Institut für Technische Informatik

Abteilung Rechnerarchitektur

Universität Stuttgart
Pfaffenwaldring 47
D - 70569 Stuttgart

Studienarbeit Nr. 2347

Parallele Partikelsimulation auf GPGPU-Architekturen zur Evaluierung von Apoptose-Signalwegen

Alexander Schöll

Studiengang:	Informatik
Prüfer:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer:	Dipl.-Inform. Claus Braun
begonnen am:	1. September 2011
beendet am:	1. März 2012
CR-Klassifikation:	C.1.4, C.4, D.1.3, I.6.3, J.3

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, 1. März 2012

Inhaltsverzeichnis

1. Motivation

Im Bereich der Krebsforschung liegt ein zentraler Forschungsschwerpunkt in der Zell- und Tumorbiologie. Eine Krebserkrankung ist die Folge eines biologischen Vorganges innerhalb einer Zelle des menschlichen Organismus, der das Erbmaterial im Zellkern schädigt.

Die Erzeugung und das Wachstum von Zellen, wie sie in menschlichen Organismen vorkommen, ist ein fortwährender Prozess, der die Grundlage eines jeden mehrzelligen Organismus darstellt. Dieser Vorgang wird Zellproliferation genannt und durch den Vorgang des Zelltodes reguliert. Dadurch entsteht ein Ausgleich zwischen Zellentstehung und Zellsterben. Der Zustand des Ausgleiches wird als Homöostase beschrieben. [1][4]

Zellen, deren Erbmaterial beschädigt wurde, können durch die biologischen Prozesse der DNA-Reparatur oder durch den Zelltod vor einer so genannten Mutation bewahrt werden. Wird die Mutation der Zelle nicht verhindert, kann sich die Mutation durch den fortlaufenden Prozess der Zellteilung vervielfältigen. Die dabei entstehenden Zellen besitzen durch die Mutation der Mutterzelle instabile Erbmaterialien, wodurch die Wahrscheinlichkeit für weitere Mutationen steigt. Die zu diesem Zeitpunkt abnormal entstandenen Zellen, die sich lokal häufen, werden Tumore genannt. Tumore werden in gutartige (benigne) und bösartige (maligne) Tumore unterschieden. Eine Zelle wird „bösartig“, wenn die Eigenschaften der Zelle dahingehend mutieren, dass die Zellteilung kontinuierlich fortgesetzt wird, während gleichzeitig der Vorgang des Zelltodes deaktiviert ist. Diese Zellen sind in Folge dessen „unsterblich“.

Beim Auftreten der bösartigen Tumorzellen ist der Vorgang der Krebsentstehung, der auch Karzinogenese genannt wird, weit fortgeschritten. Die Tumorzellen akkumulieren weitere Defekte im Erbmaterial. Tumorzell-Agglomerationen wachsen laufend weiter und bilden Tochter-Agglomerationen im gesamten Organismus, die als Metastasen bezeichnet werden. Die Tumorzellen sind in diesem Zustand fähig, gesundes Zellmaterial zu verdrängen, was in weiterer Folge zum Tod des Organismus führt. [2]

Ein wesentliches Merkmal einer Krebsentstehung ist der gestörte Zustand der Homöostase zugunsten der Zellproliferation, wobei das Auftreten des Zelltodes benachteiligt ist. Der Organismus kennt mehrere Möglichkeiten zur Gegenmaßnahme. Diese werden unter dem Begriff des Vorgangs der Antikarzinogenese zusammengefasst. Diese umfassen [3]:

- DNA-Reparatur
- Chromatin Organisation
- „Transaction Coupling“
- Zellzyklus Stop
- Apoptose

Der Vorgang des Zelltodes wird von den biologischen Vorgängen Apoptose und Nekrose durchgeführt. Der Begriff der **Apoptose** lässt sich aus dem Griechischen frei mit dem Bild des „Laubfall im Herbst“ übersetzen. Die Apoptose ist ein Mechanismus, der den programmierten Zelltod beschreibt. Bei diesem Vorgang wird eine Zelle in mehreren Schritten „demontiert“. Es findet eine kontrollierte Zerstörung der Zelle statt, wobei benachbarte Zellen nicht in Mitleidenschaft gezogen werden. Die Unterdrückung der Apoptose ist eine Bedingung für die Entstehung einer Krebserkrankung, wie sie zuvor beschrieben wurde. Die „Unsterblichkeit“ der Tumorzellen wird durch eine so genannte neoplastische Transformation ausgelöst, die zur Aussetzung des Apoptoseprogrammes führt.

Neben dem Zusammenhang mit Krebserkrankungen spielt die Apoptose auch bei anderen Erkrankungen, wie zum Beispiel AIDS, eine tragende Rolle. Hierbei führt die übermäßige Ausführung des Apoptoseprogrammes zur Aufhebung der Fähigkeit einer Immunabwehr, da die Apoptose die Anzahl der T-Zellen-Klone verringert [4].

Die Grundlagenforschung zur Apoptose - ihre Auslösung, die Ausführungsschritte und der Abschluss – sind zentraler Bestandteil der Krebsforschung. Zur Bewahrung der Homöostase liegt ein besonderes Augenmerk auf den Rahmenbedingungen des Ausbleibens der Apoptose. In diesem Zusammenhang liegt das Hauptinteresse auf der Erforschung der Ereignisse, die die Apoptose einer Zelle aktivieren. Die Apoptose wird über zwei verschiedene Initiatonsphasen aktiviert. Hierbei existieren ein extrinsischer und ein intrinsischer Signalweg. Im Rahmen des Cluster of Excellence in Simulation Technology (EXC 310/1) der Universität Stuttgart wurde ein Modell für die Untersuchung der extrinsischen Signalwege der Apoptose per Simulation entwickelt [5]. Bei diesem Signalweg verbinden sich Rezeptoren zusammen mit Liganden zu Clustern auf der Zellmembran, wodurch die Apoptose ausgelöst wird.

Abbildung 1: Visualisiertes Cluster aus Rezeptoren (grün) und Liganden (blau)

Die Ergebnisse der Simulation im Rahmen der genannten Arbeit zeigen einen sehr hohen Rechenzeitbedarf auf. Die Simulation einer halben Sekunde benötigt eine Berechnungszeit von 28 Tagen.[1] Das Modell wurde unter Verwendung der Programmiersprache C implementiert und auf einer CPU-Architektur ausgeführt (Intel Core i7). Die vorliegende Studienarbeit hat das Ziel die Berechnungszeit durch eine Abbildung des Simulationsmodells auf eine GPGPU-Architektur im Umfang dahingehend zu verringern, dass eine Erzeugung von Simulationsergebnissen in einem zeitlich handhabbaren Rahmen ermöglicht wird.

[1] Simulation von 1000 Rezeptoren und 1000 Liganden auf einer Fläche von je 1µm Breite und Länge bei einem Abstand von zwischen 10^{-9}s den Zeitschritten.

Die Notwendigkeit der Beschleunigung der Simulation stammt aus der Anforderung einer sehr hohen Anzahl von Simulationsdurchläufen. Dies begründet sich in der stochastischen Grundlage des Modells. Nimmt man die stochastische Anforderung von mindestens 10.000 Simulationsdurchläufen mit einer Simulationsdauer von einer Sekunde an, wächst die Berechnungszeit auf 1534 Jahre.

2. Entwicklungsstand der Evaluierung von Apoptose-Signalwegen

2.1 Der programmierte Zelltod - Apoptose

Die Apoptose ist eine wesentliche Prozedur im Stoffwechsel von mehrzelligen Organismen. Wie im Kapitel 1 („Motivation“) erläutert, ist ihr Zweck die Bewahrung der Homöostase – dem Gleichgewicht aus Zellwachstum und Zelltod. Die Apoptose ist eine Form des programmierten Zelltodes. Durch den Vorgang der Apoptose wird beispielsweise in fortgeschrittenen Entwicklungszuständen nahezu die Hälfte aller Neuronen im Gehirn eines Organismus beseitigt. Darüber hinaus spielt die Apoptose eine tragende Rolle in der Regulation und Beseitigung von defekten Zellen. Durch einen Defekt können weitere Schäden im Organismus ausgelöst werden. Dies kann, wie bereits erläutert, im weiteren Verlauf zu einer Krebserkrankung führen.

Zu Beginn der Apoptose einer Zelle kann eine Schrumpfung der Zelle beobachtet werden. Sie zeigt Deformationen auf und verliert die Haftung zu umgebenden Zellen. Das Chromatin kondensiert und sammelt sich an der Hülle des Zellkernes. Die Zellmembran löst sich in Form von Vesikeln auf. In einem letzten Schritt teilt sich die Zelle in Teilstücke auf, die eine geschlossene Membran aufweisen – diese werden apoptotische Körper genannt. Im weiteren Verlauf werden diese apoptotischen Körper von den Fresszellen (Leukozyten – weiße Blutkörperchen) des Immunsystems beseitigt. Die Apoptose unterscheidet sich von der Nekrose – einer weiteren Form des Zelltodes – durch den Umstand, eine „kontrollierte Demontage“ der Zelle durchzuführen. Beim Vorgang der Nekrose kann der Inhalt der Zelle unkontrolliert entweichen und andere Zellen beschädigen. Dabei kann eine Entzündung des umgebenden Gewebes beobachtet werden.

Die Apoptose ist eine Folge von verschiedenen biochemischen Ereignissen. Ein Ereignis kann hierbei außerhalb oder innerhalb der Zelle erfolgen. Auf der Zellenoberfläche kann eine so genannte Ligation von Rezeptoren stattfinden. Bei dieser Ligation verknüpfen sich Rezeptorsegmente an ihren Enden unter dem Einfluss eines Enzyms, welches als „Ligase“ bezeichnet wird. Darüber hinaus kann ein Defekt in der DNA während dem Prozess der DNA-Reperatur die Apoptose auslösen. Daneben können ein Mangel an so genannten Überlebenssignalen, widersprechende Signale des Zellstoffwechsels oder die Entwicklung von so genannten „Death Signals“ zur Auslösung des Vorgangs der Apoptose führen.

Im Rahmen des Simulationsmodells, welches in dieser Arbeit im Mittelpunkt der Untersuchung steht, wird die Ligation von Rezeptoren auf der Zellmembran untersucht.

Dieser Vorgang wird als „extrinsischer Weg – Typ 1“ bezeichnet. Hierbei wird die Apoptose von so genannten „Death Rezeptoren“, die sich auf der Zellmembran befinden, ausgelöst, nach dem sie sich im Rahmen einer Ligation an ihre zugehörigen Liganden gebunden haben. „Death Rezeptoren“ gehören zur Gruppe der Tumornekrosefaktor-Rezeptor-Superfamilie (TNFR). Die Mitglieder dieser Gruppe sind von einer hohen Genauigkeit bei der Verbindung mit ihren zugehörigen Liganden geprägt. Durch diesen Vorgang wird der „Death Rezeptor“ aktiviert, woraufhin sich Moleküle abspalten, die ein Signalkomplex zur Einleitung des Zelltodes auslösen. [11]

2.2 Modelle zur Evaluierung der Apoptose-Signalwege

Neben dem erwähnten Modell zur Evaluierung der Apoptose-Signalwege, welches im Mittelpunkt dieser Arbeit steht und im Kapitel 4 („Das Modell zur Evaluierung der Apoptose-Signalwege“) erläutert wird, gibt es eine Reihe weiterer Modelle, die sich mit dem Entstehen und dem Verhalten von Bindungen zwischen Rezeptoren und Liganden befassen. Die Auswahl der Modelle, die im Folgenden vorgestellt wird, konzentriert sich auf die Eigenschaft von Anwendungsmöglichkeiten im Bereich zur Evaluierung der Apoptose-Signalwege.

Mit einem thermodynamischen Ansatz stellen Guo und Levine [6] ein Modell vor, in dessen Mittelpunkt ein Gitter steht. In einer Abbildung werden die Rezeptoren direkt auf Gitterzellen abgebildet. Dabei entspricht die Gittergröße der minimalen Distanz, mit der sich Rezeptoren annähern können. Innerhalb des Modells wird ein System modelliert, mit dem gezeigt wird, dass die Entstehung von Bindungen für gewisse Bereiche von Dichten der Rezeptoren begünstigt wird. Hierzu werden die Prinzipien der hamiltonischen Mechanik verwendet. Mittels einer numerischen Monte-Carlo-Simulation wird die Entstehung von Bindungen untersucht. Dieses Modell ignoriert Interaktionen, an denen mehr als zwei Rezeptoren beteiligt sind und macht keine Aussagen zu Schwellenwerten von Liganden- oder Bindungsstrukturen.

In einem weiteren gitterbasierten Ansatz stellen Gopalakrishnan, Forsten-Williams, Nugent und Täuber [7] ein Modell vor, das das Bindungsverhalten von Rezeptoren und Liganden untersucht. Besonderes Augenmerk liegt hierbei auf der Regulierung einer Neuverbindung von Liganden an Rezeptoren, in Abhängigkeit von der räumlichen Verteilung der Rezeptorproteine. Dabei wird ein unendlich großer Raum angenommen, um eine Approximation an die Oberfläche der Zellmembran entwickeln zu können. Die Modellierung der Bewegung von Liganden wird durch eine dreidimensionale Zufallsbewegung abgebildet. In einer numerischen Monte-Carlo-Simulation wird das entwickelte Modell untersucht. Die Autoren bemängeln in ihrem Fazit eine fehlende Exaktheit der Analyse. Als weitere Beobachtung wird das lose Verhalten von Eins-zu-Eins-Bindungen zwischen Rezeptoren und Liganden in den frühen Zeitabschnitten der Simulation genannt, wobei auf die Vermutung verwiesen wird, dass diese Bindungen über einen „biologisch relevanten Zeitabschnitt signifikanten Einfluss“ haben können.

2.3 Abbildungen von Simulationsmodellen

Neben der Betrachtung des Modells ist auch eine Betrachtung verwandter Abbildungslösungen von Simulationsverfahren auf GPGPU-Architekturen von Bedeutung. In verschiedenen Arbeiten wurden bereits Vorteile durch eine reduzierte Berechnungszeit von Simulationen beobachtet, die teilweise oder vollständig auf eine GPGPU-Architektur abgebildet wurden.

In einer Abbildung der Simulation von so genannten P-Systemen [31] unter der Verwendung der Applikationsschnittstelle CUDA messen Guerro, Cecilia und Garcia [8] einen Speedup von 24x. P-Systeme oder Membran-Systeme dienen der Abstraktion biologischer Prozesse der Zellmembran. Das P-System beinhaltet neben der Struktur der Zellmembran und Molekülen eine Menge von Regeln, nach denen eine Interaktion durchgeführt wird. Die Ausführung ist nichtdeterministisch, da die Reihenfolge und Menge der angewandten Regeln im biologischen Prozess zufällig ist. Um dieser Eigenschaft begegnen zu können, werden die Regeln durch ein Zufallsprinzip angewandt. Direkt daraus folgt der Bedarf an einer großen Anzahl von Simulationsdurchläufen.

In der Simulation werden drei Simulationsarten unterschieden. Es werden eine sequentielle Ausführung auf der CPU, eine massiv parallele Ausführung auf der GPU von Teilen der Simulation und eine vollständige Ausführung auf der GPU untersucht und verglichen. Als Resultat stellen die Autoren fest, dass die größtmögliche Performance bei einer ausschließlichen Nutzung der GPU erreicht wird. Die Kommunikation mit der GPU wird von einem Overhead in der Kommunikation durch den PCI-Express-Bus beeinflusst. Die Autoren deuten im Ausblick ihrer Arbeit eine Anwendung der P-Systeme zur Untersuchung des biologischen Prozess der Apoptose an.

Eine Parallelisierung und Abbildung auf GPGPU-Architekturen von Algorithmen zur Lösung von numerischen Problemen stellen Zhou, Liepe, Sheng, Stumpf und Barnes [9] vor. In ihrer Arbeit untersuchen sie den LSODA-Algorithmus zur Lösung von gewöhnlichen Differentialgleichungen (ODE), den Euler-Maruyama-Algorithmus zur Lösung von stochastischen Differentialgleichungen (SDE) und den Gillespie-Algorithmus für Markow-Prozesse (MJP). Für die Abbildung auf eine GPGPU-Architektur wird die Applikationsschnittstelle CUDA verwendet. Als Resultat messen die Autoren Speedups von 47x (ODE), 367x (SDE) und 12x (MJP). Darüber hinaus stellen sie fest, dass ihre Vergleichsimplementierung auf der CPU unter der Vorraussetzung von kleinen Simulationsmengen eine geringere Berechnungszeit, als die Vergleichsimplementierung auf der GPU, aufweist.

Suhartanto, Yanuar und Wibisono [10] untersuchen die Abbildung des GROMACS-Softwarepaketes [28] auf GPGPU-Architekturen unter der Verwendung von CUDA. Dieses Softwarepaket bietet Funktionalitäten zur Simulation und Auswertung molekulardynamischer Prozesse. Hierbei werden Newtonsche Bewegungsgleichungen auf einer beliebigen Anzahl von Molekülen angewandt. Die Autoren simulieren Proteine mit einer dynamischen Bewegung und einem aufwendigen Bindungsverhalten. Vorrangiges Ziel dieser Simulation ist die Untersuchung der Bedeutung von Proteinen.

Im Rahmen ihrer Simulation werden unter anderem Viren untersucht, die über eine Folge von Zwischenereignissen für das Ausbleiben der Apoptose verantwortlich sein können. Die Abbildung der Simulation auf eine GPGPU-Architektur liefert als Resultat einen messbaren Speedup von 11x bis 12x.

3. Modell zur Evaluierung von Apoptose-Signalwegen

Basierend auf dem Simulationsmodell, welches im Rahmen dieser Studienarbeit auf eine GPGPU-Architektur abgebildet wurde, werden in diesem Kapitel Eigenschaften des Modells und der dazugehörigen Simulation vorgestellt. Hierbei liegt das Hauptaugenmerk auf den Informationen, die für die spätere Implementierung des Simulationsmodells auf einer GPGPU-Architektur von Belang sind.

3.1 Mathematische Grundlagen

Zur Simulation der Clusterbildung von Death-Rezeptoren, die die äußeren proapoptotischen Signalwege anregen, führt die Simulationsanwendung umfangreiche Partikelsimulationen durch. Innerhalb der Simulation wird ein System von stochastischen Differentialgleichungen der Form

$$d\bar{x}_i = 6\mu^2 \bar{F}_i(\bar{\xi}, \bar{\varphi})d\bar{t} + \sqrt{2}\mu d\tilde{W}_{trans,\bar{t},i} \tag{1}$$

und

$$d\varphi_i = \kappa\mu^2\zeta^2 g_i(\bar{\xi}, \bar{\varphi})d\bar{t} + \sqrt{2}\mu\zeta dW_{rot,\bar{t},i} \tag{2}$$

gelöst [5]. Dieses Gleichungssystem beschreibt die Bewegung der so genannten Rezeptoren- und Ligandenpartikel auf der Zellmembran. Falls *N* Rezeptoren und *M* Liganden betrachtet werden, müssen *N* + *M* gekoppelte stochastische Differentialgleichungen gelöst werden. Im Wesentlichen wird eine Euler-Maruyama-Approximation [33] zur Lösung des Systems angewandt.

Die Kräfte zwischen den Partikeln werden durch Lennard-Jones-Potentiale [32] der Form

$$\bar{V}_{LJ} = \left(\left(\frac{\bar{\sigma}_{LJ}}{\bar{r}} \right)^{2n} - \left(\frac{\bar{\sigma}_{LJ}}{\bar{r}} \right)^{n} H(\delta - |\varphi - \varphi_0|) \right) \tag{3}$$

modelliert. Die Funktion *H* beschreibt die Heaviside-Funktion. Der Parameter ξ beschreibt die Größe der Partikel – der Exponent des Potentials ist typischerweise n = 6. [5]

3.2 Simulationseigenschaften und -anforderungen

Innerhalb jedes Simulationszeitschrittes werden die Kräfte und Drehmomente zwischen allen Partikeln berechnet und mögliche Bindungen evaluiert. Während der Simulation sind die Partikel in der Lage Cluster zu formen und wieder aufzulösen. In jedem tausendsten Schritt wird das System festgehalten (*sampling*) und existierende Cluster bestimmt. Die Simulation erfordert sehr kleine Zeitschritte in der Größenordnung von

$1.0e^{-9}$ Sekunden. Für aussagekräftige Ergebnisse wird eine minimale Simulationsdauer von mehreren Minuten bis zu Stunden benötigt. Außerdem wird eine hinreichend große Anzahl von Simulationsdurchgängen (mindestens 1000) benötigt, um die Ergebnisse zu mitteln. Dies ist wegen der Modellierung als stochastischer Prozess erforderlich.

Die Simulation benutzt zwei unterschiedliche Arten von Partikeln – Rezeptoren und Liganden. Die Rezeptorpartikel werden weiter unterschieden in Monomere, Dimere und Trimere, die unterschiedliche Arten und jeweils unterschiedliche Anzahlen von Bindungsregionen (siehe Abbildung 2) besitzen, in denen sie an andere Partikel gebunden werden können:

- Monomere besitzen zwei Bindungsregionen und können an Liganden oder andere Rezeptoren gebunden werden,
- Dimere besitzen ebenfalls zwei Bindungsregionen und können an zwei Liganden gebunden werden,
- Trimere besitzen drei Bindungsregionen für Liganden und
- Liganden besitzen drei Bindungsregionen für Rezeptoren

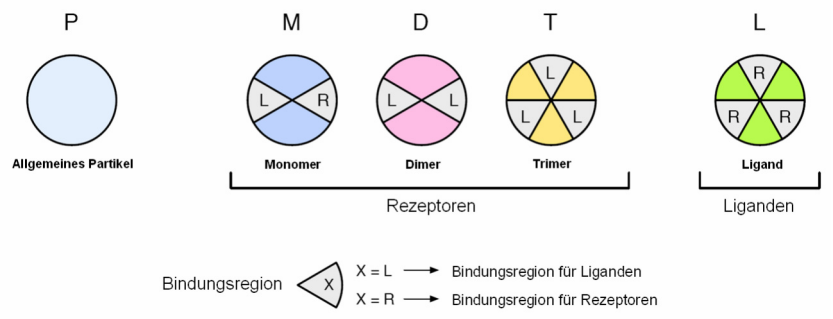


Abbildung 2: Partikeltypen samt Bindungsregionen

3.3 Übersicht der Algorithmen

Grundlage dieser Studienarbeit sind Algorithmen aus der Simulationsimplementierung, welche im Kapitel „Motivation“ eingeführt wurde [5]. In dieser Implementierung wird die Simulation mit *N* Rezeptoren und *M* Liganden durchgeführt. Algorithmus 1 zeigt die erste Ebene der Simulation. Die Routinen compute_force() und compute_moment() werden auf allen betrachteten Partikeln in jedem Zeitschritt angewandt. compute_force(ligands, receptors) berechnet für jeden Ligandenpartikel die dazugehörende Kraft zu allen Rezeptorpartikeln im System. Algorithmus 2 beschreibt die Berechnung der Kräfte zwischen Partikeln für einen Monomer und einen Liganden. Algorithmus 3 beschreibt die Berechnung des Drehmoments basierend auf dem Beispiel eines Monomer und eines Liganden.

