib,mf1,            NFLO,NMAN,NST_MULT);
```

```
59 pg_float_mult        (r4,r8,r12,             NFLO,NMAN,NST_MULT);
60 pg_float_mult        (r2,r12,r14,            NFLO,NMAN,NST_MULT);
61 pg_float_recipro     (r14,r14i,              NFLO,NMAN,NST_RECI);
62 pg_float_mult        (r14i,ia,mf2,           NFLO,NMAN,NST_MULT);
63
64 pg_float_sub         (mf2,mf1,mf,            NFLO,NMAN,NST_SUB);
65
66 pg_float_recipro     (r,ri,                  NFLO,NMAN,NST_RECI);
67 pg_float_mult        (ri,ifor_rc,cf1,        NFLO,NMAN,NST_MULT);
68 pg_float_sub         (mf,cf1,cf,             NFLO,NMAN,NST_SUB);
69
70 pg_float_expadd      (cf,fs,                 NEXAD,NFLO,NMAN,NST_ONE);
71
72 pg_float_compare     (r2,ir_cut_sq,flag,     NFLO,NMAN,NST_ONE);
73 pg_bits_mux          (flag,dr,izeros,ifdr,   NFLO,NST_ONE);
74
75 pg_float_mult        (fs,ifdr,fr,            NFLO,NMAN,NST_MULT);
76 pg_float_fixaccum    (fr,fi,                 NFLO,NMAN,NFIX,NACC,NST_ACCM);
77
78 // --------------------
79 //   Potential Energy
80 // --------------------
81
82 pg_float_recipro     (r12,r12i,              NFLO,NMAN,NST_RECI);
83 pg_float_mult        (r12i,id,mf3,           NFLO,NMAN,NST_MULT);
84 pg_float_mult        (r4,r2,r6,              NFLO,NMAN,NST_MULT);
85 pg_float_recipro     (r6,r6i,                NFLO,NMAN,NST_RECI);
86
87 pg_float_sub         (mf3,r6i,mf4,           NFLO,NMAN,NST_SUB);
88 pg_float_mult        (mf4,ic,mf5,            NFLO,NMAN,NST_MULT);
89
90 pg_float_sub         (mf5,ipot_rc,cp1,       NFLO,NMAN,NST_SUB);
91 pg_float_sub         (r,ir_cut,cp2,          NFLO,NMAN,NST_SUB);
92 pg_float_mult        (cp2,ifor_rc,cp3,       NFLO,NMAN,NST_MULT);
93 pg_float_add         (cp1,cp3,cp,            NFLO,NMAN,NST_ADD);
94
95 // if ( i != j )
96 pg_float_compz_abs   (r,flag1,               NFLO,NMAN,NST_ONE);
97
98 pg_bits_mux          (flag,cp,izero,ifcp1,   NFLO);
99 pg_bits_mux          (flag1,izero,ifcp1,ifcp,NFLO);
100 pg_float_expadd      (ifcp,fs2,              NEXAD,NFLO,NMAN,NST_ONE);
101 pg_float_fixaccum    (fs2,poti,              NFLO,NMAN,NFIX,NACC,NST_ACCM);
```

B.4 Fortran-Quelltext zum Linked-Cell-List-FPGA-Design

```
1  subroutine interaction( x, f, pot )
2    !------------------------------------------------------------------------
3    !  calculates forces and energy of the harmonic oscillator
4    !  forces are divided by the mass for use in integrator routine
5    !------------------------------------------------------------------------
6    real (kind = real_typ), dimension(:,:), intent(inout)    ::  x
7    real (kind = real_typ), dimension(:,:), intent(out)      ::  f
8    real (kind = real_typ), intent(out)                      ::  pot
9
10   real (kind = real_typ), dimension(:,:), allocatable      ::  i_particle
11   real (kind = real_typ), dimension(:,:), allocatable      ::  j_particle
12   real (kind = real_typ), dimension(:,:), allocatable      ::  f_fpga
13   real (kind = real_typ), dimension(:), allocatable, save  ::  potvek
14
15   real (kind = real_typ), dimension(26,3)   :: mic
