Dichtefunktionaltheorie für eine Flüssigkeit harter Scheiben auf Grafikkarten

Dissertation

zur Erlangung des Grades

"Doktor der Naturwissenschaften"

im Fachbereich 08, Physik, Mathematik und Informatik

der Johannes Gutenberg-Universität
in Mainz

Marlon Kai Ebert
geboren in Bingen am Rhein

Mainz, Juni 2012

Inhaltsverzeichnis

Abstract

Zweidimensionale Flüssigkeiten harter Scheiben sind in der Regel einfach zu simulieren, jedoch überraschend schwer theoretisch zu beschreiben. Trotz ihrer hohen Relevanz bleiben die meisten theoretischen Ansätze qualitativ. Hier wird eine Dichtefunktionaltheorie (DFT) vorgestellt, die erstmalig die Struktur solcher Flüssigkeiten bei hohen Dichten korrekt beschreibt und den Ansatz des Gefrierübergangs abbildet. Es wird gezeigt, dass der Ansatz der Fundamentalmaßtheorie zu einem solchen Funktional führt. Dabei werden sowohl Dichteverteilungen um ein Testteilchen als auch Zweiteilchen-Korrelationsfunktionen untersucht.

Grafikkarten bieten sehr hohe Recheneffizienz und ihr Einsatz in der Wissenschaft nimmt stetig zu. In dieser Arbeit werden die Vor- und Nachteile der Grafikkarte für wissenschaftliche Berechnungen erörtert und es wird gezeigt, dass die Berechnung der DFT auf Grafikkarten effizient ausgeführt werden kann. Es wird ein Programm entwickelt, dass dies umsetzt. Dabei wird gezeigt, dass die Ergebnisse einfacher (bekannter) Funktionale mit denen von CPU-Berechnungen übereinstimmen, so dass durch die Nutzung der Grafikkarte keine systematischen Fehler zu erwarten sind.

1. Einleitung

1.1. Motivation

Diese Arbeit untersucht einfache klassische Flüssigkeiten mit der Methode der Dichtefunktionalthorie (DFT). Die dabei auftretenden Berechnungen werden numerisch auf Grafikkarten gelöst.

Warum Dichtefunktionaltheorie?

Die Dichtefunktionaltheorie ist seit ihrer Einführung vor etwas über 50 Jahren beständig gewachsen und zu einem Standardhilfsmittel der Physik geworden. Mit ihrer Hilfe können wir Einsichten in die mikroskopischen Vorgänge in Flüssigkeiten gewinnen. Dies ist in einer Vielzahl von Fällen, beispielsweise an Übergängen zwischen zwei Flüssigkeiten oder einer Flüssigkeit und einem Gas oder auch bei der Adsorption einer Flüssigkeit an einer Wand, interessant. Von besonderem Interesse sind dabei Flüssigkeiten unter geometrischen Beschränkungen, wie z.B. die Limitierung auf zwei Dimensionen. Gerade bei der Beschränkung auf zwei Dimensionen wurden interessante Ergebnisse nahe des Phasenübergangs festgestellt, beispielsweise bei der Untersuchung von 2D-Ordnung in einem quasikristallinen Substrat [Mik08] [Sch08], Defekte in 2D-Kristallen [Eis05], Gleichgewichtsstrukturen von binären Kristallen [Law11], dem Übergang von der flüssigen zur festen Phase [Zah99] oder dem Glasübergang [Maz09] [Haj11].

Mit Hilfe der DFT kann man Dichteprofile, Korrelationsfunktionen, die Freie Energie und Phasenübergänge für solche Flüssigkeiten bestimmen. Obwohl der DFT-Formalismus sehr allgemein ist sind die Anwendungen vielfältig. Gerade in den letzten Jahren wurden große Fortschritte bei der Untersuchung von Flüssigkeiten harter Kugeln und ihren Mischungen sowie von Lennard-Jones-Flüssigkeiten gemacht

[Han10] [Rot10]. Auch Systeme weicher Kugeln [Bar87] und Polymer-Mischungen [Xu12] können erfolgreich mit Hilfe der DFT beschrieben werden. Zweidimensionale Systeme, insbesondere Systeme harter Scheiben, wurden allerdings bisher deutlich weniger untersucht.

Ein Vorteil der DFT gegenüber Simulationen besteht darin, dass sie eine numerische Näherung an ein analytisches Ergebnis darstellen. Dies erlaubt es beispielsweise, DFT-Ergebnisse als Ausgangsgleichungen für die Modenkopplungstheorie zu nutzen. Zudem ist die Lösung der in der DFT auftretenden Gleichungen meist deutlich schneller als eine Simulation. Die Berechnung eines Dichteprofils, auch für große Systeme, kann i.d.R. innerhalb von Minuten oder Stunden erfolgen.

Der wesentliche Nachteil der DFT ist die Tatsache, dass die meisten Freien Energiefunktionale nicht exakt bekannt sind. Die Suche nach Funktionalen, die die zu untersuchende Flüssigkeit korrekt beschreiben, ist nicht trivial und sehr aufwändig. Deshalb liegt der Fokus dieser Arbeit auf einfachen Flüssigkeiten, für die auch neue Funktionale untersucht werden können.

Wozu Berechnungen auf der Grafikkarte ausführen?

Die Möglichkeit, eine Grafikkarte für allgemeine Rechenoperationen zu nutzen, besteht erst seit wenigen Jahren. Grafikkarten bieten eine einzigartige Kombination aus sehr hoher Rechenleistung und niedrigen Anschaffungs- sowie Betriebskosten. Aus diesem Grund werden Grafikkarten immer öfter für wissenschaftliche Zwecke eingesetzt und sie sind auf dem Weg, ein Standardhilfsmittel in der rechnergestützten Forschung zu werden. Beispielsweise werden Grafikkarten zur Signalverarbeitung [Par11], in der Astronomie [Way09], der physikalischen Chemie [Put10], der Geophysik [Mic10], neuronalen Netzen [Nag09] und physikalischen Modellen wie dem Ising-Modell [Pre09] [Blo10] eingesetzt.

Eine breitere Adaption wird dabei bisher hauptsächlich durch zwei Probleme verhindert: Einerseits benötigt man eine spezielle Programmiersprache, um die Grafikkarte ansprechen zu können. Das in dieser Arbeit genutzte CUDA [CUD09] [CUD12] ist herstellerabhängig und funktioniert nur mit Karten von NVidia. Mit der Einführung von OpenCL [OCL12] wurde dieses Problem entschärft, eine spezielle Sprache bleibt jedoch weiterhin nötig. Damit ist ein Umstieg auf die Grafikkartenprogram-

mierung mit hohem Aufwand verbunden und für viele Projekte, die auf bestehenden Codestrukturen aufbauen, nicht praktikabel. Andererseits ist die Grafikkarte durch ihre Architektur beschränkt. Die hohe Leistung bei niedrigen Kosten wird durch einen sehr hohen Grad der Spezialisierung erreicht. Grafikkarten können nur effizient eingesetzt werden, wenn sich das Problem parallelisieren lässt und dabei gleichzeitig nicht zu speicherintensiv ist, da der verfügbare Arbeitsspeicher auf der Grafikkarte stark beschränkt ist.

Beide Punkte konnten im Rahmen dieser Arbeit überwunden werden. Für die hier durchgeführten Berechnungen wurde ein neues Programm geschrieben, es musste nicht auf bestehendem Code aufgebaut werden. Zudem treten bei der Lösung der DFT-Gleichungen leicht parallelisierbare Rechenschritte auf, die die Architektur der Grafikkarte sehr effizient ausnutzen können. Das Problem der Speicherbeschränkung ist für die DFT in drei Dimensionen relevant, da hier allerdings zweidimensionale Systeme behandelt werden, bieten moderne Grafikkarten ausreichenden Arbeitsspeicher, um auch große Systeme betrachten zu können.

Damit sind Grafikkarten sehr gut geeignet, um alle DFT-Berechnungen dieser Arbeit durchzuführen.

1.2. Aufbau

Wir starten mit der Beschreibung der klassischen Dichtefunktionaltheorie (DFT) in Kapitel 2. Dabei werden die theoretischen Grundlagen erläutert, der allgemeine Formalismus erklärt sowie einfache Beispielsysteme vorgestellt. Es folgt eine detailierte Herleitung von Rosenfelds “Fundamental Measure Theory”, die eine wesentliche Grundlage der modernen DFT darstellt. Abschließend werden Anwendungen der DFT erörtert.

In Kapitel 3 folgt die Einführung in die Grundlagen der Durchführung von Berechnungen auf Grafikkarten (“General-purpose computing on graphics processing units”, GPGPU). Der Aufbau einer modernen Grafikkarte und die Besonderheiten bei der Programmierung für die Grafikkarte werden vorgestellt. Dabei wird auf die genutzte Architektur sowie ihre Vor- und Nachteile eingegangen. Insbesondere wird die Relevanz der technisch bedingten Beschränkungen des GPGPU für die DFT geprüft.

Mit diesen Grundlagen können wir in Kapitel 4 Ergebnisse für ein einfaches Funktional auf der Grafikkarte gewinnen. Dies wird genutzt, um Performancevergleiche zwischen GPU und CPU zu erzielen und die numerische Genauigkeit der Grafikkarte zu bestimmen. Die wesentlichen Algorithmen für die Berechnung der DFT werden vorgestellt und ihre Umsetzung auf der Grafikkarte diskutiert.

Dies erlaubt es uns, in Kapitel 5 das bekannte Rosenfeld-Funktional in zwei Dimensionen abzuleiten und Dichteverteilungen damit zu berechnen. Die Ergebnisse können mit der Literatur verglichen werden, so dass die korrekte Arbeit der Grafikkarte bestätigt wird. Die Schwachstellen des Rosenfeld-Funktionals, insbesondere die Eigenschaft, den Gefrierübergang nicht abzubilden, werden durch explizite Berechnung gezeigt.

Um den Gefrierübergang näher untersuchen zu können leiten wir in Kapitel 6 ein neues Tensorfunktional her. Basierend auf diesem Funktional werden Dichteverteilungen und Korrelationsfunktionen bestimmt und es wird gezeigt, dass das neue Funktional in der Lage ist, den Übergang zur Gitterformation bei hohen Dichten zu reproduzieren.

In Kapitel 7 werden die wesentlichen Ergebnisse dieser Arbeit zusammengefasst.

2. Dichtefunktionaltheorie

2.1. Entwicklung der Dichtefunktionaltheorie (DFT)

2.1.1. Einführung in die DFT

Die Gleichgewichts-Dichtefunktionaltheorie wird in der klassischen statistischen Mechanik eingesetzt, um Dichteverteilungen, Korrelationsfunktionen, die Freie Energie und Phasenübergänge in Modellflüssigkeiten zu berechnen. In diesem Kapitel wird der allgemeine DFT-Formalismus vorgestellt und die Anwendung auf ein zweidimensionales System harter Scheiben diskutiert.

Grundlagen der klassischen DFT

Die Einführung von Funktionalen in die statistische Physik kann auf [Bog46] zurückgeführt werden. Die Behandlung von inhomogenen Flüssigkeiten basiert wesentlich auf den Arbeiten [Mor61], [Dom62], [Sti62] und [Leb63]. Bekannt wurden funktionale Methoden zur Beschreibung von Flüssigkeiten durch [Per64]. Der unmittelbare Vorgänger der der klassischen DFT ist die quantenmechanische DFT, die erstmals in [Koh65] formuliert wurde. Sie wird genutzt um den quantenmechanischen Grundzustand eines Vielelektronensystems zu bestimmen und nutzt dabei die ortsabhängige Dichte der Elektronen im System. Damit kann der Grundzustand bestimmt werden, ohne die vollständige Schrödingergleichung lösen zu müssen. Im gleichen Jahr schuf Mermin in [Mer65] die Grundlagen für eine Erweiterung auf klassische Systeme. Eine erste umfassende Beschreibung der klassischen DFT ist in [Eva79] zu finden. Diese klassische DFT basiert auf der Erkenntnis, dass ein Funktional $\tilde{\Omega}[\rho]$ abhängig von der Dichteverteilung $\rho(\mathbf{r})$ einer Flüssigkeit existiert, dessen Minimum das Großkanonische Potential ist. Dazu nutzt man das Freie Energiefunktional $F[\rho]$, welches unabhängig von einem äußeren Potential $V(\mathbf{r})$ ist und dessen Form nur auf

den Eigenschaften der Flüssigkeit basiert. Ist dieses Funktional für eine Flüssigkeit bekannt, so kann die Dichteverteilung berechnet werden. Die klassische Dichtefunktionaltheorie ist zu einem Standardhilfsmittel in der statistischen Physik geworden, um thermodynamische Eigenschaften und Korrelationsfunktionen in (inhomogenen) Modellflüssigkeiten zu berechnen. Ihre allgemeinen Grundlagen werden hier [Eva09] und [Dwa11] folgend beschrieben.

Der DFT-Formalismus

Wir starten mit der Hamiltonfunktion einer n-atomigen Flüssigkeit

$$H_n = \sum_{i=1}^{n} \frac{\mathbf{p}_i^2}{2m} + \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_n) + \sum_{i=1}^{n} V(\mathbf{r}_i), \tag{2.1}$$

wobei der erste Term den kinetischen Energieanteil K, der zweite Term das interne Potential Φ und der dritte Term ein (beliebiges) externes Potential V beschreibt. Das Großkanonische Potential Ω ist durch

$$\Omega = -\beta^{-1} \ln Z_{gk} \tag{2.2}$$

definiert, wobei

$$Z_{gk} = \mathrm{Tr}_{\mathrm{kl}}\, e^{-\beta(H_n - \mu n)} = \sum_{n=0}^{\infty} e^{\beta\mu n} Z_k \tag{2.3}$$

die Großkanonische Zustandssumme abhängig vom chemischen Potential μ und $\beta = (k_B T)^{-1}$ ist. Der Operator $\mathrm{Tr}_{\mathrm{kl}}$ beschreibt die klassische Spur

$$\mathrm{Tr}_{\mathrm{kl}} = \sum_{n=0}^{\infty} \frac{1}{h^{3n} n!} \int d\mathbf{r}_1 \cdots \int d\mathbf{r}_n \int d\mathbf{p}_1 \cdots \int d\mathbf{p}_n. \tag{2.4}$$

Der Vorfaktor $\frac{1}{h^{3n}}$, in dem h die Planck-Konstant ist, wird aufgrund von Dimensionsüberlegungen eingeführt, so dass die Impulsfreiheitsgerade integriert werden können.

Definieren wir $u(\mathbf{r}) \equiv \mu - V(\mathbf{r})$ so erhalten wir eine Reihe von Korrelationsfunktionen durch die Differenzierung von Ω nach u. Die erste Ableitung ist die Dichte

$$\rho(\mathbf{r}) = \bar{\rho}(\mathbf{r}) = -\frac{\delta\Omega}{\delta u}. \tag{2.5}$$

Dabei ist $\bar{\rho}$ die mittlere Dichte. Die nächste Ableitung,

$$G(\mathbf{r}_1, \mathbf{r}_2) \equiv -\beta^{-1}\frac{\delta^2\Omega}{\delta u(\mathbf{r}_1)\ \delta u(\mathbf{r}_2)} = \langle(\rho(\mathbf{r}_1) - \bar{\rho}(\mathbf{r}_1))\,(\rho(\mathbf{r}_2) - \bar{\rho}(\mathbf{r}_2))\rangle\,, \tag{2.6}$$

produziert die Zweikörperdichtefluktuationsfunktion $G(\mathbf{r}_1, \mathbf{r}_2)$, wobei $\langle\rangle$ den Mittelwert über alle Teilchen beschreibt. Die Verbindung zur Zweikörperdichteverteilung $\rho^{(2)}$ ist durch

$$G(\mathbf{r}_1, \mathbf{r}_2) = \rho^{(2)}(\mathbf{r}_1, \mathbf{r}_2) - \rho(\mathbf{r}_1)\rho(\mathbf{r}_2) + \rho(\mathbf{r}_1)\delta(\mathbf{r}_1 - \mathbf{r}_2) \tag{2.7}$$

gegeben. Analog definieren Ableitungen höherer Ordnung Mehrteilchenfluktuationsfunktionen.

Das Großkanonische Potential kann durch

$$\Omega[f] = \mathrm{Tr}_{\mathrm{kl}}\, f(H_n + \mu n + \beta^{-1}\ln f) \tag{2.8}$$

dargestellt werden, wobei f eine beliebige Testwahrscheinlichkeitsverteilung ist, die die Normierung $\mathrm{Tr}_{\mathrm{kl}}\, f = 1$ erfüllt.

Betrachten wir die Gleichgewichts-Wahrscheinlichkeitsdichte im Großkanonischen Potential

$$f_0 = Z_{gk}^{-1}\,\mathrm{Tr}_{\mathrm{kl}}\, e^{-\beta(H_n - \mu n)} \tag{2.9}$$

und setzen sie in Gl. (2.8) ein so erhalten wir das Großkanonische Gleichgewichtspotential

$$\Omega[f_0] = \mathrm{Tr}_{\mathrm{kl}}\, f_0(H_n + \mu n + \beta^{-1}\ln f_0) = -\beta^{-1}\ln Z_{gk} \equiv \Omega_0. \tag{2.10}$$

Damit lässt sich Gl. (2.8) in

$$\Omega[f] = \Omega[f_0] + \beta^{-1}\,\mathrm{Tr}_{\mathrm{kl}}\, f \ln\left(\frac{f}{f_0}\right) \tag{2.11}$$

zerlegen. Mit der Gibbs-Ungleichung folgt

$$f \ln\left(\frac{f}{f_0}\right) < f\left(\frac{f}{f_0} - 1\right) \tag{2.12}$$

und somit auch

$$\mathrm{Tr}_{\mathrm{kl}}\, f \ln\left(\frac{f}{f_0}\right) > \mathrm{Tr}_{\mathrm{kl}}(f - f_0). \tag{2.13}$$

Da sowohl f als auch f_0 normiert sind gilt $\mathrm{Tr}_{\mathrm{kl}}(f - f_0) = 0$ und der zweite Term in Gl. (2.11) muss positiv sein. Somit erfüllt das Funktional (2.8) das Variationsprinzip

$$\Omega[f] > \Omega[f_0], \quad f \neq f_0. \tag{2.14}$$

Auf f_0 aufbauend lässt sich das Freie Energiefunktional

$$F[f_0] = \mathrm{Tr}_{\mathrm{kl}}\left[f_0\left(K + \Phi + \beta^{-1} \ln f_0\right)\right] \tag{2.15}$$

definieren. Dieses Funktional ist zunächst von f_0 abhängig. Deshalb zeigen wir nun, dass es ein eindeutiges Funktional $F[\rho]$ gibt, für welches $\Omega_0 = \Omega[f_0]$ gilt und das damit das Großkanonische Potential minimiert. Dazu starten wir mit einem alternativen Funktional,

$$F_L[\rho] = \min_{f \to \rho} \mathrm{Tr}_{\mathrm{kl}}\left[f\left(K + \Phi + \beta^{-1} \ln f\right)\right]. \tag{2.16}$$

Hierbei wird eine Levy-Minimierung durchgeführt und f ist eine beliebige Testwahrscheinlichkeitsverteilung, die die Normierung $\mathrm{Tr}_{\mathrm{kl}}\, f = 1$ erfüllt. Die Dichte kann aus der Wahrscheinlichkeitsverteilung über

$$\rho(\mathbf{r}) = \mathrm{Tr}_{\mathrm{kl}}\, f\tilde{\rho}(\mathbf{r}) \tag{2.17}$$

gewonnen werden. Dabei ist

$$\tilde{\rho}(\mathbf{r}) = \sum_{i=1}^{n} \delta(\mathbf{r} - \mathbf{r}_i) \tag{2.18}$$

der Dichteoperator. Eine Funktion f, die diese Minimierung für ein ρ_0 erfüllt, nennen wir $f_{min}^{\rho_0}$.

Das Großkanonische Potential für ein gegebenes externes Potential lautet damit

$$\Omega_L[\rho] = F_L[\rho] + \int d\mathbf{r}\ (V(\mathbf{r}) - \mu)\rho(\mathbf{r}). \tag{2.19}$$

Im Gleichgewicht muss dies in das Großkanonische Potential

$$\Omega_L[\rho_0] = \Omega_0 \tag{2.20}$$

übergehen. Da dieser Wert das Minimum des Potentials darstellt, muss

$$\Omega_L[\rho] \geq \Omega_0 \tag{2.21}$$

gelten. Für die Gleichgewichtsdichte gilt damit

$$F_L[\rho_0] = \mathrm{Tr}_{\mathrm{kl}}\left[f_{min}^{\rho_0}\left(K + \Phi + \beta^{-1}\ln f_{min}^{\rho_0}\right)\right]. \tag{2.22}$$

Nach der Definition von $f_{min}^{\rho_0}$ muss

$$\mathrm{Tr}_{\mathrm{kl}}\left[f_{min}^{\rho_0}\left(K + \Phi + \beta^{-1}\ln f_{min}^{\rho_0}\right)\right] \leq \mathrm{Tr}_{\mathrm{kl}}\left[f_0\left(K + \Phi + \beta^{-1}\ln f_0\right)\right] \tag{2.23}$$

erfüllt sein. Es kann gezeigt werden [Dwa11], dass mit Gl. (2.10) und Gl. (2.11) ebenfalls

$$\mathrm{Tr}_{\mathrm{kl}}\left[f_{min}^{\rho_0}\left(K + \Phi + \beta^{-1}\ln f_{min}^{\rho_0}\right)\right] \geq \mathrm{Tr}_{\mathrm{kl}}\left[f_0\left(K + \Phi + \beta^{-1}\ln f_0\right)\right] \tag{2.24}$$

gilt. Dies kann nur bei der Gleichheit

$$\mathrm{Tr}_{\mathrm{kl}}\left[f_{min}^{\rho_0}\left(K + \Phi + \beta^{-1}\ln f_{min}^{\rho_0}\right)\right] = \mathrm{Tr}_{\mathrm{kl}}\left[f_0\left(K + \Phi + \beta^{-1}\ln f_0\right)\right] \tag{2.25}$$

erfüllt sein, so dass mit Gl. (2.22) auch

$$F_L[\rho_0] = \mathrm{Tr}_{\mathrm{kl}}\left[f_0\left(K + \Phi + \beta^{-1}\ln f_0\right)\right]. \tag{2.26}$$

gelten muss. Setzt man Gl. (2.26) in den Ausdruck für Ω_0,

$$\Omega_0 = \int d\mathbf{r}\ (V(\mathbf{r}) - \mu)\rho_0(\mathbf{r}) + \mathrm{Tr}_{\mathrm{kl}}\left[f_0\left(K + \Phi + \beta^{-1}\ln f_0\right)\right], \tag{2.27}$$

ein, so erhalten wir die Gleichheit in Gl. (2.20). Das Freie Energiefunktional ist somit tatsächlich nur von ρ und nicht von f_0 abhängig,

$$F[\rho] = \mathrm{Tr}_{\mathrm{kl}}\left[f_0\left(K + \Phi + \beta^{-1}\ln f_0\right)\right]. \tag{2.28}$$

Über die Legendretransformation erhalten wir

$$\tilde{\Omega}[\tilde{\rho}] = F[\tilde{\rho}] - \int d\mathbf{r}\ u(\mathbf{r})\tilde{\rho}(\mathbf{r}). \tag{2.29}$$

Für $\tilde{\rho} = \rho$, d.h. bei der Gleichgewichtsdichte, geht $\tilde{\Omega}$ in das Großkanonische Potential über. Das Großkanonische Potential Ω minimiert $\tilde{\Omega}[\tilde{\rho}]$ und wir erhalten das Variationsprinzip

$$\left.\frac{\delta\tilde{\Omega}[\tilde{\rho}]}{\delta\tilde{\rho}(r)}\right|_{\tilde{\rho}=\rho} = 0. \tag{2.30}$$

Wir können eine zweite Reihe von Korrelationsfunktionen generieren, indem wir das Freie Energiefunktional F ableiten. Es bietet sich an, dieses dafür zunächst in den Anteil des idealen Gases (id) und den Rest („excess“, ex) aufzuteilen,

$$F[\rho] = F^{id}[\rho] + F^{ex}[\rho]. \tag{2.31}$$

Der ideale Gasanteil ist bekannt,

$$F^{id} = \int d\mathbf{r}\ \beta^{-1}\rho(\mathbf{r})\left(\ln\left(\lambda^3\rho(\mathbf{r})\right) - 1\right). \tag{2.32}$$

λ^3 ist die De-Broglie-Wellenlänge des Teilchens, die den Logarithmus dimensionslos werden lässt und die durch die Ausintegration der Impulsintegrale in Gl. (2.4) entsteht. Für Berechnungen wird ihr Betrag i.d.R. gleich 1 gesetzt. Durch die Differenzierung des Exzessanteils erhalten wir die direkten Korrelationsfunktionen

$$
\begin{aligned}
c^{(1)}(\mathbf{r}) &= -\frac{\delta\beta F^{ex}[\rho]}{\delta\rho(\mathbf{r})} \\
c^{(2)}(\mathbf{r}_1,\mathbf{r}_2) &= \frac{\delta c^{(1)}(\mathbf{r}_1)}{\delta\rho(\mathbf{r}_2)} = -\frac{\delta^2\beta F^{ex}[\rho]}{\delta\rho(\mathbf{r}_1)\;\delta\rho(\mathbf{r}_2)} \\
c^{(n)}(\mathbf{r}_1,\ldots,\mathbf{r}_n) &= \frac{\delta c^{(n-1)}(\mathbf{r}_1,\ldots,\mathbf{r}_{n-1})}{\delta\rho(\mathbf{r}_n)}.
\end{aligned}
\tag{2.33}
$$

Dabei ist $c^{(2)}$ die direkte Paarkorrelationsfunktion, die aus der Ornstein-Zernike-Relation bekannt ist.

Harte Stäbe in einer Dimension

Dieser Formalismus erlaubt es, durch die Freie Energie die Dichteverteilung zu bestimmen. Leider ist das Freie Energiefunktional meist nicht direkt zugänglich. Harte Stäbe der Länge l in einer Dimension sind ein exakt lösbares Modell, an dessen Beispiel wir die Lösung der DFT illustrieren.

Wir starten mit

$$
\tilde{\Omega}[\rho] = F^{id}[\rho] + F^{ex}[\rho] - \int dz\; u(z)\rho(z). \tag{2.34}
$$

Der Exzessanteil des Freien Energiefunktionals ist durch

$$
F^{ex}[\rho] = \beta^{-1}\int dz\; \rho(z)\ln\left(1-t(z)\right) \tag{2.35}
$$

mit

$$
t(z) = \int_{z-l}^{z} dy\; \rho(y) \tag{2.36}
$$

gegeben. Diese Form wurde von Robledo und Varea in [Rob81] hergeleitet.

Die Minimierung von $\tilde{\Omega}[\rho]$ führt zu

$$
\beta u(z) = \ln\frac{\rho(z)}{1-t(z)} + \int_{z}^{z+l} dy\; \frac{\rho(y)}{1-t(y)}, \tag{2.37}
$$

was der Percus-Gleichung für das Dichteprofil in einem beliebigen externen Po-

tential entspricht. Da Percus in [Per76] die gleiche Integralgleichung mit Hilfe von Funktionalableitungen des Großkanonischen Potentials ableiten konnte ist damit bewiesen, dass Gl. (2.35) das exakte Funktional für harte Stäbe darstellt.

Gl. (2.35) kann auch mit Hilfe von gewichteten Dichten ausgedrückt werden. Für harte Stäbe kann man eine Gewichtsfunktion definieren, die mit dem Ende der Stäbe assoziiert ist,

$$\omega_i^{(0)}(z) = \frac{1}{2}\left(\delta(z - R_i) + \delta(z + R_i)\right), \tag{2.38}$$

und eine Funktion, die die Länge des Stabes beschreibt,

$$\omega_i^{(1)}(z) = \Theta(R_i - |z|). \tag{2.39}$$

Mit Hilfe dieser Gewichtsfunktionen lassen sich gewichtete Dichten n_α konstruieren,

$$n_\alpha(z) = \int dz' \; \rho_i(z')\omega_i^{(\alpha)}(z - z'). \tag{2.40}$$

Damit lässt sich die der Exzessanteil des Freien Energiefunktionals harter Stäbe durch

$$F^{ex}[\rho] = \int dz \Phi\left(n_0(z), n_1(z)\right) \tag{2.41}$$

mit

$$\Phi\left(n_0(z), n_1(z)\right) = -n_0 \ln(1 - n_1) \tag{2.42}$$

ausdrücken.

Die simple Struktur des exakten Freien Energiefunktionals für harte Stäbe stellte eine wichtige Grundlage für Rosenfelds spätere Verallgemeinerung auf dreidimensionale harte Kugeln dar, die in Kapitel 2.1.2 beschrieben wird.

Approximative Freie Energiefunktionale: Die Gradientenentwicklung

Da das Freie Energiefunktional nicht für alle Systeme bekannt ist ist es nötig, approximative Funktionale zu finden. Das verbreiteteste approximative Funktional erhält

man, indem man die funktionale Taylorentwicklung der Exzessenergie in der zweiten Ordnung abbricht. Die Berechnung wird nach [Eva79] durchgeführt.

Wir starten mit der Annahme, dass die Freie Energiedichte sich durch eine Reihe von Gradienten ausdrücken lässt,

$$F^{ex}[\rho] = \int dr\ f[\rho] \tag{2.43}$$

und

$$f[\rho] = f_0(\rho(\mathbf{r})) + f_i(\rho(\mathbf{r}))\nabla_i\rho(\mathbf{r}) + f^{(1)}_{i,j}(\rho(\mathbf{r}))\nabla_i\rho(\mathbf{r})\nabla_j\rho(\mathbf{r}) + f^{(2)}_{i,j}\nabla_i\nabla_j\rho(\mathbf{r}) + \dots. \tag{2.44}$$

Da $f[\rho]$ als Funktional von ρ unabhängig vom externen Potential $V(\mathbf{r})$ ist muss f invariant unter Rotationen um $\mathbf{r}$ sein. Damit kann die Form von $f[\rho]$ vereinfacht werden,

$$f[\rho] = f_0(\rho) + f_2^{(1)}(\rho)\boldsymbol{\nabla}^2\rho + f_2^{(2)}(\rho)\boldsymbol{\nabla}\rho\cdot\boldsymbol{\nabla}\rho + \mathrm{O}(\boldsymbol{\nabla}^4). \tag{2.45}$$

Nutzt man, dass

$$f_2^{(1)}(\rho)\boldsymbol{\nabla}^2\rho = \boldsymbol{\nabla}\cdot\left(f_2^{(1)}(\rho)\boldsymbol{\nabla}\rho\right) - \frac{\partial f_a^{(1)}(\rho)}{\partial\rho}\boldsymbol{\nabla}\rho\cdot\boldsymbol{\nabla}\rho \tag{2.46}$$

gilt so sieht man, dass der Divergenzterm bei der Integration von $f[\rho]$ über $\mathbf{r}$ verschwinden wird. Folglich gilt

$$f[\rho] = f_0(\rho) + f_2(\rho)\boldsymbol{\nabla}\rho\cdot\boldsymbol{\nabla}\rho + \mathrm{O}(\boldsymbol{\nabla}^4). \tag{2.47}$$

Der Koeffizient f_0 kann sofort als die Freie Energiedichte einer gleichförmigen Flüssigkeit der Dichte ρ interpretiert werden. Um den Koeffizienten f_2 zu bestimmen betrachten wir die funktionale Taylorentwicklung von $F[\rho]$,

$$F[\rho] = F[\rho_0] + \int d\mathbf{r}\ \frac{\delta F[\rho]}{\delta\rho(\mathbf{r})}\bigg|_{\rho_0} \Delta\rho(\mathbf{r}) + \frac{1}{2}\int d\mathbf{r}\ d\mathbf{r}'\ \frac{\delta^2 F[\rho]}{\delta\rho(\mathbf{r})\delta\rho(\mathbf{r}')}\bigg|_{\rho_0} \Delta\rho(\mathbf{r})\Delta\rho(\mathbf{r}') + \ldots \tag{2.48}$$

mit $\Delta\rho(\mathbf{r}) = \rho(\mathbf{r}) - \rho_0$. In Gl. (2.33) sehen wir, dass die zweite Ableitung von $F[\rho]$ mit der $c^{(2)}$-Funktion verknüpft ist. In [Eva79] wird gezeicht, dass ein Vergleich der Entwicklungen von $F[\rho]$ auf

$$f_2\left(\rho(r)\right) = (12\beta)^{-1} \int d\mathbf{r}\ r^2 c^{(2)}(\rho; r) \tag{2.49}$$

führt. In einer ersten Näherung ist das Freie Energiefunktional somit durch die bekannte $c^{(2)}$-Funktion definiert.

Der Abbruch der Entwicklung in der zweiten Ordnung, d.h. nur unter Betrachtung der direkten Paarkorrelationsfunktion und unter Vernachlässigung höherer Mehrteilchenkorrelationsfunktionen, ist eine einfache, allerdings keine besonders gute Näherung. Sie ist nur im Bereich langsam variierender Dichten und für Systeme, die sich gut über Mean-field-Modelle nähern lassen, gültig. Ionisierte Systeme werden beispielsweise sehr gut in dieser Näherung beschrieben. Sie ist jedoch nicht geeignet, um Phasenübergänge und Systeme mit hoher Dichte zu beschreiben. In Kapitel 4 werden wir dieses Funktional nutzen, um die Geschwindigkeit der Rechnung auf der Grafikkarte im Vergleich zur CPU zu bestimmen.

2.1.2. "Fundamental Measure Theory" für ein System harter Kugeln und Scheiben

Rosenfelds "Fundamental Measure Theory" (FMT) wurde ursprünglich für ein System harter Kugeln in drei Dimensionen entwickelt. Sie basiert auf der Idee, dass sich das Freie Energiefunktional mit Hilfe von Gewichtsfunktionen beschreiben lässt, die wiederum mit den Fundamentalmaßen der Kugeln, Volumen, Oberfläche und Radius, verknüpft werden können.