Erste Ebene der Simulation

```
/* Initialisierung der Partikelkoordinaten und –ausrichtung mit zufälligen Werten */
initialize_particles(ligands, receptors)

/* Initialisierende Berechnung der Kräfte */
compute_force(ligands, ligands)                    Zwischen allen Liganden
compute_force(receptors, receptors)                Zwischen allen Rezeptoren
compute_force(ligands, receptors)          Zwischen Liganden und Rezeptoren

/* Initialisierende Berechnung der Drehmomente */
compute_moment(ligands, ligands)                   Zwischen allen Liganden
compute_moment(receptors, receptors)               Zwischen allen Rezeptoren
compute_moment(ligands, receptors)         Zwischen Liganden und Rezeptoren

while time < end time do
    sample ++
    time += delta time

    /* Berechnung von neuen zufälligen Koordinaten */
    compute coordinates(ligands)
    compute coordinates(receptors)

    // Zurücksetzung der Kräfte
    reset forces(ligands, receptors)

    // Berechnung der Interaktionen
    compute_force(ligands, ligands)                Zwischen allen Liganden
    compute_force(receptors, receptors)            Zwischen allen Rezeptoren
    compute_force(ligands, receptors)      Zwischen Liganden und Rezeptoren

    compute_moment(ligands, ligands)               Zwischen allen Liganden
    compute_moment(receptors, receptors)           Zwischen allen Rezeptoren
    compute_moment(ligands, receptors)     Zwischen Liganden und Rezeptoren

    // Erzeugung eines Zwischenergebnisses
    do if sample = 1000
        determine_clusters(ligands, receptors)
        sample = 0
    od
od
```

Algorithmus 1: Erste Ebene der Simulation

Berechnung der Kräfte (Beispiel Monomer und Ligand)

/* Berechnung des Abstandes zwischen Monomer und Ligand */
r = compute distance(particle i, particle j)

/* Winkel zwischen Partikeln */
φ_i = compute angle((i.x - j.x), (i.y - j.y))
φ_j = compute angle((j.x - i.x), (j.y - i.y))

φ_{0i} = compute angle(j.E.x, j.E.y)
φ_{0j} = compute angle(i.E.x, i.E.y)

/* Erste Komponente des Kraftvektors des Partikels i */

$$F_{i,x} = F_{i,x} + \frac{10,0}{r} * \cos(\varphi_i) * \left(2 * \left(\frac{\sigma}{r}\right)^{12} - 2^{-5} * \left(\frac{\sigma}{r}\right)^{6} \left(heaviside\left(\frac{2\pi}{3} - |\varphi_i - \varphi_{0i}|\right)\right.\right.$$
$$\left.\left. + heaviside\left(\frac{2\pi}{3} - |\varphi_i - \varphi_{0i} - \frac{2\pi}{3}|\right) + heaviside\left(\frac{2\pi}{3} - |\varphi_i - \varphi_{0i} - \frac{4\pi}{3}|\right)\right)\right)$$

/* Zweite Komponente des Kraftvektors des Partikels i */

$$F_{i,y} = F_{i,y} + \frac{10,0}{r} * \sin(\varphi_i) * \left(2 * \left(\frac{\sigma}{r}\right)^{12} - 2^{-5} * \left(\frac{\sigma}{r}\right)^{6} \left(heaviside\left(\frac{2\pi}{3} - |\varphi_i - \varphi_{0i}|\right)\right.\right.$$
$$\left.\left. + heaviside\left(\frac{2\pi}{3} - |\varphi_i - \varphi_{0i} - \frac{2\pi}{3}|\right) + heaviside\left(\frac{2\pi}{3} - |\varphi_i - \varphi_{0i} - \frac{4\pi}{3}|\right)\right)\right)$$

/* Erste Komponente des Kraftvektors des Partikels j */

$$F_{j,x} = F_{j,x} + \frac{10,0}{r} * \cos(\varphi_i) * \left(2 * \left(\frac{\sigma}{r}\right)^{12} - 2^{-5} * \left(\frac{\sigma}{r}\right)^{6} \left(heaviside\left(\frac{2\pi}{3} - |\varphi_j - \varphi_{0j}|\right)\right)\right)$$

/* Zweite Komponente des Kraftvektors des Partikels j */

$$F_{j,y} = F_{j,y} + \frac{10,0}{r} * \sin(\varphi_i) * \left(2 * \left(\frac{\sigma}{r}\right)^{12} - 2^{-5} * \left(\frac{\sigma}{r}\right)^{6} \left(heaviside\left(\frac{2\pi}{3} - |\varphi_j - \varphi_{0j}|\right)\right)\right)$$

Algorithmus 2: Berechnung der Kräfte (Beispiel Monomer und Ligand)

Berechnung der Momente (Beispiel Monomer und Ligand)

/* Winkel zwischen Partikeln */
φ_{ij} = compute_angle((j.x - i.x), (j.y - i.y))
φ_{Ei} = compute_angle(i.E.x, i.E.y)
$\varphi = \varphi_{ij} - \varphi_{Ei}$

if $\varphi > \pi$ then
    $\varphi = \varphi - 2 \cdot \pi$
else if $\varphi < -\pi$ then
    $\varphi = \varphi + 2 \cdot \pi$
fi od

φ_{ji} = compute_angle((i.x – j.x), (i.y – j.y))
φ_{Ej} = compute_angle(j.E.x, j.E.y)
$\psi = \varphi_{ji} - \varphi_{Ej}$
if ($\psi >= \frac{\pi}{3}$ && $\psi < \pi$) then
    $\psi = \psi - \frac{2}{3}\pi$
else if ($\psi >= \pi$ && $\psi < \frac{5}{3}\pi$) then
    $\psi = \psi - \frac{4}{3}\pi$
else if ($\psi >= \frac{5}{3}\pi$) then
    $\psi = \psi - 2\pi$
fi

i.moment = *i.moment* + φ
j.moment = *j.moment* + ψ

end

Algorithmus 3: Berechnung der Momente (Beispiel: Monomer und Ligand)

4. Grundlagen zur Abbildung des Simulationsmodells

4.1 GPGPU

4.1.1 Einführung

Das „**G**eneral **P**urpose Computations on **G**raphics **P**rocessing **U**nits“-Modell beschreibt das Konzept der Verwendung von Grafikprozessoren für allgemeine Anwendungen. Hierbei werden besondere Eigenschaften der GPU-Architekturen ausgenutzt, die in dieser Form in den Architekturen der Hauptprozessoren (CPU) nicht oder nur bedingt implementiert sind, da sie sich deutlich in ihrer Taxonomie der Parallelität unterscheiden. Besondere Bedeutung erlangt dieses Konzept im Bereich der nebenläufigen Programmierung und des Entwurfs von parallelen Algorithmen, da ein beachtlicher Performancevorteil im Zusammenhang mit der Abbildung auf das GPGPU-Modell beobachtet werden kann. [12][13]

4.1.2 Die Implementierung des GPGPU Modells

Eine GPU-Architektur basiert auf der „Single Instruction Multiple Data“-Klassifikation nach Flynn. Diese Klassifikation beruht auf der klassischen Funktion der GPU das Rendering von Szenen in Hardware zu implementieren. Damit wird die CPU um diese Aufgaben entlastet. Im Bereich des Rendering werden komplexe Szenen in Polygonnetze unterteilt, die sich aus Dreiecken zusammensetzen. Das Rendering der Dreiecke kann massiv parallel durchgeführt werden. Ein und dieselbe Instruktionsfolge kann auf einer sehr großen Anzahl von Parameterdaten angewandt werden. Folglich basiert die Aufgabe des Rendering auf der „Single Instruction Multiple Data“-Klassifikation.

GPGPU basiert auf der freien Programmierbarkeit von Teilen der Grafikhardware, die im klassischen Sinne das Rendering durch benutzerdefinierte Funktionen ergänzt. Durch die freie Programmierbarkeit der Grafikhardware in Kombination mit der massiven Parallelität der SIMD-GPUs eignet sich das GPGPU-Modell besonders für Parallelisierungsaufgaben, bei denen eine SIMD-Parallelität angestrebt werden kann.

Zur Anwendung des GPGPU-Konzeptes wurden Programmierschnittstellen wie

- Nvidia CUDA [14]
- AMD(ATI) FireStream [16]
- Khronos Group - OpenCL [17]

entwickelt.

Darüber hinaus wurden Architekturen für Grafikhardware entwickelt, die sich von den speziellen Anforderungen des Rendering von 3D-Grafik distanzieren und sich den Spezifikationen der heutzutage üblichen CPU-Parameter annähern. Hierbei lassen sich Annäherungen an heutzutage übliche CPU-Parameter wie der Einsatz von les- und beschreibbaren Caches sowie der Behandlung von Verzweigungen in Instruktionsflüssen aufzählen. Beispiele für solche Architekturen, die diese Ansätze verfolgen, sind:

- Nvidia Fermi
- AMD(ATI) Stream

4.2 Vergleich der Parallelität von CPU und GPU-Architekturen

Die Parallelisierungsstrategien von CPU und GPU-Architekturen lassen sich am deutlichsten anhand der **Granularität** kategorisieren.

Hauptaufgabe der klassischen CPU-Architektur ist die Verarbeitung von komplexen sequentiellen Instruktionsflüssen, wobei jede Instruktion generell auf höchstens einem Datensatz angewandt wird. Diese Architektur lässt sich nach der Taxonomie von Flynn als Single Instruction Single Data (SISD) kategorisieren. Optimierungen dieser Architektur werden mit folgenden Mitteln erreicht:

- Caches
- Pipelining (Mit Optimierungen wie Bypassing, Forwarding oder Stalling)
- Super-Skalarität

Eine Parallelisierung im Bereich der CPU-Architekturen wird allgemein als **grob-granulare Parallelität** implementiert, da die betrachtete Menge von CPU-Prozessen fast ausschließlich unabhängig voneinander auf getrennten Daten - also ressourcenfremd - zueinander arbeiten. Die Parallelisierung findet überwiegend auf Prozessebene statt. Prozesse können Threads definieren, die sich die Ressourcen mit dem Prozess teilen. Die Ausrichtung der CPU-Architekturen auf Parallelität wird mit folgenden Mitteln erreicht:

- Multi-Core-Architekturen
- Simultaneous Multithreading

Multi-Core-Architekturen sind durch unabhängige Prozessoreinheiten auf einem einzigen Die gekennzeichnet. Die Implementierung der einzelnen Kerne verfolgt im Wesentlichen die SISD-Taxonomie, wobei heutzutage üblicherweise SIMD-Erweiterungen in deren Befehlssätzen vorkommen. Ein Beispiel hierfür ist SSE (Streaming SIMD Extensions). [12][13]

Aktuelle Multi-Core-Architekturen verfügen von 6 Kernen (Intel i7-970) [18], 8 Kernen (IBM Cell) [19], über 80 Kernen (Intel Tera-Scale) [20][21] bis hin zu 800 Kernen

(Xelerator X11) [22]. Das Einsatzgebiet von Multi-Core-Prozessoren liegt in der nebenläufigen Ausführung von unabhängigen und ressourcenfremden Prozessen. Beim Einsatz von Cache werden die Prozessorkerne zusammen mit den verfügbaren Caches hierarchisch in einer Baumstruktur angeordnet.

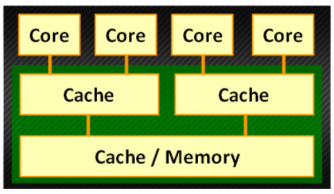


Abbildung 3: Hierarchische Struktur

Simultaneous Multithreading ermöglicht die simultane Ausführung mehrerer Kontrollflüsse, indem zwischen diesen Kontrollflüssen nach einer definierten Strategie umgeschaltet wird. Simultaneous Multithreading führt zu keiner echten parallelen Ausführung mehrerer Threads, sondern zu einer verbesserten Ausnutzung der verfügbaren Ressourcen, basierend auf der Duplizierung von Registereinheiten, die Prozesszustände speichern

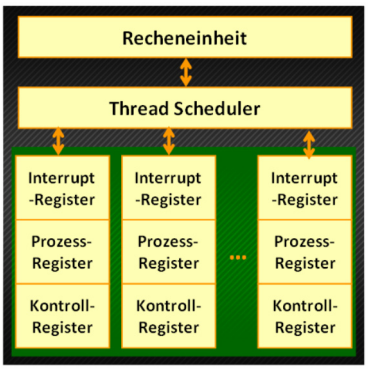


Abbildung 4: Simultaneous Multithreading (auf vereinfachtem RT-Level)

Wie bereits zuvor angedeutet, neigt dagegen die Entwicklung der GPU-Architekturen zum Weg der größtmöglichen Parallelisierung und damit der größtmöglichen Anzahl der parallel zu bearbeitenden Daten im Sinne einer „Single Instruction Multiple Data"-Kategorisierung. Dabei wird eine feingranulare Parallelisierung vordergründig angestrebt.

Mit der Einführung der programmierbaren Shader und der darauf basierenden GPGPU-Schnittstellen wie CUDA verschwimmen die Grenzen der Granularität zunehmend. Zum einen kann die Anzahl der Instruktionen mittels Shader deutlich erhöht werden und zum anderen bieten aktuelle Architekturen wie Nvidia Fermi die Möglichkeit der Parallelisierung von verschiedenen Instruktionsflüssen an, die auf verschiedenen Daten ausgeführt werden. Hierbei wird nach der Klassifikation von Flynn bereits eine Multiple

Instruction Multiple Data-Parallelität erreicht, die auch von Multi-Core-Architekturen von CPUs erfüllt werden.

Ein weiteres deutliches Unterscheidungsmerkmal der Architekturen von GPU und CPU lässt sich an der **Widmung der Transistoren** festmachen. In der Praxis zeigt sich, dass CPU-Architekturen vom Einsatz von Caches und aufwändigen Instruktionsflussstrategien dominiert werden, weswegen ein sehr großer Anteil der Prozessorfläche dem Cache und den Instruktionscontrollern gewidmet wird. Diese Herangehensweise basiert auf den zuvor genannten Optimierungsstrategien wie Pipelining, Caching sowie Super-Skalarität.

GPU-Architekturen verfolgen nahezu das entgegen gesetzte Ziel; Caches und Kontrollstrukturen werden zu Gunsten von Berechnungseinheiten reduziert, um über die bloße Anzahl der verfügbaren Berechnungseinheiten eine größtmögliche Parallelität zu erreichen.

Abbildung 5: Unterschiedliche Widmungs-Strategien von GPU und CPU-Architekturen[14]

Wie bereits angedeutet, unterliegen GPU-Architekturen seit der Einführung des GPGPU-Modells einem Wandel, der zunehmend die Architekturen in Richtung der Spezifikationen der heutzutage üblichen CPU-Parameter annähert. So wurden bei der Nvidia Fermi-Architektur neben einen hierarchischen Cache auch Fehler korrigierende Maßnahmen (ECC) eingeführt. Im direkten Vergleich von aktuellen GPUs und CPUs lässt sich eine deutlich höhere Transistorenanzahl bei den GPUs beobachten. Dies begründet sich in der Erweiterung der GPUs um Caches und Instruktionscontrollern, unter der Vermeidung einer Einschränkung der verfügbaren parallelen Recheneinheiten. [14][15]

In der Praxis lässt sich dies an aktuellen CPUs und GPUs festmachen. So verfügen GPUs der Fermi-Architektur (*Nvidia*) rund 3 Milliarden Transistoren [15], während eine aktuelle CPU (*Intel Core i7*) dagegen 1 Milliarde Transistoren besitzt [21].

4.3 Konzepte der parallelen Programmierung

Die drei wesentlichen Hauptstrategien zur Ausnutzung von Parallelität von sequentiellen Algorithmen sind [12]

- die **Domain Decomposition**
- die **Task Decomposition (Functional Decomposition)** und
- das **Pipelining**.

Die **Domain Decomposition** beschreibt die Untersuchung zur Aufteilbarkeit der Daten des sequentiellen Algorithmus und der Zuweisung an parallele Tasks, die jeweils eine Partition der Daten erhalten. Die **Task Decomposition** untersucht dagegen die Instruktionsfolge auf Parallelisierbarkeit. In beiden Fällen wird als Kriterium der Untersuchung die Unabhängigkeit von Daten beziehungsweise Instruktionen herangezogen. Lässt sich diese Unabhängigkeit feststellen, können die dabei entstehenden Daten- beziehungsweise Instruktionspartitionen parallel ausgeführt werden.

Das **Pipelining** ist ein Spezialfall der Task Decomposition. Hierbei werden aufeinander folgende Instruktionen parallel ausgeführt, wobei die behandelten Daten von Pipeline-Schritt zu Pipeline-Schritt weitergereicht werden.

Das bereits zuvor erwähnte Pipelining als Optimierung von einzelnen Instruktionsflüssen und das Pipelining, wie es in GPU-Architekturen vorkommt, entsprechen in beiden Fällen der Task Decomposition – wobei der Unterschied in der Granularität der Tasks liegt. Während auf der CPU selbst einzelne Instruktionen in ihre Teilbefehle (Fetch, Decode, Execute und Writeback) zerlegt werden, werden im Gegensatz dazu auf der GPU die zahlreichen Instruktionen eines Bestandteils der Grafikpipeline zu einem Task zusammengefasst.

Ein bereits erwähntes Konzept zur Beschreibung von parallelen Programmierkonzepten ist die **Taxonomie nach Flynn**. Diese sehr grundlegende und abstrakte Klassifikation unterscheidet Architekturen nach der Art des Umfangs von Instruktionen und Daten. Hierbei wird ausschließlich zwischen mehreren und einzelnen Daten unterschieden.

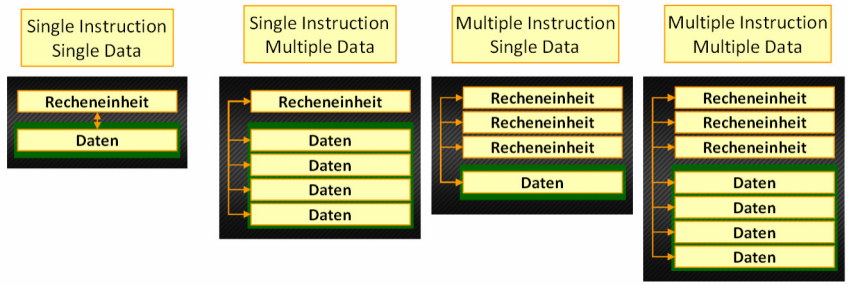


Abbildung 6: Flynn's Taxonomie

Algorithmen die nach einer Decomposition in Tasks aufgeteilt wurden, die keinerlei Kommunikation in Form des Austauschs von Daten benötigen, werden auch als peinlich parallel (**embarrasingly parallel**) bezeichnet.

Die Handhabung der Daten wird folgendermaßen unterschieden [12]:

- Single Program Multiple Data (SPMD)
- Multiple Programm Multiple Data (MPMD)

SPMD ist eine Erweiterung der SIMD-Klassifikation nach Flynn. Während bei SIMD jede Instruktion für sich isoliert betrachtet auf eine Menge von Daten angewandt wird, rückt bei SPMD der Kontext einer Instruktion - die Instruktionsfolge samt Branches und Konditionen - in den Vordergrund. Diese Erweiterung ermöglicht es, Instruktionen in Abhängigkeit der betrachteten Daten auf diesen auszuführen. Es werden also nicht in allen Fällen notwendigerweise alle Instruktionen auf der Menge der betrachteten Daten ausgeführt, sondern möglicherweise nur ein Teil davon. **MPMD** basiert auf der Kombination von SPMD-Einheiten, so dass mehrere unterschiedliche Programme auf verschiedenen Datenmengen ausgeführt werden.

4.4 Verfahren zur Messung der Performance

Zur Messung der Performance von parallelen Implementierungen existieren Kenngrößen, die im Folgenden erläutert werden. Unter Verwendung dieser Kenngrößen wird die im Rahmen dieser Studienarbeit entwickelte Implementierung analysiert und ein Vergleich zur ursprünglichen sequentiellen Implementierung auf der CPU entwickelt. [30]

- Serielle Laufzeit T*(n)
  Entspricht der Zeit zwischen Start und Ende der Programmausführung einer sequentiellen Implementierung auf einer einzigen Recheneinheit.

- Parallele Laufzeit $T_p(n)$
  Entspricht der Zeit zwischen Start und Ende der Programmausführung auf allen

Prozessoren. Zur Laufzeit einer Programmausführung auf einer parallelen Architektur zählen die Summe aller Ausführungszeiten aller Recheneinheiten, Synchronisationszeiten sowie Leerlaufzeiten.

- Speedup Sp(n)
  Der Speedup beschreibt die relative Steigerung der Performance zwischen einer sequentiellen Implementierung und einer parallelisierten Implementierung. Diese Kenngröße ist die zentrale Größe zur Ermittlung des gewonnen Nutzens, der durch die Parallelisierung erreicht wird.

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

  Theoretisch kann der Speedup niemals höher als die Anzahl der eingesetzten Recheneinheiten ausfallen. *($S_p(n) \leq p$)*
  Jedoch kann der Speedup tatsächlich größer ausfallen, falls Cache-Effekte auftreten. *($S_p(n) > p$)*

4.5 Die Applikationsschnittstelle CUDA

4.5.1 Einführung

Die **Compute Unified Device Architecture (CUDA)** ist eine von Nvidia entwickelte Software- und Hardware-Architektur, die das GPGPU-Modell implementiert.

Im Hardware-Bereich wurden die von Nvidia entwickelten GPU-Architekturen für die Anwendung des GPGPU-Modells angepasst. Im Bereich der Software wurden Programmiersprachen um Bibliotheken und Schnittstellen ergänzt, die eine Abstraktion des GPGPU-Modells im Zusammenhang mit GPU-Architekturen durchführen, um Benutzern einen abstrahierten Zugang zu ermöglichen. Die Besonderheit der Abstraktion liegt in der Eigenschaft, dass ein Benutzer von CUDA sich nur mit denjenigen Eigenschaften von GPU-Architekturen auseinandersetzen muss, die für seine Probleme relevant sind. Ein Benutzer muss sich nicht mit den Eigenheiten der Grafikpipeline auseinandersetzen, die die eigentliche Grundlage der GPU-Architektur darstellt.

Nvidia verwendet im Zusammenhang mit dessen Applikationsschnittstelle CUDA und seinen CUDA-fähigen GPU-Architekturen eine eigene Terminologie, dessen der Entwickler mächtig sein muss, um beim Studium von begleitender Literatur einen Informationsgewinn erreichen zu können. [14]

4.5.2 Die CUDA-Architektur

Genauer betrachtet, besteht die CUDA-Architektur aus folgenden Komponenten:

- Parallele Recheneinheiten in Nvidia GPUs
- Erweiterungen des Betriebssystem-Kernels zur Initialisierung, Konfiguration und Interaktion der Hardware
- Treiber, die eine Applikationsschnittstelle zur Hardware für Entwickler anbieten
- PTX (Parallel Thread Execution) Befehlssatz-Architektur zur parallelen Ausführung von Kernel und Funktionen.

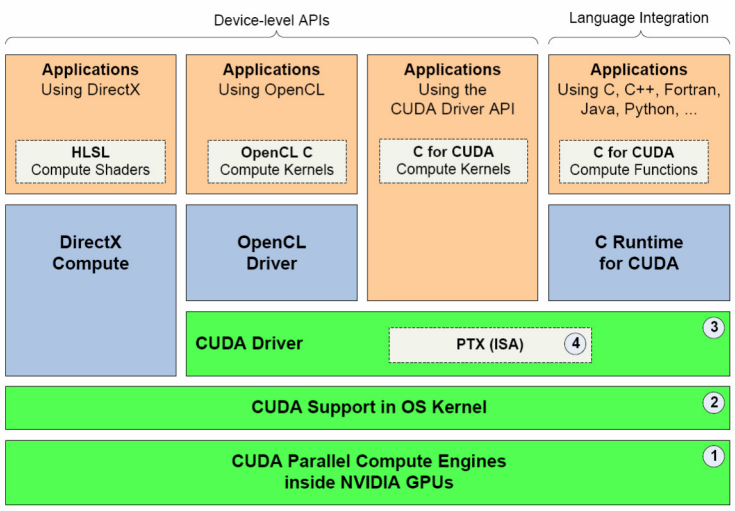


Abbildung 7: Nvidia CUDA Architektur [14]

Nvidia unterscheidet zwei Herangehensweisen bzw. Programmierschnittstellen für CUDA:

- Benutzung der Programmierschnittstelle auf Device-Level
- Benutzung einer Schnittstelle, die in einer Programmiersprache integriert wurde

Bei der Programmierschnittstelle auf Device-Level handelt es sich um die direkte Benutzung der Applikationsschnittstelle des Treibers. Programmteile, die für die GPU bestimmt sind werden in einfachem C oder in der assemblerartigen PTX-Sprache implementiert.