16   real (kind = real_typ), dimension(3)      :: zeros
17   real (kind = real_typ)                    :: A,B,C,D
18
19   integer, dimension(:,:,:), allocatable  ::  head, cell_entry
20   integer, dimension(:), allocatable      ::  entry, save_i
21   integer, dimension(26,3)                ::  nc_list
22   integer  ::  i,j,k,m,n,dim
23   integer  ::  icx,icy,icz, jcx,jcy,jcz, ix,iy,iz, nc, ic, iff
24   integer  ::  num_i, num_j, i_index, j_index
25   integer  ::  i_begin, i_end, j_begin, j_end
26
27   dim = size(x,1)
28   n   = size(x,2)
29
30   A = exp1*4.0*epsilon*(sigma)**exp1
31   B = exp2*4.0*epsilon*(sigma)**exp2
32   C = 4.0*epsilon*(sigma)**exp2
33   D = (sigma)**exp2
34   zeros = 0.0
35   nc = box_length/r_cut
36
37   if( .not. allocated(head) )       allocate( head(nc,nc,nc) )
38   if( .not. allocated(cell_entry) ) allocate( cell_entry(nc,nc,nc) )
39   if( .not. allocated(entry) )      allocate( entry(n) )
40   if( .not. allocated(save_i) )     allocate( save_i(n) )
41   if( .not. allocated(i_particle) ) allocate(i_particle(dim,n))
42   if( .not. allocated(j_particle) ) allocate(j_particle(dim,n))
43   if( .not. allocated(f_fpga) )     allocate(f_fpga(dim,n))
44   if( .not. allocated(potvek) )     allocate(potvek(n))
45
46   f = 0.0
47   pot = 0.0
48   head = 0
49   cell_entry = 0
50   entry = 0
51
52   do i = 1, n
53      ix = (real(nc)*x(1,i))/box_length + 1
54      iy = (real(nc)*x(2,i))/box_length + 1
55      iz = (real(nc)*x(3,i))/box_length + 1
56      entry(i) = head(ix,iy,iz)
57      head(ix,iy,iz) = i
58      cell_entry(ix,iy,iz) = cell_entry(ix,iy,iz) + 1
```

```
59    enddo
60
61    ! loop over all cells
62    do icx = 1, nc
63       do icy = 1, nc
64          do icz = 1, nc
65             j = head(icx,icy,icz)
66             num_i = cell_entry(icx,icy,icz)
67             num_j = cell_entry(icx,icy,icz)
68             i_index = 0
69             j_index = 0
70
71             ! start loop over local cell
72             do
73                if( j == 0 ) exit
74
75                i_index = i_index + 1
76                j_index = j_index + 1
77                save_i(i_index) = j
78
79                i_particle(:,i_index) = x(:,j)
80                j_particle(:,j_index) = x(:,j)
81
82                j = entry(j)
83             enddo
84
85             if (num_i /= 0) then
86
87                nc_list = neighbor_cells(icx,icy,icz,nc,mic)
88                ! start loop over 26 neighbor cells
89                do ic = 1, 26
90                   jcx = nc_list(ic,1)
91                   jcy = nc_list(ic,2)
92                   jcz = nc_list(ic,3)
93                   j = head(jcx,jcy,jcz)
94
95                   num_j = num_j + cell_entry(jcx,jcy,jcz)
96                   do
97                      if( j==0 )exit
98                      j_index = j_index + 1
99                      j_particle(1,j_index) = x(1,j) + mic(ic,1)
100                     j_particle(2,j_index) = x(2,j) + mic(ic,2)
101                     j_particle(3,j_index) = x(3,j) + mic(ic,3)
102
103                     j = entry(j)
104                  enddo
105               enddo