Grundlagen der FMT

Für niedrige Dichten $\rho(\mathbf{r}) \to 0$ ist

$$\beta F^{ex}[\rho] = -\frac{1}{2}\int d\mathbf{r} \int d\mathbf{r'}\ \rho_i(\mathbf{r})\rho_j(\mathbf{r'})f_{ij}(|\mathbf{r}-\mathbf{r'}|) \tag{2.50}$$

das exakte Freie Energiefunktional, wobei

$$f_{ij}(r) = e^{-\beta\Phi_{ij}(r)} - 1 \tag{2.51}$$

den Mayer f-Bond und Φ_{ij} das Potential zwischen den Teilchen i und j darstellt. Für harte Kugeln mit Radius R_i und R_j vereinfacht sich dies zu

$$f_{ij}(\mathbf{r}) = -\Theta(R_i + R_j - r). \tag{2.52}$$

Dieses Mayer f-Bond kann mit Hilfe von Gewichtsfunktionen nach [Ros89] zerlegt werden,

$$\begin{aligned} -f_{ij} = {} & \omega_i^{(3)} \otimes \omega_j^{(0)} + \omega_i^{(0)} \otimes \omega_j^{(3)} + \omega_i^{(2)} \otimes \omega_j^{(1)} \\ & + \omega_i^{(1)} \otimes \omega_j^{(2)} - \boldsymbol{\omega}_i^{(1)} \otimes \boldsymbol{\omega}_j^{(2)} - \boldsymbol{\omega}_i^{(2)} \otimes \boldsymbol{\omega}_j^{(1)}. \end{aligned} \tag{2.53}$$

Der Operator $\otimes$ beschreibt die Konvolution

$$\omega_i^{(a)} \otimes \omega_j^{(b)} = \int d\mathbf{x}\ \omega_i^{(a)}(\mathbf{r}_i - \mathbf{x}) \cdot \omega_j^{(b)}(\mathbf{r}_j - \mathbf{x}). \tag{2.54}$$

Die Gewichtsfunktionen sind dabei durch

$$\begin{aligned}
\omega_i^{(3)}(\mathbf{r}) &= \Theta(R_i - r), \quad & \omega_i^{(2)}(\mathbf{r}) &= \delta(R_i - r), \\
\omega_i^{(1)}(\mathbf{r}) &= \frac{\omega_i^{(2)}}{4\pi R_i}, & \omega_i^{(0)}(\mathbf{r}) &= \frac{\omega_i^{(2)}}{4\pi R_i^2}, \\
\boldsymbol{\omega}_i^{(2)}(\mathbf{r}) &= \frac{\mathbf{r}}{r}\omega_i^{(2)}, & \boldsymbol{\omega}_i^{(1)}(\mathbf{r}) &= \frac{\boldsymbol{\omega}_i^{(2)}}{4\pi R_i}
\end{aligned} \tag{2.55}$$

gegeben. Die Gewichtsfunktionen beschreiben eine Kugel mit Radius R_i. Integriert man sie, so wird der Zuammenhang mit den Fundamentalmaßen der Kugel deutlich: Das Integral über $\omega_i^{(3)}(\mathbf{r})$ liefert das Volumen $V_i = \frac{4}{3}\pi R_i^3$, $\omega_i^{(2)}(\mathbf{r})$ integriert ergibt die Oberfläche $A_i = 4\pi R_i^2$, $\omega_i^{(1)}(\mathbf{r})$ die mittlere Krümmung (hier also den Radius) und $\omega_i^{(0)}(\mathbf{r})$ die Euler-Charakteristik, die für Kugeln 1 ist.

Durch diese Gewichtsfunktionen werden sechs gewichtete Dichten

$$n_\alpha(\mathbf{r}) = \int d\mathbf{r}' \; \rho_i(\mathbf{r}')\omega_i^{(\alpha)}(\mathbf{r} - \mathbf{r}') \tag{2.56}$$

analog zum eindimensionalen Fall definiert. Wie für das System harter Stäbe wird der Ansatz

$$\beta F^{ex}[\rho] = \int dz \Phi\left(n_\alpha(\mathbf{r})\right) \tag{2.57}$$

genutzt. Über einen Dimensionenvergleich können alle zulässigen Terme für Φ identifiziert werden:

$$\begin{aligned}
\Phi\left(n_\alpha(\mathbf{r})\right) = {} & f_1(n_3)n_0 + f_2(n_3)n_1 n_2 + f_3(n_3)\mathbf{n}_1 \cdot \mathbf{n}_2 \\
& + f_4(n_3)n_2^3 + f_5(n_3)n_2 \mathbf{n}_2 \cdot \mathbf{n}_2.
\end{aligned} \tag{2.58}$$

Um die Koeffizienten f_α zu fixieren nutzte Rosenfeld aus, dass im Grenzwert $\rho \to 0$ Gl. (2.50) erfüllt sein muss sowie die Bedingung

$$\lim_{R_i \to \infty} \left(\frac{\mu_i^{ex}}{V_i} \right) = p, \tag{2.59}$$

welche aussagt, dass dass Chemische Exzesspotential, welches durch die Einführung einer neuen großen Kugel in das System entsteht, pV_i entspricht. Zudem wird gefordert, dass $c^{(2)} = -\frac{\delta^2 F[\rho]}{\delta\rho(\mathbf{r})\delta\rho(\mathbf{r})}$ bis in $\mathrm{O}(\rho^3)$ exakt sein muss. Mit Hilfe dieser Be-

dingungen konnte Rosenfeld die Koeffizienten als

$$\begin{aligned} f_1(n_3) &= -\ln(1-n_3), \quad f_2(n_3) = \frac{1}{1-n_3}, \quad f_3(n_3) = -f_2(n_3), \\ f_4(n_3) &= \frac{1}{24\pi(1-n_3)^3}, \quad f_5(n_3) = -3f_4(n_3) \end{aligned} \tag{2.60}$$

bestimmen. Damit wird der Exzessanteil des Freien Energiefunktionals zu

$$\begin{aligned} \beta F^{ex} &= \int d\mathbf{r} \ (\Phi_1 + \Phi_2 + \Phi_3) \\ &= \int d\mathbf{r} \ \left(-n_0 \ln(1-n_3) + \frac{n_1 n_2 - \mathbf{n}_1 \mathbf{n}_2}{1-n_3} + \frac{n_2^3 - 3n_2 \mathbf{n}_2 \cdot \mathbf{n}_2}{24\pi(1-n_3)^2} \right). \end{aligned} \tag{2.61}$$

Bedeutung für die DFT

Mit dem Funktional aus Gl. (2.61) lassen sich analog zu Gl. (2.33) Korrelationsfunktionen generieren. Für eine gleichmäßige Flüssigkeit mit ortsunabhängigen gewichteten Dichten erhalten wir

$$c_{ij}^{(2)}(\mathbf{r}) = -\frac{\partial^2 \Phi}{\partial n_\alpha \partial n_\beta} \omega_i^{(\alpha)} \otimes \omega_j^{(\beta)}. \tag{2.62}$$

Diese direkten Korrelationsfunktionen sind identisch mit denen der Percus-Yevick-Theorie. Rosenfelds rein geometrischer Ansatz führt folglich sowohl zu einer bereits bekannten und sehr guten Näherung der Freien Energiedichte als auch zu den Paarkorrelationsfunktionen einer Flüssigkeit harter Kugeln. Dies war ein großer Schritt auf dem Gebiet der DFT, da insbesondere inhomogene Systeme ein Problem darstellten.

An dieser Stelle sei angemerkt, dass die Faktorisierung des Mayer f-Bonds nicht einzigartig ist. In [Kie90] demonstrierten Kierlik und Rosinberg, dass eine Faktorisierung mit Hilfe von vier skalaren Gewichtsfunktionen ebenfalls möglich ist. Es konnte gezeigt werden, dass beide Ansätze die gleichen Ergebnisse liefern.

Die Ergebnisse der FMT halten i.d.R. dem Vergleich mit Simulationsdaten stand. Die FMT reproduziert kurzreichweitige Korrelationen und Effekte, die bei unterschiedlichen Kugelradien auftreten. Allerdings ist sie nicht in der Lage, den Gefrierübergang für dreidimensionale harte Kugeln zu reproduzieren.

Verallgemeinerung auf niedrigere Dimensionen

Um das FMT-Funktional zu verbessern und auch den Gefrierübergang abbilden zu können nutzte Rosenfeld das Konzept des "Dimensional Crossover", die Einschränkung des 3D-Funktionals auf niedrigere Dimensionen. Die Idee dahinter ist simpel aber von hoher Tragweite: Das (unbekannte) exakte 3D-Funktional muss, wenn man es auf 2 Dimensionen einschränkt, d.h. $\rho_i(\mathbf{r}) = \delta(z)\rho_i^{(2D)}$, in das exakte (ebenfalls unbekannte) 2D-Funktional übergehen. Die mit diesem Ansatz erhaltene Freie Energiedichte, die mit Hilfe von Gl. (2.56) berechnet wird, konnte daraufhin mit bekannten Resultaten verglichen werden. Wie im dreidimensionalen Fall ergibt sich die Freie Exzessenergie über Gl. (2.61). [Kie91] und [Ros93] zeigen, dass die so gewonnenen Ergebnisse numerisch in gutem Einklang mit den bekannten Daten stehen. Da das exakte 2D-Funktional allerdings auch unbekannt ist reicht diese Reduzierung der Dimension noch nicht aus, um Hinweise für eine mögliche Verbesserung des 3D-Funktionals zu erhalten.

Der Übergang in eine Dimension, $\rho_i(\mathbf{r}) = \delta(z)\delta(y)\rho_i^{(1D)}$, ist ebenfalls möglich. In einer Dimension ist das exakte Freie Energiefunktional bekannt. Mit Hilfe von Gl. (2.56) können die gewichteten Dichten genau wie im zwei- und dreidimensionalen Fall berechnet werden. Führt man jedoch das Integral in Gl. (2.61) aus so sieht man, dass bereits die ersten beiden Terme die exakte eindimensionale Freie Exzessenergie produzieren und der dritte Term divergiert. Im Umkehrschluss bedeutet dies, dass der dritte Term im 3D-Funktional modifiziert werden muss.

Um dieses Problem zu lösen, betrachtet man zunächst den nulldimensionalen Grenzfall, d.h. eine Aussparung die nur eine einzige Kugel aufnehmen kann und in der die Kugel keine räumlichen Freiheitsgerade hat. Da die Freie Energie einer solchen Aussparung bekannt ist ist es möglich, numerisch einen dritten Term in Gl. (2.61) zu generieren, der sie korrekt beschreibt. In [Ros97] wurde gezeigt, dass

$$\Phi_3^{(0D)} = \frac{2n_2^3\xi(1-\xi)^2}{24\pi(1-n_3)^2} \tag{2.63}$$

mit $\xi(\mathbf{r}) = |\frac{\mathbf{n}_2(\mathbf{r})}{n_2(\mathbf{r})}|$ diese Bedigung erfüllt. Diese rein empirische Modifikation beschreibt zwar den 0D-Grenzfall richtig, sorgt allerdings für Fehler in der dreidimensionalen Flüssigkeit. Allerdings brachte die Betrachtung von $\Phi_3^{(0D)}$ die Erkenntnis, dass eine Antisymmetrisierung bezüglich der $\mathbf{r}$-Integration von Φ_3 die Divergenz

im 0D-Grenzfall beseitigt und so generiert werden kann, dass das dreidimensionale Funktional nicht signifikant geändert wird. Rosenfeld schlug in [Ros97]

$$\Phi_3^{asym} = \frac{n_2^3(1-\xi)^q}{24\pi(1-n_3)^2} \tag{2.64}$$

mit $q \geq 2$ vor. Dies ist allerdings immernoch eine rein empirische Modifikation der ursprünglichen FMT. Tarazona zeigte in [Tar00], dass die Einführung einer zusätzlichen gewichteten Tensordichte systematisch genutzt werden kann, um die Divergenz im 0D-Grenzfall zu beseitigen und gleichzeitig die dreidimensionale Flüssigkeit korrekt zu beschreiben. Dieses auf einer Tensordichte basierende Tensorfunktional wird in Kapitel 6 im Detail erläutert und bildet die Basis für wesentliche Ergebnisse dieser Arbeit.

2.2. Moderne Anwendungen

Die Einsatzgebiete der DFT sind vielfältig, berechnet werden können beispielsweise:

- Struktur von Flüssigkeiten [Eva79] [Sch00] [Han06]
- Struktur von Flüssigkeitsmischungen [Sch00] [Han06]
- Phasengleichgewicht von Flüssigkeiten und Mischungen [Sch02]
- Selbstorganisierte Systeme, Oberflächenstrukturen und konkurrierende Phasen [Eva79] [Tar84]
- Zwei-Phasen-Oberflächen [Cur87] [Arc03]
- Struktur von Flüssigkeiten in der Nähe von Oberflächen [Tar84]
- Flüssigkeitsdiffusion [Mat06]
- Phasenübergänge an Flüssigkeitsfilmen [Med87]
- Eigenschaften von atomaren, asymmetrischen und gebundenen Flüssigkeiten [Bal88] [Ros96]
- Geladene Systeme [Eva80]
- Oberflächen und Flüssigkeiten, die sich aus beliebigen einfachen geometrischen Anordnungen zusammensetzen [Eva80] [Ros96] [Han06]

Diese Liste ist nicht vollständig, gibt allerdings einen guten ersten Eindruck von dem breiten Spektrum, in dem die klassische DFT eingesetzt werden kann.

Durch dieses breite Spektrum an Anwendungsmöglichkeiten erfreut sich die klassische DFT heute sehr großer Beliebtheit. In der Regel ist es möglich, diese Fragestellungen mit hoher Effizienz, d.h. deutlich schneller als mit Monte-Carlo- oder Molekulardynamik-Simulationen, zu lösen. Gerade deshalb ist es interessant zu untersuchen, inwiefern die Geschwindigkeit der Berechnung der DFT durch den Einsatz von Grafikkarten erhöht werden kann.

3. General Purpose GPU Programming

3.1. Übersicht über moderne Grafikkarten

Im folgenden Kapitel wird eine Übersicht über die Entwicklung der Grafikprozessoren, ihre Funktionsweise und ihre Nutzung zur Berechnung allgemeiner Probleme gegeben.

Geschichte der GPU

Die Entwicklung der GPUs wurde hauptsächlich durch die Nutzung moderner PCs als Spielesysteme motiviert. Spiele, in denen alle drei Raumdimensionen zugänglich sind, erfreuten sich seit dem Ende der 90er Jahre einer steigenden Beliebtheit. Interaktive Objekte in drei Raumdimensionen zu berechnen und dann in Echtzeit auf einem zweidimensionalen Bildschirm darzustellen erfordert allerdings sehr viel Rechenleistung. Insbesondere da Spielehersteller Kunden mit einer immer beeindruckerenden Optik gewinnen wollten reichten die üblichen CPUs schnell nicht mehr aus, um den steigenden Bedarf an Rechenleistung zu befriedigen. Eine 3D-Computerdarstellung wird in der Regel dadurch erziehlt, dass man räumliche Objekte aus Polygonen, die wiederum aus einzelnen Dreiecken bestehen, zusammensetzt. Hat man einen Raum aus Polygonen aufgebaut so muss ihr Aussehen vom momentanen Blickwinkel des Spielers berechnet werden, d.h. man bestimmt Objekteigenschaften wie Beleuchtung und muss eine geeignete Projektion auf den 2D-Monitor finden. Dies geschieht in der Grafikpipeline (rendering pipeline), in der die nötigen Transformationen in Matrixform dargestellt und mit den Polygonkoordinaten verrechnet werden. Mit steigender Bildqualität wurden sog. Vertex- und Pixelshader

etabliert, die zusätzliche komplexe Transformationen auf Vertex- oder Pixelbasis durchführen können. Anders als die CPU, die allgemeine Rechnungen durchführen muss, mussten die ersten GPUs deshalb nur möglichst schnell möglich viele Matrixmultiplikationen lösen können, was ein hochoptimiertes Design ermöglicht hat. Trotz der anfangs geringen Komplexität der GPUs erreichten sie durch diese Spezialisierung sehr hohe Rechengeschwindigkeiten und etablierten sich schnell zum Standard zur Darstellung von Computerspielen. Die erste weit verbreitete Karte dieser Art, die 3dfx Voodoo, ist in Abb. 3.1 zu sehen. Sie wurde zusätzlich zu einer normalen 2D-Grafikkarte genutzt und stellte spezielle Rechenoperationen für die dreidimensionale Darstellung zur Verfügung. Erst später wurden 2D- und 3D-Funktionen auf einer Karte vereint.

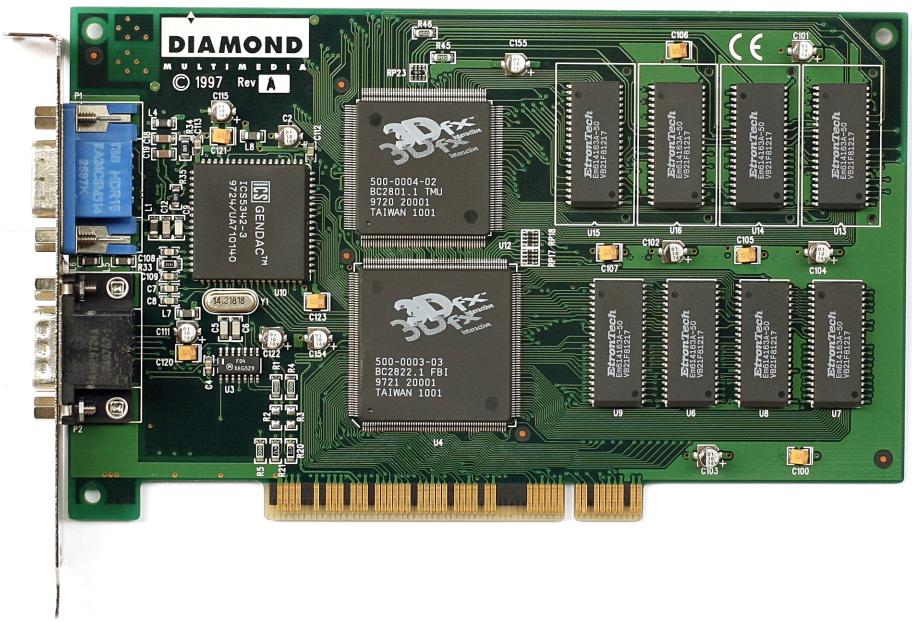

Abbildung 3.1.: Einer der ersten 3D-Beschleuniger, eine 3dfx Voodoo. Diese PCI-Karten wurden 1996 verkauft und zusätzlich zu einer herkömmlichen Grafikkarte genutzt. Spiele konnten dann die 3D-Berechnungen auf diese Karte auslagern. Quelle: Wikipedia

Der mit Computerspielen erzielte Umsatz lag in 2011 in Deutschland bei 1.6 Milliarden Euro [Sta12] und ist somit vergleichbar mit dem Umsatz der Filmindustrie in Deutschland von 2.6 Milliarden Euro in 2009 [Sta12] und dem der Musikindustrie von 1.5 Milliarden Euro in 2010 [Sta12]. Angetrieben durch diesen riesigen Markt wurden die GPUs zu den am schnellsten weiterentwickelten Prozessoren der Welt.

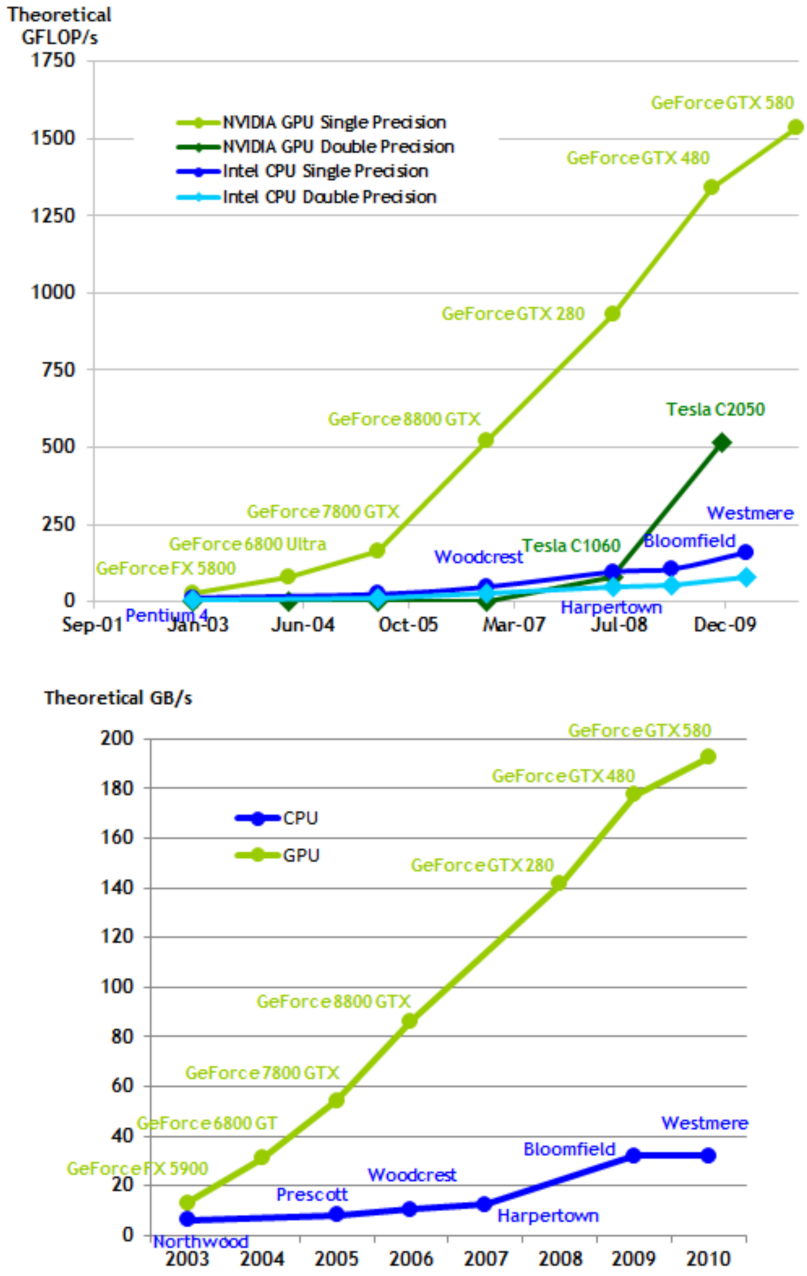

Abbildung 3.2.: Entwicklung der Rechenleistung sowie der Speicherbandbreite der NVidia GPU und Intel CPUs. Durch starke Parallelisierung rechnen Grafikkarten deutlich schneller als CPUs. Quelle: NVidia C Programming Guide

Grafikberechnungen lassen sich sehr leicht paralellisieren, was den GPUs einen weiteren Vorteil gegenüber den CPUs bot: Sie konnten von Anfang an auf diese Parallelisierung zugeschnitten werden. In nur wenigen Jahren überstieg die Komplexität einer modernen GPU die der CPUs und GPUs boten theoretisch eine Maximierung der Rechenleistung bei sehr günstigem Preis. Allerdings nur theoretisch, da sie streng auf die Grafikberechnung ausgelegt waren und Berechnungen über speziell zugeschnittene Sprachen wie OpenGL oder DirectX erfolgten. Mit einigen Tricks war es zwar möglich, gewisse allgemeine Berechnungen so umzuformulieren, dass sie von der Grafikkarte als 3D-Bilddarstellung interpretiert und mit Hilfe von OpenGL berechnet werden konnten [Ake08], diese Methoden waren allerdings sehr umständlich und in den Möglichkeiten stark beschränkt, weshalb sie keine weite Anwendung fanden. Die Shadertechnologie ist jedoch beständig weiter gewachsen und wurde immer ausgereifter, so dass diese Shaderprozessoren die höchste Rechenleistung in einem PC aufwiesen, es fehlte lediglich eine Programmiersprache, um sie allgemein ansprechen zu können. Abb. 3.2 illustriert die Steigerung der Geschwindigkeit der Grafikkarten im Vergleich zur CPU. Dieses ungenutzte Potential erkannten auch die GPU-Hersteller, was zur Entwicklung der „General Purpose Computation on Graphics Processing Unit“ (GPGPU) führte. Mit einer speziellen Programmiersprache (CUDA für NVidia, Brook+ für ATI und später OpenCL plattformübergreifend) wurde es möglich, nahezu beliebigen Code auf der GPU auszuführen. Es gibt allerdings einige Einschränkungen, die auf die Architektur der GPU zurückzuführen sind.

Funktionsweise und Programmierung einer modernen GPU

Jede GPU ist im Wesentlichen ein kompakter Cluster. Sie besteht aus mehreren Hundert Rechenkernen, die jeweils einzelne Berechnungen durchführen können. Diese Kerne werden in Multiprozessoren gegliedert, die auf gemeinsame schnelle Speicherbänke zugreifen können, wie in Abb. 3.3 zu sehen ist.

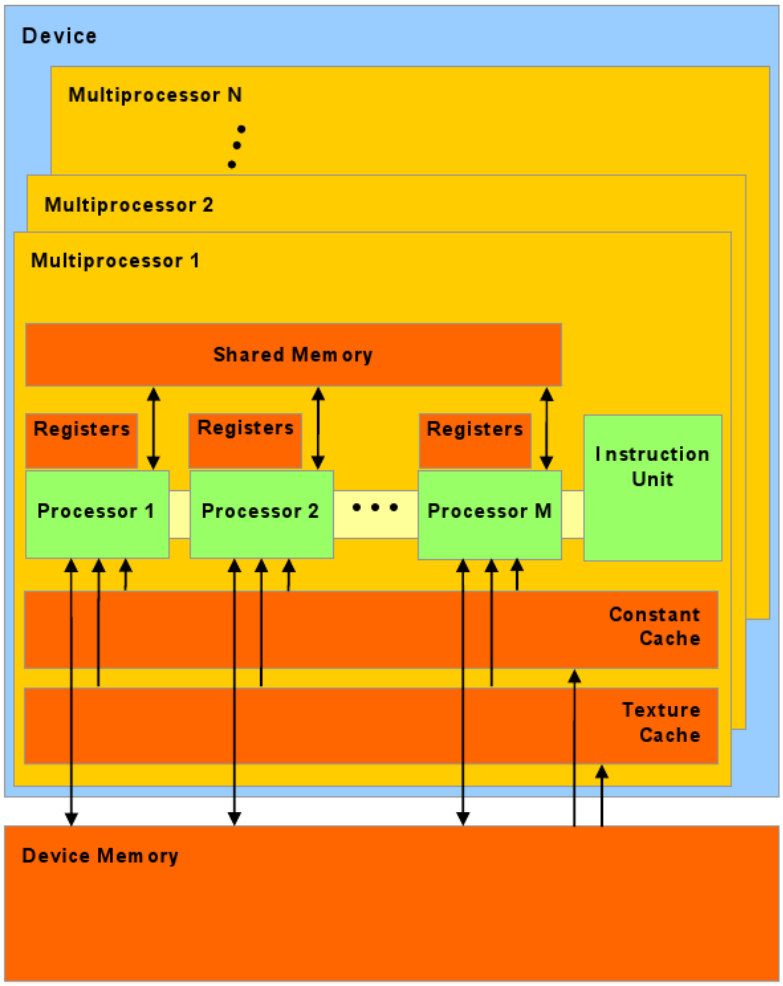

Abbildung 3.3.: Aufbau einer modernen GPU. Quelle: NVidia C Programming Guide

Gesteuert werden diese Cores mit Hilfe einer eigenen Programmiersprache. NVidia veröffentlichte CUDA, die Sprache, die zur Gewinnung der Resultate dieser Arbeit genutzt wurde, im Jahr 2006. CUDA basiert auf C und erlaubt es, mit sehr C-

ähnlichem Code parallelisierte Programme auf einer NVidia GPU auszuführen. Das Gegenstück von AMD/ATI namens Brook+ erfüllt prinzipiell die gleiche Aufgabe, wurde allerdings im Rahmen dieser Arbeit nicht eingesetzt. Mit dem Anstieg der Beliebtheit der GPGPU wurde 2008 ein neuer Standard, OpenCL, geschaffen, mit dessen Hilfe man herstellerunabhängig Grafikkarten ansprechen kann. Da die OpenCL-Spezifikationen noch sehr neu und ungetestet ist wurde in dieser Arbeit darauf verzichtet, sie zu benutzen.

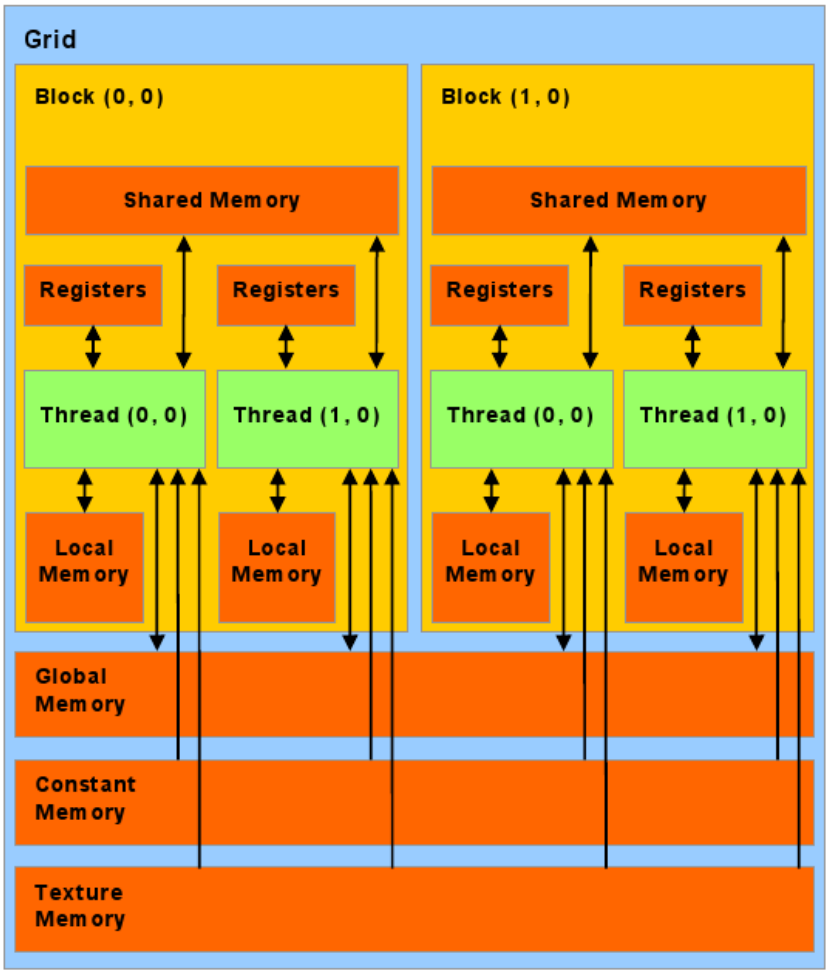

Abbildung 3.4.: Speicherstruktur in CUDA und Zugriffsbereiche der einzelnen Threads und Blocks auf die jeweiligen Speicherbereiche. Quelle: NVidia C Programming Guide

Die große Herausforderung für all diese Programmiersprachen ist es, die Skalier-

barkeit des Codes auf eine Vielzahl von GPUs sicherzustellen, die jeweils mit einer hohen aber unterschiedlichen Anzahl von Cores ausgerüstet sind. Die zentrale Aufgabe von CUDA ist es deshalb, eine parallele Ausführung des Programmcodes zu ermöglichen, die problemlos mit der Anzahl der Cores skaliert. Zu diesem Zweck wurde C um Syntax erweitert, der einfaches Multirhreading ermöglicht. Das Codebeispiel 3.1 zeigt diesen Syntax.

Um dies zu erreichen wird die Parallelisierung als ein Gitter (Grid) betrachtet. Dieses Grid besteht aus einer Anzahl von Blöcken (Blocks) B, die jeweils T Prozesse (Threads) enthalten. Die Funktion wird dann für jeden Block T mal aufgerufen, so dass sie insgesamt $B \cdot T$ mal ausgeführt wird. Abb. 3.4 illustriert diese Aufteilung. In der Funktionsdefinition kann auf die Block- und Threadzahl des jeweiligen Aufrufs zugegriffen werden, so dass jeder Aufruf auf unterschiedliche Daten zugreifen kann („SIMD“, single instruction, multiple data parallel computing).

```
__global__ void example_function(float variable)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;

    //code

}

int main(int argc, char** argv)
{
    example_function<<<dimGrid,dimBlock>>>(float_variable);
}
```

Listing 3.1: Beispiel einer CUDA-Funktion und eines CUDA-Funktionsaufrufs. blockIdx und threadIdx sind der Index der des jeweiligen Blocks und Threads, blockDim ist die Blockgröße.

Die Unterteilung in ein Grid aus Blocks und Threads ist nötig, um die gute Skalierbarkeit der Sprache sicherzustellen. Abb. 3.5 zeigt, wie ein CUDA-Programm auf Grafikkarten mit unterschiedlicher Anzahl an Cores ausgeführt werden kann. Ein Block wird auf einem Multiprozessor abgearbeitet. Die Blockgröße ist i.d.R.

höher als die Anzahl der Cores im Multiprozessor, so dass die im Block enthaltenen Threads dann sequenziell auf den einzelnen Cores abgearbeitet werden, was durch die Farben in Abb. 3.3 und Abb. 3.4 veranschaulicht wird. Die Aufteilung des Grids auf die Multiprozessoren erfolgt ähnlich: Jeder Multiprozessor bearbeitet ein Grid. Gibt es mehr Blocks als Multiprozessoren zur Verfügung stehen so werden die Blocks sequenziell bearbeitet.

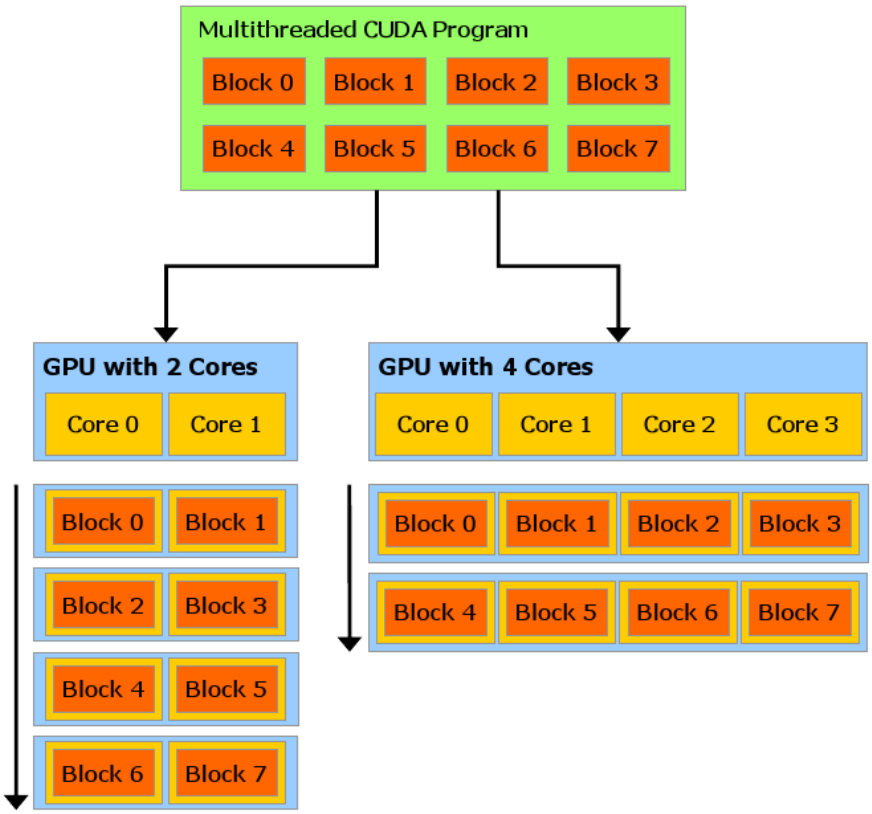

Abbildung 3.5.: Veranschaulichung der Ausführung eines CUDA-Programms auf Grafikkarten mit einer unterschiedlichen Anzahl von Multiprozessoren (in der Grafik „Core“ genannt). Die Blocks und Threads werden auf die verfügbaren Cores verteilt. Quelle: NVidia C Programming Guide

Da sich alle Threads den Speicher eines Multiprozessors teilen, gibt es ein Limit für die maximale Threadzahl pro Block. Für die hier meist genutzte NVidia Geforce GTX 280 ist dieses Limit 512, neueste Karten können bis zu 1024 Threads pro Block verwalten. Die Blockzahl dagegen wird von den zu bearbeitenden Daten (der Zahl

der benötigten Funktionsaufrufe) vorgegeben und ist lediglich durch den insgesamt auf der Grafikakrte verfügbaren Speicher beschränkt.