Bei der integrierten Schnittstelle in einer Programmiersprache handelt es sich um eine Abstraktion, um eine Programmierung auf Hochsprachen-Niveau zu ermöglichen. Der

Vorteil liegt in der Möglichkeit, Standardtypen sowie eigenentwickelte Typen wie Strukturen zu benutzen. Diese Erweiterung von Programmiersprachen wird in der Literatur „C for CUDA“ genannt. [14] Im Wesentlichen können folgende Fähigkeiten zusammengefasst werden, die dem Benutzer dieser Programmiersprachen-Erweiterung zur Verfügung gestellt werden:

- Spezifikation von Funktionen, die auf der GPU ausgeführt werden
- Verwaltung des GPU-Speichers
- Bestimmung der Parameter zur parallelen Ausführung auf der GPU

4.5.3 Die Abstraktion der GPU als Modell zur Programmierung mit CUDA

Wie bereits angedeutet, verwendet Nvidia eine eigene Terminologie in der Dokumentation seiner CUDA-Architektur. [14]

Ein **Thread** ist im Gegensatz zum CPU-Thread keine Kombination aus Programmkontext und Programm, sondern ein Grundelement, das eine Menge von zu bearbeitenden Daten beschreibt. Threads werden zu so genannten **Warps** zusammengefasst. Diese Warps basieren auf der Eigenschaft der SIMD-Architektur der GPU, die eine Instruktion gleichzeitig auf einer Menge von Daten bearbeitet. Diese Menge entspricht dem Warp.[2]

Nvidia verwendet in seiner Literatur daher statt dem Begriff SIMD den Begriff **SIMT** (Singe Instruction Multiple Thread). Nvidia argumentiert in der Dokumentation [14] damit, dass SIMT Instruktionen das Verhalten der Ausführung und Verzweigung jedes einzelnen Threads bestimmen. In anderer Literatur [12] wird diese Form der parallelen Taxonomie als **SPMD** (Single Programm Multiple Data) bezeichnet. In beiden Fällen wird das SIMD-Modell um den Aspekt der Betrachtung der Instruktionsfolge erweitert, mit der Absicht Verzweigungen einzuführen, wodurch die Daten nach Bedingungen partitioniert werden, auf denen unterschiedliche Instruktionsfolgen ausgeführt werden. (engl. Branching)

Ein Warp ist dementsprechend die entscheidende Größe bei der Einteilung und Zuweisung einer Menge von Threads an SIMD-Prozessoren. Warps lassen sich für die Anwendung auf größeren Datenmengen zu **Blöcken** zusammenfassen. Die Threads eines Blocks können den gesamten Prozessorkern ausfüllen. Daher ist die Größe eines Blocks durch die Threads eines Prozessorkerns begrenzt. Diese Zahl liegt in der Größenordnung zwischen 768 und 1536 je nach Compute Capability. Die Compute Capability ist ein Index, den Nvidia CUDA-fähige GPUs vergibt, um Spezifikationen und Eigenschaften der unterschiedlichen GPU-Generationen unterscheiden zu können. [23]

[2] Der Begriff Warp stammt aus der Weberei, in der mehrere Fäden (engl. Thread) zu Warps zusammengefasst werden.

Die gesamte Anzahl an verwendeten Threads ist jedoch nicht durch die Obergrenze der Threads pro Block begrenzt. Stattdessen können mehrere gleichartige Blöcke erzeugt werden, die nun die vielfache Anzahl der Threads pro Block zusammenfassen. Die Zusammenfassung mehrerer Blöcke wird als **Gitter** (Grid) bezeichnet.

Die Instruktionsfolge die auf einem Gitter - also auf einer hierarchischen Zusammenfassung von Threads ausgeführt wird - wird als **Kernel** bezeichnet. Generell wird die GPU als Ko-Prozessor zur CPU betrachtet. CPU und dazugehöriger Arbeitspeicher werden unter dem Begriff **Host** zusammengefasst. Wohingegen die GPU samt ihrem Speicher als **Device** bezeichnet wird.

Ein besonderes Merkmal dieses Abstraktionsmodells ist die Verwaltung der verfügbaren Ressourcen durch die Hardware. Stehen für eine Menge von Blöcken keine ausreichenden Ressourcen zur Verfügung, werden die Blöcke sequentiell abgearbeitet.

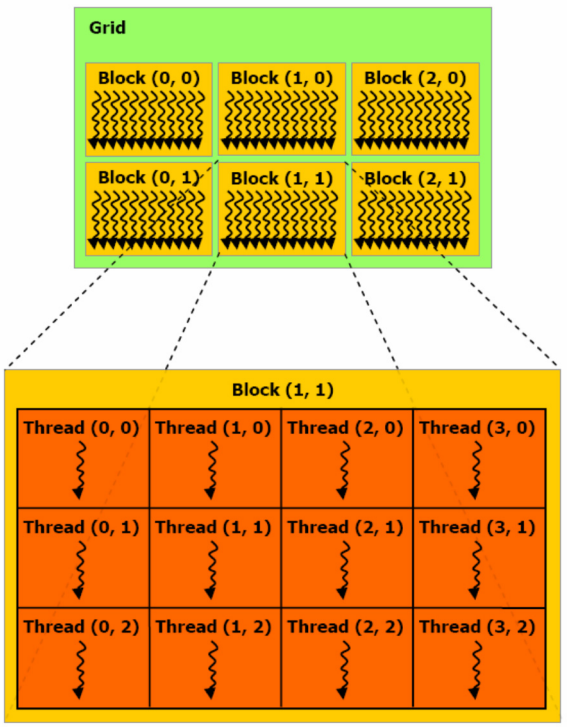


Abbildung 8: Das Abstraktionsmodell von CUDA [14]

4.5.4 Das Speichermodell von CUDA

Die GPU-Architektur beinhaltet unterschiedliche Speicherarten, die auf den Erfordernissen der Grafikpipeline basieren. CUDA ermöglicht den Threads eines Kernels den Zugriff auf alle vorhandenen Speicherarten.

Jeder Thread besitzt lokale 32-Bit Register, der einen Teil des Registerfiles des Prozessorkernes darstellt. In Abhängigkeit der bereits erwähnten Compute Capability ist die Größe dieses Registerfiles unterschiedlich dimensioniert und beträgt zwischen 8K und 32K. Darüber hinaus besitzt jeder Thread lokalen Speicher, der sich im globalen Hauptspeicher der GPU befindet. Als weiterer Speicher, der sich direkt auf der GPU befindet, existiert ein gemeinsamer Speicher, dessen Inhalt für jeden Thread eines Blocks sichtbar ist. Der Hauptspeicher der GPU ist als globaler Speicher für jeden Thread sichtbar.

Daneben existieren zwei Nur-Lese-Speicher, der Textur und Konstantenspeicher.

Speicher	**Lokalität**	**gecached**	**Zugriff**
Register	Auf der GPU	Nein	Lesen/Schreiben
Lokal	Im Hauptspeicher	Ja*	Lesen/Schreiben
Gemeinsamer Speicher	Auf der GPU	Nein	Lesen/Schreiben
Global	Im Hauptspeicher	Ja*	Lesen/Schreiben
Konstanten	Im Hauptspeicher	Ja	Nur Lesen
Texturen	Im Hauptspeicher	Ja	Nur Lesen

Tabelle 1: Übersicht über die Speicherarten von CUDA

* Ab Compute Capability 2.0

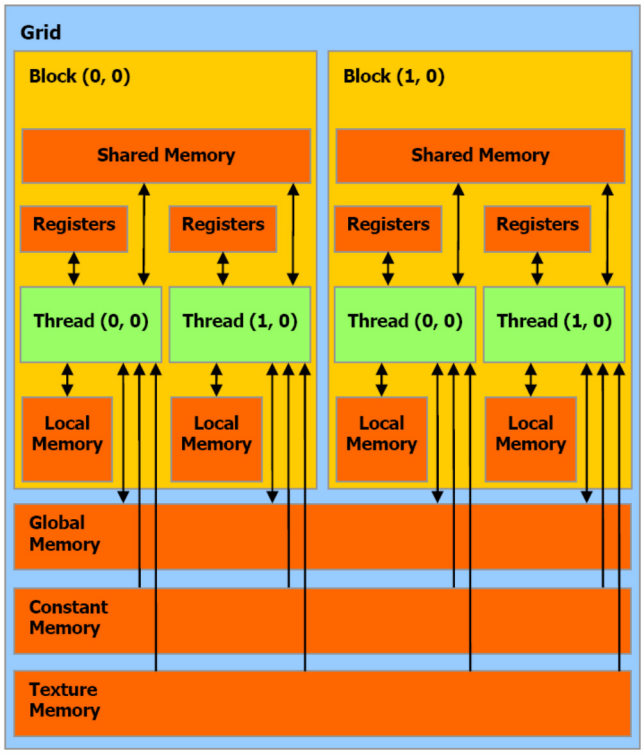


Abbildung 9: Die Speicherhierarchie von CUDA [14]

4.6 Die Kommunikation zwischen GPU und CPU

Die Bandbreite zwischen den Hauptkomponenten GPU und CPU ist eine maßgebende Größe für die Gesamtperformance, da die Initialisierung der Ausführung von Kerneln von einem Hostprogramm auf der CPU durchgeführt wird. Darüber hinaus sind auch die Initialisierung, das Senden und die Weiterverarbeitung von Daten seitens der CPU ebenfalls von dieser Bandbreite abhängig.

Heutige Grafikkomponenten werden üblicherweise per PCI-Express (**P**eripheral **C**omponent **I**nterconnect Express) mit dem Chipsatz der CPU verbunden.

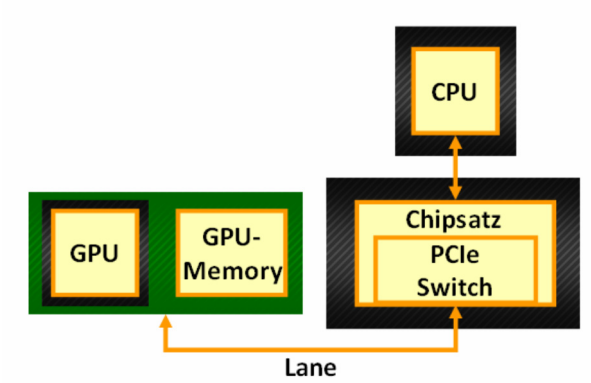


Abbildung 10: PCI-Express

PCI-Express ist im Gegensatz zu seinem Vorgängerstandard kein Bus-System sondern eine serielle Punkt-zu-Punkt-Verbindung, wobei die einzelnen Komponenten über so genannte „Links“ zu einem zentralen Switch verbunden werden. Ein Link besteht aus sogenannten „Lanes“. Ein Lane ist definiert als 1-Bit serielles Übertragungsmedium, das eine Bandbreite von 250 MB/s aufweist. Ein solches Lane besteht aus 2 Leitungspaaren, wobei ein Leitungsband zum Senden und eines zum Empfangen verwendet wird.

Zur Anbindung von hoch performanten Grafikkomponenten werden mehrere Lanes zusammengefasst, um höhere Bandbreiten zu erzielen. Maximal können 32 Lanes in einem Slot zusammengefasst werden. In der Praxis werden für Grafikkomponenten jedoch nur 16 Lanes eingesetzt, wofür die Bezeichnung PEG (PCI Express for Graphics) verwendet wird. [24]

4.7 Die Fermi-Architektur

Die Fermi-Architektur besitzt besondere Merkmale, die auf den Erfordernissen der GPGPU-Architektur basieren. Die wesentlichen Merkmale, die für den Einsatz von GPGPU und daneben auch für die parallele Partikelsimulation dieser Studienarbeit relevant sind, werden im Folgenden dargestellt. [15]

- Implementierung von Arithmetik für Gleitkommazahlen doppelter Präzision samt vollständiger Unterstützung des IEEE 754-2008 Standards. Diese Erweiterung besitzt ein Performancepotential für Berechnungen, die eine hohe Präzision benötigen und daher Gleitkommazahlen doppelter Präzision verwenden.

- Erweiterung der Anbindung an den globalen Hauptspeicher um eine zweistufige Cache-Hierarchie. Durch das lokale Zwischenspeichern von Daten können räumliche und temporale Kohärenzen für einen Performancevorteil genutzt werden.

- Möglichkeit der parallelen Ausführung mehrerer Kernel. Wie bereits angedeutet, bietet diese Erweiterung die Möglichkeit einer Parallelisierung von unabhängigen Instruktionsfolgen. Im Zusammenhang mit dieser Studienarbeit werden mehrere Simulationsdurchläufe, deren Instruktionsfolgen durch die Vielfalt der Partikelkombinationen unterschiedlich sein können, durch diese Eigenschaft parallelisiert. Nvidia bezeichnet diese Funktionalität als „GigaThread Engine“.

- Einführung von Fehlerkorrigierenden Codes (ECC – Error Checking and Correction). Diese Codes sind besonders im Zusammenhang mit sensiblen Daten und Simulationen wie zum Beispiel der Medizinischen Volumenvisualisierung von Vorteil, da in diesen Bereichen ein hohes Maß an Datenintegrität vorausgesetzt wird.

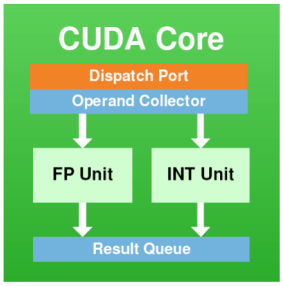


Abbildung 11: Architektur eines CUDA Cores (Streamprozessor) [15]

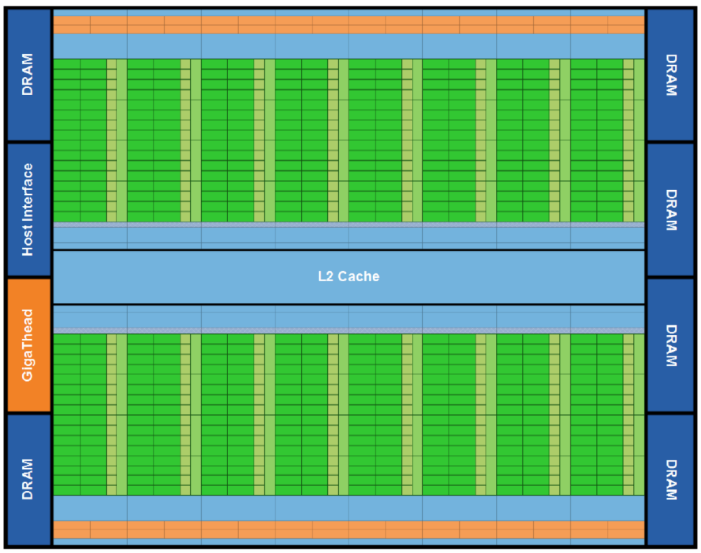


Abbildung 12: Übersicht Nvidia Fermi-Architektur [15]

4.8 Die Zielhardware

Nvidia differenziert die Spezifikationen und Eigenschaften unterschiedlicher GPU-Generationen durch die Compute Capability, die durch einen Index repräsentiert wird. Als Zielhardware, auf denen die Implementierung der parallelen Partikelsimulation ausgeführt werden soll, kommen GPUs mit der Compute Capability 2.0 und höher in Frage.

Die Compute Capability 2.0 beschreibt unter anderem eine Fähigkeit der damit ausgezeichneten GPUs, die es Benutzern ermöglicht, mehrere Kernel parallel ausführen zu lassen. Die parallele Partikelsimulation, die in Kapitel 4 beschrieben wurde, basiert auf einem stochastischen Prozess. Dieser setzt die Bedingung voraus, über die Ergebnisse einer großen Anzahl von Simulationsdurchläufen zu mitteln. Basierend auf dieser Fähigkeit, wird in einem weiteren Lösungsansatz eine Abbildung von je einem Simulationsdurchlauf auf einen Aufruf eines Kernels vorgenommen.

Die Entwicklung der parallelen Partikelsimulation wurde auf der GPU Nvidia Tesla C2050/C2070 durchgeführt. Diese GPU zeichnet sich durch folgende Kenngrößen aus [23][25]:

- Anzahl Streaming-Multiprozessoren: 14
- Anzahl Streamprozessoren (Auch Cuda Cores): 448
- Maximale Anzahl von Threads per Multiprozessor: 1536
- Maximale Anzahl von Threads: 21504
- Takt der Multiprozessoren: 1,15 GHz
- Speicherbandbreite: 144 GB/s
- Speicherschnittstelle: 384 Bit
- Speichertakt: 1,5 GHz
- (Theoretische) Leistungsfähigkeit 1,03 TFlops (Single Prec.)
- Größe des Speichers 3 GB GDDR5
- Energieverbrauch 238W

Der Grafikprozessor C2050/C2070 basiert auf einer speziell für die Erfordernisse des GPGPU-Modells optimierten Fermi-Architektur.

5. Die Abbildung des Simulationsmodells auf GPGPU-Architekturen

5.1 Allgemeines

Nach der genauen Betrachtung der Anforderungen an die Implementierung, die in dieser Studienarbeit angefertigt werden sollte, wurde ein besonderes Augenmerk auf eine Lösung unter Ausnutzung der Parallelität von mehreren Simulationsdurchläufen gelegt, wie sie im Einführungskapitel bereits erläutert wurde.

Die im Rahmen dieser Studienarbeit verwendete Programmiersprache ist C/C++ mit CUDA spezifischen Erweiterungen. In der Literatur wird dies auch als „C for CUDA“ bezeichnet [14]. In dieser Arbeit wurde eine klare Trennung zwischen CUDA-spezifischen Programmteilen und der gesamten restlichen Implementierung vorgenommen. Implementierungsteile, die vollständig auf der GPU ausgeführt werden, halten sich strikt an die Konventionen der Programmiersprache C. Währenddessen in der restlichen Implementierung der Fokus auf einer hohen Wartbarkeit und eines einfachen Verständnis der Funktionalität im Vordergrund liegt – daher wurden hier die objektorientierten Eigenschaften der Programmiersprache C++ benutzt.

Resultierend auf dieser Trennung war die erste Aufgabe dieser Studienarbeit die Entwicklung eines objektorientierten Modells der Simulationsfunktionalitäten.

5.2 Grundlegende Implementierungen

Zu Beginn der Studienarbeit existierten zwei Implementierungen der Simulation.

- Die erste Implementierung iteriert in einer sequentiellen Form über alle Partikel und berechnet dabei Kräfte und Drehmomente sowie die zufälligen Verschiebungen und Interaktionen zwischen den Partikeln. Die Simulation findet auf der CPU statt.

- Die zweite Implementierung bedient sich des GPGPU-Modells unter der Benutzung der Applikationsschnittstelle CUDA. Diese Implementierung führt die Simulation GPU-basiert aus. Dabei wird die Beobachtung ausgenutzt, aus der hervorgeht, dass die einzelnen Berechnungen der Kräfte und Drehmomente innerhalb eines Zeitschrittes unabhängig zueinander sind. In dieser Implementierung wird jedes Partikel einem Thread zugeordnet.

Die ursprüngliche Modellimplementierung fand unter der Verwendung des Computeralgebrasystems Matlab statt. Der Laufzeitoverhead, der durch die schlichte Interpretierung der Quelltextzeilen verursacht wurde, veranlasste eine Entwicklung unter der Verwendung der Programmiersprache C. Diese kompilierte ausführbare C-Version des Simulationsmodells war ebenfalls für aussagekräftige Ergebnisse kaum zu gebrauchen, da der ernorme Berechnungsaufwand für die Simulation einen sinnvollen zeitlichen Rahmen bei weitem übersteigt.

Um diesen Laufzeitoverhead zu reduzieren, wurde die Domäne in ein äquidistantes Gitter überführt, wodurch die Anzahl der Interaktionen zwischen weit entfernten Partikeln ignoriert wurde. Jedoch liegt der Berechnungsaufwand dieser Version ebenfalls weit fernab eines sinnvollen zeitlichen Rahmens. Durch die Betrachtung der Prämisse einer großen Vielzahl von Simulationsdurchläufen begründet sich der Bedarf nach einer Optimierung zur Reduzierung der Berechnungszeit.

Laut dem biologischen Modell, auf dem die Partikelsimulation basiert, ziehen sich die Partikel aufgrund wirkender Gravitationskräfte und Drehmomente an und bewegen sich basierend auf der Braunschen Molekularbewegung durch die Domäne. Zur Berechnung von Kräften und Drehmomenten zwischen den Partikeln findet folglich eine Interaktion statt.

Dem Simulationsmodell zufolge interagiert jedes Partikel mit jedem anderen Partikel, das sich innerhalb der Domäne befindet (siehe Algorithmus 2). Diese Menge aller möglichen Interaktionen wird in einer sequentiellen Form auf die CPU abgebildet. In jedem Simulationsschritt wird eine Iteration über jedes Partikel vorgenommen, um innerhalb der Iteration wiederum über jeden Interaktionspartner zu iterieren.

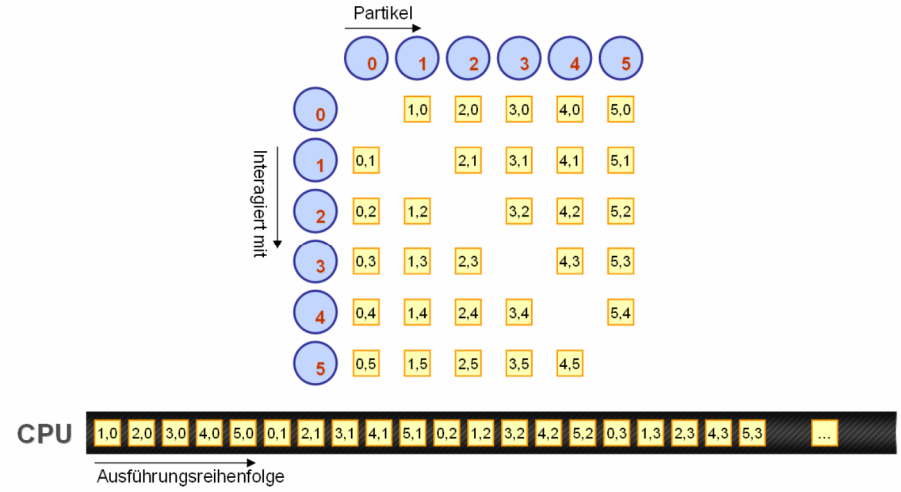


Abbildung 13: Abbildung des Simulationsmodells auf die CPU

Eine Durchführung mehrerer Simulationsdurchläufe ist in der sequentiellen Version auf der CPU nicht vorgesehen. Das notwendige Kriterium der hinreichend großen Vielfalt von Simulationsergebnissen zur Mittelung wird nicht erfüllt.

Um dieses Kriterium in die spätere Analyse einfließen zu lassen, wird die Berechnungsdauer eines einzelnen Simulationsdurchlaufs linear skaliert. Dabei wird eine sequentielle Hintereinanderausführung des Programms angenommen.

5.3 Abstraktion der Simulation in Form eines objektorientierten Modells

Um der reinen Simulation einen sinnvollen und gebräuchlichen Rahmen zu geben, wurden einrahmende Objekte samt Schnittstellen eingeführt, um eine höchstmögliche Wartbarkeit und Flexibilität zu ermöglichen.

Dabei wurde vor allem auf die Ersetzbarkeit von einzelnen Komponenten ein großer Wert gelegt. Um dieses Ziel zu erreichen, wurde das Design der Objekte einer sehr hohen Unabhängigkeit der einzelnen Komponenten unterworfen, wie sie in Abbildung 14 dargestellt ist. Die Implementierung der **Verwaltung der Simulationsdurchläufe** ermöglicht einen kompletten Austausch der Simulationsprogrammierschnittstelle. Die entwickelte Lösung verwendet die Programmierschnittstelle CUDA – diese kann durch weitere Applikationsschnittstellen wie OpenCL, OpenMP oder ATIStream ersetzt werden, ohne einen Einfluss auf die übrigen Komponenten zu nehmen.

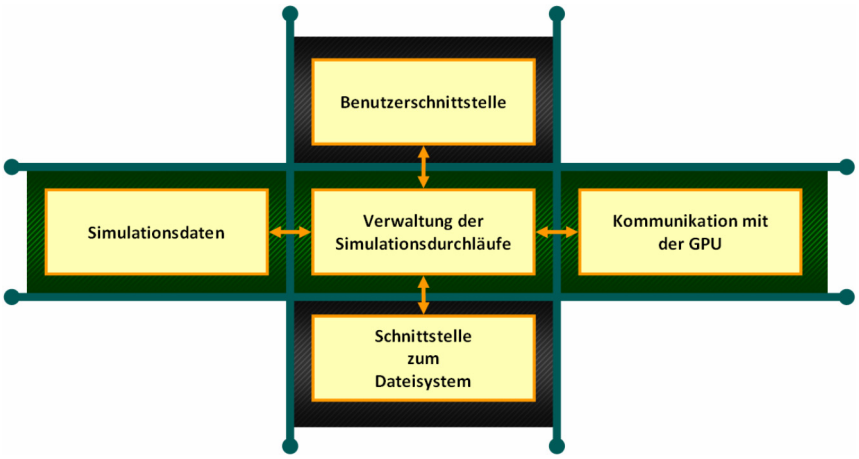


Abbildung 14: Objektorientiertes Design der Simulationsverwaltung

Dabei wurde vor allem auf die Ersetzbarkeit von einzelnen Komponenten ein großer Wert gelegt. Um dieses Ziel zu erreichen, wurde das Design der Objekte einer sehr hohen Unabhängigkeit der einzelnen Komponenten unterworfen, wie sie in Abbildung 17 dargestellt ist. Die Implementierung der **Verwaltung der Simulationsdurchläufe** ermöglicht einen kompletten Austausch der Simulationsprogrammierschnittstelle. Die entwickelte Lösung verwendet die Programmierschnittstelle CUDA – diese kann durch weitere Applikationsschnittstellen wie OpenCL, OpenMP oder ATIStream ersetzt werden, ohne einen Einfluss auf die übrigen Komponenten zu nehmen.

Die aktuell eingesetzte **Benutzerschnittstelle** ist kommandozeilenorientiert und bietet keinerlei grafische Funktionen. Es bietet sich hier die Möglichkeit an, eine grafische Benutzerschnittstelle zu entwickeln, die die Organisation von mehreren Simulationsdurchläufen, sowie den damit verbundenen Parametern, wie Partikeltypen oder Simulationsdauer, vereinfacht. Die restlichen Simulationskomponenten benutzen

die Funktionen der Benutzerschnittstelle, um dem Benutzer über den aktuellen Zustand der Simulation zu informieren.

Die **Schnittstelle zum Dateisystem** wird dazu verwendet, Simulationsergebnisse zur Weiterverarbeitung in Dateien zu speichern. Zu dieser Weiterverarbeitung zählt die Visualisierung durch das Tool CellVis des Institutes für Visualisierung und Interaktive Systeme der Universität Stuttgart [5]. Die aktuell eingesetzte Schnittstelle nutzt ein sehr einfaches textbasiertes Dateiformat. Im objektorientierten Design der Implementierung existiert die Möglichkeit, die Funktionalität durch andere Schnittstellen zum Dateisystem zu ersetzen, die komplexere Dateiformate - wie zum Beispiel das XML-Format - unterstützen.

Darüber hinaus wurden die Datenstrukturen, die die **Simulationsdaten** verwalten, sehr flexibel und abstrahiert gestaltet, um eine Möglichkeit zur Verwendung für allgemeine Simulationsprobleme unter Verwendung von Partikeln auf GPGPU-Architekturen zu gewährleisten. Es findet eine Trennung zwischen Partikeln, den Datenstrukturen zur Verwaltung von Partikeln, den Simulationsdurchläufen und den Parametern einer Simulation statt.

Wie bereits erläutert, kann die Applikationsschnittstelle zur Hardware vollständig ersetzt werden. Daher ist die Implementierung der **Kommunikation mit der GPU** getrennt von der Simulationsverwaltung. Bibliotheksspezifische Anweisungen und Funktionen der CUDA Applikationsschnittstelle werden ausschließlich in diesem Teil des objektorientierten Designs verwendet. Daraus folgt die Unabhängigkeit der Simulationsaufgabe von der spezifischen Applikationsschnittstelle.Basierend auf der erläuterten groben Abstraktion der Simulation wurde ein Klassendiagramm entwickelt.

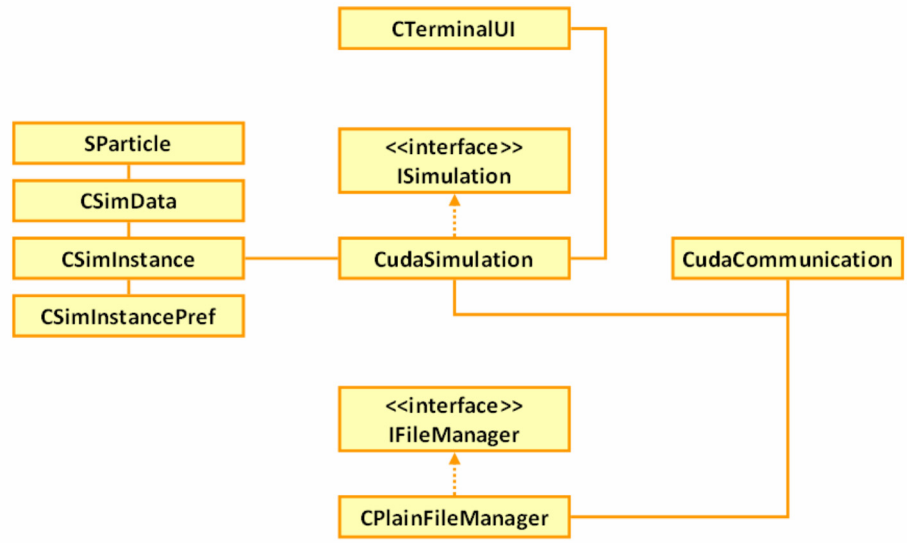
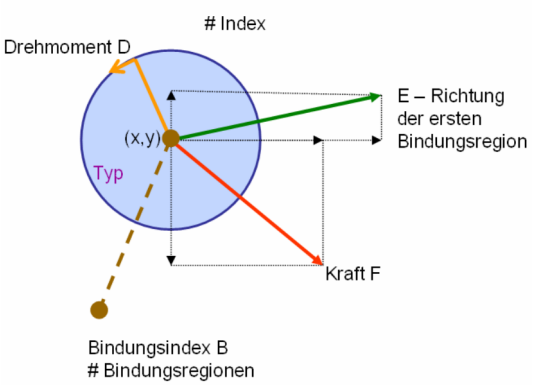


Abbildung 15: Klassendiagramm

5.4 Verwaltung der Partikeldaten auf der CPU-Seite

Ein Partikel verfügt über folgende Eigenschaften und wurde folgendermaßen in der Struktur **SParticle** implementiert:



SParticle	
x	: int [2]
F	: double [2]
E	: double [2]
D	: double
B	: int[3]
INDEX	: int
count_binding_regions	: int
Typ	: char

Abbildung 16: Partikeleigenschaften und Implementierung

Bei der Entwicklung des Lösungsansatzes zur Verwaltung der Partikeldaten auf der CPU-Seite wurden zwei Lösungsansätze entwickelt und von einander abgewägt:

- Im Hauptspeicher der CPU wird nur Speicher für einen einzigen Simulationsdurchlauf angelegt. Dieser Speicher dient zur Erzeugung der Partikeldaten in der Initialisierungsphase der Simulation. Im späteren Verlauf der Simulation wird dieser Speicher als Puffer für die Zwischenergebnisse der Simulation genutzt. Die Daten werden in diesem Puffer zwischengespeichert, wenn sie von der GPU gelesen und in Dateien geschrieben werden. Des Weiteren befinden sich im Hauptspeicher der CPU nur Daten zur Verwaltung der Simulationsdurchläufe auf der GPU in Form von Zeigern auf Speicheradressen im globalen Speicher der GPU.

- Im Hauptspeicher der CPU wird eine umfangreiche Datenstruktur zur Verwaltung der kompletten Simulationsdaten angelegt. Die Partikeldaten werden getrennt nach Typ in Listen gespeichert. Für jeden Simulationszwischenschritt werden Instanzen angelegt, so dass Aussagen über den Simulationsverlauf getroffen werden können. Jeder Simulationsverlauf wird zu einer Simulationsinstanz zusammengefasst, der einem Simulationsdurchlauf auf der GPU entspricht. Im globalen Speicher der GPU befinden sich zur Simulationszeit nur die aktuellen Daten, auf denen die Simulation ausgeführt wird. Im Hauptspeicher der CPU befindet sich stattdessen neben den Zeigern zur Verwaltung der Daten auf der GPU die Datenstruktur, die die Verwaltung von mehreren Simulationsdurchläufen mit historischen Daten ermöglicht.

Als Lösung wurde der zweite Lösungsansatz gewählt und implementiert. Ein wesentlicher Bestandteil der Motivation der Abbildung auf einer GPGPU-Architektur liegt darin, mehrere Simulationsdurchläufe zu verwalten und über Ergebnisse zu mitteln. Daher ist es sinnvoll, neben dem bloßen Schreiben der Daten in Dateien eine effiziente umfangreiche Datenstruktur zu verwalten, um neben Ergebnissen aus mehreren Simulationsdurchläufen auch Ergebnisse aus der Entwicklung innerhalb der Simulation erzeugen zu können.

Die Klasse **CSimData** verwaltet die Partikeldaten eines Simulationszwischenschrittes. Dazu enthält sie eine Liste von Partikeltypen, wobei jedes Element aus einer Liste von expliziten Partikeldaten besteht. Als Kontextinformation werden die Typenanzahl, die Partikelanzahlen und die Typnamen gespeichert. Darüber hinaus enthält die Klasse Methoden zur Erzeugung, Abfrage und Löschung dieser Listen. Die Methode GetParticlesofType wird dazu benutzt, die Listen der Partikeldaten abzufragen, um sie dann an den globalen Speicher der GPU zu senden.

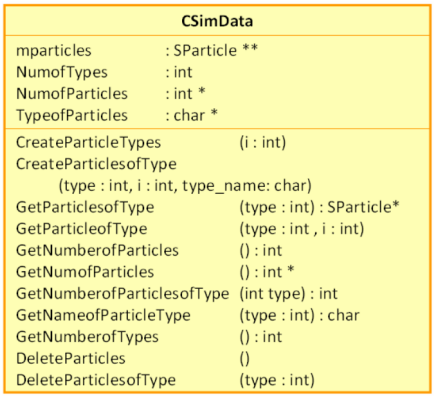


Abbildung 17: Klassendiagramm CSimData

Instanzen der Klasse CSimData werden von der Klasse **CSimInstance** verwaltet. Ein Objekt der Klasse CSimInstance entspricht einem Simulationsdurchlauf. Neben den aktuellen Zwischenergebnissen der Simulation werden auch die bereits erläuterten vergangenen Zustände gespeichert, um die Anforderung zu erfüllen, Aussagen über den Simulationsverlauf erzeugen zu können.

Diese Zwischenergebnisse werden in Form einer einfachen Liste durch ein Vector-Objekt repräsentiert. Da die Simulation insgesamt eine sehr große Anzahl von Daten im Hauptspeicher hält, kann die Anzahl der Elemente in dieser Liste begrenzt werden. Es gibt zwei Modi, die bestimmen, wie neue Elemente in der Liste behandelt werden. Der Modus kann durch das Enumeral *SimDataMode* und der dazugehörigen Methode *setSimDataMode* bestimmt werden. Die Anzahl der maximal gehaltenen Elemente kann über *setMaxNumSimData* bestimmt werden. Die beiden Modi bewirken im Einzelnen:

- Im 1. Modus wird die Liste kontinuierlich gefüllt, bis die maximale Anzahl an Elementen erreicht wurde. Danach wird beim Einfügen eines Elementes das letzte Element überschrieben. (*DATA_LATEST_OVERWRITE*)

- Im 2. Modus wird die Liste als Ring verwaltet. Wird das Ende der Liste erreicht, wird das erste Element der Liste überschrieben. Weitere Elemente werden fortlaufend eingefügt. Der Index des zuletzt eingefügten Elementes kann über die Methode GetIndexofLatestElement abgefragt werden. Daneben kann auch direkt ein Zeiger auf das zuletzt eingefügte Element über die Methode GetLatestDataElement zurückgegeben werden. (*DATA_LOOP_OVERWRITE)*

Wird die Verarbeitung von Daten des Simulationsverlaufs nicht benötigt, kann Rechenzeit eingespart werden, wenn die Größe der Liste auf ein Element begrenzt und der 1. Modus verwendet wird.

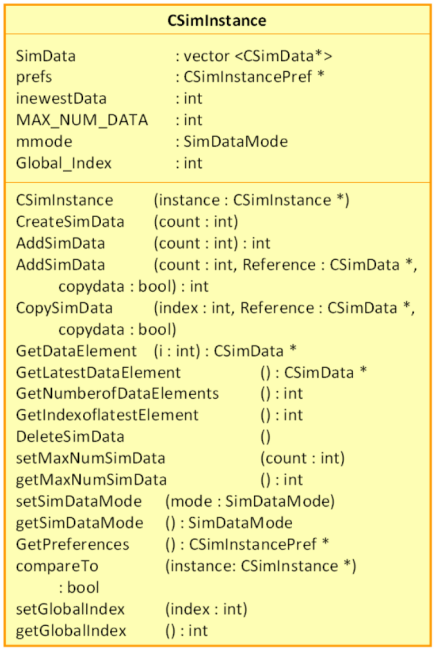


Abbildung 18: Klassendiagramm CSimInstance

Für den Benutzer der Klasse gibt es Methoden zur Erzeugung, Hinzufügung, Abfrage und Löschung von Simulationszwischenständen. Darüber hinaus besitzt jedes Objekt dieser Klasse einen Index, der für die Erzeugung von Dateinamen und der damit verbundenen Zuordnung von Simulationsdaten an Simulationsdurchläufen benutzt wird.

Die Methode compareTo dient dazu, die Rahmenbedingungen zweier Simulationsdurchläufe zu vergleichen. Für ein paralleles Verarbeiten von Simulationsdurchläufen ist es notwendig, dass diese identische Rahmenbedingungen besitzen. Zu diesen Rahmenbedingungen zählen:

- Identische Gittereigenschaften
- Identische Simulationsparameter (Simulationszeit, Simulationsauflösung, Zwischenergebnisintervall)
- Identische Partikeltypen und -anzahlen

Die Parameter einer Simulation werden in den Eigenschaften der Klasse **CSimInstancePref** gespeichert.

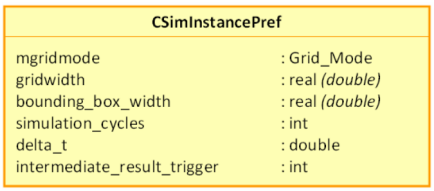


Abbildung 19: Klassendiagramm CSimInstancePref

Diese Klasse speichert die Parameter zum Einsatz eines Gitters, sowie die Parameter Simulationszeit, Simulationsauflösung und Zwischenergebnisintervall.

5.5 Die Verwaltung der Simulationsdurchläufe

Die Simulationsdurchläufe werden von der Klasse **CudaSimulation** zentral verwaltet. Diese Klasse implementiert die Schnittstelle **ISimulation** für die Anwendung der Applikationsschnittstelle CUDA. Die Klasse übernimmt die Koordination der Simulationsdurchläufe während der Simulation, in dem sie aus der Menge der vorhandenen Simulationsdurchläufe Teilmengen auswählt, die parallel auf einer oder mehreren GPUs simuliert werden können. Die gesamte restliche Kommunikation mit der Hardware wurde an die Klasse CudaCommunication delegiert.

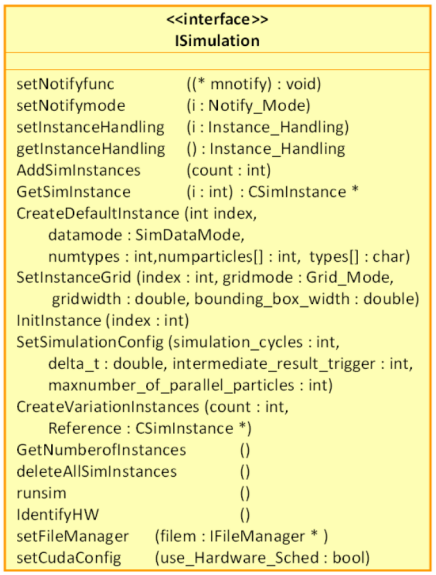


Abbildung 20: Klassendiagramm ISimulation

Die Schnittstelle ISimulation definiert zwei unterschiedliche Modi, die die Art der Behandlung von mehreren Simulationsdurchläufen beschreibt:

- Mehrere Simulationsdurchläufe werden sequentiell ausgeführt. (*INS_NORMAL_MODE*)

- Die vorhandenen Ressourcen werden vollständig genutzt und mit Simulationsdurchläufen gefüllt, um eine parallele Ausführung zu erzielen (*INS_FILL_UP_MODE*)

Darüber hinaus definiert die Schnittstelle den Umgang mit Informationen, die an den Benutzer gesendet werden. Die Schnittstelle erwartet einen Funktionszeiger, der in der gesamten Simulation dazu verwendet wird, Statusinformationen an eine Benutzerschnittstelle zu senden. Hierfür kann die Methode setNotifyfunc verwendet werden. Die Methode setNotifymode definiert eine Relevanzgrenze für Informationen, die an den Benutzer über die Benutzerschnittstelle gesendet werden.

Zur Anlegung, Abfrage und Löschung von Simulationsdurchläufen definiert die Schnittstelle eine Reihe von Methoden. Die Methode CreateDefaultInstance erzeugt in einem Simulationsdurchlauf die für eine Simulation benötigten Datenstrukturen und setzt die Partikeldaten auf zufällige Werte. Die Parameter eines Simulationsdurchlaufs werden mit den Methoden SetInstanceGrid und SetSimulationConfig erzeugt. Die Methode CreateVariationInstances dient zur Vervielfältigung eines erzeugten Simulationsdurchlaufes. Die Parameter eines Simulationsdurchlaufs werden kopiert,

wobei die Partikeldaten zufällig verändert werden, um Variationen eines initialen Simulationsdurchlaufs zu erhalten.

Die Methode *runsim* nutzt die Parameter der erzeugten Simulationsdurchläufe um eine Konfiguration für die Hardware aus Teilmengen der vorhandenen Simulationsdurchläufe zu erzeugen und die eigentliche Simulation durchzuführen.

Die Klasse CudaSimulation erweitert die Schnittstelle um private Methoden, die der Ausgabe von Simulationsinformationen dienen, ausführbare Teilmengen der Simulationsdurchläufe erzeugen und Partikeldaten zufällig verändern.

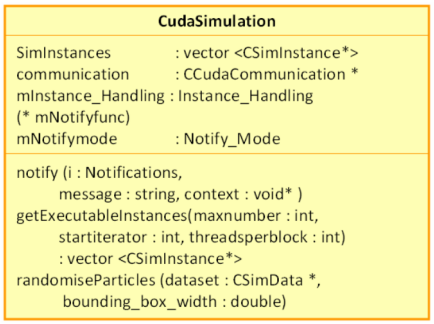


Abbildung 21: Klassendiagramm CudaSimulation

5.6 Verwaltung der Daten auf der GPU-Seite

Zu Beginn der Simulation wird eine Teilmenge der Simulationsdurchläufe ausgewählt, die parallel auf einer GPU und parallel auf mehreren GPUs ausgeführt werden soll. Diese Auswahl führt die Methode runsim der Klasse CudaSimulation durch. Die Klasse **CudaCommunication** erhält diese Teilmenge in Form einer quadratischen einfachen Liste in der Methode runInstances. Diese Methode überprüft nun im ersten Schritt die Ausführbarkeit der Simulationsdurchläufe, in dem sie die Summe der benötigten Threads für die übergebene Menge von Simulationsdurchläufen mit der Kapazitätsgrenze einer GPU vergleicht.

In einem zweiten Schritt werden die Partikeldaten der Simulationsdurchläufe an die einzelnen GPUs gesendet. Dabei wird eine Datenstruktur aufgebaut, um die einzelnen Simulationsdurchläufe jeder GPU und die Verwaltung mehrerer GPUs zu ermöglichen. Der Aufbau dieser Datenstruktur wird von der Methode SendInstancestoGPU durchgeführt.

Diese Datenstruktur muss drei Anforderungen erfüllen:

- Die Datenstruktur muss flexibel einsetzbar sein. Sie muss von Kerneln verwendbar sein, die mehrere Simulationsdurchläufe parallel bearbeiten. Sie muss aber auch von Kerneln benutzbar sein, die nur einen einzigen Simulationsdurchlauf bearbeiten und über die Nvidia Fermi-Eigenschaft der parallelen Ausführung von Kerneln auf eine parallele Ausführung abgebildet werden.

- Die Datenstruktur muss um die Anwendung mehrerer GPUs erweitert werden.

- Für den Einsatz eines gitterbasierten Ansatzes muss die Datenstruktur Eigenschaften aufweisen, womit die Verwaltung dieses Ansatzes ermöglicht wird.

Als Implementierung der Datenstruktur wurde eine baumartige Struktur gewählt. In den Blättern befinden sich die Partikeldaten. Für die Partikeldaten wurde die Struktur SParticle übernommen, wie sie auch im Hauptspeicher der CPU verwendet wird. Die Partikeldaten werden zu Listen je Typ zusammengefasst. Eine weitere Liste speichert die Zeiger auf die Listen der Partikeldaten. Diese Liste umfasst die gesamten Partikeldaten eines Simulationsdurchlaufs. Die Wurzel dieser quadratischen Liste wird samt den Kontextinformationen eines Simulationsdurchlaufs in der Struktur InstanceData hinterlegt. Diese Kontextinformationen umfassen im Wesentlichen Verwaltungsdaten für den gitterbasierten Ansatz. Diese werden im Kapitel „Gitterbasierter Lösungsansatz“ erläutert.

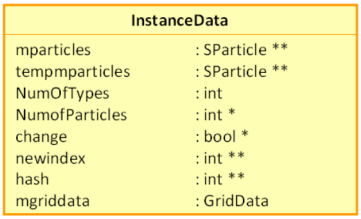


Abbildung 22: Klassendiagramm InstanceData

Kernel, die mehrere Simulationsdurchläufe parallel bearbeiten, benötigen darüber hinaus eine weitere Liste, in denen die Informationen zu den einzelnen Simulationsdurchläufen hinterlegt sind. Diese Informationen wurden in der Struktur KernelData gespeichert.

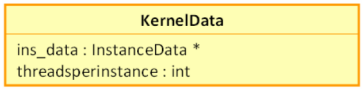


Abbildung 23: Klassendiagramm KernelData

Im Hauptspeicher der CPU befinden sich die KernelData-Objekte, dessen Informationen zur Laufzeit der Simulation an die unterschiedlichen Kernel übergeben werden. Kernel, die nur einen Simulationsdurchlauf betrachten, erhalten einen Zeiger auf ein

InstanceData-Objekt. Wohingegen Kernel, die mehrere Simulationsdurchläufe parallel behandeln, das gesamte KernelData-Objekt erhalten.