106
107               ! ------------------------
108               ! Calculation on the FPGA
109               ! ------------------------
110               i_begin = 1
111               i_end = max_ipar
112
113               do
114                  if (i_end >= num_i) i_end = num_i
115
116                  j_begin = 1
117                  j_end = max_jpar
118
```

```
119                  do
120                     if (j_end >= num_j) j_end = num_j
121
122                     CALL force(j_particle(:,j_begin:j_end),
123                                i_particle(:,i_begin:i_end),&
124                                %VAL(r_cut_sq),%VAL(r_cut),%VAL(pot_rc),&
125                                %VAL(for_rc),%VAL(A),%VAL(B),%VAL(C),&
126                                %VAL(D),zeros,f_fpga(:,i_begin:i_end),&
127                                potvek(i_begin:i_end),%VAL(i_end-i_begin+1),&
128                                %VAL(j_end-j_begin+1))
129
130                     ! Write force to array f and accumulate potential energy
131                     do iff = i_begin, i_end
132                        pot = pot + potvek(iff)
133                        f(:,save_i(iff)) = f(:,save_i(iff)) + f_fpga(:,iff)
134                     enddo
135
136                     if (j_end == num_j) exit
137
138                     j_begin = j_begin + max_jpar
139                     j_end = j_end + max_jpar
140                  enddo
141
142                  if (i_end == num_i) exit
143
144                  i_begin = i_begin + max_ipar
145                  i_end = i_end + max_ipar
146               enddo
147            endif
148         enddo
149      enddo
150   enddo
151
152   pot = pot * cnv_energy_au2jmol / 2.0
153
154 end subroutine interaction
```

B.5 Fortran-Quelltext zur Ermittlung der Nachbarzellen

```
1  function neighbor_cells(ix,iy,iz,nc,mic)   result(nc_list)
2    !---------------------------------------------------------------------
3    !  calculates neighbor cells in the linked cell algorithm
4    !---------------------------------------------------------------------
5    integer, intent(in)     ::  ix,iy,iz,nc
6    real (kind = real_typ), dimension(:,:), intent(inout)   ::  mic
7    integer, dimension(26,3)  ::  nc_list
8    integer  ::  ix_m1, ix_p1, iy_m1, iy_p1, iz_m1, iz_p1
9
10   mic=0.0
11
12   ix_m1 = ix-1
13   if( ix_m1 < 1 ) then
14      ix_m1 = ix_m1+nc
15      mic(1,1) =-box_length; mic(4,1) =-box_length; mic(7,1) =-box_length
16      mic(11,1)=-box_length; mic(12,1)=-box_length; mic(13,1)=-box_length
17      mic(18,1)=-box_length; mic(21,1)=-box_length; mic(24,1)=-box_length
18   endif
19
20   iy_m1 = iy-1
21   if( iy_m1 < 1 ) then
22      iy_m1 = iy_m1+nc
23      mic(1,2) =-box_length; mic(2,2) =-box_length; mic(3,2) =-box_length
24      mic(13,2)=-box_length; mic(16,2)=-box_length; mic(17,2)=-box_length
25      mic(18,2)=-box_length; mic(19,2)=-box_length; mic(20,2)=-box_length
26   endif
27
28   iz_m1 = iz-1
29   if( iz_m1 < 1 )  then
30      iz_m1 = iz_m1+nc
31      mic(1,3) =-box_length; mic(2,3) =-box_length; mic(3,3) =-box_length
32      mic(4,3) =-box_length; mic(5,3) =-box_length; mic(6,3) =-box_length
33      mic(7,3) =-box_length; mic(8,3) =-box_length; mic(9,3) =-box_length
34   endif
35
36   ix_p1 = ix+1
37   if( ix_p1 > nc )  then
38      ix_p1 = ix_p1-nc
39      mic(3,1) = box_length; mic(6,1) = box_length; mic(9,1) = box_length
40      mic(14,1)= box_length; mic(15,1)= box_length; mic(16,1)= box_length
41      mic(20,1)= box_length; mic(23,1)= box_length; mic(26,1)= box_length
42   endif
43
44   iy_p1 = iy+1
45   if( iy_p1 > nc )  then
46      iy_p1 = iy_p1-nc
47      mic(7,2) = box_length; mic(8,2) = box_length; mic(9,2) = box_length
48      mic(10,2)= box_length; mic(11,2)= box_length; mic(14,2)= box_length
49      mic(24,2)= box_length; mic(25,2)= box_length; mic(26,2)= box_length
50   endif
51
52   iz_p1 = iz+1
53   if( iz_p1 > nc )  then
54      iz_p1 = iz_p1-nc
55      mic(18,3)= box_length; mic(19,3)= box_length; mic(20,3)= box_length
56      mic(21,3)= box_length; mic(22,3)= box_length; mic(23,3)= box_length
57      mic(24,3)= box_length; mic(25,3)= box_length; mic(26,3)= box_length
58   endif
```

```
59
60    nc_list(1,1) = ix_m1;  nc_list(1,2) = iy_m1;  nc_list(1,3) = iz_m1
61    nc_list(2,1) = ix   ;  nc_list(2,2) = iy_m1;  nc_list(2,3) = iz_m1
62    nc_list(3,1) = ix_p1;  nc_list(3,2) = iy_m1;  nc_list(3,3) = iz_m1
63    nc_list(4,1) = ix_m1;  nc_list(4,2) = iy   ;  nc_list(4,3) = iz_m1
64    nc_list(5,1) = ix   ;  nc_list(5,2) = iy   ;  nc_list(5,3) = iz_m1
65    nc_list(6,1) = ix_p1;  nc_list(6,2) = iy   ;  nc_list(6,3) = iz_m1
66    nc_list(7,1) = ix_m1;  nc_list(7,2) = iy_p1;  nc_list(7,3) = iz_m1
67    nc_list(8,1) = ix   ;  nc_list(8,2) = iy_p1;  nc_list(8,3) = iz_m1
68    nc_list(9,1) = ix_p1;  nc_list(9,2) = iy_p1;  nc_list(9,3) = iz_m1
69
70    nc_list(13,1) = ix_m1; nc_list(13,2) = iy_m1; nc_list(13,3) = iz
71    nc_list(12,1) = ix_m1; nc_list(12,2) = iy   ; nc_list(12,3) = iz
72    nc_list(11,1) = ix_m1; nc_list(11,2) = iy_p1; nc_list(11,3) = iz
73    nc_list(10,1) = ix   ; nc_list(10,2) = iy_p1; nc_list(10,3) = iz
74    nc_list(14,1) = ix_p1; nc_list(14,2) = iy_p1; nc_list(14,3) = iz
75    nc_list(15,1) = ix_p1; nc_list(15,2) = iy   ; nc_list(15,3) = iz
76    nc_list(16,1) = ix_p1; nc_list(16,2) = iy_m1; nc_list(16,3) = iz
77    nc_list(17,1) = ix   ; nc_list(17,2) = iy_m1; nc_list(17,3) = iz
78
79    nc_list(18,1) = ix_m1; nc_list(18,2) = iy_m1; nc_list(18,3) = iz_p1
80    nc_list(19,1) = ix   ; nc_list(19,2) = iy_m1; nc_list(19,3) = iz_p1
81    nc_list(20,1) = ix_p1; nc_list(20,2) = iy_m1; nc_list(20,3) = iz_p1
82    nc_list(21,1) = ix_m1; nc_list(21,2) = iy   ; nc_list(21,3) = iz_p1
83    nc_list(22,1) = ix   ; nc_list(22,2) = iy   ; nc_list(22,3) = iz_p1
84    nc_list(23,1) = ix_p1; nc_list(23,2) = iy   ; nc_list(23,3) = iz_p1
85    nc_list(24,1) = ix_m1; nc_list(24,2) = iy_p1; nc_list(24,3) = iz_p1
86    nc_list(25,1) = ix   ; nc_list(25,2) = iy_p1; nc_list(25,3) = iz_p1
87    nc_list(26,1) = ix_p1; nc_list(26,2) = iy_p1; nc_list(26,3) = iz_p1
88
89
90  end function neighbor_cells
```

Index