Stärken, Schwächen und Anwendungsbeispiele

Die größte Stärke des GPGPU ist auch gleichzeitig seine größte Schwäche: Die strikte Auslegung auf parallele Berechnungen. Probleme, die sich nicht effizient parallelisieren lassen lassen sich ebenfalls nicht effizient auf der GPU berechnen. Ein weiteres Problem bei Berechnungen auf der Grafikkarte ist der Grafikspeicher. Spielekarten weisen i.d.R. lediglich 1GB bis 2GB auf, professionelle, auf GPGPU ausgelegte (und damit auch teurere) Karten liefern 4GB bis 6GB. Für sehr speicherintensive Berechnungen ist jedoch auch dies noch zu wenig. Ist das Problem separierbar so kann man Teile der Daten auf den RAM auslagern, was lediglich Zeit kostet. Wird jedoch ein großer geschlossener Datensatz für die Berechnung benötigt, so eignet sich die GPU nicht dafür. Der verfügbare Speicher ist die wesentliche Beschränkung für die hier genutzte DFT. Dreidimensionale Gitter in hoher Auflösung lassen sich nicht sinnvoll im Grafikspeicher unterbringen. Die zweidimensionale Gitterauflösung ist auf maximal 4096^2 begrenzt, was für die hier durchgeführten Berechnungen ausreichend ist.

Eine weitere Beschränkung finden wir in der Rechengenauigkeit. Moderne GPUs unterstützen erst seit kurzem doppelte Rechengenauigkeit (double precision, DP), jedoch nicht in voller Geschwindigkeit. Günstige NVidia-Karten (Spiele-Karten) liefern nur 12,5% ihrer Leistung in DP, NVidia Tesla und Karten von AMD verlieren 50% ihrer Leistung bei Rechnungen in DP. Will man alle Multiprozessoren mit maximaler effizienz nutzen, so kann man nur in einfacher Genauigkeit (single precision, SP) rechnen. Desweiteren verfügen die Grafikkarten i.d.R. im Gegensatz zur CPU nicht über eine Fehlerkorrekter. Da sie ursprünglich für Computerspiele entwickelt wurden war es nicht besonders wichtig, wenn eine Berechnung einen falschen Wert lieferte, da sie für das nächste Bild ja wiederholt wird und ein so kleiner Bildfehler dem Betrachter nicht auffällt. Testläufe haben allerdings gezeigt, dass die GPUs auch ohne Fehlerkorrektur ausreichend zuverlässig rechnen, so dass dies in der Praxis keine Beschränkung darstellt. Desweiteren wurde die neuste Generation von professionellen Grafikkarten mit einer Fehlerkorrektur ausgestattet, so dass

diese Möglichkeit, wenn auch zu einem deutlichen Aufpreis im Vergleich zu den Spielekarten, zur Verfügung steht.

Die Anwendungsgebiete der GPGPU sind vielfältig. Es gibt zahlreiche physikalische Probleme, die sich sehr gut parallelisiert lösen lassen. Ein bekanntes Beispiel hierfür is das Ising-Modell [Pre09]. Generell bietet sich GPGPU für alle Probleme an, die sich auf einem Gitter diskretisieren lassen. Die „Fast Fourier Transformation“ (FFT) kann ebenfalls schnell auf der GPU berechnet werden. Physikalische Berechnungen, die über Fourier-Transformationen lösbar sind, profitieren damit stark von der GPGPU. Dies ist für die DFT der Fall und die FFT wird in dieser Arbeit intensiv genutzt. Programme wie MATLAB können die GPU für Berechnungen nutzen und es gibt bereits GPU-basierte MD-Simulationen wie beispielsweise HOOMD blue. Auch in der Informatik wird die GPGPU eingesetzt. Parallelisierbare Algorithmen wie der „k-nearest neighbor algorithm“ lassen sich effizient implementieren [Gar08].

Sowohl in der Bild- als auch in der Audiobearbeitung spielt die GPU mittlerweile eine beachtliche Rolle. Die Unterdrückung ungewünschter Geräusche (z.B. ein übersteuerndes Mikrofon) kann dank schneller GPUs problemlos in Echtzeit berechnet werden. Die Berechnung von 3D-Audio stellt ähnliche Anforderungen wie die Berechnung von 3D-Szenen in Computerspielen, so dass die Grafikkarten auch hierfür sehr gut geeignet sind. In der Bildbearbeitung werden oft komplexe Filter eingesetzt, um die Bildqualität zu verbessern. Auch diese Operationen lassen sich meist schnell auf der Grafikkarte berechnen [Obe11].

Ebenfalls sehr erfolgreich ist die Nutzung der Grafikkarte zur Berechnung von Daten auf den Finanzmärkten. High frequency trading (HFT) basiert auf der Idee, Posten schnell zu kaufen und zu verkaufen und minimale Kursschwankungen für einen Gewinn auszunutzen. Dies geschieht i.d.R. über eine Software, die die Kursverläufe analysiert. Je schneller diese Software arbeiten kann, desto eher lassen sich Gewinne realisieren. Die hohe Rechenleistung der GPU bietet hier offensichtliche Vorteile [Pvw09].

Es gibt jedoch noch eine weitere, weniger bekannte Anwendung der Grafikkarten in der Finanzwelt: Die digitale Währung „Bitcoins“ basiert auf SHA-256 hashes, d.h. einer besonderen Form von i.d.R. computergenerierten Abbildungen, die einer sehr großen Quellmenge eine Ausgabe aus einer kleineren Zielmenge, den sog. Hashcodes, zuordnet. Diese Hashcodes sind zur Signierung einer Bitcoin-Transaktion nötig, was

die Sicherheit der Währung gewährleistet. Die Berechnung dieser Werte ist sehr effizient auf der GPU möglich, so dass die gesamte Währung mittlerweile stark von der GPGPU abhängig ist. Die große Mehrheit aller Hashcodes wird auf GPU-Clustern berechnet [Bit09].

Auch für den umgekehrten Vorgang in der Kryptographie, entziffern von Passwörtern, eignen sich Grafikkarten sehr gut. Durch ihre parallele Struktur können sie sog. „brute force“-Algorithmen effizient umsetzen, so dass Passwörter, die noch vor wenigen Jahren als ausreichend sicher galten mittlerweile in wenigen Tagen und ohne großen finanziellen Aufwand herausgefunden werden können [Hu09].

Die Nutzung von Grafikkarten ist in vielen Bereichen auf dem Vormarsch. Als effiziente und kostengünstige „Clustercomputer unter dem Schreibtisch“ eröffnen sie viele neue Möglichkeiten. Die DFT erfüllt alle Voraussetzungen, um stark von der Nutzung der GPGPU zu profitieren, weshalb diese neue Technologie hier eingesetzt wurde.

4. Performanceanalyse

4.1. Implementierung eines simples Funktionals

4.1.1. Das Taylor-Funktional zweiter Ordnung

In diesem Kapitel wird die Implementierung eines einfachen Funktionals sowohl auf der CPU als auch auf der GPU beschrieben. Die wesentlichen Eigenschaften des Funktionals werden herausgestellt und der Geschwindigkeitsgewinn durch Nutzung der Grafikkarte wird bestimmt.

Herleitung des Funktionals

Wir starten mit der Separation des allgemeinen Energiefunktionals in den Anteil des idealen Gases (id) und den Rest, der als Exzess-Anteil (ex) bezeichnet wird, analog zu Kapitel 2:

$$F[\rho] = F^{id}[\rho] + F^{ex}[\rho]. \tag{4.1}$$

Das Funktional für das ideale Gas ist bekannt,

$$\beta F^{id} = \int d\mathbf{r}\ \rho(\mathbf{r}) \cdot \left(\ln \lambda^3 \rho(\mathbf{r}) - 1\right), \tag{4.2}$$

der Exzess-Anteil ist bei niedrigen Dichten durch

$$\beta F^{ex} = -\frac{1}{2} \int dr \int dr' \rho_i(\mathbf{r}) f_{ij}(\mathbf{r} - \mathbf{r'}) \rho_j(\mathbf{r'}) \tag{4.3}$$

gegeben, wobei f den Mayer f-Bond darstellt, welches für harte Scheiben mit Radien R_i und R_j im Abstand r die Form

$$f_{ij}(\mathbf{r}) = -\Theta(R_i + R_j - r) \tag{4.4}$$

annimmt. Betrachten wir die funktionale Taylor-Entwicklung von Gl. (4.3) in zweiter Ordnung für eine einkomponentige Flüssigkeit so erhalten wir, wie in Kapitel 2.1.1 gezeigt wurde,

$$\beta F^{ex} = -\frac{1}{2}\int d\mathbf{r}\; d\mathbf{r'}\; c^{(2)}(\mathbf{r}-\mathbf{r'};\rho_0)\;\Delta\rho(\mathbf{r})\;\Delta\rho(\mathbf{r'}) + \beta\mu^{ex}\int d\mathbf{r}\;\Delta\rho(\mathbf{r}) + \beta F_0(\rho_0), \tag{4.5}$$

$c^{(2)}$ ist die aus der Ornstein-Zernike-Relation bekannte direkte Korrelationsfunktion und $\Delta\rho(\mathbf{r}) = \rho(\mathbf{r}) - \rho_0$.

Differenzieren wir nun das Großkanonische Potential Ω nach ρ und nutzen das Maximalprinzip so erhalten wir eine Gleichung für ρ selbst:

$$\begin{aligned}\Omega = &\int d\mathbf{r}\;\rho(\mathbf{r})(\ln\lambda^3\rho(\mathbf{r}) - 1) + F_0[\rho_0] + \mu^{ex}\int d\mathbf{r}\;\Delta\rho(\mathbf{r}) \\ &- \frac{1}{2}\beta\int d\mathbf{r}\;c^{(2)}(\mathbf{r}-\mathbf{r'};\rho_0)\Delta\rho(\mathbf{r})\Delta\rho(\mathbf{r}) + \int d\mathbf{r}\;(-\mu+V)\rho(\mathbf{r})\end{aligned} \tag{4.6}$$

$$0 = \frac{\delta}{\delta\rho}\Omega = \beta^{-1}\ln\lambda^3\rho(\mathbf{r}) + \underbrace{\mu^{ex} - \mu}_{-\beta^{-1}\ln(\lambda^3\rho_0)} + V(\mathbf{r}) - \beta^{-1}\int d\mathbf{r}\;c^{(2)}(\mathbf{r}-\mathbf{r'};\rho_0)\Delta\rho(\mathbf{r}) \tag{4.7}$$

$$\iff \qquad \ln(\rho(\mathbf{r})) - \ln(\rho_0) = -\beta V(\mathbf{r}) + c^{(2)} * \Delta\rho \tag{4.8}$$

$$\iff \qquad \rho(\mathbf{r}) = \rho_0 \cdot \exp\left(-\beta V(\mathbf{r}) + c^{(2)} * \Delta\rho\right). \tag{4.9}$$

Dabei ist $V(\mathbf{r})$ das Potential der harten Scheiben. Diese Gleichung lässt sich durch Interation bis zur Selbstkonsistenz auf einem Gitter lösen.

Vorteile und Beschränkungen

Der Vorteil dieses Ansatzes liegt vor allem in seiner Einfachheit. Die erhaltene Gleichung enthält eine Faltung, die im Fourierraum ausgewertet werden kann. Da

der FFT-Algorithmus leicht auf Grafikkarten optimiert werden kann, ist eine einfache Umsetzung der Gleichung auf der GPU mit hohem Geschwindigkeitsgewinn gegenüber der CPU zu erwarten.

Auf einem zweidimensionalen Gitter muss $\rho_{i+1}(\mathbf{r}) = \rho_0 e^{-\beta V(\mathbf{r}) + c^{(2)} * \Delta\rho_i}$ iteriert werden. Dabei ist $V(\mathbf{r})$ ein konstantes Potential, welches unendlich groß an der Position des Teilchens und 0 überall sonst ist. Da $c^{(2)}$ konstant ist, muss lediglich die Faltung mit $\Delta\rho$ in einem Rechenschritt ermittelt werden um die Iteration durchführen zu können.

Der Nachteil dieses Ansatzes ist, dass er eine schlechte Näherung darstellt, d.h. dass er nur ungenaue Resultate liefert. Die Beschränkung auf die direkte Korrelationsfunktion, d.h. der Abbruch der Taylor-Reihe nach dem ersten Glied, ist mathematisch nicht zu rechtfertigen. Für geringe Dichten ist es eine gute Näherung aber wir wissen, dass der vernachlässigte Anteil für hohe Dichten nicht konvergiert, da dort ein Phasenübergang stattfindet. Das Taylor-Funktional kann deshalb nur als „proof of concept“ angesehen werden. Dank seiner Einfachheit ist es ideal dafür, Performancevergleiche zwischen GPU und CPU durchzuführen. Die wesentlichen Rechenschritte, nämlich die Bestimmung der Faltung und die Auswertung auf dem Gitter, werden auch bei den später betrachteten Funktionalen gleich bleiben. Komplexere Funktionale werden lediglich die Anzahl der benötigten Faltungen und Rechenoperationen erhöhen, nicht aber die fundamentale Methode. Deshalb ist es gerechtfertigt, dieses einfache Funktional zur Geschwindigkeitsanalyse heranzuziehen, die Ergebnisse sind auf komplexere Funktionale übertragbar.

4.1.2. Die Fast-Fourier-Transformation

Herleitung der Fast Fourier Transformation

Nachdem wir festgestellt haben, dass die Bestimmung der Faltung und somit die Berechnung der Fourier-Transformationen den wesentlichen Teil der Rechenzeit beanspruchen werden lohnt es sich, einen Blick auf den dafür genutzten Algorithmus zu werfen, welcher in [Fri05], [Fri98], [Fri99], [Bur12], [Lei99] und [Joh07] ausführlich beschrieben wird. Die Grundidee der schnellen Fourier-Transformation ("Fast Fourier Transformation", FFT) ist es, die Auswertung der Fourier-Transformation (FT) der Größe n,

$$X[k] = \sum_{l=0}^{n-1} x_l e^{-i2\pi k \frac{l}{n}}, \tag{4.10}$$

welche für jedes der N Punkte X_k die Summe über n Elemente auswerten muss, also in $\mathrm{O}(n^2)$ läuft, in zwei Transformationen der Größe $n/2$ aufzuteilen. Dafür nutzen wir, dass

$$\begin{aligned} X[k] &= \sum_{l=0}^{n-1} x_l e^{-i2\pi k \frac{l}{n}} \\ &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-i2\pi k \frac{2m}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-i2\pi k \frac{2m+1}{N}} \\ &= \sum_{m=0}^{M-1} x_{2m} e^{-i2\pi k \frac{m}{M}} + e^{-\frac{i2\pi}{N}k} \sum_{m=0}^{M-1} x_{2m+1} e^{-i2\pi k \frac{m}{M}} \\ &= \begin{cases} E[k] + e^{-\frac{i2\pi}{N}k} O[k] & k < M, \\ E[k-M] - e^{-\frac{i2\pi}{N}(k-M)} O[k-M] & k \geq M. \end{cases} \end{aligned} \tag{4.11}$$

mit $E[k+M] = E[k]$ gerader und $O[k+M] = O[k]$ ungerader FT gilt. Die Fourier-Transformationen E und O können nun wiederum mit dieser Methode in ihren geraden und ungeraden Anteil zerlegt werden. Falls die Gittergröße eine Zweierpotenz ist kann diese Rekursion forgesetzt werden, bis man eine triviale einelementige FT erreicht. Durch die Nutzung dieses binären Baums lässt sich die Transformation ei-

nes Punktes in $O(\log n)$ berechnen, d.h. die Transformation aller Punkte läuft in $O(n \log n)$. Dies nennt man den "Radix 2 Cooley-Tukey-Algorithmus" [Coo65], da es auf der Separation in zwei Teile beruht. Dieser Algorithmus stellt eine der effizientesten Berechnungsmethoden für Fourier-Transformationen dar, weshalb in dieser Arbeit nur Gitter in Größe einer Zweierpotenz genutzt wurden.

Allgemein lässt sich eine FT mit n Punkten $\mathbf{Y}[l]$ ($l = \{0, \ldots, n-1\}$) und $\omega_n = e^{-i2\pi k\frac{1}{n}}$

$$X[k] = \sum_{l=0}^{n-1} Y[l]\omega_k^l \tag{4.12}$$

zerlegen, wenn n in $n = n_1 n_2$ faktorisiert. Setzt man $l = l_1 n_2 + l_2$ und $k = k_1 + k_2 n_1$ erhalten wir

$$X[k_1 + k_2 n_1] = \sum_{l_2=0}^{n_2-1} \left(\left(\sum_{l_1=0}^{n_1-1} \mathbf{Y}[l_1 n_2 + l_2] \omega_{n_1}^{l_1 k_1} \right) \omega_n^{l_2 k_1} \right) \omega_{n_2}^{l_2 k_2}. \tag{4.13}$$

Dies ermöglicht eine Zerlegung der Transformation in n_2 Teile der Größe n_1 (innere Summe), welche mit den sog. Twiddle-Faktoren $\omega_n^{l_2 k_1}$ multipliziert werden. Danach können die verbleibenden n_1 Transformationen der Größe n_2 (äußere Summe) ausgeführt werden. Damit lassen sich auch Fourier-Transformationen, bei denen die Gittergröße keine Zweierpotenz ist, in $O(n \log n)$ berechnen. Aus dieser allgemeinen Separation erhält man den Radix 2 Algorithmus für $n_2 = 2$.

Da die Bestandteile dieses binären Baums einzeln berechnet werden können liegt es auf der Hand, dass sich die FFT sehr gut parallelisieren lässt und somit stark von der Berechnung auf der Grafikkarte profitieren wird. Insbesondere die bei einer mehrdimensionalen FT auftretenden äußeren Schleifen lassen sich sehr gut auf die Multiprozessoren verteilen.

Der Cooley-Tukey-Algorithmus ist effizient für Transformationen mit vielen Elementen, solange sich die Elementzahl zerlegen lässt. Für Transformationen weniger Elemente ist die $O(n \log n)$-Skalierung weniger relevant und es gibt andere, effizientere Berechnungsmethoden wie beispielsweise den in [Opp99] beschriebenen Primfaktoralgorithmus. Diese können eingesetzt werden, um die Berechnung der letzten (kleinen) Transformationen im Cooley-Tukey-Algorithmus zu beschleunigen. Transformationen bei großen Gittern und einer Elementzahl, die sich nicht in kleine Teile

zerlegen lässt (d.h. Primzahlen oder Produkte großer Primzahlen) sind ebenfalls mit Hilfe anderer Algorithmen effizienter als in $O(n^2)$ berechenbar. Da die hier verwendeten Elementzahlen jedoch alle eine Zweierpotenz sind, wurde dies nicht genutzt.

4.1.3. CPU-Implementierung

Die CPU-Referenz

Um die Berechnung des Funktionals auf GPU und CPU vergleichen zu können wurde ein rein CPU-basiertes Programm entwickelt, welches das Taylor-Funktional analog zum GPU-Programm berechnet. Dabei muss der CPU-Code mehrere Anforderungen erfüllen:

Da moderne CPUs i.d.R. mehrere Cores haben ist es sinnvoll, sich als Referenz auf einen Core zu beschränken. Der CPU-Code wurde deshalb nicht parallelisiert und läuft sequenziell auf einem Core.

Um die Vergleichbarkeit zu gewährleisten wurde der CPU-Code auf single precision beschränkt. Dabei ist jedoch zu beachten, dass moderne CPUs intern Register mit double precision verwenden, so dass kein Geschwindigkeitsunterschied in der Verarbeitung der Werte zu erwarten ist. Der Geschwindigkeitsgewinn durch single precision auf der CPU beschränkt sich auf die geringere Bandbreite, die beim Datentransfer gebraucht wird, da Datensätze nur halb so viel Speicher beanspruchen wie in double precision.

Ausgeführt wurden die Testläufe auf einem Intel Core2 Quad Q6700 mit 2,66GHz.

C-Bibliothek zur FFT-Berechnung: FFTW

Die Programmausführung soll vergleichbar sein. Dies bedeutet, dass der CPU-Code einfach gehalten werden muss und dem gleichen Ablauf wie der GPU-Code folgt. Gleichzeitig muss jedoch sichergestellt werden, dass die Berechnung der Fouriertransformationen genau wie auf der GPU mit maximaler Geschwindigkeit erfolgt.

Zu diesem Zweck wurde die Programmbibloithek „FFTW“ (Fastest Fourier Transform in the West, [Fri05] [Bur12]) für C zur Berechnung der FFT genutzt. FFTW führt die Transformationen optimiert aus und unterstützt standardmäßig erweiterte Befehlssätze wie SSE und SSE2. Das sind spezialisierte Register auf der CPU, die ähnlich wie eine GPU spezialisierte Berechnungen effizient ausführen können. Abb. 4.1 zeigt, dass FFTW im Vergleich mit vielen anderen CPU-basierten FFT-Bibliotheken gute Ergebnisse erzielt und somit eine gute Grundlage für den Geschwindigkeitsvergleich zwischen CPU und GPU darstellt. Da die Berechnung der

FFTs den größten Teil der Rechenzeit in Anspruch nimmt ist es sinnvoll, näher auf die Funktionsweise der dafür genutzten Bibliothek einzugehen.

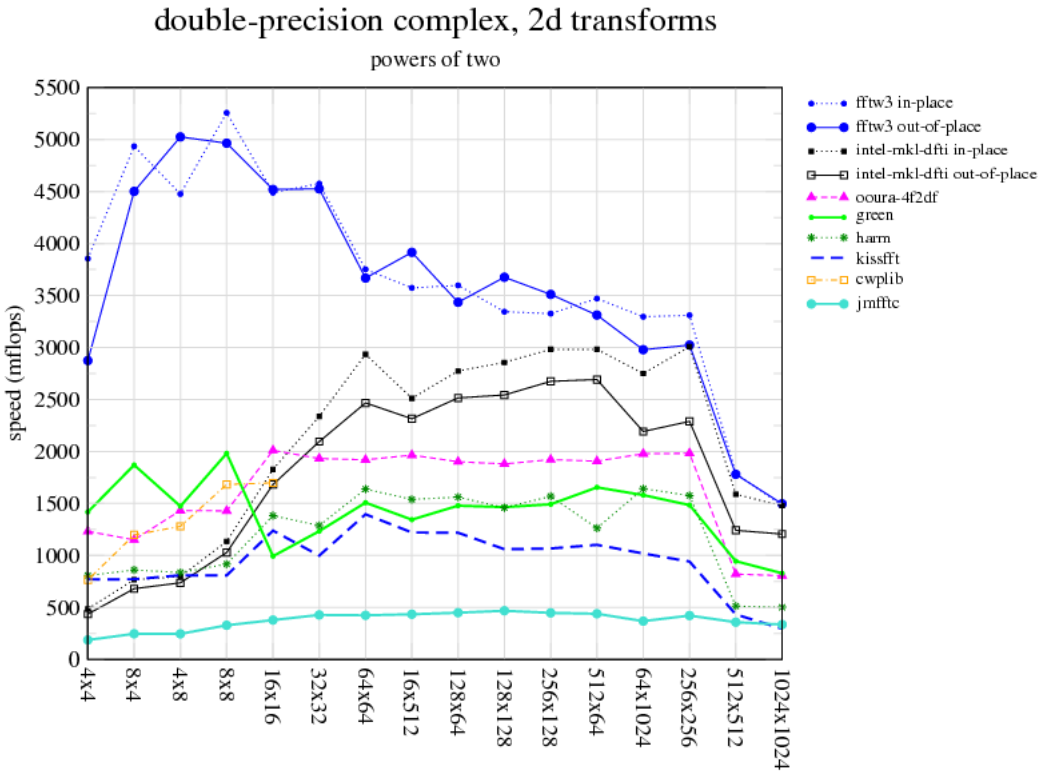

Abbildung 4.1.: Geschwindigkeitsvergleich unterschiedlicher CPU-basierter FFT-Bibliotheken. CPU: 3.6 GHz Pentium 4. Quelle: www.fftw.org/speed

FFTW ist eine C-Bibliothek zur Berechnung diskreter Fourier-Transformationen, die seit über 10 Jahren in Entwicklung ist und zum Zeitpunkt dieser Arbeit in der Version 3.3 vorliegt. FFTW ist die schnellste frei verfügbare FFT-Bibliothek und sie steht auch den herstelleroptimierten Bibliotheken in nichts nach. Dies wird erreicht, indem FFTW diverse Verfahren wie den in Kapitel 4.1.2 beschriebenen Cooley-Tukey-Algorithmus implementiert. Für ein konkretes Problem kann dann aus allen Verfügbaren Berechnungsmethoden die jeweils beste herangezogen werden. Zudem fließen diverse Optimierungen ein. Beispielsweise wird im Radix-2-Algorithmus die binäre Teilung nicht bis zum letzten Element forgesetzt sondern früher abgebrochen, um die kleinelementigen FTs mit einer effizienteren Methode zu berechnen. Dabei

zieht FFTW die Eigenschaften des Problems (d.h. insbesondere die Gittergröße) sowie die Eigenschaften der Rechenarchitektur in Betracht. Das Resultat ist ein hoch optimierter Programmcode, der FFTs auf der CPU um ca. einen Faktor 10 schneller berechnen kann als die direkte (unoptimierte) Implementierung [Bur12].

Das Codebeispiel 4.1 zeigt, wie ein Plan in FFTW erstellt und eine Vorwärts- sowie Rückwärtstransformation eines komplexen Feldes durchgeführt wird.

```
// Definition der genutzten Felder
float           input[size];
fftw_complex    *data, *fft_result, *ifft_result;
fftw_plan       plan_forward, plan_backward;

// Plaene fuer die Hin- und Ruecktransformation werden angelegt
plan_forward  = fftw_plan_dft_1d( size, data, fft_result,
   FFTW_FORWARD, FFTW_ESTIMATE );
plan_backward = fftw_plan_dft_1d( size, fft_result, ifft_result,
    FFTW_BACKWARD, FFTW_ESTIMATE );

// FFTW nutzt diese Plaene, um die FTs durchzufuehren
fftw_execute( plan_forward );
fftw_execute( plan_backward );
```

Listing 4.1: Beispiel einer mit Hilfe von FFTW berechneten Fourier-Transformation. Für das Gitter und die genutzte Rechenarchitektur wird dynamisch ein Plan (eine Sammlung optimaler Algorithmen) generiert, der dann zur Berechnung der FT umgesetzt.

Durch den hohen Grad der Optimierung von FFTW ist sichergestellt, dass die Berechnungen auch auf der CPU effizient erfolgen und ein Geschwindigkeitsvergleich mit der GPU möglich wird.

4.1.4. Vergleich mit der GPU-Implementierung

Besonderheiten der GPU

Zur Implementierung der DFT auf der GPU müssen einige Hindernisse überwunden werden.

Zunächst gibt es die schon erwähnten Speicherbeschränkungen. Ein Datenpunkt in einfacher Genauigkeit benötigt 4 Byte an Speicher. Ein einziger zweidimensionales Feld aus 4096^2 Punkten benötigt also schon 64 MB an Grafikspeicher. Da mehrere solche Datensätze genutzt werden stellt 4096^2 die maximale Gittergröße auf der Grafikkarte dar. Bei einem 8192^2 Gitter würde ein Datensatz 256 MB beanspruchen, so dass selbst eine Grafikkarte mit 4 GB Speicher schon mit wenigen Datensätzen überfordert wäre.

Für zweidimensionale Berechnungen ist dies allerdings mehr als ausreichend. Sogar 2048^2 ist eine ausreichende Gittergröße, um präzise Ergebnisse zu erhalten. Der Grafikspeicher ist somit lediglich für dreidimensionale Berechnungen, für die er maximal 256^3 erlaubt, eine relevante Beschränkung.

Die nächste Frage ist die Einbettung des GPU-Codes in das Host-Programm. Um maximale Leistung sicherzustellen, werden die benötigten Datenfelder hier auf der CPU initialisiert und dann auf die Grafikkarte kopiert. Die gesamte Berechnung findet dann auf der Grafikkarte statt, ohne die CPU hinzuzuziehen. Damit kann die GPU zwar schnell rechnen, leider ist es aber durch diesen Ansatz schwer, Zwischenergebnisse auszugeben, da diese nicht auf der CPU verfügbar sind. Nach der Bearbeitung der Iteration wird lediglich das Endergebnis zurück an die CPU übergeben, so dass es herausgeschrieben werden kann.

Ausgeführt wurden die Testläufe auf einer NVidia GeForce GTX 280. Für Rechnungen mit großen Gittern wurde zudem eine NVidia Tesla M2070 mit 6 GB Speicher genutzt.

CUDA-Bibliothek zur FFT-Berechnung: CUFFT

Zur Berechnung der in dieser Arbeit auftretenden FFTs wird auf der GPU genau wie auf der CPU ebenfalls eine Programmbibliothek genutzt. NVidias CUDA stellt die Bibliothek „CUFFT“ [CUF12] bereit, die sich sowohl in Nutzung als auch Funktionsweise stark an der etablierten C-Bibliothek FFTW orientiert. Abb. 4.2 zeigt die

mit CUFFT erzielbare Geschwindigkeit der Fourier-Transformationen im Vergleich mit der CPU-basierten MKL-Bibliothek, die auch in Abb. 4.1 dargestellt ist.

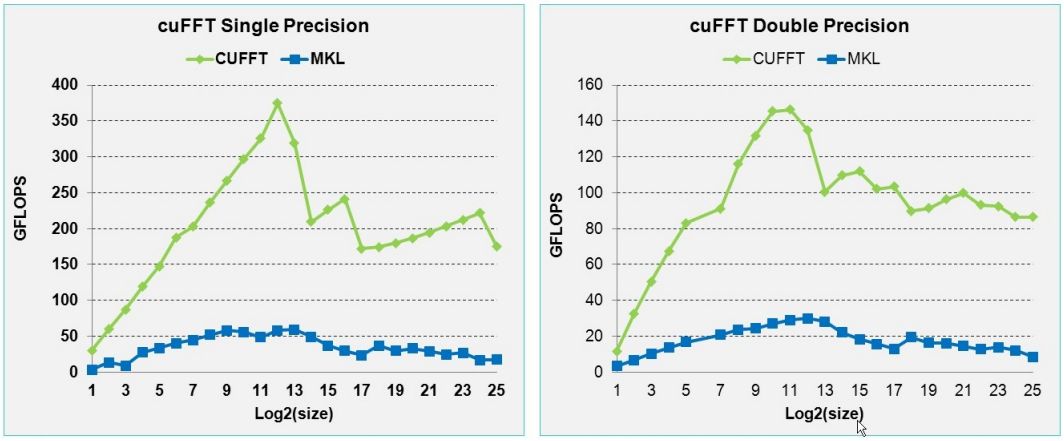

Abbildung 4.2.: Geschwindigkeitsvergleich diverser in CUFFT implementierter Algorithmen mit einer CPU-Referenztransformation unter Nutzung der Intel Math Kernel Library, die ebenfalls in Fig. 4.1 zu finden ist. MKL: Xeon x5680 Six-Core, 3.33 GHz. CuFFT: Tesla M2090. Quelle: http://developer.nvidia.com/cufft

In CUFFT wird ebenfalls vor den Transformationen ein Plan formuliert, der den optimalen Algorithmus für das Problem und die Plattform enthält und mit dessen Hilfe die Transformation effizient berechnet werden kann, was im Codebeispiel 4.2 veranschaulicht wird. Da sich die GPU stark von der CPU unterscheidet wird hier auch bei gleichen Gittern ein anderer Algorithmus als von der FFTW eingesetzt werden. Auf der GPU ist insbesondere die Verteilung der zahlreichen kleinelementigen FTs auf die hohe Anzahl von Kernen eine Herausforderung.

Allerdings sollte an dieser Stelle erwähnt werden, dass deutlich weniger Entwicklungszeit in die relativ neue CUFFT geflossen ist als in die etablierte und hoch optimierte FFTW. CUFFT implementiert noch nicht alle fortgeschrittenen Rechenmethoden, die FFTW zur Geschwindigkeitsmaximierung einsetzt, was man leicht daran sieht, dass der von CUFFT generierte Plan von weniger Parametern abhängt als bei der FFTW. Insbesondere nutzt CUFFT keine Informationen über die zu transformierende Datenstruktur selbst oder die Transformationsrichtung, nur die Gittergröße geht in den Plan ein. Dies betrifft jedoch im Wesentlichen Datensätze,

```
// Definition der genutzten Felder
int                     size_x, size_y;
__device__ cuComplex*   d_n1, d_n2;
__device__ cufftHandle  plan;

// Ein Plan fuer die Feldgroesse wird formuliert
cufftPlan2d(&plan, size_x, size_y, CUFFT_C2C);

// CUFFT nutzt diese Plaene, um die FTs durchzufuehren
cufftExecC2C(plan, d_n1, d_n2, CUFFT_FORWARD);
cufftExecC2C(plan, d_n2, d_n1, CUFFT_INVERSE);
```

Listing 4.2: Beispiel einer mit Hilfe von CUFFT berechneten Fourier-Transformation. Die Syntax ist FFTW nachempfunden.

deren Größe keine Zweierpotenz ist, was für die hier genutzten Gitter nicht relevant ist. Sollte es in Zukunft jedoch noch signifikante Verbesserungen an den Bibliotheken geben so ist zu erwarten, dass die Grafikkarte mehr an Geschwindigkeit gewinnt als die CPU.

4.2. Ergebnisse

Bestimmung einer Dichteverteilung

Die Iteration des Taylor-Funktionals konnte erfolgreich durchgeführt werden. Die Selbstkonsistenz wird auf der Grafikkarte innerhalb weniger Sekunden erreicht und das Ergebnis entspricht der Erwartung. Es gilt $g(r) = \frac{\rho(r)}{\rho_0}$.

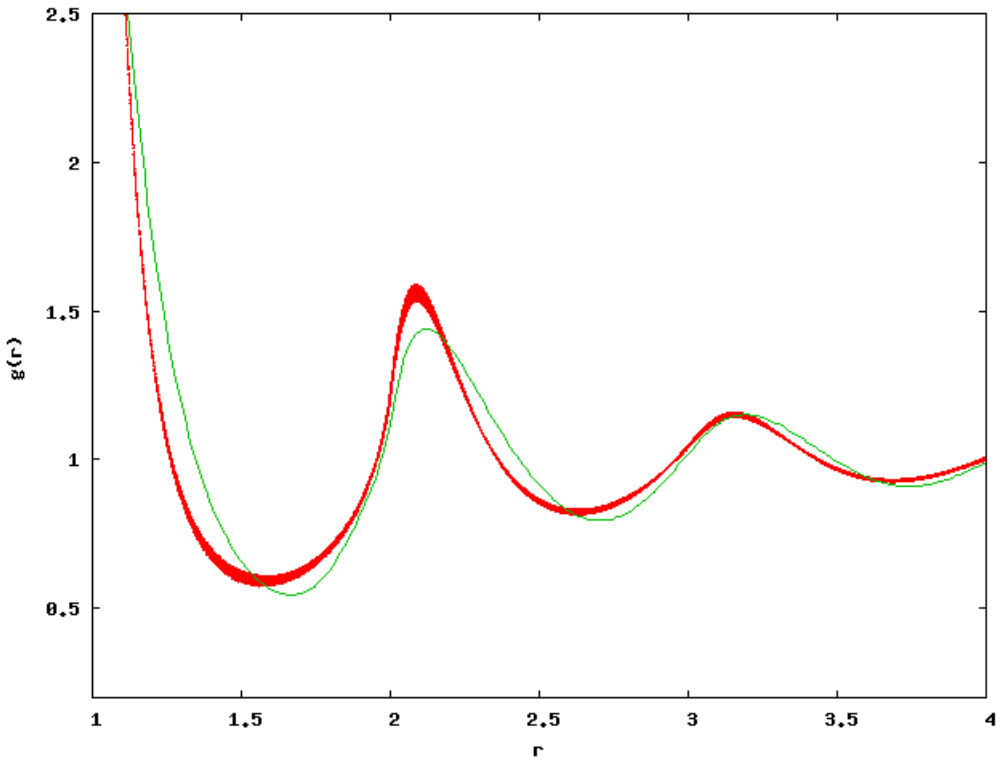

Abbildung 4.3.: Ein auf der GPU unter Nutzung von Gl. (4.9) berechnetes Dichteprofil für eine starre Scheibe bei der Dichte $\rho = 0.764$ (rot) und die gleiche Dichteverteilung bestimmt durch eine MD-Simulation (grün).

Zur Veranschaulichung ist ebenfalls das Ergebnis einer Molekulardynamik-Simulation gezeigt. Diese und alle folgenden MD-Referenzdaten wurden von M. Oettel zur Verfügung gestellt. Die Abweichung der Kurven ist auf die beim Taylor-Funktional eingesetzten Näherungen zurückzuführen. Bei geringeren Dichten wird diese Abweichung kleiner, bei hohen Dichten wird die Taylor-Näherung schlechter und die

Abweichung größer. Zu beachten ist hierbei, dass in Abb. 4.3 um $x = 2$ im Taylor-Funktional kein „Knick“ zu sehen ist, wie ihn die Funtionale zeigen, die später in dieser Arbeit diskutiert werden.

Präzision und Unterschiede in der Berechnung

Zur Veranschaulichung der Datensätze wurde eine Berechnung auf einem kleinen Gitter durchgeführt und ein Ausschnitt betrachtet.

CPU							
0	0	0.450	0.396	0.168	0.874	0	0
0	0.647	0.211	0.596	0.386	0.173	1.039	0
0.450	0.211	0.559	0.426	0.486	0.318	0.174	0.742
0.396	0.596	0.426	0.254	0.179	0.583	0.570	0.196
0.168	0.386	0.486	0.179	0.975	0.355	0.214	0.695
0.874	0.173	0.318	0.583	0.355	0.151	0.064	0.416
0	1.039	0.174	0.570	0.214	0.064	2.055	0
0	0	0.742	0.196	0.695	0.416	0	0

GPU							
0	0	0.441	0.389	0.170	0.851	0	0
0	0.633	0.212	0.582	0.379	0.174	1.016	0
0.441	0.212	0.543	0.419	0.475	0.311	0.176	0.724
0.389	0.582	0.419	0.251	0.182	0.572	0.556	0.197
0.170	0.379	0.475	0.182	0.944	0.351	0.213	0.673
0.851	0.174	0.311	0.572	0.351	0.149	0.067	0.416
0	1.016	0.176	0.556	0.213	0.067	2.004	0
0	0	0.724	0.197	0.673	0.417	0	0

Tabelle 4.1.: Darstellung des Gitters bei einer Größe von 8 und einer Dichte von $\rho = 0.40$, berechnet auf der GPU und CPU, im direkten Vergleich. Die 0-Einträge zeigen die Position des fixierten Testteilchens im zweidimensionalen periodischen Gitter, die Zahlenwerte entsprechen der Dichteverteilung um das Teilchen.

Wie man in Tabelle 4.1 sehen kann liegen die Werte nahe beieinander, sie sind jedoch nicht identisch. Dies ist darauf zurückzuführen, dass CUFFT und FFTW zwar

die gleiche Funktionsweise haben, die Berechnung der FTs allerdings für die jeweilige Plattform optimieren und somit nicht exakt den gleichen Algorithmus verwenden.

Vergleicht man die Dichteverteilungen so sieht man, dass GPU und CPU nahezu die gleichen Ergebnisse liefern. Die Kurven in Abb. 4.4 sind beinahe deckungsgleich.

Um die Auswirkung von doppelter Genauigkeit (double precision, DP) festzustellen, wurde der CPU-Code ebenfalls mit DP ausgeführt. Auch hier sieht man, dass der in Abb. 4.5 gezeigte Dichteverlauf keine signifikanten Abweichungen zeigt. Die Kurven sind exakt identisch.

Single precision kann somit als ausreichend genau zur Iteration der zweidimensionalen DFT angesehen werden. Dies ist plausibel, da einerseits der durch die FFT entstehende numerische Fehler größer ist als die Zahlgenauigkeit und andererseits zusätzlich durch die Gitterauflösung ein Fehler eingeführt wird, so dass das Ergebnis nicht von double precision profitieren kann.

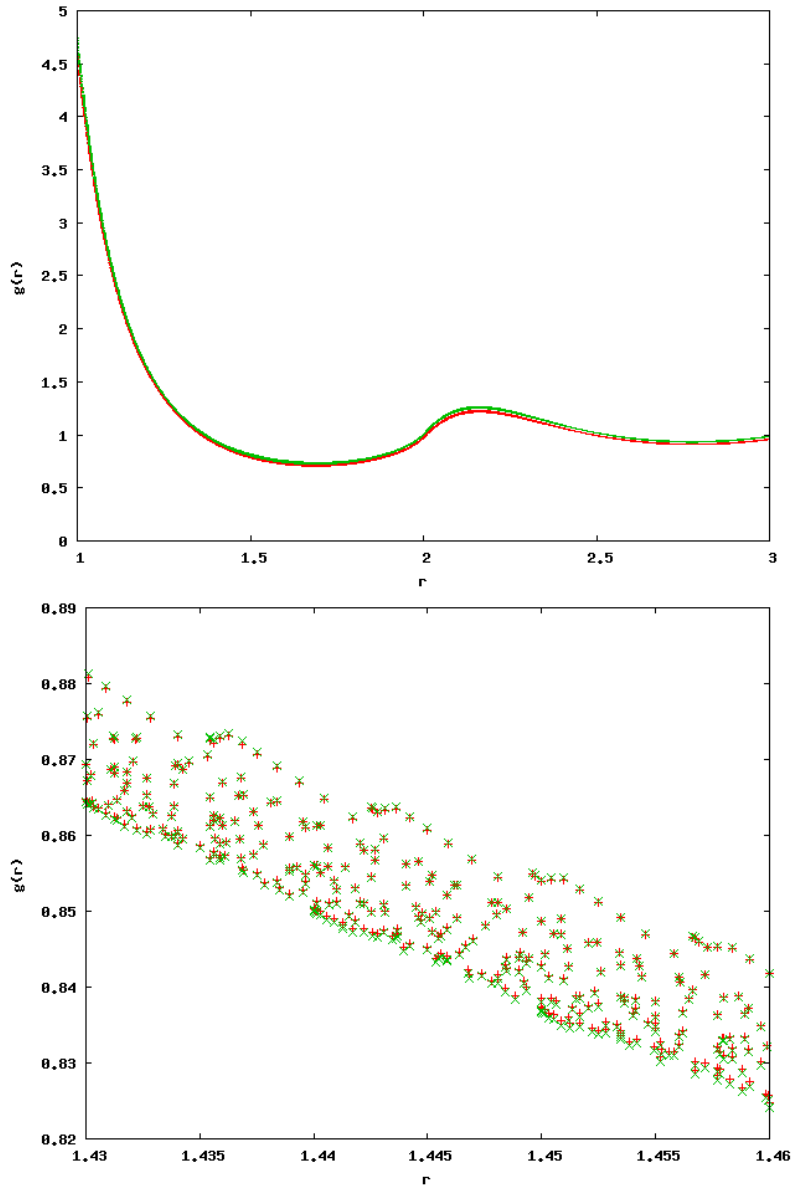

Abbildung 4.4.: Oben: Vergleich der Dichteverteilungen bei $\rho = 0.65$, berechnet in einfacher Genauigkeit auf CPU (rot) und GPU (grün). Da die Kurven identisch verlaufen wurde die GPU-Kurve zur besseren Erkennbarkeit um 0.02 Einheiten in die positive Y-Richtung verschoben. Unten: Ein kleinerer Ausschnitt der gleichen Daten, jedoch nicht verschoben. Die Breite der Kurve ist auf das Gitter zurückzuführen. es sind minimale Unterschiede zwischen GPU und CPU zu erkennen.

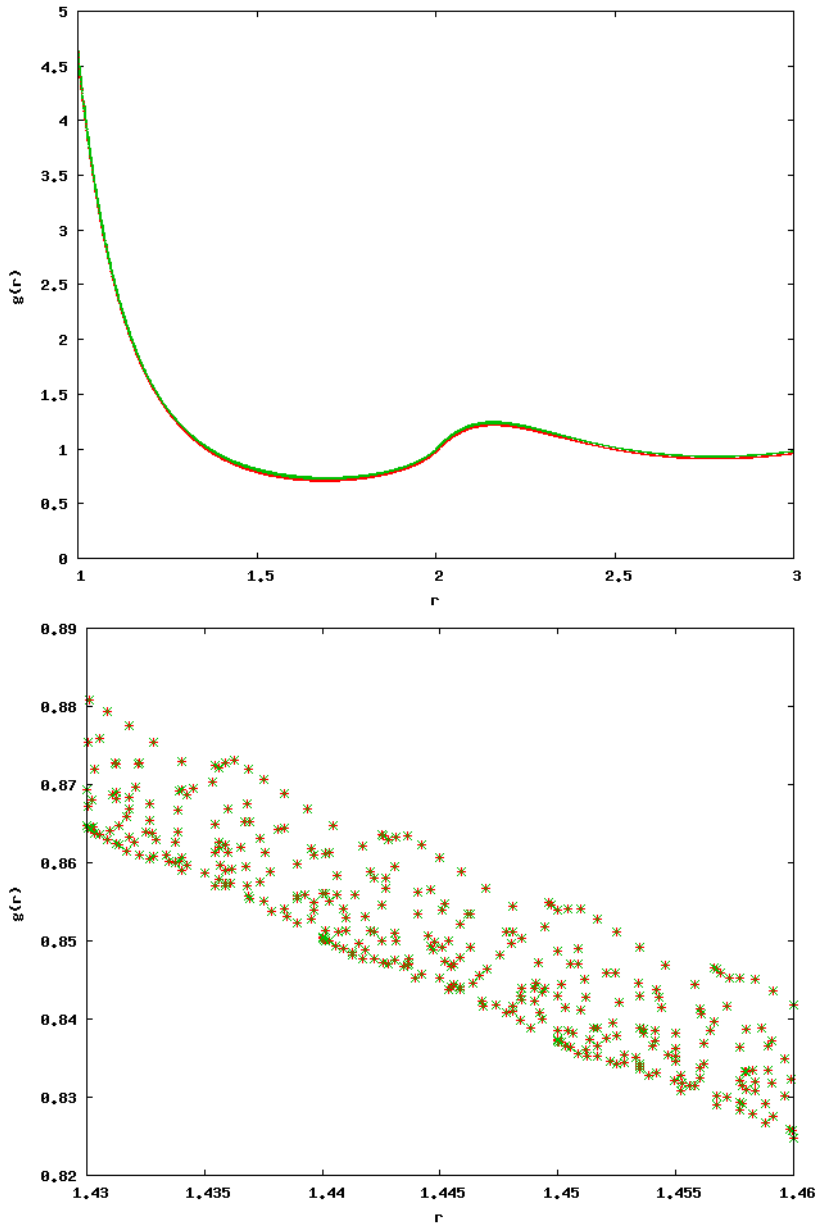

Abbildung 4.5.: Oben: Vergleich der Dichteverteilungen bei $\rho = 0.65$, berechnet in einfacher Genauigkeit (rot) und doppelter Genauigkeit (grün) auf der CPU. Da die Kurven identisch verlaufen wurde die SP-Kurve zur besseren Erkennbarkeit um 0.02 Einheiten in die positive Y-Richtung verschoben. Unten: Ein kleinerer Ausschnitt der gleichen Daten, ebenfalls verschoben. Man sieht, dass die Zahlenwerte exakt identisch sind.

Es ist jedoch denkbar, dass die Zahlgenauigkeit einen Einfluss auf die Geschwindigkeit der Konvergenz hat. Um dies zu prüfen wurden die Interationsschritte bis zur Selbstkonsistenz gezählt.

Dichte ρ	Gittergröße	Schritte GPU	Schritte CPU SP	Schritte CPU DP
0.40	256	85	86	86
0.50	256	95	95	95
0.60	256	100	100	100
0.70	256	104	103	103
0.40	512	81	81	81
0.50	512	89	89	89
0.60	512	103	102	102
0.70	512	122	121	121
0.40	1024	74	75	75
0.50	1024	81	81	81
0.60	1024	92	92	92
0.70	1024	113	112	112

Tabelle 4.2.: Vergleich der benötigten Rechenschritte bis zur Selbstkonsistenz bei verschiedenen Gittergrößen und Dichten.

In Tabelle 4.2 ist zu sehen, dass die CPU sowohl in single als auch in double precision die gleiche Zahl von Rechenschritten benötigt. Die Zahl der Rechenschritte auf der GPU ist sehr ähnlich, weicht jedoch stellenweise um einen Schritt ab. Da die Abbruchbedingung zur Bestimmung der Selbstkonsistenz im Rahmen der Rechengenauigkeit liegt ist dies nicht verwunderlich und der Unterschied von nur einem Iterationsschritt kann durch die unterschiedlichen Algorithmen zur Berechnung der FFT begründet werden. Wie erwartet dauert die Iteration länger, je höher die Dichte ist. Der Gefrierübergang kann nicht erreicht werden, was bei dem genutzten Funktional zu erwarten war.

Geschwindigkeitsvergleich

Zum Vergleich der Geschwindigkeit wurden mehrere Läufe mit unterschiedlichen Gittergrößen und Dichten durchgeführt.

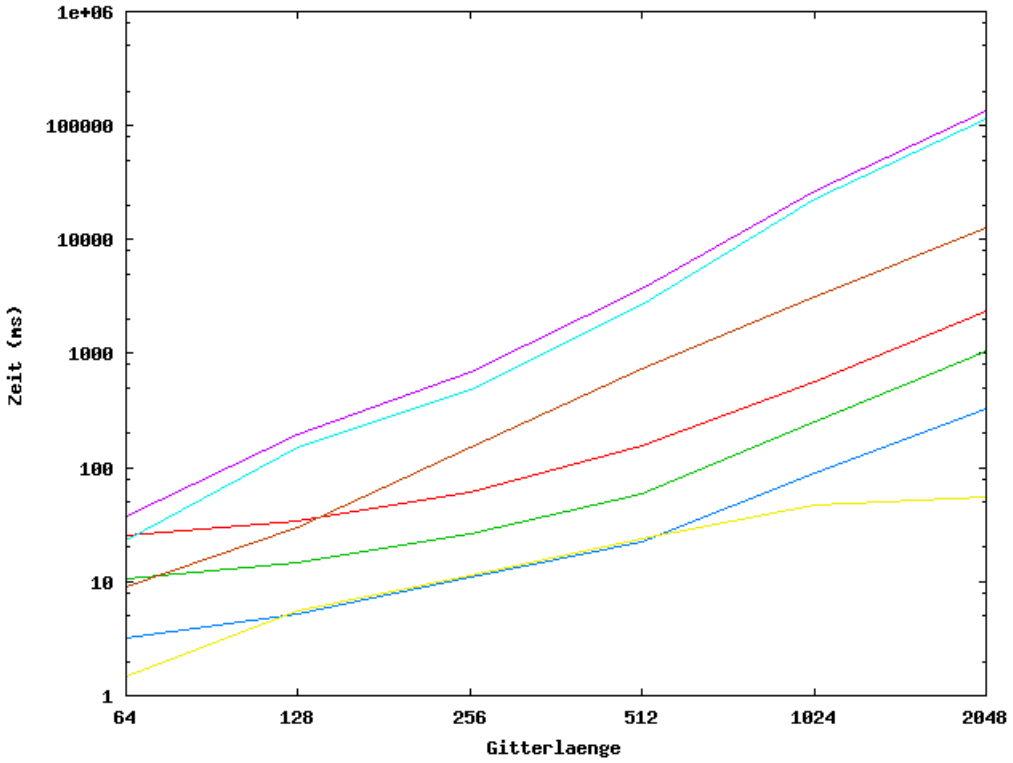

Abbildung 4.6.: Vergleich der Dauer der Iteration bei unterschiedlichen Gittergrößen und Dichten. Gezeigt wird die gesamte Berechnungszeit für die CPU (lila) und die GPU (rot) sowie die Zeit, die von CPU (hellblau) und GPU (grün) zur Berechnung der FFT und die Zeit, die von CPU (braun) und GPU (blau) für die Auswertung der diversen Funktionen auf dem Gitter benötigt wird. Zudem ist der Faktor dargestellt, um den die GPU schneller als die CPU rechnet (gelb).

Man sieht in Abb. 4.6, dass die GPU bei großen Gittern deutlich schneller als eine CPU rechnet, da dort die Parallelisierung effizient erfolgen kann. Kleine Gitter profitieren dagegen kaum. Die Laufzeit hängt ebenfalls schwach von der Dichte ab, hier ist jedoch kein signifikanter Unterschied zwischen GPU und CPU festzustellen. Ist das Gitter groß genug, um von der GPU ausgenutzt zu werden, so ergibt für die hier genutzten Prozessoren ein Geschwindigkeitsvorteil bis zu einem Faktor von 50.

Mit diesem großen Geschwindigkeitsgewinn kann die Umsetzung der DFT auf der Grafikkarte als Erfolg gewertet werden. Die Berechnung erfolgt deutlich schneller und mit ausreichender Genauigkeit, so dass wir mit gutem Gewissen komplexere

Funktionale mit Hilfe der GPU auswerten können.

5. Dichteverteilung in Flüssigkeiten und der Gefrierübergang

5.1. Das Rosenfeld-Funktional

In diesem Kapitel wird das in Kapitel 2 eingeführte Rosenfeld-Funktional für harte Kugeln in drei Dimensionen auf zwei Dimensionen reduziert. Die wesentlichen Unterschiede zu dem in Kapitel 4 eingeführten Taylorfunktional werden herausgestellt und die Implementation des Rosenfeld-Funktionals auf der GPU wird diskutiert. Letztendlich werden Dichteverteilungen sowohl für zwei als auch für drei Teilchen unterschiedliche Nahe am Gefrierübergang berechnet. Dabei wird explizit gezeigt, dass das Rosenfeld-Funktional den Gefrierübergang nicht abbilden kann.

Eigenschaften des Rosenfeld-Funktionals

Das bekannte 3D-Rosenfeldfunktional (siehe Gl. (2.61)) lautet

$$\beta F^{ex} = \int d\mathbf{r} \ \left(-n_0 \ln(1-n_3) + \frac{n_1 n_2 - \mathbf{n}_1 \mathbf{n}_2}{1-n_3} + \frac{n_2^3 - 3 n_2 \mathbf{n}_2 \cdot \mathbf{n}_2}{24\pi(1-n_3)^2} \right) \tag{5.1}$$

mit

$$\begin{aligned} \omega_i^{(3)}(\mathbf{r}) &= \Theta(R_i - r), & \omega_i^{(2)}(\mathbf{r}) &= \delta(R_i - r), \\ \omega_i^{(1)}(\mathbf{r}) &= \frac{\omega_i^{(2)}}{4\pi R_i}, & \omega_i^{(0)}(\mathbf{r}) &= \frac{\omega_i^{(2)}}{4\pi R_i^2}, \\ \boldsymbol{\omega}_i^{(2)}(\mathbf{r}) &= \frac{\mathbf{r}}{r}\omega_i^{(2)}, & \boldsymbol{\omega}_i^{(1)}(\mathbf{r}) &= \frac{\boldsymbol{\omega}_i^{(2)}}{4\pi R_i} \end{aligned} \tag{5.2}$$

und

$$n_\alpha(\mathbf{r}) = \int d\mathbf{r}' \; \rho_i(\mathbf{r}')\omega_i^{(\alpha)}(\mathbf{r}-\mathbf{r}'). \tag{5.3}$$

Dieses Funktional folgt der Form

$$\beta F^{ex}[\rho_i(\mathbf{r})] = \int d^D x \; \Phi(n_\alpha(\mathbf{r})) \tag{5.4}$$

mit der Annahme, dass F^{ex} nur von den fundamentalen geometrisch motivierten D-dimensionalen gewichteten Dichten

$$n_\alpha(\mathbf{r}) = \sum_i \int d^D\mathbf{r}' \; \rho_i(\mathbf{r}')\omega_i^{(\alpha)}(\mathbf{r}-\mathbf{r}') \tag{5.5}$$

abhängt. [Ros90] zeigt, dass dieser Ansatz für zwei Dimensionen und mit nur einer Teilchensorte auf

$$F^{ex} = \int d^2 r \left[-n_0 \cdot \ln(1-n_2) + \frac{1}{4\pi}\frac{n_1^2 - \mathbf{n}^2}{1-n_2}\right] \tag{5.6}$$

$$\text{mit } n_i(\mathbf{r}) = \int \rho^{(2D)}(\mathbf{r}') \cdot \omega_i(\mathbf{r}-\mathbf{r}')d\mathbf{r}' \tag{5.7}$$

$$\text{und } \omega_0 = \frac{\omega_1}{2\pi}R,\; \omega_1 = \delta(R-r),\; \omega_2 = \Theta(R-r),\; \boldsymbol{\omega} = -\boldsymbol{\nabla}\omega_2 \tag{5.8}$$

führt, welches als das zweidimensionale Rosenfeld-Funktional bekannt ist.

Dieses Funktional produziert eine deutlich realistischere Dichteverteilung als das Taylor-Funktional, wie in Abb. 5.1 zu sehen ist. Die Implementierung des Rosenfeld-Funktionals erlaubt es uns, realistische Dichteprofile auf der Grafikkarte zu berechnen. Die Ergebnisse stimmen mit denen aus der Literatur überein, so dass die korrekte Arbeitsweise der Grafikkarte nochmals bestätigt wird.

Erhöht man die Dichte so sieht man, dass das Rosenfeld-Funktion (im Gegensatz zum Taylor-Funktional) einen „Knick“ beim doppelten Teilchenabstand produziert, wie in Abb. 5.2 gezeigt ist. Allerdings ist dies nur ein “Knick” und kein “Buckel”. Diese Form ist von großer Bedeutung für den Gerfierübergang.

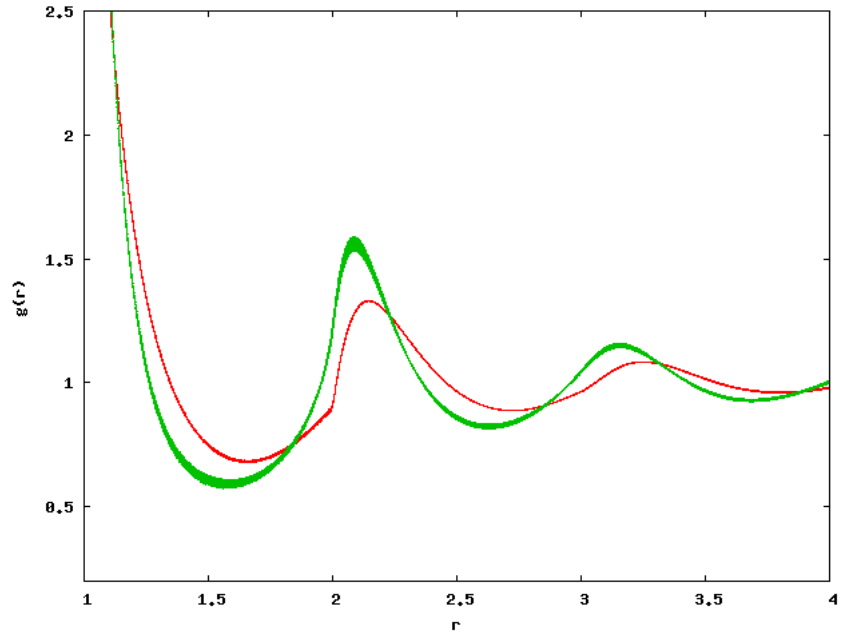

Abbildung 5.1.: $g(r)$ um eine feste Scheibe mit Radius $r = 0.5$ bei $\rho = 0.764$, berechnet mit dem Taylor-Funktional (grün) und dem Rosenfeld-Funktional (rot).

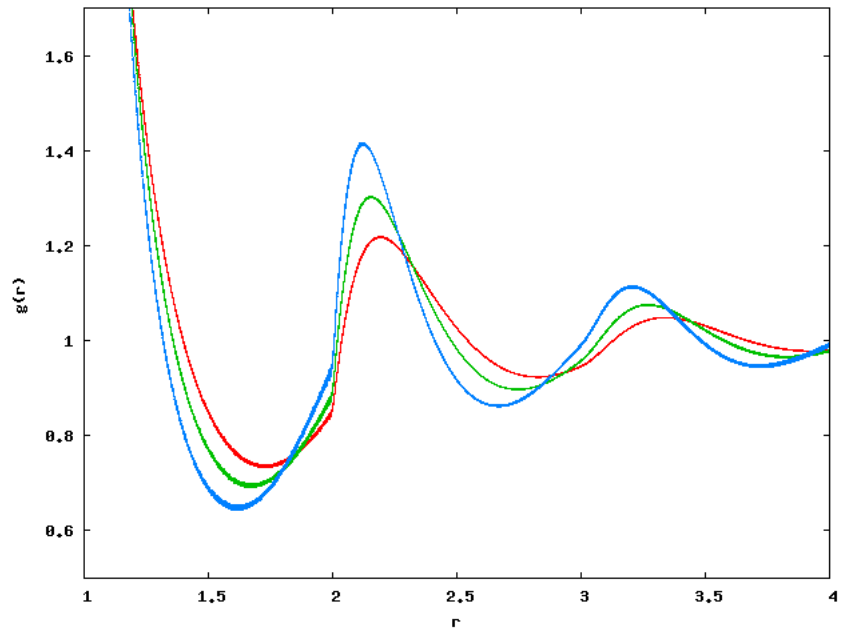

Abbildung 5.2.: Dichteverteilung bei $\rho = 0.7$ (rot), $\rho = 0.75$ (grün) und $\rho = 0.8$ (blau). Der „Knick“ bei $r = 2$ ist deutlich zu sehen.

Die strukturelle Vorstufe des Gefrierübergangs

Es ist bekannt, dass ein System harter Scheiben bei hoher Dichte gefriert, d.h. dass die Scheiben eine Kristallstruktur einnehmen. In Abb. 5.3 werden Daten einer MD-Simulation für ein solches System harter Scheiben bei hohen Dichten gezeigt.

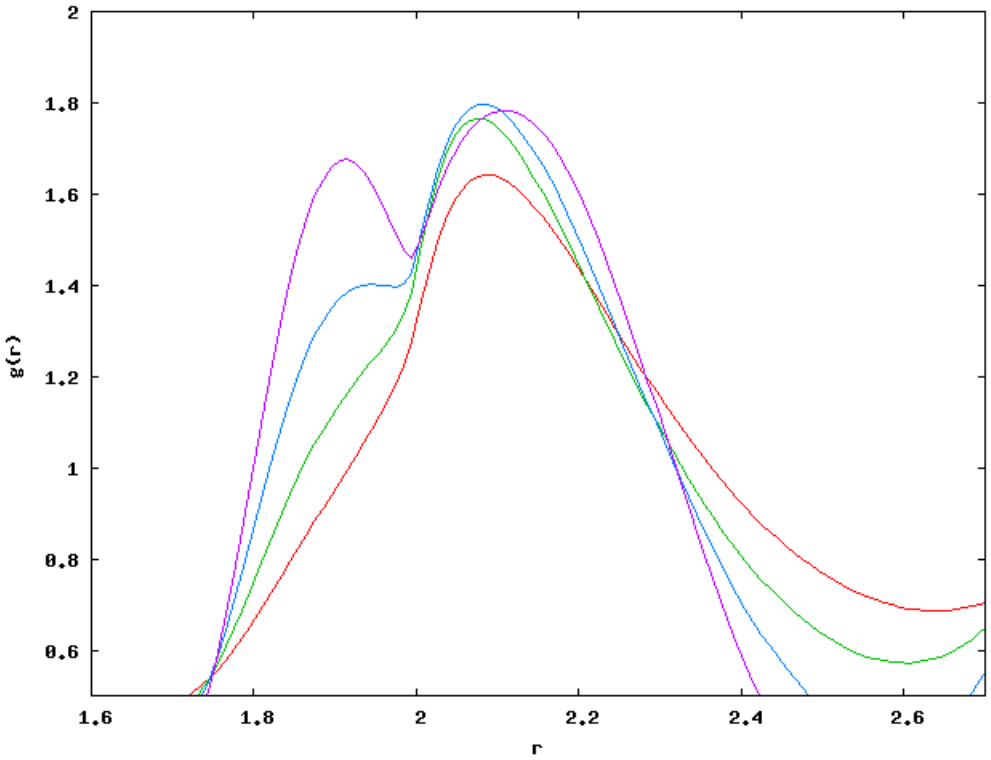

Abbildung 5.3.: Dichteverteilung bei $\rho = 0.65$ (rot), $\rho = 0.68$ (grün), $\rho = 0.70$ (blau) und $\rho = 0.72$ (lila). Der „Knick“ bei $r = 2$ zu sehen, jedoch bildet sich vor diesem “Knick” bei hohen Dichten ein “Buckel”.

Dabei ist der “Knick”, genau wie in Abb. 5.2, zu sehen. Vor diesem “Knick” ist jedoch ein “Buckel”. Dieser ist stärker ausgeprägt, je weiter die Dichte erhöht wird, d.h. je näher das System am Gefrierpunkt ist. Da der “Buckel” den Gefrierübergang einleitet, wird er als strukturelle Vorstufe (“structural precursor”) bezeichnet, siehe [Tor98]. Durch ihn wird eine höhere Aufenthaltswahrscheinlichkeit in einem bestimmten Abstand, der von den normalen Dichtefluktuationen in der flüssigen Phase abweicht, ausgedrückt. Dies ist genau der Teilchenabstand der Kristallstruktur in

der festen Phase.

Die strukturelle Vorstufe des Gefrierübergangs wurde auch mit Hilfe anderer Methoden untersucht. Beispielsweise kann man ihn ebenfalls mit Hilfe inhomogener Integralgleichungen nachweisen [Bra08]. Es kann gezeigt werden, dass diese Stufe den Phasenübergang einleitet [Tru98].

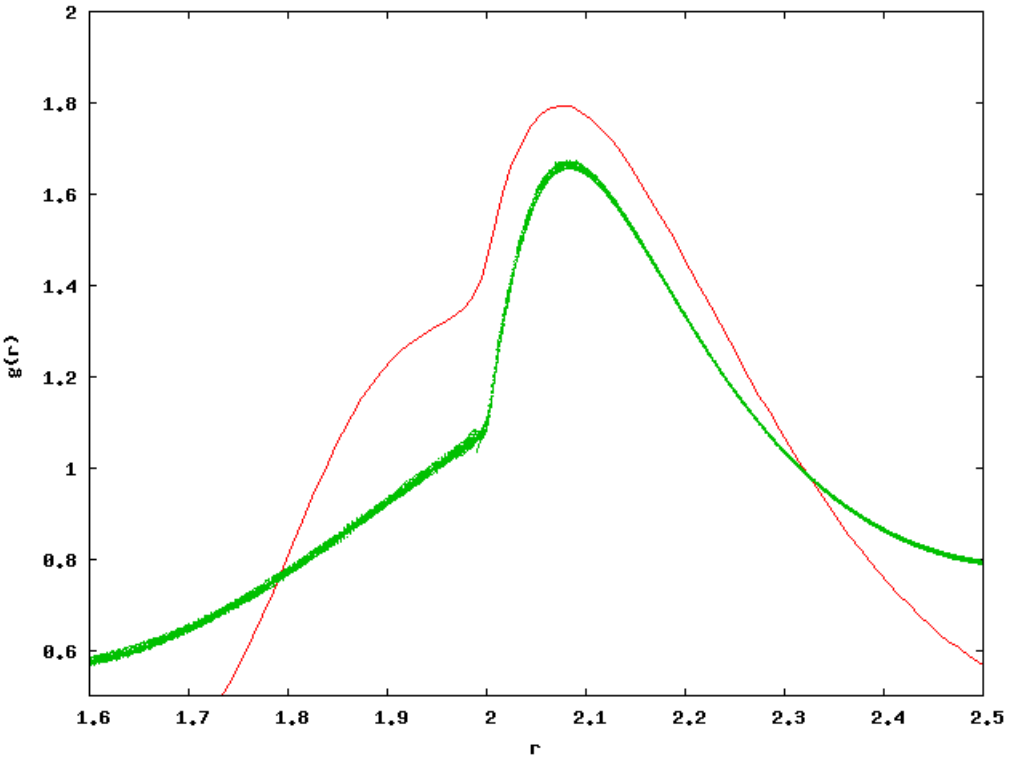

Abbildung 5.4.: Dichteverteilung bei $\rho = 0.69$, berechnet mit einer MD-Simulation (rot) und mit dem Rosenfeld-Funktional (grün).

Betrachten wir das Ergebnis der DFT-Berechnung mit Hilfe des Rosenfeld-Funktionals im Detail so sehen wir, dass der in den MD-Simulationen auftauchende "Buckel" nicht reproduziert werden kann. Das Rosenfeld-Funktional bildet die strukturelle Vorstufe des Gefrierübergangs somit nicht ab, wie in Abb. 5.4 zu sehen ist.

Besonderheiten bei der Implementierung auf der Grafikkarte

Ein großer Vorteil der DFT auf der Grafikkarte ist, dass die wesentlichen Rechenschritte unabhängig vom benutzten Funktional sind. Wie beim Taylorfunktional wird der größte Teil der Rechenzeit darauf verwendet, die Fouriertransformationen zu berechnen. Anstatt nur die $c^{(2)}$-Funktion zu falten müssen hier nun allerdings fünf Faltungen, drei für die skalaren gewichteten Dichten und zwei für die gewichtete Vektordichte, berechnet werden. Um jede dieser gewichteten Dichten berechnen zu können muss jeweils eine Faltung mit der Gewichtsfunktion ausgeführt werden, was die Zahl der Fouriertransformationen wiederum verdoppelt. Zudem müssen alle Gewichtsfunktionen und gewichteten Dichten im Grafikspeicher bereitgehalten werden, so dass auch der Speicherbedarf im Vergleich zu dem Taylorfunktional stark ansteigt. Gittergrößen von maximal 2048^2 sind auf einer Karte mit 1GB Speicher realisierbar.

5.2. Drei-Teilchen-Korrelationsfunktion

Fixierung zweier Testteilchen

Die Fixierung eines Testteilches erlaubt es uns nicht, Gitterstrukturen zu untersuchen. Um die räumliche Anordnung mehrer Teilchen analysieren zu können müssen wir mindestens zwei Testteilchen und die Dichteverteilung um diese herum betrachten, wie in Abb. 5.5 gezeigt wird.

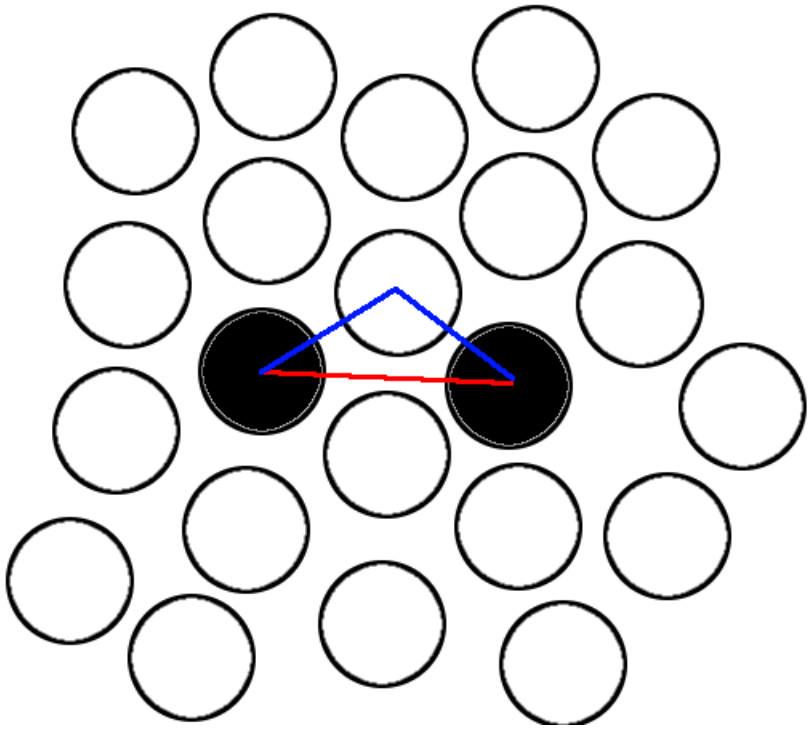

Abbildung 5.5.: Veranschaulichung zweier fixierter Testteilchen (schwarz gefüllt) im Abstand d (rote Linie). Mit Hilfe von zwei fixierten Teilchen kann eine Mehrteilchen-Korrelationsfunktion zu den umliegenden Scheiben der Flüssigkeit (blaue Linien) berechnet werden.

Allerdings genügt es nicht, zwei Teilchen bei einem fixen Abstand zu betrachten. Um eine vollständige Korrelationsfunktion zu erhalten ist es nötig, alle möglichen Teilchenabstände der beiden fixierten Testteilchen durchzufahren. Auf einem Gitter der Größe n gibt es somit aus Symmetriegründen $\frac{n}{2} - 1$ mögliche Abstande der

beien Teilchen, die alle berechnet werden müssen. Hier zeigt sich der große Vorteil der Programmierung auf der GPU, da der hohe Geschwindigkeitsgewinn auf der Grafikkarte bei diesen zahlreichen Berechnungen zu tragen kommt.

Die Implementierung der beiden Testteilchen wird durch eine Änderung des Potentials erreicht. Bisher wurde eine harte Scheibe mit Radius $\frac{R}{2}$ fixiert, deren Ausschlusskreis somit den Radius R hat und Potential lautete

$$V(\mathbf{r}) = \begin{cases} 0, & \text{wenn } |\mathbf{r}| \geq R \\ \infty, & \text{wenn } |\mathbf{r}| < R \end{cases}. \tag{5.9}$$

Für zwei Teilchen mit dem gleichen Radius wird das modifizierte Potential

$$V(\mathbf{r}) = \begin{cases} \infty, & \text{wenn } |\mathbf{r} - \frac{\mathbf{d}}{2}| < R \\ \infty, & \text{wenn } |\mathbf{r} + \frac{\mathbf{d}}{2}| < R \\ 0, & \text{sonst} \end{cases}. \tag{5.10}$$

genutzt, wobei $\mathbf{d}$ den Relativvektor zwischen den beiden Teilchen und sein Betrag d den Teilchenabstand darstellt. Durch die Einführung dieses Relativvektors ist das Problem nicht mehr rotationssymmetrisch. Bis zu diesem Punkt wäre es möglich gewesen, die Berechnungen auf eine Dimension zu reduzieren und die zweidimensionale Lösung über die Kugelsymmetrie zu rekonstruieren. Erst die Einführung eines zweiten Testteilchens rechtfertigt die vollständige Berechnung in zwei Dimensionen. Es gibt allerdings eine verbleibende Symmetrie, die Spiegelsymmetrie an $\mathbf{d}$. Legt man die x-Achse des Koordinatensystems entlang $\mathbf{d}$ so kann die Achsensymmetrie genutzt und der Rechenaufwand minimiert werden.

Bestimmung der Dichteverteilung

Mit diesem neuen Potential lässt sich nun die Dichte um die Testteilchen bestimmen.

Interessant ist insbesondere der Bereich um $d = 2r$, d.h. der Abstand, bei ein Teilchen exakt zwischen die fixierten Teilchen passt. In Abb. 5.6 ist zu sehen, dass die Dichte zwischen den Testteilchen sehr stark ansteigt. Für Teilchen, die weiter

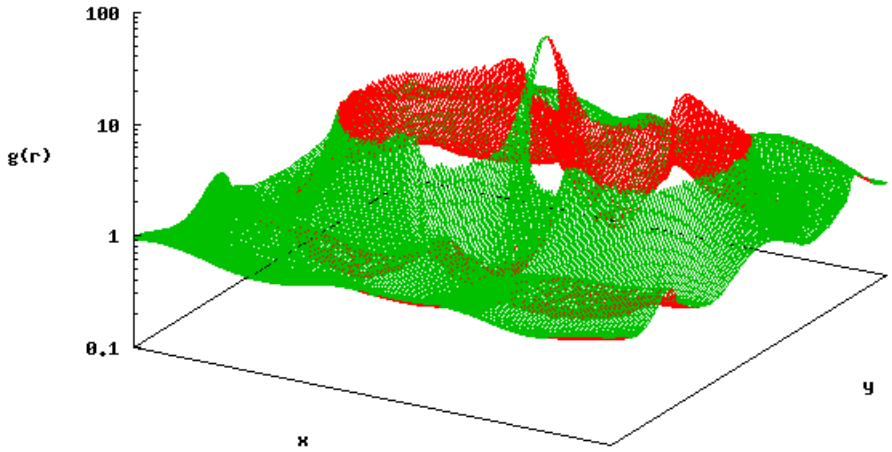

Abbildung 5.6.: Dichteverteilung bei $d = 2r$ und $\rho = 0.828$.

voneinander entfernt sind, geht die Dichteverteilung wie erwartet in die aus der Paarkorrelation (mit einem fixierten Teilchen) über, da sich die beiden Teilchen bei einem ausreichend großen Abstand nicht mehr stark beeinflussen.

In Abb. 5.7 sind exemplarisch zwei Dichteverteilungen um Teilchen mit unterschiedlichem Abstand, jedoch nicht nahe $d = 2r$, zu sehen.

Durch die so gewonnen Informationen lassen sich interessante Aspekte der Flüssigkeit berechnen. Betrachtet man eine der beiden fixierten Scheiben so zieht man, dass die Dichteverteilung bezüglich der x-Achse nicht mehr symmetrisch ist. Durch die unterschiedlichen Dichten auf den Seiten der Scheibe entsteht eine resultierende Kraft, die die Teilchen entweder zusammen- oder auseinanderdrückt.

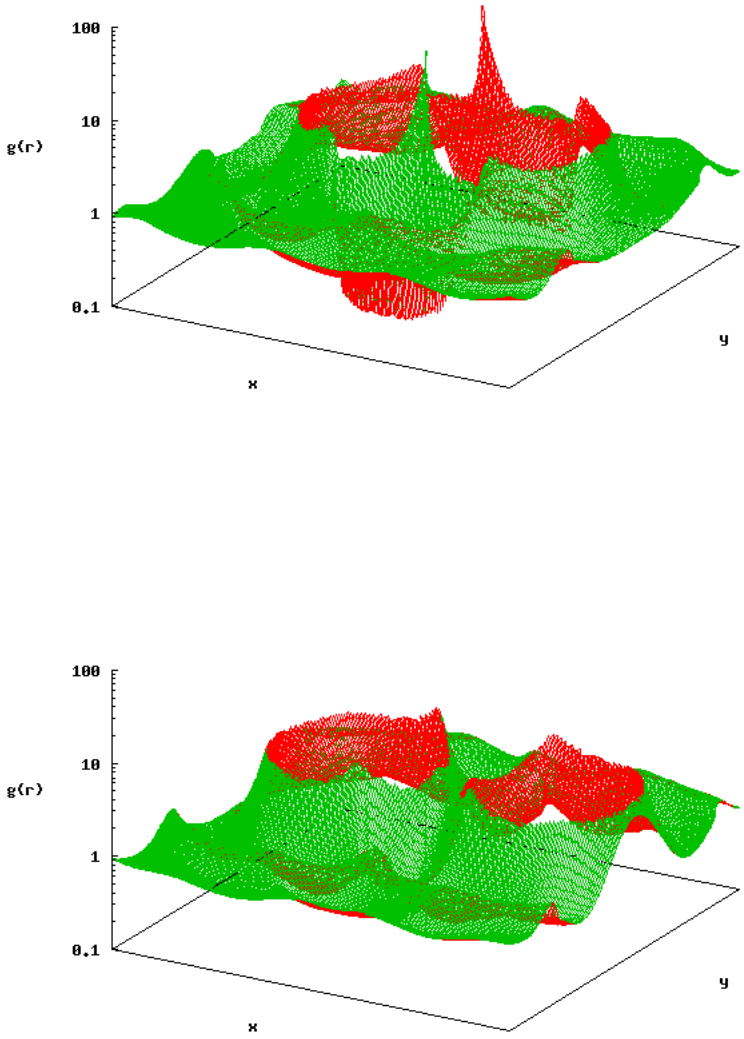

Abbildung 5.7.: Dichteverteilung für verschiedene Teilchenabstände bei $\rho = 0.828$ mit $d < 2r$ (oben) und $d > 2r$ (unten).

5.3. Kräfte zwischen Teilchen

Bestimmung der auf die Teilchen wirkenden Kräfte

Aus dem nun bekannten Dichteverlauf lassen sich die auf die Teilchen wirkenden Kräfte bestimmen. Dabei nutzen wir, dass nach [Han06] mit dem "Potential of mean force (PMF)"-Ansatz

$$\beta W^r = -\ln g(r) \tag{5.11}$$

gilt. Durch Differenziation nach r erhalten wir die auf ein Testteilchen wirkende Kraft F^r

$$\beta F^r = -\partial_r \ln g(r) \tag{5.12}$$

für ein fixiertes Testteilchen. Da $g(r)$ bekannt ist, lässt sich die Kraft aus dieser Gleichung einfach bestimmen.

Bei zwei fixierten Testteilchen ist die resultierende Kraft auf eines der Teilchen genau die Summe aller Teilkräfte. Allgemein gilt somit

$$\mathbf{F} = \int d\mathbf{r}\rho(\mathbf{r}) \left(-\boldsymbol{\nabla} V(\mathbf{r})\right). \tag{5.13}$$

Das hier genutzte Potential harter Scheiben hat jedoch eine Stufe am Rand des Teilchens, so dass das Integral nicht direkt berechenbar ist. Deshalb nutzen wir, dass

$$\boldsymbol{\nabla} e^{-\beta V(\mathbf{r})} = -\beta \boldsymbol{\nabla} V(\mathbf{r}) e^{-\beta V(\mathbf{r})} \tag{5.14}$$

gilt. Damit erhalten wir

$$\beta \mathbf{F} = \int d\mathbf{r} \left[\rho(\mathbf{r}) e^{\beta V(\mathbf{r})}\right] \boldsymbol{\nabla} e^{-\beta V(\mathbf{r})}. \tag{5.15}$$

Setzen wir nun das Potential

$$V(\mathbf{r}) = \begin{cases} 0, & \text{wenn } |\mathbf{r}| \geq R \\ \infty, & \text{wenn } |\mathbf{r}| < R \end{cases}. \tag{5.16}$$

ein so sehen wir, dass $e^{-\beta V(\mathbf{r})}$ zu einer Stufenfunktion wird. Die Ableitung davon ist somit eine Deltafunktion, die dafür sorgt, dass nur der Rand des Teilchens einen Beitrag zum Integral leistet. Dort gilt $e^{\beta V(\mathbf{r})} = 1$ und das Integral kann auf eine Integration über den Rand reduziert werden,

$$\mathbf{F} = \int_{\partial R} d\phi \; 2\pi \cdot 2R\hat{\mathbf{R}}\rho(R,\phi). \tag{5.17}$$

Dabei ist ∂R der Rand eines der Teilchen und die Dichte wird über den Winkel ϕ integriert, $\hat{\mathbf{R}}$ ist die Normale auf dem Teilchenrand. Aufgrund der Symmetrie des Systems wissen wir, dass sich die Kräfte in y-Richtung wegheben müssen. Die resultierende Kraft hat also nur einen x-Anteil, welchen wir durch Multiplikation von Gl. (5.17) mit $\hat{e}_x$ erhalten. Dieses Integral kann nun numerisch auf dem Gitter ausgewertet werden,

$$F_x \approx \frac{1}{N}\sum_{n=0}^{N} \; 4\pi R \cos\left(2\pi\frac{n}{N}\right)\rho\left(R, \left(2\pi\frac{n}{N}\right)\right), \tag{5.18}$$

so dass wir einen zweiten Weg zur Kraftbestimmung auf Testteilchen gefunden haben. Es besteht keine direkte Verbindung zwischen diesem Weg und Gl. (5.12), so dass diese beiden unterschiedlichen Methoden zur Berechnung der gleichen physikalischen Größe einen sehr guten Test sowohl an die Theorie als auch an die Güte der Simulation darstellen.

Diese Kraftberechnung erlaubt es uns, die Unterschiede in der Nähe des Gefrierübergangs für ein und für zwei fixierte Testteilchen näher zu untersuchen.

Wie wir in Abb. 5.8 sehen, unterscheiden sich die Kräftekurven für ein und zwei fixierte Testteilchen deutlich im Bereich des doppelten Teilchenabstands. Die Annäherung an den Gefrierübergang erlaubt es uns, diesen Unterschied zu interpretieren.

Interpretation der Kräfte in der Nähe des Gefrierpunkts

Die Struktur der festen Phase zeichnet sich dadurch aus, dass Teilchen an festen Positionen mit definiertem Abstand sitzen. Damit dies ein stabiler Zustand ist, müssen die Teilchen an dieser Position kräftefrei sein. Betrachtet man die Kräftekurven in Abb. 5.9 so sieht man, dass durch den "Buckel", der sich bei hohen Dichten bildet, eine Kraft von Null bei einem Abstand $d \approx 1.96$ und einer Dichte von $\eta = \frac{\pi}{4}\rho = 0.70$

erreicht wird.

Betrachten wir den mit Hilfe des Rosenfeld-Funktionals berechneten Kraftverlauf so sehen wir, dass ein schwach ausgeprägter "Buckel" auftritt, der Nulldurchgang der Kraft jedoch nicht erreicht werden kann. Zudem ist gerade bei hohen Dichten keine gute Übereinstimmung mit den MD-Daten mehr gegeben. Aus diesem Grund wenden wir uns einem neuen Funktional zu, das den Gefrierübergang korrekter abbilden soll.

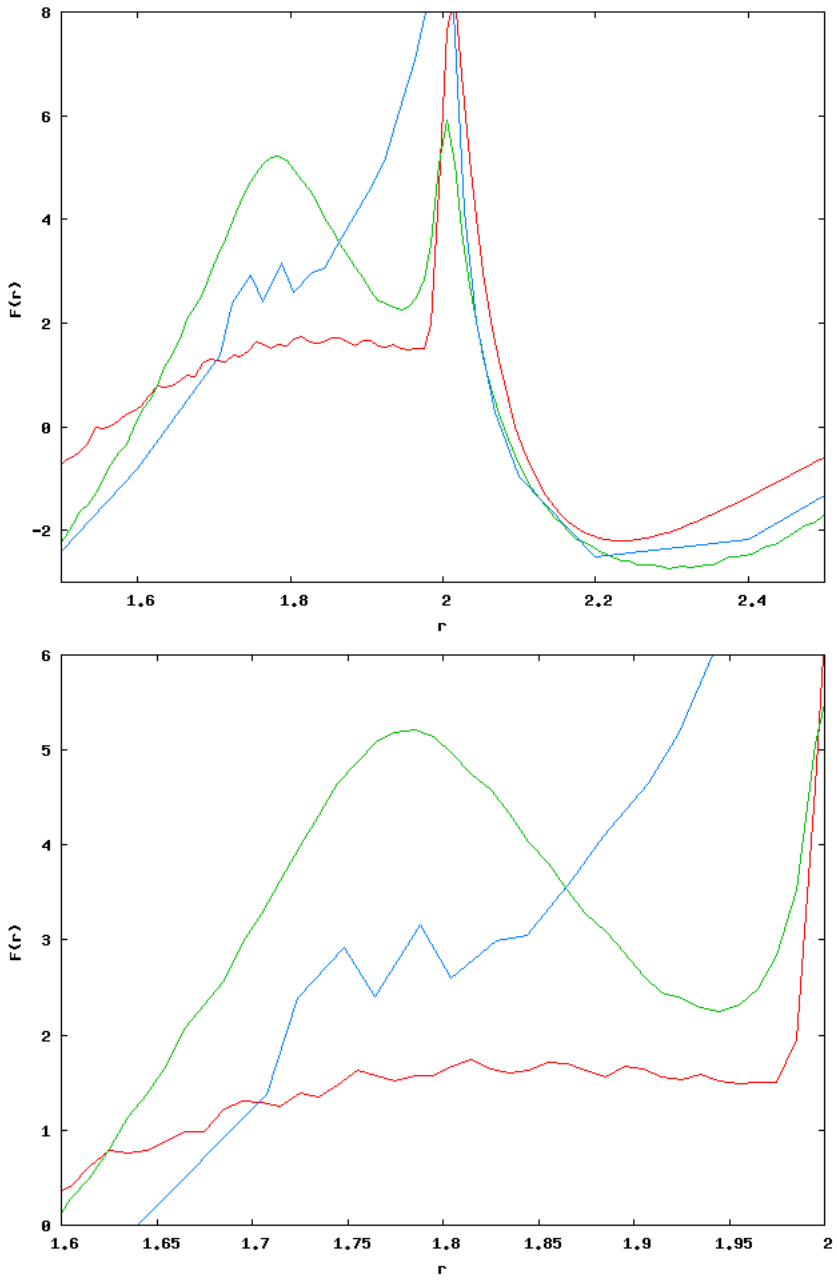

Abbildung 5.8.: Kraft auf ein Testteilchen für ein (nach Gl. (5.12), rot) und zwei (nach Gl. (5.17), grün) fixierte Teilchen bei $\rho = 0.853$. Als Referenz ist das Ergebnis einer MD-Simulation (blau) gezeigt.

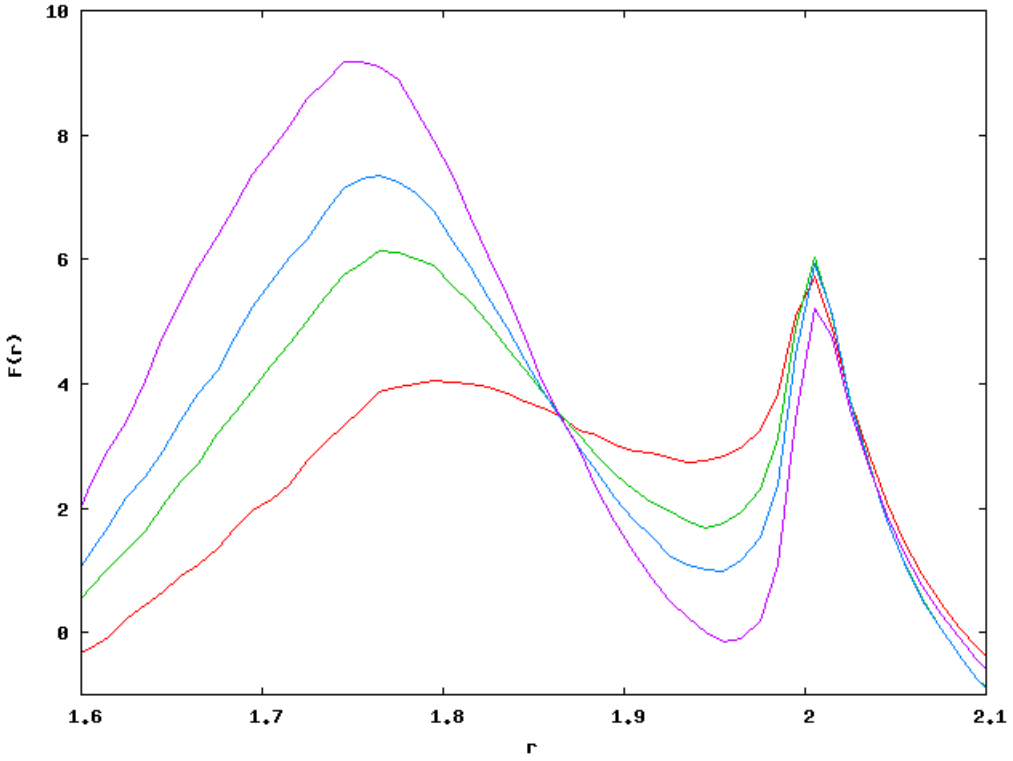

Abbildung 5.9.: Mit Hilfe einer MD-Simulation bestimmte Kräfte auf Testteilchen bei $\eta = 0.65$ (rot), $\eta = 0.68$ (grün), $\eta = 0.69$ (blau) und $\eta = 0.70$ (lila).

6. Das Tensorfunktional

6.1. Herleitung

Da das Rosenfeld-Funktional den Gefrierübergang für harte Scheiben nicht abbilden kann ist ein besseres Funktional nötig. Der in Kapitel 2 beschriebene "dimensional crossover" ermöglicht es Korrekturterme zu finden, die zu einem solchen Funktional führen. Die Herleitung des neuen Funktionals erfolgt nach [Oet11], [Ros98] und [Tar96]. Dazu betrachten wir zunächst eine einfache Aussparung mit dem Radius R. Da diese Aussparung von nur exakt einem Teilchen ausgefüllt werden kann handelt es sich um ein 0-dimensionales Problem in der DFT. Die Energiefunktional für eine Aussparung, die nur ein Teilchen aufnehmen kann, ist exakt bekannt und lautet

$$\beta F_0^{ex} = (1 - N) \cdot \ln(1 - N) + N, \tag{6.1}$$

wobei N die mittlere Besetzungszahl darstellt.

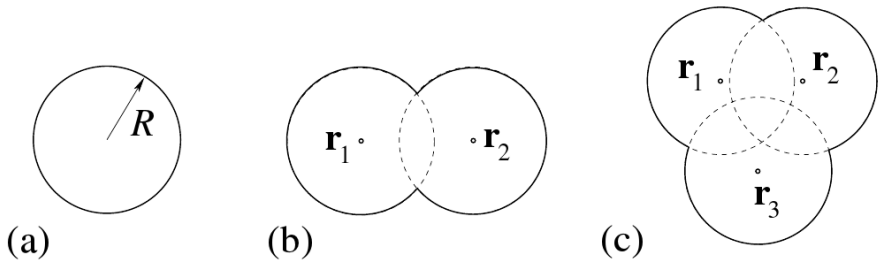

Abbildung 6.1.: Drei mögliche Aussparungen, in denen jeweils ein Teilchen mit dem Radius R platziert werden kann.[Oet11]

Die Aussparung ist in Abb. 6.1 (a) illustriert. Erweitert man diese Aussparung

nun wie in Abb. 6.1 (b) gezeigt, so dass immer noch nur exakt ein Teilchen die Aussparung füllt, dieses jedoch zwei Positionen einnehmen kann, so erhalten wir in zwei Dimensionen

$$F_1^{ex} = -\int d^2r\ n_0 \ln(1-n_2) - \frac{1}{1-n_2}\int d^2R_1\ d^2R_2\ \delta(R-R_1)\delta(R-R_2) \\ \rho(\mathbf{r}+\mathbf{R}_1)\rho(\mathbf{r}+\mathbf{R}_2)\ P_2(\mathbf{R}_1-\mathbf{R}_2) \tag{6.2}$$

als Energiefunktional, wobei

$$P_2 = \frac{1}{\pi}\frac{r}{2R}\sqrt{1-\frac{r^2}{4R^2}}\arcsin\frac{r}{2R}, r = |\mathbf{r}| \tag{6.3}$$

die Integrationen über R_1 und R_2 verbindet, was die Berechnung des Integrals erschwert. Dies beschreibt auch schon die Aussparungen in Abb. 6.1 (c) korrekt, solange es einen gemeinsamen Überlapp aller drei Kreise gibt. Existiert ein Loch in der Mitte so wird die Aussparung nicht korrekt durch dieses Funktional beschrieben, was jedoch hier vernachlässigt wird. Die aus der "Fundamental Measure Theory" bekannte Separation $F^{ex} = \int \phi\left(n_i(\mathbf{r})\right)$ mit $n_i = \omega_i * \rho$ ist nicht gegeben. Um diese Berechnung effizient ausführen zu können ist es nötig, R_1 und R_2 in P_2 zu separieren. Dazu startet man mit der Reihenentwicklung

$$P_2\left(\mathbf{R}_1-\mathbf{R}_2\right) = \frac{1}{\pi}\left(x^2 - \sum_{n=2}^{\infty}\frac{(2n-4)!!}{(2n-1)!!}x^{2n}\right), \tag{6.4}$$

mit

$$x = \frac{|\mathbf{R}_1-\mathbf{R}_2|}{2R}. \tag{6.5}$$

Die gerade Potenzen x^{2n} können in Tensorform durch

$$x^{2n} = \frac{1}{2^n}\sum_{j=0}^{n}\frac{1}{R^{2j}}(-1)^j\binom{n}{j}R_1^{i_1}\cdots R_j^{i_j}\ R_2^{i_1}\cdots R_2^{i_j} \tag{6.6}$$

ausgedrückt werden.

Damit lässt sich der zweite Term in Gl. (6.2) entkoppeln,

$$\begin{aligned}&\int d^2R_1 \int d^2R_2 \; \delta(R-R_1)\delta(R-R_2) \; \rho(\mathbf{r}+\mathbf{R}_1)\rho(\mathbf{r}+\mathbf{R}_2) \; x^{2n} \\ =&\frac{1}{2^n}\sum_{j=0}^{n}\frac{1}{R^{2j}}(-1)^j\binom{n}{j}n_T^{(j)}(\mathbf{r})\circ n_T^{(j)}(\mathbf{r})\end{aligned} \tag{6.7}$$

und mit den gewichteten Tensordichten

$$n_T^{(j)}(\mathbf{r}) = \rho(\mathbf{r})\cdot\omega_T^{(j)}(\mathbf{r}) \tag{6.8}$$

$$\text{und} \quad \omega_T^{(j)}(\mathbf{r}) = \delta\left(R-|\mathbf{r}|\right)\hat{r}^{i_1}\cdots\hat{r}^{i_j} \tag{6.9}$$

ausdrücken. Wir erhalten

$$\begin{aligned}\Phi_2 = \frac{1}{1-n_2}\frac{1}{\pi}\Bigg(&\frac{\pi}{8}n_1^2-\frac{1}{4}\mathbf{n}_1^2 \\ &+\sum_{n=1}^{\infty}\left(-\frac{\pi}{8}\frac{(2n-3)!!}{(2n)!!}n_T^{(2n)}\circ n_T^{(2n)}+\frac{1}{4}\frac{(2n-2)!!}{(2n+1)!!}n_T^{(2n+1)}\circ n_T^{(2n+1)}\right)\Bigg).\end{aligned} \tag{6.10}$$

Durch Gl. (6.10) erhalten wir Korrekturen an Rosenfelds ursprüngliches Funktional. Zur praktischen Berechnung ist es sinnvoll, sich auf eine endliche Zahl von Korrekturtermen zu beschränken. Wir brechen die Reihe bei $n = 2$ ab, d.h. nur die Korrektur in erster Ordnung wird betrachtet.

Dabei ist zu beachten, dass der Abbruch der Reihe einer polynomiellen Approximation $P_2^a = b_1 + b_2x^2 + b_3x^4$ entspricht, bei der das Integral der Approximationsfunktion i.d.R. nicht mit dem Integral über die zu approximierende Funktion übereinstimmt, d.h. i.d.R. ist

$$\int d\mathbf{r} \; P_2^a \neq \int d\mathbf{r} \; P_2. \tag{6.11}$$

Für Gl. (6.3), ausgewertet bei $\rho = \text{const.}$ gilt

$$\int_0^1 dx \frac{1}{\sqrt{1-x^2}} P_2(x) = \frac{1}{8}, \tag{6.12}$$

was wir polynomiell mit

$$P_2^a \approx \frac{1}{\pi}\left(ax^2 + \frac{2-4a}{3}x^4\right) \tag{6.13}$$

nähern können, wobei a ein freier Parameter ist. In Kombination mit Gl. (6.10) in erster Ordnung führt uns dies auf

$$\Phi_2 \approx \frac{1}{1-n_2}\frac{1}{\pi}\left(\frac{1+a}{6}n_1^2 + \frac{a-2}{6}\mathbf{n}_1^2 + \frac{1-2a}{6}n_T^{(2)} \circ n_T^{(2)}\right). \tag{6.14}$$

Den freien Parameter a können wir nutzen, um die Ungleichheit in Gl. (6.11) zu beseitigen. Er lässt sich fixieren, indem man die die Abweichung des genäherten Integrals vom echten Integral, d.h. die Funktion

$$\int dx \frac{1}{\sqrt{1-x^2}} \left(P_2 - P_2^a\right)^2, \tag{6.15}$$

minimiert, was zu $a = \frac{11}{8}$ führt. Damit erhalten wir

$$F_{ex} = \int d\mathbf{r} \left(-n_0 \ln(1-n_2) + \frac{1}{1-n_2}\frac{1}{\pi}\left(\frac{19}{48}n_1^2 - \frac{5}{48}\mathbf{n}_1^2 - \frac{14}{48}n_T^{(2)} \circ n_T^{(2)}\right)\right). \tag{6.16}$$

als Tensorfunktional.

Für $a = \frac{1}{2}$ wird der Vorfaktor $\frac{1-2a}{6}$ des Tensorterms in Gl. (6.14) Null und die Faktoren vor dem Skalar- und Vektoranteil produzieren das bekannte Rosenfeld-Funktional. Dieses ist demnach als Spezialfall im Tensorfunktional erhalten.

6.2. Vergleich der Dichtefunktionen

Dichteverteilung

Mit diesem neuen Funktional können wir nun analog zu Kapitel 5.1 die Dichteverteilung bestimmen.

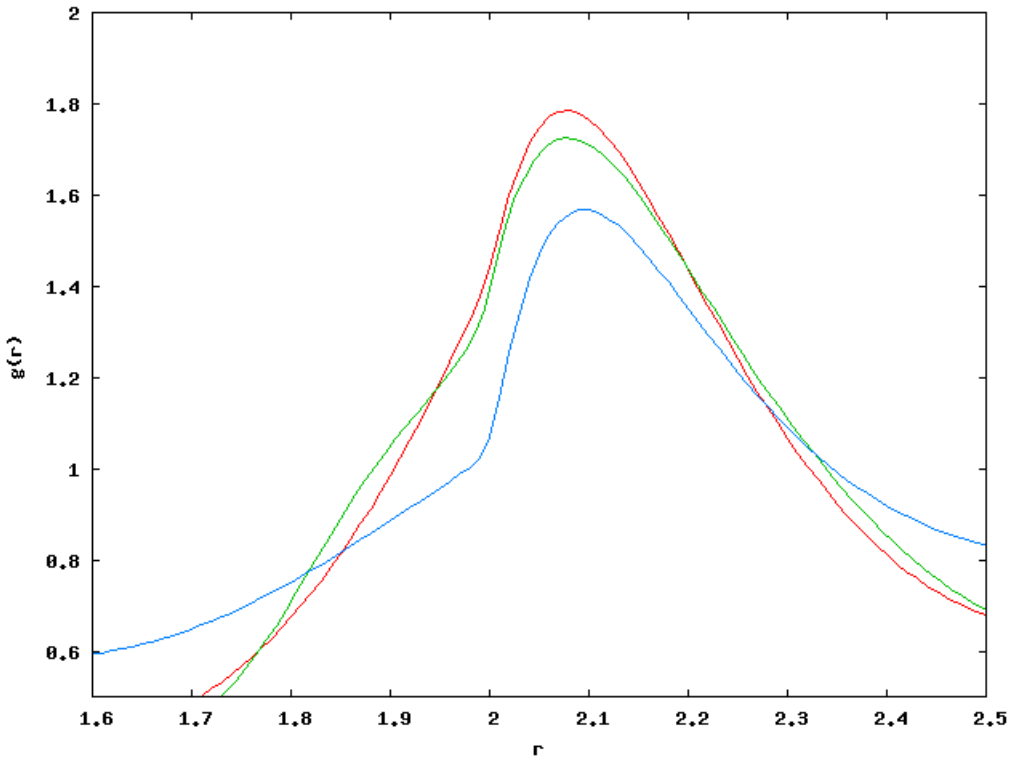

Abbildung 6.2.: Vergleich der Dichteverteilung zwischen Tensorfunktional (rot), MD-Simulation (grün) und Rosenfeld-Funktional (blau) bei einer Dichte von $\rho = 0.853$.

In Abb. 6.2 sehen wir, dass der "Knick" um $X = 2$ im Tensorfunktional weniger stark ausgeprägt ist als bei Rosenfeld. Dies steht in sehr guter Übereinstimmung mit den Simulationsdaten. Das Tensorfunktional erziehlt also eine deutliche Verbesserung, sogar für einfache Dichteverteilungen.

Drei-Teilchen-Korrelation

Analog zu Kapitel 5.2 lässt sich die Drei-Teilchen-Korrelation bestimmen.

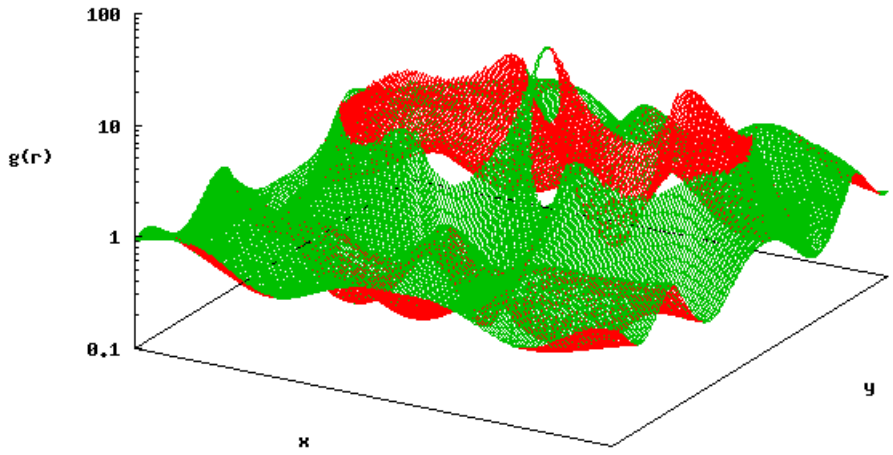

Abbildung 6.3.: Dichteverteilung bei $d = 2r$ und $\rho = 0.828$.

Optisch lässt sich in Abb. 6.3 und Abb. 6.4 nur schwer ein Unterschied zu den Ergebnissen des Rosenfeld-Funktionals in Abb. 5.6 und Abb. 5.7 feststellen, die Nutzung dieser Daten zur Kräfteberechnung hingegen ist aufschlussreich.

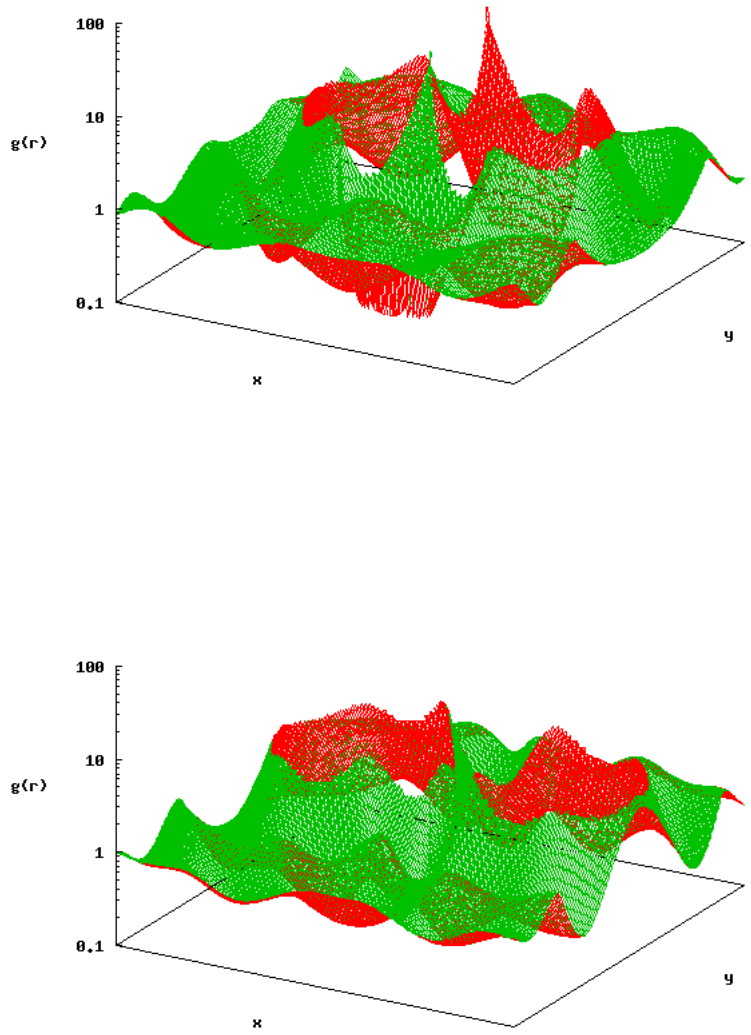

Abbildung 6.4.: Dichteverteilung für verschiedene Teilchenabstände bei $\rho = 0.828$ mit $d < 2r$ (oben) und $d > 2r$ (unten).

Kräftebetrachtung

Wie in Kapitel 5.3 bestimmen wir aus der Drei-Teilchen-Korrelation auch die Kräfte auf die Teilchen.

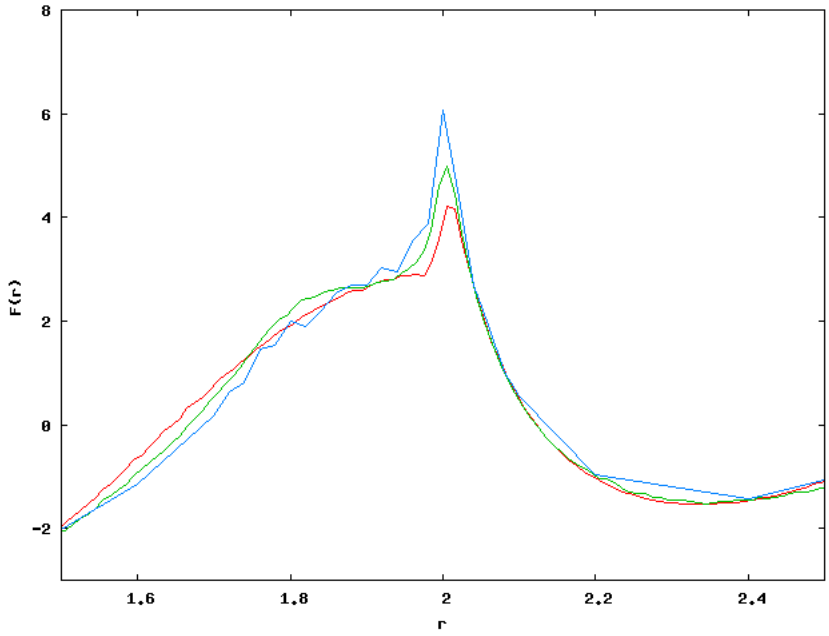

Abbildung 6.5.: Kräfte zwischen Teilchen bei niedriger Dichte $\rho = 0.764$, berechnet mit dem Tensorfunktional und $a = \frac{11}{8}$: Ein fixiertes Testteilchen (rot), MD-Simulation (grün) und zwei fixierte Testteilchen (blau).

Die resultierende Kraftkurve ist in Fig. 6.5 zu sehen. Es wurde eine deutlich bessere Übereinstimmung mit den Simulationsdaten erziehlt als mit dem Rosenfeld-Funktional.

Dabei ist zu beachten, dass die Dichte in Fig. 6.5 noch deutlich unter dem Gefrierpunkt liegt. Eine schrittweise Erhöhung der Dichte erlaubt es uns, diesen Übergang genauer zu untersuchen.

6.3. Der Gefrierübergang

Bei der Annäherung an den Gefrierübergang gibt es zwei numerische Probleme bei der Lösung der Integralgleichung. Einerseits gibt es in Gl. (6.14) einen Vorfaktor $\frac{1}{1-n_2}$, der für $n_2 \to 1$ divergiert. Andererseits werden bei einer Fixierung von zwei Testteilchen gerade im Bereich der Kontaktstelle um $d \approx 2r$ (Distanz der Scheiben entspricht dem doppelten Ausschlussradius) die numerischen Werte sehr groß, so dass selbst kleine Unterschiede in der Lage der Gitterpunkte große Unterschiede bewirkt, was wiederum die Fourier-Transformation erschwert.

Der Term $\frac{1}{1-n_2}$

Wie in Gl. (5.8) definiert ist n_2 durch

$$n_2 = \int d\mathbf{r}' \; \rho(\mathbf{r}') \cdot \omega_2(\mathbf{r} - \mathbf{r}'), \quad \omega_2 = \Theta(R - r), \quad r = |\mathbf{r}| \tag{6.17}$$

gegeben, wobei R den Radius der Scheiben darstellt. n_2 liegt im Intervall $[0:1)$, kann jedoch gerade für große Dichten sehr nahe bei 1 liegen. Dies führt dazu, dass $\frac{1}{1-n_2}$ sehr groß wird, was wiederum zu großen Schwankungen in der berechneten Dichte bei nur kleinen Änderungen von n_2 führt. Diese Schwankungen können so groß ausfallen, dass die Iteration der Integralgleichung bis zur Selbstkonsistenz bei hohen Dichten nicht mehr zuverlässig durchgeführt werden kann. Dieses Problem kann minimiert werden, indem die Dichte schrittweise erhöht wird, so dass n_2 für die jeweils betroffenen Gitterpunkte nur schrittweise und langsam anwächst. Jedoch stößt auch diese Methode bei sehr hohen Dichten an ihre Grenze, weshalb es eine obere Grenze der Dichte gibt, die mit der hier gewählten Methodik berechnet werden kann. Ab $n_2 = 0.999$ sind die numerischen Schwankungen zu groß, so dass die fortlaufende Fourier-Transformation nicht mehr ausgeführt werden kann. Dies tritt ab $\rho \approx 0.9$ auf. Hier spielt die Beschränkung der Grafikkarte auf einfache Genauigkeit eine Rolle, bei doppelter Genauigkeit können Werte noch näher an 1 weiterhin zuverlässig unterschieden werden, was zu einer Erhöhung der maximal berechenbaren Dichte führt. Bevor dieses Problem eintritt, sehen wir jedoch den Einfluss des Gitters.

Das Gitter am Kontaktpunkt der Scheiben

Da Berechnungen auf einem Gitter stattfinden wird eine diskrete Fourier-Transformation genutzt. In den Bereichen, in denen n_2 nahe 1 liegt, d.h. in den Bereichen mit sehr großen numerischen Werten die fouriertransformiert werden, können sich die einzelnen Gitterpunkte sehr stark unterscheiden. Insbesondere am Kontaktpunkt der zwei fixierten Scheiben gibt es sehr hohe Dichten, wie wir in Abb. 6.3 gesehen haben. Wenn ein Gitterpunkt genau am Punkt der höchsten Dichte liegt dann wird sich dieser Wert sehr stark von demjenigen unterscheiden, den wir sehen, wenn der Punkt der höchsten Dichte zwischen die Gitterpunkte fällt. Die Position des Gitters verfälscht somit in diesem Bereich die Ergebnisse.

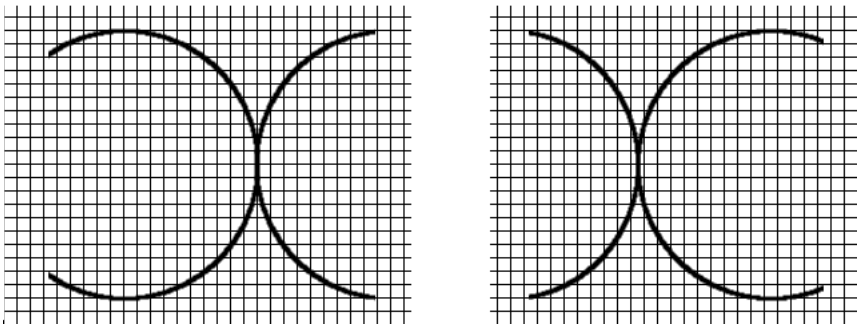

Abbildung 6.6.: Mögliche Gitterpositionen um die Kontaktstelle der Scheiben. Liegt die Kontaktstelle genau auf einer Gitterreihe (links) so wird eine andere Dichte berechnet als wenn die Kontaktstelle zwischen Gitterpunkte fällt (rechts).

Um diesen Effekt zu minimieren ist es am einfachsten, die Gitterauflösung zu erhöhen. Deshalb wurden für die Berechnungen in diesem Kapitel Grafikkarten mit 6 GB Speicher eingesetzt, so dass Gittergrößen von 4096^2 möglich wurden. Zudem wurde eine 4-Punkt-Interpolation zur Bestimmung der Kraftkurve genutzt, um die Dichte exakt an der Teilchenoberfläche mit hoher Genauigkeit bestimmen zu können, so dass die Position der Gitterpunkte einen möglichst geringen Einfluss hat. Desweiteren wurden für einen Teilchenabstand mehrere Rechenläufe durchgeführt, bei denen das Gitter jeweils leicht verschoben wurde, d.h. die Position der fixierten Scheiben auf dem Gitter wurde um einen Betrag kleiner als der Gitterabstand variiert und die Meßwerte stellen den Mittelwert der verschiedenen Läufe dar.

Ergebnisse nahe des Gefrierübergangs

Diese Maßnahmen erlauben es uns, Dichteprofile nahe des Gefrierübergangs zu generieren und daraus Kräfteprofile zu bestimmen.

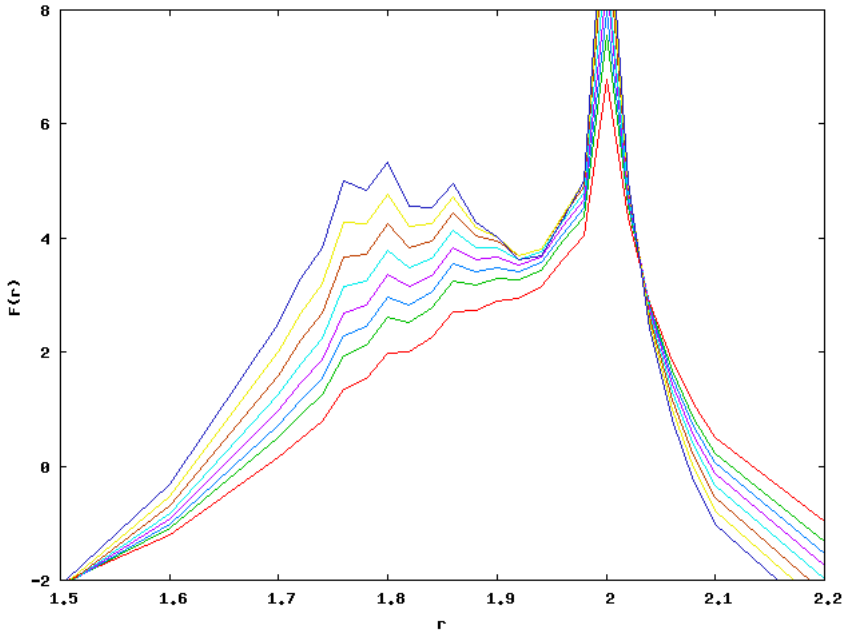

Abbildung 6.7.: Kraft auf eine Scheibe bei $\eta = 0.60$ (rot), 0.62 (grün), 0.63 (blau), 0.64 (lila), 0.65 (hellblau), 0.66 (braun), 0.67 (gelb) und 0.68 (dunkelblau), berechnet mit Hilfe zweier fixierter Testteilchen.

In Abb. 6.7 sehen wir solche Profile für diverse Dichten. Es ist deutlich zu sehen, dass sich bei einer Erhöhung der Dichte ein „Buckel“ um den Teilchenabstand von $\approx 1.8r$ bildet, was die Kristallstruktur und somit den Gefrierübergang einleitet. Dieser „Buckel“ ist deutlich stärker ausgeprägt als bei Berechnungen mit Hilfe des Rosenfeld-Funktionals. Dabei ist zu beachten, dass der Übergang mit dem Tensorfunktional später zu erkennen ist als in MD-Simulationen, wie wir in Abb. 6.8 sehen können. Die “Zacken” in Abb. 6.7 zeigen die maximale Genauigkeit des Gitters und der hier genutzten Iterationsmethode.

Dieser Unterschied ist durch die bei der Herleitung des Tensorfunktionals genutzten Näherungen zu begründen. Auch wenn der exakte Punkt des Gefrierübergangs nicht reproduziert werden kann so ist es dennoch beachtlich, dass das Tensorfunk-

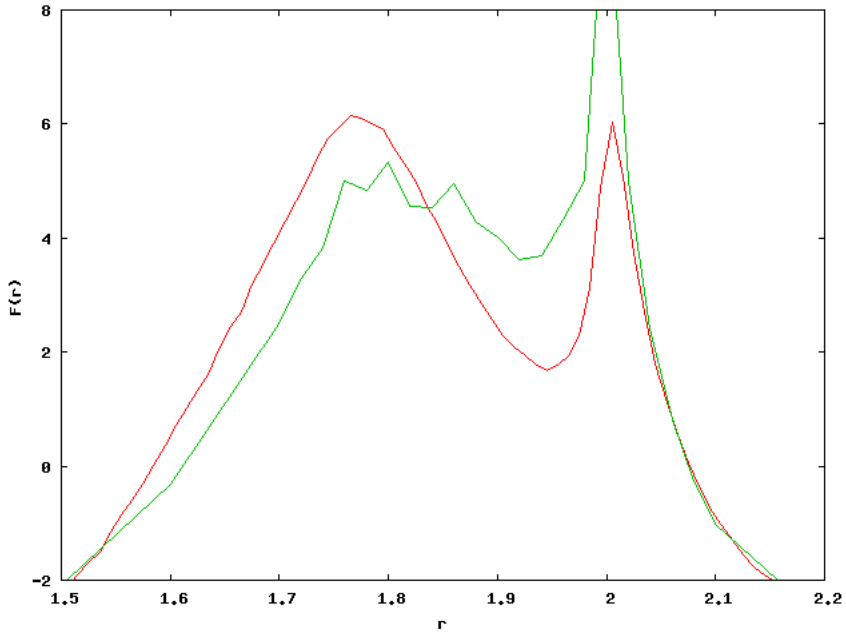

Abbildung 6.8.: Vergleich der mit Hilfe des Tensorfunktionals generierten Kraftkurve mit einer MD-Simulation bei $\eta = 0.68$.

tional ihn zumindest qualitativ abzubilden vermag. Der Funktionswert bei $r = 2$, d.h. bei Kontakt der Scheiben, wird vom Tensorfunktional falsch dargestellt. Dies ist verständlich, da an diesem Punkt die numerischen Werte maximal werden, so dass die eingangs erwähnten numerischen Probleme bei der Berechnung des Gitters deutlich auftreten. Eine weitere Erhöhung der Dichte ist aufgrund dieser numerischen Probleme leider nicht möglich. So kann nur der Ansatz des Gefrierübergangs beobachtet werden, nicht der Übergang selbst.

7. Fazit

Das Ziel dieser Arbeit war zweigeteilt. Einerseits galt es, eine neue Möglichkeit zur Berechnung physikalischer Systeme zu testen und die Zuverlässigkeit der auf einer Grafikkarte erzielten Ergebnisse zu prüfen. Andererseits wurde ein physikalisches System untersucht, das bisher in der Literatur keine große Aufmerksamkeit erfahren hat und das numerisch schwer zu fassen ist.

Der Einsatz der Grafikkarte kann als Erfolg betrachtet werden. Die erzielten Geschwindigkeitsgewinne gegenüber der CPU sind sehr deutlich. In Kombination mit dem sehr günstigen Anschaffungspreis sowie der geringen Strom- und Kühlkosten im Vergleich zu CPU-basierten Clusterrechnern ist die GPU-Leistung mehr als hoch genug, so dass sich der zusätzliche Programmieraufwand leicht rechtfertigen lässt. Wenn ein Problem rechenintensiv und parallelisierbar ist und nicht im Konflikt mit den Beschränkungen (insbesondere dem limitierten Grafikspeicher) der Grafikkarte steht so ist der Einsatz einer GPU zu empfehlen. In dieser Arbeit konnten keine Speicher- oder Rechenfehler festgestellt werden. CUDA ist zudem mittlerweile ausgereift genug, um zuverlässig in wissenschaftlichen Projekten eingesetzt zu werden. Im Rahmen dieser Arbeit wurden jedoch auch weitere Möglichkeiten der Grafikkartenprogrammierung erkundet, beispielsweise der CUDA-Fortran-Compiler, welcher sich als nicht praxistauglich und sehr fehlerhaft erwiesen hat. Trotz der guten Ergebnisse dieser Arbeit ist deshalb zu beachten, dass die Grafikkartenprogrammierung erst seit wenigen Jahren möglich ist und noch in den Kinderschuhen steckt. Obwohl die Basis, CUDA selbst, stabil und zuverlässig arbeitet, kann dies noch nicht für viele periphere Programme gesagt werden. Die Umstellung existierender großer CPU-basierter Projekte, insbesondere wenn sie nicht vollständig in C geschrieben sein sollten, gestaltet sich deshalb noch immer sehr schwierig.

Bei der Berechnung der DFT auf der GPU konnte dieses Problem umgangen werden, indem von Grund auf neuer Code auf der Grafikkarte geschrieben wur-

de. Dies war sehr erfolgreich, die Ergebnisse der bekanntesten Funktionale aus der Literatur, das Taylor-Funktional und das Rosenfeld-Funktional, konnten effizient reproduziert werden. Zudem ist es gelungen, den bisher nicht im Detail betrachteten Gefrierübergang in zwei Dimensionen näher zu untersuchen. Die Grenzen der bekannten Funktionale wurden aufgezeigt und es konnte ein (Tensor-)Funktional eingeführt werden, das den Gefrierübergang einer Flüssigkeit harter Scheiben in zwei Dimensionen abbildet und den bekannten Funktionalen überlegen ist. Dabei wurden zudem die numerischen Grenzen der Berechnung auf der Grafikkarte aufgezeigt.

Seit Beginn dieser Arbeit fand die GPU-Programmierung zunehmende Akzeptanz. Es ist anzunehmen, dass weiterhin neue Einsatzgebiete für Grafikkarten gefunden und genutzt werden. Die hier vorgestellten Methoden der DFT auf Grafikkarten bilden einen Grundstein für zukünftige Berechnungen auf diesem Gebiet und lassen sich leicht auch andere Dimensionen oder andere Funktionale erweitern.

Basierend auf den mit Hilfe des Tensorfunktionals gewonnenen Daten sind weitere Arbeiten zur Untersuchung des Gefrierübergangs in der DFT denkbar. Insbesondere eine weitere Annäherung an den Übergang, beispielsweise durch die Nutzung einer logarithmischen Skala oder eines sehr hoch auflösenden Gitters (welches dann jedoch aufgrund der Speicherbeschränkungen nicht mehr auf der GPU berechnet werden kann) ist möglich.

A. Anhang

A.1. Quellcode

A.1.1. GPU Tensor DFT

```
/**
 *
 * DFT using CUDA
 *
 */

//Headers
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cutil.h>

#include <cufft.h>
#include <cublas.h>
#include <cutil_inline.h>

#include "writer.c"
#include "omega_fu.c"

#define BLOCK_SIZE 16
#define BLOCK_LENGTH 256
#define MAX_BLOCK 512

const float INF = 1e+7;
const float a = 1.0/2.0;

int size_x = 1024;
int size_y = 1024;
float dx = 0.012;

const float radius = 0.5;
float rho0 = 0.60 / (4.0*radius*radius);
float min_mixing = 0.0001;
float max_mixing = 0.02;
float mixing = min_mixing;

float epsilon = 0;
float epsilon_old = 0;

float precision = 2.5e-08;

cuComplex* h_rho;
cuComplex* h_v;
```

```
cuComplex* h_tmp;

cuComplex* h_w1;
cuComplex* h_w2;
cuComplex* h_wvx;
cuComplex* h_wvy;
cuComplex* h_wTxx;
cuComplex* h_wTyy;
cuComplex* h_wTxy;

cuComplex* h_nadd;
cuComplex* h_padd;

__device__ cuComplex* d_rho;
__device__ cuComplex* d_deltarho;
__device__ cuComplex* d_v;
__device__ cuComplex* d_tmp;
__device__ cuComplex* d_force;

__device__ cuComplex* d_n1;
__device__ cuComplex* d_n2;
__device__ cuComplex* d_nvx;
__device__ cuComplex* d_nvy;
__device__ cuComplex* d_nTxx;
__device__ cuComplex* d_nTyy;
__device__ cuComplex* d_nTxy;

__device__ cuComplex* d_w1;
__device__ cuComplex* d_w2;
__device__ cuComplex* d_wvx;
__device__ cuComplex* d_wvy;
__device__ cuComplex* d_wTxx;
__device__ cuComplex* d_wTyy;
__device__ cuComplex* d_wTxy;

__device__ cuComplex* d_p1;
__device__ cuComplex* d_p2;
__device__ cuComplex* d_pvx;
__device__ cuComplex* d_pvy;
__device__ cuComplex* d_pTxx;
__device__ cuComplex* d_pTyy;
__device__ cuComplex* d_pTxy;

__device__ cuComplex* d_nadd;
__device__ cuComplex* d_padd;

unsigned int t1 = 0;
unsigned int t2 = 0;
unsigned int t3 = 0;

__device__ cufftHandle plan;

bool running = true;
bool flag = true;
int loops = 0;
const int max_loops = 3000;

cuComplex ij_to_r(int x, int y)
{
    cuComplex r;
    if (x >= size_x/2 && y < size_y/2)
    {
        r.x = -(size_x-x);
        r.y = y;
    }
    else if (x < size_x/2 && y >= size_y/2)
    {
```

```
        r.x = x;
        r.y = -(size_y-y);
    }
    else if (x >= size_x/2 && y >= size_y/2)
    {
        r.x = -(size_x-x);
        r.y = -(size_y-y);
    }
    else
    {
        r.x = x;
        r.y = y;
    }
    r.x *= dx;
    r.y *= dx;
    return r;
}

/*cuComplex ij_to_r(int x, int y)
{
    cuComplex r;
    r.x = dx * (float) (x - size_x/2);
    r.y = dx * (float) (y - size_y/2);
    return r;
}*/

float absolute(cuComplex x)
{
    return sqrt(x.x*x.x + x.y*x.y);
}

/*float V(cuComplex r, float dr)
{
   if (absolute(r) > 2*radius)
        return 0;
    else
        return INF;
}*/

float V(cuComplex r, float dr, float disp)
{
    cuComplex tmp; tmp.x = r.x; tmp.y = r.y;
    if (tmp.x > 0) tmp.x -= dr;
    else if (tmp.x <= 0) tmp.x += dr;
    tmp.x -= disp;
    if (absolute(tmp) > 2*radius)
        return 0;
    else
        return INF;
}

__global__ void get_delta(cuComplex* drho, cuComplex* rho, float rho0, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        drho[pos].x = rho[pos].x - rho0;
        drho[pos].y = 0.0;
        __syncthreads();
    }
}

__global__ void pointmul(cuComplex* a, cuComplex* b, cuComplex* c, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        a[pos].x = b[pos].x * c[pos].x - b[pos].y * c[pos].y;
        a[pos].y = b[pos].y * c[pos].x + b[pos].x * c[pos].y;
        __syncthreads();
    }
}

__global__ void mix(cuComplex* res, cuComplex* a, cuComplex* b, float mix, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        res[pos].x = (1.0-mix) * a[pos].x + mix * b[pos].x;
        res[pos].y = 0.0;
        __syncthreads();
    }
}

__global__ void function(cuComplex* res, cuComplex* p1, cuComplex* p2, cuComplex* pvx, cuComplex
    * pvy, cuComplex* pTxx, cuComplex* pTyy, cuComplex* pTxy, cuComplex* v, float r, int sx, int
     sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        res[pos].x = r * exp(- v[pos].x - p1[pos].x - p2[pos].x + pvx[pos].x + pvy[pos].x - pTxx
            [pos].x - pTyy[pos].x - 2*pTxy[pos].x);
        res[pos].y = 0.0;
        __syncthreads();
    }
}

__global__ void force(cuComplex* res, cuComplex* p1, cuComplex* p2, cuComplex* pvx, cuComplex*
    pvy, cuComplex* pTxx, cuComplex* pTyy, cuComplex* pTxy, float r, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        res[pos].x = r * exp(- p1[pos].x - p2[pos].x + pvx[pos].x + pvy[pos].x - pTxx[pos].x -
            pTyy[pos].x - 2*pTxy[pos].x);
        res[pos].y = 0.0;
        __syncthreads();
    }
}

__global__ void compare_gpu_step1(cuComplex* a, cuComplex* b, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        a[pos].x = (a[pos].x - b[pos].x) * (a[pos].x - b[pos].x) / (sx*sy);
    }
}

__global__ void compare_gpu_step2(cuComplex* a, int s, int sx, int sy)
{
```

```
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        if (pos < s)
            {
                a[pos].x += a[pos + s].x;
                a[pos].y = 0.0;
            }
        }
}

void compare(cuComplex* a, cuComplex* b, int sx, int sy, dim3 dG, dim3 dB)
{
    compare_gpu_step1<<<dG,dB>>>(a, b, sx, sy);
    for(int s=sx*sy/2; s>0; s/=2)
        compare_gpu_step2<<<dG,dB>>>(a, s, sx, sy);
}

__global__ void compare_fast(cuComplex* a, cuComplex* b, int sx, int sy, float dx, float r)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (pos < MAX_BLOCK)
    {
        __shared__ float c[MAX_BLOCK];
        c[pos] = a[pos + (int) (r/dx)].x - b[pos + (int) (r/dx)].x;
        c[pos] *= c[pos];
        c[pos] /= (float) MAX_BLOCK;
        __syncthreads();
        for (int s=MAX_BLOCK/2; s>0; s/=2)
        {
            if (pos < s)
            {
                c[pos] += c[pos + s];
            }
            __syncthreads();
        }
        if (pos == 0)
            {
                a[0].x = c[0];
                a[0].y = 0;
            }
    }
}

void g_calc_n(cuComplex* n1, cuComplex* n2, cuComplex* nvx, cuComplex* nvy, cuComplex* nTxx,
     cuComplex* nTyy, cuComplex* nTxy, cuComplex* nadd, cuComplex* w1, cuComplex* w2, cuComplex*
     wvx, cuComplex* wvy, cuComplex* wTxx, cuComplex* wTyy, cuComplex* wTxy, cuComplex* tmp,
     cuComplex* rho, int sx, int sy, dim3 dimBlock, dim3 dimGrid)
{
    float size = sx*sy;
    float A = dx * sx * dx * sy;
    cuComplex temp2;
    temp2.x = A / size;
    temp2.y = 0.0;
    cuComplex temp3;
    temp3.x = 1.0 / A;
    temp3.y = 0.0;

    cufftSafeCall(cufftExecC2C(plan, rho, tmp, CUFFT_FORWARD));
    cublasCscal(size, temp2, tmp, 1);

    pointmul<<<dimGrid,dimBlock>>>(n1, w1, tmp, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(n2, w2, tmp, sx, sy);
```

```
    pointmul<<<dimGrid,dimBlock>>>(nvx, wvx, tmp, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(nvy, wvy, tmp, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(nTxx, wTxx, tmp, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(nTyy, wTyy, tmp, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(nTxy, wTxy, tmp, sx, sy);

    cufftSafeCall(cufftExecC2C(plan, n1, n1, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, n2, n2, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, nvx, nvx, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, nvy, nvy, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, nTxx, nTxx, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, nTyy, nTyy, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, nTxy, nTxy, CUFFT_INVERSE));
    cublasCscal(size, temp3, n1, 1);
    cublasCscal(size, temp3, n2, 1);
    cublasCscal(size, temp3, nvx, 1);
    cublasCscal(size, temp3, nvy, 1);
    cublasCscal(size, temp3, nTxx, 1);
    cublasCscal(size, temp3, nTyy, 1);
    cublasCscal(size, temp3, nTxy, 1);

    cuComplex temp4; temp4.x = 1.0; temp4.y = 1.0;
    cublasCaxpy(size, temp4, &nadd[0], 0, n1, 1);
    cublasCaxpy(size, temp4, &nadd[1], 0, n2, 1);
    cublasCaxpy(size, temp4, &nadd[2], 0, nvx, 1);
    cublasCaxpy(size, temp4, &nadd[2], 0, nvy, 1);
    cublasCaxpy(size, temp4, &nadd[3], 0, nTxx, 1);
    cublasCaxpy(size, temp4, &nadd[3], 0, nTyy, 1);
    cublasCaxpy(size, temp4, &nadd[2], 0, nTxy, 1);
}

__global__ void calc_p1(cuComplex* p, cuComplex* n1, cuComplex* n2, cuComplex* nvx, cuComplex*
    nvy, cuComplex* nTxx, cuComplex* nTyy, cuComplex* nTxy, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        if (n2[pos].x > 0.9999) n2[pos].x = 0.9999;

        p[pos].x = ( 1.0 / (1.0 - n2[pos].x) ) * (1.0/M_PI) * ((1.0+a)/3.0) * n1[pos].x - log(1-
            n2[pos].x) / (2*radius*M_PI);
        p[pos].y = ( 1.0 / (1.0 - n2[pos].y) ) * (1.0/M_PI) * ((1.0+a)/3.0) * n1[pos].y - log(1-
            n2[pos].y) / (2*radius*M_PI);
        __syncthreads();
    }
}

__global__ void calc_p2(cuComplex* p, cuComplex* n1, cuComplex* n2, cuComplex* nvx, cuComplex*
    nvy, cuComplex* nTxx, cuComplex* nTyy, cuComplex* nTxy, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        p[pos].x = ( 1.0 / ( (1.0 - n2[pos].x)*(1.0 - n2[pos].x) ) ) * (1.0/M_PI) * ( ((1.0+a)
            /6.0)*n1[pos].x*n1[pos].x + ((a-2.0)/6.0)*(nvx[pos].x*nvx[pos].x + nvy[pos].x*nvy[
            pos].x) + ((1.0-2*a)/6.0)*(nTxx[pos].x*nTxx[pos].x + nTyy[pos].x*nTyy[pos].x + 2*
            nTxy[pos].x*nTxy[pos].x) ) + n1[pos].x / ( 2*radius*M_PI*(1-n2[pos].x) );
        p[pos].y = ( 1.0 / ( (1.0 - n2[pos].y)*(1.0 - n2[pos].y) ) ) * (1.0/M_PI) * ( ((1.0+a)
            /6.0)*n1[pos].y*n1[pos].y + ((a-2.0)/6.0)*(nvx[pos].y*nvx[pos].y + nvy[pos].y*nvy[
            pos].y) + ((1.0-2*a)/6.0)*(nTxx[pos].y*nTxx[pos].y + nTyy[pos].y*nTyy[pos].y + 2*
            nTxy[pos].y*nTxy[pos].y) ) + n1[pos].y / ( 2*radius*M_PI*(1-n2[pos].y) );
        __syncthreads();
    }
```

```
}

__global__ void calc_pv(cuComplex* px, cuComplex* py, cuComplex* n1, cuComplex* n2, cuComplex*
    nvx, cuComplex* nvy, cuComplex* nTxx, cuComplex* nTyy, cuComplex* nTxy, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        px[pos].x = + nvx[pos].x * ( 1.0 / (1.0 - n2[pos].x) ) * (1/M_PI) * ((a-2.0)/3.0);
        px[pos].y = + nvx[pos].y * ( 1.0 / (1.0 - n2[pos].y) ) * (1/M_PI) * ((a-2.0)/3.0);

        py[pos].x = + nvy[pos].x * ( 1.0 / (1.0 - n2[pos].x) ) * (1/M_PI) * ((a-2.0)/3.0);
        py[pos].y = + nvy[pos].y * ( 1.0 / (1.0 - n2[pos].y) ) * (1/M_PI) * ((a-2.0)/3.0);

        __syncthreads();
    }
}

__global__ void calc_pT(cuComplex* pTxx, cuComplex* pTyy, cuComplex* pTxy, cuComplex* n1,
    cuComplex* n2, cuComplex* nvx, cuComplex* nvy, cuComplex* nTxx, cuComplex* nTyy, cuComplex*
    nTxy, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        pTxx[pos].x = + nTxx[pos].x * ( 1.0 / (1.0 - n2[pos].x) ) * (1.0/M_PI) * ((1.0-2*a)/3.0)
            ;
        pTxx[pos].y = + nTxx[pos].y * ( 1.0 / (1.0 - n2[pos].y) ) * (1.0/M_PI) * ((1.0-2*a)/3.0)
            ;

        pTyy[pos].x = + nTyy[pos].x * ( 1.0 / (1.0 - n2[pos].x) ) * (1.0/M_PI) * ((1.0-2*a)/3.0)
            ;
        pTyy[pos].y = + nTyy[pos].y * ( 1.0 / (1.0 - n2[pos].y) ) * (1.0/M_PI) * ((1.0-2*a)/3.0)
            ;

        pTxy[pos].x = + nTxy[pos].x * ( 1.0 / (1.0 - n2[pos].x) ) * (1.0/M_PI) * ((1.0-2*a)/3.0)
            ;
        pTxy[pos].y = + nTxy[pos].y * ( 1.0 / (1.0 - n2[pos].y) ) * (1.0/M_PI) * ((1.0-2*a)/3.0)
            ;

        __syncthreads();
    }
}

void g_calc_phi(cuComplex* p1, cuComplex* p2, cuComplex* pvx, cuComplex* pvy, cuComplex* pTxx,
    cuComplex* pTyy, cuComplex* pTxy, cuComplex* padd, cuComplex* n1, cuComplex* n2, cuComplex*
    nvx, cuComplex* nvy, cuComplex* nTxx, cuComplex* nTyy, cuComplex* nTxy, int sx, int sy, dim3
     dimBlock, dim3 dimGrid)
{
    calc_p1<<<dimGrid,dimBlock>>>(p1, n1, n2, nvx, nvy, nTxx, nTyy, nTxy, sx, sy);
    calc_p2<<<dimGrid,dimBlock>>>(p2, n1, n2, nvx, nvy, nTxx, nTyy, nTxy, sx, sy);
    calc_pv<<<dimGrid,dimBlock>>>(pvx, pvy, n1, n2, nvx, nvy, nTxx, nTyy, nTxy, sx, sy);
    calc_pT<<<dimGrid,dimBlock>>>(pTxx, pTyy, pTxy, n1, n2, nvx, nvy, nTxx, nTyy, nTxy, sx, sy);

    float size = sx*sy;
    cuComplex alpha; alpha.x = 1.0; alpha.y = 1.0;
    cublasCaxpy(size, alpha, &padd[0], 0, p1, 1);
    cublasCaxpy(size, alpha, &padd[1], 0, p2, 1);
    cublasCaxpy(size, alpha, &padd[2], 0, pvx, 1);
    cublasCaxpy(size, alpha, &padd[2], 0, pvy, 1);
    cublasCaxpy(size, alpha, &padd[3], 0, pTxx, 1);
    cublasCaxpy(size, alpha, &padd[3], 0, pTyy, 1);
    cublasCaxpy(size, alpha, &padd[2], 0, pTxy, 1);
```

```
}

void g_calc_conv(cuComplex* r1, cuComplex* r2, cuComplex* rvx, cuComplex* rvy, cuComplex* rTxx,
     cuComplex* rTyy, cuComplex* rTxy, cuComplex* p1, cuComplex* p2, cuComplex* pvx, cuComplex*
     pvy, cuComplex* pTxx, cuComplex* pTyy, cuComplex* pTxy, cuComplex* w1, cuComplex* w2,
     cuComplex* wvx, cuComplex* wvy, cuComplex* wTxx, cuComplex* wTyy, cuComplex* wTxy, int sx,
     int sy, dim3 dimBlock, dim3 dimGrid)
{
    float size = sx*sy;
    float A = dx * sx * dx * sy;
    cuComplex temp2;
    temp2.x = A / size;
    temp2.y = 0;
    cuComplex temp3;
    temp3.x = 1.0 / A;
    temp3.y = 0;

    cufftSafeCall(cufftExecC2C(plan, p1, p1, CUFFT_FORWARD));
    cufftSafeCall(cufftExecC2C(plan, p2, p2, CUFFT_FORWARD));
    cufftSafeCall(cufftExecC2C(plan, pvx, pvx, CUFFT_FORWARD));
    cufftSafeCall(cufftExecC2C(plan, pvy, pvy, CUFFT_FORWARD));
    cufftSafeCall(cufftExecC2C(plan, pTxx, pTxx, CUFFT_FORWARD));
    cufftSafeCall(cufftExecC2C(plan, pTyy, pTyy, CUFFT_FORWARD));
    cufftSafeCall(cufftExecC2C(plan, pTxy, pTxy, CUFFT_FORWARD));
    cublasCscal(size, temp2, p1, 1);
    cublasCscal(size, temp2, p2, 1);
    cublasCscal(size, temp2, pvx, 1);
    cublasCscal(size, temp2, pvy, 1);
    cublasCscal(size, temp2, pTxx, 1);
    cublasCscal(size, temp2, pTyy, 1);
    cublasCscal(size, temp2, pTxy, 1);

    pointmul<<<dimGrid,dimBlock>>>(r1, p1, w1, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(r2, p2, w2, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(rvx, pvx, wvx, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(rvy, pvy, wvy, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(rTxx, pTxx, wTxx, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(rTyy, pTyy, wTyy, sx, sy);
    pointmul<<<dimGrid,dimBlock>>>(rTxy, pTxy, wTxy, sx, sy);

    cufftSafeCall(cufftExecC2C(plan, r1, r1, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, r2, r2, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, rvx, rvx, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, rvy, rvy, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, rTxx, rTxx, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, rTyy, rTyy, CUFFT_INVERSE));
    cufftSafeCall(cufftExecC2C(plan, rTxy, rTxy, CUFFT_INVERSE));
    cublasCscal(size, temp3, r1, 1);
    cublasCscal(size, temp3, r2, 1);
    cublasCscal(size, temp3, rvx, 1);
    cublasCscal(size, temp3, rvy, 1);
    cublasCscal(size, temp3, rTxx, 1);
    cublasCscal(size, temp3, rTyy, 1);
    cublasCscal(size, temp3, rTxy, 1);
}

void g_calc_function(cuComplex* rho, cuComplex* tmp, cuComplex* p1, cuComplex* p2, cuComplex*
     pvx, cuComplex* pvy, cuComplex* pTxx, cuComplex* pTyy, cuComplex* pTxy, cuComplex* v, int sx
     , int sy, dim3 dimBlock, dim3 dimGrid)
{
    function<<<dimGrid,dimBlock>>>(tmp, p1, p2, pvx, pvy, pTxx, pTyy, pTxy, v, rho0, sx, sy);
    mix<<<dimGrid,dimBlock>>>(rho, rho, tmp, mixing, sx, sy);
}

void g_calc_force(cuComplex* f, cuComplex* p1, cuComplex* p2, cuComplex* pvx, cuComplex* pvy,
     cuComplex* pTxx, cuComplex* pTyy, cuComplex* pTxy, int sx, int sy, dim3 dimBlock, dim3
     dimGrid)
```

```
{
    force<<<dimGrid,dimBlock>>>(f, p1, p2, pvx, pvy, pTxx, pTyy, pTxy, rho0, sx, sy);
}

void g_calc_delta(cuComplex* drho, cuComplex* rho, float rho0, int sx, int sy, dim3 dimBlock,
    dim3 dimGrid)
{
    get_delta<<<dimGrid,dimBlock>>>(drho, rho, rho0, sx, sy);
}

void iterate_gpu(dim3 dimBlock, dim3 dimGrid)
{
    CUT_SAFE_CALL(cutStartTimer(t2));
    g_calc_delta(d_deltarho, d_rho, rho0, size_x, size_y, dimBlock, dimGrid);
    g_calc_n(d_n1, d_n2, d_nvx, d_nvy, d_nTxx, d_nTyy, d_nTxy, d_nadd, d_w1, d_w2, d_wvx, d_wvy,
        d_wTxx, d_wTyy, d_wTxy, d_tmp, d_deltarho, size_x, size_y, dimBlock, dimGrid);

    float n2max = 0;
    cutilSafeCall(cudaMemcpy(h_w2, d_n2, sizeof(cuComplex)*size_x*size_y, cudaMemcpyDeviceToHost
        ));
    for (int i = 0; i < size_x*size_y; i++)
      if (h_w2[i].x > n2max) n2max = h_w2[i].x;

    g_calc_phi(d_p1, d_p2, d_pvx, d_pvy, d_pTxx, d_pTyy, d_pTxy, d_padd, d_n1, d_n2, d_nvx,
        d_nvy, d_nTxx, d_nTyy, d_nTxy, size_x, size_y, dimBlock, dimGrid);
    g_calc_conv(d_p1, d_p2, d_pvx, d_pvy, d_pTxx, d_pTyy, d_pTxy, d_p1, d_p2, d_pvx, d_pvy,
        d_pTxx, d_pTyy, d_pTxy, d_w1, d_w2, d_wvx, d_wvy, d_wTxx, d_wTyy, d_wTxy, size_x, size_y
        , dimBlock, dimGrid);
    g_calc_function(d_rho, d_tmp, d_p1, d_p2, d_pvx, d_pvy, d_pTxx, d_pTyy, d_pTxy, d_v, size_x,
        size_y, dimBlock, dimGrid);

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t2));

    CUT_SAFE_CALL(cutStartTimer(t3));

    compare_fast<<<1, MAX_BLOCK>>>(d_tmp, d_rho, size_x, size_y, dx, radius);
    cutilSafeCall(cudaMemcpy(h_v, d_tmp, sizeof(cuComplex), cudaMemcpyDeviceToHost));
    epsilon_old = epsilon;
    epsilon = h_v[0].x;
    if (epsilon < precision)
    {
        running = false;
        g_calc_force(d_force, d_p1, d_p2, d_pvx, d_pvy, d_pTxx, d_pTyy, d_pTxy, size_x, size_y,
            dimBlock, dimGrid);
    }

    printf("#Step: %4u  Eta: %4f  Rho: %4f  Epsilon: %g  Mixing: %g  n2max: %
        g\n", loops, rho0*M_PI/4.0, rho0, epsilon, mixing, n2max);
    mixing *= 1.1;
    if (epsilon_old < epsilon) mixing *= 0.3;
    if (mixing > max_mixing) mixing = max_mixing;
    if (mixing < min_mixing) mixing = min_mixing;
    if (isnan(epsilon))
    {
        write_force(d_tmp, "error_tmp", radius, -1);
        write_force(d_rho, "error_rho", radius, -1);
        write_force(d_n1, "error_n1", radius, -1);
        write_force(d_n2, "error_n2", radius, -1);
        write_force(d_nvx, "error_nvx", radius, -1);
        write_force(d_nTxx, "error_nTxx", radius, -1);
        write_force(d_nTxy, "error_nTxy", radius, -1);
        write_force(d_p1, "error_p1", radius, -1);
        write_force(d_p2, "error_p2", radius, -1);
        write_force(d_pvx, "error_pvx", radius, -1);
        write_force(d_pTxx, "error_pTxx", radius, -1);
        write_force(d_pTxy, "error_pTxy", radius, -1);
```

```
        exit(1);
    }

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t3));
}

void cleanup()
{
    //Destroy CUFFT context
    cufftSafeCall(cufftDestroy(plan));
    cudaThreadExit();
}

void memory_allocation()
{
    h_rho = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_v = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_tmp = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_w1 = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_w2 = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_wvx = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_wvy = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_wTxx = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_wTyy = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);
    h_wTxy = (cuComplex*)malloc(sizeof(cuComplex) * size_x * size_y);

    h_nadd = (cuComplex*)malloc(sizeof(cuComplex) * 4);
    h_padd = (cuComplex*)malloc(sizeof(cuComplex) * 4);

    cutilSafeCall(cudaMalloc((void**)&d_v, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_rho, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_deltarho, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_tmp, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_force, sizeof(cuComplex)*size_x*size_y));

    cutilSafeCall(cudaMalloc((void**)&d_w1, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_w2, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_wvx, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_wvy, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_wTxx, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_wTyy, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_wTxy, sizeof(cuComplex)*size_x*size_y));

    cutilSafeCall(cudaMalloc((void**)&d_n1, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_n2, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_nvx, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_nvy, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_nTxx, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_nTyy, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_nTxy, sizeof(cuComplex)*size_x*size_y));

    cutilSafeCall(cudaMalloc((void**)&d_p1, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_p2, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_pvx, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_pvy, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_pTxx, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_pTyy, sizeof(cuComplex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_pTxy, sizeof(cuComplex)*size_x*size_y));

    cutilSafeCall(cudaMalloc((void**)&d_nadd, sizeof(cuComplex)*4));
    cutilSafeCall(cudaMalloc((void**)&d_padd, sizeof(cuComplex)*4));
}

void initialize_v(float dr)
{
    for (int i = 0; i < size_x; i++)
```

```
    {
        for (int j = 0; j < size_y; j++)
        {
            cuComplex r = ij_to_r(i, j);

            h_v[i+j*size_x].x = V(r, dr, dx*0.8);
            h_v[i+j*size_x].y = 0;

            h_rho[i+j*size_x].x = (h_v[i+j*size_x].x==0?rho0:0.0);
            h_rho[i+j*size_x].y = 0;
        }
    }

    float n1 = 2.0 * M_PI * radius * rho0;
    float n2 = M_PI * radius * radius * rho0;
    float nT = M_PI * radius * rho0;

    h_nadd[0].x = n1;
    h_nadd[0].y = 0;
    h_nadd[1].x = n2;
    h_nadd[1].y = 0;
    h_nadd[2].x = 0;
    h_nadd[2].y = 0;
    h_nadd[3].x = nT;
    h_nadd[3].y = 0;

    h_padd[0].x = -( 1.0/(1-n2) * 1.0/M_PI * ( (1.0+a)/3.0 ) * n1 - log(1-n2) / (2*M_PI*radius)
        );
    h_padd[0].y = 0;
    h_padd[1].x = -( 1.0/( (1-n2)*(1-n2) ) * 1.0/M_PI * ( ( (1.0+a)/6.0 ) * n1*n1 + ( (1.0-2*a)
        /6.0 ) * 2.0*nT*nT) + n1 / (2*M_PI*radius*(1-n2)) );
    h_padd[1].y = 0;
    h_padd[2].x = 0;
    h_padd[2].y = 0;
    h_padd[3].x = -( nT * 1.0/(1-n2) * 1.0/M_PI * ( (1.0-2*a)/3.0 ) );
    h_padd[3].y = 0;

    // Copy host memory to device
    if (flag) cutilSafeCall(cudaMemcpy(d_rho, h_rho, sizeof(cuComplex)*size_x*size_y,
        cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMemcpy(d_v, h_v, sizeof(cuComplex)*size_x*size_y, cudaMemcpyHostToDevice))
        ;
    cutilSafeCall(cudaMemcpy(d_nadd, h_nadd, sizeof(cuComplex)*4, cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMemcpy(d_padd, h_padd, sizeof(cuComplex)*4, cudaMemcpyHostToDevice));
    flag = false;
}

void initialize_w()
{
    for (int i = 0; i < size_x; i++)
    {
        for (int j = 0; j < size_y; j++)
        {
            cuComplex r = ij_to_r(i, j);
            cuComplex e = ij_to_r(i, j);
            float k = absolute(e);
            if (k == 0)
            {
                e.x = 1.0;
                e.y = 0.0;
            } else
            {
                e.x /= k;
                e.y /= k;
            }

            h_w1[i+j*size_x].x = w1(k);
```

```
            h_w1[i+j*size_x].y = 0;

            h_w2[i+j*size_x].x = w2(k);
            h_w2[i+j*size_x].y = 0;

            h_wvx[i+j*size_x].x = 0;
            h_wvx[i+j*size_x].y = wvx(k, e, r);
            h_wvy[i+j*size_x].x = 0;
            h_wvy[i+j*size_x].y = wvy(k, e, r);

            h_wTxx[i+j*size_x].x = wTxx(k, e, r);
            h_wTxx[i+j*size_x].y = 0;
            h_wTyy[i+j*size_x].x = wTyy(k, e, r);
            h_wTyy[i+j*size_x].y = 0;
            h_wTxy[i+j*size_x].x = wTxy(k, e, r);
            h_wTxy[i+j*size_x].y = 0;
        }
    }

    // CUFFT plan
    cufftSafeCall(cufftPlan2d(&plan, size_x, size_y, CUFFT_C2C));

    // Copy host memory to device
    cutilSafeCall(cudaMemcpy(d_w1, h_w1, sizeof(cuComplex)*size_x*size_y, cudaMemcpyHostToDevice
        ));
    cutilSafeCall(cudaMemcpy(d_w2, h_w2, sizeof(cuComplex)*size_x*size_y, cudaMemcpyHostToDevice
        ));
    cutilSafeCall(cudaMemcpy(d_wvx, h_wvx, sizeof(cuComplex)*size_x*size_y,
        cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMemcpy(d_wvy, h_wvy, sizeof(cuComplex)*size_x*size_y,
        cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMemcpy(d_wTxx, h_wTxx, sizeof(cuComplex)*size_x*size_y,
        cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMemcpy(d_wTyy, h_wTyy, sizeof(cuComplex)*size_x*size_y,
        cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMemcpy(d_wTxy, h_wTxy, sizeof(cuComplex)*size_x*size_y,
        cudaMemcpyHostToDevice));

}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
void loop(dim3 dimBlock, dim3 dimGrid, float delta)
{
    bool write = true;
    float tr = 1.0 / (M_PI*radius*radius);
    float eta_steps[] = {0.50, 0.54, 0.58, 0.60, 0.62, 0.63, 0.64, 0.65, 0.66, 0.67, 0.675,
        0.68, 0};
    float* arrow = eta_steps;
    flag = true;
    while (*arrow != 0)
    {
        float eta = *arrow;
        rho0 = eta*tr;
        loops = 0;
        mixing = min_mixing;
        initialize_v(delta);
        printf("#delta_%4.2f\n", delta);
            while (running && loops < max_loops)
            {
                loops++;
//              if (loops == max_loops) write = false;
                iterate_gpu(dimBlock, dimGrid);
//running = false;
            }
        running = true;
```

```
        arrow++;
    }

    char name[100];
    sprintf(name, "force_eta68_%4.2f_a12.dat", delta);
    if (write) write_force(d_force, name, radius, delta);
    write = true;
}


int main(int argc, char** argv)
{
    if (argc >= 2) {
        size_x = atoi(argv[1]);
        size_y = atoi(argv[2]);
    }

    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutCreateTimer(&t1));
    CUT_SAFE_CALL(cutCreateTimer(&t2));
    CUT_SAFE_CALL(cutCreateTimer(&t3));

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    int GRID_X = (int) ceil((float) size_x/(float) BLOCK_SIZE);
    int GRID_Y = (int) ceil((float) size_y/(float) BLOCK_SIZE);
    dim3 dimGrid(GRID_X, GRID_Y);

    cublasInit();
    CUT_SAFE_CALL(cutStartTimer(t1));
    memory_allocation();
    initialize_w();
    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t1));

    loop(dimBlock, dimGrid, 0);
    for (float delta = 0.5; delta < 0.851; delta += 0.15)
        loop(dimBlock, dimGrid, delta);
    for (float delta = 0.85; delta < 1.05; delta += 0.004)
        loop(dimBlock, dimGrid, delta);
    for (float delta = 1.1; delta < 1.31; delta += 0.1)
        loop(dimBlock, dimGrid, delta);

    printf("#Initialization time: %g ms\n", cutGetTimerValue(t1));
    printf("#Convolution time: %g ms\n", cutGetTimerValue(t2));
    printf("#Convergence time: %g ms\n", cutGetTimerValue(t3));
    printf("#loops: %u\n", loops);

    cleanup();
    fflush(stdout);
    fflush(stderr);
    exit(EXIT_SUCCESS);
}
```

Listing A.1: GPU-basierte Implementierung des Tensorfunktionals

```
#include <string.h>

extern float j0(float x);
extern float j1(float x);
extern float j2(float x);
extern float dx;
extern const float radius;

float w1(float r)
{
    return 2.0 * M_PI * radius * j0(radius * r * 2.0 * M_PI / (size_x * dx * dx));
```

```
}

float w2(float r)
{
    if (r == 0)
      return M_PI / 4.0;
    else
      return 2.0 * M_PI * radius * 1.0/(r * 2.0 * M_PI / (size_x * dx * dx)) * j1(radius * r *
          2.0 * M_PI / (size_x * dx * dx));
}

float wvx(float k, cuComplex e, cuComplex r)
{
      if (r.x == dx * size_x / 2.0)
        return 0;
      else
        return w2(k) * k * 2 * M_PI / (size_x * dx * dx) * e.x;
}

float wvy(float k, cuComplex e, cuComplex r)
{
      if (r.y == dx * size_y / 2.0)
        return 0;
      else
        return w2(k) * k * 2 * M_PI / (size_x * dx * dx) * e.y;
}

float wTxx(float k, cuComplex e, cuComplex r)
{
    float kR = radius * k * 2.0 * M_PI / (size_x * dx * dx);
    return M_PI * radius * ( j0(kR) + j2(kR) ) - 2.0 * M_PI * radius * j2(kR) * e.x * e.x;
}

float wTyy(float k, cuComplex e, cuComplex r)
{
    float kR = radius * k * 2.0 * M_PI / (size_x * dx * dx);
    return M_PI * radius * ( j0(kR) + j2(kR) ) - 2.0 * M_PI * radius * j2(kR) * e.y * e.y;
}

float wTxy(float k, cuComplex e, cuComplex r)
{
    float kR = radius * k * 2.0 * M_PI / (size_x * dx * dx);
    return 2.0 * M_PI * radius * j2(kR) * e.x * e.y;
}
```

Listing A.2: Berechnung der ω-Funktionen im Fourierraum

A.1.2. GPU Taylor DFT

```
//Headers
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cutil.h>

#include <cufft.h>
#include <cublas.h>
#include <cutil_inline.h>

#define BLOCK_SIZE 16
#define BLOCK_LENGTH 256
#define MAX_BLOCK 512

// Complex data type
typedef float2 Complex;

////////////////////////////////////////////////////////////////////////////////
// declaration, forward
void initialize();
void iterate_gpu();
void iterate_shared();
void cleanup();
float V(int x, int y);
float c2(int i, int j, float d, float rho, int sx, int sy);
float compare(Complex* a, Complex* b, int size_x, int size_y);

int size_x = 1024;
int size_y = 1024;
float dx = 0.01;

float radius = 0.5;
float rho0 = 0.65 / (4*radius*radius);
float min_mixing = 0.001;
float max_mixing = 0.1;
float mixing = sqrt( min_mixing * max_mixing );

float epsilon = 0;
float epsilon_old = 0;

Complex* h_fold;
Complex* h_rho;
Complex* h_c2;
Complex* h_v;
Complex* h_drho;

__device__ Complex* d_fold;
__device__ Complex* d_rho;
__device__ Complex* d_c2;
__device__ Complex* d_v;
__device__ Complex* d_drho;
__device__ Complex* d_tmp;

unsigned int t1 = 0;
unsigned int t2 = 0;
unsigned int t3 = 0;

__device__ cufftHandle plan;

bool running = true;
int loops = 0;

////////////////////////////////////////////////////////////////////////////////
```

```
// Program main
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
    if (argc >= 2) {
        size_x = atoi(argv[1]);
        size_y = atoi(argv[2]);
    }

    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutCreateTimer(&t1));
    CUT_SAFE_CALL(cutCreateTimer(&t2));
    CUT_SAFE_CALL(cutCreateTimer(&t3));

    CUT_SAFE_CALL(cutStartTimer(t1));
    cublasInit();
    initialize();
    printf("#Initialization time: %g ms\n", cutGetTimerValue(t1));

    while (running && loops < 1e+06)
    {
        loops++;
        iterate_gpu();
    }

    printf("#FFT time: %g ms\n", cutGetTimerValue(t2));
    printf("#Function evaluation time: %g ms\n", cutGetTimerValue(t3));

    cutilSafeCall(cudaMemcpy(h_rho, d_rho, sizeof(Complex)*size_x*size_y, cudaMemcpyDeviceToHost
        ));

    for (int i = 0; i < size_x/2; i++)
      for (int j = 0; j < size_y/2; j++)
        printf("%f %f\n", dx*sqrt(i*i + j*j), h_rho[i*size_x+j].x);

    printf("#loops: %u\n", loops);

    cleanup();

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t1));
    printf("#Overall time: %g ms\n", cutGetTimerValue(t1));

    fflush(stdout);
    fflush(stderr);
    exit(EXIT_SUCCESS);
}

float c2(int i, int j, float d, float rho, int sx, int sy)
{
    float r;
    if (i > sx/2 && j > sy/2) r = dx*sqrt((sx-i-1)*(sx-i-1)+(sy-j-1)*(sy-j-1));
    else if (i < sx/2 && j > sy/2) r = dx*sqrt(i*i+(sy-j-1)*(sy-j-1));
    else if (i > sx/2 && j < sy/2) r = dx*sqrt((sx-i-1)*(sx-i-1)+j*j);
    else r = dx*sqrt(i*i+j*j);

    float x = r/d;
    float eta = M_PI_4 * d*d * rho;
    float res = 0.0;

    if (x <= 1.0)
    {
        float dV = 2/M_PI * (acos(x) - x*sqrt(1-x*x));
        float dA = 2/M_PI * acos(x);

        float g = 1 / (1-eta);
        float chi = (1+eta) / ((1-eta)*(1-eta)*(1-eta));
```

```
        float a = (1+chi*(2*eta-1)+2*eta*g)/eta;
        float b = (chi*(1-eta)-1-3*eta*g)/eta;

        res = -(a*dV + b*dA + g);
    }
    return res;
}

float V(int x, int y)
{
    if (x > size_x/2 && y < size_y/2) return(dx*sqrt((size_x-x-1)*(size_x-x-1) + y*y)>2*radius
        ?0:1e+50);
    else if (x < size_x/2 && y > size_y/2) return(dx*sqrt(x*x + (size_y-y-1)*(size_y-y-1))>2*
        radius?0:1e+50);
    else if (x > size_x/2 && y > size_y/2) return(dx*sqrt((size_x-x-1)*(size_x-x-1) + (size_y-y
        -1)*(size_y-y-1))>2*radius?0:1e+50);
    else return(dx*sqrt(x*x + y*y)>2*radius?0:1e+50);
}

void initialize()
{
    // Allocate host memory for the signal
    h_fold = (Complex*)malloc(sizeof(Complex) * size_x * size_y);
    h_c2 = (Complex*)malloc(sizeof(Complex) * size_x * size_y);
    h_rho = (Complex*)malloc(sizeof(Complex) * size_x * size_y);
    h_v = (Complex*)malloc(sizeof(Complex) * size_x * size_y);
    h_drho = (Complex*)malloc(sizeof(Complex) * size_x * size_y);

    // Initalize stuff
    for (int i = 0; i < size_x; i++)
    {
        for (int j = 0; j < size_y; j++)
        {
            h_v[i+j*size_x].x = V(i,j);
            h_v[i+j*size_x].y = 0;

            h_c2[i+j*size_x].x = c2(i, j, 2*radius, rho0, size_x, size_y);
            h_c2[i+j*size_x].y = 0;

            h_rho[i+j*size_x].x = rho0;
            h_rho[i+j*size_x].y = 0;

            h_drho[i+j*size_x].x = 0;
            h_drho[i+j*size_x].y = 0;
        }
    }

    // CUFFT plan
    cufftSafeCall(cufftPlan2d(&plan, size_x, size_y, CUFFT_C2C));

    // Allocate device memory for signal
    cutilSafeCall(cudaMalloc((void**)&d_fold, sizeof(Complex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_c2, sizeof(Complex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_rho, sizeof(Complex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_v, sizeof(Complex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_drho, sizeof(Complex)*size_x*size_y));
    cutilSafeCall(cudaMalloc((void**)&d_tmp, sizeof(Complex)*size_x*size_y));
    // Copy host memory to device
    cutilSafeCall(cudaMemcpy(d_c2, h_c2, sizeof(Complex)*size_x*size_y, cudaMemcpyHostToDevice))
        ;
    cutilSafeCall(cudaMemcpy(d_rho, h_rho, sizeof(Complex)*size_x*size_y, cudaMemcpyHostToDevice
        ));
    cutilSafeCall(cudaMemcpy(d_v, h_v, sizeof(Complex)*size_x*size_y, cudaMemcpyHostToDevice));
    cutilSafeCall(cudaMemcpy(d_drho, h_drho, sizeof(Complex)*size_x*size_y,
        cudaMemcpyHostToDevice));
```

```
    cufftSafeCall(cufftExecC2C(plan, (cufftComplex *)d_c2, (cufftComplex *)d_c2, CUFFT_FORWARD))
        ;
}

__global__ void pointmul(cufftComplex* a, cufftComplex* b, cufftComplex* c, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        a[pos].x = b[pos].x * c[pos].x - b[pos].y * c[pos].y;
        a[pos].y = b[pos].x * c[pos].y + b[pos].y * c[pos].x;
        __syncthreads();
    }
}

__global__ void function(cufftComplex* res, cufftComplex* v, cufftComplex* fold, float r, int sx
    , int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        res[pos].x = r * exp(- v[pos].x + fold[pos].x);
        res[pos].y = 0;
        __syncthreads();
    }
}

__global__ void mix(cufftComplex* res, cufftComplex* a, cufftComplex* b, float mix, int sx, int
    sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        res[pos].x = mix * a[pos].x + (1.0-mix) * b[pos].x;
        res[pos].y = 0;
        __syncthreads();
    }
}

__global__ void get_delta(cufftComplex* res, cufftComplex* a, float b, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        res[pos].x = a[pos].x - b;
        res[pos].y = 0;
        __syncthreads();
    }
}

__global__ void compare_gpu_step1(cufftComplex* a, cufftComplex* b, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        a[pos].x = (a[pos].x - b[pos].x) * (a[pos].x - b[pos].x) / (sx*sy);
    }
```

```
}

__global__ void compare_gpu_step2(cufftComplex* a, int s, int sx, int sy)
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (X < sx && Y < sy)
    {
        if (pos < s)
            {
                a[pos].x += a[pos + s].x;
                a[pos].y = 0;
            }
        }
}

void compare(cufftComplex* a, cufftComplex* b, int sx, int sy, dim3 dG, dim3 dB)
{
    compare_gpu_step1<<<dG,dB>>>(a, b, sx, sy);
    for(int s=sx*sy/2; s>0; s/=2)
        compare_gpu_step2<<<dG,dB>>>(a, s, sx, sy);
}

__global__ void compare_fast(cufftComplex* a, cufftComplex* b, int sx, int sy, float dx, float r
    )
{
    int X = blockIdx.x * blockDim.x + threadIdx.x;
    int Y = blockIdx.y * blockDim.y + threadIdx.y;
    int pos = X+Y*sx;
    if (pos < MAX_BLOCK)
    {
        __shared__ float c[MAX_BLOCK];
        c[pos] = a[pos + (int) (r/dx)].x - b[pos + (int) (r/dx)].x;
        c[pos] *= c[pos];
        c[pos] /= (float) MAX_BLOCK;
        __syncthreads();
        for (int s=MAX_BLOCK/2; s>0; s/=2)
        {
            if (pos < s)
            {
                c[pos] += c[pos + s];
            }
            __syncthreads();
        }
        if (pos == 0)
            {
                a[0].x = c[0];
                a[0].y = 0;
            }
    }
}

void iterate_gpu()
{
    CUT_SAFE_CALL(cutStartTimer(t2));

    // Transform signal
    cufftSafeCall(cufftExecC2C(plan, (cufftComplex *)d_drho, (cufftComplex *)d_drho,
        CUFFT_FORWARD));

    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    int GRID_X = (int) ceil((float) size_x/(float) BLOCK_SIZE);
    int GRID_Y = (int) ceil((float) size_y/(float) BLOCK_SIZE);
    dim3 dimGrid(GRID_X, GRID_Y);
    pointmul<<<dimGrid,dimBlock>>>(d_fold, d_drho, d_c2, size_x, size_y);
```

```
    cufftSafeCall(cufftExecC2C(plan, (cufftComplex *)d_fold, (cufftComplex *)d_fold,
        CUFFT_INVERSE));

    float temp1 = size_x * size_y;
    Complex temp2;
    temp2.x = dx*dx / temp1;
    temp2.y = 0;
    cublasCscal(temp1, (cufftComplex) temp2, (cufftComplex *)d_fold, 1);

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t2));

    cutilSafeCall(cudaMemcpy(d_tmp, d_rho, sizeof(Complex)*size_x*size_y,
        cudaMemcpyDeviceToDevice));

    CUT_SAFE_CALL(cutStartTimer(t3));
    function<<<dimGrid,dimBlock>>>(d_fold, d_v, d_fold, rho0, size_x, size_y);
    mix<<<dimGrid,dimBlock>>>(d_rho, d_fold, d_rho, mixing, size_x, size_y);
    get_delta<<<dimGrid,dimBlock>>>(d_drho, d_rho, rho0, size_x, size_y);

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t3));

    compare_fast<<<1, MAX_BLOCK>>>(d_tmp, d_rho, size_x, size_y, dx, radius);
    //compare(d_tmp, d_rho, size_x, size_y, dimGrid, dimBlock);
    cutilSafeCall(cudaMemcpy(h_c2, d_tmp, sizeof(Complex), cudaMemcpyDeviceToHost));
    epsilon_old = epsilon;
    epsilon = h_c2[0].x;
    if (epsilon < 1e-10) running = false;
    mixing *= 1.1;
    if (epsilon_old < epsilon) mixing *= 0.3;
    if (mixing > max_mixing) mixing = max_mixing;
    if (mixing < min_mixing) mixing = min_mixing;

}

void cleanup()
{
    //Destroy CUFFT context
    cufftSafeCall(cufftDestroy(plan));

    // cleanup memory
    free(h_fold);
    free(h_rho);
    free(h_c2);
    free(h_v);
    free(h_drho);
    cutilSafeCall(cudaFree(d_fold));
    cutilSafeCall(cudaFree(d_rho));
    cutilSafeCall(cudaFree(d_c2));
    cutilSafeCall(cudaFree(d_v));
    cutilSafeCall(cudaFree(d_drho));
    cutilSafeCall(cudaFree(d_tmp));

    cudaThreadExit();
}
```

Listing A.3: GPU-basierte Implementierung des Taylor-Funktionals

A.1.3. CPU SP Taylor DFT

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cutil.h>

#include <fftw3.h>
#include <cufft.h>
#include <cublas.h>
#include <cutil_inline.h>

#define BLOCK_SIZE 16
#define BLOCK_LENGTH 256
#define MAX_BLOCK 512

////////////////////////////////////////////////////////////////////////////
// declaration, forward
void initialize();
void iterate_cpu();
void iterate_shared();
void cleanup();
float V(int x, int y);
float c2(int i, int j, float d, float rho, int sx, int sy);

int size_x = 1024;
int size_y = 1024;
float dx = 0.01;

float radius = 0.5;
float rho0 = 0.65 / (4*radius*radius);
float min_mixing = 0.001;
float max_mixing = 0.1;
float mixing = 0.1;

float epsilon = 0;
float epsilon_old = 0;

fftwf_complex* h_fold;
fftwf_complex* h_rho;
fftwf_complex* h_c2;
fftwf_complex* h_v;
fftwf_complex* h_drho;
fftwf_complex* h_tmp;

unsigned int t1 = 0;
unsigned int t2 = 0;
unsigned int t3 = 0;

fftwf_plan p_forward, p_reverse, p_c2init;

bool running = true;
int loops = 0;

////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
    if (argc >= 2) {
        size_x = atoi(argv[1]);
        size_y = atoi(argv[2]);
    }

    CUDA_SAFE_CALL(cudaThreadSynchronize());
```

```
    CUT_SAFE_CALL(cutCreateTimer(&t1));
    CUT_SAFE_CALL(cutCreateTimer(&t2));
    CUT_SAFE_CALL(cutCreateTimer(&t3));

    CUT_SAFE_CALL(cutStartTimer(t1));
    //cublasInit();
    initialize();
    printf("#Initialization time: %g ms\n", cutGetTimerValue(t1));

    while (running && loops < 1e+06)
    {
        loops++;
        iterate_cpu();
    }

    printf("#FFT time: %g ms\n", cutGetTimerValue(t2));
    printf("#Function evaluation time: %g ms\n", cutGetTimerValue(t3));

    for (int i = 0; i < size_x/2; i++)
      for (int j = 0; j < size_y/2; j++)
        printf("%f %f\n", dx*sqrt(i*i + j*j), h_rho[i*size_x+j][0]);

    printf("#loops: %u\n", loops);

    cleanup();

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t1));
    printf("#Overall time: %g ms\n", cutGetTimerValue(t1));

    fflush(stdout);
    fflush(stderr);
    exit(EXIT_SUCCESS);
}

float c2(int i, int j, float d, float rho, int sx, int sy)
{
    float r;
    if (i > sx/2 && j > sy/2) r = dx*sqrt((sx-i-1)*(sx-i-1)+(sy-j-1)*(sy-j-1));
    else if (i < sx/2 && j > sy/2) r = dx*sqrt(i*i+(sy-j-1)*(sy-j-1));
    else if (i > sx/2 && j < sy/2) r = dx*sqrt((sx-i-1)*(sx-i-1)+j*j);
    else r = dx*sqrt(i*i+j*j);

    float x = r/d;
    float eta = M_PI_4 * d*d * rho;
    float res = 0.0;

    if (x <= 1.0)
    {
        float dV = 2/M_PI * (acos(x) - x*sqrt(1-x*x));
        float dA = 2/M_PI * acos(x);

        float g = 1 / (1-eta);
        float chi = (1+eta) / ((1-eta)*(1-eta)*(1-eta));

        float a = (1+chi*(2*eta-1)+2*eta*g)/eta;
        float b = (chi*(1-eta)-1-3*eta*g)/eta;

        res = -(a*dV + b*dA + g);
    }
    return res;
}

float V(int x, int y)
{
    if (x > size_x/2 && y < size_y/2) return(dx*sqrt((size_x-x-1)*(size_x-x-1) + y*y)>2*radius
        ?0:1e+50);
```

```
    else if (x < size_x/2 && y > size_y/2) return(dx*sqrt(x*x + (size_y-y-1)*(size_y-y-1))>2*
        radius?0:1e+50);
    else if (x > size_x/2 && y > size_y/2) return(dx*sqrt((size_x-x-1)*(size_x-x-1) + (size_y-y
        -1)*(size_y-y-1))>2*radius?0:1e+50);
    else return(dx*sqrt(x*x + y*y)>2*radius?0:1e+50);
}

void initialize()
{
    // Allocate host memory for the signal
    h_fold = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * size_x * size_y);
    h_c2 = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * size_x * size_y);
    h_rho = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * size_x * size_y);
    h_v = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * size_x * size_y);
    h_drho = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * size_x * size_y);
    h_tmp = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex) * size_x * size_y);

    // Initalize stuff
    for (int i = 0; i < size_x; i++)
    {
        for (int j = 0; j < size_y; j++)
        {
            h_tmp[i+j*size_x][0] = 0;
            h_tmp[i+j*size_x][1] = 0;

            h_v[i+j*size_x][0] = V(i,j);
            h_v[i+j*size_x][1] = 0;

            h_c2[i+j*size_x][0] = c2(i, j, 2*radius, rho0, size_x, size_y);
            h_c2[i+j*size_x][1] = 0;

            h_rho[i+j*size_x][0] = rho0;
            h_rho[i+j*size_x][1] = 0;

            h_drho[i+j*size_x][0] = 0;
            h_drho[i+j*size_x][1] = 0;
        }
    }

    p_forward = fftwf_plan_dft_2d(size_x, size_y, (fftwf_complex *)h_drho, (fftwf_complex *)
        h_drho, FFTW_FORWARD, FFTW_ESTIMATE);
    p_reverse = fftwf_plan_dft_2d(size_x, size_y, (fftwf_complex *)h_fold, (fftwf_complex *)
        h_fold, FFTW_BACKWARD, FFTW_ESTIMATE);
    p_c2init = fftwf_plan_dft_2d(size_x, size_y, (fftwf_complex *)h_c2, (fftwf_complex *)h_c2,
        FFTW_FORWARD, FFTW_ESTIMATE);
    fftwf_execute(p_c2init);
}

void iterate_cpu()
{
    CUT_SAFE_CALL(cutStartTimer(t2));

    fftwf_execute(p_forward);

    for (int i = 0; i < size_x*size_y; i++)
    {
      h_fold[i][0] = h_drho[i][0] * h_c2[i][0] - h_drho[i][1] * h_c2[i][1];
      h_fold[i][1] = h_drho[i][0] * h_c2[i][1] + h_drho[i][1] * h_c2[i][0];
    }

    fftwf_execute(p_reverse);

    float temp1 = size_x * size_y;
    fftwf_complex temp2;
    temp2[0] = dx*dx / temp1;
    temp2[1] = 0;
```

```
    for (int i = 0; i < size_x*size_y; i++)
    {
      h_fold[i][0] = h_fold[i][0] * temp2[0];
      h_fold[i][1] = temp2[1];
    }

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t2));

    for (int i = 0; i < size_x*size_y; i++)
    {
      h_tmp[i][0] = h_rho[i][0];
      h_tmp[i][1] = h_rho[i][1];
    }

    CUT_SAFE_CALL(cutStartTimer(t3));

    // function
    for (int i = 0; i < size_x*size_y; i++)
    {
        h_fold[i][0] = radius * exp(- h_v[i][0] + h_fold[i][0]);
        h_fold[i][1] = 0;
    }

    // mix
    for (int i = 0; i < size_x*size_y; i++)
    {
        h_rho[i][0] = mixing * h_fold[i][0] + (1.0-mixing) * h_rho[i][0];
        h_rho[i][1] = 0;
    }

    // delta
    for (int i = 0; i < size_x*size_y; i++)
    {
        h_drho[i][0] = h_rho[i][0] - rho0;
        h_drho[i][1] = 0;
    }

    CUT_SAFE_CALL(cutStopTimer(t3));

    epsilon_old = epsilon;
    epsilon = 0;
    for (int i=0; i<size_x*size_y; i++)
      epsilon += ((h_tmp[i][0] - h_rho[i][0]) * (h_tmp[i][0] - h_rho[i][0])) / (size_x*size_y);
    if (epsilon < 1e-10) running = false;

    printf("#Epsilon: %g\n", epsilon);

}

void cleanup()
{
    // cleanup memory
    free(h_fold);
    free(h_rho);
    free(h_c2);
    free(h_v);
    free(h_drho);
}
```

Listing A.4: CPu Code mit einfacher Genauigkeit

A.1.4. CPU DP Taylor DFT

```
//Headers
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cutil.h>

#include <fftw3.h>
#include <cufft.h>
#include <cublas.h>
#include <cutil_inline.h>

#define BLOCK_SIZE 16
#define BLOCK_LENGTH 256
#define MAX_BLOCK 512

////////////////////////////////////////////////////////////////////////////
// declaration, forward
void initialize();
void iterate_cpu();
void iterate_shared();
void cleanup();
double V(int x, int y);
double c2(int i, int j, double d, double rho, int sx, int sy);

int size_x = 1024;
int size_y = 1024;
double dx = 0.01;

double radius = 0.5;
double rho0 = 0.65 / (4*radius*radius);
double min_mixing = 0.001;
double max_mixing = 0.1;
double mixing = 0.1;

double epsilon = 0;
double epsilon_old = 0;

fftw_complex* h_fold;
fftw_complex* h_rho;
fftw_complex* h_c2;
fftw_complex* h_v;
fftw_complex* h_drho;
fftw_complex* h_tmp;

unsigned int t1 = 0;
unsigned int t2 = 0;
unsigned int t3 = 0;

fftw_plan p_forward, p_reverse, p_c2init;

bool running = true;
int loops = 0;

////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
    if (argc >= 2) {
        size_x = atoi(argv[1]);
        size_y = atoi(argv[2]);
    }
```

```
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutCreateTimer(&t1));
    CUT_SAFE_CALL(cutCreateTimer(&t2));
    CUT_SAFE_CALL(cutCreateTimer(&t3));

    CUT_SAFE_CALL(cutStartTimer(t1));

    initialize();
    printf("#Initialization time: %g ms\n", cutGetTimerValue(t1));

    while (running && loops < 1e+06)
    {
        loops++;
        iterate_cpu();
    }

    printf("#FFT time: %g ms\n", cutGetTimerValue(t2));
    printf("#Function evaluation time: %g ms\n", cutGetTimerValue(t3));

    for (int i = 0; i < size_x/2; i++)
      for (int j = 0; j < size_y/2; j++)
        printf("%f %f\n", dx*sqrt(i*i + j*j), h_rho[i*size_x+j][0]);

    printf("#loops: %u\n", loops);

    cleanup();

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t1));
    printf("#Overall time: %g ms\n", cutGetTimerValue(t1));

    fflush(stdout);
    fflush(stderr);
    exit(EXIT_SUCCESS);
}

double c2(int i, int j, double d, double rho, int sx, int sy)
{
    double r;
    if (i > sx/2 && j > sy/2) r = dx*sqrt((sx-i-1)*(sx-i-1)+(sy-j-1)*(sy-j-1));
    else if (i < sx/2 && j > sy/2) r = dx*sqrt(i*i+(sy-j-1)*(sy-j-1));
    else if (i > sx/2 && j < sy/2) r = dx*sqrt((sx-i-1)*(sx-i-1)+j*j);
    else r = dx*sqrt(i*i+j*j);

    double x = r/d;
    double eta = M_PI_4 * d*d * rho;
    double res = 0.0;

    if (x <= 1.0)
    {
        double dV = 2/M_PI * (acos(x) - x*sqrt(1-x*x));
        double dA = 2/M_PI * acos(x);

        double g = 1 / (1-eta);
        double chi = (1+eta) / ((1-eta)*(1-eta)*(1-eta));

        double a = (1+chi*(2*eta-1)+2*eta*g)/eta;
        double b = (chi*(1-eta)-1-3*eta*g)/eta;

        res = -(a*dV + b*dA + g);
    }
    return res;
}

double V(int x, int y)
{
```

```
    if (x > size_x/2 && y < size_y/2) return(dx*sqrt((size_x-x-1)*(size_x-x-1) + y*y)>2*radius
        ?0:1e+50);
    else if (x < size_x/2 && y > size_y/2) return(dx*sqrt(x*x + (size_y-y-1)*(size_y-y-1))>2*
        radius?0:1e+50);
    else if (x > size_x/2 && y > size_y/2) return(dx*sqrt((size_x-x-1)*(size_x-x-1) + (size_y-y
        -1)*(size_y-y-1))>2*radius?0:1e+50);
    else return(dx*sqrt(x*x + y*y)>2*radius?0:1e+50);
}

void initialize()
{
    // Allocate host memory for the signal
    h_fold = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * size_x * size_y);
    h_c2 = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * size_x * size_y);
    h_rho = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * size_x * size_y);
    h_v = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * size_x * size_y);
    h_drho = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * size_x * size_y);
    h_tmp = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * size_x * size_y);

    // Initalize stuff
    for (int i = 0; i < size_x; i++)
    {
        for (int j = 0; j < size_y; j++)
        {
            h_tmp[i+j*size_x][0] = 0;
            h_tmp[i+j*size_x][1] = 0;

            h_v[i+j*size_x][0] = V(i,j);
            h_v[i+j*size_x][1] = 0;

            h_c2[i+j*size_x][0] = c2(i, j, 2*radius, rho0, size_x, size_y);
            h_c2[i+j*size_x][1] = 0;

            h_rho[i+j*size_x][0] = rho0;
            h_rho[i+j*size_x][1] = 0;

            h_drho[i+j*size_x][0] = 0;
            h_drho[i+j*size_x][1] = 0;
        }
    }

    p_forward = fftw_plan_dft_2d(size_x, size_y, (fftw_complex *)h_drho, (fftw_complex *)h_drho,
         FFTW_FORWARD, FFTW_ESTIMATE);
    p_reverse = fftw_plan_dft_2d(size_x, size_y, (fftw_complex *)h_fold, (fftw_complex *)h_fold,
         FFTW_BACKWARD, FFTW_ESTIMATE);
    p_c2init = fftw_plan_dft_2d(size_x, size_y, (fftw_complex *)h_c2, (fftw_complex *)h_c2,
        FFTW_FORWARD, FFTW_ESTIMATE);
    fftw_execute(p_c2init);
}

void iterate_cpu()
{
    CUT_SAFE_CALL(cutStartTimer(t2));

    // Transform signal
    fftw_execute(p_forward);

    for (int i = 0; i < size_x*size_y; i++)
    {
      h_fold[i][0] = h_drho[i][0] * h_c2[i][0] - h_drho[i][1] * h_c2[i][1];
      h_fold[i][1] = h_drho[i][0] * h_c2[i][1] + h_drho[i][1] * h_c2[i][0];
    }

    fftw_execute(p_reverse);

    double temp1 = size_x * size_y;
    fftw_complex temp2;
```

```
    temp2[0] = dx*dx / temp1;
    temp2[1] = 0;

    for (int i = 0; i < size_x*size_y; i++)
    {
      h_fold[i][0] = h_fold[i][0] * temp2[0];
      h_fold[i][1] = temp2[1];
    }

    cudaThreadSynchronize();
    CUT_SAFE_CALL(cutStopTimer(t2));

    for (int i = 0; i < size_x*size_y; i++)
    {
      h_tmp[i][0] = h_rho[i][0];
      h_tmp[i][1] = h_rho[i][1];
    }

    CUT_SAFE_CALL(cutStartTimer(t3));

    // function
    for (int i = 0; i < size_x*size_y; i++)
    {
        h_fold[i][0] = radius * exp(- h_v[i][0] + h_fold[i][0]);
        h_fold[i][1] = 0;
    }

    // mix
    for (int i = 0; i < size_x*size_y; i++)
    {
        h_rho[i][0] = mixing * h_fold[i][0] + (1.0-mixing) * h_rho[i][0];
        h_rho[i][1] = 0;
    }

    // delta
    for (int i = 0; i < size_x*size_y; i++)
    {
        h_drho[i][0] = h_rho[i][0] - rho0;
        h_drho[i][1] = 0;
    }

    CUT_SAFE_CALL(cutStopTimer(t3));

    epsilon_old = epsilon;
    epsilon = 0;
    for (int i=0; i<size_x*size_y; i++)
      epsilon += ((h_tmp[i][0] - h_rho[i][0]) * (h_tmp[i][0] - h_rho[i][0])) / (size_x*size_y);
    if (epsilon < 1e-10) running = false;

    printf("#Epsilon: %g\n", epsilon);
}

void cleanup()
{
    // cleanup memory
    free(h_fold);
    free(h_rho);
    free(h_c2);
    free(h_v);
    free(h_drho);
}
```

Listing A.5: CPU Code mit doppelter Genauigkeit

A.1.5. Kraftberechnung

```
#include <cstdlib>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <cmath>

int size_x = 1024;
int size_y = 1024;
float x;
float y;
float* field;
float dx = 0.02;
float rho0 = 0;
float radius = 0.5;
float delta = 0;

using namespace std;

fstream ofile;

struct Complex
{
    float x; float y;
};

bool read_field(string fname)
{
    fstream file;
    file.open(fname.c_str(), ios::in);
    if (file.fail())
    {
        cout << "can_not_open_file" << endl;
        exit(1);
    }
    char temp;
    file >> temp;
    file >> size_x;
    file >> size_y;
    file >> dx;
    file >> rho0;
    file >> radius;
    file >> delta;

    field = (float*)malloc(sizeof(float) * size_x * size_y);
    for (int i = 0; i < size_x; i++)
    {
        for (int j = 0; j < size_y; j++)
        {
            file >> field[i+j*size_x];
        }
    }
    file.close();
    return true;
}

float get_point(float x, float y)
{
    if (x > size_y || y > size_y || x<0 || y<0) return -1;

    struct Complex a, b, c, d, r;

    a.x = x;
```

```
    a.y = y;

    b.x = floor(a.x);
    b.y = floor(a.y);

    c.x = ceil(a.x);
    c.y = ceil(a.y);

    d.x = a.x - b.x;
    d.y = a.y - b.y;

    int x1, x2, y1, y2, p11, p21, p12, p22;
    x1 = (int) b.x; y1 = (int) b.y;
    x2 = (int) c.x; y2 = (int) c.y;
    if (x1 == size_x) x1 = 0;
    if (x2 == size_x) x2 = 0;
    if (y1 == size_y) y1 = 0;
    if (y2 == size_y) y2 = 0;
    p11 = x1 + y1*size_x;
    p21 = x2 + y1*size_x;
    p12 = x1 + y2*size_x;
    p22 = x2 + y2*size_x;

    float tmp1, tmp2, result;
    tmp1 = (1.0 - d.x) * field[p11] + d.x * field[p21];
    tmp2 = (1.0 - d.x) * field[p12] + d.x * field[p22];
    result = (1.0 - d.y) * tmp1 + d.y * tmp2;
    return result;
}

float get_next(float x, float y)
{
    if (x > size_y || y > size_y || x<0 || y<0) return -1;

    int a, b;
    a = round(x);
    b = round(y);
    if (a == size_x) a = 0;
    if (b == size_y) b = 0;
    return field[a+size_x*b];
}

float get_contact(int steps, float r, float dr)
{
    float norm = (2.0*r) * (2.0*M_PI) / (float) steps;
    r *= 2.0/dx;
    float phi = 0;
    float resx = 0;
    float resy = 0;
    while (phi < 2.0*M_PI)
    {
        float x = r*cos(phi);
        float y = r*sin(phi) + dr/dx;
        if (x < 0) x = size_x + x;
        if (y < 0)
        {
//          x = size_x + x;
            phi += 2.0 * M_PI / (float) steps;
            continue;
        }
        float tmp = get_point(x, y+0.8);
        ofile << phi << "       " << tmp << endl;
        resx += cos(phi) * tmp * norm;
        resy -= sin(phi) * tmp * norm;
        phi += 2.0 * M_PI / (float) steps;
    }
    return resy;
```

```
}

float get_gr(int steps, float r)
{
    r /= dx;
    float phi = 0;
    float res = 0;
    while (phi < 2.0*M_PI)
    {
        float x = r*cos(phi);
        float y = r*sin(phi);
        if (x < 0) x = size_x + x;
        if (y < 0) y = size_y + y;
        float tmp = get_point(x, y);
        res += tmp;
        phi += 2.0 * M_PI / (float) steps;
    }
    res /= (float) steps;
    return res;
}

float center_data()
{
    float temp = delta / dx;
    float d = 2.0 * radius / dx;
    int e = d*d;
    for (int i = size_x/2; i < size_x; i++)
    {
        for (int j = size_y/2; j < size_y; j++)
        {
            int a = size_x - (i+1);
            int b = size_y - (j+1+temp);
            int c = a*a+b*b;
            if (c<e) ofile << "0" << " ";
            else ofile << field[i+j*size_x] << " ";
        }
        for (int j = 0; j < size_y/2; j++)
        {
            int a = size_x - (i+1);
            int b = j-temp;
            int c = a*a+b*b;
            if (c<e) ofile << "0" << " ";
            else ofile << field[i+j*size_x] << " ";
        }
    ofile << endl;
    }
    for (int i = 0; i < size_x/2; i++)
    {
        for (int j = size_y/2; j < size_y; j++)
        {
            int a = i;
            int b = size_y - (j+1+temp);
            int c = a*a+b*b;
            if (c<e) ofile << "0" << " ";
            else ofile << field[i+j*size_x] << " ";
        }
        for (int j = 0; j < size_y/2; j++)
        {
            int a = i;
            int b = j-temp;
            int c = a*a+b*b;
            if (c<e) ofile << "0" << " ";
            else ofile << field[i+j*size_x] << " ";
        }
    ofile << endl;
    }
    return 0;
```

```
}

int main(int argc, char** argv)
{
    vector<string> v(argv, argv + argc);
    string fname = v[1];
    read_field(fname);
    //string fname2 = "data_" + fname;
    //string fname2 = "centered_" + fname;
    string fname2 = "gr_" + fname;
    ofile.open(fname2.c_str(), ios::out);
    //ofile << "# " << size_x << " " << size_y << " " << dx << " " << rho0 << " " << radius << "
        " << delta << endl;

    //float res = get_contact(1000, radius, delta);
    //float res = center_data();
    //cout << 2*delta << "      " << res << endl;

    for (float i=0.0; i<4.0; i+=0.01)
    {
        float temp = get_gr(100, i);
        ofile << i << "\t" << temp << endl;
    }

    ofile.close();

    /*for (float x = 2*radius; x < 6.0; x += 5.0/1000.0)
    {
        float tmp = get_gr(1000, x);
        cout << x << "  " << tmp << endl;
    }*/
}
```

Listing A.6: Hilfsklasse zur Kraftberechnung sowie zur Erstellung von Ableitungen und zur Interpolation zwischen Gitterpunkten

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[Bog46] N. N. Bogoliubov: *Kinetic Equations*, J. Phys. URSS 10, 265, 1946.

[Mor61] T. Morita, K. Hiroike: *A new approach to the theory of classical fluids*, Prog. Theor. Phys. Osaka 25, 4, 537, 1961.

[Dom62] C. de Dominicis: *Variational Formulations of Equilibrium Statistical Mechanics*, J. Math. Phys. 3, 5, 983, 1962.

[Sti62] F. H. Stillinger, F. P. Buff: *Equilibrium statistical mechanics of inhomogeneous fluids*, J. Chem. Phys. 37, 1, 1962.

[Leb63] J. L. Lebowitz, J. K. Percus: *Statistical thermodynamics of nonuniform fluids*, J. Math. Phys. 4, 1, 116, 1963.

[Hoh64] P. Hohenberg, W. Kohn: *Inhomogeneous electron gas*, Phys. Rev. 136, 3B, 864, 1964.

[Per64] J. K. Percus, G. Stell, H.L. Frisch, J. L. Lebowitz: *The Equilibrium Theory of Classical Fluids*, 1964.

[Coo65] J. W. Cooley, J. W. Tukey: *An algorithm for the machine computation of the complex Fourier series*, Math. Computation 19, 297-301, 1965.

[Koh65] W. Kohn, L. J. Sham: *Self-Consistent Equations Including Exchange and Correlation Effects*, Phys. Rev. 140, 4A, 1133, 1965.

[Mer65] N. D. Mermin: *Thermal Properties of the Inhomogeneous Electron Gas*, Phys. Rev. 137, 5A, 1441, 1965.

[Per76] J. K. Percus: *Equilibrium state of a classical fluid of hard rods in an external-field*, J. Stat. Phys. 15, 6, 505, 1976.

[Wer76] M. S. Wertheim: *Correlations in liquid-vapor interface*, J. Chem. Phys. 65, 6, 2377, 1976.

[Eva79] R. Evans: *The nature of the liquid-vapour interface and other topics in the statistical mechanics of non-uniform, classical fluids*, Advances in Physics, 28, 2, 143, 1979.

[Eva80] R. Evans, T. J. Sluckin: *A density functional theory for inhomogeneous charged floids - application to the surfaces of molten-salts*, Mol. Phys. 40, 413, 1980.

[Tel80] M. M. T. da Gama, R. Evans, T. J. Sluckin: *The structure and surface-tension of the luquid-vapor interface of a model of a molten-salt*, Mol. Phys. 41, 1355, 1980.

[Slu81] T. J. Sluckin: *Applications of the density-functional theory of charged fluids*, J. Chem. Soc. Faraday Trans. II, 77, 1029, 1981.

[Rob81] A. Robledo, C. Varea: *On the relationship between the density functional formalism and the potential distribution-theory for nonuniform fluids*, J. Stat. Phys. 26, 3, 513, 1981.

[Tar84] P. Tarazona, R. Evans: *A simple density functional theory for inhomogeneous liquids - wetting by gas at a solid liquid interface*, Mol. Phys 52, 4, 847-857, 1984.

[Cur85] W. A. Curtin, N. W. Ashcroft: *Weighted-density-functional theory of inhomogeneous liquids and the freezing transition*, Phys. Rev. A, 32, 5, 2909, 1985.

[Han86] J. P. Hansen, I. R. McDonald: *Theory of Simple Liquids*, Academic Press, London, 1986.

[Bar87] J. L. Barrat, J. P. Hansen, G. Pastore, E. M. Waisman: *Density functional theory of soft sphere freezing*, J. Chem. Phys. 86, 11, 6360, 1987.

[Cur87] W. A. Curtin: *Density-functional theory of the solid-liquid interface*, Phys. Rev. Lett 59, 1228-1231, 1987.

[Med87] L. Mederos, P. Tarazona, G. Navascues: *Density-functional approach to phase-transitions of submonolayer films*, Phys. Rev. B 35, 7, 3376, 1987.

[Bal88] P. C. Ball, R. Evans: *The density profile of a confined fluid - comparison of density functional theory and simulation*, Mol. Phys 63, 1, 159, 1988.

[Jon89] R. Jones, O. Gunnarsson: *The density functional formalism, its applications and prospects*, Rev. Mod. Phys. 61, 3, 689, 1989.

[Par89] R. G. Parr, W. Yang: *Density Functional Theory of Atoms and Molecules*, Oxford University Press, 1989.

[Ros89] Y. Rosenfeld: *Free-energy model for the inhomogeneous hard-sphere fluid mixture and density-functional theory of freezing*, Phys. Rev. Lett 63, 980 1989.

[Kie90] E. Kierlik, M. L. Rosinberg: *Free-energy density functional for the inhomogeneous hard-sphere fluid - application to interfacial adsorption*, Phys. Rev. A 42, 3382, 1990.

[Ros90] Y. Rosenfeld: *Free-energy model for the inhomogeneous hard-sphere fluid in D dimensions: Structure factors for the hard-disk (D=2) mixtures in simple explicit form*, Phys. Rev. A, 42, 10, 1990.

[Kie91] E. Kierlik, M. L. Rosinberg: *Density-functional theory for inhomogeneous fluids - adsorption of binary-mixtures*, Phys. Rev. A 44, 5025, 1991.

[Ros93] Y. Rosenfeld: *Free-energy model for inhomogeneous fluid mixtures - yukawa-charged hard-spheres, general interactions, and plasmas*, J. Chem. Phys. 98, 10, 8126, 1993.

[Ros96] Y. Rosenfeld: *Geometrically based density-functional theory for confined fluids of asymmetric ("complex") molecules*, Chemical applications of density-functional theory, ACS Symposium Series, 629, 198, 1996.

[Cue97] J. A. Cuesta, Y. Martinez-Raton: *Dimensional crossover of the fundamental-measure functional for parallel hard cubes*, Phys. Rev. Letters, 78, 3681, 1997.

[Gon97] A. Gonzalez, J. A. White, R. Evans: *Density functional theory for hard-sphere fluids: A generating function approach*, J. Phys. Cond. Mat., 9, 11, 2375, 1997.

[Ros97] Y. Rosenfeld, M. Schmidt, H. Löwen, P. Tarazona: *Fundamental-measure free-energy density functional for hard spheres: Dimensional crossover and freezing*, Phys. Rev. E 55, 4245, 1997.

[Tar96] P. Tarazona, Y. Rosenfeld: *From zero-dimension cavities to free-energy functionals for hard disks and hard spheres*, Phys. Rev. E, 55, 5, 1996.

[Tar97] P. Tarazona, Y. Rosenfeld: *Fundamental-measure free-energy density functional for hard spheres: Dimensional crossover and freezing*, Phys. Rev. E 55, R4873, 1997.

[Fri98] M. Fringo, S.G. Johnson: *FFTW: An Adaptive Software Architecture for the FFT*, Proc. ICASSP 3, 1381, 1998.

[Fri98] M. Frigo, S. G. Johnson: *FFTW: An Adaptive Software Architecture for the FFT*, ICASSP conference proceedings vol 3, 1381-1384, 1998.

[Ros98] Y. Rosenfeld, P. Tarazona: *Free energy density functional from 0D cavities*, Proceedings of the NATO Advanced Study Institute, 1998.

[Tar98] P. Tarazona, Y. Rosenfeld: *New Approaches to Problems in Liquid State Theory*, Proceedings of the NATO Advanced Study Institute, 1998.

[Tor98] T. M. Truskett, S. Torquato, S. Sastry, P. G. Debenedetti, F. H. Stillinger: *Structural precursor to freezing in the hard-disk and hard-sphere systems*, Phys. Rev. E, 58, 1998.

[Tru98] T. M. Truskett, S. Torquato, S. Satry, P. G. Debenedetti, F. H. Stilinger: *Structural precursor to freezing in the hard-disk and hard-sphere systems*, Phys. Rev. E, 58, 3, 1998.

[Fri99] M. Frigo: *A Fast Fourier Transform Compiler*, Proceedings of the ACM SIGPLAN Conferance PLDI Atlanta, Georgia, 1999.

[Lei99] C. E. Leiserson, M. Frigo, H. Prokop, S. Ramachandran: *Cache-Oblivious Algorithms*, FOCS 99, 1999.

[Opp99] A. V. Oppenheim, R. W. Schafer, J. R. Buck: *Discrete-Time Signal Processing*, Upper Saddle River, NJ: Prentice-Hall, 1999.

[Zah99] K. Zahn, R. Lenke, G. Maret: *Two-stage melting of paramagnetic colloidal crystals in two dimensions*, Phys. Rev. Lett 82, 2721, 1999.

[Sch00] M. Schmidt, H. Löwen, J. M. Brader, R. Evans: *Density functional for a colloid-polymer mixture*, Phys. Rev. Lett 85, 1934, 2000.

[Tar00] P. Tarazona: *Density functional for hard sphere crystals: A fundamental measure approach*, Phys. Rev. Letters 84, 694, 2000.

[Sch02] M. Schmidt, H. Löwen, J. M. Brader, R. Evans: *Density functional theory for a model colloid-polymer mixture: bulk fluid phases*, J. Ühys. Cond. Mat, 14, 9353, 2002.

[Arc03] A. J. Archer, R. Evans: *Solvent-mediated interactions and solvation close to fluid-fluid phase separation: A density functional treatment*, J. Chem. Phys. 118, 21, 9726-9746, 2003.

[Eis05] C. Eisenmann, U. Gasser, P. Keim: *Pair interaction of dislocations in two-dimensional crystals*, Phys. Rev. Lett 95, 185502, 2005.

[Fri05] M. Frigo, S. G. Johnson: *The Design and Implementation of FFTW3*, Proceedings of the IEEE93, 216-231. 2005.

[Han06] J-P. Hansen, I. R. McDonald: *Theory of Simple Liquids*, Academic Press, 2006.

[Mat06] D. Matuszak, G. L. Aranovich, M. D. Donohue: *Modeling fluid diffusion using the lattice density functional theory approach: counterdiffusion in an external field*, Physical Chemistry Chem. Phys. 8, 14, 1663-1674, 2006.

[Arc07] A. J. Archer, P. Hopkins, M. Schmidt: *Dynamics in inhomogeneous liquids and glasses via the test particle limit*, Phys. Rev. E, 75, 2007.

[Joh07] S. G. Johnson, M. Frigo: *A modified split-radix FFT with fewer arithmetic operations*, IEEE Trans. Signal Processing 55, 111-119, 2007.

[Ake08] T. Akenine-Möller, E. Haines, and N. Hoffman: *Real-Time Rendering*, A. K. Peters Ltd, 2008.

[Bra08] J. M. Brader: *Structural precursor to freezing: An integral equation study*, J. Chem. Phys 128, 10, 104503, 2008.

[Gar08] V. Garcia, E. Debreuve, M. barlaud: *Fast k nearest neighbor search using GPU*, Proceedings of the CVPR Workshop on Computer Vision on GPU, 2008.

[Mar08] Y. Martinez-Raton, J. A. Capitan, J. A. Cuesta: *Fundamental-measure density functional for mixtures of parallel hard cylinders*, Phys. Rev. E, 77, 51205, 2008.

[Mik08] J. Mikhael, J. Roth, L. Helden, C. Bechinger: *Archimedean-like tiling on decagonal quasicrystalline surfaces*, Nature 454 501, 2008.

[Sch08] M. Schmiedeberg, H. Stark: *Colloidal Ordering on a 2D Quasicrystalline Substrate*, Phys. Rev. Lett 101, 218302, 2008.

[CUD09] NVidia: *NVIDIA CUDA C Programming Guide 2.3*, `http://developer.nvidia.com/cuda-toolkit-23-downloads`, 2009.

[Bit09] Bitcoin Project: *Why a GPU mines faster than a CPU*, `https://en.bitcoin.it/wiki/Why_a_GPU_mines_faster_than_a_CPU`, 2009.

[Eva09] R. Evans: *Density Functional Theory for Inhomogeneous Floids I*, University of Bristol, BS8 1TL, 2009.

[Hu09] G. hu, J. Ma, B. Huang: *Password Recovery for RAR Files Using CUDA*, IEEE Dependable, Autonomic and Secure Computing, 486-490, 2009.

[Maz09] S. Mazoyer, F. Ebert, G. Maret, P. Keim: *Dynamics of particles and cages in an experimental 2D glass former*, EPL 88, 66004, 2009.

[Nag09] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A. V. Veidenbaum: *A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors*, Neural Networks 22, 791, 2009.

[Pre09] T. Preis, P. Virnau, W. Paul, J. J. Schneider: *GPU accelerated Monte Carlo simulations of the 2D and 3D Ising model*, Journal of Computational Physics 228, 12, 4468-4477, 2009.

[Pvw09] T. Preis, P. Virnau, W. Paul, J. J. Schneider: *Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets*, New J. Phys. 11, 93024, 2009.

[Way09] R. B. Way, L. J. Greenhill, F. H. Briggs: *A GPU-based Real-time Software Correlation System for the Murchison Widefield Array Prototype*, Publications of the Astronomical Society of the Pacific 121, 857, 2009.

[Blo10] B. Block, P. Virnau, T. Preis: *Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising Model*, Computer Physics Communications 181, 9, 1549-1556, 2010.

[Han10] H. Hansen-Goos, K. R. Mecke: *Tensoial density functional theory for non-spherical hard-body fluids*, J. Phys. Condens. Matter 22, 364107, 2010.

[Mic10] D. Michea, D. Komatitsch: *Accelterating a three-dimensional finite-difference wave propagation code using GPU graphics cards*, Geophysical Journal International 182, 389, 2010.

[Oet10] M. Oettel, S. Görig, A. Härtel, H. Löwen, M. Radu, T. Schilling: *Free energies, vacancy concentrations, and density distribution anisotropies in hard-sphere crystals: A combined density functional and simulation study*, Phys. Rev. E, 82, 51404, 2010.

[Put10] V. B. Putz, J. Dunkel, J. M. Yeomans: *CUDA simulations of active dumbbell suspensions*, Chem. Phys. 375, 557, 2010.

[Rot10] R. Roth: *Fundamental measure theory for hard-sphere mixtures: a review*, J. Phys. Condens. Matter 22, 63102, 2010.

[Dwa11] W. S. B. Dwandaru, M. Schmidt: *Variational principle of classical density functional theory via Levy's constrained search method*, Phys. Rev. E, 83, 2011.

[Haj11] D. Hajnal, M. Oettel, R. Schilling: *Glass transition of binary mixtures of dipolar particles in two dimensions*, J. Noncryst Solids 357, 302, 2011.

[Law11] A. D. Law, D. M. A. Buzza, T. S. Horozov: *Two-dimentional Colloidal Alloys*, Phys. Rev. Lett 106, 128302, 2011.

[Oet11] M. Oettel: *Fundamental measure theory for hard disks*, Johannes-Gutenberg-Universität Mainz, 2011.

[Obe11] T. Oberhuber, A. Suzuki, J. Vacata, V. Zabka: *Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods*, Journal of Math-for-Industry, 3, 73-79, 2011.

[Par11] S. J. Park, J. A. Ross, D. R. Shires, D. A. Richie, B. J. Henz, L. H. Nguyen: *Hybrid Core Acceleration of UWB SIRE Radar Signal Processing*, IEEE Transactions on Parallel and Distributed Systems 22, 46, 2011.

[Bur12] C. S. Burrus, M. Frigo, S. G. Johnson: *Implementing FFTs in Practice*, Connexions module m16336, 2012.

[CUD12] NVidia: *NVIDIA CUDA C Programming Guide 4.2*, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA\_C\_Programming\_Guide.pdf`, 2012.

[CUF12] NVidia cuFFT library: *CUFFT Library User Guide*, `http://developer.nvidia.com/cufft`, 2012.

[OCL12] Khronos Group: *OpenCL - The open standard for parallel programming of heterogeneous systems*, `http://www.khronos.org/opencl/`, 2012.

[Rot12] R. Roth, K. Mecke, M. Oettel: *Communication: Fundamental measure theory for hard disks: Fluid and solid*, J. Chem. Phys. 136, 8, 81101, 2012.

[Sta12] Statistica GmbH: `http://de.statista.com/statistik/faktenbuch/77/a/branche-industrie-markt/filmindustrie/`

filmindustrie-umsatz/, http://de.statista.com/statistik/faktenbuch/128/a/branche-industrie-markt/it-telekommunikation/softwareindustrie/, http://de.statista.com/statistik/faktenbuch/116/a/branche-industrie-markt/musikindustrie/musikindustrie-umsatz/, 2012.

[Xu12] X. Xu, D. E. Cristancho, S. Costeux: *Density-Functional Theory for Polymer-Carbon Dioxide Mixtures*, Industrial and Engineering Chemistry Research 51, 9, 3832, 2012.