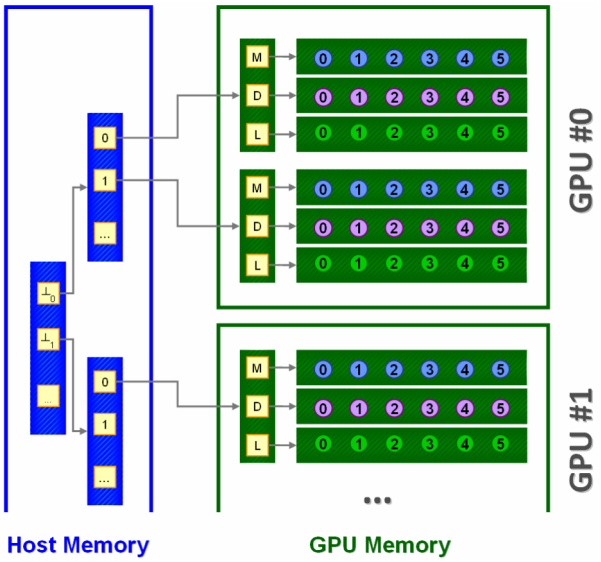


Abbildung 24: Datenstruktur der Partikeldaten auf der GPU

Im Rahmen der Optimierung des Speedups wurde eine Trennung der Sparticle-Datenstruktur in zwei Unterstrukturen untersucht. Dabei wurden die simulationsrelevanten Daten von Datenstrukturinformationen wie Index und Bindungsinformationen getrennt. Die Motivation dieser Trennung liegt in der Art der Benutzung der Daten durch die Simulation. Die Bindungsinformationen dienen der reinen Analyse der Simulationsergebnisse und werden nur erzeugt, wenn die Partikeldaten in Form eines Zwischenergebnisses von der GPU gelesen in eine Datei geschrieben werden. Während der restlichen Simulationszeit liegen sie im globalen GPU-Speicher und könnten auch in Caches gespeichert werden, wo sie jedoch nicht benötigt werden.

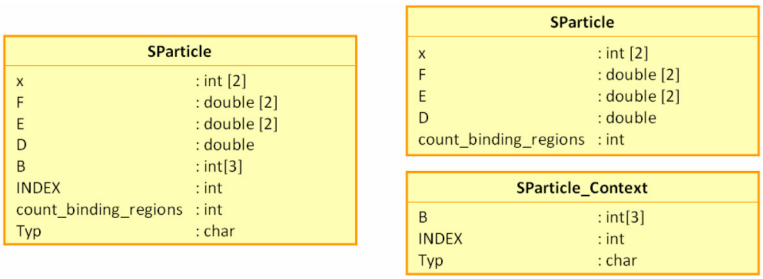


Abbildung 25: Optimierungsversuch zur Trennung der Partikeldaten

Diese Optimierung ergab keinerlei Rechenzeitverkürzung und wurde daher in der weiteren Entwicklung nicht weiter verfolgt.

5.7 Implementierung der Simulationsschleife

Die Simulationsschleife wird von der Methode SimulateInstances durchführt. In einem ersten Schritt wird die Kommunikation mit der GPU initialisiert. Dazu werden die Parameter für die einzelnen Kernelaufrufe in einer Liste gespeichert. Die beinhalten die Konfiguration zum Aufruf der Kernel, sowie die Parameter, die an die Kernel gesendet werden.

In einem zweiten Schritt findet eine Initialisierungsphase für die Zufallszahlen sowie für den Aufbau der Datenstrukturen für den gitterbasierten Lösungsansatzes statt. Für die Analyse der Berechnungsdauer wird der Zeitpunkt des Beginns der Simulation festgestellt. Darüber hinaus wird ein erster Zwischenstand der Simulation erzeugt. Dazu werden die Kräfte berechnet und mögliche Bindungen in Form von Clustern ausgewertet.

Der dritte Schritt enthält die eigentliche Simulationsschleife, die für jeden Simulationsschritt die Positionen der Partikel auswertet sowie die Kräfte und Drehmomente zwischen den Partikeln berechnet. Innerhalb der Simulationsschleife wird ein Zähler verwendet, der dazu verwendet wird, Zwischenergebnisse nach einer vordefinierten Anzahl von Simulationsschritten zu erzeugen.

Bei der Erzeugung eines Zwischenschrittes, werden mögliche Bindungen in Form von Clustern ausgewertet, sowie die Daten von der GPU gelesen und in Dateien geschrieben. An die Benutzerschnittstelle werden Informationen über den Fortschritt der Simulation sowie die Berechnungsdauer gesendet. Die Berechnungsdauer wird hierbei aus der Differenz des aktuellen Zeitpunkts zum Zeitpunkt des Beginns der Simulation gebildet.

Die einzelnen Funktionen der Simulationsschleife starten die Kernel asynchron auf den verfügbaren GPUs. Jeder Kernelaufruf wird in einem eigenen Stream gestartet, um die Unabhängigkeit der Ausführungen zu gewährleisten. Je nach eingestelltem Modus, werden die Simulationsdurchführungen von einem einzigen Kernel ausgeführt oder es werden einzelne Kernel für jede Simulationsdurchführung in einem eigenen Stream ausgeführt.

Erste Ebene der Simulation – Implementierung der Initialisierungsphase

```
/* Feststellung des Zeitpunktes zu Beginn der Simulation */
ftime(&instance_start_time);
ftime(&step_start_time);

/* Allokation und Erzeugung von Zufallszahlen auf der GPU  */
allocateSetupStates();

/* Erzeugung der Datenstruktur für den gitterbasierten Ansatz */
initGrid();
/* Initialisierung der Datenstruktur für den gitterbasierten Ansatz */
updateGrid();

/* Initialisierung des Modells */
computeForcesMoments();

/* Erkennung von Bindungen in Form von Clustern */
detectClusters();

/* Erzeugung eines Zwischenstandes – Schreiben der Daten in Dateien */
createIntermediateResult();
```

Algorithmus 4: Erste Ebene der Simulation – Implementierung der Initialisierungsphase

Erste Ebene der Simulation – Implementierung der Simulationsschleife

```
/* Beginn der Simulationsschleife */
t = 0.0;
sample_counter = 0;
for(int i=0; i<=simulation_cycles; i++){

  /* Berechnung der Positionen der Partikel*/
  computePositions();
  /* Berechnung der Kräfte und Drehmomente*/
  computeForcesMoments();

  /* Aktualisierung der Datenstruktur des gitterbasierten Ansatzes */
  updateGrid();

  /* Falls eine gewünschte Zahl von Simulationsschritten erreicht wurde,
     wird ein Zwischenstand erzeugt */
    if (sample_counter == intermediate_result_trigger) {
    sample_counter = 0;
```

```
      /* Erkennung von Bindungen in Form von Clustern */
      detectClusters();

      /* Lesen der Daten von der GPU und Schreiben in Dateien*/
      createIntermediateResult();

      /* Senden von Zwischenstandsinformationen an die
         Benutzerschnittstelle – (gekürzt) */
     ftime(&current_time);
     time_diff = timediff(&current_time, &step_start_time);
     notify(SIM_TIME, "Elapsed time for simulation step: ", &time_diff);

      /* Restart timing information */
     ftime(&step_start_time);
}

t += delta_t;
sample_counter++;
}
```

Algorithmus 5: Implementierung der Simulationsschleife

Berechnung der Position – Beispiel für eine Funktion der Simulationsschleife

```
int  number_gpus = GetNumberofGPUs();
int  current_gpu;
bool useGrid;
int  threadsperblock;

/* Im Modus S_KERNEL_SCHEDULE führt ein Kernel eine Menge von
Simulationsdurchläufen aus */
if (smode == S_KERNEL_SCHEDULE)
{
  /* Verteilung der Aufgaben an alle verfügbaren GPUs */
  for (current_gpu = 0; current_gpu < number_gpus; current_gpu++){

      /* Für jeden Typ wird eine Liste bearbeitet */
      for(int j = 0; j < 3; j++){
```

```
          /* Aufruf des Kernels auf der GPU, der die Position berechnet */
          compX_LC_d <<< nBlocks, threadsperblock >>>
            (kernel_data[current_gpu],
             useGrid,
             j,
             delta_t,
             devUniformRandoms,
             devNormalRandoms);
        }
    }
}
/* Im Modus S_HARDWARE_SCHEDULE führt ein Kernel einen einzigen
Simulationsdurchlauf aus */
else if (smode == S_HARDWARE_SCHEDULE)
{
   /* Verteilung der Aufgaben an alle verfügbaren GPUs */
   for (current_gpu = 0; current_gpu < number_gpus; current_gpu++){
     if(useGPU[current_gpu] == true){

       /* Für jede Instanz wird ein eigener Kernel gestartet */
       for(i = 0; i < #Instanzen; i ++){

         /* Für jeden Typ wird eine Liste bearbeitet */
         for(int j = 0; j < 3; j++){
```

Algorithmus 6: Beispiel für eine Funktion der Simulationsschleife

5.8 Die Abbildung der Partikelsimulationsschritte

Für die Abbildung der Simulation auf CUDA Threads wurde eine 1:1-Abbildung gewählt. Jeder Thread führt für ein Partikel die Interaktionen mit der Menge der übrigen Partikel aus. Zu einem Zeitpunkt werden alle Threads genau ein explizites Partikel betrachten und auf parallele Art die Interaktion mit diesem Partikel und des Partikels, das auf diesen Thread abgebildet wurde, durchführen. In Abbildung 28 ist die dabei entstehende Parallelität dargestellt. Die Menge der Interaktionen wird je Spalte auf die Threads aufgeteilt.

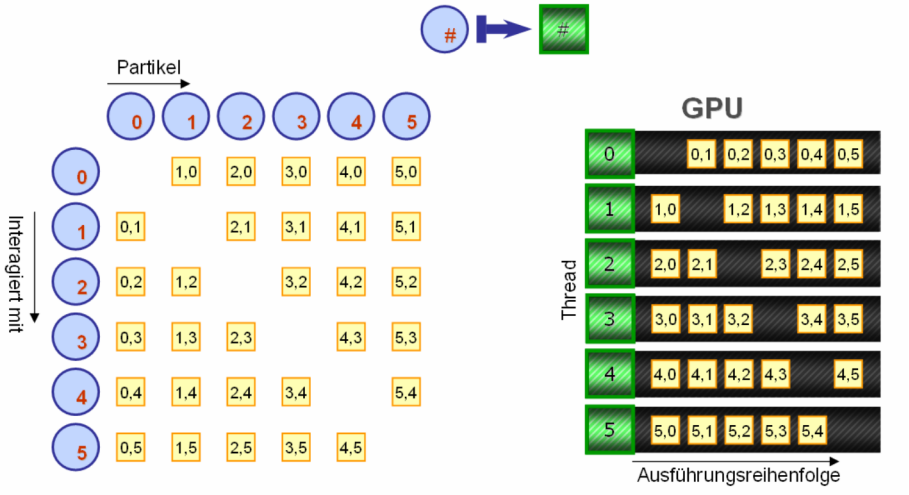


Abbildung 26: Abbildung der Partikel auf CUDA Threads

5.9 Die Implementierung der Kernel

Die Simulation weist innerhalb eines Simulationsschrittes folgende Aufgaben jeweils einer eigenen Gruppe von Kerneln zu:

- Berechnung von Kräften und Momenten
- Positionsberechnungen
- Bindungsuntersuchung in Form von Clusterung
- Verwaltung des gitterbasierten Ansatzes

Die Partikel interagieren untereinander. Unterschiedliche Partikeltypen interagieren auf unterschiedliche Weise mit einander. Daher gibt es für jede Konstellation eine eigene Gruppe von Kernel, die speziell für die jeweiligen Fälle angepasst wurde. Jedoch interagieren nicht alle beliebigen Kombinationen miteinander, da das Simulationsmodell nicht für alle Partikelkombinationen eine Interaktion vorsieht. Folgende Tabellen zeigen die möglichen Interaktionen:

	Monomere *M*	Dimere *D*	Trimere *T*	Liganden *L*
Monomere *M*	*MM*			*ML*
Dimere *D*				*DL*
Trimere *T*				*TL*
Liganden *L*				

Abbildung 27: Berechnung von Kräften und Drehmomenten sowie Clusterbildung

	Monomere *M*	Dimere *D*	Trimere *T*	Liganden *L*
Monomere *M*		*MD*		
Dimere *D*		*DD*		
Trimere *T*			*TT*	
Liganden *L*				*LL*

Abbildung 28: Ausschließliche Berechnung von Kräften

Kräfte-, Drehmoment- und Clusterkernel bestehen aus jeweils aus drei Komponenten in der Implementierung:

- Kernaufgabe in einer Funktion - Beispiel: Berechnung der Kräfte anhand der Modellvorschriften
- Kernel, der mehrere Simulationsdurchläufe verwaltet
- Kernel, der nur einen einzigen Simulationsdurchlauf verwaltet

Diese Aufteilung wurde gewählt, um die Redundanz von Quellcodesegmenten zu reduzieren. Die Kernel, die die Simulationsdurchläufe verwalten, besitzen eine Initialisierungsphase, in denen sie die benötigten Daten aus dem globalen Speicher laden. In der zweiten Phase übernehmen sie dann die Verwaltung des gitterbasierten Ansatzes und wenden im dritten Schritt die Funktion auf die gesammelten Daten an.

In der Initialisierungsphase wird das zugehörige Partikel bestimmt. Hierzu wird der Index des Threads ausgelesen und direkt auf die Liste der Partikel abgebildet.

Initialisierungsphase eines Kernels für einen Simulationsdurchlauf

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
SParticle ** particles = ins_data->mparticles;

if(idx >= ins_data->NumofParticles[type1])
  return;

extern __shared__ SParticle current_particle[];
current_particle[threadIdx.x] = particles[type1][idx];
```

Algorithmus 7: Initialisierungsphase eines Kernels für einen Simulationsdurchlauf

Die Initialisierungsphase des Kernels, der mehrere Kernel betrachtet, wird um die Findung des zugehörigen Simulationsdurchlaufs erweitert. Es wird angenommen, dass sämtliche Simulationsdurchläufe in ihren Typen genau die gleiche Anzahl an Partikel aufweisen. Mit dieser Zahl bestimmt der Kernel den zugehörigen Simulationsdurchlauf und das dazugehörige Partikel.

Initialisierungsphase eines Kernels für mehrere Simulationsdurchläufe

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
int instanceidx = idx / data.threadsperinstance;
idx = idx % data.threadsperinstance;
SParticle ** particles = data.ins_data[instanceidx].mparticles;

if(idx >= data.ins_data[instanceidx].NumofParticles[type1])
  return;

extern __shared__ SParticle current_particle[];
current_particle[threadIdx.x] = particles[type1][idx];
```

Algorithmus 8: Initialisierungsphase eines Kernels für mehrere Simulationsdurchläufe

Der Schritt der Bestimmung des zugehörigen Simulationsdurchlaufs fällt bei der Benutzung der Fermi-Eigenschaft der parallelen Ausführung von Kernel weg und sollte daher weniger Rechenzeit benötigen, da in jedem Kernelaufruf weniger Instruktionen verarbeitet werden.

Die Kernel, die die Simulationsdurchläufe verwalten, rufen im letzten Schritt eine Funktion auf, die die Modellvorschriften auf die Partikel anwendet. Dazu bekommt sie die Daten des zugehörigen Partikels und des Typs geliefert, worauf die Interaktionen angewendet werden sollen. Für die Anwendung der Interaktionen iteriert diese Funktion über die Daten sämtlicher Partikel des Zieltyps.

Iteration eines Kernels über die Menge aller Partikel

```
for(int i = start; i < end; i++){

  /* Ein Partikel darf nicht mit sich selbst interagieren */
  if(idx == i)continue;

  /* Berechnung des Abstandes zum aktuell betrachteten Partikel */
  real r = 0.0;
  real r1 = (particles[type2][i].x[0] - current_particle[threadIdx.x].x[0]);
  real r2 = (particles[type2][i].x[1] - current_particle[threadIdx.x].x[1]);

  r += r1 * r1;
  r += r2 * r2;

  if( r<=RCUT_SQ  && r>=RMIN_SQ ){
    /* Führe Interaktion
       (Kräfteberechnung,
       Drehmomentberechnung,
       Clustererkennung) durch */
  }
}
```

Algorithmus 9: Iteration über die Menge aller Partikel

5.10 Arithmetische Optimierungen

Die Modellvorschriften wurden aus der seriellen Version der Partikelsimulation übernommen und optimiert. Das Hauptaugenmerk lag auf der Vermeidung redundanter Berechnungen und auftretender Kontrollflussverzweigungen (engl. Branches).

- Die meisten Partikeleigenschaften sind Vektoren. Diese Vektoren bestehen aus mindestens zwei Komponenten. Die Berechnungen der meisten Komponenten setzen sich weitestgehend aus den gleichen Operationen zusammen, wobei sie sich nur im Einsatz von trigonometrischen Funktionen oder Vorzeichenvertauschungen unterscheiden. Aus diesem Grund wurden die Modellvorschriften auf redundante Teilvorschriften untersucht und diese entfernt.

- Innerhalb einer Vorschrift treten redundante Teilvorschriften auf. Dies sind im Wesentlichen Differenzen von Winkeln und die mehrfache Verwendung einer Potenz unterschiedlichen Grades. Diese Teilvorschriften wurden dahingehend optimiert, dass vorberechnete Teilergebnisse verwendet werden.

- Die Vermeidung von Branches unter der Verwendung unter CUDA ist ein vorrangiges Ziel. Treten innerhalb eines Warps Branches auf, so wird die Ausführung der unterschiedlichen Branches auf eine sequentielle Art durchgeführt. Die Auswertung der Bedingungen und das Abschalten von Instruktionspfaden verzögert die gesamte Ausführung. Daher wurden Branches in Modellvorschriften durch arithmetische Vergleichsoperationen ersetzt, die die einzelnen Threads parallel ausführen.

Beispielhafte Optimierung einer Berechnung von Kräften

```
/* Anfangszustand */
current_particle[threadIdx.x].F[0] += cos(phii) * 1.0/r*POT*(2.0*
pow(SIGMA_M/r,POT)* pow(SIGMA_M/r,POT)-
ALPHA_N*pow(SIGMA_M/r,POT)*heaviside_dev(DELTA-abs(phii-phi0i)));;
current_particle[threadIdx.x].F[1] += sin(phii) * 1.0/r*POT*(2.0*
pow(SIGMA_M/r,POT)* pow(SIGMA_M/r,POT)-
ALPHA_N*pow(SIGMA_M/r,POT)*heaviside_dev(DELTA-abs(phii-phi0i)));;

/* Optimierung */
temp1 = pow(SIGMA_M/r,POT);
temp2 = 1.0/r*POT*(2.0*temp1*temp1-ALPHA_N*temp1*heaviside_dev
(DELTA-abs(phii-phi0i)));
current_particle[threadIdx.x].F[0] += cos(phii) * temp2;
current_particle[threadIdx.x].F[1] += sin(phii) * temp2;
```

Algorithmus 10: Beispielhafte Optimierung einer Berechnung von Kräften

Beispielhafte Optimierung zur Vermeidung von Branches

```
/* Optimierung der Heaviside-Funktion */
/* Anfangszustand */
if (t>=0){
  return 1.0;
} else {
  return 0.0;
}

/* Optimierung */
return (t >= 0)*1.0;
```

Algorithmus 11: Beispielhafte Optimimierung zur Vermeidung von Branches

5.11 Die Erzeugung von Zufallszahlen

Um die Eigenschaften der Brown'schen Molekularbewegung auf die Bewegung der Partikel übertragen zu können, kann die einfache Erzeugung von pseudo-zufälligen Zahlen seitens der Hardware nicht verwendet werden, da die Zufallszahlen eine zu hohe Abweichung gegenüber der Normalverteilung aufweisen. Auf Basis der Anwendung des zentralen Grenzwertsatzes wird in der Simulation der Partikelbewegung die Summe von zwölf unabhängigen Zufallszahlen gebildet und als Zufallszahl benutzt, um die Normalverteilung der Zufallszahlen annähern zu können.

Erzeugung einer Zufallszahl – Basierend auf dem zentralen Grenzwertsatz

```
/* Erzeugung einer Zufallszahl */
x = 0.0;
for (int k = 0; k < 12; k++){
  x += curand_uniform_double();
}
x -= 6.0;

/* Speicherung der Zufallszahl */
result[idx] = x;
```

Algorithmus 12: Erzeugung einer Zufallszahl – Basierend auf dem zentralen Grenzwertsatz

Zur Erzeugung dieser pseudo-zufälligen Zahlen wird die CURAND Library [29] verwendet. Hierbei wird ein Array von Zuständen erzeugt, woraus Sequenzen von zufälligen Zahlen berechnet werden. Hierzu wird die Funktion *curand_uniform_double* verwendet, die eine gleichverteilte pseudo-zufällige Zahl zwischen 0.0 und 1.0 zurückliefert. Die Zustände sind an die Partikel gebunden, um die Normalverteilung der Zufallszahlen zu bewahren. Diese Bindung wird durch das Mitsortieren der Zustände beim Sortieren der Partikel erreicht. In der Entwicklungsphase wurde diese Sortierung nicht durchgeführt, wodurch die Partikel bei einer Neusortierung anderen Zuständen zugewiesen wurden. Die Folge war das Auftreten eines Phänomens, bei dem eine Agglomeration von Partikeln an den Rändern der Bounding-Box beobachtet werden konnte.

5.12 Die Gewährleistung der Partikelposition innerhalb der Bounding-Box im numerischen Kontext

Die Partikel bewegen sich nicht frei in der Simulationsdomäne. Sie werden durch eine Bounding-Box in einem begrenzten Bereich gehalten. Wenn sie aufgrund der Simulation der Braun'schen Molekularbewegung oder des Einflusses einer Kraft diesen Bereich verlassen, werden sie jeweils auf der gegenüberliegenden Seite der Bounding-Box wieder eingesetzt. Intuitiv betrachtet verhält sich die Fläche der Bounding-Box wie die Oberfläche einer Kugel.

In der vorliegenden Studienarbeit gab es die Anforderung zur Optimierung der Simulation durch eine deutliche Verkleinerung der Bounding-Box. Die intuitive Lösung durch einfache Anwendung der Modulo-Operation kann numerische Probleme auslösen. Dies wird bei der Betrachtung der zugehörigen Anweisungen deutlich.

Grundlegende Anweisung zur Gewährleistung der Bounding-Box-Eigenschaft

```
particles[idx].x[0] -=
floor(particles[idx].x[0]/bounding_box_width)*bounding_box_width;
```

Algorithmus 13: Grundlegende Anweisung zur Gewährleistung der Bounding-Box-Eigenschaft

Ist der Wert für die Variable bounding_box_width kleiner als 1, so wird beim Teilvorgang innerhalb der Funktion floor das Zwischenergebnis größer als der ursprüngliche Operand. Falls sich diese Zahl nahe der maximal darstellbaren Zahl (aus numerischer Sicht im Kontext von Gleitkommazahlen) befindet, können Überläufe entstehen. Diese werden von der GPU nicht vollständig abgefangen und resultieren daher in fehlerhaften Ergebnissen. Konkret bedeutet dies, dass die Bounding-Box-Eigenschaft in diesen Fällen verletzt wird.

Um dieses Problem zu lösen, wurden Anweisungen zur Bewahrung der Bounding-Box-Eigenschaft implementiert, die die Randfälle in der Gleitkommazahlen-Behandlung abfangen.

- Fälle in denen Partikel trotz der oben genannten Anweisung außerhalb der Bounding-Box liegen.
- Fälle, in denen die gesamte Berechnung fehlschlägt und NaN (Not a Number) produziert wird.

Hinreichende Anweisung zur Gewährleistung der Bounding-Box-Eigenschaft

```
particles[idx].x[0] *=  1.0*( particles[idx].x[0] < bounding_box_width &&
                              particles[idx].x[0] >= 0.0);
if(isnan(particles[idx].x[0]))
  {particles[idx].x[0] = zvx[0] * bounding_box_width * (1-RCUT);}
```

Algorithmus 14: Hinreichende Anweisung zur Gewährleistung der Bounding-Box-Eigenschaft

5.13 Der gitterbasierte Lösungsansatz

5.13.1 Grundlagen

Die Auswirkungen der Interaktionen zwischen den Partikeln hängen von der Distanz ab. Mit zunehmender Distanz verringert sich der dabei entstehende Einfluss auf Kräfte, Drehmomente und Bindungschancen. Diese Verringerung kann bis zur vollständigen Aufhebung einer gegenseitigen Wirkung gehen. Daher wurde im Simulationsmodell ein Schwellwert eingeführt, mit dem die maximale Distanz angegeben wird, unter denen Partikel interagieren. Dieser wird mit der Konstante RCUT angegeben und bezeichnet den Berechnungshorizont eines Partikels. Aus dieser praktischen Überlegung heraus wurde zur Optimierung des Speedup der Lösungsansatz eines gitterbasierten Ansatzes gewählt.

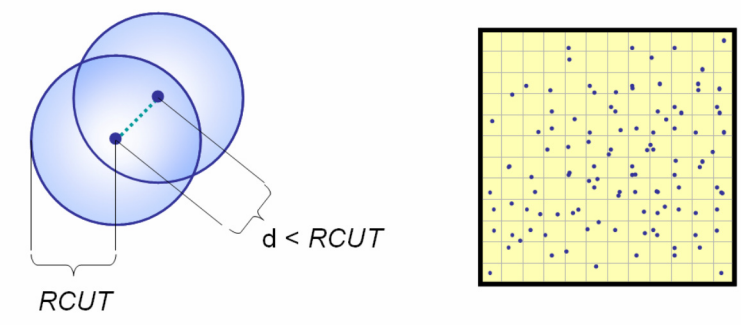


Abbildung 29: Berechnungshorizont und Unterteilung in äquidistantes Gitter

Für die Implementierung wurde ein äquidistantes Gitter gewählt, da der Berechnungshorizont kreisförmig um ein Partikel aufgespannt ist und die Berechnungsschritte zur Verwaltung des Gitters implizit berechnet werden können. Die Zugehörigkeit eines Partikels zu einer Gitterzelle kann direkt aus dessen Position errechnet werden und muss nicht gespeichert werden. In Folge dessen werden zusätzliche Speicheranfragen vermieden.

Für alle Partikel wird die Menge der möglichen Interaktionspartner reduziert. Es werden pro Partikel nur noch die Partikel der eigenen und direkt umgebenden Gitterzellen untersucht.

5.13.2 Der Prototyp basierend auf dem CUDA SDK-Beispiel „Particles“

Zur Implementierung dieser Optimierung wurde eine Orientierung am Beispiel „Particles“, dessen Quellcode in der CUDA SDK Library vorhanden ist, vorgenommen [26]. Für die Implementierung der Datenstruktur, die das Gitter verwaltet, gibt es zwei Lösungsansätze, die auf den verschiedenen Architekturen (CPU & GPU) durch ihre jeweiligen Eigenschaften spezifische Eignungen aufweisen.

- Im ersten Lösungsansatz wird eine zweidimensional verkettete Liste verwendet. In den Blättern dieser Liste befinden sich Zeiger auf die Partikeldaten. Diese Zeiger sind in einfach verketteten Listen angeordnet, wobei jede dieser Listen die Partikel einer Gitterzelle verwaltet. Die Anker der Listen der einzelnen Gitterzellen werden in einer einfach verketteten Liste gehalten.

- Der zweite Lösungsansatz sieht eine Sortierung der Partikeldaten anhand eines Gitterzellenindizes vor. Dabei wird jeder Gitterzelle ein Index zugewiesen, der als Hashwert aus der Position der darin enthaltenen Partikel berechnet werden kann. Die Partikeldaten befinden sich in einer einfachen Liste, in den keine Zeiger verwendet werden. Nach einer Sortierung finden sich die Partikel einer Gitterzelle als geschlossene Teilliste innerhalb dieser Liste wieder. Für jede Gitterzelle muss lediglich der Start und der End-Index innerhalb dieser Liste gespeichert werden.

In der sequentiellen Implementierung für eine CPU-Architektur wurde der erste Lösungsansatz gewählt. Basierend auf der Existenz großer Caches samt der Eigenschaft der Verwendung größerer Cacheblöcke kann auf dieser Architektur ein Speedup erreicht werden, da ein Overhead nur durch die Umverteilung von Zeigern auf den Partikeldaten entsteht. Auf einer GPU-Architektur stehen diese Cachearchitekturen nur bedingt oder schlicht überhaupt nicht zur Verfügung. Daher muss das Ziel einer größtmöglich hohen räumlichen Kohärenz erreicht werden, damit Speicherzugriffe, die große Datenmengen parallel laden, effizient werden. Dieser Aspekt wird noch deutlicher beim Einsatz von Caches, wie sie bei der Fermi-Architektur zur Verfügung stehen. Hier führt die Ausnutzung von räumlicher Kohärenz zu einer deutlichen Verbesserung in der Effizienz.

Eine Implementierung des zweiten Lösungsansatzes findet sich im eben erwähnten Particles-Beispiel wieder. Die relevanten Ideen zur Implementierung werden im Folgenden erläutert.

Die Berechnung des Gitterzellenindexes wird für jedes Partikel direkt aus dessen Position bestimmt. Dabei werden die Indizes fortlaufend zeilenweise vergeben.

Berechnung des Gitterzellenindexes

```
int calcHash_d(real x, real y, real width, int numcells){
  return (int) (x/width) + numcells * (int)(y/width);
}
```

Algorithmus 15: Berechnung des Gitterzellenindex

Bei einem Neuaufbau des Gitters werden die Partikeldaten anhand des zugehörigen Gitterzellenindexes sortiert. Bei dieser Sortierung wird der Gitterzellenindex als Hashwert jedes Partikels genutzt Der Sortiervorgang teilt sich in zwei Phasen auf:

1. Sortierung der Liste der Partikelindizes anhand der berechneten Hashwerte der Partikel. Hierzu wird das Radixsort-Sortierverfahren angewandt, dessen Algorithmus ein hohes Potential für eine Parallelisierung aufweist. In der Implementierung wird die Radixsort-Implementierung der Thrust-Library angewandt. In der Dokumentation dieser Implementierung behaupten die Entwickler die „schnellste veröffentlichte Implementierung von Radix-Sort“ entwickelt zu haben. Diese Implementierung verfügt über eine lineare Laufzeit. [27]

2. Sortierung der Partikeldaten anhand der vorsortierten Liste der Partikelindizes.

Zum Abschluss des Aufbaus des Gitters werden die Listen erzeugt, in denen die Start- und End-Informationen jeder Gitterzelle gespeichert werden. Hierzu vergleicht jedes Partikel seinen Gitterzellenindex mit dem Gitterzellenindex seines Nachbarpartikels. Da die Partikel sortiert nach diesem Gitterzellenindex in der Liste stehen, bedeutet ein Unterschied zwischen den Gitterzellenindizes einen Übergang von einer Gitterzelle zur nächsten.

Berechnung der Start- und End-Information der Gitterzellen

```
extern __shared__ int sharedHash[];
int hash;

hash = hashes[type][idx];
sharedHash[threadIdx.x+1] = hash;

if (idx > 0 && threadIdx.x == 0){
  sharedHash[0] = hashes[type][idx-1];
}

if (idx == 0 || hash != sharedHash[threadIdx.x])
{
   gridcellbegin[type][hash] = idx;
   if (idx > 0)
     gridcellend[type][sharedHash[threadIdx.x]] = idx;
}
if (idx == numParticles - 1)
      mgriddata.gridcellend[type][hash] = idx + 1;
```

Algorithmus 16: Berechnung der Start- und End-Information der Gitterzellen

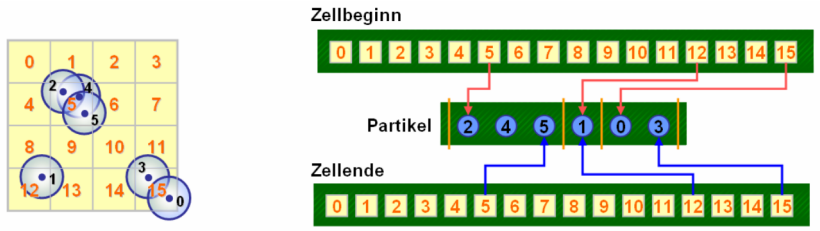


Abbildung 30: Datenstruktur bei der Verwendung eines äquidistanten Gitters

Aufbau des Gitters – Sortierung der Partikeldaten

```
/* Auslesen der Informationen für situationsabhängiges Sortieren */
for (current_gpu = 0; current_gpu < number_gpus; current_gpu++){
  cudaMemcpyAsync(change, ins_data->change,
  cudaMemcpyDeviceToHost);
}

/* Sortieren der Partikelindizes */
for (int j = 0; j < 3; j++){
  for (current_gpu = 0; current_gpu < number_gpus; current_gpu++)
    for (int i = 0; i < #instances; i++){
      if (change[i*3+j] == false) continue;
        thrust::sort_by_key(ins_data->hash[j],
                            ins_data->hash[j]+number_of_particles_of_type)
                            ins_data->newindex[j]);
      }
    }
  }
}

/* Sortieren der Partikeldaten */
for (int j = 0; j < 3; j++){
  for (current_gpu = 0; current_gpu < number_gpus; current_gpu++){
    SortParticles_d <<<  nBlocks, threadsperblock >>>
      (*k_data[current_gpu],j);
  }
}
```

Algorithmus 17: Aufbau des Gitters – Sortierung der Partikeldaten

5.13.3 Implementierte weitergehende Optimierungen

Die erläuterte Implementierung enthält außer der Erhöhung der räumlichen Kohärenz bei Speicherzugriffen keine weiteren Optimierungen zur Reduzierung der Berechnungszeit. Daher wurden im Rahmen dieser Studienarbeit Potentiale zur Optimierung untersucht und implementiert, die im Folgenden vorgestellt werden.

Die erste im Rahmen dieser Studienarbeit entwickelte Optimierung dient der Reduzierung der Sortiervorgänge in Form einer **fallbedingten Sortierung**. Die Partikel

bewegen sich nach den Gesetzen der Brown'schen Molekularbewegung durch das Gitter. In Simulationsschritten, in denen kein Partikel eine Gitterzelle verlässt, ist ein Sortiervorgang unnötig und kann vermieden werden. Daher wird bei der Berechnung der Bewegung eines Partikels der neue Hashwert mit dem bisherigen verglichen. Falls eine Änderung auftritt, wird die Variable „change" des zugehörigen Gitters gesetzt. Im Sortiervorgang wird diese Variable überprüft und in Abhängigkeit des Zustandes eine Sortierung durchgeführt oder übersprungen. Aus dieser Optimierung heraus entsteht in direkter Folge eine weitere Untersuchung, die die Größe einer einzelnen Gitterzelle fokussiert. Folgende Überlegungen müssen bei der Wahl der Gittergröße getroffen werden:

- Große Gitterzellen reduzieren die Anzahl der Sortierungen. In diesem Fall ist die Wahrscheinlichkeit geringer, dass Partikel in einem Simulationsschritt eine Gitterzelle verlassen. Daher wird seltener ein Sortiervorgang durchgeführt. Zu große Zellen erhöhen den Berechnungsaufwand, da zu viele Partikel pro Gitterzelle existieren, die keine Interaktionen durchführen. In diesem Falle werden unnötige Iterationen durchgeführt.

- Kleine Gitterzellen reduzieren die Anzahl der Iterationen über Partikel der betrachteten Gitterzelle. Zu kleine Gitterzellen erhöhen die Anzahl der Sortierungen, da die Wahrscheinlichkeit für einen Austritt aus einer Gitterzelle pro Partikel steigt.

Als Folge der Überlegung muss für eine Simulationsdurchlaufskonfiguration eine **optimale Gitterzellengröße** bestimmt werden. Beispielsweise liegt das Optimum bei einer Konfiguration von Monomeren und Liganden mit je 1344 Partikeln bei 400 Zellen. Bei einer gleichmäßigen Verteilung liegen in einer Zelle 3-4 Partikel. Die Größenverhältnisse der Berechnungshorizonte und Gittergrößen dieses Beispiels wurden in Abbildung 31 dargestellt.

Im Laufe der Untersuchung der Optimierungen wurde ein **paralleles Sortieren** aller vorhandenen Gitter der Simulationsdurchläufe untersucht. Wie bereits erläutert, wird der eigentliche Sortiervorgang von der Radixsort-Implementierung der Thrust Library durchgeführt. In der ersten Implementierung wurde jedes Gitter einzeln in einer sequentiellen Form an diese Implementierung übergeben. In der Untersuchung wurden die Datenstrukturen dahingehend erweitert, die gesamte vorhandene Datenstruktur und somit alle Gitter gleichzeitig von Radix-Sort sortieren zu lassen. Hierzu wurde ein Offset-Wert für jedes weitere Gitter eingesetzt damit die Indizes im 2. Schritt der Sortierung voneinander unterschieden werden können. Ohne diesen Offset-Wert würden die Gitterzellenindizes mehrfach vorkommen, wodurch keine Unterscheidung zwischen den Partikeln der einzelnen Gitter mehr möglich wäre.

Dieser Optimierungsversuch wurde nicht weiter verfolgt, da der Nutzen dem sequentiellen Sortieren der Gitter mit Radixsort unterliegt. Dies begründet sich in der linearen Laufzeit von Radixsort. Eine parallele Sortierung mehrerer Gitter in einem Sortierdurchlauf verglichen mit der sequentiellen Sortierung mehrerer Gitter in einer Vielzahl von Sortierschritten zeigt eine nahezu identische Berechnungszeit auf. Der

klare Vorteil der sequentiellen Sortierung rührt von der ersten Optimierung, die es ermöglicht, Sortierungen zu vermeiden, wenn sie nicht benötigt werden. Der Nutzen, der durch den Einsatz einer Vorbereitung hinsichtlich des situationsabhängigen parallelen Sortierens entstehen würde, könnte bedingt aus der linearen Laufzeit von Radixsort das sequentielle Sortieren nicht übertreffen. Der Quellcode, der die parallele Sortierung durchführt, existiert für weitere Untersuchungen in auskommentierter Form im Quellcode der Klasse CudaCommunication.

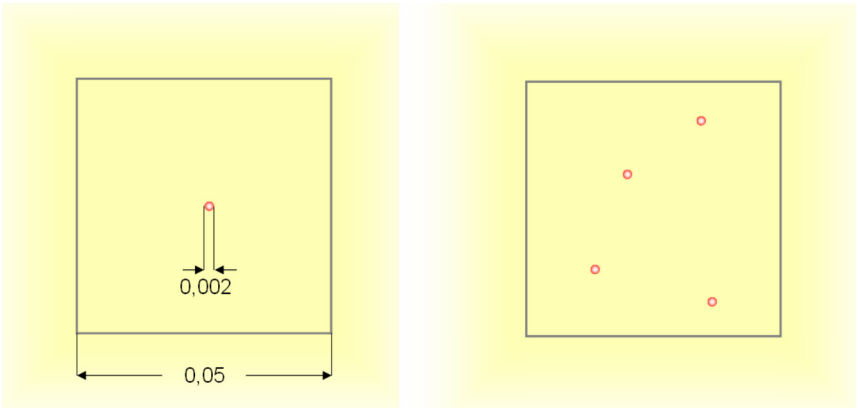


Abbildung 31: Partikel in einer Gitterzelle mit dargestelltem Berechnungshorizont

Die Partikel können neben den Interaktionen innerhalb ihrer Gitterzelle auch Interaktionen mit Partikeln aus den umliegenden neun Gitterzellen ausführen. Im Particles-Beispiel betrachten alle Partikel zu einem Zeitpunkt dieselbe Richtung. Zeilenweise werden alle Nachbarzellen abgearbeitet.

Diese Herangehensweise führt zu einem deutlichen Overhead an Speicheranfragen. Wie man in Abbildung 32 erkennen kann, werden die Partikel jeder Gitterzelle im Laufe der Behandlung der Nachbarzellen neun Mal abgefragt. In dieser Abbildung wurde ein 3x3 Ausschnitt gewählt, um zu verdeutlichen, wie das Particles-Beispiel die Nachbarschaftsbehandlung durchführt.

In einem **ersten Optimierungsschritt** wurde die Durchführung der Nachbarschaftsbehandlung dahingehend koordiniert, die Anzahl der Partikeldatenabfragen zu reduzieren. Dies wird in Abbildung 33 verdeutlicht. Nach dieser Optimierung wird jedes Partikel genau ein einziges Mal gelesen.

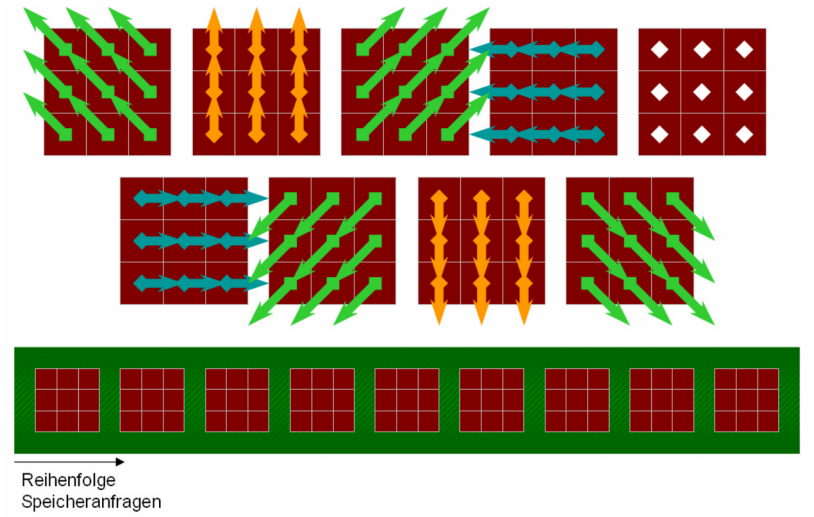


Abbildung 32: Durchführung der Nachbarschaftsbehandlung im Particles-Beispiel

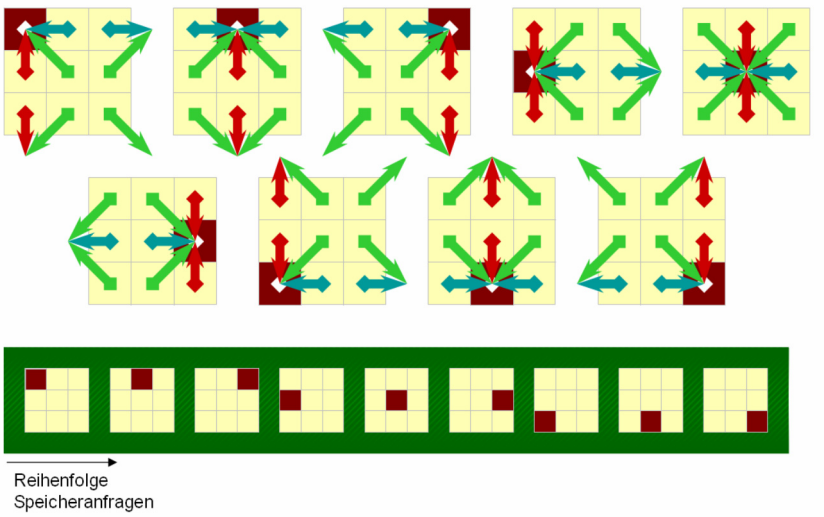


Abbildung 33: Durchführung der Nachbarschaftsbehandlung nach der 1. Optimierung

In einem zweiten Optimierungsschritt wurde die Nachbarschaftsbehandlung aus der Sicht eines Partikels innerhalb einer Gitterzelle genauer untersucht. Ein Partikel kann zu Partikeln aus maximal drei benachbarten Gitterzellen Interaktionen durchführen. Dies wird in Abbildung 34 deutlich. Darin wird deutlich, dass insgesamt vier mögliche Fälle für die Lage des Berechnungshorizontes eines Partikels existieren.

Basierend auf dieser Überlegung, kann eine Gitterzelle in vier Quadranten zerlegt werden. Die Partikel der einzelnen Quadranten müssen maximal die Partikel von drei Nachbarzellen und der eigenen Zelle auslesen.

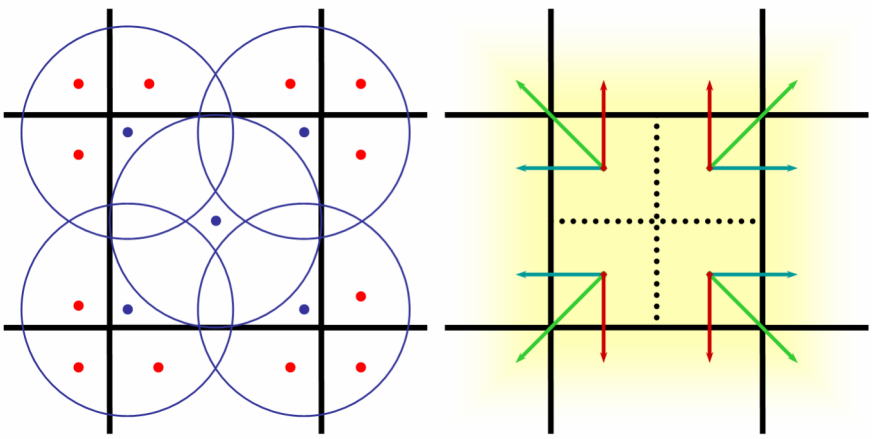


Abbildung 34: Unterteilung einer Gitterzelle in vier Quadranten

Als Folge dieses **zweiten Optimierungsschrittes** reduziert sich die Anzahl der Speicherzugriffe von neun auf vier. Diese Reduktion ist in Abbildung 35 dargestellt.

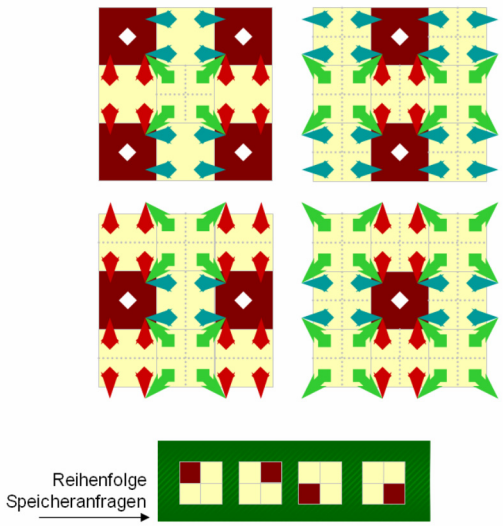


Abbildung 35: Durchführung der Nachbarschaftsbehandlung nach der 2. Optimierung

Für die Implementierung dieser Optimierung wurde ein Algorithmus entwickelt, der Branches vermeidet. Die Unterteilung der Partikel in ihre Quadranten und die Koordination der Partikel zum Lesen einer gemeinsam ausgewählten Zelle werden direkt aus dem Gitterzellenindex hergeleitet.

Zuweisung der Partikel zu Quadranten und gemeinsame Koordinierung

```
/* Berechnung des Quadranten innerhalb der Gitterzelle */
int xq = int(x[0]/(gridwidth/2.0)) % 2;
int yq = int(x[1]/(gridwidth/2.0)) % 2;

/* Koordination der Partikel – Ausrichtung auf die Zelle (1,1) in jedem 2x2-
   Teilgitter */
int xh  = 1-(hash % mgriddata.num_cells) % 2;
int yh  = 1-(hash /   mgriddata.num_cells) % 2;

real x_adj,y_adj,x_adjl,y_adjl;

for(int y=0; y<2; y++) {
  for(int x=0; x<2; x++) {

     /* Berechnung der Position der gemeinsam betrachteten Nachbarzelle */
    x_adj = x[0] + ((real)(-2 * xq + 1)*(xh - 1)) * gridwidth;
    y_adj = x[1] + ((real)(-2 * yq + 1)*(yh - 1)) * gridwidth;

    /* Berechnung des Endpunktes des Berechnungshorizontes */
    x_adjl = x[0] + ((real)(-2 * xq + 1)*(xh - 1)) * RCUT;
    y_adjl = x[1] + ((real)(-2 * yq + 1)*(yh - 1)) * RCUT;

    /* Falls der Berechnungshorizont in die Nachbarzelle reicht, werden
       Interaktionen durchgeführt */
    int neighbourhashl = calcHash_d(x_adjl,y_adjl);
    int neighbourhash  = calcHash_d(x_adj, y_adj);

    if (neighbourhashl == neighbourhash){
      /* Interaktionen zur Nachbarzelle */
    }
  }
}
```

Algorithmus 18: Zuweisung der Partikel zu Quadranten und gemeinsame Koordinierung

Als Konzept für eine weitere Optimierung wurde der Einsatz einer **raumfüllenden Kurve** für die Berechnung des Gitterzellenindexes untersucht. Hierzu wurde die Hilbert-Kurve eingesetzt, deren Implementierung in Algorithmus 19 beschrieben ist.

Berechnung des Gitterzellenindex nach Hilbert

```
void rot(int n, int *x, int *y, int rx, int ry) {
   int t;
   if (ry == 0) {
      if (rx == 1) {
        *x = n-1 - *x;
         *y = n-1 - *y;
      }
      t  = *x;
      *x = *y;
      *y = t;
   }
}
int calcHilbertHash_d(real x, real y, int n, real bounding_box_width){
                           int xd = (int)(x / bounding_box_width);
                           int yd = (int)(y / bounding_box_width);
   int rx, ry, s, d=0;
   for (s=n/2; s>0; s/=2) {
      rx = (x & s) > 0;
      ry = (y & s) > 0;
      d += s * s * ((3 * rx) ^ ry);
      rot(s, &x, &y, rx, ry);
   }
   return d;
}
```

Algorithmus 19: Berechnung des Gitterzellenindex nach Hilbert

Die Erwartung, die mit dieser Untersuchung verbunden war, lag in der gesteigerten räumlichen Kohärenz der Daten. Diese Anordnung mittels der Hilbert-Kurve ist der bisherigen zeilenweisen Anordnung überlegen. Die gesteigerte räumliche Kohärenz ist in der Abbildung 36 dargestellt.

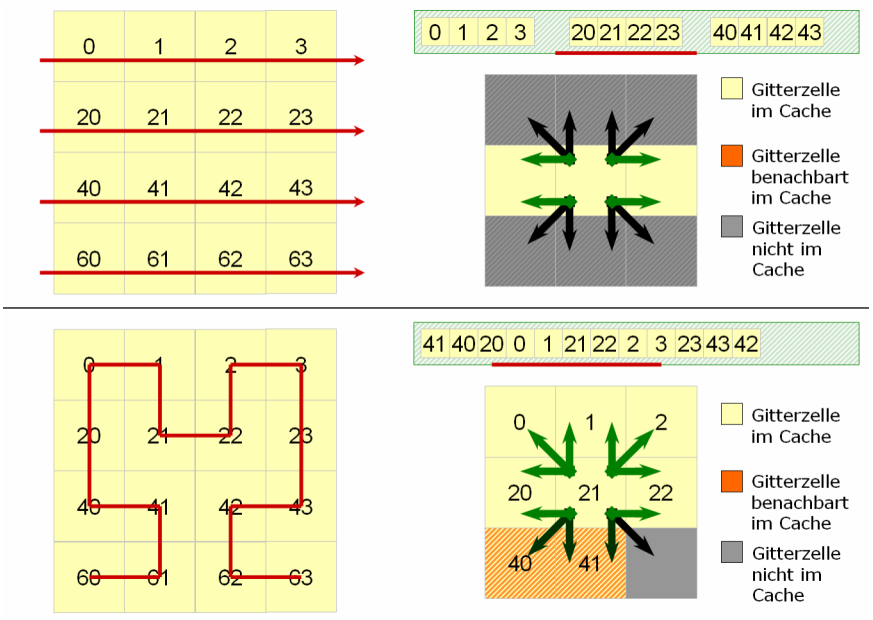


Abbildung 36: Vergleich der zeilenweisen Anordnung und der Hilbert-Kurve

Die Analyse der Berechnungszeit der Implementierung zeigte einen deutlichen Overhead durch die Berechnung des Hilbertkurven-Indexes auf. Die gesteigerte räumliche Kohärenz überwiegt nicht dem Overhead, der durch die aufwendigere Gitterzellenindex-Berechnung verursacht wird. Daher wurde diese Implementierung nicht weiter verfolgt.

5.14 Die Benutzerschnittstelle

Die implementierte Simulationsumgebung verfügt über eine Interaktion mit dem Benutzer über das Terminal. Die Funktionalität für die Benutzerinteraktion wurde in der Klasse CTerminalUI implementiert. Kern dieser Klasse ist die Methode run, die die Parameter von der Konsole einliest und daraus eine Menge von Simulationsdurchläufen generiert und ausführt. Innerhalb dieser Methode wird ein Objekt der Klasse CudaSimulation erzeugt, welches die Funktionalität der Simulationsverwaltung für den Einsatz der Applikationsschnittstelle CUDA ermöglicht.

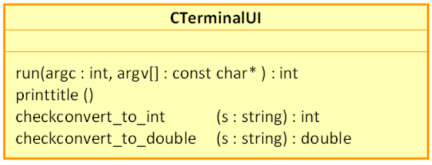


Abbildung 37: Klassendiagramm CTerminalUI

In dieser Klasse befindet sich auch eine Implementierung einer Funktion, die Information des Simulationsverlaufs im Terminal ausgibt. Diese Funktion „handleNotification“ wird als Funktionenzeiger an die Simulation übergeben. Die Parameter für die Erzeugung einer Menge von Simulationsdurchläufen sind in folgender Tabelle angegeben und erklärt.

Parameter	Erläuterung
-big	Es wird ein Gitter mit der Bounding-Box-Größe 10 verwendet. Die Gitterzellenweite beträgt 0,05
-small	Es wird ein Gitter mit der Bounding-Box-Größe 10 verwendet. Die Gitterzellenweite beträgt 0,05
-singlecell	Es wird ein Gitter mit der Bounding-Box-Größe 1 verwendet. Die Gitterzellenweite beträgt 1
-bbw *width*	Es wird ein Gitter mit der Bounding-Box-Größe *width* verwendet.
-cw *width*	Die Gitterzellenweite beträgt *width*.
-scyc *cycles*	Es werden *cycles* Simulationsschritte durchgeführt.
-irt *trigger*	Nach *trigger* Simulationsschritten wird ein Zwischenergebnis erzeugt und in eine Datei geschrieben.
-m *number*	Es werden *number* Monomere angelegt.
-d1 *number*	Es werden *number* Dimere an Position 1 angelegt.
-d2 *number*	Es werden *number* Dimere an Position 2 angelegt.
-t *number*	Es werden *number* Trimere angelegt.
-l *number*	Es werden *number* Ligangen angelegt.
-numvar *variations*	Es werden *variations* Simulationsdurchläufe aus den angegebenen Parametern erzeugt und ausgeführt.
-parpart *number*	Es werden maximal *number* Partikel auf einer GPU behandelt.

Tabelle 2: Konsolenparameter für Parsim

5.15 Die Schnittstelle zum Dateisystem

Die Schnittstelle IFileManager stellt die Funktionalität für die Ausgabe der Simulationsdaten in Form von Dateien bereit. Dazu stellt diese Schnittstelle eine überladene Methode SimInstanceToFile zur Verfügung, die in vier Implementierungen eine flexible Ausgabe von einzelnen Simulationsdurchläufen oder einer Menge von Simulationsdurchläufen in Dateien zur Verfügung stellt. Neben der automatischen Vergabe von Dateinamen, dessen Modus über die Methode setFilenameMode bestimmt werden kann, stehen zwei weitere Methoden zur Verfügung, mit denen ein Dateiname explizit angegeben werden kann.

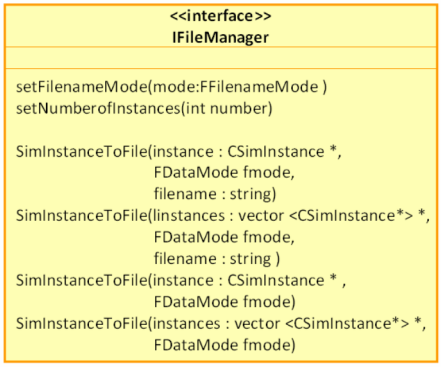


Abbildung 38: Klassendiagramm IFileManager

Für die Ausgabe der Simulationsdaten wurde die Weiterverarbeitung von Form einer Visualisierung in der Klasse CPlainFileManager berücksichtigt. Wie bereits erläutert, können andere Implementierungen der Schnittstelle andere Dateiformate für die Ausgabe der Simulationsdaten ermöglichen.

Das Dateiformat wurde folgendermaßen festgelegt:

X	Y	E[0]	E[1]	F[0]	F[1]	Typ	Index	#Bindungsregionen	D	B[0]	B[1]	B[2]

Tabelle 3: Dateiformat der Partikeldaten

Die Dateien der Simulation werden in einen Unterordner „Daten“ gespeichert. Dieser Ordner wird automatisch angelegt, sofern er nicht vorhanden ist. In diesem Unterordner werden für jeden Simulationsdurchlauf eigene Ordner angelegt, wobei jeder Ordner mit einer fortlaufenden Nummer benannt wird. Die Dateien eines Simulationsdurchlaufs werden nun fortlaufend nummeriert benannt und nach dem zuvor genannten Dateiformat gespeichert.

6. Analyse

6.1 Grundlagen

Wesentliches Ziel im Rahmen dieser Studienarbeit ist die Reduzierung des Berechnungsaufwandes der Simulationsanwendung. In den folgenden Analysen wird die Erreichung dieses Ziels untersucht. Dabei wird die Berechnungszeit der parallelen Simulation auf der GPU, deren Implementierung im vorherigen Kapitel erläutert wurde, mit der Berechnungszeit der sequentiellen Simulation auf der CPU unter gleichen Simulationsrahmenbedingungen verglichen, wobei der resultierende Speedup ermittelt wird.

Der in den folgenden Analysen ermittelte Speedup bezieht sich auf den Rechenzeitvergleich mit der sequentiellen Implementierung auf der CPU, die ebenfalls einen gitterbasierten Ansatz verwendet. Für eine optimale Vergleichbarkeit wurden für beide Implementierungen jeweils optimale Gitterparameter benutzt, aus denen minimale Berechnungszeiten folgen.

6.2 Untersuchungen verschiedener Anzahlen von Partikeln

Ein grundlegendes Resultat ist der Speedup, der bei der Betrachtung einer einzigen Simulationsausführung gemessen werden kann. Hierbei werden verschiedene Anzahlen von Partikeln auf der GPU und CPU simuliert und die resultierenden Berechnungszeiten verglichen. Der Speedup wächst proportional mit dem Anteil der Ausnutzung der vorhandenen Architektur. Bei der Nutzung der gesamten Menge der verfügbaren Threads wurde ein Speedup in der Größenordnung von 320x gemessen.[3]

In dieser Untersuchung werden die Vorteile durch die Nutzung der massiven Parallelität der GPU deutlich. Der Speedup wächst nahezu linear mit der Anzahl der verwendeten Partikel, wobei keine Sättigung festgestellt werden kann.

Bei der Betrachtung der Untersuchung zur sequentiellen Version kann ein nahezu quadratisches Wachstum der Berechnungszeit beobachtet werden. Dies begründet sich in Art der Menge der Interaktionen, die pro Simulationsschritt berechnet werden. Da jedes Partikel mit jedem anderen Partikel interagieren kann, entsteht eine quadratische Menge, die sich in quadratischer Form auf die Berechnungszeit auswirkt. (siehe Kapitel 5.2 „Grundlegende Implementierungen“)

[3] 320,8 – Bei einer Simulationskonfiguration von jeweils 21504 Monomeren und Liganden

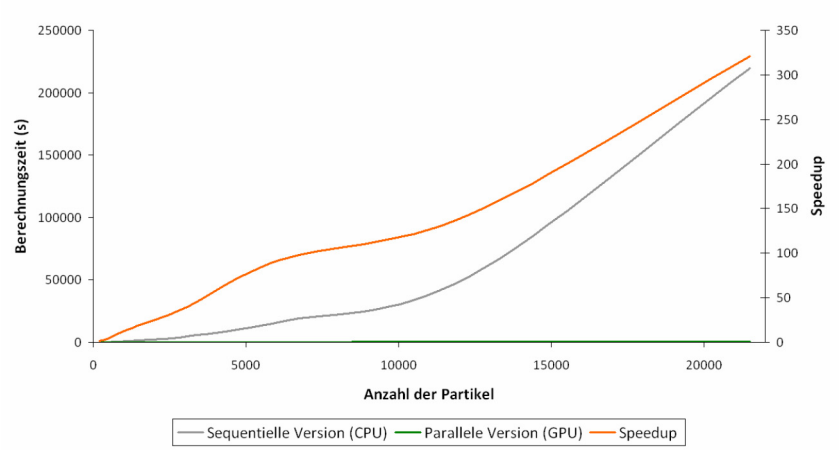


Abbildung 39: Berechnungszeit eines Simulationsdurchlaufs mit 100.000 Simulationsschritten bei verschiedener Anzahl von Partikeln

6.3 Untersuchungen verschiedener Anzahlen von parallel ausgeführten Simulationsdurchläufen

Eine Anforderung an die parallele Simulation ist neben dieser schlichten Abbildung eines einzigen Simulationsdurchlaufs auf eine GPGPU-Architektur die vollständige Nutzung dieser Architektur. Dazu wurde das Simulationsmodell um den Aspekt der parallelen Ausführung von mehreren Simulationsdurchläufen erweitert. Auf einer einzigen GPU, deren Architektur vollständig benutzt wurde, werden Speedups in der Größenordnung von 30x gemessen.[4]

In dieser Untersuchung wurde eine Simulationskonfiguration eingesetzt, in denen jeweils gleiche Anzahlen von Rezeptoren und Liganden verwendet werden. Hierbei wird eine Sättigung des Speedups ab der Verwendung von mehr als fünf parallel ausgeführten Simulationsdurchläufen beobachtet. Dies entspricht 6720 Partikeln oder 31% der genutzten Architektur.

Diese Sättigung basiert auf dem linearen Wachstum der Berechnungszeit der sequentiellen Version. Im Gegensatz zur Untersuchung im Kapitel 6.2 wächst die Menge der Interaktionen pro Simulationsschritt nicht quadratisch, sondern linear mit jedem weiteren betrachteten Simulationsdurchlauf. Bei der Verwendung der gesamten GPU-Architektur werden beispielsweise 21504 Partikel pro Partikeltyp eingesetzt. Die sequentielle Version auf der CPU benötigt zur Berechnung von 100.000 Simulationsschritten dieser Anzahl von Partikeln 219741 Sekunden (60,5 Stunden)[5]. Wird die gleiche Anzahl von Partikeln auf 16 Simulationsdurchläufe verteilt, benötigt diese Implementierung lediglich 20160 Sekunden (5,6 Stunden). Dies entspricht einer

[4] 28,9 – Bei einer Simulationskonfiguration von jeweils 1344 Monomeren und Liganden
[5] Bei einer Simulationskonfiguration mit einer jeweils gleichen Anzahl von Monomeren und Liganden

Verkürzung der Berechnungszeit um 91%. Hierbei wurden die Simulationsdurchläufe sequentiell ausgeführt.

Bei der Verwendung der gesamten GPU-Architektur unter gleichen Bedingungen benötigt die parallele Version 685 Sekunden für die Berechnung der Interaktionen von 21504 Partikeln. Bei einer Aufteilung der Menge der Partikel auf 16 Simulationsdurchläufe wird eine Berechnungszeit von 726 Sekunden gemessen. Dies entspricht einer Steigerung der Berechnungszeit um 6%.

Die Berechnungszeit der parallelen Version erhöht sich auf Grund des erhöhten Verwaltungsaufwandes der Datenstrukturen des gitterbasierten Ansatzes. Während im Kapitel 6.2 insgesamt zwei Gitter (je eines pro Partikeltyp) eingesetzt wurden, werden in dieser Untersuchung zwei Gitter pro Simulationsdurchlauf verwendet. Die Sortierung hat keinen Einfluss auf die beobachtete Sättigung des Speedups, da deren Berechnungszeit linear wächst. Solange die Summe der eingesetzten Partikel äquivalent ist, ist es vernachlässigbar, ob ein Simulationsdurchlauf mit einer Vielzahl von Partikeln ausgeführt wird oder eine große Menge von Simulationsdurchläufen mit einer geringen Menge von Partikeln.

Als Resultat kann festgestellt werden, dass beide Implementierungen bei der Erhöhung der Simulationsdurchläufe ein nahezu lineares Wachstum in ihrer Berechnungszeit besitzen. Daher resultiert ein gesättigter, nahezu konstanter Speedup, der als Faktor zwischen den betrachteten Berechnungszeitverläufen betrachtet werden kann.

Verschiedene Konfigurationen von Monomeren, Dimeren, Trimeren und Liganden, deren Summe jeweils einer äquivalenten Anzahl von Partikeln entspricht, besitzen nahezu identische Berechnungszeitverläufe. Dies begründet sich in der hohen Übereinstimmung der Berechnungsvorschriften in den verschiedenen Fällen.

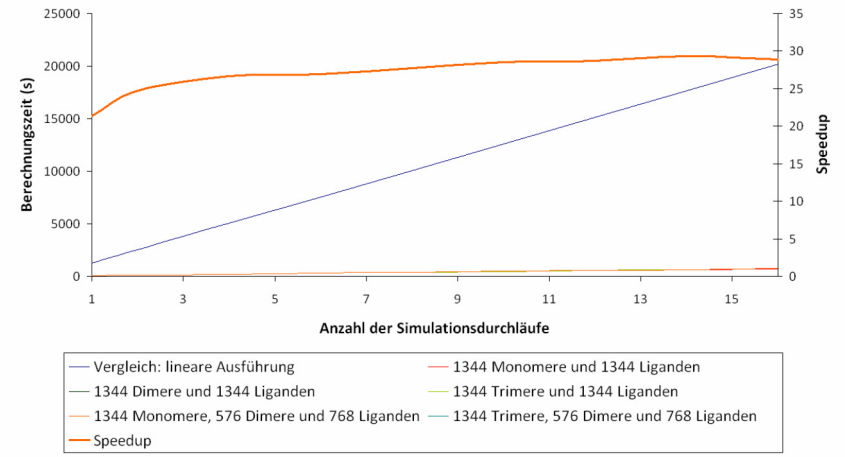


Abbildung 40: Berechnungszeit mehrerer Simulationsdurchläufe nach 100.000 Simulationsschritten

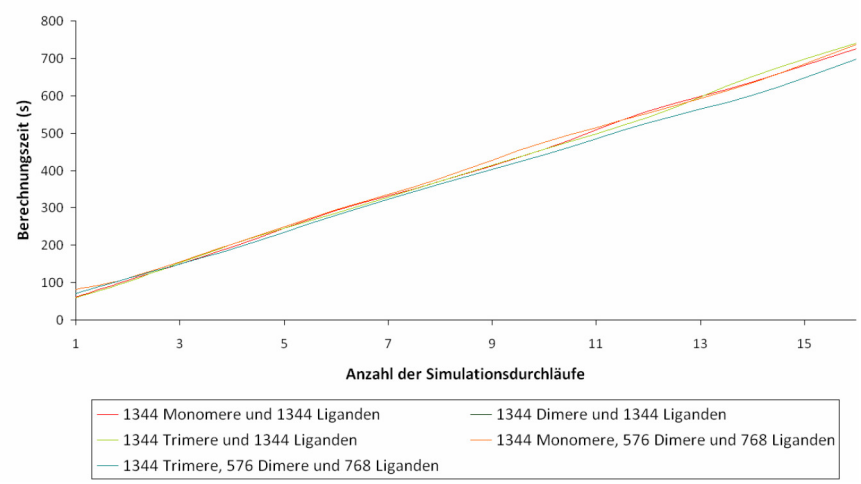


Abbildung 41: Vergleich der Berechnungszeit mehrerer Simulationsdurchläufe unterschiedlicher Konfigurationsart nach 100.000 Simulationsschritten (Ergänzung zu Abbildung 40)

Weitere Analysen wurden für ein Rezeptoren-Liganden-Verhältnis von 5:1 durchgeführt, welches in der späteren Anwendung der Simulation gebräuchlich ist.

Der Einsatz eines Verhältnisses von 5:1 resultiert in einer deutlich verringerten Anzahl von Interaktionen. Die Interaktionen werden auf der GPU massiv parallel durchgeführt. Dabei ist es für die Berechnungszeit nahezu unerheblich, wie viele Interaktionen für eine Berechnung vorhanden sind. Die Parallelität der Ausführung dieser Interaktionen ist die Grundlage für diese nahezu konstante Berechnungszeit verschiedener Anzahlen von Partikeln. Auf der CPU hingegen bewirkt die Reduzierung von Partikeln eines Typs eine deutliche Verkürzung der Berechnungszeit, da die Interaktionen sequentiell zur Berechnung herangezogen werden. Jede Reduzierung der Partikelanzahl hat daher einen direkten Einfluss auf die Berechnungszeit.

Aufgrund der verringerten Ausnutzung der Parallelität resultieren geringere Speedups in der Größenordnung von 26x-27x.

Darüber hinaus kann eine Sättigung des Speedups beobachtet werden, wie er auch in der vorherigen Untersuchung vorkam. In dieser Untersuchung tritt diese Sättigung erst bei zehn parallel ausgeführten Simulationsdurchläufen auf. Bei einer kleineren Anzahl von Simulationsdurchläufen wirkt sich die reduzierte Ausnutzung der Parallelität stärker aus.

Verschiedene Konfigurationen von Monomeren, Dimeren, Trimeren und Liganden, deren Summe jeweils einer äquivalenten Anzahl von Partikeln entspricht, besitzen auch in dieser Untersuchung nahezu identische Berechnungszeitverläufe.

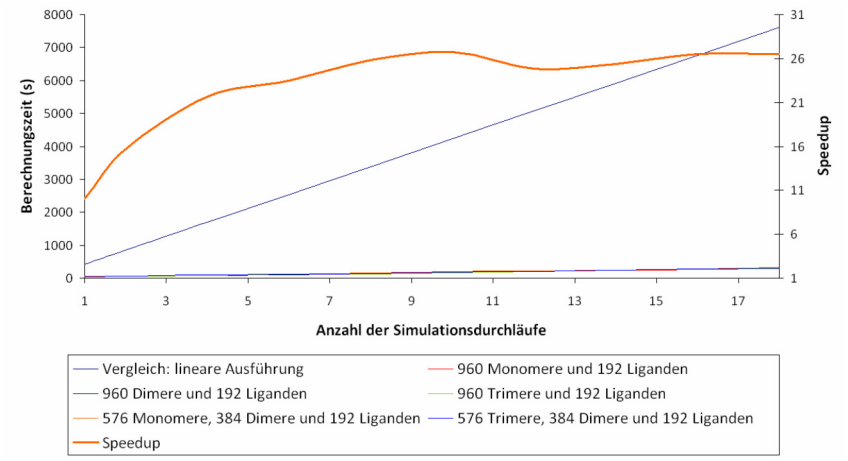


Abbildung 42: Berechnungszeit mehrerer Simulationsdurchläufe nach 100.000 Simulationsschritten (Analyse einer gebräuchlichen Simulationskonfiguration)

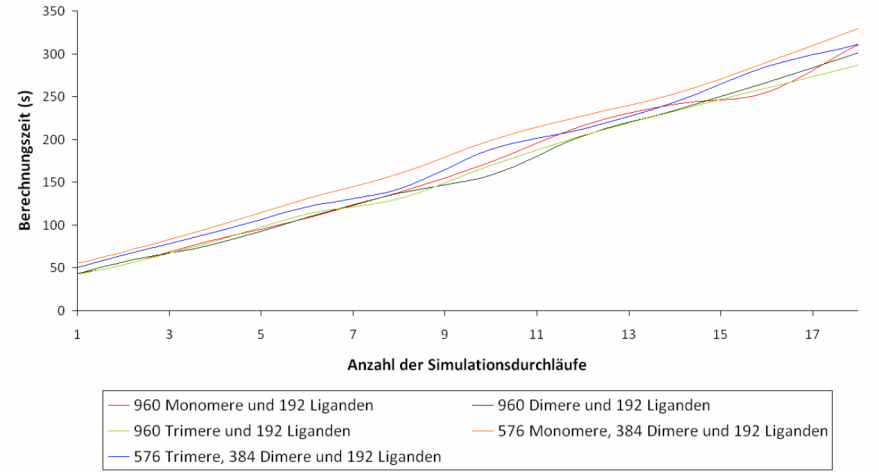


Abbildung 43: Vergleich der Berechnungszeit mehrerer Simulationsdurchläufe unterschiedlicher Konfigurationsart nach 100.000 Simulationsschritten (Ergänzung zu Abbildung 42)

6.4 Untersuchungen zur Multi-Kernel-Fähigkeit der Fermi-Architektur

Die Fermi-Architektur besitzt die Fähigkeit zur Ausführung mehrerer Kernel in einer parallelen Strategie. Die Fähigkeit wurde im Zusammenhang einer Abbildung von je einem Simulationsdurchlauf auf eine Ausführung eines Kernels untersucht. Resultat dieser Analyse ist der deutliche Overhead in der Berechnungszeit, aus dem hervorgeht, dass eine Nutzung der Erweiterungen der Fermi-Architektur keine Vorteile bietet. Die Verwaltung der Simulationsdurchläufe durch die Verwendung eines einzigen Kernels resultiert in einer geringeren Berechnungszeit und daher in einem größeren Speedup.

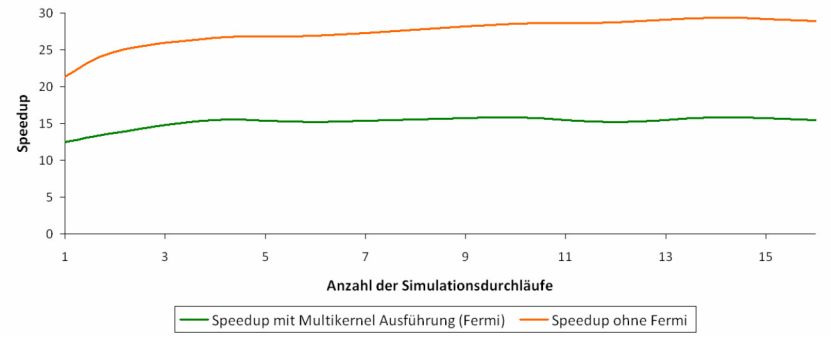


Abbildung 44: Vergleich der Speedups von Fermi's Fähigkeit zur Ausführung von mehreren Kerneln und der gewöhnlichen Ausführung von mehreren Simulationsdurchläufen

6.5 Untersuchungen zum Einsatz von mehreren GPUs

Neben der Nutzung der gesamten vorhandenen Architektur durch mehrere Simulationsdurchläufe sollte auch die Möglichkeit geschaffen werden, mehrere GPUs, die in einem System vorhanden sind, ebenfalls nutzen zu können. Dazu wurde die Verteilung der Simulationsdurchläufe auf mehrere GPUs erweitert. Hierbei wurden Speedups im Bereich von 40x gemessen.

Die Erkenntnisse aus den Untersuchungen des Kapitels 6.3, in denen verschiedene Anzahlen von Simulationsdurchläufen auf einer einzigen GPU eingesetzt wurden, übertragen sich vollständig auf die Ausführung dieser Simulationsdurchläufe auf mehreren GPUs.

Bei der Verteilung der Simulationsdurchläufe auf mehrere GPUs entsteht ein Overhead durch den zusätzlichen Verwaltungsaufwand, der bei der Zuweisung von Aufgaben an GPUs entsteht. Unter der Verwendung der gesamten Architektur werden beispielsweise beim Einsatz einer einzigen GPU 726 Sekunden für die Berechnung der Interaktionen von 21504 Partikeln gemessen. Hierbei wurden die Partikel zu gleichen Teilen auf 16 Simulationsdurchläufe aufgeteilt.

Werden vier GPUs eingesetzt und jeweils vier Simulationsdurchläufe pro GPU ausgeführt, werden unter denselben Simulationsbedingungen 589 Sekunden gemessen. Dies entspricht einer Verkürzung der Berechnungszeit um 19 Prozent.

.

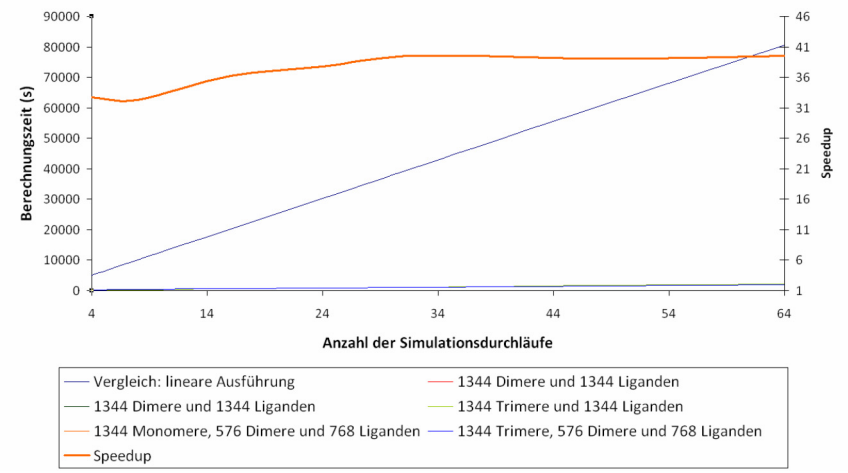


Abbildung 45: Berechnungszeit mehrerer Simulationsdurchläufe nach 100.000 Simulationsschritten auf 4 GPUs

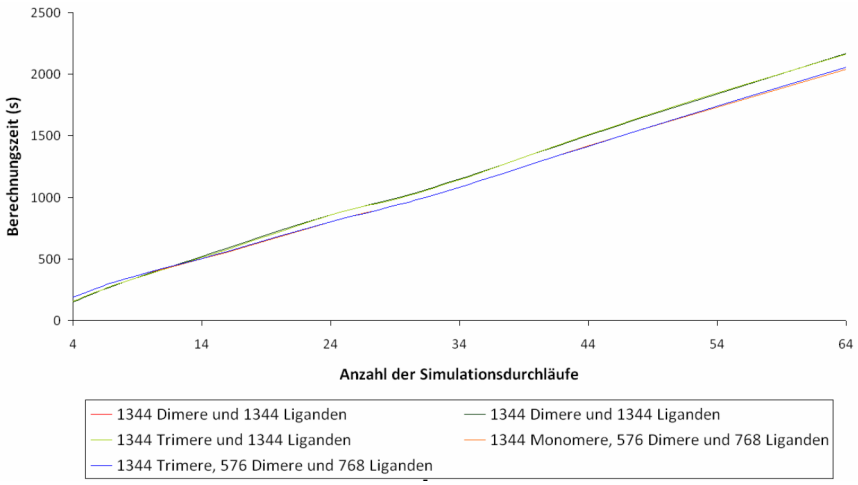


Abbildung 46: Berechnungszeit mehrerer Simulationsdurchläufe unterschiedlicher Konfigurationsart nach 100.000 Simulationsschritten auf 4 GPUs (Ergänzung zu Abbildung 45)

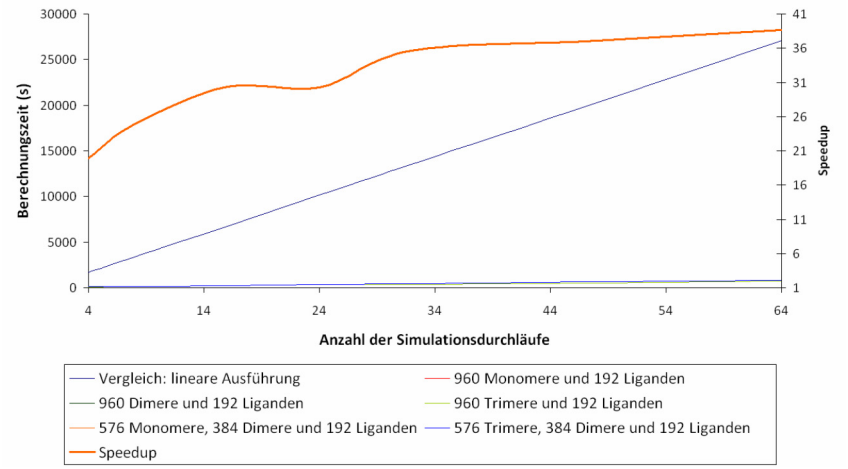


Abbildung 47: Berechnungszeit mehrerer Simulationsdurchläufe nach 100.000 Simulationsschritten auf 4 GPUs (Analyse einer gebräuchlichen Simulationskonfiguration)

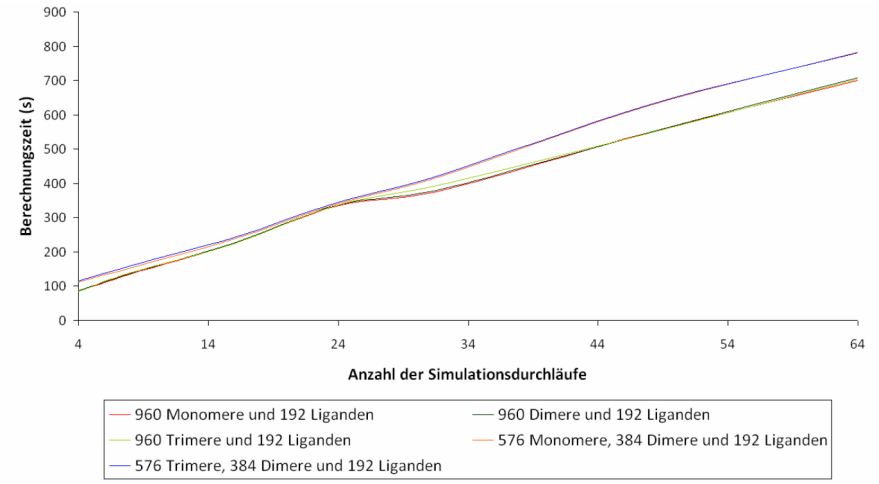
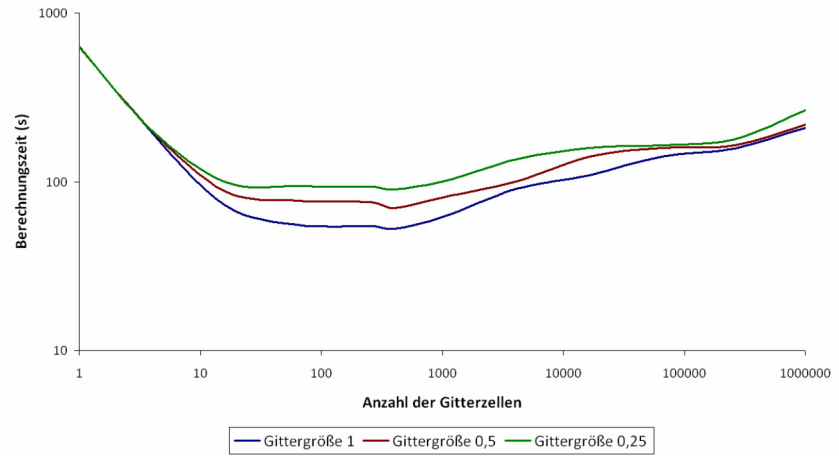


Abbildung 48: Vergleich der Berechnungszeit mehrerer Simulationsdurchläufe unterschiedlicher Konfigurationsart nach 100.000 Simulationsschritten auf 4 GPUs (Ergänzung zu Abbildung 47)

6.6 Beobachtungen im Zusammenhang des gitterbasierten Ansatzes

Während der Entwicklung des gitterbasierten Ansatzes zur Optimierung der Menge der potentiellen Interaktionspartner wurde ein Verhalten der Berechnungszeit beobachtet, welches direkte Folgen für die Wahl der Größe einer Gitterzelle hat. Die Wahl der optimalen Größe einer Gitterzelle muss durch ein iteratives Verfahren bestimmt werden, da diese nicht der Größe des Berechnungshorizontes eines Partikels entspricht. Eine Interpretation dieses Verhaltens wurde im Kapitel über die Implementierung des gitterbasierten Ansatzes vorgestellt.



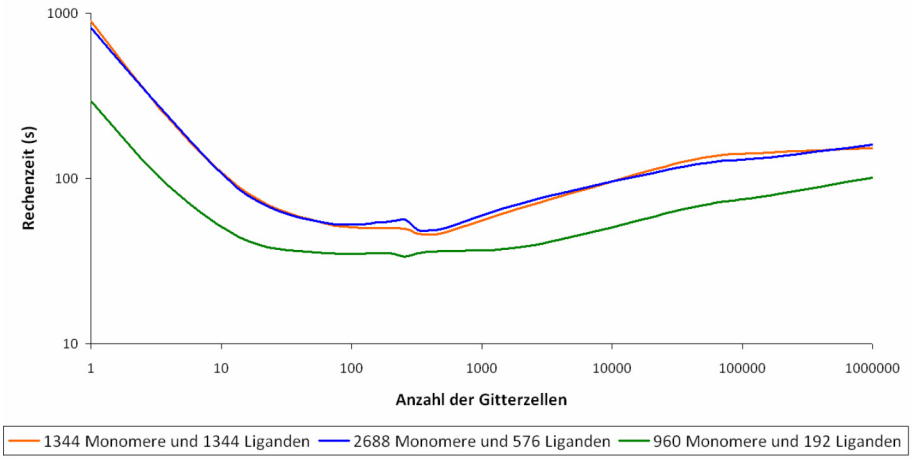


Abbildung 49 + Abbildung 50: Berechnungszeit verschiedener Anzahlen von Gitterzellen nach 100.000 Simulationsschritten bei verschiedenen Simulationsbedingungen

7. Zusammenfassung

Im Rahmen dieser Studienarbeit wurde ein Simulationsmodell zur Evaluierung von Apoptose-Signalwegen von einer CPU-Implementierung auf eine GPGPU-Architektur abgebildet, wobei die Applikationsschnittstelle CUDA eingesetzt wurde. Das vorrangige Ziel der Verkürzung der Berechnungszeit wurde dabei erfüllt. Zusätzlich wurden Maßnahmen im Bereich der Flexibilität der Simulationsumgebung für eine nachträgliche Anpassung an weitere Simulationsaufgaben durchgeführt. Es wurde die Möglichkeit geschaffen, eine Vielzahl von Simulationsdurchläufen auf einer beliebigen Anzahl von GPUs parallel ausführen zu können. Da die Simulation auf einem stochastischen Prozess basiert, wird hiermit der Anforderung begegnet, eine Mittelung über eine Vielzahl von Simulationsergebnissen durchführen zu können.

Die Implementierung des Simulationsmodells wurde im Blick auf Optimierungsmöglichkeiten mehreren Untersuchungen unterzogen. Neben der Optimierung in arithmetischen Teilen des Simulationsmodells wurden auch die Datenstrukturen hinsichtlich eines gitterbasierten Ansatzes optimiert, dessen Implementierung im Kontext von insgesamt drei Optimierungsuntersuchungen beleuchtet wurde, von denen zwei zu einer deutlichen Erhöhung des Speedups führten. Erwähnenswert sind hierbei die Koordinierung der Lesezugriffe auf Nachbarzellen, die den Scatteringanteil eines Lesezugriffes deutlich erhöhen und die Partitionierung einer Gitterzelle in 4 Quadranten, in denen die Anzahl der betrachteten (Nachbar-)gitterzellen von neun auf vier reduziert wurde.

Ein Resultat der Analyse ist eine Aussage in der Frage nach einer optimalen Gittergröße. Wird in der Literatur [26] intuitiv von einer Gittergröße ausgegangen, die dem Berechnungshorizont entspricht, zeigt die Analyse von verschiedenen Gittergrößen, dass in diesen Fällen unnötige Berechnungszeit aufgewandt wird. Eine optimale Berechnungszeit muss durch einen iterativen Prozess gefunden werden.

Der erreichte Speedup für einen einzigen Simulationsdurchlauf beträgt 320x unter der Ausnutzung der gesamten vorhandenen Architektur. Da es für das Simulationsmodell die stochastische Anforderung einer sehr großen Anzahl von Simulationsdurchläufen gibt, kann der für die Simulationsaufgabe relevante Speedup von 40x betrachtet werden. Hierbei wurden 64 Simulationsdurchläufe verteilt auf vier GPUs parallel ausgeführt.

Aufgrund des in dieser Arbeit erreichten Speedups wurde die Berechnungszeit der Simulation des Modells zur Evaluierung der Apoptosesignalwege in einen handhabbaren zeitlichen Bereich verkürzt. Dies ermöglicht in direkter Folge die Untersuchung der Apoptosesignalwege im Rahmen einer Simulation.

Literaturverzeichnis

[1] Hengst, Ludger, *Regulation der Zellproliferation*, Max-Planck-Institut für Biochemie, Tätigkeitsbericht 2003.[Online].http://www.biochem.mpg.de/institute/research_reports/pdf/2003_hengst.pdf

[2] S. Beneke, *Chemische Carzinogenese und ihre Mechanismen,* LS Bürkle*,* Molecular Toxicology, University of Konstanz, October 28, 2005. [Online]. http://gutenberg.biologie.uni-konstanz.de/biomed0506/2005-10-28_Chemische%20Karzinogenese.pdf

[3] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter, *The Molecular Biology of the Cell*, 4. Ausgabe, Kapitel 23.

[4] J. Thakar, *Computational models for the study of responses to infections*, Universität Würzburg, 2005. [Online]. http://opus.bibliothek.uni-wuerzburg.de/volltexte/2006/1726/pdf/thesis4-2.pdf

[5] M.Falk, M. Daub, G.Schneider, T. Ertl, *Modeling and Visualization of Receptor Clustering on the Cellular Membrane,* IEEE Symposium on Biological Data Visualization (BioVis), 2011. [Online]. http://www.simtech.uni-stuttgart.de/publikationen/prints.php?ID=328

[6] C. Guo and H. Levine, *A thermodynamic model for receptor clustering*, Biophysical Journal, 77(5):2358–2365, 1999.

[7] M. Gopalakrishnan, K. Forsten-Williams, M. Nugent, and U. C. Täuber, *Effects of receptor clustering on ligand dissociation kinetics: Theory and simulations*, Biophysical Journal*,* 89(6):3686-700, 2005.

[8] G. D. Guerrero, J. M. Cecilia, J. M. Garcia, M. A. Martinez, I. Perez, M. J. Jimenez, *Analysis of P systems simulation on CUDA.* [Online]. http://www.cs.us.es/~marper/investigacion/Jornadas-paralelismo-2009.pdf

[9] Y. Zhou, J. Liepe, X. Sheng, M. P. H. Stumpf, C. Barnes, *GPU accelerated biochemical network simulation*, Bioinformatics Applications Note, Vol. 27 no. 6 2011, pages 874–876. [Online]. http://bioinformatics.oxfordjournals.org/content/27/6/874.full.pdf+html

[10] H. Suhartanto, A. Yanuar and A. Wibisono, *Performance Analysis Cluster and GPU Computing Environment on Molecular Dynamic Simulation of BRV-1 and REM2 with GROMACS*, IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 4, No 2, July 2011.[Online] http://ijcsi.org/papers/IJCSI-8-4-2-131-135.pdf

[11] A. Gewies, *Introduction to Apoptosis*, ApoReview, 2003 [Online]: http://www.celldeath.de/encyclo/aporev/apointro.pdf

[12] B. Barney, *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory. [Online]. https://computing.llnl.gov/tutorials/parallel_comp/

[13] A. Zibula, *General Purpose Computation on Graphics Processing Units (GPGPU) using CUDA*, Seminar Paper, Dezember 2012. [Online]. http://www.wi.uni-muenster.de/pi/lehre/ws0910/pppa/papers/gpgpu.pdf

[14] Nvidia Corporation, *Nvidia Cuda C Programming Guide 4.0*. [Online]. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

[15] Nvidia Corporation, *Nvidia's Next Generation CUDA Compute Architecture: Fermi*, Whitepaper. [Online]. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[16] A. Martens, *AMD Stream Computing Environment (SCE)*. [Online]. http://www1.informatik.uni-mainz.de/lehre/cg/SS2008_PA/data/V09c_AMD_StreamComputingEnvironment.pdf

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

[18] Intel Corporation, *Intel® Core™ i7-970 Processor, Specifications*. [Online]. http://ark.intel.com/products/47933/Intel-Core-i7-970-Processor-%2812M-Cache-3_20-GHz-4_80-GTs-Intel-QPI%29

[19] IBM, *Cell Broadband Engine Architecture*, [Online]. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA277638725706000 6E61BA/$file/CBEA_v1.02_11Oct2007_pub.pdf

[20] Intel Corporation, *Teraflops Research Chip*. [Online]. http://techresearch.intel.com/projectdetails.aspx?id=151

[21] Intel Corporation, *Hintergrund Strukturgrößen 32nm*, März 2011

[22] R. Rauber, *Parallele Programmierung*, 2. Auflage, S.102

[23] Nvidia Corporation, *Nvidia Cuda C Programming Guide 4.0 – Anhang F*. [Online]. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit /docs/CUDA_C_Programming_Guide.pdf

[24] A. J. Bhatt, *Creating a PCI Express Interconnect*, Technology and Research Labs, Intel Corporation. [Online]. http://www.pcisig.com/specifications/pciexpress/ resources/PCI_Express_White_Paper.pdf

[25] Nvidia Corporation, *TESLA C2050 / C2070*. [Online].http://www.nvidia.de/docs/ IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf

[26] S. Green, *Particle Simulation using CUDA*, Nvidia Corporation. [Online]. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/particles/doc/particles.pdf

[27] N. Satish, M. Harris, M. Garland, *Designing Efficient Sorting Algorithms for Manycore GPUs*, University of California, Berkeley, NVIDIA Corporation. [Online]. http://mgarland.org/files/papers/gpusort-ipdps09.pdf

[28] *GROMACS* [Online]. http://www.gromacs.org

[29] Nvidia Corporation, *CUDA CURAND LIBRARY*. [Online]. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CURAND_Library.pdf

[30] R. Rauber, *Parallele Programmierung*, 2. Auflage, Kapitel 4.2, S. 167

[31] Gheorghe Paun, *Introduction to Membrane Computing*, Institute of Mathematics of the Romanian Academy, 2004. [Online]. http://psystems.disco.unimib.it/ download/MembIntro2004.pdf

[32] J. E. Jones, *On the Determination of Molecular Fields, I*, *From the Variation of the Viscosity of a Gas with Temperature*. Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, 106(738):441–462, 1924.

[33] P.E. Kloeden, E. Platen, *Numerical Solution of Stochastic Differential Equations*, Springer, 1999, S.305

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen