Institut für Technische Informatik
Universität Stuttgart
Pfaffenwaldring 47
D–70569 Stuttgart

Studienarbeit Nr. 2384

Framework für beschleunigte Monte-Carlo-Molekularsimulationen auf hybriden Architekturen

Sebastian Halder

Studiengang: Informatik

Prüfer: Prof. Dr. Hans-Joachim Wunderlich

Betreuer: Dipl.-Inform. Claus Braun, Dipl.-Inf. Stefan Holst

begonnen am: 01. Juni 2012

beendet am: 01. Dezember 2012

CR-Klassifikation: C.1.3, J.2

Kurzfassung

In der Thermodynamik können Monte-Carlo-Molekularsimulationen eingesetzt werden, um makroskopische Eigenschaften eines Molekularsystems zu beobachten. Diese Simulationen sind äußerst rechenintensiv.

Aktuelle und kommende Generation von eng gekoppelten Mehrkernprozessoren und Grafikprozessoren (GPGPUs) bieten ein großes Potential an Rechenleistung, welches sie für solche Simulationsanwendungen besonders interessant macht.

Die dieser Arbeit zu Grunde liegende Markov-Chain-Monte-Carlo-Molekularsimulation (MCMC/GCMC) basiert jedoch auf der Erzeugung einer Markovkette, d.h. jeder Simulationsschritt hängt vom Vorhergehenden ab. Diese inhärente serielle Abhängigkeit erschwert die Parallelisierung des Problems erheblich.

In der vorliegenden Arbeit wurden Konzepte und Implementierungen für ein Framework entwickelt, welches eine effiziente Simulation von Monte-Carlo-Simulationen mit Markovketteneigenschaften auf hybriden Architekturen ermöglicht. Diese Konzepte umfassen eine Simulations-Zustandsmaschine mit Unterstützung verschiedener Architekturen und eine Schnittstelle für mehrere simultan zu simulierende Monte-Carlo-Schritte. Darüber hinaus wurde die zu Grunde liegende Parallelisierung einer Grand-Canonical Monte-Carlo-Simulation auf hybriden Architekturen weiterentwickelt und beschleunigt. Die entstandene Implementierung wurde auf die erzielbare Leistung überprüft. Alle im Rahmen dieser Arbeit entstandenen Simulationsergebnisse wurden durch Vergleich mit einer Referenzimplementierung auf ihre Korrektheit überprüft.

Im Vergleich zu einer rein seriellen Simulation wurde dabei ein Speedup durch den Einsatz von hybriden Architekturen von 494x erreicht.

Inhaltsverzeichnis

1 Einleitung 11

2 Thermodynamik 15
2.1 Molekularsysteme 15
2.2 Monte-Carlo-Simulationen 16
2.3 Grand-Canonical Monte-Carlo-Simulation 16

3 Rechnerarchitekturen 19
3.1 Latenzoptimierte Prozessoren 19
3.2 Durchsatzoptimierte Prozessoren 20
3.3 Ausblick 21

4 Hybrid-Simulation-State-Machine 23
4.1 Konzept 23
4.1.1 Definition von Simulationskerneln 23
4.1.2 Replicated-Data-Schema 25
4.1.3 Architekturübergreifende Kommunikation 26
4.2 Hybrid-Simulation-Manager 26
4.2.1 Verwaltung von Simulationskerneln 26
4.2.2 Automatisierte Wahl von Simulationskerneln 27
4.2.3 Verifikation von Ergebnissen und Fehlerbehandlung 32
4.3 Die Hybrid-Simulation-State-Machine als endlicher Zustandsautomat 33
4.3.1 Initialisierungsphase 33
4.3.2 Simulationsphase 34
4.3.3 Fehlerzustand 34

5 Change-Linked-Simulation 35
5.1 Anforderungen an die Schnittstelle 35
5.2 Konzept 36
5.3 Ablauf der Change-Linked-Simulation in der Hybrid-Simulation-State-Maschine 37
5.4 Einhaltung der Markovketteneigenschaft 38

6 Beschleunigte Berechnung des elektrostatischen Potentials 41
6.1 Berechnung des Coulomb-Potentials 41
6.2 Beschleunigte Berechnung des Fourierraumanteils für mehrere simultane *Monte-Carlo*-Schritte 42
6.3 Betrachtung der Zeitkomplexität 44

7 Simulationskernel 45
7.1 SeriellNormal . . . 45
7.2 SeriellBeschleunigt . . . 47
7.3 CudaNormal . . . 49
7.4 CudaBeschleunigt . . . 51

8 Experimentelle Ergebnisse 53
8.1 Referenzsystem . . . 53
8.2 Hardware-Plattform . . . 53
8.3 Ergebnisse . . . 54
8.3.1 Variation der Anzahl von Molekülen . . . 54
8.3.2 Variation der Anzahl von Mutanten . . . 61
8.3.3 Automatisierte Wahl von Simulationskerneln . . . 65
8.4 Diskussion . . . 73
8.4.1 Vergleich der seriellen Simulationskernel mit den Simulationskerneln auf hybriden Architekturen . . . 73
8.4.2 Vergleich der Simulationskernel CudaNormal und CudaBeschleunigt 74
8.4.3 Automatisierte Wahl von Simulationskerneln . . . 75

9 Zusammenfassung 77

Literaturverzeichnis 79

Abbildungsverzeichnis

2.1 Das *Lennard-Jones-Potential* und das *Coulomb-Potential*. 15

4.1 Zwei Simulationskernel zur Summation von vier Zahlen. Simulationskernel A basiert auf der Summation von partiellen Summen, Simulationskernel B auf einer rein seriellen Summation. Ein Simulationskernel beschreibt in der *Hybrid-Simulation-State-Machine* eine für eine bestimmte Architektur optimierte Methode zur Auswertung von *Monte-Carlo*-Schritten. 24
4.2 Zwei verschiedene Simulationskernel A und B. Simulationskernel A nutzt die Datenstruktur eines Arrays. Simulationskernel B nutzt die Datenstruktur einer verketteten Liste. 25
4.3 Überblick über eine *Hybrid-Simulation-State-Machine* mit zwei verschiedenen Devicearchitekturen und zugehörigen Simulationskerneln. Kommunikationswege sind mit durchgehenden Pfeilen dargestellt, Zugriffsrechte auf Speicherstrukturen mit gestrichelten Pfeilen. 27
4.4 Zustandsdiagramm der *Hybrid-Simulation-State-Machine* 33

5.1 Zwei *Änderungsinstanzen* mit Darstellung von Schreibrechten (durchgehende Pfeile) und Leserechten (gestrichelte Pfeile) 36
5.2 Ablaufdiagramm für einen Simulationsschritt der *Change-Linked Simulation*. . . 37
5.3 Oben: Zwei aus einem gemeinsamen Zustand t_s generierte Änderungen a und b und daraus resultierende Zustände t_a und t_b. Unten: Verletzung der Markov Kette bei Akzeptierung zweier aus dem gleichen Zustand t_s generierten Änderungen a und b. 38

6.1 Darstellung des *Monte-Carlo*-Schritts j durch das Entfernen von $A_{o,j}$ aus der alten Konfiguration N_o und dem Hinzufügen von $A_{n,j}$. 43

8.1 Durchschnittliche Laufzeit der Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) bei Simulation eines Mutanten. 54
8.2 Durchschnittliche Laufzeit der Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) für zwei Mutanten. 56
8.3 Durchschnittliche Laufzeit zur Evaluation eines *Monte-Carlo*-Schritts für die Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) für vier Mutanten. 58

8.4 Durchschnittliche Laufzeit zur Evaluation eines *Monte-Carlo*-Schritts für die Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) für acht Mutanten. 60
8.5 Laufzeiten von CudaNormal und CudaBeschleunigt für 128, 256 und 512 Moleküle über die Anzahl der Mutanten 64
8.6 Laufzeiten von CudaNormal und CudaBeschleunigt für 4096 und 8192 Moleküle über die Anzahl der Mutanten 64
8.7 Maximale Laufzeit der Simulationsinstanzen für unterschiedliche Anzahl nebenläufiger Simulationsinstanzen bei 64 Molekülen. S bF max beschreibt die maximale Laufzeit unter Einsatz von SeriellBeschleunigt, C n max die maximale Laufzeit unter Einsatz von CudaNormal. AS 1 und AS 2 bezeichnen die beiden Messreihen mit automatischem Scheduling von Simulationskerneln. 69
8.8 Maximale Laufzeit der Simulationsinstanzen für unterschiedliche Anzahl nebenläufiger Simulationsinstanzen. S bF max beschreibt die maximale Laufzeit unter Einsatz von SeriellBeschleunigt, C n max die maximale Laufzeit unter Einsatz von CudaNormal. *AS 1* und AS 2 bezeichnen die beiden Messreihen mit automatischem Scheduling von Simulationskerneln. 72

Tabellenverzeichnis

8.1 Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle bei Simulation eines Mutanten. 55
8.2 Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle bei der Simulation von zwei Mutanten. 57
8.3 Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle und vier Mutanten. 59
8.4 Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle und acht Mutanten. 61
8.5 *Cn*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit fest gewähltem Simulationskernel CudaNormal bei 64 Molekülen. 66
8.6 *SbF*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit festem Simulationskernel SeriellBeschleunigt für 64 Moleküle. 67
8.7 *AS 1*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal bei 64 Molekülen. 67

8.8 *AS 2*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal bei 64 Molekülen. . . . 68
8.9 Maximale Laufzeiten 69
8.10 *Cn*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit festem Simulationskernel CudaNormal bei 32 Molekülen. . . . 70
8.11 *SbF*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit festem Simulationskernel SeriellBeschleunigt bei 32 Molekülen. . . . 71
8.12 *AS 1*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal bei 32 Molekülen. . . . 71
8.13 *AS 2*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal. . . . 72
8.14 Maximale Laufzeiten 73

Verzeichnis der Algorithmen

4.1 AutomaticSchedulingSimulation, Initialisierung der Simulation 29
4.2 Automatisches Scheduling für zwei Simulationskernel 30
4.3 Simulation für GPGPUSim mit randomisiertem Kernelwechsel bei Speedup von GPGPUSim gegenüber CPUSim 31
4.4 Simulation für CPUSim mit randomisiertem Kernelwechsel bei Speedup von CPUSim gegenüber GPGPUSim 32

7.1 Serieller Algorithmus zur Berechnung der Energiedifferenz mehrerer Mutanten 46
7.2 Berechnung der paarweisen Energie 46
7.3 Algorithmus zur unbeschleunigten Berechnung des Fourierraumanteils 47
7.4 Algorithmus zur beschleunigten Berechnung des Fourierraumanteils 48
7.5 Kernel K_Pair zur paarweisen Energieberechnung nach [1] 49
7.6 Kernel K_Fourier zur Berechnung der Koeffizienten für den Fourierraumanteil modifiziert nach [1] 50
7.7 Kernel K_Fourier_Sum zur Summation des Fourierraumanteil nach [1] 51
7.8 Kernel K_Fourier_NoChange zur Berechnung der Koeffizienten $C_{k,konst}$ des Fourierraumanteils für alle Partikel des Systems ohne Berücksichtigung eines bestimmten Mutanten 51
7.9 Kernel K_Fourier_Changeonly zur Berechnung der Koeffizienten $\hat{c}_{k,cidx}$ des Fourierraumanteils für die von *Monte-Carlo*-Schritten betroffenen Partikel . . . 52
7.10 Kernel K_Fourier_Sum_New 52

1 Einleitung

Molekularsimulationen werden in vielen Anwendungsgebieten eingesetzt, um die Eigenschaften und das Verhalten von Molekülstrukturen untersuchen zu können. Ein wichtiger Vertreter dieser Molekularsimulationen ist die Markovkettenmolekularsimulation (engl.: *Markov-Chain Monte-Carlo* (MCMC)-Simulation). Hierbei handelt es sich um eine Unterklasse der auf [2] zurückgehenden *Monte-Carlo*-Simulation.

In der Thermodynamik kann mit dieser Methode die auf quantenmechanischen Grundlagen beruhende Verteilung von Mikrozuständen eines im Gleichgewicht befindlichen Systems untersucht werden. Ist diese Verteilung bekannt, können Rückschlüsse auf makroskopische Eigenschaften des Systems gezogen werden. Der Zustandsraum wird dabei durch *Monte-Carlo*-Schritte exploriert, ein Schritt besteht aus der randomisierten Änderung eines einzelnen im System befindlichen Moleküls. Diese Änderung des Systems wird dann mit einer bestimmten Akzeptanzwahrscheinlichkeit angenommen. Diese Wahrscheinlichkeiten sind abhängig von verschiedenen, den Zustand des Systems betreffende Kriterien. Ein Beispiel für solch ein Kriterium ist die durch diesen *Monte-Carlo*-Schritt induzierte Energiedifferenz. Für die Ermittlung dieser Eigenschaften und der damit verbundenen Exploration des Zustandsraums ergeben sich zwei gegensätzliche Möglichkeiten bezüglich der gewählten Schrittweite: Kleine Änderungen haben eine große Akzeptanzwahrscheinlichkeit, führen allerdings zu einer langsamen Exploration des gesamten Zustandsraums. Im Gegensatz dazu bewirken große Änderungen eine schnelle Exploration des Zustandsraums, die Akzeptanzwahrscheinlichkeit für einzelne Schritte ist jedoch entsprechend klein. Eine fundamentale Eigenschaft dieser MCMC-Simulation ist die Abhängigkeit eines Schrittes von allen vorhergehenden Schritten, entsprechend schwierig gestaltet sich daher die Parallelisierung des Problems.

Der Ausgangspunkt dieser Studienarbeit ist eine Parallelisierung der *Grand-Canonical Monte-Carlo* (GCMC)-Simulation auf hybriden Rechnerarchitekturen [1], bei der zwei primäre Faktoren bei der Parallelisierung zum Einsatz kommen:

- Simultane Auswertung mehrerer *Monte-Carlo*-Schritte mit großer Schrittweite und kleiner Akzeptanzwahrscheinlichkeit zur schnellen und effektiven Exploration des Zustandsraums
- Parallelisierung von algorithmischen Schritten bei der Evaluation von *Monte-Carlo*-Schritten auf durchsatzoptimierten Architekturen (GPGPUs) und latenzoptimierten Architekturen (CPUs)

Die Untersuchungen in [1] haben gezeigt, dass im Vergleich zu einer rein seriellen Simulation von MCMC-Systemen ein Geschwindigkeitszuwachs von bis zu **87x** bei der Simulation auf hybriden Architekturen erreicht werden kann.

Das Ziel dieser Studienarbeit bestand in der Konzeption einer hybriden Simulationsumgebung, die die Vorteile der latenz- und durchsatzoptimierten Architekturen in einem gemeinsamen Simulationsframework vereinigt, sowie dem Anwender eine transparente und sichere Möglichkeit zur Simulation von Molekularsystemen ermöglicht. Dazu wurden die Konzepte der *Hybrid-Simulation-State-Machine* und der *Change-Linked-Simulation* entworfen. Diese beiden Konzepte wurden dabei so allgemein formuliert, dass sie sich direkt auf andere Anwendungen übertragen lassen.

Besonderer Wert wurde dabei auf eine klare Schnittstelle und die Konsistenz von Daten auf unterschiedlichen Architekturen gelegt, damit Fehler in der Verwendung dieses Simulationsframeworks minimiert werden. Zur optionalen Verifikation von Berechnungsmethoden wurde das Framework so gestaltet, dass jegliche Berechnungen durch eine zweite Referenzrechnung abgeglichen werden können.

Bei der Konzeption der *Hybrid-Simulation-State-Machine* wurden mehrere Faktoren für eine schnelle, sichere und für andere Architekturen erweiterbare Simulationsumgebung berücksichtigt:

- Minimierung der Kommunikation von Daten zwischen den Architekturen bei gleichbleibender Konsistenz der Daten durch ein *Replicated-Data*-Schema
- Klar definierte Schnittstellen gegenüber dem Anwender bezüglich des Zustands der Simulation

Die *Change-Linked-Simulation* wurde als Kommunikationsschema zwischen Anwender und Simulationsframework mit den folgenden Zielen entwickelt:

- Minimierung der Kommunikation zwischen Anwender und Simulationsframework
- Möglichkeit zur Generierung und Simulation mehrerer simultan auszuwertender *Monte-Carlo*-Schritte
- Zuordnung von Simulationsergebnissen zu den vom Anwender generierten *Monte-Carlo*-Schritten und einfache Möglichkeit zur Akzeptierung eines *Monte-Carlo*-Schrittes

Des Weiteren wurde ein Teil der von [1] vorgestellten Methoden zur Parallelisierung der Energieberechnung weiter entwickelt und beschleunigt. Darüber hinaus sollte in dieser Studienarbeit das Laufzeitverhalten einer GCMC-Simulation auf hybriden Architekturen untersucht werden um festzustellen, ob und in welchem Maße diese Simulation von solchen hybriden Architekturen profitiert.

In einer möglichen Anwendung dieser GCMC-Simulationen in der Thermodynamik wird das Verhalten von Systemen bezüglich unterschiedlicher Anzahlen von Molekülen untersucht. Dazu werden mehrere GCMC-Simulationen mit unterschiedlicher minimaler und maximaler Molekülanzahl simuliert und die aus diesen Simulationen resultierenden Ergebnisse zusammengefasst, um Aussagen über den gesamten untersuchten Bereich treffen zu

können. Bei einer Simulation auf hybriden Architekturen ist daher die Frage interessant, ob es einen Schwellenwert mit einer bestimmten Anzahl von Molekülen gibt, ab dem ein Wechsel der zur Berechnung eingesetzten Architektur sinnvoll ist.

Ein weiterer zu untersuchender Punkt ist das dynamische Verhalten von mehreren nebenläufigen Simulationen und eine dadurch höhere Auslastung der Architekturen. Hieraus wurde in dieser Studienarbeit eine Methode entwickelt, die für jede Simulationsinstanz unabhängig von einer globalen Synchronisationseinheit entscheidet, auf welcher Architektur die Simulation durchgeführt werden soll, um eine optimierte Laufzeit zur erreichen. Durch diese lokalen Entscheidungsmethoden wird erreicht, dass auch bei der Auslastung von Architekturen durch andere auf dem System ausgeführten Anwendungen entsprechend auf diese Auslastung reagiert werden kann.

2 Thermodynamik

2.1 Molekularsysteme

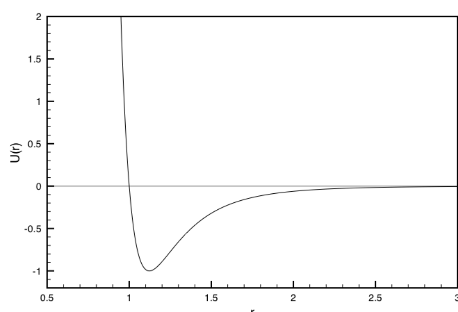
In der Molekularsimulation werden die zu Molekülen gehörenden Atome durch *Partikel*, welche die Eigenschaften von Atomen für diese Anwendungsbereich hinreichend genau wiedergeben, beschrieben. Diese Abstraktion beinhaltet unter anderem die Position im Raum, die Masse, sowie die Ladung des Atoms. Je nach Einsatzgebiet können die zwischen zwei Partikeln i und j wirkenden Kräfte über verschiedene Potentiale beschrieben werden. Hierbei wird zumeist zwischen kurzreichweitigen und langreichweitigen Potentialen unterschieden. Ein oft verwendetes Beispiel für kurzreichweitige Potentiale ist das *Lennard-Jones-Potential*. Dieses wird durch eine Funktion, abhängig von der Distanz r_{ij} zweier Partikel i und j beschrieben, wobei σ und ϵ von den Atomen i und j abhängige Konstanten darstellen.

$$U_{LJ}(r_{ij}) = 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) \tag{2.1}$$

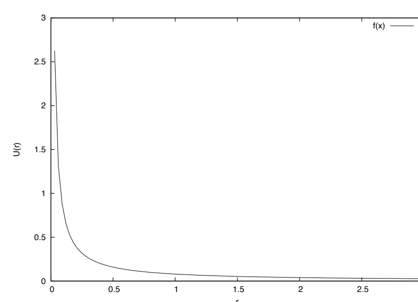
Ein entsprechendes Beispiel für ein langreichweitiges Potential ist das *Coulomb-Potential* für die elektrostatische Interaktion zwischen den Partikeln i und j:

$$U_{Coul}(r_{ij}) = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}} \tag{2.2}$$

Dieses Potential kann durch die Ewald-Summation [3] in einen Realraumanteil und einen Fourierraumanteil getrennt werden. Die Abbildung 2.1a zeigt das *Lennard-Jones-Potential* für $\sigma = 1$ und $\epsilon = 1$, in Abbildung 2.1b ist das *Coulomb-Potential* dargestellt.



(a) *Lennard-Jones-Potential*



(b) *Coulomb-Potential*

Abbildung 2.1: Das *Lennard-Jones-Potential* und das *Coulomb-Potential*.

Für die vollständige Beschreibung eines Molekularsystems fehlen noch die Definition des Simulationsgebiets, sowie die an den Rändern des Simulationsgebiets geltenden Randbedingungen. Je nach Einsatzgebiet der Molekularsimulation müssen unterschiedliche Randbedingungen betrachtet werden. Hierbei sind speziell die periodische Randbedingungen zu nennen: Es wird angenommen, dass sich das Simulationsgebiet mit der aktuellen Konfiguration in alle Raumrichtungen periodisch fortsetzt. Durch diese Vorgehensweise können Rückschlüsse von einem relativ kleinen Simulationsfenster auf größere Systeme getroffen werden.

2.2 Monte-Carlo-Simulationen

Molekularsimulationen können in der Thermodynamik verwendet werden, um makroskopische Eigenschaften eines Systems im Gleichgewichtszustand zu extrahieren. Eine der wichtigsten Methoden hierfür ist die *Monte-Carlo*-Simulation, welche auf [2] zurückgeht. Die *Monte-Carlo*-Simulation bedient sich dabei den Regeln der statistischen Mechanik. Das fundamentale Postulat der statistischen Mechanik besagt, dass ein sich im Gleichgewichtszustand befindliches isoliertes System mit gleicher Wahrscheinlichkeit in allen seinen Mikrozuständen befindet. Bei der *Monte-Carlo*-Simulation wird diese Verteilung von Mikrozuständen, die mit einem entsprechenden Gleichgewichtszustand kompatibel sind, im Phasenraum untersucht. Dazu wird aus einem gegebenen Anfangszustand mittels *Importance Sampling* ein neuer Zustand generiert [4]. Diese Modifikation des Ausgangszustands wird als *Monte-Carlo*-Schritt bezeichnet. Nach entsprechenden Akzeptanzwahrscheinlichkeiten wird dieser *Monte-Carlo*-Schritt entweder akzeptiert oder verworfen. Wird diese durch den *Monte-Carlo*-Schritt beschriebene Modifikation akzeptiert, dann wird dieser neue Zustand als Ausgangspunkt für weitere *Monte-Carlo*-Schritte verwendet. Falls diese Modifikation nicht akzeptiert wird, bleibt der ursprüngliche Zustand des Systems als Ausgangszustand für weitere *Monte-Carlo*-Schritte erhalten. Um Werte von statistischer Signifikanz zu erhalten, wird eine große Anzahl solcher *Monte-Carlo*-Schritte evaluiert. Aus dem Verhältnis der Anzahl der akzeptierten *Monte-Carlo*-Schritte zur gesamten Anzahl an *Monte-Carlo*-Schritten können dann makroskopische Eigenschaften des Systems abgeleitet werden.

Die Folge akzeptierter Zustände stellt eine Markovkette dar, da jeder neue Zustand von den vorherigen Zuständen abhängt. Die *Monte-Carlo*-Molekularsimulation kann damit zu den MCMC-Simulationen gezählt werden und stellt ein inhärent serielles Problem dar. Entsprechend schwierig gestaltet sich die Parallelisierung.

2.3 Grand-Canonical Monte-Carlo-Simulation

Die in dieser Studienarbeit betrachtete Implementierung einer MCMC-Simulation ist eine GCMC-Simulation. Es handelt sich hierbei um ein offenes System mit konstanter Temperatur T und konstantem Volumen V. Die in der GCMC-Simulation auftretenden *Monte-Carlo*-Schritte können in drei Typen von *Monte-Carlo*-Schritten aufgeteilt werden:

- Bewegen eines Moleküls durch Translation oder Rotation
- Hinzufügen eines Moleküls
- Entfernen eines Moleküls

Die Wahrscheinlichkeit, dass eine neue Konfiguration akzeptiert wird, hängt dabei vom Typ des *Monte-Carlo*-Schritts ab. Für die Translation beziehungsweise Rotation eines Moleküls ergibt sich die Akzeptanzwahrscheinlichkeit zu

$$Pr(o \to n) = \min\left(1, \exp\left(-\frac{E_n - E_o}{k_B T}\right)\right). \tag{2.3}$$

Der Term $E_n - E_o$ bezeichnet dabei die Energiedifferenz, die das System durch das Bewegen eines Moleküls $(o \to n)$ erfährt. Die Konstante k_B bezeichnet die Boltzmann-Konstante. Für das Hinzufügen beziehungsweise Entfernen eines Partikels müssen neben der Energiedifferenz zusätzliche Parameter für die Akzeptanzwahrscheinlichkeit berücksichtigt werden. Eine genaue Beschreibung dieser Wahrscheinlichkeiten wird in [4] gegeben.

Werden mehrere verschiedene *Monte-Carlo*-Schritte von einer Ausgangskonfiguration erzeugt, werden diese als *Mutanten* [1] bezeichnet. Handelt es sich bei jedem dieser *Mutanten* um eine Translation oder Rotation eines Moleküls, kann nach der von [5] vorgestellten Methode der erfolgsversprechendste *Mutant* gewählt werden und mit der in Gleichung 2.3 beschriebenen Wahrscheinlichkeit akzeptiert werden.

3 Rechnerarchitekturen

Prozessoren lassen sich nach der Ausrichtung ihrer Architektur in zwei Kategorien einteilen: latenzoptimierte Prozessoren und durchsatzoptimierte Prozessoren. Während latenzoptimierte Prozessoren darauf ausgelegt sind, einen Strom von Instruktionen auf einer einzelnen Datenmenge möglichst schnell zu verarbeiten, sind durchsatzoptimierte dafür konzipiert einen Instruktionsstrom auf eine Vielzahl von Datensätzen parallel anzuwenden. Diese Unterschiede in der Architektur entspringen unterschiedlichen Anforderungen durch die Einsatzbereiche der jeweiligen Prozessoren.

3.1 Latenzoptimierte Prozessoren

Die Entwicklung von latenzoptimierten Prozessoren (CPUs) wurde vor allem durch den Einsatz in Personal Computern und Servern vorangetrieben. Die Nachfrage nach immer leistungsfähigeren CPUs führte dabei nicht nur zu einer Skalierung der Technologiegröße, um höhere Taktfrequenzen zu erzielen. Es wurden darüberhinaus auch eine Reihe von Techniken zur Ausnutzung der Unabhängkeit zwischen verschiedenen Instruktionen (*Instruction-Level Parallelism*) entwickelt, die eine höhere Auslastung der Prozessoren erlauben und somit die Leistung dieser Prozessoren steigern. Als Vertreter dieser Techniken sind zum Beispiel der Einsatz verschieden stark ausgeprägter Pipelines, Methoden zur Sprungvorhersage oder das dynamischen Scheduling mehrerer Threads zu nennen [6]. Das Resultat dieser Entwicklungen stellt eine Rechnerarchitektur dar, die einen Strom von Instruktionen so schnell wie möglich verarbeiten kann. In aktuellen latenzoptimierten Prozessoren werden des Weiteren mehrere Kerne pro Prozessor verwendet, wodurch vor allem die parallele Verarbeitung mehrer Prozesse ermöglicht wird (*Thread-Level Parallelism*).

Die Speicherarchitektur dieser Prozessoren besteht meist aus einer Hierarchie von Cache-Speichern unterschiedlicher Größe und unterschiedlicher Zugriffszeiten. Diese Cache-Speicher sind durch verschiedene Ebenen klassifiziert, der sogenannte Level-1 Cache bezeichnet dabei den Cache, der für den Prozessor die kleinste Zugriffszeit bietet, jedoch auch die geringste Kapazität hat. Aufsteigend in dieser Hierarchie sind die Cache-Speicher nach ihrer Kapazität geordnet, die Zugriffzeit nimmt dabei von Ebene zu Ebene zu. Wird ein Datensatz angefordert, wird diese Cache-Hierarchie aufsteigend durchsucht und gegebenenfalls aus dem Hauptspeicher geladen. Dabei werden nicht einzelne Datenwörter, sondern ein kompletter Block an Daten geladen. Neben der Ausnutzung der gesamten Breite des Datenbusses werden dadurch benachbarte Daten mit in den Cache-Speicher geladen, die bei einer sukzessiven Verarbeitung eventuell in den darauffolgenden Berechnungsschritten

benötigt werden. Diese Cache-Hierarchie stellt dabei eine transparente Speicherstruktur dar, deren Verwaltung direkt durch den Prozessor erfolgt.

Als Beispiel für einen latenzoptimierten Prozessor sei der Intel® Core i7 Prozessor mit sechs Kernen, einer Taktfrequenz von 3.3 GHz und drei Level von Cache-Speichern angeführt.

3.2 Durchsatzoptimierte Prozessoren

Im Gegensatz zu latenzoptimierten Prozessoren sind durchsatzoptimierte Prozessoren (GPGPUs) darauf ausgelegt, einen Strom von Instruktionen auf eine große Datenmenge parallel anzuwenden. Diese Eigenschaft erfüllen Grafikprozessoren, die beispielsweise für jedes Ausgabepixel über entsprechende Berechnungen bestimmen müssen, welche Farbe an diesem Pixel angezeigt werden soll. Lange Zeit wurden dazu feste Hardwareeinheiten verwendet, die für eine bestimmte Funktion in der Computergrafik-Pipeline entworfen wurden. Durch die Forderung nach immer flexibler konfigurierbaren Grafikprozessoren wandelten sich diese in programmierbare, parallele Prozessoren [7].

Im Gegensatz zu den latenzoptimierten Prozessoren wird bei den durchsatzoptimierten Prozessoren auf eine große Anzahl einfacher Kerne gesetzt welche in der Lage sind, einen großen Satz von Daten parallel zu verarbeiten. Für die Parallelisierung der Ausführung von Instruktionen wird eine bestimmte Anzahl von einzelnen Kernen zu einem Multiprozessor zusammengefasst, wobei alle Kerne dieses Multiprozessors den gleichen Strom von Instruktionen auf mehrere Datensätze anwenden (*Single Instruction Multiple Data*). Ergeben sich hierbei Verzweigungen im Programmablauf werden beide Ausführungszweige nacheinander betrachtet und die Instruktionen nur auf den Kernen ausgeführt, die diesen Zweig beschreiten. Alle anderen Kerne des Multiprozessors werden während dieser Zeit pausiert oder wenden sich anderen Berechnungsaufgaben zu, wenn dies durch die Architektur erlaubt wird. Diese massive Parallelität stellt besondere Anforderungen an die Speicherarchitektur und den Umgang mit diesen Speicherstrukturen bei der Implementierung einer bestimmten Berechnung.

Die verschiedenen Ebenen einer Speicherarchitektur können nach ihrer Lokalität klassifiziert werden und besitzen unterschiedliche Zugriffszeiten. Jeder parallel ausgeführte Thread besitzt auf dem jeweiligen Kern einen lokalen Speicher, auf den nur von diesem Thread aus zugegriffen werden kann. Dieser lokale Speicher (*Local Memory*) wird meist durch Register realisiert und hat sehr kurze Zugriffszeiten. Zur Kommunikation zwischen den auf den Kernen eines Multiprozessors ausgeführten Threads wird ein gemeinsam genutzter Speicher (*Shared Memory*) benötigt, der ebenfalls einen schnellen Zugriff auf die Daten in diesem Speicher erlaubt. Der globale Speicher (*Global Memory*) stellt die weitreichendste Ebene in der Speicherarchitektur dar, in den von allen Threads auf allen Kernen aus zugegriffen werden kann. Dieser besitzt die größte Kapazität, die Zugriffszeiten auf diesen Speicher sind jedoch am höchsten.

Je nach spezifischer Architektur variiert die Größe der eingesetzten Speicher auf diesen drei Ebenen, werden noch zusätzliche Caches eingeführt oder zusätzliche Speicherstrukturen

mit einem bestimmten Ziel eingeführt. Als Beispiel für diese speziellen Speicherstrukturen können schreibgeschützte, globale Speicher mit schnellen Zugriffszeiten genannt werden.

Bei der Entwicklung von Anwendungen für diese Architektur muss durch den Programmierer festgelegt werden, welche Speicher für welche Variablen verwendet werden soll. Ein entsprechendes Wissen um die unterschiedlichen Speicher und ihre Zugriffszeiten bestimmt dabei die Effizienz der Implementierung. Die Kommunikation zwischen Threads erfordert bei der Implementierung des Weiteren spezielle Instruktionen, die die Kerne eines Multiprozessors während der Ausführung synchronisieren.

In der aktuellen Generation von GPGPUs werden beispielweise in der NVIDIA® Kepler® Architektur 1536 CUDA® Kerne eingesetzt, welche zu je 192 Kernen pro *Streaming Multiprocessor* gruppiert werden. Die Taktfrequenz dieser Kerne beträgt rund 1000 MHz.

3.3 Ausblick

Die beiden beschriebenen Architekturen für Prozessoren wurden für unterschiedliche Problemstellungen entwickelt und optimiert. In der aktuellen Entwicklung lässt sich ein Trend erkennen, bei dem diese verschiedenen Architekturen stärker zusammenwachsen und somit verstärkt von den jeweiligen Vorteilen der anderen Architektur profitieren können. [7, 8]. Des Weiteren lässt sich feststellen, dass die für eine Architektur entwickelten Konzepte verstärkt auch Einsatz in der anderen Architektur finden. Als Beispiel hierfür können für GPGPUs einerseits komplexer werdende Pipelines für die einzelnen Kerne angeführt werden, andererseits werden dynamische Scheduling-Methoden in kommenden Generationen von GPGPUs verstärkt eine Parallelisierung der Nutzung der GPGPU auf Prozessebene ermöglichen. Aktuelle Entwicklungen gehen bereits in die Richtung, latenzoptimierte Mehrkernprozessoren und durchsatzoptimierte GPGPUs auf einem einzigen Chip zu integrieren [9, 10].

Für die weitere Betrachtung unterschiedlicher Rechnerarchitekturen in dieser Arbeit werden diese folgendermaßen eingeteilt: Die Hostarchitektur als Hauptsystemarchitektur und die optionalen Devicearchitektur(en).

4 Hybrid-Simulation-State-Machine

4.1 Konzept

Die simultane Verwendung mehrerer Rechnerarchitekturen in ein und derselben Anwendung stellt eine besondere Anforderungen an ein Konzept, das die Vorteile der jeweiligen Architekturen vereinen will. Jede Architektur besitzt spezielle Eigenheiten, deren Beachtung oder Nichtbeachtung Auswirkungen auf die Effizienz der Implementierung haben. Darüberhinaus ist die architekturübergreifende Kommunikation von Daten ein Faktor, der großen Einfluss auf die Laufzeit der gesamten Anwendung haben kann. Eine Anforderung an die Konzeption einer solchen Anwendung ist also die Minimierung der architekturübergreifenden Kommunikation von Daten. Das in dieser Studienarbeit entwickelte Framework zur Molekularsimulation wurde als endlicher Automat unter Verwendung einer *Replicated-Data* Strategie entworfen: Die *Hybrid-Simulation-State-Machine*.

4.1.1 Definition von Simulationskerneln

Hauptaugenmerk der Konzeption der *Hybrid-Simulation-State-Machine* liegt auf der Geschwindigkeit der Evaluation von *Monte-Carlo*-Schritten, da dies den Großteil der Laufzeit einer MCMC-Simulation umfasst. Die Menge aller für eine bestimmte Architektur optimierter Berechnungsschritte zur Evaluation von *Monte-Carlo*-Schritten wird im Folgenden als Simulationskernel bezeichnet. Setzt sich das Resultat der Auswertung von *Monte-Carlo*-Schritten aus mehreren unabhängigen Teilberechnungen zusammen, kann ein Simulationskernel auch nur für eines dieser Teilprobleme definiert werden. Ein Simulationskernel beschreibt demnach eine bestimmte Methode, eine Berechnung für ein Teilproblem der Auswertung von *Monte-Carlo*-Schritten durchzuführen.

Diese Abstraktion der Berechnung hat mehrere Vorteile zur Folge: Zum Einen wird die *Hybrid-Simulation-State-Machine* unabhängig von einer bestimmten Devicearchitektur und zum Anderen erlaubt diese Einteilung eine hardwarenahe Implementierung für jeden Simulationskernel.

Durch die Unabhängigkeit von der Devicearchitektur kann die *Monte-Carlo* Simulation ohne Weiteres nur durch einen Simulationskernel auf Hostarchitektur durchgeführt werden. Außerdem kann jederzeit ein für eine andere Architektur optimierter Simulationskernel eingesetzt oder sogar mehrere unterschiedliche Simulationskernel auf unterschiedlichen Architekturen in der gleichen MCMC-Simulation verwendet werden. Durch die diversen Parameter für die Simulation haben MCMC-Simulationen haben je nach Einsatzgebiet viele

verschiedene Freiheitsgrade. Manche Berechnungsmethoden können nur unter bestimmten Voraussetzungen für diese Parameter eingesetzt werden oder wurden unter bestimmten Annahmen optimiert. Je nach Parameter der Simulation können damit unterschiedliche, für diese Szenarien optimierte Simulationskernel eingesetzt werden, um eine möglichst effiziente Simulationsumgebung zu schaffen. Die Abbildung 4.1 zeigt zwei verschiedene Kernel zur Summation von vier Zahlen auf unterschiedlichen Architekturen. Erlaubt Architektur A die parallele Ausführung von Instruktionen, kann diese Summe von Simulationskernel A in 2 Zeitschritten ausgewertet werden. Ist dies nicht der Fall, benötigen Simulationskernel A und Simulationskernel B beide 3 Zeitschritte zur Ermittlung der Summe.

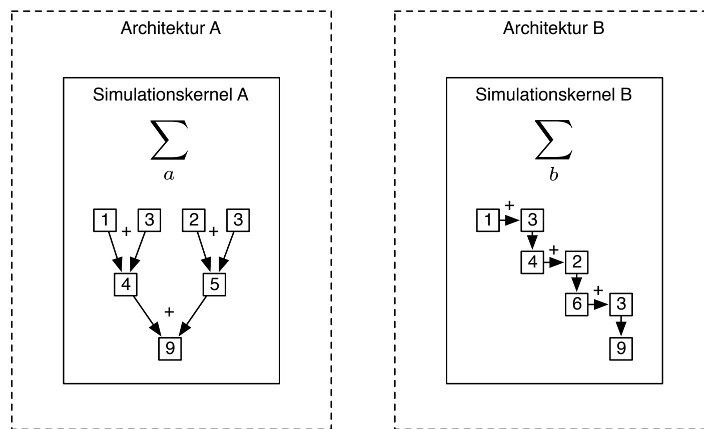


Abbildung 4.1: Zwei Simulationskernel zur Summation von vier Zahlen. Simulationskernel A basiert auf der Summation von partiellen Summen, Simulationskernel B auf einer rein seriellen Summation. Ein Simulationskernel beschreibt in der *Hybrid-Simulation-State-Machine* eine für eine bestimmte Architektur optimierte Methode zur Auswertung von *Monte-Carlo*-Schritten.

Die hardwarenahe Implementierung umfasst nicht nur die Konzeption der Berechnung, sondern auch die Organisation der Datenstrukturen. Je nach Architektur müssen entsprechende Randbedingungen eingehalten werden, um einen effizienten Datenzugriff zu ermöglichen (siehe Kapitel 3).

Eine flexible, für weitere Simulationskernel erweiterbare Lösung wurde deshalb bei der Konzeption der *Hybrid-Simulation-State-Machine* angestrebt.

Im Zuge dieser Studienarbeit wurden zwei Simulationskernel auf Hostarchitektur, sowie zwei Simulationskernel auf hybriden Architekturen für eine GCMC-Molekularsimulation implementiert. Die Simulationskernel auf Hostarchitektur beruhen dabei auf einer rein seriellen Implementierung, für die Simulationskernel auf hybriden Architekturen wurde die Parallelisierung von GCMC-Simulationen nach [1] implementiert. Des Weiteren wurde jeweils eine Version mit der in Kapitel 6 beschleunigten Berechnung implementiert. Alle Simulationskernel können gleichermaßen zur Berechnung der verschiedenen Anteile der Gesamtenergie eingesetzt werden.

4.1.2 Replicated-Data-Schema

Nachdem die Simulationskernel nun für jede Architektur mit einer lokalen Datenstruktur sehr flexibel definiert wurden und dementsprechend viel Freiraum bei der Optimierung der Simulationskernel bieten, stellt die Kommunikation von Daten zwischen den Architekturen einen möglichen Flaschenhals für die Simulation dar.

Eine Minimierung der Kommunikationskomplexität zwischen den Architekturen bedeutet das lokale Vorhalten signifikanter Daten in jeder Architektur. Hier kommt die *Replicated-Data* Strategie zur Anwendung: Jeder aktive Simulationskernel hält ein eigenes, vollständiges, kompaktes und auf die Architektur angepasstes Abbild der Simulation in seinem lokalen Speicher. Auf diese Weise kann in jeder Architektur für jeden Simulationskernel der schnelle Zugriff auf die benötigten Daten sichergestellt werden. Weiterhin kann die zur Speicherung verwendete Datenstruktur von Simulationskernel zu Simulationskernel variieren, um den Anforderungen der optimierten Berechnung gerecht zu werden. In Abbildung 4.2 sind zwei Simulationskernel A und B dargestellt, welche unterschiedliche Datenstrukturen zur Speicherung des Datensatzes verwenden.

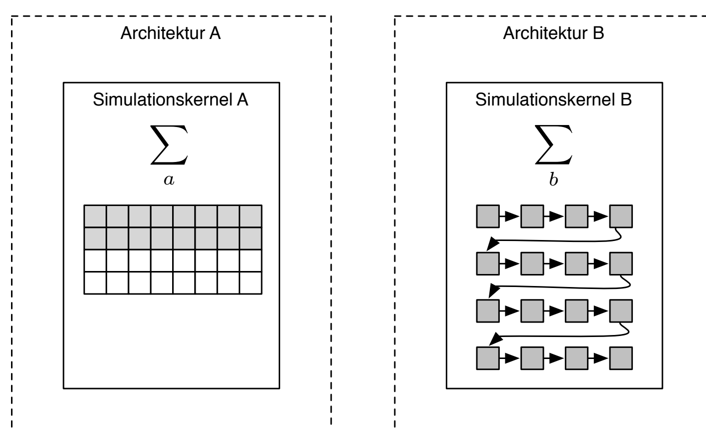


Abbildung 4.2: Zwei verschiedene Simulationskernel A und B. Simulationskernel A nutzt die Datenstruktur eines Arrays. Simulationskernel B nutzt die Datenstruktur einer verketteten Liste.

Im Bereich der Molekulardynamik kann für kurzreichweitige Potentiale eine räumliche Dekomposition der Daten in Zellen vorgenommen werden, sodass für ein Partikel einer bestimmten Zelle nur diese und die benachbarten Zellen zur Auswertung des Potentials dieses Partikels herangezogen werden müssen. Dieses Schema ist als *Linked-Cell* Methode bekannt und bevorzugt die Repräsentation der Partikel einer Zelle als verkettete Liste [11]. Für die Berechnung von langreichweitigen Potentialen hingegen muss jeder Partikel unabhängig von seiner räumlichen Position zur Berechnung herangezogen werden. Das Speichern der Partikel in solch einer Datenstruktur erfährt keine Beschleunigung, unter bestimmten Umständen kann sogar das Gegenteil der Fall sein.

4.1.3 Architekturübergreifende Kommunikation

Durch den Einsatz des *Replicated-Data*-Schemas und einer entsprechenden Initialisierung des Startzustands für alle Architekturen kann sichergestellt werden, dass sich alle Architekturen im selben Zustand der Simulation befinden. Im Laufe der Simulation müssen dementsprechend alle vom Anwender akzeptierten Änderungen am Zustand der Simulation an die verschiedenen aktiven Simulationskernel propagiert werden. Nur diese akzeptierten Änderungen zu kommunizieren stellt dabei das Minimum an Kommunikation dar, die aufgewendet werden muss, um den Zustand der Simulationen aller Simulationskernel konsistent zu halten. Auch zur Auswertung der *Monte-Carlo*-Schritte müssen die diesen Schritt betreffenden Daten an die Simulationskernel weitergeleitet werden. In der *Hybrid-Simulation-State-Machine* wird durch eine lokale Speicherung der Beschreibung dieser *Monte-Carlo*-Schritte für die Auswertung in den Simulationskerneln erreicht, dass diese Daten bei Akzeptierung eines *Monte-Carlo*-Schrittes nicht erneut kommuniziert werden müssen.

Die in der MCMC-Simulation auszuführenden Änderungen am System betreffen meist lediglich einen Bruchteil der gesamten Datenmenge. Im Fall der verwendeten GCMC-Simulation erfährt pro Simulationsschritt nur ein Partikel aus der Menge aller N Partikeln eine Änderung. Dieser *Monte-Carlo*-Schritt kann dabei aus dem Bewegen eines Partikels durch Translation oder Rotation, dem Hinzufügen eines Partikels oder dem Entfernen eines Partikels bestehen.

4.2 Hybrid-Simulation-Manager

4.2.1 Verwaltung von Simulationskerneln

Mit steigender Anzahl an Berechnungsmöglichkeiten und verschiedenen Architekturen steigt auch die Nachfrage nach einer Instanz, die zuständig für die Verwaltung der unterschiedlichen Simulationskernel ist. Die verschiedene Änderungen müssen an die unterschiedlichen Simulationskernel weitergegeben werden, Simulationen gestartet und Ergebnisse für den Anwender vorgehalten werden. Diese Aufgaben erfüllt der *Hybrid-Simulation-Manager*. Als übergeordnete Instanz auf Hostarchitektur übernimmt er die Arbeit des Vermittlers zwischen Simulationskernel und Anwender.

Dazu gehört einerseits die Bereitstellung einer Schnittstelle für den Anwender. Über diese Schnittstelle erfolgt die Initialisierung der Simulation und die Definition des Startzustandes. Während der Simulation erhält der Anwender einen transparenten Zugriff auf den Zustand der Simulation, ohne die Konsistenz der Daten zu gefährden. Über diese Schnittstelle werden weiterhin vom Anwender generierte *Monte-Carlo*-Schritte entgegengenommen, sowie die zu diesen *Monte-Carlo*-Schritten gehörenden Ergebnisse bereitgestellt.

Andererseits stellt der *Hybrid-Simulation-Manager* auch die Schnittstelle zu den Simulationskerneln dar. In dieser Aufgabe ist er für die Einhaltung der Konsistenz des *Replicated-Data*-Schemas verantwortlich.

Der *Hybrid-Simulation-Manager* erfüllt außerdem kleinere Berechnungsaufgaben, die nicht von einer Portierung profitieren und demnach zugunsten einer schlanken Konzeption der Simulationskernel aus deren Verantwortungsbereich gezogen werden.

Die Abbildung 4.3 zeigt die *Hybrid-Simulation-State-Machine* mit zwei unterschiedlichen Devicearchitekturen und drei verschiedenen Simulationskerneln. Simulationskernel A und Simulationskernel B arbeiten auf ihrer jeweiligen Devicearchitektur, Simulationskernel H wird direkt auf Hostarchitektur eingesetzt. Der *Hybrid-Simulation-Manager* steht dabei als Vermittler zwischen Anwender und den Simulationskerneln. Die lokale Datenstruktur der Simulationskernel im *Replicated-Data*-Schema wird jedem Simulationskernel selbst überlassen, jegliche Änderung von Daten erfolgt über die mit durchgehenden Pfeilen dargestellten Kommunikationswege. Lediglich der *Hybrid-Simulation-Manager* hat direkten Zugriff auf den Speicher der Hostarchitektur, um dem Anwender Informationen über den aktuellen Zustand der Simulation bereitstellen zu können.

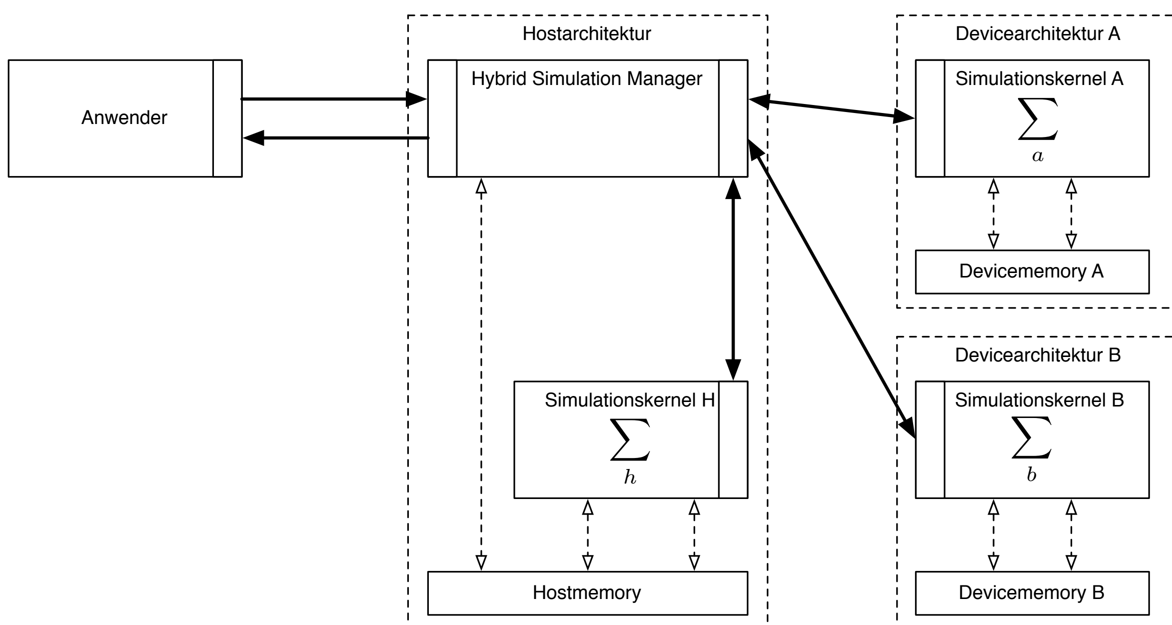


Abbildung 4.3: Überblick über eine *Hybrid-Simulation-State-Machine* mit zwei verschiedenen Devicearchitekturen und zugehörigen Simulationskerneln. Kommunikationswege sind mit durchgehenden Pfeilen dargestellt, Zugriffsrechte auf Speicherstrukturen mit gestrichelten Pfeilen.

4.2.2 Automatisierte Wahl von Simulationskerneln

Der *Hybrid-Simulation-Manager* kann zur automatisierten Wahl von Simulationskernel eingesetzt werden, allerdings sind die dazu benötigten Methoden sehr von der verwendeten

MCMC Simulation und von den verwendeten Simulationskerneln abhängig. Sie müssen nach einer expliziten Betrachtung des spezifischen Problems für jede MCMC-Simulation separat implementiert werden.

Die vorliegende GCMC-Simulation erlaubt in jedem *Monte-Carlo*-Schritt nur das Hinzufügen bzw. Entfernen maximal eines Moleküls. Unter der Annahme, dass für eine bestimmte Simulation eine konstante Anzahl von simultan evaluierten *Monte-Carlo*-Schritten betrachtet wird, kann zu Beginn der Simulation eine Laufzeitermittlung für eine Menge von K Simulationskerneln erstellt werden. Für den schnellsten Simulationskernel wird diese Laufzeit als t_{ref} gespeichert, zusammen mit der Molekülanzahl n_{ref} im System zu diesem Zeitpunkt. Für jeden Simulationskernel wird weiterhin die Laufzeit $t_{i,ref}$ und der aktuelle Beschleunigungsfaktor s_i des schnellsten Simulationskernel zum Simulationskernel i gespeichert.

Weicht die Molekülanzahl n_0 zu einem beliebigen Zeitpunkt um mehr als einen festgelegten Prozentsatz x von n_{ref} ab, wird eine erneute Laufzeitermittlung durchgeführt. Ergibt sich dabei, dass ein Simulationskernel schneller als der zur Zeit verwendete Simulationskernel ist, wird dieser im Anschluss zur Simulation eingesetzt.

Um auf eine Auslastung der Architekturen durch andere Anwendungen zu reagieren, wurde in dieser Studienarbeit eine Methode zur automatischen Wahl von Simulationskerneln für die vorliegende GCMC-Simulationen entwickelt. Ein Ziel bei der Konzeption war es dabei, dieses Scheduling unabhängig von einer globalen Synchronisationseinheit für mehrere nebenläufige Simulationsinstanzen durchzuführen. Für jede Simulationsinstanz soll lokal entschieden werden, welcher Simulationskernel eingesetzt werden soll, diese Entscheidung ist abhängig von der Ausführungszeit der unterschiedlichen Simulationskernel kann damit auch auf die Auslastung der Architekturen durch andere Anwendungen reagieren.

Die hier vorgestellte Methode des Schedulings von zwei Simulationskerneln $CPUSim$ und $GPGPUSim$ nutzt dabei aus, dass durch das *Replicated-Data*-Schema ein schneller Wechsel der Simulationskernel erreicht werden kann, ohne Datenmengen von einem Simulationskernel zum Anderen zu kommunizieren. Ein weiterer Vorteil des *Replicated-Data*-Schemas liegt darin, dass die zur Laufzeitmessung auf anderen Simulationskerneln ausgeführten Simulationsschritte nicht zusätzlich auf einem anderen Simulationskernel nochmals berechnet werden müssen. Die in diesem Simulationsschritt ermittelten Ergebnisse stimmen mit den Ergebnissen von anderen Simulationskerneln überein und können direkt als Ergebnis für den Anwender bereitgestellt werden.

Im Folgenden wird angenommen, dass die betrachtete Molekülanzahl, sowie die Anzahl simultan evaluierter *Monte-Carlo*-Schritte annähernd konstant ist. Schwankungen in der Laufzeit der betrachteten Simulationskernel kommen also aus einer erhöhten Gesamtauslastung der Architekturen durch andere Anwendungen. Im ersten Simulationsschritt werden beide Simulationskernel zur Berechnung eingesetzt und dabei von jedem Simulationskernel die benötigte Laufzeit gemessen. Die Laufzeit des schnellsten Simulationskernels wird als t_{ref} gespeichert und dieser Simulationskernel wird anschließend für die nachfolgenden Simulationsschritte eingesetzt. Dieser schnellste Simulationskernel sei im Folgendenden der Simulationskernel $CPUSim$, die Betrachtung für $GPGPUSim$ als schnellsten Simulationskernel ist analog. Zusätzlich zur Laufzeit des schnellsten Simulationskernels wird

der Beschleunigungsfaktor (Speedup) $s = t_{GPGPUSim}/t_{CPUSim}$ gespeichert, um den *CPUSim* schneller ist als *GPGPUSim*, damit gilt $s > 1$. $t_{CPUSim} = t_{ref}$ ist dabei die Laufzeit von *CPUSim*, $t_{GPGPUSim}$ die Laufzeit von *GPGPUSim*. Im Algorithmus wird t_{ref} als die minimale Ausführungszeit eines beliebigen Simulationskernels gespeichert. Die Initialisierung zu Beginn der Simulation ist in Algorithmus 4.1 dargestellt.

Algorithmus 4.1 AutomaticSchedulingSimulation, Initialisierung der Simulation

```
1:  function INITIALIZATION()
2:      t_CPUSim ← simulateCPUSim()
3:      t_GPGPUSim ← simulateGPGPUSim()
4:      if t_CPUSim < t_GPGPUSim then
5:          currentKernel ← CPUSim
6:          s ← t_GPGPUSim / t_CPUSim
7:          t_ref ← t_CPUSim
8:      else
9:          currentKernel ← GPGPUSim
10:         s ← t_CPUSim / t_GPGPUSim
11:         t_ref ← t_GPGPUSim
12:     end if
13: end function
```

Bei jedem weiteren normalen Simulationsschritt wird die aktuelle Laufzeit t_{new} von *CPUSim* gemessen und dazu eine durchschnittliche Laufzeit $t_{CPUSim,avg}$ aktualisiert. Dabei wird ein konstantes Intervall von Simulationsschritten Δp betrachtet, die Laufzeiten werden in einem Feld t_{CPUSim} mit Größe Δp gespeichert. Für den Simulationsschritt p wird dabei der Speicherort $t_{CPUSim}[mod(p, \Delta p)]$ verwendet. Die Aktualisierung der durchschnittlichen Laufzeit $t_{CPUSim,avg}$ erfolgt durch Subtraktion von $t_{CPUSim}[mod(p, \Delta p)]/\Delta p$ und Addition der aktuellen Laufzeit $t_{new}/\Delta p$. Der Wert von $t_{CPUSim,avg}$ stellt somit die durchschnittliche Laufzeit der letzten Δp untersuchten Simulationsschritte von *CPUSim* dar.

Die Idee bei dieser Methode ist es nun, andere Simulationskernel mit einer gewissen Wahrscheinlichkeit zur Auswertung der Simulationsschritte anstatt dem gerade eingesetzten Simulationskernel zu verwenden und die Laufzeit dieser Simulationskernel dabei zu beobachten. Durch einen Parameter a ($a > 1$) wird ein relativer Schwellenwert definiert, bei dessen Überschreitung der durchschnittlichen Laufzeit von *CPUSim* eine Simulation auf *GPGPUSim* in Betracht gezogen wird. Dieser Schwellenwert liegt bei $t_{CPUSim,avg} > a \cdot t_{ref}$. Durch den Parameter a kann dabei gesteuert werden, wie stark *CPUSim* versucht, wieder an die minimale Laufzeit t_{ref} zu kommen. Bleibt die durchschnittliche Laufzeit von *CPUSim* im Vergleich zu dem zu Beginn der Simulation ermittelten Referenzwert t_{ref} konstant, so wird keine Betrachtung durchgeführt. Überschreitet $t_{CPUSim,avg}$ diesen Schwellenwert in einem Simulationsschritt, wird der Simulationskernel *GPGPUSim* mit einer Wahrscheinlichkeit von $\frac{1}{s}$ zur Simulation dieses Simulationsschritts eingesetzt und dabei die Laufzeit von *GPGPUSim* ermittelt. Je höher der Beschleunigungsfaktor von *CPUSim* zu *GPGPUSim* ist, desto länger braucht *GPGPUSim* vermutlich für die Simulation und umso unwahrscheinlicher ist es, dass

GPGPUSim anstatt *CPUSim* zur Simulation eingesetzt und dessen Laufzeit ermittelt wird. Der Algorithmus 4.2 zeigt die Simulation mit automatischer Wahl von Simulationskerneln. In Zeile 9 ist die Bedingung für die Überschreitung des Schwellenwerts gegeben, in Zeile 10 ist wird abhängig vom Beschleunigungsfaktor s randomisiert entschieden, ob eine Simulation auf dem gerade nicht aktiven Kernel durchgeführt werden soll.

Algorithmus 4.2 Automatisches Scheduling für zwei Simulationskernel

1: **function** AutomaticSchedulingSimulation()
2:     // Initialisierung
3:     **if** $firstStep$ **then**
4:         $Initialization()$
5:         $firstStep \leftarrow false$
6:     **end if**
7:
8:     // Simulation auf anderem Kernel zur Laufzeituntersuchung
9:     **if** $simulatedSteps[currentKernel] > \Delta p$ & $t_{currentKernel,avg} > a \cdot t_{ref}$ **then**
10:         **if** $rand() < 1/s$ **then**
11:             **if** $currentKernel == CPUSim$ **then**
12:                 $benchmarkedSimulationGPGPUSim()$
13:             **else if** $currentKernel == GPGPUSim$ **then**
14:                 $benchmarkedSimulationCPUSim()$
15:             **end if**
16:             $simulated \leftarrow true$
17:         **end if**
18:     **end if**
19:
20:     // normale Simulation
21:     **if** $simulated \neq true$ **then**
22:         **if** $currentKernel = CPUSim$ **then**
23:             $t_{new} \leftarrow simulateCPUSim()$
24:             $updateTavg(CPUSim, t_{new})$
25:             $inc(simulatedSteps[CPUSim])$
26:         **else if** $currentKernel == GPGPUSim$ **then**
27:             $t_{new} \leftarrow simulateGPGPUSim()$
28:             $updateTavg(GPGPUSim, t_{new})$
29:             $inc(simulatedSteps[GPGPUSim])$
30:         **end if**
31:     **end if**
32: **end function**

Da auch die Laufzeit von *GPGPUSim* Schwankungen unterliegen kann, wird hierbei ebenfalls eine Reihe von Δp Simulationsschritten verlangt, um ein ausgeglicheneres Bild des Laufzeitverhaltens von *GPGPUSim* zu erhalten. Wurde der Schwellenwert von *CPUSim*

oft genug überschritten und *GPGPUSim* in Δp Simulationsschritten zur Simulation eingesetzt und entsprechende Werte zur Laufzeit ermittelt, können die durchschnittlichen Laufzeiten von *CPUSim* und *GPGPUSim* verglichen werden: Es ergibt sich ein aktueller Geschwindigkeitsfaktor $s_{new} = t_{CPUSim,avg} / t_{GPGPUSim,avg}$. Für $s_{new} > 1$ ist *GPGPUSim* schneller als *CPUSim* und kann als neuer Simulationskernel für die Simulation verwendet werden. Um einem häufiges Wechsel zwischen Simulationskerneln entgegen zu wirken, wird dieser *GPGPUSim* nun mit einer Wahrscheinlichkeit von $1 - \frac{1}{s}$ als neuer Simulationskernel eingesetzt. Sind beide Simulationskernel in etwa gleich schnell, gilt $s_{new} \sim 1$. Die Wahrscheinlichkeit eines Wechsels des Simulationskernel ist entsprechend gering. Je größer der Geschwindigkeitsfaktor wird, desto wahrscheinlicher ist ein Wechsel.

Bei einem Wechsel von *CPUSim* zu *GPGPUSim* müssen folgende Werte neu gesetzt werden: $t_{ref} := min(t_{ref}, t_{GPGPUSim,avg})$ und $s := t_{CPUSim,avg} / t_{GPGPUSim,avg}$.

Der Algorithmus 4.3 zeigt die Simulation auf dem gerade nicht aktiven Simulationskernel *GPGPUSim* mit der Laufzeitbestimmung.

Algorithmus 4.3 Simulation für GPGPUSim mit randomisiertem Kernelwechsel bei Speedup von GPGPUSim gegenüber CPUSim

```
function AutomaticSchedulingSimulation, BenchmarkedSimulationGPGPUSim
    t_new ← simulateGPGPUSim
    updateTavg(GPGPUSim, t_new)
    inc(simulatedSteps[GPGPUSim])
    s_new ← t_CPUSim,avg / t_GPGPUSim,avg
    if simulatedSteps[GPGPUSim] > Δp & s_new > 1 then
        // randomisierter Kernelwechsel
        if rand() < 1 − 1/s_new then
            // GPGPUSim als neuen Simulationskernel wählen
            currentKernel ← GPGPUSim
            simulatedSteps[GPGPUSim] ← 0
            simulatedSteps[CPUSim] ← 0
            s ← s_new
            t_ref ← min(t_ref, t_GPGPUSim,avg)
        end if
    end if
end function
```

Algorithmus 4.4 Simulation für CPUSim mit randomisiertem Kernelwechsel bei Speedup von CPUSim gegenüber GPGPUSim

```
function AutomaticSchedulingSimulation, BenchmarkedSimulationCPUSim
    t_new ← simulateCPUSim
    updateTavg(CPUSim, t_new)
    inc(simulatedSteps[CPUSim])
    s_new ← t_GPGPUSim,avg / t_CPUSim,avg
    if simulatedSteps[CPUSim] > Δp == 0)  &  s_new > 1 then
        // randomisierter Kernelwechsel
        if rand() < 1 − 1/s_new  then
            // CPUSim als neuen Simulationskernel wählen
            currentKernel ← CPUSim
            simulatedSteps[GPGPUSim] ← 0
            simulatedSteps[CPUSim] ← 0
            s ← s_new
            t_ref ← min(t_ref, t_CPUSim,avg)
        end if
    end if
end function
```

4.2.3 Verifikation von Ergebnissen und Fehlerbehandlung

Die Verwendung verschiedener Simulationskernel auf unterschiedlichen Architekturen birgt großes Potential bei der beschleunigten Berechnung von *Monte-Carlo*-Schritten. Dabei kann es durch die diversen Ausführungsreihenfolgen von Operationen in den verschiedenen Algorithmen und der beschränkten Rechengenauigkeit zu Rundungsfehlern bei der Berechnung von ein und desselben *Monte-Carlo*-Schrittes durch die unterschiedlichen Simulationskernel kommen. Besonders durch das große Spektrum an Freiheitsgraden bei der GCMC-Simulation können nicht alle Fehlermöglichkeiten ausgeschlossen werden. Eine entsprechende Fehlerbehandlung muss bei der Konzeption der *Hybrid-Simulation-State-Machine* in Betracht gezogen werden. Diese Aufgabe fällt ebenfalls in den Bereich des *Hybrid-Simulation-Manager*. Er verarbeitet dabei die Fehlermeldungen der Simulationskernel und kann entsprechend auf diese reagieren. Mögliche Reaktionen auf Fehlermeldungen sind die Speicherung des Zustands der Simulation, um diese nach einer Funktionsprüfung fortführen zu können, die Neuinitialisierung von Devicearchitekturen oder die Verwendung eines Simulationskernel einer anderen Architektur. Zudem kann er die Ergebnisse der auf Devicekerneln durchgeführten Berechnungen mit einer auf dem Hostkernel durchgeführten Berechnung vergleichen und so die Qualität der berechneten Ergebnisse sicherstellen.

In der in dieser Studienarbeit verwendeten GCMC-Simulation wurde der *Hybrid-Simulation-Manager* zur Verifikation von Ergebnissen der Devicearchitektur gegenüber der Referenzimplementierung auf Hostarchitektur eingesetzt.

4.3 Die Hybrid-Simulation-State-Machine als endlicher Zustandsautomat

Die *Hybrid-Simulation-State-Machine* wurde einerseits aus Aspekten der Integrität von Daten, andererseits aus Gründen der Effizienz als endlicher Zustandsautomat entworfen. Die Simulation wird hierbei in zwei Phasen getrennt: Die Initialisierungs- und die Simulationsphase. Eine MCMC-Simulation besitzt im Allgemeinen eine große Reihe an Parametern, die Einfluss auf die Größe und Laufzeit der MCMC-Simulation haben. Diese Parameter werden zumeist erst zur Laufzeit der Simulation festgelegt, entsprechend flexibel muss mit der Größe des zu simulierenden Systems umgegangen werden.

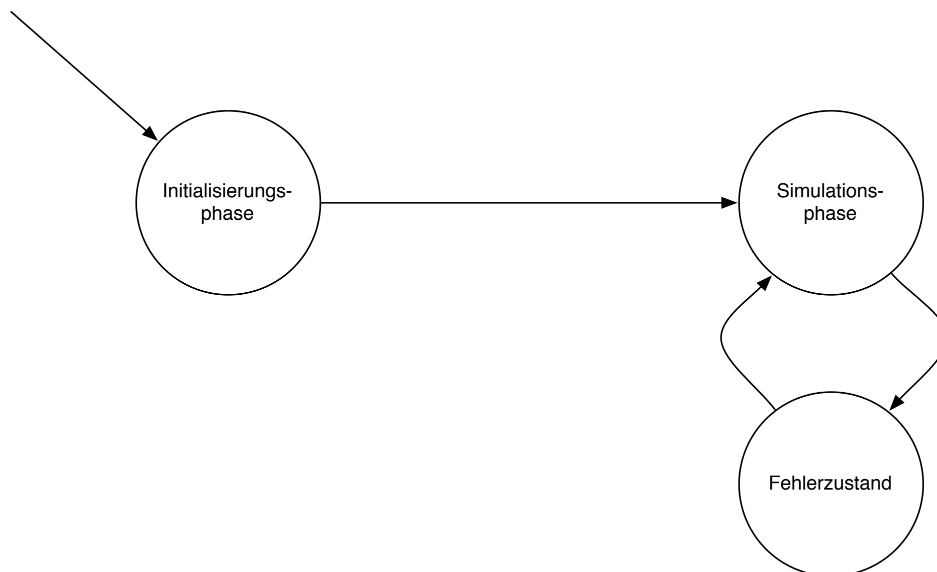


Abbildung 4.4: Zustandsdiagramm der *Hybrid-Simulation-State-Machine*

4.3.1 Initialisierungsphase

Zur Vereinfachung der Anforderungen an die Simulationskernel erfolgt die Initialisierung der Simulation in der Initialisierungsphase deshalb mit der für diese Zwecke definierten Schnittstelle zwischen Anwender und *Hybrid-Simulation-Manager*.

Zu dieser Initialisierung gehört neben der Definition der Simulationsparameter und der Simulationsumgebung auch der Startzustand für die MCMC-Simulation. Nach dieser Initialisierungsphase werden keine weiteren Änderungen dieser Parameter angenommen, um die Konsistenz der Simulationszustands in den verschiedenen Simulationskerneln nicht zu gefährden und den Simulationskerneln die Möglichkeit des Aufbaus einer effizienten Datenstruktur zu bieten.

Erst beim Übergang zur Simulationsphase werden die Simulationsparameter und der Startzustand der Simulation in einer kompakten Form an die Simulationskernel propagiert. An

dieser Stelle kann eine Pufferung von Daten aus Berechnungen angestellt werden, die keine Abhängigkeit vom aktuellen Zustand der Simulation zeigen. Da in der *Hybrid-Simulation-State-Machine* nach der Initialisierungsphase keine weiteren Änderungen an Moleküldefinitionen zugelassen werden, erfahren diese Parameter keine Änderung in der Simulationsphase und können somit als konstant angesehen werden.

Für die GCMC-Simulation sieht die Initialisierungsphase der Simulation wie folgt aus: Es werden Atome mit ihren spezifischen Eigenschaften, wie zum Beispiel Masse und Ladung, definiert. Anschließend kann aus diesen Atomdefinitionen ein Molekül definiert werden. Neben verschiedenen Simulationsparametern wie der Größe des Simulationsgebiets und der vorherrschenden Temperatur muss zusätzlich ein Startzustand konfiguriert werden. Dazu werden die Moleküle im Simulationsgebiet verteilt. Am Ende dieser Initialisierungsphase werden alle Daten durch den *Hybrid-Simulation-Manager* an die aktiven Simulationskernel propagiert, welche ihre Kopie des Systemzustands in einer für sie effizienten Datenstruktur speichern.

4.3.2 Simulationsphase

Während der Simulationsphase erfolgt die Kommunikation zwischen Anwender und dem *Hybrid-Simulation-Manager* nach dem im Kapitel 5 beschriebenen Schema der *Change-Linked Simulation*. Der *Hybrid-Simulation-Manager* erlaubt während dieser Phase nur Abfragen bestimmter, für die Generierung von *Monte-Carlo*-Schritten erforderlicher Variablen und lässt keine Änderungen an denen in der Initialisierungsphase gesetzten Parametern zu.

4.3.3 Fehlerzustand

Da bei einer Simulation auf hybriden Architekturen Fehler nicht ausgeschlossen werden können, wird ein zusätzlicher Fehlerzustand in der *Hybrid-Simulation-State-Machine* eingeführt. Der *Hybrid-Simulation-Manager* kümmert sich hierbei um eine Fehlerbehandlung (s.o.).

5 Change-Linked-Simulation

In der Simulationsphase der *Hybrid-Simulation-State-Machine* erfolgt die Untersuchung der Verteilung von Zuständen eines MCMC-Systems im Gleichgewichtszustand über die Simulation von *Monte-Carlo*-Schritten. Die *Monte-Carlo*-Schritte werden dabei durch vom Anwender definierte Methoden generiert und müssen in der *Hybrid-Simulation-State-Machine* simuliert werden. Die resultierenden Ergebnisse werden dann für den Anwender bereitgestellt. Die *Change-Linked-Simulation* stellt das Konzept einer effizienten Schnittstelle für diese Aufgabe zwischen Anwender und *Hybrid-Simulation-State-Machine* dar.

5.1 Anforderungen an die Schnittstelle

Die Schnittstelle zwischen Anwender und *Hybrid-Simulation-State-Machine* soll eine intuitive, einfache Benutzung der *Hybrid-Simulation-State-Machine* während der Simulationsphase ermöglichen und die für eine effiziente Simulation benötigte Minimierung der Kommunikation berücksichtigen. Speziell bei der in dieser Studienarbeit betrachteten Parallelisierung der GCMC-Simulation werden mehrere *Monte-Carlo*-Schritte simultan im selben Simulationsschritt ausgewertet. Jeder dieser simultan ausgewerteten *Monte-Carlo*-Schritte hat andere Auswirkungen auf das Gesamtsystem. Dementsprechend liefert die Simulation mehrerer unabhängiger *Monte-Carlo*-Schritte für jeden *Monte-Carlo*-Schritt ein Ergebnis. Aufgrund dieser Mannigfaltigkeit von *Monte-Carlo*-Schritten in einem Simulationsschritt ist die exakte Zuordnung von Ergebnissen zum jeweiligen *Monte-Carlo*-Schritt erforderlich. Die Schnittstelle zwischen Anwender und der *Hybrid-Simulation-State-Machine* während der Simulationsphase muss demnach mehrere, in der Anzahl variable *Monte-Carlo*-Schritte für einen Simulationsschritt verwalten können. Da es sich bei der GCMC-Simulation um eine MCMC-Simulation handelt, sind weitere Restriktionen bei der Simulation und Akzeptierung von *Monte-Carlo*-Schritten notwendig, um die Markovketteneigenschaft der Simulation nicht zu verletzen. In einer MCMC-Simulation können verschiedene Typen *Monte-Carlo*-Schritten existieren, was ebenfalls eine weitere Anforderung an diese Schnittstelle darstellt. In der in dieser Studienarbeit betrachteten GCMC-Simulation kann ein *Monte-Carlo*-Schritt beispielsweise einem von drei Typen von *Monte-Carlo*-Schritten zugeordnet werden: Dem Bewegen eines Partikels durch Translation oder Rotation, dem Hinzufügen eines Partikels und dem Entfernen eines Partikels aus dem System.

5.2 Konzept

Das Kernkonzept der *Change-Linked-Simulation* besteht in einer Abtraktion bzw. Repräsentation von *Monte-Carlo*-Schritten der MCMC-Simulation durch *Änderungsinstanzen*. Jeder *Monte-Carlo*-Schritt wird dabei durch eine seperate *Änderungsinstanz* beschrieben. Die *Änderungsinstanzen* sind dabei gemeinsam genutzte Objekte, auf die sowohl von Anwenderseite, als auch von Seiten des *Hybrid-Simulation-Manager* aus zugegriffen werden kann. Bei der simultanen Evaluation mehrerer *Monte-Carlo*-Schritte wird ein Ergebnis für jeden *Monte-Carlo*-Schritt erzeugt. Für eine intuitive Schnittstelle ist es daher notwendig, die zum jeweiligen *Monte-Carlo*-Schritt gehörenden Ergebnisse der MCMC-Simulation in der jeweiligen *Änderungsinstanz* zu speichern. In Abbildung 5.1 wird diese Zuordnung der Ergebnisse der MCMC-Simulation anhand zweier *Änderungsinstanzen* dargestellt.

Jede *Änderungsinstanz* speichert die Beschreibung des zu simulierenden *Monte-Carlo*-Schritts, sowie die zugehörigen Ergebnisse aus der Simulation. Die Schreibrechte werden hierbei als durchgehende Pfeile, die Leserechte hingegen als gestrichelte Pfeile dargestellt.

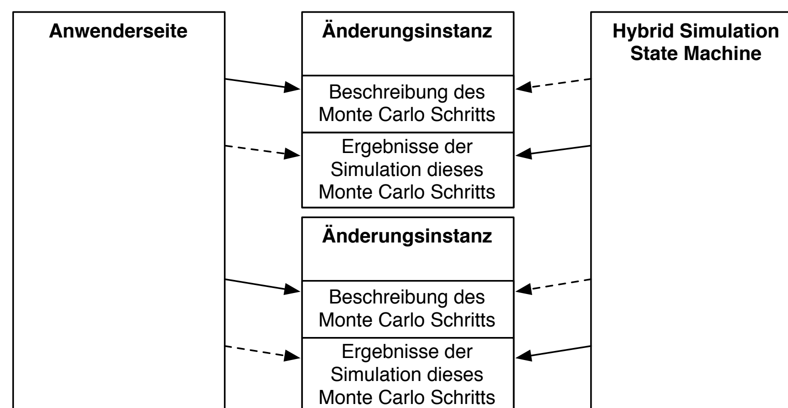


Abbildung 5.1: Zwei *Änderungsinstanzen* mit Darstellung von Schreibrechten (durchgehende Pfeile) und Leserechten (gestrichelte Pfeile)

Ausgehend von diesen *Änderungsinstanzen* wird bei der Simulation die Information über den *Monte-Carlo*-Schritt durch den *Hybrid-Simulation-Manager* in ein für den jeweiligen aktiven Simulationskernel der *Hybrid-Simulation-State-Machine* verständliches Format gebracht und an diesen Simulationskernel propagiert. Der Simulationskernel speichert diese Beschreibung des *Monte-Carlo*-Schritts aus zweierlei Gründen lokal. Einerseits muss die Information zur effizienten Berechnung des *Monte-Carlo*-Schritts lokal vorliegen. Andererseits wird bei der Akzeptierung eines *Monte-Carlo*-Schritts dadurch nur die Information benötigt, *welche* Änderungsinstanz akzeptiert wurde und nicht die gesamte Beschreibung des *Monte-Carlo*-Schritts. Der Simulationskernel kann dann aufgrund dieser Information die lokal vorhandene Repräsentation des akzeptierten *Monte-Carlo*-Schritts in seine lokale Repräsentation des Zustands der Simulation einfügen, ohne dass die expliziten Informationen über den *Monte-Carlo*-Schritt nochmals kommuniziert werden müssen.

5.3 Ablauf der Change-Linked-Simulation in der Hybrid-Simulation-State-Maschine

Die Abbildung 5.2 zeigt ein Ablaufdiagramm mit strukturellen Eigenschaften für einen Simulationsschritt der *Hybrid-Simulation-State-Machine* unter Verwendung der *Change-Linked-Simulation*. Auf der Anwenderseite werden dabei nach den Regeln der zu simulierenden MCMC-Simulation *m Monte-Carlo*-Schritte generiert und in den entsprechenden Änderungsinstanzen gespeichert. Der Typ jedes *Monte-Carlo*-Schrittes kann dabei frei gewählt werden. Zur Simulation der *Monte-Carlo*-Schritte werden diese nun durch den *Hybrid-Simulation-Manager* in ein für die vom Anwender gewählten Simulationskernel passendes Format gebracht und simuliert. Die Ergebnise jedes *Monte-Carlo*-Schrittes werden in den entsprechenden *Änderungsinstanzen* gespeichert und für den Anwender zur Weiterverarbeitung damit direkt mit der zugehörigen *Änderungsinstanz* verknüpft. Im Anschluss entscheidet der Anwender nach den Akzeptanzbedingungen für *Monte-Carlo*-Schritte der verwendeten MCMC-Simulation, welcher dieser *m Monte-Carlo*-Schritte akzeptiert werden soll. Diese Information nimmt der *Hybrid-Simulation-Manager* entgegen und leitet sie an die aktiven Simulationskernel der *Hybrid-Simulation-State-Machine* weiter, welche die Änderungen am Systemzustand durch diesen *Monte-Carlo*-Schritt auf Basis der bereits lokal vorhandenen Repräsentation des *Monte-Carlo*-Schritts durchführen.

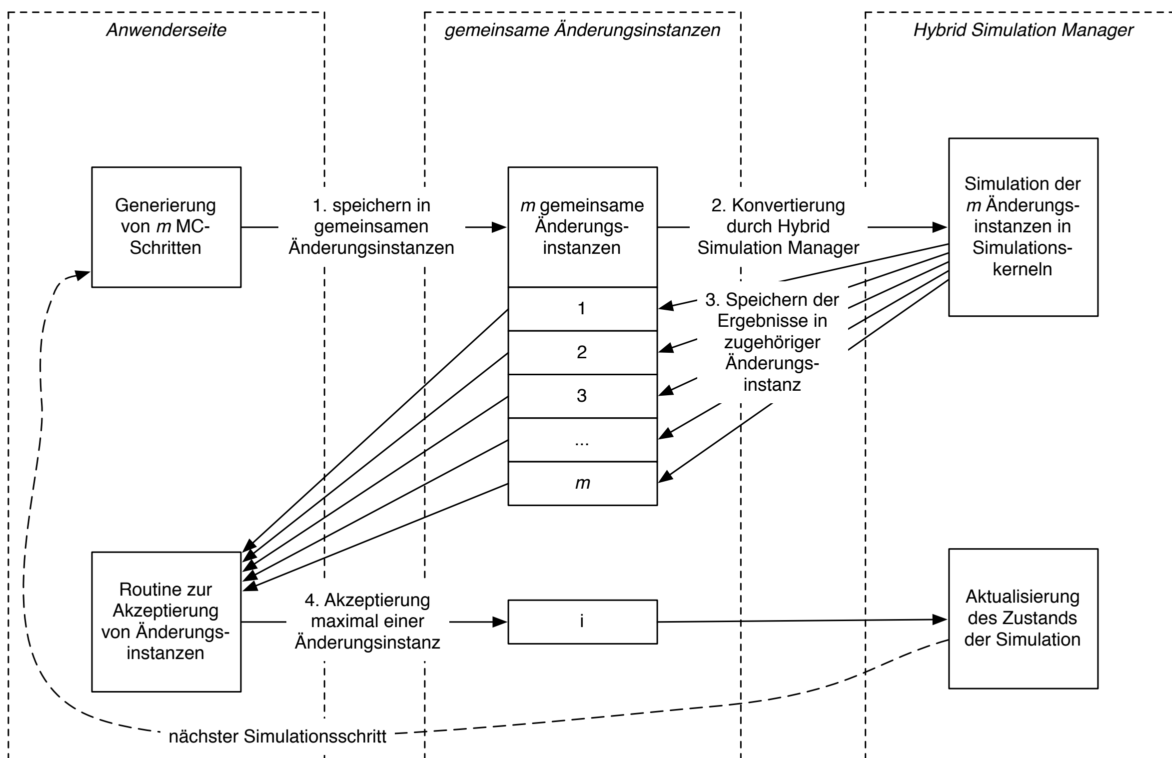


Abbildung 5.2: Ablaufdiagramm für einen Simulationsschritt der *Change-Linked Simulation*.

5.4 Einhaltung der Markovketteneigenschaft

Bei der Verwendung mehrerer *Monte-Carlo*-Schritte für jeden Simulationsschritt muss besonderer Wert darauf gelegt werden, die Markovketteneigenschaft der Simulation nicht zu verletzen. Entsprechend darf nur maximal ein von einem bestimmten Zustand der Simulation ausgehender *Monte-Carlo*-Schritt akzeptiert werden. Die zusätzliche Akzeptierung eines weiteren von diesem Zustand ausgehenden *Monte-Carlo*-Schritts würde die durch den ersten Schritt beschriebene Änderung am System nicht berücksichtigen. In Abbildung 5.3 ist die Verletzung der Markovketteneigenschaft exemplarisch an zwei *Monte-Carlo*-Schritten a und b dargestellt, die aus dem Zustand der Simulation zum Zeitpunkt t_s generiert wurden.

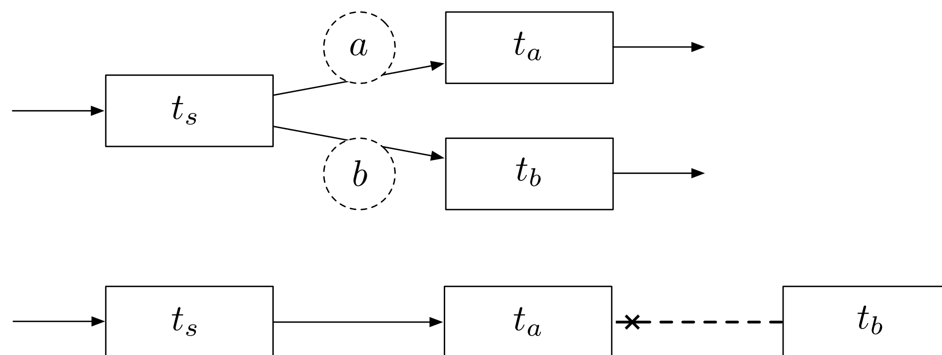


Abbildung 5.3: Oben: Zwei aus einem gemeinsamen Zustand t_s generierte Änderungen a und b und daraus resultierende Zustände t_a und t_b.
Unten: Verletzung der Markov Kette bei Akzeptierung zweier aus dem gleichen Zustand t_s generierten Änderungen a und b.

Nach Akzeptierung des *Monte-Carlo*-Schrittes a befindet sich das System im Zustand t_a, eine nachfolgende Akzeptierung des *Monte-Carlo*-Schritts b verletzt die Markovketteneigenschaft, da die von a induzierten Änderungen am System bei der Evaluation von b nicht berücksichtigt wurde.

In der *Hybrid-Simulation-State-Machine* wird diese Einschränkung dadurch eingehalten, dass pro Simulationsschritt nur eine Änderung vom Anwender akzeptiert werden kann. Erst die Ausführung eines neuen Simulationsschritts erlaubt die Akzeptierung eines weiteren *Monte-Carlo*-Schritts.

Da die Anzahl an *Änderungsinstanzen* variabel gehalten ist, muss darauf geachtet werden, dass nur Änderungen akzeptiert werden können, die sich nicht auf einen früheren Zeitpunkt in der Markovkette beziehen. Werden beispielsweise einige Änderungsinstanzen temporär nicht genutzt, dürfen diese zu einem späteren Zeitpunkt nicht akzeptiert werden.

In der *Hybrid-Simulation-State-Machine* wird dazu für jede *Änderungsinstanz* bei der Übertragung der Ergebnisse die Information gespeichert, zu welchem Simulationsschritt t_s dieses

Ergebnis berechnet wurde. Eine Änderung wird nur dann akzeptiert, wenn der Zustand der Simulation t und dieser zum Ergebnis gespeicherte Zeitpunkt übereinstimmen ($t = t_s$).

6 Beschleunigte Berechnung des elektrostatischen Potentials

Bei der GCMC-Molekularsimulation setzt sich die Gesamtenergie des Systems aus zwei paarweise zwischen allen Molekülen des Systems ausgewerteten Anteilen zusammen. Der erste Anteil an der Gesamtenergie berücksichtigt die Interaktion zwischen Elektronen unterschiedlicher Atome auf kurze Distanz, die sogenannten kurzreichweitigen Potentiale. Diese kurzreichweitigen Potentiale werden meist durch das *Lennard-Jones-Potential* beschrieben [12]. Der zweite Anteil an der Gesamtenergie beruht auf der elektrostatischen Interaktion zwischen Molekülen auf größerer Distanz und wird meist durch das *Coulomb-Potential*, einem langreichweitigen Potential, beschrieben. Während der Einfluss kurzreichweitiger Potentiale bei größeren Abständen schnell abnimmt und dadurch ein Abschneideradius definiert werden kann, ab welchem zwei Atome keine Interaktion mehr aufeinander ausüben, müssen für langreichweitige Potentiale die Interaktionen aller Molekülpaare des Systems betrachtet werden. Darüber hinaus müssen bei periodischen Randbedingungen auch der Einfluss von periodischen Bildern des Simulationsgebiets berücksichtigt werden.

6.1 Berechnung des Coulomb-Potentials

Die paarweise Berechnung des *Coulomb-Potential* ist mit einem signifkanten Rechenaufwand verbunden, besonders wenn dabei periodische Bilder des Simulationsgebiets berücksichtigt werden. Um diese elektrostatischen Interaktionen effizient zu berechnen, kann die Ewald-Summation [3] herangezogen werden. Bei dieser Methode setzt sich die Gesamtenergie aus zwei Teilen zusammen: Dem Real- und dem Fourierraumanteil.

Die Realraumanteil besteht dabei aus der Summe von paarweise ausgewerten Anteilen:

$$U_{real}(r_{ij}) = \frac{1}{4\pi\epsilon_0} \frac{q_i q_j \operatorname{erfc}(\sqrt{\alpha} r_{ij})}{r_{ij}}. \tag{6.1}$$

Für die Berechnung des Fourierraumanteils werden im ersten Schritt C komplexe Koeffizienten c_k berechnet:

$$c_k = \sum_{i \in N} f(i,k) \text{ mit } 1 \leq k \leq C. \tag{6.2}$$

Die komplexwertige Funktion f hängt dabei nur von der Position des Atoms i ab. N beschreibt die Menge aller im System befindlichen Atome. Im zweiten Schritt erfolgt die

Multiplikation dieser Koeffizienten mit einer Konstanten und eine daran anschließende Summation:

$$(6.3)\quad F = \sum_{k=1}^{C} g[k] \cdot |c_k|^2.$$

Die Gesamtenergie durch das Coulomb-Potential setzt sich damit zusammen aus

$$(6.4)\quad E = F + \sum_{i \in N, i \neq j} U_{real}(r_{ij}).$$

Der Vorteil dieser Methdode liegt in einer schnellen Konvergenz und beachtet die Interaktion über mehrere periodische Bilder des Simulationsgebiets. Bei mehreren simultan evaluierten *Monte-Carlo*-Schritten ist das Verhältnis zwischen der Anzahl der von *Monte-Carlo*-Schritten betroffene Partikeln und der gesamten Partikelanzahl des Systems meist klein. Werden für jeden simultan auszuwertenden *Monte-Carlo*-Schritt diese Koeffizienten seperat berechnet, so werden viele Koeffizienten redundant berechnet.

6.2 Beschleunigte Berechnung des Fourierraumanteils für mehrere simultane *Monte-Carlo*-Schritte

Zur Auswertung der M *Monte-Carlo*-Schritte $1 \leq j \leq M$ wird der jeweilige Fourierraumanteil F_j benötigt:

$$(6.5)\quad F_j = \sum_{k=1}^{C} g[k] \cdot |c_{k,j}|^2.$$

Hierbei beschreibt

$$(6.6)\quad c_{k,j} = \sum_{i \in N_j} f(i,k) \text{ mit } 1 \leq k \leq C$$

die Koeffizienten für das durch den *Monte-Carlo*-Schritt j definierte System mit den Atomen N_j.

Sei N_o die Menge aller im System befindlichen Atome ohne Berücksichtigung eines bestimmten *Monte-Carlo*-Schritts. Ein *Monte-Carlo*-Schritt der GCMC-Simulation kann ausgedrückt werden durch das Entfernen von Atomen $A_o \subseteq N_o$ und dem Hinzufügen von Atomen A_n. Die Mengen A_o und A_n können hierbei auch leer sein. Es ergibt sich die entsprechende Zuordnung zu verschiedenen Typen von *Monte-Carlo*-Schritten:

- $A_o = \emptyset$ und $A_n \neq \emptyset$: Hinzufügen eines Partikels
- $A_o \neq \emptyset$ und $A_n = \emptyset$: Entfernen eines Partikels

- $A_o \neq \emptyset$ und $A_n \neq \emptyset$: Ändern der Position eines Partikels

Um mehrere *Monte-Carlo*-Schritte gleichzeitig evaluieren zu können, wird zusätzlich der Index j eingeführt, der die Zuordnung der Mengen $A_{o,j}$ und $A_{n,j}$ zu einem bestimmten *Monte-Carlo*-Schritt j ermöglicht. Damit gilt:

(6.7) $N_j = (N_o \setminus A_{o,j}) \cup A_{n,j}$.

Die Abbildung 6.1 zeigt exemplarisch die Änderung der Systemkonfiguration für einen *Monte-Carlo*-Schritt j durch Entfernen von $A_{o,j}$ und Hinzufügen von $A_{n,j}$.

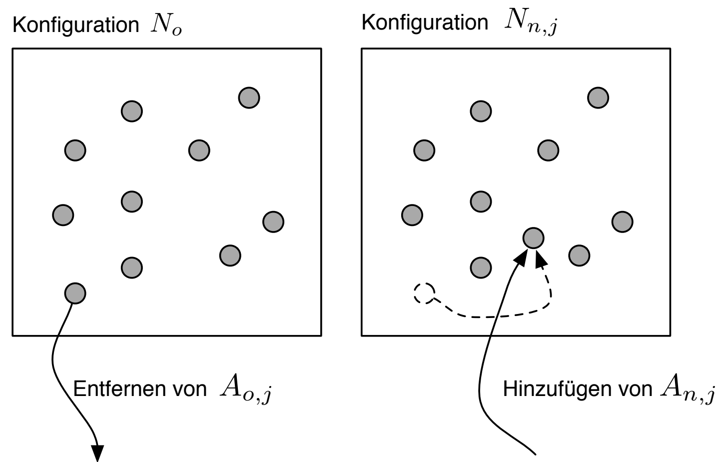


Abbildung 6.1: Darstellung des *Monte-Carlo*-Schritts j durch das Entfernen von $A_{o,j}$ aus der alten Konfiguration N_o und dem Hinzufügen von $A_{n,j}$.

Durch die Einteilung der Atome nach Gleichung 6.7 in drei verschiedene Mengen kann $c_{k,j}$ folgendermaßen ausgedrückt werden:

(6.8) $$c_{k,j} = \sum_{i \in N_j} f(i,k) = \underbrace{\sum_{i \in N_o} f(i,k)}_{C_{k,konst}} \underbrace{- \sum_{i \in A_{o,j}} f(i,k) + \sum_{i \in A_{n,j}} f(i,k)}_{\hat{c}_{k,j}}.$$

Der Term $C_{k,konst}$ ist dabei unabhängig von dem zu evaluierenden *Monte-Carlo*-Schritt und kann somit in einem Vorverarbeitungsschritt berechnet werden. Der dem *Monte-Carlo*-Schritt j zugeordnete Fourierraumanteil ist damit:

(6.9) $$F_j = \sum_{k=1}^{C} g[k] \cdot |C_{k,konst} + \hat{c}_{k,j}|^2.$$

Ist dem Simulationskernel bekannt, welche Teilmenge von Partikeln durch die Menge aller simultan auszuwertender *Monte-Carlo*-Schritte betroffen sind, kann diese Berechnung der Koeffizienten vereinfacht werden:

1. Berechne Koeffizienten $1 \leq k \leq C$ für die Summe aller Partikel des Systems zu $C_{k,konst}$
2. Für alle simultan auszuwertende *Monte-Carlo*-Schritte $1 \leq j \leq M$:
   - Berechne $\hat{c}_{k,j}$
   - Berechne F_j mittels Koeffizienten $C_{k,konst}$ und $\hat{c}_{k,j}$ nach Formel 6.9

Diese Methode zur beschleunigten Berechnung des Fourierraumanteils wurde in dieser Studienarbeit für einen seriellen Simulationskernel, sowie für einen parallelen Simulationskernel implementiert. Eine genaue Beschreibung dieser Simulationskernel findet sich in Kapitel 7, experimentelle Ergebnisse zur Laufzeit werden in Kapitel 8 diskutiert.

6.3 Betrachtung der Zeitkomplexität

Sei C die (konstante) Anzahl an Koeffizienten, N die Anzahl der im System befindlichen Atome und M die Anzahl simultan evaluierter *Monte-Carlo*-Schritte, bestehend aus der Änderung einer konstanten Anzahl an Atomen.

In der unbeschleunigten Version lag die Komplexität der Berechnung in $\mathcal{O}(M \cdot N \cdot C)$, da für jeden der M *Monte-Carlo*-Schritte $N \cdot C$ Koeffizienten berechnet wurden. Durch die vorgestellte Beschleunigung liegt die Komplexität der Berechnung nun in $\mathcal{O}(N \cdot C + M \cdot C)$. Der erste Term $\mathcal{O}(N \cdot C)$ ergibt sich aus der Berechnung von $C_{k,konst}$, hierbei wird der Beitrag aller im System befindlichen Atome zu den C Konstanten berechnet. Für die Berechnung der Koeffizienten der M *Monte-Carlo*-Schritte allerdings müssen nur die von der Änderung betroffenen Atome betrachtet werden, was durch den zweiten Term $\mathcal{O}(M \cdot C)$ beschrieben wird.

7 Simulationskernel

In dieser Studienarbeit wurden folgende vier Simulationskernel implementiert:

1. SeriellNormal: Serieller Simulationskernel ohne beschleunigter Berechnung des Fourierraumanteils
2. SeriellBeschleunigt: Serieller Simulationskernel mit beschleunigter Berechnung des Fourierraumanteils
3. CudaNormal: CUDA Simulationskernel ohne beschleunigter Berechnung des Fourierraumanteils
4. CudaBeschleunigt: CUDA Simulationskernel mit beschleunigter Berechnung des Fourierraumanteils

Jeder dieser vier Simulationskernel ist in der Lage, von einem bestimmten Zustand der Simulation ausgehend mehrere verschiedene *Monte-Carlo*-Schritte (im Folgenden als Mutanten bezeichnet) zu untersuchen und berechnet das *Lennard-Jones-Potential*, sowie das *Coulomb-Potential* unter Einsatz der Ewald-Summation [3]. Diese vier Simulationskernel werden im Folgenden genauer vorgestellt.

7.1 SeriellNormal

Der serielle Simulationskernel besteht aus zwei Teilen. Im ersten Teil werden die paarweisen Energien zwischen zwei Atomen basierend auf deren Abstand berechnet, im zweiten Teil erfolgt eine Berechnung des Fourierraumanteils des elektrostatischen Potentials, welche nur auf der Position der im System befindlichen Atome basiert. Bei der Evaluierung von Mutanten wird nicht die gesamte im System befindliche Energie neu berechnet, sondern jeweils nur die Änderung dieser Energie durch den Mutanten.

Der Algorithmus Serial_Ediff (Algorithmus 7.1) berechnet diese Energiedifferenz für paarweise Interaktionen für jeden Mutanten. Dabei wird ein Mutant analog zu Kapitel 6 die Änderung des Systems durch den Mutanten in die entfernten Atome $A_{o,i}$ (*mutants[midx].remP* im Algorithmus) und hinzugefügten Atome $A_{n,i}$ (*mutants[midx].addP* im Algorithmus) aufgeteilt. Für jedes Atom dieser beiden Menge werden die paarweisen Energien durch das *Lennard-Jones-Potential* und den Realraumanteil des *Coulomb-Potential* mit allen anderen, im System befindlichen Atom durch die Methode Pair_Energy (Algorithmus 7.2) berechnet.

Für die durch einen Mutanten hervorgerufene Änderung der Energie *e_diff* wird die berechnete Energie für die aus dem System entfernten Atome *e_remove* von der Energie der hinzugefügten Atome *e_add* subtrahiert.

Algorithmus 7.1 Serieller Algorithmus zur Berechnung der Energiedifferenz mehrerer Mutanten

```
1:  function SERIAL_EDIFF(mutants, nparts)
2:      for midx ← 1 to mutants.size do
3:          e_add[midx] ← 0
4:          e_rem[midx] ← 0
5:          for pidx ← 1 to nParticles do
6:              if mutants[midx].addP.particleID ≠ pidx then
7:                  e_add[midx] ← e_add[midx] + pair_energy(particles[pidx], mutants[midx].addP)
8:              end if
9:              if mutants[midx.remP.particleID ≠ pidx then
10:                 e_rem[midx] ← e_rem[midx] + pair_energy(particles[pidx], mutants[midx].remP)
11:             end if
12:         end for
13:         e_diff[midx] ← e_add[midx] − e_rem[midx]
14:     end for
15:     return e_diff
16: end function
```

Die Methode PAIR_ENERGY berechnet den Abstand zwischen den gegebenen Atomen, basierend auf diesem Abstand wird das *Lennard-Jones-Potential* unter Berücksichtigung des Abschneideradius und der Realraumanteil des *Coulomb-Potential* für dieses Atompaar berechnet und aufsummiert. Diese Methode ist in Algorithmus 7.2 dargestellt:

Algorithmus 7.2 Berechnung der paarweisen Energie

```
1:  function PAIR_ENERGY(particle_A, particle_B)
2:      e ← 0
3:      for a ← 1 to numAtoms do
4:          for b ← 1 to numAtoms do
5:              dx ← particle_A.atom[a].x − particle_B.atom[b].x
6:              dy ← particleA.atom[a].y − particleB.atom[b].y
7:              dz ← particleA.atom[a].z − particleB.atom[b].z
8:              dx ← min_image(dx)
9:              dy ← min_image(dy)
10:             dz ← min_image(dz)
11:             r ← dx^2 + dy^2 + dz^2
12:             if r > 0 then
13:                 e ← e + φ_LJ(r) + φ_real(r)
14:             end if
15:         end for
16:     end for
17:     return e
18: end function
```

Im Algorithmus SERIAL_FOURIER (Algorithmus 7.3) erfolgt die Berechnung des Fourierraumanteils für jeden Mutanten nach einem seriellen Schema. Dazu werden für jedes Atom deren Beiträge zu den C komplexwertigen Koeffizienten *complex_coeff* berechnet. Anschließend werden diese Koeffizienten mit einer konstanten Tabelle multipliziert und so der zu einem bestimmten Mutanten gehörende Fourierraumanteil durch die Summation dieser Werte berechnet.

Algorithmus 7.3 Algorithmus zur unbeschleunigten Berechnung des Fourierraumanteils

```
function SERIAL_FOURIER(mutants)
    complex_coeff[C]
    for midx ← 1 to mutants.size do
        for pidx ← 1 to nParticles do
            if mutants[midx].remP.particleID ≠ pidx then
                // Beitrag von Partikel pidx zu complex_coeff
                for a ← 1 to nAtoms do
                    (x, y, z) ← particles[pidx].atom[a]
                    for k ← 1 to C do
                        complex_coeff[k] ← complex_coeff[k] + f((x, y, z), k)
                    end for
                end for
            end if
        end for
        // Beitrag des hinzugefügten Partikels
        for a ← 1 to nAtoms do
            (x, y, z) ← mutants[midx].addP.atom[a]
            for k ← 1 to C do
                complex_coeff[k] ← complex_coeff[k] + f((x, y, z), k)
            end for
        end for
        // Multiplikation mit konstanter Tabelle und Summation
        F[midx] ← 0
        for k ← 1 to C do
            F[midx] ← g[k] · abs2(complex_coeff[k])
        end for
    end for
end function
```

Die Methode $abs2(c)$ bezeichnet dabei den quadrierten Wert des Betrags der komplexen Zahl c.

7.2 SERIELLBESCHLEUNIGT

Bei dem seriellen Simulationskernel mit beschleunigter Berechnung des Fourierraumanteils werden die paarweisen Interaktionen analog zu dem unbeschleunigten seriellen Algorithmus berechnet. Dazu wird ebenfalls die Methode SERIAL_EDIFF (Algorithmus 7.1) eingesetzt. Im

Gegensatz zum Simulationskernel SERIELLNORMAL wurde der Algorithmus SERIAL_FOURIER dahingehend verändert, dass redundante Berechnungen der Koeffizienten für alle Mutanten verringert wurden. Im ersten Schritt von SERIAL_FOURIER_MULTIMUT (Algorithmus 7.4) werden dabei die C komplexwertigen Koeffizienten für alle im System befindlichen Atome ohne Betrachtung eines bestimmten Mutanten berechnet und in $complex_coeff_intermediate$ gespeichert. Im Anschluss werden die Auswirkungen auf diese Koeffizienten durch die einen Mutanten $midx$ beschreibenden Mengen von Atomen für die entfernten Atome $A_{o,midx}$ und die hinzugefügten Atome $A_{n,midx}$ berechnet. Die dabei auftretende Änderung für jeden Koeffizienten und jeden Mutanten wird dabei in $complex_coeff_mutants$ gespeichert. Sind alle diese Werte bekannt, wird zu jedem Mutanten der zugehörige Fourierraumanteil $F[midx]$ berechnet.

Algorithmus 7.4 Algorithmus zur beschleunigten Berechnung des Fourierraumanteils

```
function SERIAL_FOURIER_MULTIMUT(mutants)
    complex_coeff_intermediate[C]
    complex_coeff_mutants[C][mutants.size]

    // Berechnung der Koeffizienten für alle Partikel ohne Berücksichtigung von Mutanten
    for pidx ← 1 to nParticles do
        for a ← 1 to nAtoms do
            (x, y, z) ← particles[pidx].atom[a]
            for k ← 1 to C do
                complex_coeff_intermediate[k] ← complex_coeff[k] + f((x, y, z), k)
            end for
        end for
    end for

    // Berechnung der Koeffizienten nur für Mutanten
    for midx ← 1 to mutants.size do
        for a ← 1 to nAtoms do
            for k ← 1 to C do
                (x, y, z) ← mutants[midx].addP.atom[a]
                complex_coeff_mutants[k][midx] ← f((x, y, z), k)
                (x, y, z) ← mutants[midx].remP.atom[a]
                complex_coeff_mutants[k][midx] ← complex_coeff_mutants[k][midx] − f((x, y, z), k)
            end for
        end for
    end for

    // Berechnung des Fourierraumanteils für jeden Mutanten
    for midx ← 1 to mutants.size do
        F[midx] ← 0
        for k ← 1 to C do
            F[midx] ← g[k] · abs2(complex_coeff[k] + complex_coeff_mutants[k][midx])
        end for
    end for
end function
```

7.3 CudaNormal

Dieser Simulationskernel basiert auf den von [1] vorgestellten GPGPU Kerneln zur Berechnung des *Lennard-Jones-Potentials* sowie des *Coulomb-Potentials* zur Evaluierung von mehreren Mutanten auf hybriden Architekturen.
Dabei werden drei GPGPU Kernel eingesetzt:

- K_Pair (Alg. 7.5),
- K_Fourier (Alg. 7.6)
- K_Fourier_Sum (Alg. 7.7)

Der GPGPU Kernel K_Pair (Algorithmus 7.5) berechnet dabei die paarweise Energie zwischen einem durch einen *Monte-Carlo*-Schritt hinzugefügten beziehungsweise entfernten Partikel mit allen anderen im System befindlichen Partikeln für das *Lennard-Jones-Potential* und den Realraumanteil des *Coulomb-Potential*. Für M Mutanten und N im System befindliche Partikel werden hierfür $M \cdot N$ GPGPU Threads gestartet. Jeder GPGPU Thread berechnet dabei ein eindeutiges Paar von Mutant ($o \triangleright n$) und Partikel a des Systems. Bei dieser Notation bezeichnet o das durch den Mutanten aus dem System entfernte Partikel und n das durch den Mutanten dem System hinzugefügte Partikel. Für jeden Mutanten werden im Anschluss an die Berechnung diese Beiträge zur Gesamtenergie durch partielle Summation ($\sum_{tid} e[tid]$) aufaddiert und global gespeichert.

Algorithmus 7.5 Kernel K_Pair zur paarweisen Energieberechnung nach [1]

```
function K_Pair(a, o ▷ n)
    e[tid] = 0
    if a ≠ o then
        dx ← global_load(a.x) − global_load(n.x)
        dy ← global_load(a.y) − global_load(n.y)
        dz ← global_load(a.z) − global_load(n.z)
        dx ← min_image(dx)
        dy ← min_image(dy)
        dz ← min_image(dz)
        r ← dx² + dy² + dz²
        if r > then
            e[tid] ← φ_LJ(r) + φ_real(r)
        end if
    end if

    e ← ∑_tid e[tid]
    if tid == 0 then
        global_write(e)
    end if
end function
```

Die sich dadurch für jeden Block von GPGPU Threads ergebenden Energien werden im Anschluss auf der CPU aufsummiert.

Zur Berechnung des Fourierraumanteils des *Coulomb-Potentials* werden zwei GPGPU Kernel benötigt: K_Fourier und K_Fourier_Sum.

Der GPGPU Kernel K_Fourier wird mit $(N+1) \cdot M$ GPGPU Threads gestartet, wovon jeder GPGPU Thread den Einfluss eines bestimmten Partikels a für einen bestimmten Mutanten $o \triangleright n$ auf jeden der C Koeffizienten berechnet. Der zusätzliche GPGPU Thread mit $a == \emptyset$ wird benutzt, um das Hinzufügen eines Partikels zum System zu simulieren. Der Einfluss jedes Partikels auf einen bestimmten dieser C Koeffizienten wird wieder durch partielle Summation addiert.

Algorithmus 7.6 Kernel K_Fourier zur Berechnung der Koeffizienten für den Fourierraumanteil modifiziert nach [1]

```
function K_Fourier(a, o ▷ n)
    er[tid] ← 0
    ei[tid] ← 0
    skip ← false
    if a == o & o ≠ n then                // nicht berücksichtigen, wenn Partikel entfernt wird
        skip ← true
    end if
    if a == n or a == ∅ then
        x ← global_load(n.x)
        y ← global_load(n.y)
        z ← global_load(n.z)
    else
        x ← global_load(a.x)
        y ← global_load(a.y)
        z ← global_load(a.z)
    end if
    for k ← 1 to C do
        (er[tid], ei[tid]) ← f((x, y, z), k)
        if skip then
            (er[tid], ei[tid]) ← (0, 0)
        end if

        er ← ∑_tid er[tid]
        ei ← ∑_tid ei[tid]
        if tid == 0 then
            global_write(er, ei)
        end if
    end for
end function
```

Im Anschluss wird ein zweiter GPGPU Kernel K_Fourier_Sum gestartet, der diese partiellen Summen addiert und durch Multiplikation der Koeffizienten mit Einträgen aus einer konstanten Tabelle den jeweiligen Anteil an F_j für den Mutanten j berechnet.

Algorithmus 7.7 Kernel K_FOURIER_SUM zur Summation des Fourierraumanteil nach [1]

function K_FOURIER_SUM(k, $psums$)
  $e[tid] \leftarrow 0$
  $er \leftarrow 0$
  $ei \leftarrow 0$
  for $s \leftarrow 1$ to $psums$ **do**
    $(er, ei) \leftarrow (er, ei) + global_load(\mathbf{c}_k[s])$
  end for
  $e[tid] \leftarrow e[tid] + g[k] \cdot (er^2 + ei^2)$
  $e \leftarrow \sum_{tid} e[tid]$
  if $tid == 0$ **then**
    $global_write(e)$
  end if
end function

7.4 CUDABESCHLEUNIGT

Dieser Simulationskernel besteht aus der Weiterentwicklung der drei in Kapitel 7.3 beschriebenen GPGPU Kernel nach der in Kapitel 6 beschriebenen Methode: Der GPGPU Kernel zur Berechnung der paarweisen Interaktion (Alg. 7.5) ist dabei identisch, lediglich für die Berechnung des Fourierteils wird eine Änderung vorgenommen. Der GPGPU Kernel K_FOURIER_NOCHANGE (Alg. 7.8) wird nicht mit $N \cdot M$ GPGPU Threads gestartet, sondern im ersten Schritt benutzt um die Koeffizienten der N Partikel ohne jegliche Betrachtung eines Mutanten zu berechnen.

Algorithmus 7.8 Kernel K_FOURIER_NOCHANGE zur Berechnung der Koeffizienten $C_{k,konst}$ des Fourierraumanteils für alle Partikel des Systems ohne Berücksichtigung eines bestimmten Mutanten

function K_FOURIER_NOCHANGE(a)
  $er[tid] \leftarrow 0$
  $ei[tid] \leftarrow 0$
  $x \leftarrow global_load(a.x)$
  $y \leftarrow global_load(a.y)$
  $z \leftarrow global_load(a.z)$
  for $k \leftarrow 1$ to C **do**
    $(er[tid], ei[tid]) \leftarrow \mathbf{f}((x, y, z), k)$
    $er \leftarrow \sum_{tid} er[tid]$
    $ei \leftarrow \sum_{tid} ei[tid]$
    if $tid == 0$ **then**
      $global_write(er, ei)$
    end if
  end for
end function

Ein leicht modifizierter GPGPU Kernel von K_FOURIER wird im Anschluss verwendet, um für jeden Mutanten die Auswirkungen auf die Koeffizienten zu berechnen: K_FOURIER_CHANGEONLY.

Algorithmus 7.9 Kernel K_FOURIER_CHANGEONLY zur Berechnung der Koeffizienten $\hat{c}_{k,cidx}$ des Fourierraumanteils für die von *Monte-Carlo*-Schritten betroffenen Partikel

```
function K_FOURIER_CHANGEONLY(o ▷ n)
    er[tid] ← 0
    ei[tid] ← 0
    if tid == 0 then
        c ← o
    else if tid == 1 then
        c ← n
    end if
    x ← global_load(c.x)
    y ← global_load(c.y)
    z ← global_load(c.z)
    for k ← 1 to C do
        (er[tid], ei[tid]) ← f((x, y, z), k)
        if tid == 0 then
            er ← −er[0] + er[1]
            ei ← −ei[0] + ei[1]
            global_write(er, ei)
        end if
    end for
end function
```

Im anschließenden GPGPU Kernel K_FOURIER_SUM_NEW werden diese Beiträge $\hat{c}_{k,cidx}$ dann auf die bereits berechneten für alle Mutanten konstanten Koeffizienten $C_{k,konst}$ addiert und analog zu Algorithmus 7.7 mit der Tabelle von Konstanten multipliziert und summiert.

Algorithmus 7.10 Kernel K_FOURIER_SUM_NEW

```
function K_FOURIER_SUM_NEW(k, psums)
    e[tid] ← 0
    er ← 0
    ei ← 0
    for s ← 1 to psums do
        (er, ei) ← (er, ei) + global_load(C_{k,konst}[s])
    end for
    (er, ei) ← (er, ei) + global_load(ĉ_{k,diff}[cidx])
    e[tid] ← e[tid] + g[k] · (er² + ei²)
    e ← Σ_tid e[tid]
    if tid == 0 then
        global_write(e)
    end if
end function
```

8 Experimentelle Ergebnisse

8.1 Referenzsystem

Als Referenzsystem wurde in dieser Studienarbeit eine GCMC-Molekularsimulation verwendet. Bei dem dabei simulierten Molekül handelte es sich um *Simple-Point-Charge* Wasser [13]. Für die Berechnung des Fourierraumanteils des *Coulomb-Potential* wurden 276 Koeffizienten betrachtet.

Die Parameter des Systems sind dabei folgendermaßen variiert worden:

- 1 bis 8 simultane *Monte-Carlo*-Schritte (Mutanten)
- 8 bis 8192 Moleküle im System

8.2 Hardware-Plattform

Die Referenzarchitektur bestand aus:

- CPU: Intel® Core® i7 CPU mit einer Taktfrequenz von 3.1 GHz
- GPU: NVIDIA® Fermi GPGPU mit 336 CUDA® Kernen in 7 Multiprozessoren mit einer Taktfrequenz von 1.60 GHz, CUDA® Version 4.2
- Betriebssystem: Mac OS 10.7.4

8.3 Ergebnisse

Bei der Ermittlung der durchschnittlichen Laufzeiten der Simulationskernel SeriellNormal, SeriellBeschleunigt, CudaNormal und CudaBeschleunigt wurden jeweils 1000 Simulationsschritte für jede betrachtete Anzahl von Molekülen durchgeführt. Von den dabei ermittelten Werten für die Laufzeit wurden die Mittelwerte gebildet. Die Laufzeit eines Simulationskernels bezieht sich in diesen Experimenten auf die Auswertung eines Simulationsschritts auf der jeweiligen Architektur ohne Betrachtung des Kommunikationsoverheads.

8.3.1 Variation der Anzahl von Molekülen

Betrachtung für einen Mutanten

In Abbildung 8.1 sind die durchschnittlichen Laufzeiten der vier unterschiedlichen Simulationskernel für eine variierende Anzahl von Molekülen für einen Mutanten dargestellt.

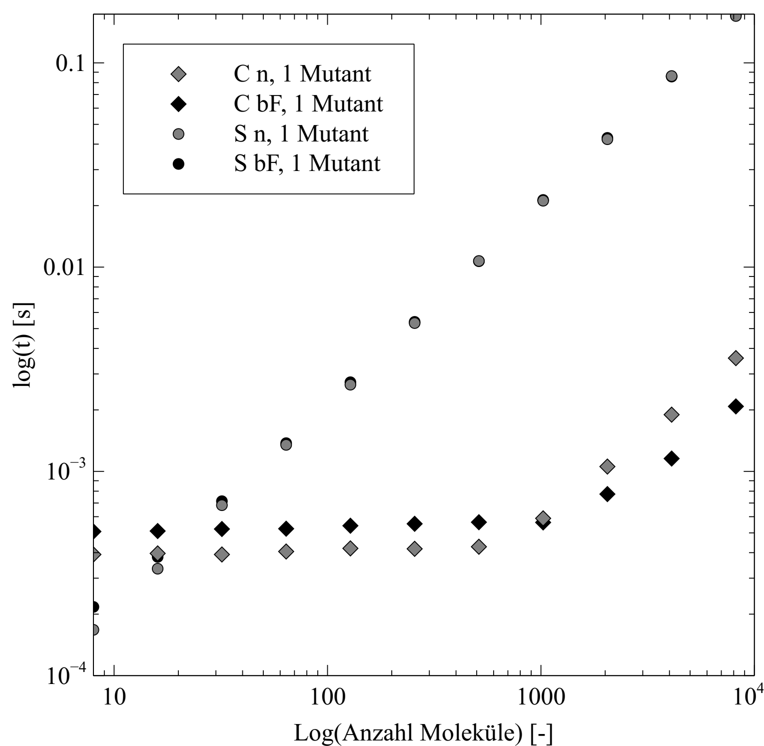


Abbildung 8.1: Durchschnittliche Laufzeit der Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) bei Simulation eines Mutanten.

Die Datenreihen für die Simulationskernel SeriellNormal und SeriellBeschleunigt zeigen über alle Messpunkte hinweg einen nahezu identischen Wert. Aus den Werten aus Tabelle 8.1 lässt sich eine maximale Differenz von circa 1.4 ms bestimmen. Diese Datenreihen weisen dabei in dem betrachteten Bereich ein lineares Verhalten zwischen der Anzahl betrachteter Moleküle und der Laufzeit auf. Die in der Abbildung mit Rautensymbol dargestellten Datenreihen für die Simulationskernel CudaNormal (grau) und CudaBeschleunigt (schwarz) können dahingegen durch eine andere Beziehung beschrieben werden: Bis zu einer Molekülanzahl von 512 Molekülen haben die Werte der beide Datenreihen einen annährend konstanten Wert. Dabei pendelt die Laufzeit von CudaNormal um einen Wert von 0.39 ms und ist damit in diesem Bereich um einen Faktor von circa 1.4x schneller als CudaBeschleunigt, dessen Laufzeit bei circa 0.58 ms liegt. Ab einer Molekülanzahl von 2048 Molekülen benötigt CudaNormal mehr Zeit zur Ausführung als CudaBeschleunigt. Für 2048 Moleküle ist CudaBeschleunigt um einen Faktor 1.34x schneller als CudaNormal. Bei 4096 Molekülen ist dieser Faktor rund 1.63x und bei 8192 Molekülen ergibt sich ein Beschleunigungsfaktor von rund 1.72x.

Für diese Datenreihen sind die auf CUDA basierenden Simulationskernel CudaBeschleunigt und CudaNormal ab einer Molekülanzahl von 32 schneller als die seriellen Simulationskernel SeriellNormal und SeriellBeschleunigt. Für 8192 Moleküle ergibt sich in dieser Reihe ein maximaler Beschleunigungsfaktor von über 81x von CudaBeschleunigt gegenüber SeriellNormal.

In Tabelle 8.1 sind die ermittelten Werte der durchschnittlichen Laufzeit für die vier Simulationskernel für einen Mutanten aufgeführt.

Anzahl Moleküle	S n [ms]	S bF [ms]	C n [ms]	C bF [ms]
8	0.17 ± 0.00	0.22 ± 0.01	0.38 ± 0.08	0.51 ± 0.16
16	0.33 ± 0.01	0.38 ± 0.00	0.38 ± 0.09	0.51 ± 0.17
32	0.68 ± 0.02	0.71 ± 0.01	0.38 ± 0.13	0.52 ± 0.20
64	1.35 ± 0.02	1.37 ± 0.01	0.38 ± 0.12	0.52 ± 0.20
128	2.66 ± 0.07	2.70 ± 0.01	0.38 ± 0.12	0.54 ± 0.22
256	5.32 ± 0.07	5.40 ± 0.08	0.40 ± 0.13	0.55 ± 0.23
512	10.71 ± 0.18	10.67 ± 0.03	0.42 ± 0.10	0.56 ± 0.23
1024	21.14 ± 0.17	21.23 ± 0.20	0.55 ± 0.12	0.56 ± 0.15
2048	42.29 ± 0.29	43.67 ± 0.42	1.03 ± 0.18	0.77 ± 0.30
4096	86.18 ± 1.35	86.13 ± 1.53	1.90 ± 0.05	1.16 ± 0.01
8192	169.60 ± 1.88	169.47 ± 0.42	3.58 ± 0.03	2.08 ± 0.02

Tabelle 8.1: Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle bei Simulation eines Mutanten.

Betrachtung für zwei Mutanten

Die Abbildung 8.2 zeigt ebenfalls die durchschnittlichen Laufzeiten der vier betrachteten Simulationskernel für 8 bis 8192 Moleküle für zwei Mutanten.

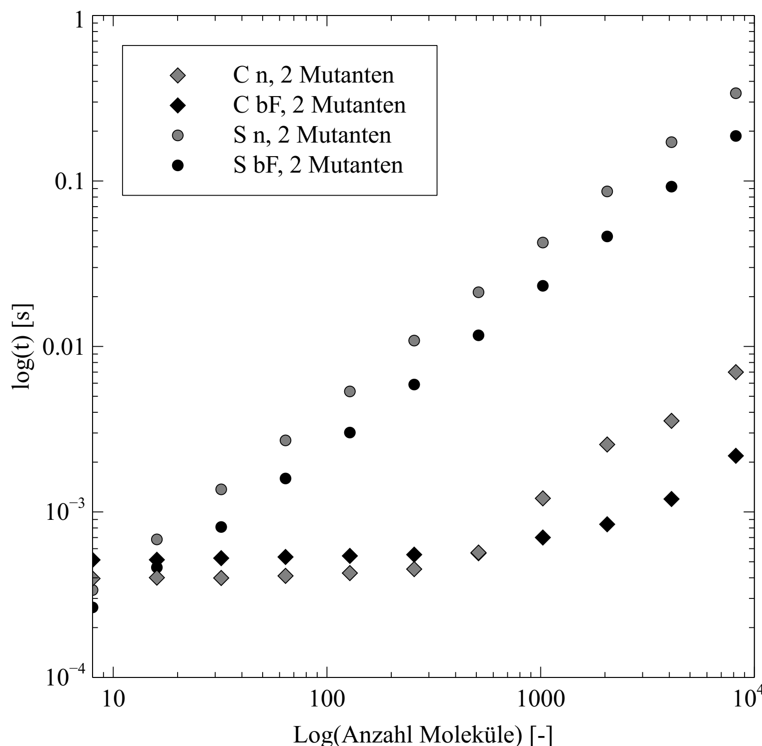


Abbildung 8.2: Durchschnittliche Laufzeit der Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) für zwei Mutanten.

Bei den in der Abbildung 8.2 dargestellten Datenreihen lässt sich, wie schon in 8.1 beschrieben, für die Simulationskernel SeriellNormal und SeriellBeschleunigt ein linearer Zusammenhang zwischen der Anzahl Moleküle und der benötigten Laufzeit für den betrachteten Bereich feststellen. Diese beiden Simulationskernel zeigen im Vergleich zu den Werten für einen Mutanten nicht die gleichen Werte, vielmehr kann ein konstanter Beschleunigungsfaktor von 1.8x zwischen SeriellBeschleunigt und SeriellNormal festgestellt werden. Die Datenreihen der Simulationskernel CudaNormal und CudaBeschleunigt weisen Paralellen zu den Werten der für einen Mutanten betrachteten Vorgänge auf. Auch die Daten dieser Reihe, welche in der Abbildung durch ein Rautensymbol dargestellt sind, zeigen eine nahezu konstante Laufzeit bis zu einem bestimmten Punkt. Während dieser konstanten Phase ist CudaNormal um einen Faktor von 1.35x schneller als CudaBeschleunigt. Der Punkt, nach dem die konstante Laufzeit nicht mehr zu beobachten ist, ergibt sich in dieser Messreihe schon bei einer Molekülanzahl von 512 Molekülen. Im Anschluss zeigt sich für die Datenreihen von CudaNormal ein schnellerer Anstieg der Laufzeiten als für die Datenreihen

von CudaBeschleunigt. Während die Laufzeit von CudaNormal bei Verdopplung der Molekülanzahl für 512 bis 8192 um durchschnittlich 88 % zunimmt, verlängert sich die Laufzeit von CudaBeschleunigt in diesem Bereich bei Verdopplung der Molekülanzahl um 42 %. Für 2048 Moleküle lässt sich damit ein Beschleunigungsfaktor von CudaBeschleunigt gegenüber CudaNormal von 2.24x ermitteln, bei 4096 liegt dieser Faktor bei 2.96x. Zwischen CudaNormal und CudaBeschleunigt beträgt der maximale Beschleunigungsfaktor 3.2x von 6.96 ms für CudaNormal auf 2.18 ms für CudaBeschleunigt, ebenfalls bei 8192 Molekülen. Für den Vergleich der seriellen Simulationskernel, bestehend aus SeriellNormal und SeriellBeschleunigt, mit den CUDA Simulationskerneln CudaNormal und CudaBeschleunigt zeigt sich, dass die Laufzeit ab einer Molekülanzahl von 32 Molekülen die CUDA Simulationskernel deutlich schneller sind. Der maximale Beschleunigungsfaktor von 183x dieser Datenreihen ergibt sich zwischen SeriellNormal und CudaBeschleunigt bei einer Molekülanzahl von 8192 Molekülen.

Im Folgenden werden die ermittelten Werte dieser Datenreihen tabellarisch dargestellt.

Anzahl Moleküle	S n [ms]	S bF [ms]	C n [ms]	C bF [ms]
8	0.34 ± 0.04	0.27 ± 0.00	0.39 ± 0.09	0.51 ± 0.17
16	0.68 ± 0.01	0.45 ± 0.01	0.39 ± 0.13	0.51 ± 0.18
32	1.37 ± 0.04	0.81 ± 0.00	0.38 ± 0.13	0.53 ± 0.21
64	2.71 ± 0.11	1.54 ± 0.01	0.39 ± 0.12	0.53 ± 0.23
128	5.35 ± 0.06	3.02 ± 0.03	0.40 ± 0.15	0.54 ± 0.23
256	10.85 ± 0.18	6.08 ± 0.08	0.41 ± 0.11	0.55 ± 0.22
512	21.26 ± 0.12	11.79 ± 0.10	0.56 ± 0.16	0.56 ± 0.22
1024	42.45 ± 0.25	23.40 ± 0.10	1.05 ± 0.22	0.70 ± 0.38
2048	86.47 ± 1.34	46.24 ± 0.10	1.88 ± 0.06	0.84 ± 0.34
4096	171.64 ± 2.67	93.29 ± 0.40	3.55 ± 0.07	1.20 ± 0.04
8192	339.08 ± 2.93	188.32 ± 3.43	6.96 ± 0.13	2.18 ± 0.17

Tabelle 8.2: Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle bei der Simulation von zwei Mutanten.

Betrachtung für vier Mutanten:

Die in Abbildung 8.3 dargestellten Datenreihen beziehen sich auf die vier betrachteten Simulationskernel. Dabei werden im Folgenden vier Mutanten betrachtet.

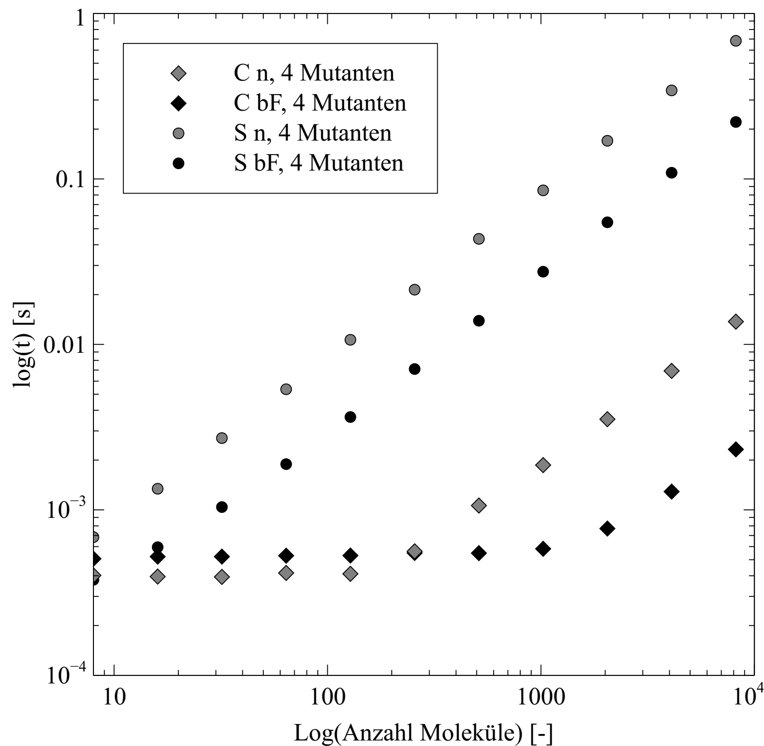


Abbildung 8.3: Durchschnittliche Laufzeit zur Evaluation eines *Monte-Carlo*-Schritts für die Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) für vier Mutanten.

Wie schon zuvor bei den Datenreihen für einen und zwei Mutanten, kann auch bei dieser Datenreihen ein linearer Verlauf der Datenreihen von SeriellNormal und SeriellBeschleunigt im betrachteten Bereich beobachtet werden. Der Simulationskernel SeriellBeschleunigt ist konstant um einen Faktor von circa 3.0x schneller ist als SeriellNormal. Die Simulationskernel CudaNormal und CudaBeschleunigt zeigen zu Beginn der Reihe wieder nahezu konstante Laufzeit. Mit einer Laufzeit von etwa 0.41 ms ist CudaNormal um einen Faktor 1.28x schneller als CudaBeschleunigt mit 0.52 ms in diesem Bereich konstanter Laufzeit. Der Umschaltpunkt, an dem CudaBeschleunigt kürzere Laufzeiten aufweist als CudaNormal, wird in dieser Datenreihe bereits bei 256 Molekülen erreicht. Im Vergleich zu den zwei vorherigen Messreihen fällt dabei auf, dass dieser Umschaltpunkt bei einer steigenden Anzahl Mutanten früher erreicht wird. Im Anschluss nimmt in dieser Datenreihe der Beschleunigungsfaktor von CudaBeschleunigt gegenüber CudaNormal zu. So beträgt

der Beschleunigungsfaktor bei 2048 Molekülen 4.58x, bei 4096 Molekülen 5.35x und bei 8192 Molekülen 5.91x.

Der serielle Simulationskernel SeriellBeschleunigt ist in diesen Datenreihen nur für acht Partikel schneller als die CUDA Simulationskernel. Für alle anderen Messpunkte konnte kein serieller Simulationskernel die Laufzeiten der Simulationskernelauf hybriden Architekturen erreichen. Für eine Molekülanzahl von 8192 ergibt sich der maximale Beschleunigungsfaktor dieser Datenreihen zwischen CudaBeschleunigt und SeriellNormal mit einem Wert von 294.5x.

In Tabelle 8.3 werden die ermittelten Laufzeiten dargestellt.

Anzahl Moleküle	S n [ms]	S bF [ms]	C n [ms]	C bF [ms]
8	0.68 ± 0.10	0.38 ± 0.01	0.40 ± 0.14	0.51 ± 0.06
16	1.34 ± 0.03	0.60 ± 0.00	0.40 ± 0.12	0.52 ± 0.12
32	2.71 ± 0.06	1.05 ± 0.01	0.39 ± 0.15	0.52 ± 0.12
64	5.36 ± 0.12	1.95 ± 0.03	0.42 ± 0.14	0.53 ± 0.14
128	10.66 ± 0.12	3.72 ± 0.02	0.41 ± 0.11	0.53 ± 0.13
256	21.41 ± 0.29	7.32 ± 0.10	0.56 ± 0.17	0.55 ± 0.15
512	43.43 ± 0.65	14.52 ± 0.51	1.06 ± 0.26	0.55 ± 0.09
1024	85.28 ± 1.01	28.42 ± 0.29	1.86 ± 0.02	0.58 ± 0.01
2048	169.98 ± 1.31	47.02 ± 0.95	3.53 ± 0.03	0.77 ± 0.21
4096	342.71 ± 5.22	112.92 ± 0.84	6.92 ± 0.05	1.29 ± 0.09
8192	683.19 ± 9.87	227.24 ± 3.30	13.73 ± 0.15	2.32 ± 0.02

Tabelle 8.3: Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle und vier Mutanten.

Betrachtung für acht Mutanten

Die Abbildung 8.4 zeigt die Laufzeiten der vier betrachteten Simulationskernel für acht Mutanten.

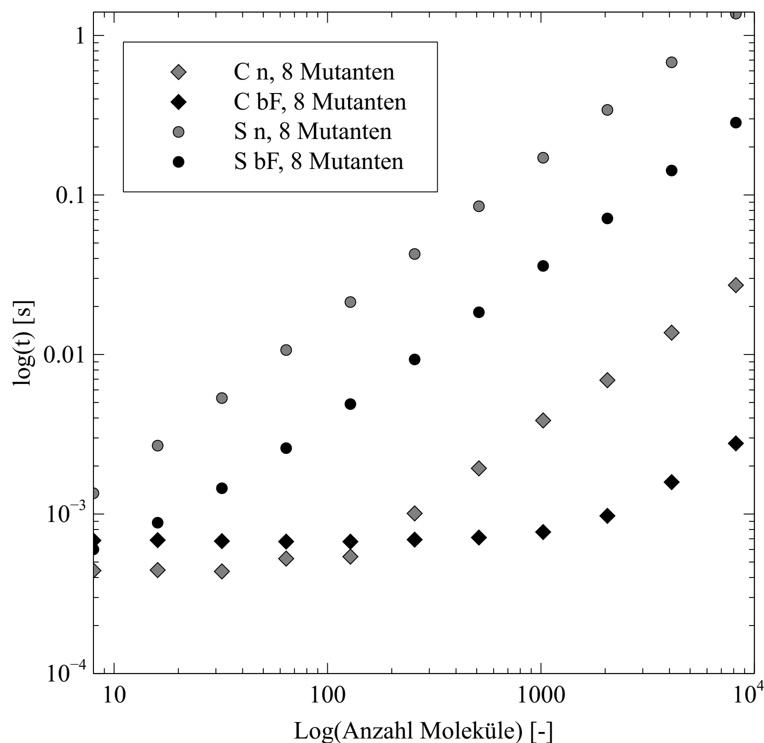


Abbildung 8.4: Durchschnittliche Laufzeit zur Evaluation eines *Monte-Carlo*-Schritts für die Simulationskernel CudaNormal (C n), SeriellNormal (S n), CudaBeschleunigt (C bF) und SeriellBeschleunigt (S bF) für acht Mutanten.

Wie schon in den vorangegangenen Messreihen für weniger als acht Mutanten zeigen auch in dieser Messreihe die seriellen Simulationskernel SeriellNormal und SeriellBeschleunigt im betrachteten Bereich einen linearen Verlauf. SeriellBeschleunigt ist um einen konstanten Faktor von circa 4.8x schneller als SeriellNormal. Beim Vergleich der Laufzeiten dieser seriellen Simulationskernel mit den CUDA Simulationskerneln CudaNormal und CudaBeschleunigt kann beobachtet werden, dass keiner dieser seriellen Simulationskernel eine kürzere Laufzeit als beide CUDA Simulationskernel zeigt. Der Punkt, an dem CudaBeschleunigt schneller ist als CudaNormal, wird anhand der Datenreihen zwischen 128 und 256 Molekülen erwartet. Auch bei dieser Datenreihe nimmt der Faktor zwischen den Laufzeiten von CudaNormal und CudaBeschleunigt ab diesem Punkt zu, ein maximaler Beschleunigungsfaktor ergibt sich für 8192 Moleküle von 9.8x.

Der maximale Beschleunigungsfaktor dieser Datenreihe findet sich bei 8192 Molekülen zwischen SeriellNormal und CudaBeschleunigt mit 494.6x.

In Tabelle 8.4 werden die Resultate dieser Messreihe aufgeführt.

Anzahl Moleküle	S n [ms]	S bF [ms]	C n [ms]	C bF [ms]
8	1.35 ± 0.07	0.62 ± 0.01	0.44 ± 0.12	0.68 ± 0.15
16	2.68 ± 0.06	0.88 ± 0.01	0.45 ± 0.14	0.69 ± 0.15
32	5.33 ± 0.08	1.45 ± 0.01	0.44 ± 0.14	0.68 ± 0.12
64	10.66 ± 0.14	2.59 ± 0.05	0.53 ± 0.10	0.67 ± 0.10
128	21.32 ± 0.24	4.84 ± 0.01	0.56 ± 0.15	0.67 ± 0.08
256	42.65 ± 0.50	9.32 ± 0.09	1.00 ± 0.12	0.69 ± 0.11
512	84.92 ± 0.70	18.20 ± 0.06	1.88 ± 0.16	0.71 ± 0.08
1024	171.11 ± 2.75	36.02 ± 0.43	3.55 ± 0.18	0.77 ± 0.04
2048	341.12 ± 5.38	71.17 ± 0.08	6.95 ± 0.18	0.97 ± 0.00
4096	678.24 ± 6.64	143.18 ± 1.95	13.77 ± 0.33	1.59 ± 0.01
8192	1370.20 ± 29.49	284.10 ± 2.87	27.23 ± 0.29	2.77 ± 0.03

Tabelle 8.4: Durchschnittliche Laufzeiten der Simulationskernel SeriellNormal (S n), SeriellBeschleunigt (S bF), CudaNormal (C n), CudaBeschleunigt (C bF) für unterschiedliche Anzahl Moleküle und acht Mutanten.

8.3.2 Variation der Anzahl von Mutanten

Die in diesem Kapitel dargestellten Messwerte beziehen sich auf die im vorhergehenden Kapitel beschriebenen Messreihen.

Beschleunigung von SeriellBeschleunigt gegenüber SeriellNormal

In der folgenden Tabelle sind die über alle Moleküle konstanten Faktoren in der Laufzeit zwischen SeriellNormal und SeriellBeschleunigt dargestellt:

# Mutanten	1	2	4	8
Beschleunigungsfaktor	1x	1.8x	3.0x	4.8x

Bei der Simulation für einen Mutanten zeigten beide Simulationskernel die gleiche Laufzeit, mit zunehmender Anzahl von Mutanten nimmt dieser Beschleunigungsfaktor zu. Bei einer Anzahl von 8 Mutanten beträgt der Beschleunigungsfaktor von SeriellBeschleunigt gegenüber SeriellNormal 4.8x.

Maximale Beschleunigung von CudaBeschleunigt gegenüber CudaNormal

In der folgenden Tabelle sind die maximalen Beschleunigungsfaktoren CudaNormal und CudaBeschleunigt bei 8192 Molekülen dargestellt für unterschiedliche Mutantenanzahlen dargestellt:

# Mutanten	1	2	4	8
Speedup	1.7x	3.2x	5.91x	9.8x

Auch hier zeigt eine Zunahme der betrachteten Anzahl von Mutanten eine Zunahme des Beschleunigungsfaktors von CudaBeschleunigt gegenüber CudaNormal. Bei den betrachteten Messreihen wurde ein maximaler Beschleunigungsfaktor von 9.8x bei einer Anzahl von acht Mutanten und einer Molekülanzahl von 8192 Molekülen festgestellt.

Maximaler Bescheunigungsfaktor über alle Datenreihen

In der nachfolgenden Tabelle werden die ermittelten maximalen Beschleunigungsfaktoren zwischen den Simulationskerneln SeriellNormal und CudaBeschleunigt bei 8192 Molekülen zusammengefasst, welche jeweils die größten Beschleunigungsfaktoren bei einer festen Anzahl von Molekülen zeigten:

# Mutanten	1	2	4	8
Beschleunigungsfaktor	81x	183x	294.5x	494.6x

Auch hierbei zeigt sich eine Zunahme des Beschleunigungsfaktors zwischen SeriellNormal und CudaBeschleunigt mit steigender Anzahl Mutanten.

Werte zu den Umschaltpunkten für SeriellNormal und CudaNormal

Der Umschaltpunkt, bei dem eine Simulation auf hybriden Architekturen CudaNormal kürzere Laufzeiten aufweist als die seriellen Implementierung SeriellNormal, liegt bei Simulation von einem Mutanten bei 32 Molekülen (s. Tabelle 8.1). Bei den Messreihen für zwei Mutanten ergab sich dieser Umschaltpunkt bereits bei 16 Molekülen (s. Tabelle 8.2), für vier und acht Mutanten wies CudaNormal für jede betrachtete Anzahl von Molekülen eine kürzere Laufzeit als SeriellNormal auf.

Werte zu den Umschaltpunkten für SeriellBeschleunigt und CudaBeschleunigt

Für die Simulationskernel SeriellBeschleunigt und CudaBeschleunigt lag der Umschaltpunkt für einen Mutanten, an dem CudaBeschleunigt kürzere Laufzeiten aufwies als CudaNormal wie bei der Betrachtung der Simulationskernel SeriellNormal und CudaNormal bei 32 Molekülen (s. Tabelle 8.1). Für zwei Mutanten ergab sich der Umschaltpunkt analog zu der Betrachtung von SeriellNormal und CudaNormal ebenfalls bei 16 Molekülen (s. Tabelle 8.2). Bei der Betrachtung einer Anzahl von vier und acht Mutanten lag dieser Umschaltpunkt ebenfalls bei 8 Molekülen (s. Tabelle 8.3 und 8.4).

Dieser Umschaltpunkt liegt demnach für serielle Simulationskernel im Vergleich zu den Simulationskerneln auf hybriden Architekturen bereits bei sehr einer sehr kleinen Anzahl von Molekülen.

Werte zu den Umschaltpunkten für CudaBeschleunigt und CudaNormal

In Abbildung 8.5 sind die Laufzeiten von CudaNormal und CudaBeschleunigt für 128, 256 und 512 Moleküle dargestellt. Hierbei zeigen alle drei Datenreihen von CudaBeschleunigt für 128, 256 und 512 Moleküle für einen, zwei und vier Mutanten annähernd konstante Werte. Bei allen drei Datenreihen ist für acht Mutanten ein Anstieg der Laufzeit in der selben Größenordnung zu erkennen. Die Datenreihen von CudaNormal hingegen weisen mit zunehmender Anzahl Moleküle einen starken Anstieg in der Laufzeit für steigende Anzahl Mutanten auf. Für 128 Moleküle zeigt CudaNormal dabei eine kürzere durchschnittliche Laufzeit als CudaBeschleunigt für alle betrachteten Anzahl Mutanten. Bei 256 Moleküle ist CudaBeschleunigt ab einer Anzahl von 4 Mutanten schneller als CudaNormal, für 512 Moleküle liegt dieser Punkt bei 2 Mutanten.

In der folgenden Tabelle sind die ermittelten Umschaltpunkte abgetragen, ab denen CudaBeschleunigt eine kürzere Laufzeit aufweist als CudaNormal:

# Mutanten	1	2	4	8
# Moleküle	1024	512	256	128-256

Mit zunehmender Anzahl von Mutanten sinkt dabei die Molekülanzahl, ab der CudaBeschleunigt kürzere durchschnittliche Laufzeiten aufweist als CudaNormal.

CudaNormal und CudaBeschleunigt 4096 und 8192 Molekülen

Das Verhalten von CudaNormal und CudaBeschleunigt für große Molekülanzahlen von 4096 und 8192 Molekülen ist in Abbildung 8.6 dargestellt. Hierbei zeigt CudaNormal im Vergleich zu CudaBeschleunigt einen stärkeren Anstieg der Laufzeit mit zunehmender Anzahl Mutanten.

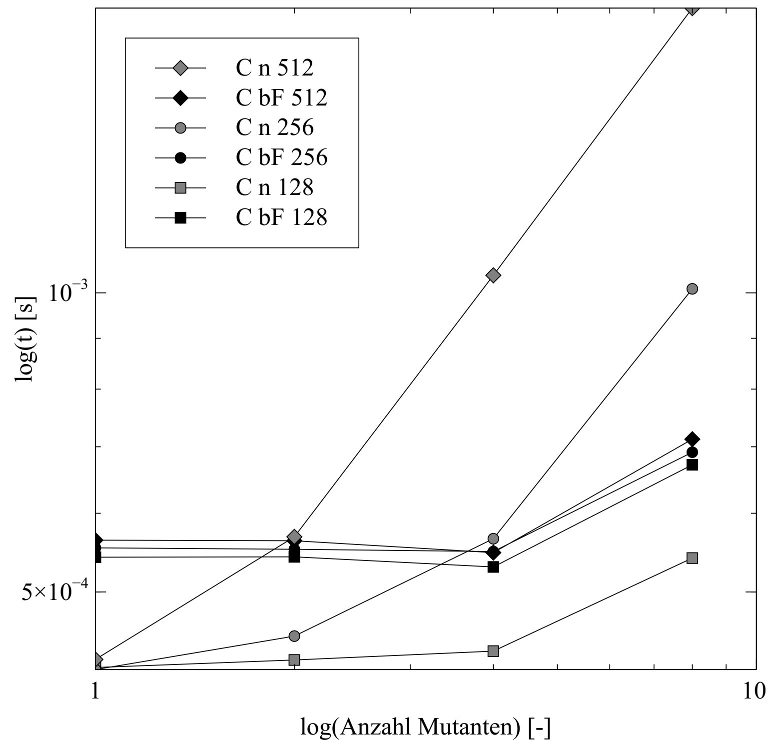


Abbildung 8.5: Laufzeiten von CudaNormal und CudaBeschleunigt für 128, 256 und 512 Moleküle über die Anzahl der Mutanten

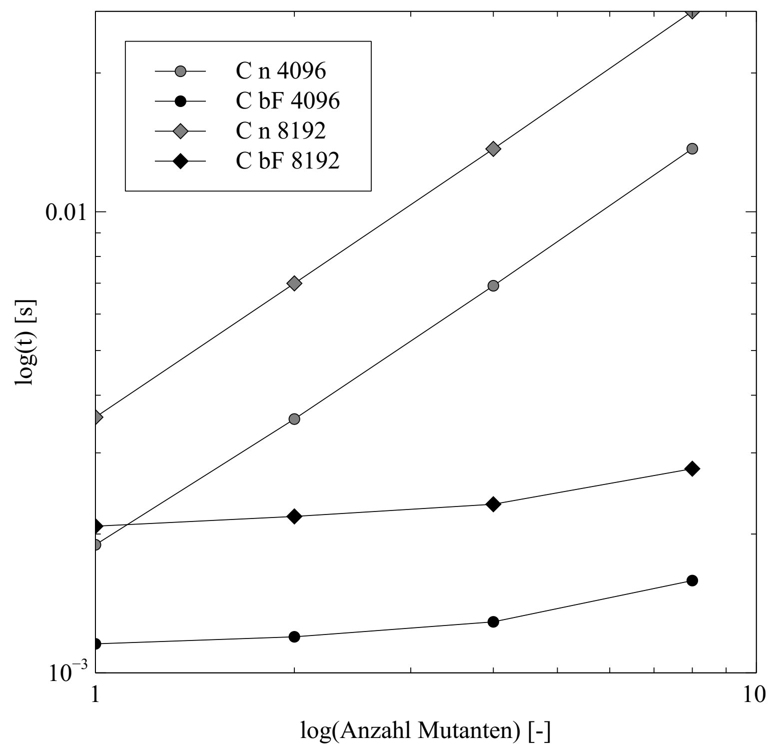


Abbildung 8.6: Laufzeiten von CudaNormal und CudaBeschleunigt für 4096 und 8192 Moleküle über die Anzahl der Mutanten

8.3.3 Automatisierte Wahl von Simulationskerneln

Für die Experimente zur automatisierten Wahl von Simulationskerneln wurden zur gleichen Zeit mehrere unabhängige Instanzen der Simulation gestartet, wobei für jede dieser Simulationsinstanzen die automatische Wahl von Simulationskerneln aktiviert wurde. Die Simulationsparameter wurden dabei für alle Simulationsinstanzen gleich gewählt. Um eine insgesamt längere Simulationsphase zu betrachten, wurden für jede Instanz der Simulation 100 000 Simulationsschritte simuliert. Bei den für die automatische Wahl von Simulationskerneln eingesetzten Simulationskernel handelt es sich um CudaNormal und SeriellBeschleunigt, da bei der automatisierten Wahl von Simulationskerneln unterschiedliche Architekturen betrachtet werden sollten, um das dynamische Verhalten bei wechselnder Last auf diesen Architekturen beobachten zu können. Die Simulationsparameter wurden dabei so gewählt, dass die Laufzeiten von CudaNormal und SeriellBeschleunigt etwa in der gleichen Größenordnung liegen. Aus den in Kapitel 8.3.1 beschriebenen Ergebnissen ergibt sich, dass dies besonders für eine kleine Molekül- und Mutantenanzahl der Fall ist. Für die Experimente zur automatisierten Wahl von Simulationskerneln wurden Simulationen mit 64, sowie 32 Molekülen mit einem Mutanten eingesetzt. Zum Vergleich der Ergebnisse der Simulationsinstanzen mit automatischer Wahl von Simulationskerneln wurden ebenfalls noch Simulationen gestartet, die mit einem festgesetzten Simulationskernel ausgeführt wurden.

64 Moleküle In der ersten experimentellen Reihe wurde ein System mit 64 Molekülen und einem Mutanten betrachtet. Die Tabelle 8.5 stellt die Laufzeiten der unterschiedlichen Simulationsinstanzen Sim A bis Sim F für eine bestimmte Anzahl nebenläufiger Simulationsinstanzen (Anz. nebenl. SI) für die Simulation unter Einsatz des Simulationskernels CudaNormal dar. Diese Messreihe wird im Nachfolgenden mit *Cn* bezeichnet. Dabei wurde die Anzahl nebenläufiger Simulationsinstanzen im Bereich von einer einzigen Instanz bis zu einer Anzahl von sechs nebenläufigen Simulationsinstanzen variiert.

Die Laufzeiten nehmen dabei für eine größer werdende Anzahl nebenläufiger Simulation zu. Die Betrachtung dieser Laufzeiten ergibt, dass für jede zusätzliche Simulationsinstanz in der betrachteten Messreihe die Laufzeit der einzelnen Simulationen um circa 40 s anstieg. Die Laufzeiten der einzelnen Simulationsinstanzen für eine bestimmte Anzahl an nebenläufigen Simulationsinstanzen variiert dabei in einem Bereich von weniger als drei Sekunden.

Anz. nebenl. SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F[s]
1	38.16					
2	82.13	82.12				
3	122.55	122.83	122.25			
4	162.78	163.97	163.94	163.45		
5	204.89	205.3	205.49	205.23	203.47	
6	245.49	245.79	246.04	246.02	245.64	243.92

Tabelle 8.5: *Cn*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit fest gewähltem Simulationskernel CudaNormal bei 64 Molekülen.

Die folgende Tabelle zeigt die Laufzeiten einer einzelnen Simulation mit Simulationskernel CudaNormal bei einer Molekülanzahl, die der Summe der Anzahl Moleküle der in der nebenläufigen Simulation entspricht.

Anz. Moleküle	128	196	256	320	384
Laufzeit [s]	38.50	39.25	39.34	40.96	40.80

Diese Divergenz entsteht durch eine Schwäche der eingesetzten Architektur: Es wird maximal einem nebenläufigen CPU-Thread Zugriff auf die GPGPU gewährt.

Die nachfolgende Tabelle 8.6 zeigt die gemessenen Laufzeiten für bis zu sechs nebenläufige Simulationen für SeriellBeschleunigt (*SbF*). Hierbei nehmen die Laufzeiten ebenfalls mit steigender Anzahl nebenläufiger Simulationen zu. In dem Bereich von einer einzelnen Simulation bis zu einer Anzahl von vier nebenläufiger Simulationen, welche zur Anzahl Kerne der CPU Referenzarchitektur korrespondiert, nehmen die Laufzeiten nur geringfügig zu. Während für eine Simulationsinstanz Sim A die Laufzeit 141.10 s beträgt, können drei nebenläufige Simulationsinstanzen Sim A, Sim B und Sim C in etwa derselben Zeit von circa 143 s abgearbeitet werden. Für vier und fünf nebenläufige Simulationsinstanzen nimmt die Laufzeit der einzelnen Simulationsinstanzen zu, für vier nebenläufige Instanzen beträgt die Laufzeit rund 148.5 s und für fünf nebenläufige Instanzen steigt die Laufzeit für jede Simulationsinstanz um über 22 s auf circa 171 s. Beim Einsatz von sechs nebenläufigen Instanzen steigt die Laufzeit im Vergleich zu fünf Simulationsinstanzen um weitere 24 s auf circa 195 s. Im Vergleich zu den in Tabelle 8.5 dargestellten Ergebnissen für eine nebenläufige Ausführung mehrerer Simulationinstanzen für die Simulation mit einem festen Simulationskernel CudaNormal kann festgestellt werden, dass die Messreihe *SbF* für bis zu drei nebenläufige Simulationsinstanzen insgesamt schnellere Laufzeiten aufweist als diese zu CudaNormal gehörende Messreihe *Cn*. Bei der Simulation einer einzelnen Instanz ist CudaNormal mit einer Laufzeit von 38.16 s um einen Faktor von rund 3.7x schneller als SeriellBeschleunigt mit einer Laufzeit von 141.1 s. Bei zwei nebenläufige Simulationsinstanzen beträgt dieser Faktor circa 1.73x (82.13 s für CudaNormal und 142.26 s für SeriellBeschleunigt), bei drei nebenläufigen Simulationsinstanzen nur noch rund 1.17x (122.83 s für CudaNormal und 143.13 s für SeriellBeschleunigt). Ab einer Anzahl von

vier nebenläufigen Simulationsinstanzen zeigt SeriellBeschleunigt eine insgesamt kürzere Simulationsdauer als CudaNormal. Während SeriellBeschleunigt bei vier nebenläufigen Simulationsinstanzen maximal 148.58 s zur Simulation benötigt, dauert die Simulation von CudaNormal maximal 163.94 s. Damit ist SeriellBeschleunigt um einen Faktor 1.1x schneller als SeriellNormal. Für fünf Simulationsinstanzen beträgt dieser Faktor 1.19x (205.49 s für CudaNormal und 172.82 s für SeriellBeschleunigt), für sechs nebenläufige Simulationsinstanzen beträgt dieser Faktor 1.25x.

Anz. nebenl. SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F[s]
1	141.10					
2	142.26	141.22				
3	142.82	142.81	143.13			
4	148.46	148.52	148.58	148.37		
5	171.48	170.82	172.59	171.15	172.25	
6	194.27	194.40	194.43	194.24	195.10	195.53

Tabelle 8.6: *SbF*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit festem Simulationskernel SeriellBeschleunigt für 64 Moleküle.

In den folgenden Tabellen 8.7 und 8.8 sind die Laufzeiten von bis zu sechs nebenläufigen Simulationsinstanzen mit einer automatisierten Wahl von Simulationskerneln für zwei Messreihen *AS 1* und *AS 2* dargestellt. Für *AS 1* und *AS 2* kann hierbei eine steigende Laufzeit mit der Zunahme der Anzahl von nebenläufigen Simulationsinstanzen beobachtet werden. Bis zu einer Anzahl von drei nebenläufigen Simulationsinstanzen erfolgte kein Wechsel zwischen den zur automatischen Wahl von Simulationskerneln aktivierten Simulationskerneln CudaNormal und SeriellBeschleunigt, ab einer Anzahl von vier nebenläufigen Simulationsinstanzen ließen sich Wechsel in den Simulationskerneln sowohl bei *AS 1*, als auch bei *AS 2* feststellen.

Anz. nebenl. SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F [s]
1	38.30					
2	75.17	77.78				
3	83.99	103.82	103.56			
4	118.72	127.44	119.50	124.41		
5	149.62	152.66	149.20	149.49	147.81	
6	168.48	167.57	169.70	168.93	169.93	169.65

Tabelle 8.7: *AS 1*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal bei 64 Molekülen.

Anz. nebenl. SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F [s]
1	37.77					
2	76.80	78.78				
3	94.99	98.07	96.60			
4	141.35	142.25	139.03	141.68		
5	147.55	150.05	151.54	149.66	150.41	
6	167.57	168.26	166.22	168.41	165.48	169.10

Tabelle 8.8: *AS 2*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal bei 64 Molekülen.

Im Vergleich zu den Ergebnissen der Simulationen mit einer festen Verwendung der Simulationskernel SeriellBeschleunigt (*SbF*, Tabelle 8.6) und CudaNormal (*Cn*, Tabelle 8.5) kann betrachtet werden, dass für beide Messreihen zu *AS 1* und *AS 2* ab einer Anzahl von zwei nebenläufigen Simulationsinstanzen die Laufzeit der Simulation mit automatischer Wahl des Simulationskernels sowohl schneller als die Simulation mit CudaNormal, als auch schneller als die Simulation mit SeriellBeschleunigt mit entsprechender Anzahl nebenläufiger Simulationsinstanzen war.

Die Abbildung 8.7 zeigt die Darstellung der Laufzeit der in der Laufzeit maximalen Simulationsinstanz abgetragen über die Anzahl der nebenläufigen Simulationsinstanzen. Die dazugehörigen numerischen Werte sind in Tabelle 8.9 aufgeführt.

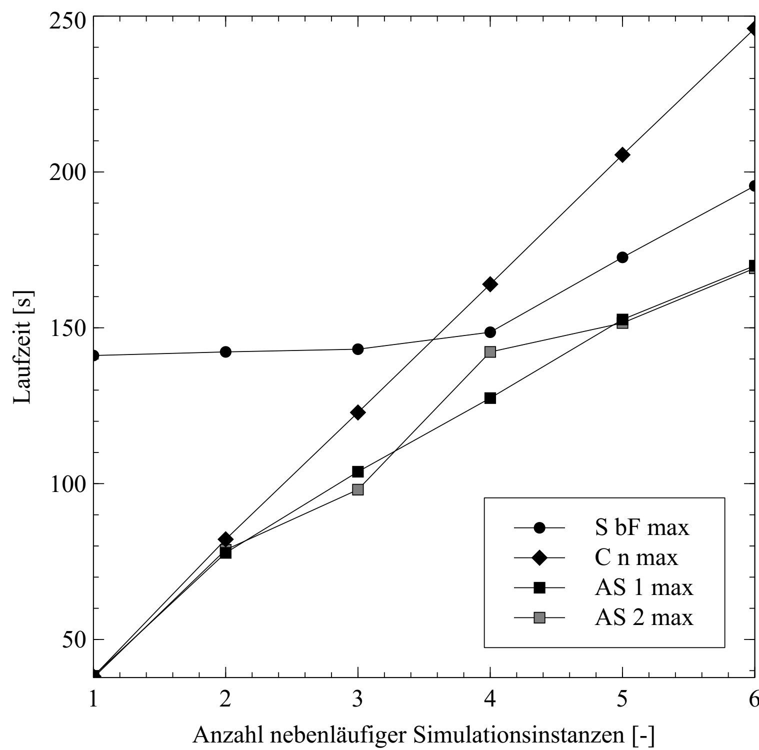


Abbildung 8.7: Maximale Laufzeit der Simulationsinstanzen für unterschiedliche Anzahl nebenläufiger Simulationsinstanzen bei 64 Molekülen. S bF max beschreibt die maximale Laufzeit unter Einsatz von SeriellBeschleunigt, C n max die maximale Laufzeit unter Einsatz von CudaNormal. AS 1 und AS 2 bezeichnen die beiden Messreihen mit automatischem Scheduling von Simulationskerneln.

Anz. nebenl. SI	C n max	S bF max	AS 1 max	AS 2 max
1	38.16	141.10	38.30	37.77
2	82.13	142.26	77.78	78.78
3	122.83	143.13	103.82	98.07
4	163.97	148.58	127.44	142.25
5	205.49	172.59	152.66	151.54
6	246.04	195.53	169.93	169.10

Tabelle 8.9: Maximale Laufzeiten

32 Moleküle Eine weitere Untersuchung bezüglich der automatisierten Wahl von Simulationskerneln wurde für 32 Moleküle durchgeführt. Dazu wurden ebenfalls bis zu sechs nebenläufige Simulationsinstanzen gestartet und die Laufzeit jeder Instanz Sim A bis Sim

F gemessen. Auch in dieser Konfiguration wurden zum Vergleich bis zu sechs Simulationsinstanzen einer Simulation ohne automatische Wahl des Simulationskernels jeweils mit dem Simulationskernel SeriellBeschleunigt (Datenreihe *SbF*) und dem Simulationskernel CudaNormal (Datenreihe *Cn*) betrachtet. Die entsprechenden Laufzeiten der Simulationsinstanzen Sim A bis Sim F finden sich in Tabelle 8.11 für den Simulationskernel SeriellBeschleunigt und in Tabelle 8.10 für den Simulationskernel CudaNormal. Zur Betrachtung der Laufzeiten einer Simulation von mehreren nebenläufigen Simulationsinstanzen mit automatischer Wahl des Simulationskernel zwischen SeriellBeschleunigt und CudaNormal wurden hier ebenfalls zwei Messreihen *AS 1* (Tabelle 8.12) und *AS 2* (Tabelle 8.13) druchgeführt.

Die in Tabelle 8.10 für jede Simulationsinstanz dargestellten Laufzeiten für eine unterschiedliche Anzahl von nebenläufigen Simulationinstanzen mit einem festem Simulationskernel CudaNormal für 32 Moleküle zeigt Ähnlichkeiten zu den Ergebnissen der Simulation mit 64 Molekülen. Dabei kann ebenfalls eine Zunahme der Laufzeit für jede zusätzliche nebenläufige Simulationsinstanz von etwa 40 s festgestellt werden.

Anz. nebenl.SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F [s]
1	37.18					
2	80.35	80.35				
3	119.96	120.25	119.68			
4	159.63	160.41	160.39	160.06		
5	200.21	200.61	200.59	200.77	199.33	
6	238.90	240.55	240.97	240.97	240.74	240.35

Tabelle 8.10: *Cn*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit festem Simulationskernel CudaNormal bei 32 Molekülen.

Die in der folgenden Tabelle 8.11 dargestellten Ergebnisse für die Simulation mit festem Simulationskernel SeriellBeschleunigt zeigen ebenfalls einen Anstieg der Laufzeiten mit steigender Anzahl nebenläufiger Simulationsinstanzen. Wie in den zuvor vorgestellten Ergebnissen zur Simulation von 64 Molekülen nimmt auch hier die Laufzeit der Simulationsinstanzen bis zu einer Anzahl von drei nebenläufigen Simulationsinstanzen nur geringfügig zu, ab einer Anzahl von vier nebenläufigen Instanzen ist ein deutlicher Anstieg zu erkennen. Die Laufzeiten der Simulationsinstanzen nehmen dabei von drei auf vier nebenläufige Simulationsinstanzen um 3 s zu. Bei Hinzunahme einer fünften Simulationsinstanz nimmt die Laufzeit um fast 10 s zu, beim Schritt von fünf auf sechs nebenläufige Simulationsinstanzen ergibt sich ein weiterer Anstieg der Laufzeit um circa 12 s.

Anz. nebenl.SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F [s]
1	72.44					
2	72.81	72.79				
3	73.45	73.34	73.42			
4	77.93	77.87	77.82	77.96		
5	87.73	87.39	88.33	87.38	88.78	
6	98.59	99.74	99.04	100.10	99.03	98.65

Tabelle 8.11: *SbF*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit festem Simulationskernel SeriellBeschleunigt bei 32 Molekülen.

Die nachfolgende Tabelle 8.12 zeigt die Laufzeit der unterschiedlichen Simulationsinstanzen für die Simulation mit automatischer Wahl von Simulationskerneln CudaNormal und SeriellBeschleunigt. Diese Messreihe *SA 1* zeigt im Vergleich zur Simulation mit festem Simulationskernel CudaNormal kürzere Laufzeiten ab einer Anzahl von zwei nebenläufiger Simulationsinstanzen. Im Vergleich zur Simulation mit festem Simulationskernel SeriellBeschleunigt zeigen sich kürzere Laufzeiten für zwei, drei und vier nebenläufige Simulationsinstanzen. Für fünf und sechs nebenläufige Simulationsinstanzen wurden ähnliche Laufzeiten wie für die Simulation mit festem Simulationskernel SeriellBeschleunigt gemessen.

Anz. nebenl.SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F [s]
1	38.94					
2	50.80	51.85				
3	57.42	66.02	56.18			
4	63.83	69.20	68.64	65.14		
5	86.35	86.79	85.38	87.58	85.89	
6	97.50	98.25	99.33	98.20	97.22	98.22

Tabelle 8.12: *AS 1*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal bei 32 Molekülen.

Die Laufzeiten der Messreihe *SA 2* zeigt ein analoges Verhalten zur Messreihe *SA 1*: Für eine bis vier nebenläufige Simulationsinstanzen liegt die Laufzeit der einzelnen Simulationsinstanzen von *SA 1* unter der Laufzeit der Messreihe für festen Simulationskernel SeriellBeschleunigt (Datenreihe *SbF*). Bei fünf und sechs nebenläufigen Simulationsinstanzen befinden sich die Laufzeiten der Simulationsinstanzen von *SA 2* im Bereich der Laufzeiten für festen Simulationskernel SeriellBeschleunigt (Datenreihe *Cn*).

Anz. nebenl.SI	Sim A [s]	Sim B [s]	Sim C [s]	Sim D [s]	Sim E [s]	Sim F [s]
1	39.10					
2	66.54	68.31				
3	69.68	72.74	69.06			
4	75.61	68.51	74.71	73.36		
5	88.53	91.36	90.62	89.71	89.19	
6	98.23	98.44	97.98	98.25	97.40	97.54

Tabelle 8.13: *AS 2*: Laufzeiten für bis zu sechs nebenläufige Simulationsinstanzen mit automatisierter Wahl von Simulationskerneln SeriellBeschleunigt und CudaNormal.

Die nachfolgende Abbildung 8.8 zeigt die maximalen Laufzeiten der Simulationsinstanzen für eine bestimmte Anzahl an nebenläufiger Simulationsinstanzen für alle vier gemessenen Datenreihen.

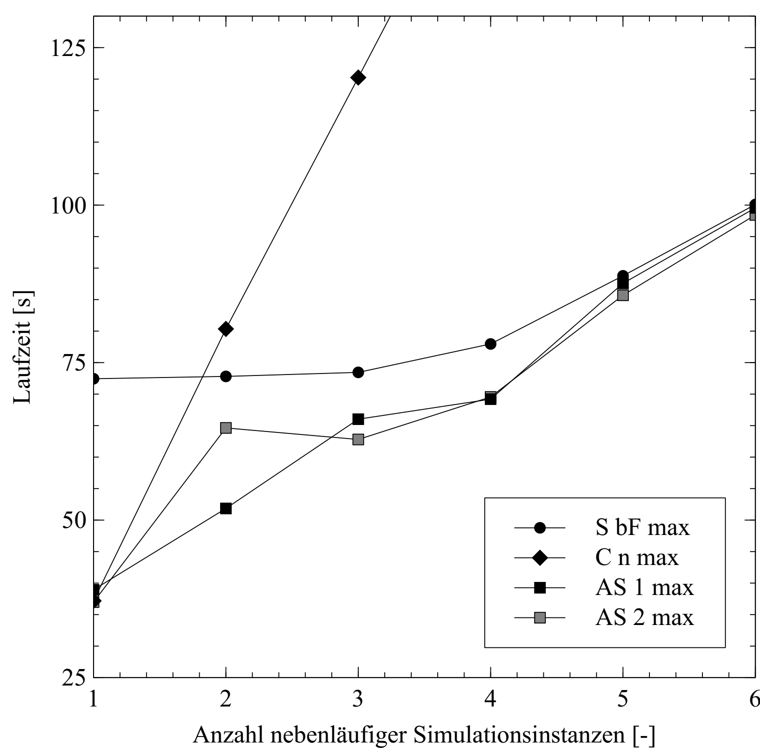


Abbildung 8.8: Maximale Laufzeit der Simulationsinstanzen für unterschiedliche Anzahl nebenläufiger Simulationsinstanzen. S bF max beschreibt die maximale Laufzeit unter Einsatz von SeriellBeschleunigt, C n max die maximale Laufzeit unter Einsatz von CudaNormal. *AS 1* und AS 2 bezeichnen die beiden Messreihen mit automatischem Scheduling von Simulationskerneln.

Dabei lässt sich erkennen, dass die maximale Laufzeit der Simulationsinstanzen von der Simulation mit festem Simulationskernel CudaNormal in dem betrachteten Bereich stark zunimmt. Die ermittelten maximalen Laufzeiten der beiden Messreihen *AS 1* und *AS 2* mit automatischer Wahl des Simulationskernels bewegen sich dabei im Bereich der für einen festen Simulationskernel SeriellBeschleunigt ermittelten maximalen Laufzeit dessen Simulationsinstanzen. In der folgenden Tabelle 8.14 sind diese maximalen Laufzeiten der vier betrachteten Messreihen pro Anzahl nebenläufiger Simulationsinstanzen dargestellt:

Anz. nebenl.SI	C n max	S bF max	AS 1 max	AS 2 max
1	37.18	72.44	38.94	36.99
2	80.35	72.81	51.85	64.63
3	120.25	73.45	66.02	62.80
4	160.41	77.96	69.20	69.55
5	200.77	88.78	87.58	85.69
6	240.97	100.10	99.56	98.44

Tabelle 8.14: Maximale Laufzeiten

Für weitere angestrebte Messreihen mit 128 Molekülen konnten keine weiteren Messreihen durchgeführt werden, da die Simulation auf CPU gegenüber der Simulation auf GPU in diesem Bereich zu langsam war.

8.4 Diskussion

8.4.1 Vergleich der seriellen Simulationskernel mit den Simulationskerneln auf hybriden Architekturen

Die Betrachtung der durchschnittlichen Laufzeiten der Simulationskernel CudaNormal und SeriellNormal, sowie analog dazu CudaBeschleunigt und SeriellBeschleunigt, zeigte eine deutlich kürzere Laufzeit für die Simulationskernel CudaNormal und CudaBeschleunigt auf hybriden Architekturen. Lediglich für sehr kleine Molekül- und Mutantenanzahlen waren die seriellen Simulationskernel in der Lage, die Ergebnisse schneller zu berechnen. Des Weiteren wiesen auch die Bereiche der Datenreihen von CudaNormal und CudaBeschleunigt, in denen die Simulationskernel auf hybriden Architekturen mit zunehmender Molekülanzahl länger zur Evaluation benötigten, einen geringeren Anstieg der Laufzeit als die seriellen Simulationskernel SeriellNormal und SeriellBeschleunigt auf. Es wurden Beschleunigungsfaktoren von bis zu 494.6x des Simulationskernel CudaBeschleunigt gegenüber SeriellNormal bei 8192 Molekülen und 8 Mutanten ermittelt. Für eine

Diese Ergebnisse sprechen für die von [1] vorgestellte Parallelisierung der GCMC-Simulation und unterstützen die dazu ermittelten Werte. Die massive Parallelität der GPGPU erlaubt hier eine enorme Beschleunigung bei der Auswertung von Mutanten.

Für die zu Beginn dieser Studienarbeit gegebenen seriellen Simulationskernel wurde eine recht naive Implementierung besonders für die kurzreichweitigen Potentiale verwendet. Eine sich in diesem Fall anbietende räumliche Dekomposition wie zum Beispiel durch das *Linked-Cell-Verfahren* [11] wurde nicht durchgeführt und könnte die Laufzeit der seriellen Simulationskernel weiter verbessern. Dies war jedoch nicht Gegenstand dieser Studienarbeit . Für eine weitere Betrachtung der GCMC-Simulation auf hybriden Architekturen wäre der kombinierte Einsatz von durchsatzoptimierten Architekturen und latenzoptimierten Architekturen auch für eine räumliche Dekomposition für die Berechnung der kurzreichweitigen Potentiale interessant: Eine räumliche Aufteilung des Simulationsgebiets in Zellen, wobei jede Zelle in einen statischen und einen dynamischen Block getrennt wird. Der statische Block enthält eine an die Kernanzahl der GPGPU angepasste Anzahl von Molekülen, sodass eine optimale Ausnutzung des Durchsatzes der GPGPU bei einer Anfrage erreicht wird. Der dynamische Block enthält all die Moleküle, die aus Platzgründen nicht in den statischen Block der Zelle passen beziehungsweise zu einem weiteren, nicht vollständig befüllten Block führen würden. Diese Moleküle können parallel zur Ausführung der Anfrage an die GPGPU auf der CPU ausgewertet werden.

8.4.2 Vergleich der Simulationskernel CudaNormal und CudaBeschleunigt

Bei den Ergebnissen zur Laufzeit der Simulationskernel CudaNormal und CudaBeschleunigt zeigte sich, dass CudaNormal während der Phase der konstanten Laufzeiten im Bereich kleiner Molekülanzahlen schneller als CudaBeschleunigt war. Jedoch konnten mit CudaBeschleunigt im sich anschließenden Bereich steigender Laufzeiten schnellere Laufzeiten als bei CudaNormal beobachtet werden. Mit zunehmender Anzahl betrachteter Mutanten wurde dieser Punkt stiegender Laufzeiten für CudaNormal bei immer kleineren Molekülanzahlen erreicht, während bei CudaBeschleunigt dieser Punkt konstant bei etwa 1000 Molekülen lag. Eine interessante Beobachtung ließ sich bei der Simulation von einem Mutanten feststellen: Während CudaBeschleunigt für dieses Szenario nach den in Kapitel 6 angestellten Überlegungen keine Beschleunigung erfahren sollte, ließ sich trotzdem ab einem gewissen Punkt ein beschleunigtes Verhalten von CudaBeschleunigt gegenüber CudaNormal feststellen.

Diese Ergebnisse zeigen, dass die vorgestellte beschleunigte Berechnung des Fourierraumanteils für die Simulationskernel auf hybriden Architekturen erst dann zu einer Beschleunigung führt, wenn die Kapazität der GPGPU erreicht ist, diese jedoch danach zu schnelleren Ergebnisse führt. Das Verhalten von CudaNormal, bei der Betrachtung einer zunehmenden Anzahl von Mutanten bereits bei jeweils kleineren Molekülanzahlen in den Bereich einer steigenden Laufzeit zu wechseln, spiegelt die Stärke der beschleunigten Berechnung des Fourierraumanteils für mehrere simultane *Monte-Carlo*-Schritte wieder: Während CudaNormal $N \cdot M$ GPGPU Thread zur Berechnung der Koeffizienten benutzt, werden in CudaBeschleunigt $N + M$ GPGPU Thread zur Berechnung der Koeffizienten für alle Mutanten benötigt.

Die Beobachtung der Differenz in der Laufzeit zwischen CudaNormal und CudaBeschleunigt während der Phase konstanter Laufzeit resultiert vermutlich daraus, dass CudaNormal drei GPGPU Kernel verwendet, während bei CudaBeschleunigt vier GPGPU Kernel eingesetzt wurden. Das Verhalten von CudaNormal gegenüber CudaBeschleunigt nach dem Auslastungspunkt für einen einzelnen Mutanten lässt sich möglicherweise durch unterschiedlich stark ausgeprägte Fallunterscheidungen in den GPGPU Kerneln von CudaNormal und CudaBeschleunigt bezüglich des Typs des betrachteten Mutanten erklären. Im Kernel für die Berechnung der Koeffizienten von CudaNormal muss in jedem GPGPU Thread unterschieden werden, um welchen Typ es sich handelt. Bei CudaBeschleunigt entfällt dies bei der Berechnung der Koeffizienten für alle Atome. Die Fallunterscheidungen treten hier nur im GPGPU Kernel zur Berechnung der Koeffizienten für die von *Monte-Carlo*-Schritten betroffenen Atomen auf.

8.4.3 Automatisierte Wahl von Simulationskerneln

Auffallend bei den Messreihen zur automatisierten Wahl von Simulationskerneln war der in etwa lineare Anstieg der Laufzeiten der Simulationsinstanzen von CudaNormal bei Zunahme der Anzahl nebenläufiger Simulationsinstanzen. Einzelne Simulationen mit einer Anzahl von Molekülen in der Größe der Summe der Moleküle der betrachteten nebenläufigen Simulationsinstanzen zeigten dabei im Vergleich viel geringere Laufzeiten. Dieses Verhalten spiegelt eine Schwäche der für die Simulation eingesetzen Generation von GPGPUs wieder: Es wird nur einem Prozess der CPU simultan Zugriff auf die GPGPU gewährt. Für kommende Generationen von GPGPUs zeigen sich jedoch Entwicklungen, die die simultane Nutzung der GPGPU durch mehrere CPU-Prozesse erlauben. Um das gesamte Potential einer Architektur trotz der entsprechenden Beschränkung zu nutzen, müsste demnach eine eine parallele Simulation mehrerer Simulationsinstanzen auf Ebene von GPGPU Threads durchgeführt werden.

Die Ergebnisse zur automatisierten Wahl von Simulationskerneln ergaben weiterhin, dass mit der implementierten Methode für die betrachteten System mit 32 und 64 Molekülen insgesamt geringere Laufzeiten erreicht wurden als bei Einsatz eines festen Simulationskernels CudaNormal oder SeriellBeschleunigt, trotz der o.g. Restriktion durch die Architektur. Für die durch das Referenzsystem gestellten Beschränkungen war die Simulation mehrerer nebenläufiger Simulationsinstanzen in der Lage, sich an die höhere Auslastung der Architekturen anzupassen.

9 Zusammenfassung

Ziel dieser Arbeit war die Bereitstellung eines Frameworks für die effiziente Simulation von Monte-Carlo-Molekularsimulationen auf hybriden Architekturen. Dieses Framework sollte zum Einen eine leichte Bedienbarkeit ermöglichen, zum Anderen die potentielle Beschleunigung durch den Einsatz von aktuellen und kommenden Architekturen nutzen. Für diesen Zweck stand die Erweiterbarkeit der Simulationsumgebung im Vordergrund, die auch die Implementierung anderer Berechnungsmethoden oder die Portierung für andere Einsatzgebiete oder Architekturen ermöglicht. Ausgehend von einer Parallelisierung für GCMC-Simulationen auf hybriden Architekturen wurde dabei eine Simulationsumgebung mit Blick auf Kommunikationsminimierung und einer intuitiven, aber dennoch effizienten Schnittstelle zwischen Simulation und Anwender entworfen.

Die in dieser Studienarbeit zur Simulation von MCMC-Simulationen auf hybriden Architekturen entwickelten Konzepte wurden so allgemein gehalten, dass sie sich direkt auf andere Anwendungen übertragen lassen.

Die *Hybrid-Simulation-State-Machine* ermöglicht die Implementierung von für verschiedene Architekturen und unterschiedliche Einsatzszenarien optimierten, effizienten Simulationskerneln. Dies geschieht durch den Einsatz eines *Replicated-Data*-Schemas und der Verwendung einer klar definierten Schnittstelle. Eine Trennung von Initialisierungs- und Simulationsphase erlaubt hierbei auf der einen Seite eine einfache Konfiguration der Simulation. Auf der anderen Seite werden Fehler in der Verwendung des Simulationsframeworks, wie zum Beispiel durch Änderung von Simulationsparametern während der Simulation oder die Akzeptierung mehrerer *Monte-Carlo*-Schritte ausgeschlossen. Dies würden zu einer Minderung der Effizienz der Simulation, oder sogar zu Fehlern in den Ergebnissen der Simulation führen. Für die Simulationskernel wird durch diese klare Trennung eine auf die gegebenen Simulationsparametern angepasste Initialisierung mit Möglichkeit des Einsatzes effizienter Datenstrukturen ermöglicht. Des Weiteren wurde das Konzept der *Hybrid-Simulation-State-Machine* mit Blick auf die Erweiterbarkeit durch weitere Simulationskernel entworfen. Durch in der *Hybrid-Simulation-State-Machine* für MCMC-Simulationen implementierbare zur automatischen Wahl von Simulationskerneln ist es möglich, einen für die aktuelle Konfiguration schnellen Simulationskernel für die Simulation einzusetzen, sowie auf Änderungen der Auslastung der zur Simulation verwendeten Architektur zu reagieren. Eine solche Methode wurde für die gegebene GCMC-Simulation implementiert. Die *Hybrid-Simulation-State-Machine* ist außerdem in der Lage, auf Fehlermeldungen von einzelnen Simulationskerneln reagieren zu können und entsprechende Maßnahmen zur Beseitigung dieser Fehler einzuleiten.

Das Konzept der *Change-Linked-Simulation* wurde entwickelt, um während der Simulationsphase eine intuitive und effiziente Möglichkeit der Kommunikation zwischen Anwender

und der *Hybrid-Simulation-State-Machine* für MCMC-Simulationen bereit zu stellen. Hierbei werden unnötige Kommunikationsschritte minimiert und die zur Simulation benötigten Daten in kompakter Form an die *Hybrid-Simulation-State-Machine* und die eingesetzten Simulationskernel propagiert. Als universelle Schnittstelle für eine beliebige Anzahl simultan zu evaluierender *Monte-Carlo*-Schritte bietet die *Change-Linked-Simulation* außerdem eine klare Zuordnung der zu jedem *Monte-Carlo*-Schritt ermittelten Ergebnisse mittels der Repräsentation von *Monte-Carlo*-Schritten als gemeinsam von Anwender und Simulation genutzten *Änderungsinstanzen*. Dabei wird von der *Hybrid-Simulation-State-Machine* sichergestellt, dass bestimmte Kriterien für die Simulation, wie zum Beispiel die Markovketteneigenschaft in der MCMC-Simulation nicht verletzt werden.

Die zu Beginn dieser Studienarbeit gegebene Parallelisierung für GCMC-Simulationen wurde im Zuge der Implementierung weiter entwickelt und für eine simultane Auswertung mehrerer *Monte-Carlo*-Schritte weiter optimiert.

Die für eine GCMC-Simulation durchgeführten Experimente haben gezeigt, dass durch die Simulation auf hybriden Architekturen ein großer Performancegewinn erzielt werden kann. Bereits für eine geringe Anzahl von Molekülen des betrachteten Referenzsystems wurde dabei durch den Einsatz von durchsatzoptimierten Architekturen profitiert. Besonders im Bereich einer großen Anzahl von betrachteten Molekülen ergaben sich hohe Beschleunigungsfaktoren im dreistelligen Bereich gegenüber einer rein seriellen Simulation. Auch für die simultane Auswertung mehrerer *Monte-Carlo*-Schritte in einem Simulationsschritt konnte durch die Verbesserung der Parallelsierung von GCMC-Simulationen ein Beschleunigungsfaktor von 9.8x gegenüber der ursprünglichen, bereits sehr schnellen Implementierung erreicht werden. Die für die GCMC-Simulation implementierte Methode zur automatischen Wahl von Simulationskerneln war in der Lage, für eine gegebene Situation eine gute durchschnittliche Laufzeit für mehrere nebenläufige Simulationsinstanzen zu erzielen und sich an die geänderte Gesamtauslastung der verwendeten Architekturen anzupassen.

Literaturverzeichnis

[1] C. Braun, S. Holst, J. M. Castillo, J. Gross, and H.-J. Wunderlich, "Acceleration of Monte-Carlo Molecular Simulations on Hybrid Computing Architectures," in *Proc. of 30th IEEE International Conference on Computer Design (ICCD)*, Oct. 2012, pp. 207–212. (Zitiert auf den Seiten 9, 11, 12, 17, 24, 49, 50, 51 und 74)

[2] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953. (Zitiert auf den Seiten 11 und 16)

[3] P. P. Ewald, "Die Berechnung optischer und elektrostatischer Gitterpotentiale," *Annalen der Physik*, vol. 369, no. 3, pp. 253–287, 1921. (Zitiert auf den Seiten 15, 41 und 45)

[4] D. Frenkel and B. Smit, *Understanding Molecular Simulation, 2nd ed.* Academic Press, 2002. (Zitiert auf den Seiten 16 und 17)

[5] K. Esselink, L. D. J. C. Loyens, and B. Smit, "Parallel monte carlo simulations," *Physical Review E*, vol. 51, no. 2, pp. 1560–1568, Feb 1995. (Zitiert auf Seite 17)

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. (Zitiert auf Seite 19)

[7] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.41 (Zitiert auf den Seiten 20 und 21)

[8] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7 –17, Sept.-Oct. 2011. (Zitiert auf Seite 21)

[9] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28 –37, March-April 2012. (Zitiert auf Seite 21)

[10] M. Daga, A. Aji, and W. chun Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing," in *2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, July 2011, pp. 141 –149. (Zitiert auf Seite 21)

[11] M. P. Allen and D. J. Tildesley, *Computer simulation of liquids*. New York, NY, USA: Clarendon Press, 1989. (Zitiert auf den Seiten 25 und 74)

[12] 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*, vol. 106, no. 738, pp. 441–462, 1924. (Zitiert auf Seite 41)

[13] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, and J. Hermans, *Intermolecular Forces*, 1981, ch. Interation Models for Water in Relation To Protein Hydration, pp. 331–342. (Zitiert auf Seite 53)

Alle URLs wurden zuletzt am 29. 11. 2012 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschift