JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN

Monte-Carlo-Simulationen des Ising-Modells mit Multispin-Coding auf GPUs

Monte-Carlo-Simulations of the Ising model with multispin coding on GPUs

Bachelorthesis

in Physik

von

Milad Ghanbarpour

Matrikelnummer: 2011885

eingereicht im

September 2016

Erstgutachter: Prof. Dr. Lorenz von Smekal

Zweitgutachter: Dr. Dominik Smith

Zusammenfassung

Die hier vorliegende Arbeit beschäftigt sich mit der Monte-Carlo-Simulation des Ising-Modells im kanonischen Ensemble auf GPUs für ein zweidimensionales Gitter, wobei der dreidimensionale Fall an gewissen Stellen mit betrachtet wird. Im Vordergrund steht die Verwendung der Multispin-Coding-Methode, welches die einzelnen Bits einer Speicherstelle zur Kodierung von Spinzuständen des Gitters verwendet. Im Rahmen dieser Arbeit soll hierbei untersucht werden, ob und wie der Demon-Algorithmus in [5], [6] und [21] beschrieben, unter Verwendung der Multispin-Coding-Methode, auf die GPU umgesetzt werden kann. Das Resultat wird anschließend mit der bekannten Umsetzung, die auch Multispin-Methoden verwendet, des Metropolis-Algorithmus in [31] und [32] für GPUs verglichen. Hierbei sind beide Verfahren in dieser Arbeit explizit implementiert. Ferner wird auch der Aufbau und die Arbeitsweise von GPUs des Hersteller *Nvidia* diskutiert und die sich daraus ergebenden Konsequenzen für die Konstruktion von Implementierungen nummerischer Simulationen auf GPUs.

Es stellt sich heraus, dass die Implementierungen des Demon-Algorithmus für kleine Gitter mit circa 10^5 Spins durchaus schneller sein können als der Metropolis-Algorithmus. Für größer Gitter zeigt sich jedoch, dass der Demon-Algorithmus rapide an Geschwindigkeit einbüßt, während die des Metropolis-Algorithmus langsamer, bezogen auf die Vergrößerung des Gitters, abnimmt.

Inhaltsverzeichnis

1 Einleitung **1**

2 Das Ising-Modell **3**
2.1 Definition des Ising-Modells . . . 3
2.2 Kopplung am Wärmebad . . . 4
2.2.1 Grundlagen . . . 4
2.2.2 Phasenübergänge . . . 7
2.2.2.1 Phasenübergang für das zweidimensionale Ising-Modell . . 8
2.2.2.2 Kritisches Verhalten an Phasenübergängen . . . 10
2.2.2.3 Finite-Size-Scaling-Gesetze . . . 10

3 Monte-Carlo-Simulationen des Ising-Modells **13**
3.1 Grundlagen . . . 13
3.1.1 Markov-Kette . . . 15
3.1.2 Zufallsgeneratoren . . . 17
3.2 Algorithmen für das Ising-Modell . . . 18
3.2.1 Metropolis-Algorithmus . . . 18
3.2.2 Demon-Algorithmus . . . 19
3.2.2.1 Mikrokanonischer Update des Gesamtsystemes . . . 22
3.2.2.2 Kanonisches Update des Demon-Systems . . . 22

4 GPU-Programmierung **24**
4.1 Grafikprozessoren . . . 24
4.2 CUDA und GPU-Architektur . . . 26
4.2.1 Compute Capability . . . 28
4.2.2 Speichermodell der GPU . . . 28
4.2.2.1 Global Memory . . . 29
4.2.2.2 Shared Memory . . . 30
4.2.2.3 Zugriffsgeschwindigkeiten . . . 31
4.3 CUDA C/C++ Programmierung . . . 31
4.3.1 Programmierung von *kernels* . . . 32
4.3.2 Speicherreservierung . . . 34
4.3.2.1 Global Memory . . . 34
4.3.2.2 Shared Memory . . . 34

5 Implementation der Algorithmen **36**
5.1 Multispin-Coding . . . 36
5.2 Implementierung des Metropolis-Algorithmus . . . 37
5.2.1 Programmaufbau . . . 37
5.2.2 Gitteraufbau . . . 38
5.2.3 Update der Spinkonfiguration . . . 39
5.2.3.1 Zufallszahlen innerhalb von *threads* . . . 40
5.2.4 Berechnung der totalen Magnetisierung . . . 40
5.3 Implementierung des Demon-Algorithmus . . . 41
5.3.1 Programmaufbau . . . 41
5.3.2 Gitteraufbau . . . 41
5.3.3 Update der Spinkonfigurationen . . . 43
5.3.3.1 Mikrokanonisches Update der Spinkonfigurationen . . . 43
5.3.3.2 Kanonisches Update der Demons . . . 47
5.3.4 Berechnung der totalen Magnetisierung . . . 49

6 Analyse und Vergleich der Implementierungen **50**
6.1 Laufzeitanalyse der Implementierungen . . . 50
6.2 Simulation des zweidimensionalen Gitters . . . 55
6.2.1 Qualitative Untersuchung der Implementierungen . . . 55
6.2.2 Quantitative Untersuchungen der Implementierungen . . . 61
6.2.2.1 Statistischer Fehler und Autokorrelation . . . 61
6.2.2.2 Binderkumulante und kritischer Exponent . . . 64

7 Zusammenfassung und Ausblick **68**

8 Anhang **69**
8.1 Simulationsparameter für Abschnitt 6.2.2.2 . . . 69
8.2 Dokumentation der Quellcodes . . . 71
8.2.1 Metropolis-Algorithmus . . . 71
8.2.1.1 routines2D.hpp . . . 71
8.2.1.2 routines3D.hpp . . . 71
8.2.1.3 rng.hpp . . . 72
8.2.1.4 Lattice.hpp . . . 74
8.2.1.5 routines2D.cu . . . 77
8.2.1.6 routines3D.cu . . . 85
8.2.1.7 Lattice.cu . . . 93
8.2.2 Demon-Algorithmus . . . 95
8.2.2.1 multint.hpp . . . 95
8.2.2.2 rng.hpp . . . 96
8.2.2.3 routines2D.hpp . . . 100
8.2.2.4 routines3D.hpp . . . 103
8.2.2.5 Lattice.hpp . . . 107
8.2.2.6 routines2D.cu . . . 113

8.2.2.7 routines3D.cu 134
8.2.2.8 Lattice.cu 155

Kapitel 1

Einleitung

Heutige Rechenmaschinen bieten die Möglichkeit für viele physikalische Probleme, bei denen keine analytischen Lösungen existieren, nummerische Ergebnisse mit gewünschter Genauigkeit zu berechnen. Im Rahmen der statistischen Physik sind solche nummerische Betrachtungen von Interesse. Dies liegt daran, dass realistische Modelle, die für die Beschreibung von thermodynamischen Phänomenen dienlich sind, meist aufgrund von Wechselwirkungen von Teilchen miteinander zu analytisch schwer zu betrachtenden Verhalten führen. Folglich die entsprechenden Größen, wie die thermodynamischen Potentiale, nicht ohne weiteres exakt angegeben werde können. Die statistische Physik beschäftigt sich mit der Bestimmung von Erwartungswerten, wie Druck und Volumen, von einem gegeben System bei vorgegebenen thermodynamischen Größen. Die Aufgabe der nummerischen Simulation ist es hierbei für vorgegebene Größen entsprechend die gewünschten Erwartungswerte mit einer gewissen Genauigkeit zu berechnen.

Die hier vorliegende Arbeit beschäftigt sich mit der Simulation des Ising-Modells. Im Rahmen dieses Modells wird ein d-dimensionales Gitter betrachtet, an dessen einzelnen Gitterpunkten Spins lokalisiert sind, die je zwei Zustände einnehmen können. Hierbei wechselwirken benachbarte Spins über eine Kopplungskonstante miteinander, siehe Kapitel 2. Für das Ising-Modell existieren durchaus analytische Betrachtungen für $d = 1$ (siehe [13]) und $d = 2$ (siehe [4]). Jedoch nicht für höhere Dimensionen, wobei zusätzlich die Betrachtungen für $d = 2$ nicht trivial sind. In dieser Arbeit wird das Ising-Modell gekoppelt an ein Wärmebad mit der Temperatur T betrachtet. Es geht vorwiegend um die Simulation im kanonischen Ensemble mit Hilfe von Monte-Carlo-Methoden, wobei die besprochenen Verfahren auf Grafikprozessoren (GPU) implementiert werden und die Multispin-Coding-Methode, welches jedes Bit eines Wortes zur Abspeicherung eines Spinzustandes benutzt, verwendet wird. Es werden in dieser Arbeit die zwei Verfahren, Metropolis-Algorithmus und Demon-Algorithmus, implementiert.

Für den Metropolis-Algorithmus auf GPUs mit Multispin-Coding existieren hierbei die Arbeiten in [31] und [32], auf dessen Basis entsprechend der Metropolis-Algorithmus in dieser Arbeit implementiert wird.

Der Demon-Algorithmus wird beispielsweise in den Arbeiten [6], [21] und [20] behandelt und ist für CPUs mit Multispin-Coding implementiert und liefert besonders schnelle Simulationen bei wenigen benötigten Zufallszahlen. In dieser Arbeit wird der Fragestel-

lung, in wie weit dieser Algorithmus auf die GPU umgesetzt werden kann, nachgegangen und anschließend mit der Implementierung des Metropolis-Algorithmus verglichen. Ferner werden für die beiden Implementierungen grundlegende Simulationen für ein zweidimensionales Gitter ausgeführt und der Phasenübergang im thermodynamischen Limes untersucht, was einerseits eine Verifikation der Funktionalität der Implementierungen ist und andererseits die grundlegenden Methodik und Problematiken bei derartigen Simulationen illustriert.

Um dies zu bewerkstelligen müssen entsprechend die Grundlagen erarbeitet werden. In Kapitel 2 werden die grundlegenden Eigenschaften des Ising-Modells, wie zum Beispiel den Phasenübergang für ein Gitter mit $d = 2$, besprochen. Im darauffolgenden Kapitel werden die Grundideen der Monte-Carlo-Simulationen dargestellt und insbesondere die beiden Verfahren Metropolis-Algorithmus und Demon-Algorithmus detailliert angegeben. Kapitel 4 beschäftigt sich mit der Programmierung von Grafikprozessoren des Hersteller *Nvidia*. Aufgrund der Tatsache, dass GPUs eine anderen Aufbau als CPUs besitzen gibt es gewisse Punkte, die bei der Programmierung beachtet werden müssen und die Implementierungen maßgeblich beeinflussen.

Nachdem diese Grundlagen getroffen sind, werden in Kapitel 5 die Implementierungen der beiden Verfahren besprochen und anschließend in Kapitel 6 miteinander verglichen und entsprechend die Simulationsergebnisse diskutiert.

Kapitel 2

Das Ising-Modell

2.1 Definition des Ising-Modells

Wir wollen zunächst das Ising-Modell definieren und anschließend genauer betrachten.

Beim Ising-Modell handelt es sich, um ein physikalisches Modellsystem eines endlichen Gitters

$$G^d_{(L_1,\ldots,L_d)} := \{x | x = (x_1, \ldots, x_d) \in \mathbb{N}_0^d \wedge \forall i \in \{1, \ldots, d\} : x_i < L_i\} \tag{2.1.1}$$

mit der Dimension d und Ausdehnungen L_i in den i-ten Richtungen, bei der jedem Gitterpunkt ein Spin mit zwei Einstellungsmöglichkeiten zugeordnet ist. Die einzelnen Spins wechselwirken dabei mit ihren nächsten Nachbarn. Der Begriff Spin ist in der oberen Definition zunächst allgemein gehalten. Es muss sich nicht zwangsläufig um einen Spin von einem Teilchen, wie ein Elektron handeln, womit beispielsweise das Ising-Modell zur Beschreibung vom Ferromagnetismus verwendet wird. Ganz allgemein dient diese Definition dazu ein Gesamtsystem zu beschreiben, dass aus identischen, an den Gitterpunkten platzierten, Systemen besteht, die sich jeweils nur in einem von zwei möglichen Zuständen befinden können. Eine Illustration eines solchen Gitters für $d = 3$ findet sich in Abbildung 2.1.1. Die Zustände der Einzelsysteme werden beispielsweise mit die Zahlen -1 oder 1 gekennzeichnet oder über zwei sich entgegensetzende Pfeile, wie in Abbildung 2.1.1 vorzufinden, dargestellt. Eine Kombination der einzelnen Zustände von den Gitterpunkten wird als Spinkonfiguration bezeichnet, die über eine Abbildung

$$\sigma : G^d_{(L_1,\ldots,L_d)} \longrightarrow \{-1, 1\} \tag{2.1.2}$$

repräsentiert werden kann. Offensichtlich existieren insgesamt $2^{L_1 \cdot L_2 \cdot \ldots \cdot L_d}$ solcher Spinkonfigurationen für ein Gitter. Die Menge der Konfigurationen bezeichnen wir als $\Lambda^d_{(L_1,\ldots,L_d)}$, oder, falls es aus dem Kontext ersichtlich sein sollte, nur Λ. Die Wechselwirkung der einzelnen Gitterpunkte wird über die Hamiltonfunktion

$$H(\sigma) = -\sum_{\langle x,y \rangle} J_{\langle x,y \rangle} \sigma(x)\sigma(y) - \sum_{x \in G} h_x \sigma(x) \tag{2.1.3}$$

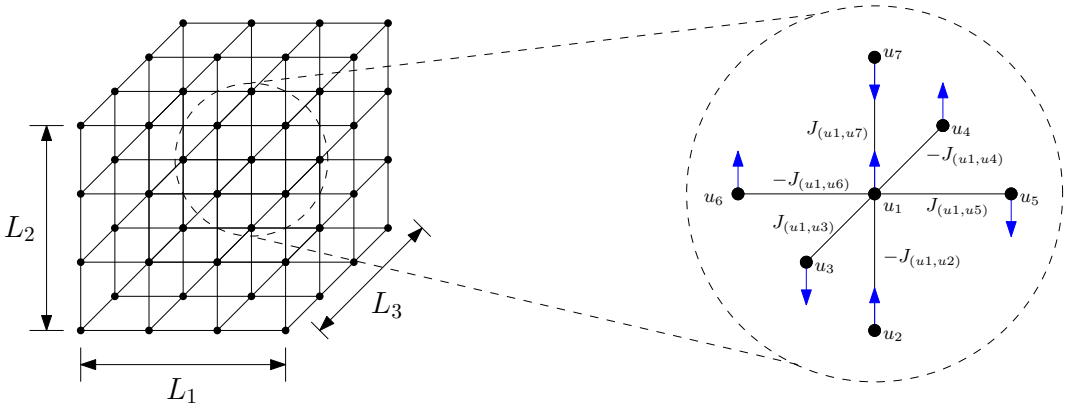


Abbildung 2.1.1: Illustration des Ising-Modells für $d = 3$.

festgelegt, wobei $\langle x, y\rangle$ für eine Kante zwischen zwei benachbarten Gitterpunkten steht, wie in Abbildung 2.1.1 dargestellt. Für Randpunkte mit $g_{L_i} = (x_1, \ldots, L_i, \ldots, x_d)$ ist der nächste Nachbar für die positive i-te Richtung der Gitterpunkt bei $g_0 = (x_1, \ldots, 0, \ldots, x_d)$. Entsprechend existiert zwischen beide Gitterpunkten eine Kante $\langle g_0, g_{L_i}\rangle$. Die erste Summe stellt somit eine Summierung über alle Kanten eines Gitters dar, wobei jeder Kante eine Kopplungskonstante $J_{\langle x,y\rangle}$ zugeordnet ist. Durch das Produkt $\sigma(x)\sigma(y)$ wird bestimmt, ob die Gesamtenergie durch parallele Ausrichtung der Spins um $-J_{\langle x,y\rangle}$ herabgesetzt oder durch anti-parallele Ausrichtung der Spins um $J_{\langle x,y\rangle}$ erhöht wird. Die zweite Summe in $H(\sigma)$ berücksichtigt die Möglichkeit, dass ein äußeres, möglicherweise ortsabhängiges, Feld h_x vorhanden sein kann.

Es sei darauf zu achten, dass $\langle x, y\rangle$ unabhängig von der Reihenfolge (x, y) ist und in diesem Fall repräsentativ für eine Kante steht. Eine Abhängigkeit der Reihenfolge führt zu einer doppelten Zählung in der ersten Summe. Folglich müsste ein Faktor $\frac{1}{2}$ vor der ersten Summe stehen, um dieser Tatsache gerecht zu werden.

Im Folgenden wird stets ein allgemeines endliches d-dimensionales Gitter im Rahmen des Ising-Modells mit Kopplungskonstanten $J_{\langle x,y\rangle}$ und äußerem Feld h_x voraus gesetzt, falls keine spezifischen Einschränkungen angegeben werden.

2.2 Kopplung am Wärmebad

2.2.1 Grundlagen

Das Ising-Modell an sich wird typischerweise im Rahmen der statistischen Physik im kanonischen Ensemble betrachtet und auch in dieser Arbeit geht es um die Simulation des Ising-Modells gekoppelt an ein Wärmebad. Dementsprechend folgen an dieser Stelle grundlegende statistische Betrachtungen, die später wiederholt aufgegriffen werden.

Koppeln man das Spin-Gitter im Rahmen des Ising-Modells an ein Wärmebad mit der Temperatur T, so erhält man für die Wahrscheinlichkeitsverteilung $P(\sigma)$ der Spinkonfi-

gurationen

$$P(\sigma) = \frac{1}{Z} e^{-\frac{1}{kT} H(\sigma)} \tag{2.2.1}$$

wobei Z die Zustandssumme und durch

$$Z = \sum_{\sigma \in \Lambda} e^{-\frac{1}{kT} H(\sigma)} \tag{2.2.2}$$

gegeben ist.

Der Erwartungswert $\langle A \rangle$ einer Observablen $A(\sigma)$ ist durch

$$\langle A \rangle = \frac{1}{Z} \sum_{\sigma \in \Lambda} e^{-\frac{1}{kT} H(\sigma)} \tag{2.2.3}$$

gegeben. Für die in dieser Arbeit getroffenen Betrachtungen sind vor allem Systeme interessant, bei denen $J_{\langle u,v \rangle} = J_c = const.$ und $h = 0$ ist. Hierbei werden die Randbedingungen, das heißt die Kopplungskonstanten $J_{\langle g_0, g_{L_i} \rangle} = J_c$ der Kanten $\langle g_0, g_{L_i} \rangle$, in diesem Fall als periodische bezeichnet. Es existieren auch anti-periodische Randbedingungen, die durch $J_{\langle g_0, g_{L_i} \rangle} = -J_c$ festgelegt sind. Für solche Systeme mit periodischen Randbedingungen lässt sich die Zustandssumme Z und der Erwartungswert $\langle A \rangle$ in Gleichung (2.2.3) schreiben als

$$Z = \sum_{\sigma \in \Lambda} \exp\left(\beta \sum_{\langle u,v \rangle} \sigma(u)\sigma(v) \right) \tag{2.2.4}$$

$$\langle A \rangle = \frac{1}{Z} \sum_{\sigma \in \Lambda} A(\sigma) \exp\left(\beta \sum_{\langle u,v \rangle} \sigma(u)\sigma(v) \right) \tag{2.2.5}$$

wobei $\beta = \frac{J_c}{kT}$ die dimensionslose inverse Temperatur ist.

Für das Ising-Modell gibt es typische Erwartungswerte, die häufig bei Betrachtungen des Modells auftreten und auch für uns von Relevanz sind. Hierzu gehört die mittlere Magnetisierung $\langle m \rangle$, die durch

$$\langle m \rangle = \langle m(\sigma) \rangle = \left\langle \frac{1}{N} \sum_{u \in G} \sigma(u) \right\rangle \tag{2.2.6}$$

gegeben ist, mit σ eine Spinkonfiguration und $N = L_1 \cdot \ldots \cdot L_d$. Die mittlere Magnetisierung kann Werte zwischen -1 und 1 annehmen und gibt anschaulich gesehen die mittlere Ausrichtung eines Spins an. Beispielsweise wäre $\langle m \rangle = -0.8$ interpretierbar als 80% der Spins befinden sich im Mittel im Zustand -1. An gewissen Stellen dieser Arbeit wird auch der Erwartungswert $\langle |m| \rangle$ als mittlere Magnetisierung bezeichnet, was jedoch an den entsprechenden Stellen angegeben wird.

Basierend auf der Magnetisierung kann man die Suszeptibilität χ für den Fall , dass $h_u = h$ unabhängig von $u \in G^d_{(L_1,\ldots,L_d)}$ ist, definieren als

$$\chi = \partial_{h'}\langle m\rangle = \frac{1}{ZN}\left(\sum_{\sigma\in\Lambda} e^{-\frac{H(\sigma)}{kT}}\left(\sum_{u\in G}\sigma(u)\right)^2\right) \tag{2.2.7}$$

$$- \langle m\rangle\frac{1}{Z^2}\left(\sum_{\sigma\in\Lambda} e^{-\frac{H(\sigma)}{kT}}\left(\sum_{u\in G}\sigma(u)\right)\right) \tag{2.2.8}$$

$$= N(\langle m^2\rangle - \langle m\rangle^2) \tag{2.2.9}$$

wobei $h' = \beta h$ gewählt ist und auch als äußeres Feld bezeichnet wird. Somit ist dieses Definition identisch mit der in [5] und [8], worauf die Aussagen in den folgenden Abschnitten beruhen. Es ist auch möglich direkt nach h abzuleiten, was jedoch zu einer anderen Definition, die sich um den Faktor $\frac{1}{kT}$ unterscheidet, führt.

Bei der Betrachtung von Phasenübergängen im Rahmen des Ising-Modells tritt die sogenannte Korrelationsfunktion auf, die über

$$G(x) := \langle\sigma(x)\sigma(0)\rangle - \langle m\rangle^2 \tag{2.2.10}$$

definiert ist, siehe [8]. Qualitativ ausgedrückt, ist die Korrelationsfunktion ein Maß für die statistische Abhängigkeit des Spins am Gitterpunkt 0 mit dem am Punkt x. Falls $G(x) = 0$ sein sollte, ist dies gleichbedeutend damit, dass die beiden Spins statistisch unabhängig zueinander sind. Eine statistische Abhängigkeit liegt vor, falls $G(x) \neq 0$ ist.

Über $G(x)$ lässt sich die Korrelationslänge ξ über

$$\xi = \left(\frac{1}{2d}\frac{\sum_{x\in G}|x|^2 G(x)}{\sum_{x\in G} G(x)}\right)^{\frac{1}{2}} \tag{2.2.11}$$

definieren, siehe [8]. Hierbei handelt es sich nicht um die exponentielle Korrelationslänge ξ_{exp}, sondern um die sogenannte *second-moment correlation length*. Für die späteren Betrachtungen im Rahmen des Phasenübergangs des Ising-Modells, spielt die Wahl der Definition der Korrelationslänge keine Rolle.

Eine weitere Größe ist die Binderkumulante U, eingeführt in [1], die über

$$U = 1 - \frac{\langle m^4\rangle}{3\langle m^2\rangle^2} \tag{2.2.12}$$

definiert ist.

Wir wollen noch den Begriff des thermodynamischen Limes einführen. Für den Fall, dass $L_i = L$, für alle $i \in \{1,\ldots,d\}$ gilt, ist der thermodynamische Limes einer Observable $A(\sigma)$ gegeben als

$$\langle A\rangle_\infty = \lim_{L\to\infty}\langle A\rangle \tag{2.2.13}$$

falls der Grenzwert existieren sollte. Beim thermodynamischen Limes handelt es sich anschaulich um die Beschreibung eines Gitters bestehend aus unendlich vielen Gitterpunkten, das heißt ein beliebig ausgedehntes Gesamtsystem.

Es sei zu beachten, dass bei allen Definitionen der Größen eine nicht explizit angegebene Abhängigkeit des äußere Feld h' vorhanden ist. Falls im folgenden Betrachtungen des Ising-Modells, mit $h_u = h$ unabhängig von u, eine explizite Unterscheidung zwischen unterschiedlichen h' gemacht werden muss, wird dies an der entsprechenden Stelle angegeben.

2.2.2 Phasenübergänge

Im thermodynamischen Gleichgewicht werden physikalische Systeme im Rahmen der statistischen Physik über einen Satz voneinander unabhängig wählbarer thermodynamischer Größen $(A_1, \ldots, A_n)$, wie zum Beispiel Temperatur, Druck, Teilchenzahl, und so weiter beschrieben. In Abhängigkeit dieser Größen ergeben sich für das System die Erwartungswerte von Observablen, die für die thermodynamische Beschreibung des Systems verwendet werden. Für eine Reihe von Systemen zeigen Erwartungswerte bei bestimmten Kombinationen von $(A_{1,c}, A_{2,c}, \ldots, A_{n,c})$ eine markante Änderung, wie zum Beispiel einen nicht stetigen Sprung des Erwartungswertes. Qualitativ bedeutet das eine Änderung der Beschreibung der Physik vom System, da zum Beispiel Wechselwirkungen, die vorher keine Rolle gespielt haben, an diesem Punkt anfangen eine Relevanz zu besitzen. Solche Übergänge von einem thermodynamischen System werden als Phasenübergänge bezeichnet. Ein typisches Beispiel hierfür ist das Gefrieren von Wasser. In der Flüssigphase sind die einzelnen Wassermoleküle nicht an ihre Position gebunden und können prinzipiell durch die gesamte Phase durch diffundieren. Wird nun bei einem bestimmten Druck P die Temperatur T verringert, so geht das Wasser ab einer kritischen Temperatur T_c von der Flüssigphase in die feste Phase über. Die einzelnen Wassermoleküle ordnen sich dabei in einer regelmäßigen Struktur an und sind an ihren relativen Plätzen im Vergleich zu den anderen Molekülen gebunden. Qualitativ lässt sich das durch die Tatsache beschreiben, dass die Wechselwirkungen zwischen den Wassermolekülen bei Temperaturen $T < T_c$ die thermischen Bewegungen überkommen und eine kristalline Struktur ausbilden können. Bei diesem Übergang gibt es beispielsweise einen Sprung in der Entropie S des Systems bei der Temperatur T_c zu einem geringeren Wert in der festen Phase.

Eine quantitative Beschreibung von Phasenübergängen kann durch einen Ordnungsparameter $\psi(T, A_2, \ldots, A_n)$ erfolgen, wobei $T = A_1$ die Temperatur des Systemes gekoppelt am Wärmebad ist. Dieser Parameter ist eine thermodynamische Größe des betrachteten Systemes, die je nach System gewählt wird. Im Rahmen des Ising-Modells wird die Magnetisierung $\psi = \langle |m| \rangle_{\infty,h=0}$ beispielsweise als Ordnungsparameter verwendet.

Ein Ordnungsparameter ψ ist über

$$\psi(T, A_{2,c}, \ldots, A_{n,c}) \begin{cases} \neq 0, & \text{für} \quad T < T_c \\ = 0, & \text{für} \quad T \geq T_c \end{cases} \tag{2.2.14}$$

definiert, wobei $(T_c, A_{2,c}, \ldots, A_{n,c})$ der kritische Punkt ist, an dem der Phasenübergang geschieht.

Je nach Verhalten an dem kritischen Punkt $(T_c, A_{2,c}, \ldots, A_{n,c})$ des Ordnungsparameters ψ erfolgt eine Klassifizierung der Phasenübergänge. Wir sprechen von einem Übergang

1. Ordnung, falls der Ordnungsparameter an der kritischen Punkt nicht stetig sein sollte. Beispiel hierfür ist das nicht stetige Verhalten der Entropie bei der obigen Diskussion. Ein Phasenübergang 2. Ordnung liegt vor, falls der Ordnungsparameter ψ am kritischen Punkt $(T_c, A_{2,c}, \ldots, A_{n,c})$ stetig , jedoch mindestens eine seiner partiellen Ableitungen $(\partial_{A_i}\psi)(T_c, A_{2,c}, \ldots, A_{n,c})$ nicht stetig, beziehungsweise divergent, sein sollte, siehe [2] und [3]. Ein Beispiel eines solchen Phasenüberganges findet sich weiter unten für das Ising-Modell.

2.2.2.1 Phasenübergang für das zweidimensionale Ising-Modell

Für das zweidimensionale Ising-Modell ohne äußeres Feld existieren analytische Lösungen für unterschiedliche endliche Gitter und dem thermodynamischen Limes, die zuerst in [4] bestimmt worden sind. Betrachten wir uns das Ising-Modell mit $J = const.$, $J > 0$ und h unabhängig von den Gitterpunkten. In diesem Fall ist im thermodynamischen Limes im Grenzfall $h' \to 0^+$ die mittlere Magnetisierung

$$\psi = \langle |m| \rangle_{\infty,h=0} = \lim_{L\to\infty} \langle |m| \rangle_{h=0} = \lim_{h'\to 0^+} \lim_{L\to\infty} \langle m \rangle_{h'} \tag{2.2.15}$$

ein Ordnungsparameter und das System besitzt einen Phasenübergang 2. Ordnung bei der dimensionslosen inversen Temperatur β_c, die sich über die Bedingung

$$\sinh(2\beta_c)^2 = 1 \tag{2.2.16}$$

ergibt, für genauere Betrachtungen sei auf [4], [5] verwiesen.

Der konkrete Wert für β_c ergibt sich offensichtlich zu

$$\beta_c = \frac{\sinh^{-1}(1)}{2} \approx 0.4406868 \tag{2.2.17}$$

gerundet auf die angegebene letzte Nachkommastelle.

Für diesen Phasenübergang 2. Ordnung divergiert die Suszeptibilität und Korrelationslänge (siehe [5])

$$\chi_\infty = \lim_{h'\to 0^+} \lim_{L\to\infty} \partial_{h'} \langle m \rangle_{h'} = \lim_{L\to\infty} \underbrace{L^2(\langle m^2 \rangle_{h=0} - \langle |m| \rangle^2_{h=0})}_{=:\chi_L} \tag{2.2.18}$$

$$\xi_\infty = \lim_{h'\to 0^+} \lim_{L\to\infty} \xi_{h'} \tag{2.2.19}$$

am kritischen Punkt β_c.

Für das physikalische Verständnis des Phasenübergangs ist es interessant qualitativ den Verlauf der mittleren Magnetisierung $\langle |m| \rangle_{\infty,h=0}$ zu kennen. Diese ist für das zweidimensionale Ising-Modell bekannt, siehe [7], und gegeben durch

$$\langle |m| \rangle_{\infty,h=0} = \left(\frac{(1+x^2)(1-6x^2+x^4)^{\frac{1}{2}}}{(1-x^2)^2} \right)^{\frac{1}{4}} \tag{2.2.20}$$

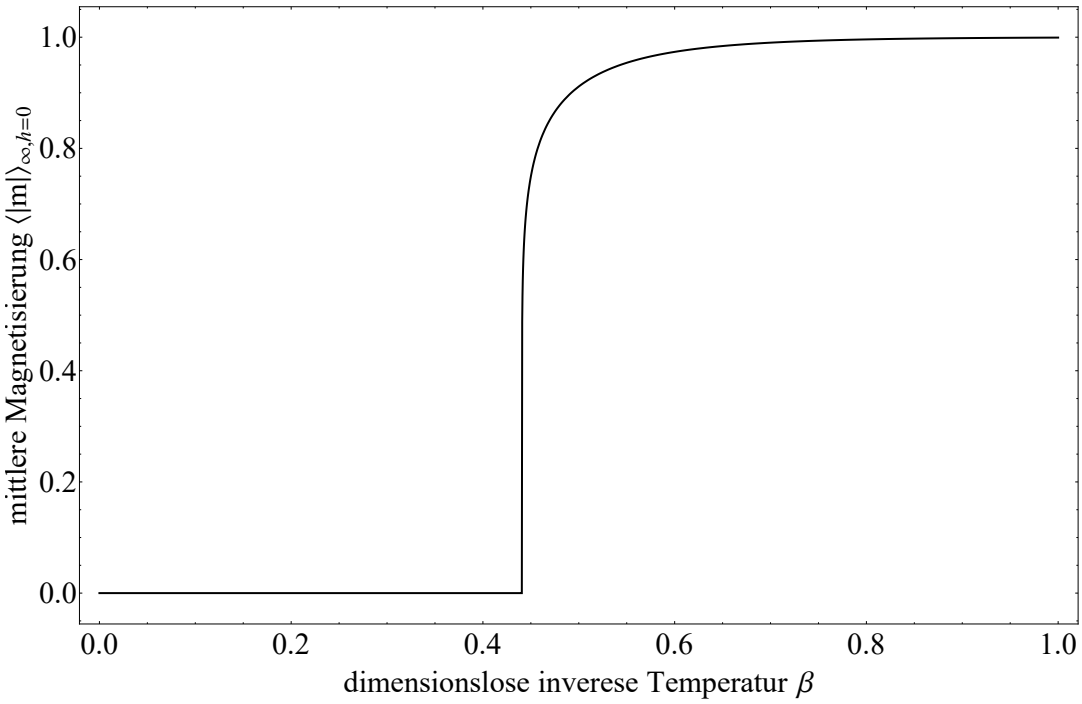


Abbildung 2.2.1: Die mittlere Magnetisierung $\langle|m|\rangle_{\infty,h=0}$ in Abhängigkeit der dimensionslosen inversen Temperatur β für das zweidimensionale Ising-Modell.

mit $x = e^{-2\beta}$, für $\beta > \beta_c$ und $\langle|m|\rangle_{\infty,h=0} = 0$ für $\beta \leq \beta_c$.

Eine Darstellung des Funktionsverlaufes von $\langle|m|\rangle_{\infty,h=0}$ ist in Abbildung 2.2.1 angegeben. In der Abbildung erkennen wir das physikalische Verhalten des Ordnungsparameters $\langle|m|\rangle_{\infty,h=0}$ im zweidimensionalen Ising-Modell im thermodynamischen Limes. In der Phase oberhalb der kritischen Temperatur $kT_c/J = \beta_c^{-1}$ ist der Ordnungsparameter null, anschaulich gibt es keine präferierte Ausrichtung der Spins und die Wechselwirkungen untereinander spielen, im Vergleich zu den Energieänderungen durch Wechselwirkung mit dem Wärmebad, eine untergeordnete Rolle. Wird jedoch die Temperatur verringert, so dass wir in der Phase unterhalb von T_c gelangen, so ändert sich dieses Verhalten plötzlich und die Wechselwirkungen zwischen den Spins werden nicht mehr vernachlässigbar im Vergleich zu der Wechselwirkung mit dem Wärmebad, so dass sich die Spins gegenseitig durch die Wechselwirkung beeinflussen und durch das vermehrte Ausrichtung in einer gemeinsamen Richtung die Kopplungsenergien die Gesamtenergie herabsetzen. Für $T = 0$, bzw. $\beta \to \infty$ wird $\langle|m|\rangle_{\infty,h=0} = 1$, dies bedeutet alle Spins sind in einer Richtung ausgerichtet.

Ein ähnliches Verhalten bei einem Phasenübergang 2. Ordnung ist bei der Bose-Einstein-Kondensation, zumindest für den dreidimensionalen Fall, zu finden, wo bei Unterschreiten einer bestimmten kritischen Temperatur $T_{B,c}$ die Besetzungszahl des bosonischen Grundzustandes zunimmt und bei $T = 0$ alle Teilchen sich im Grundzustand befinden. Der qualitative Verlauf in Abhängigkeit von T ist ähnlich zu Abbildung 2.2.1,

siehe [2].

2.2.2.2 Kritisches Verhalten an Phasenübergängen

In der Untersuchung von Phasenübergängen eines thermodynamischen Systemes wird häufig das Verhalten von unterschiedlichen Erwartungswerten an der kritischen Temperatur T_c, bei der ein Phasenübergang liegt, betrachtet. Hierbei zeigt sich häufig, dass unterschiedlichste thermodynamische Systeme ein ähnliches Verhalten am Phasenübergang, die sich beispielsweise in Potenzgesetzen mit selben Exponenten manifestieren, zeigen. Dieses Auftreten von ähnlichen Phasenübergangsverhalten wird auch unter dem Begriff Universalität geführt, [2]. Für das zweidimensionale Ising-Modell bei verschwindenden äußeren Feld, wie in Abschnitt 2.2.2.1, ergeben sich für den thermodynamischen Limes folgende Potenzgesetze in der Nähe des Phasenübergangs

$$\langle |m| \rangle_{\infty,h=0} \approx B(-t)^{\beta} \tag{2.2.21}$$

$$\chi_{\infty} \approx C^{-}(-t)^{-\gamma} \tag{2.2.22}$$

$$\xi_{\infty} \approx f^{-}(-t)^{-\nu} \tag{2.2.23}$$

wobei $t = \frac{T-T_c}{T_c} < 0$ ist und B, C^- und f^- Konstanten sind. Die kritischen Exponenten β, hier nicht die inverse Temperatur, γ und ν sind für das zweidimensionale Ising-Modell bei verschwindenden äußerem Feld exakt bekannt und betragen $\gamma = \frac{7}{4}$, $\beta = \frac{1}{8}$ und $\nu = 1$. Die Angaben und Notation stammen hierbei aus [8], wo auch andere solcher kritischen Exponenten und genauere Betrachtungen aufgeführt sind.

Eine Erklärung für die Universalität lässt sich mit Hilfe der Theorie der Renormierungsgruppe finden. An dieser Stelle soll keine genaue Behandlung der Renormierungstheorie erfolgen, hierfür wird auf [9], als mögliche Einführung, verwiesen.

2.2.2.3 Finite-Size-Scaling-Gesetze

Aufgrund der Tatsache, dass nummerische Simulationen vom Ising-Modell nur auf endlichen Spin-Gittern erfolgen können, ist es von Interesse einen Bezug zwischen dem thermodynamischen Limes, bei dem der Phasenübergang geschieht, und endlichen Gittern herzustellen. Dies kann entweder dadurch geschehen, dass wir die Gittergröße L bei gegebener Temperatur T vergrößern, um den thermodynamischen Limes in Gleichung (2.2.13) nachzuahmen, oder wir verwenden Finite-Size-Scaling-Gesetze, die einen funktionalen Bezug zwischen dem thermodynamischen Limes und dem endlichen Gitter für einen Erwartungswert, in der Nähe des Phasenüberganges, herstellen.

Für das d-dimensionale Ising-Modell mit periodischen Randbedingungen, $L_i = L$ für alle $i \in \{1, \ldots, N\}$ und $h = 0$, ergibt sich für einen Erwartungswert $S(t, L)$ des Systemes folgendes Verhalten (siehe [5],[8])

$$S(t, L) = L^{\frac{\varphi}{\nu}} \left(f_S \left(\frac{\xi_{\infty}}{L} \right) + \mathcal{O}(L^{-\omega}, \xi_{\infty}^{-\omega}) \right) \tag{2.2.24}$$

wobei für $t \approx 0$ die Beziehung

$$S_{\infty}(t) \approx C_s t^{-\varphi} \tag{2.2.25}$$

vorausgesetzt ist. Vergleichen wir diese Angabe mit dem vorherigen Abschnitt, so erkennen wir, dass an der Stelle von φ die kritischen Exponenten γ, β und ν für die entsprechenden Größen eingesetzt werden. Ferner gilt für das zweidimensionale Ising-Modell $\omega = 2$.

Wir erkennen, dass wir die Gleichung (2.2.24) über $\xi_\infty \propto t^{-\nu}$ ausdrücken können als:

$$S(t, L) = L^{\frac{\varphi}{\nu}} \left(f_S \left(tL^{\frac{1}{\nu}} \right) + \mathcal{O}(L^{-\omega}, \xi_\infty^{-\omega}) \right) \tag{2.2.26}$$

Für den Fall, dass $L, \xi_\infty \gg 1$, sodass die Annahme $\mathcal{O}(L^{-\omega}, \xi_\infty^{-\omega}) \approx 0$ begründet ist, können wir beispielsweise folgende Gesetzmäßigkeiten herleiten:

$$\chi_L(t) = L^2(\langle m^2 \rangle_{h=0} - \langle |m| \rangle_{h=0}^2) = g(tL^{\frac{1}{\nu}})L^{\frac{\gamma}{\nu}} \tag{2.2.27}$$

$$\langle |m| \rangle_{h=0}(t, L) = f(tL^{\frac{1}{\nu}})L^{\frac{\beta}{\nu}} \tag{2.2.28}$$

Es sei noch eine Anmerkung zu dem Finite-Size-Scaling-Gesetz in Gleichung (2.2.27) gegeben. Die Größe χ_L ist nicht die Suszeptibilität χ eines endlichen Gitters, gemäß Gleichung (2.2.7). Jedoch gilt das Finte-Size-Scaling-Gesetz nicht für χ. Diese Tatsache ist darin begründet, in welcher Reihenfolge die Grenzwerte in Gleichung (2.2.18) ausgeführt werden. In den unterschiedlichen Fällen erhalten wir unterschiedliche Resultate:

$$\chi_\infty = \lim_{h' \to 0^+} \lim_{L \to \infty} \partial_{h'} \langle m \rangle_{h'} = \lim_{L \to \infty} \underbrace{L^2(\langle m^2 \rangle_{h=0} - \langle |m| \rangle_{h=0}^2)}_{=:\chi_L} \tag{2.2.29}$$

$$\chi'_\infty = \lim_{L \to \infty} \lim_{h' \to 0^+} \partial_{h'} \langle m \rangle_{h'} = \lim_{L \to \infty} \underbrace{L^2(\langle m^2 \rangle_{h=0} - \langle m \rangle_{h=0}^2)}_{=:\chi} = \lim_{L \to \infty} L^2 \langle m^2 \rangle_{h=0} \tag{2.2.30}$$

Hierbei wird $\langle m \rangle_{h=0} = 0$ für ein endliches Gitter ausgenutzt, was sich durch die Symmetrie $H(\sigma) = H(-\sigma)$ ohne äußeres Feld, ergibt.

Die Größe χ_∞ ist jene, für die der kritische Exponent γ in Gleichung (2.2.22) auftaucht, während dies für χ'_∞ nicht der Fall ist, siehe [10]. Das Finite-Size-Scaling-Gesetz in Gleichung (2.2.27) basiert somit auf dem thermodynamischen Limes χ_∞. Dieser Unterschied zwischen der Verwendung von $\langle m \rangle_{h=0}$ und $\langle |m| \rangle_{h=0}$ ist zwar subtil, jedoch notwendig für genauere quantitative Betrachtungen des Exponenten γ in Rahmen von Simulationen. Dieses Problem wird beispielsweise in [11] erwähnt.

Im Rahmen der Theorie der Finite-Size-Scaling-Gesetze zeigt die Binderkumulante ein Verhalten, dass zu der Bestimmung von Phasenübergängen verwendet werden kann. Für die Binderkumulante eines d-dimensionalen endlichen Gitters mit der gleichen Ausdehnung L in allen Richtungen gilt

$$U(L, \beta_c) \propto L^{-d} \tag{2.2.31}$$

$$\lim_{L \to \infty} U(L, \beta_c) = U^* \tag{2.2.32}$$

falls $\langle |m| \rangle_{\infty, h=0}$ ein Ordnungsparameter ist mit einem Phasenübergang 2. Ordnung an der Stelle β_c. Dies bedeutet, dass Gitter mit unterschiedlichen L an einem Phasenübergang 2. Ordnung näherungsweise die selbe Binderkumulante besitzen, falls die Konvergenz bezogen auf L schnell genug sein sollte. Für das zweidimensionale Ising-Modell, wie in

Abschnitt 2.2.2.1, beträgt der Wert der Binderkumulante im thermodynamischen Limes $U^* \approx 0.61067$ (siehe [12]).

Die in diesem Kapitel besprochenen Grundlagen werden bei der Betrachtungen der in dieser Arbeit verwendeten Programme auftreten, da wir mit Hilfe der theoretischen Ergebnisse die Funktionalität der Programme verifizieren werden.

Kapitel 3

Monte-Carlo-Simulationen des Ising-Modells

In diesem Kapitel werden die Grundlagen von Monte-Carlo-Simulationen und ihre Anwendung für die Simulation des Ising-Modells gekoppelt an einem Wärmebad besprochen. Hierbei werden die wichtigsten Algorithmen für derartige Simulationen im Rahmen des Ising-Modells angegeben. Des weiteren werden die Algorithmen, auf den die Programme dieser Arbeit basieren werden an dieser Stelle detailliert dargestellt.

3.1 Grundlagen

Mit Hilfe von Automaten in Form von Computern lassen sich durch die Programmierung von nummerischen Verfahren physikalische Größe im Rahmen eines Modelles, bei gegebenen Anfangsbedingungen, bestimmen. Solche nummerische Analysen physikalischer Systeme werden häufig für Probleme angewandt, bei denen es keine analytische Betrachtungen gibt.

Dies kann beispielsweise schon in der klassischen nicht relativistischen Physik auftreten. Für ein Zweiteilchensystem mit Zentralkräften sind die Lösungen der Bewegungsgleichungen für den Fall von $\frac{1}{r^2}$-Kräften analytisch bekannt. Auf der anderen Seite gibt es schon für ein N-Teilchensystem mit $N > 2$ und keinen weiteren Zwangsbedingungen, keine analytischen allgemeinen exakten Lösungen der Bewegungsgleichungen. Jedoch ist es durchaus möglich mit Hilfe von Verfahren, wie das Runge-Kutta-Verfahren, derartige Bewegungsgleichungen nummerisch mit einer gewünschten Genauigkeit zu lösen und folglich eine Näherung der Phasenraumtrajektorie zu erhalten.

Im Bereich der statistischen Physik ist es vor allem von Interesse Erwartungswerte eines Systemes in Abhängigkeit der äußeren Variablen zu bestimmen. Speziell im kanonischen Ensemble ist man bemüht die Zustandssumme Z eines Systemes zu ermitteln, um daraus thermodynamische Größen abzuleiten oder den Erwartungswert in Gleichung (2.2.3) auszurechnen. Für einfache statistische Systeme, wie nicht wechselwirkende klassische oder quantenmechanische Vielteilchensysteme, existieren exakte analytische Lösungen. Auch für einfache Systeme mit Wechselwirkungen, wie das Ising-Modell für

$d = 1$ (siehe [13]) oder $d = 2$ existieren analytische exakte Lösungen. Jedoch für Dimensionen $d > 2$ des Ising-Modells gibt es nach aktuellem Stand nur nummerische Betrachtungen und Ergebnisse der Verhalten von thermodynamischen Größen.

Speziell betrachten wir uns in diesem Kontext die Simulation des d-dimensionalen Ising-Modells mit $J_c = const.$ und $J_c > 0$, $h = 0$ und periodischen Randbedingungen. Falls keine expliziten Angaben vorhanden sind, seien dies Voraussetzungen für die Betrachtungen in diesem Kapitel. Mit Simulation ist gemeint, dass für eine gegebene dimensionslose inverse Temperatur β den Erwartungswert in Gleichung (2.2.5) einer Observablen $A(\sigma)$, mit Hilfe eines nummerischen Verfahrens, zu berechnen.

Zunächst scheint dieses Problem einfach gelöst, in dem für eine Obeservable $A(\sigma)$ alle Spinkonfigurationen $\sigma \in \Lambda^d_{(L_1,\ldots,L_d)}$ durchgegangen werden und die Werte $A(\sigma)$ und $e^{-\frac{1}{kT}H(\sigma)}$ erfasst und somit die Zustandssumme Z als auch die Summe in Gleichung (2.2.5) berechnet wird, um $\langle A \rangle$ zu bestimmen. Dieses Verfahren lässt sich relativ leicht implementieren, indem ein Array im Speicher angelegt wird, und die Einträge entweder den Wert -1 oder 1 jeweils annehmen können, was die möglichen Zustände eines Gitterpunktes repräsentiert. Anschließend werden die einzelnen Einträge in einer festgelegten Reihenfolge zwischen den Wert -1 und 1 geändert, so dass eine Folge σ_n von Spinkonfigurationen entsteht, wobei jede Konfiguration genau einmal vorkommt. Offensichtlich terminiert dieses Verfahren nach endlich vielen Schritten und liefert ein Resultat für $\langle A \rangle$, dessen Genauigkeit nur durch die endliche Genauigkeit der verwendeten Rechenmaschine begrenzt wird.

Der Algorithmus liefert jedoch keine praktikable Lösung für die nummerische Untersuchung vom Ising-Modell, denn die Anzahl an Spinkonfigurationen, die schon für kleine Gitter durchgegangen werden müssten, führt zu einer Laufzeit, die nicht vertretbar ist. Als Beispiel betrachten wir uns ein Gitter mit $d = 2$ und $L_1 = 10$ und $L_2 = 10$. Es existieren insgesamt $2^{10 \cdot 10}$ Spinkonfigurationen. Ferner gehen wir von der unrealistischen Annahme aus, dass der verwendete Computer über eine CPU verfügt, die innerhalb eines Taktzyklus eine Spinkonfiguration erzeugen und die Messungen von $A(\sigma)$ und $e^{\frac{1}{kT}H(\sigma)}$ ausführen kann. Für den Taktfrequenz f_{CPU} der CPU können wir den realistischen Wert von $f_{CPU} = 3$ GHz annehmen. Die Gesamtzeit t der Simulation beträgt unter den gegeben Annahmen

$$t = \frac{2^{100}}{3 \text{ GHz}} \approx 1.3 \cdot 10^{13} \text{ Jahre} \tag{3.1.1}$$

Folglich muss eine andere Methode zur Simulation des Ising-Modells benutzt werden. Insbesondere wenn die Tatsache berücksichtigt wird, dass, mit Hilfe der Programme dieser Arbeit, Gitter mit der Größe 64×64 oder größer simuliert werden.

In so einem Fall gibt es nun zwei Möglichkeiten. Entweder wird der Algorithmus optimiert, indem gewisse mathematische Vorarbeiten geleistet werden, die die Laufzeit reduzieren, oder es wird auf Monte-Carlo-Simulationen zurück gegriffen.

Monte-Carlo-Simulationen basieren auf der Idee, mit Hilfe eines statistischen Prozesses, einen Wert Q zu berechnen, der sich über

$$Q = \sum_{x \in M} g(x) = \sum_{x \in M} P(x) \cdot \tilde{g}(x), \text{ mit } \sum_{x \in M} P(x) = 1 \text{ und } P(x) \geq 0 \tag{3.1.2}$$

ergibt, wobei M eine beliebige Menge ist und $g : M \to \mathbb{R}$ gilt. Die Funktion P lässt sich als eine Wahrscheinlichkeit für das Auftreten des Elementes $x \in M$ interpretieren. Angenommen wir hätten einen statistischen Prozess der gerade Elemente $x \in M$ mit der Wahrscheinlichkeit $P(x)$ realisiert, so wissen wir im Rahmen der Stochastik, dass für eine Messreihe mit N Realisierungen x_i aus der Menge M folgende Gesetzmäßigkeit gilt:

$$Q = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{N} \tilde{g}(x_i) \tag{3.1.3}$$

Anders ausgedrückt, dass arithmetische Mittel konvergiert, im Sinne des Konvergenzbegriffes der Gesetz der Großen Zahlen, gegen unseren Wert Q. Somit können wir für eine ausreichend große Messreihe den Wert Q über einen statistischen Prozess nähern. Wir geben jedoch bei Monte-Carlo-Simulationen die Fähigkeit auf eine Garantie für die Genauigkeit des nummerischen Resultates zu haben und müssen für quantitative Untersuchungen entsprechend statistische Fehler bestimmen.

Wir können erkennen, dass Monte-Carlo-Simulationen auch für die Simulation des Ising-Modells verwendet werden können. Hierzu erhalten wir mit einem Vergleich von Gleichung (2.2.3) und (3.1.2):

$$M = \Lambda^d_{(L_1,\ldots,L_d)},\ P(\sigma) = \frac{1}{Z} e^{-\frac{1}{kT}H(\sigma)} \text{ und } \tilde{g}(\sigma) = A(\sigma) \tag{3.1.4}$$

Um die Simulation des Ising-Modells praktisch umzusetzen müssen Spinkonfigurationen σ mit der Wahrscheinlichkeit $P(\sigma)$ erzeugt werden, was jedoch mit dem Problem verbunden ist, dass die Zustandssumme Z nicht bekannt ist. Dieses Problem kann mit Hilfe von Markov-Ketten gelöst werden.

3.1.1 Markov-Kette

Zunächst wollen wir den Begriff der Markov-Kette definieren:

Definition 3.1.1 (Markov-Kette, bzw. Markov-Prozess)**.**
Sei M eine Menge und $W : M \times M \to [0,1]$, wobei wir die Notation $W(x,y) = W(x \to y)$ verwenden. Falls

$$\sum_{y\in M} W(x \to y) = 1 \tag{3.1.5}$$

gilt, so bezeichnen wir W als die Übergangswahrscheinlichkeit von x nach y und (M, W) als Markov-Prozess, bzw. Markov-Kette und eine endliche Folge $(x_1, \ldots, x_n) \in M^n$ bezeichnen wir als Kette mit der Wahrscheinlichkeit

$$W^n(x_1, \ldots, x_n) := W(x_1 \to x_2)W(x_2 \to x_3) \cdot \ldots \cdot W(x_{n-1} \to x_n) \tag{3.1.6}$$

und

$$W^n(x_1 \to x_n) := \sum_{(x_2,\ldots,x_{n-1})\in M^{n-2}} W^n(x_1, x_2, \ldots, x_{n-1}, x_n) \tag{3.1.7}$$

Anschaulich beschreibt eine Markov-Kette (M, W) den statistischen Prozess beginnend bei einem Element $x_0 \in M$ mit der Wahrscheinlichkeit $W(x_n \to x_{n+1})$ ein Element x_{n+1} zu erhalten, falls das vorherige Elemente x_n war.

Für die Monte-Carlo-Simulation des Ising-Modells gibt es folgenden wichtigen Satz (siehe [6] und [15]):

Satz 3.1.1.
Sei (M, W) ein Markov-Prozess, M endlich, $P_F(x) := \frac{1}{Z}e^{-f(x)}$ für $x \in M$, $Z \in \mathbb{R}^+$ und $P_I : M \to [0, 1]$ eine Verteilung, mit

$$\sum_{x \in M} P_F(x) = 1 \text{ und } \sum_{x \in M} P_I(x) = 1 \tag{3.1.8}$$

Falls die Aussagen

1. *Ergodizität: Sei $x \in M$, so gilt*

$$\forall y \in M : \exists n > 0, W^n(x \to y) > 0 \tag{3.1.9}$$

2. *Stabilität: Sei $x \in M$, so gilt*

$$\sum_{y \in M} W(y \to x)e^{-f(y)} = e^{-f(x)} \tag{3.1.10}$$

gelten sollten, folgt für

$$P_n(x) := \sum_{k \in M} W(k \to x)P_{n-1}(k), \text{ mit } P_0 = P_I \text{ und } x \in M \tag{3.1.11}$$

die Aussage

$$||P_n - P_F|| := \sum_{x \in M} |P_n(x) - P_F(x)| \longrightarrow 0 \ (n \to \infty) \tag{3.1.12}$$

wobei $||P_n - P_F||$ in Abhängigkeit von n streng monoton fällt.

Beweis. Wir wollen an dieser Stelle keinen expliziten Beweis angeben. Für genauere Betrachtungen sei auf [15] verwiesen. □

Wir erkennen, dass dieser Satz die Möglichkeit liefert die gewünschte Verteilung $P(\sigma)$ in Gleichung (3.1.4) in einem Verfahren zu implementieren ohne die Zustandssumme Z explizit zu kennen. Hierzu benötigen wir einen Markov-Prozess $(\Lambda^d_{(L_1,\ldots,L_d)}, W)$ für den die beiden Bedingungen in Satz 3.1.1 gelten, mit $f(\sigma) = \frac{1}{kT}H(\sigma)$. Nun wählen wir eine Anfangskonfiguration σ_0 aus. Auf Basis dieser Konfiguration wird über die Wahrscheinlichkeitsverteilung $W(\sigma_0 \to \sigma_1)$ die nächste Konfiguration σ_1 bestimmt und aus dieser die Konfiguration σ_2 und so weiter. Anders ausgedrückt aus der n-ten Spinkonfiguration σ_n wird die Konfiguration σ_{n+1} zufällig mit der Wahrscheinlichkeit $W(\sigma_n \to \sigma_{n+1})$ bestimmt. Bei n solcher Schritten erhalten wir eine Kette $(\sigma_0, \ldots, \sigma_n) \in M^{n+1}$. Gemäß

der Gleichung (3.1.12) des Satzes 3.1.1 erwarten wir somit, für immer größere n, dass σ_n immer besser mit der Wahrscheinlichkeit $P(\sigma_n)$ verteilt ist. In einer praktischen Umsetzung wird typischerweise vor den eigentlichen Messungen von $A(\sigma)$ eine Anzahl n an Spinkonfigurationen $(\sigma_0, \ldots, \sigma_{n-1})$ erzeugt und diese für die Messungen verworfen, jedoch entschieden, dass die nächste Konfigurationen beginnend ab σ_n ausreichend der Verteilung P folgen.

Bei vielen Algorithmen wird für die Übergangswahrscheinlichkeit W nicht die Stabilitätsbedingung geprüft, sondern das sogenannte *detailed balance*, was durch

$$\frac{W(x \to y)}{W(y \to x)} = \frac{e^{-f(y)}}{e^{-f(x)}} \tag{3.1.13}$$

gegeben ist, wobei $x, y \in M$. Falls das *detailed balance* erfüllt sein sollte, ergibt sich über

$$\sum_{x \in M} W(x \to y) e^{-f(x)} = \sum_{x \in M} W(y \to x) e^{-f(y)} = e^{-f(y)} \tag{3.1.14}$$

unmittelbar die Stabilitätsbedienung.

3.1.2 Zufallsgeneratoren

Algorithmen im Rahmen von Monte-Carlo-Simulationen verwenden Zufallszahlen um die statistischen Prozesse zu realisieren. Mit Hilfe von Zufallsgeneratoren können solche zufälligen Zahlenfolgen erzeugt werden. Typischerweise werden Generatoren verwendet, die Zufallszahlen in einem Intervall $[a_1, a_2]$ gleich verteilt erzeugen.

Zufallsgeneratoren können entweder echte Zufallszahlen liefern, indem sie entsprechend an einem physikalischen Aufbau mit einer statistisch verteilten Größe gekoppelt sind, beispielsweise das Rauschen von elektronischen Bauteilen, oder Pseudozufallszahlen, die vollständig durch einen deterministischen Algorithmus erzeugt werden.

Die Erzeugung von echten Zufallszahlen ist im Allgemeinen aufwendiger und langsamer, als die Erzeugung von Pseudozufallszahlen, womit für Monte-Carlo-Simulationen vorwiegend letztere benutzt werden.

Die Idee hinter Pseudozufallszahlen ist es eine Zahlenfolgen $(x_n)_{n \in \mathbb{N}_0} \subset [a_1, a_2]$ zu definieren, die die Realisierungen einer gleich verteilten, oder anders verteilten, Zufallsvariablen X auf der Menge $[a_1, a_2]$ nachahmt. Hiermit ist gemeint, dass die Zahlen $x_0, x_1, x_2, \ldots$ für statistische Größen, wie Mittelwerte, Ergebnisse liefern, die auch ein echter Zufall auf Basis der Zufallsvariable X liefern würde.

Dementsprechend existieren statistische Test für Zufallsgeneratoren, um die Qualität der Pseudozufallszahlen zu beurteilen. Für unterschiedliche Monte-Carlo-Simulationen kann es sein das gewisse Zufallsgeneratoren geeignet sind, während andere ungeeignet sind. Für eine genauere Diskussion von Zufallszahlen sei auf [14] verwiesen.

In dieser Arbeit werden zwei Arten von Zufallsgeneratoren verwendet. Die erste Art ist ein linearer Kongruenzgenerator, dessen Zahlenfolge x_n über

$$x_{n+1} = (a \cdot x_n + c) \bmod m \tag{3.1.15}$$

definiert ist. Wobei in diesem Fall die Variationen

$$\textbf{1. Variation:}\ a = 16807,\ c = 0,\ m = 2^{31} - 1$$

$$\textbf{2. Variation:}\ a = 48721,\ c = 0,\ m = 2^{31} - 1$$

verwendet werden, siehe [17].

Die zweite Art von Zufallsgeneratoren ist ein Marsaglia-Zaman-Generator, genauer der Substract-with-borrow-Generator, vorgestellt in [16]. Dieser ist in [16] über

$$f(x_1, \ldots, x_r, c) := \begin{cases} (x_2, \ldots, x_r, x_{r+1-s} - x_1 - c, 0), & \text{für} \quad x_{r+1-s} - x_1 - c \geq 0 \\ (x_2, \ldots, x_r, x_{r+1-s} - x_1 - c + b, 1), & \text{für} \quad x_{r+1-s} - x_1 - c < 0 \end{cases} \tag{3.1.16}$$

definiert, wobei $f : \mathbb{N}_0^{r+1} \to \mathbb{N}_0^{r+1}$.

Der Generator arbeitet rekursiv. Dies bedeutet bei $(x_1^{(0)}, \ldots, x_r^{(0)}, c^{(0)}) \in \{0, \ldots, b\}^r \times \{0, 1\}$ beginnend ergibt sich im n-ten Schritt aus $(x_1^{(n)}, \ldots, x_r^{(n)}, c^{(n)}) \in \mathbb{N}_0^{r+1}$ der $(n+1)$-te Schritt über

$$(x_1^{(n+1)}, \ldots, x_r^{(n+1)}, c^{(n+1)}) = f(x_1^{(n)}, \ldots, x_r^{(n)}, c^{(n)}) \tag{3.1.17}$$

Hierbei ist die Zufallszahl, die durch den Generator erzeugt wird, im n-ten Schritt $x_r^{(n)} \in \{0, \ldots, b\}$ und gleich verteilt.

Für diese Arbeit ist $b = 2^{32} - 5$, $r = 43$ und $s = 22$ gewählt. Für genauere Betrachtungen sei auf [16] verwiesen.

3.2 Algorithmen für das Ising-Modell

Für die Simulation des Ising-Modells sind im Laufe der Zeit unterschiedliche Algorithmen entstanden. An dieser Stelle folgende zwei Verfahren, die im Rahmen der Programme dieser Arbeit implementiert werden. Die genauen Details der Implementationen, insbesondere die benutzten Multispin-Coding Methoden werden in Kapitel 5 diskutiert. Alle Algorithmen die hier vorgestellt werden basieren auf den Grundlagen von Markov-Prozessen für die der Satz 3.1.1 gilt. Dies bedeutet, aus der aktuellen Spinkonfiguration σ_n wird mit Hilfe einer Übergangswahrscheinlichkeit $W(\sigma_n \to \sigma_{n+1})$ eine neue Spinkonfiguration σ_{n+1} konstruiert. Wir werden den Vorgang der Erzeugung einer Spinkonfiguration σ_{n+1} aus einer Konfiguration σ_n ab dieser Stelle als Update des Spin-Gitters, bzw. der Spinkonfiguration, bezeichnen.

3.2.1 Metropolis-Algorithmus

Beim Metropolis-Algorithmus wird innerhalb eines Updates der aktuellen Spinkonfiguration σ für eine Spin des Spin-Gitters entschieden, ob dieser geflippt werden soll oder nicht. Die Übergangswahrscheinlichkeit für den Gitterpunkt $(x_1, \ldots, x_d) \in G^d_{(L_1, \ldots, L_d)}$ sieht dabei wie folgt aus (siehe [6])

$$W_{(x_1, \ldots, x_d)}(\sigma \to \sigma') = \min\left(1, e^{-\frac{1}{kT}(H(\sigma') - H(\sigma))}\right) \tag{3.2.1}$$

$$W_{(x_1, \ldots, x_d)}(\sigma \to \sigma) = 1 - W_{(x_1, \ldots, x_d)}(\sigma \to \sigma') \tag{3.2.2}$$

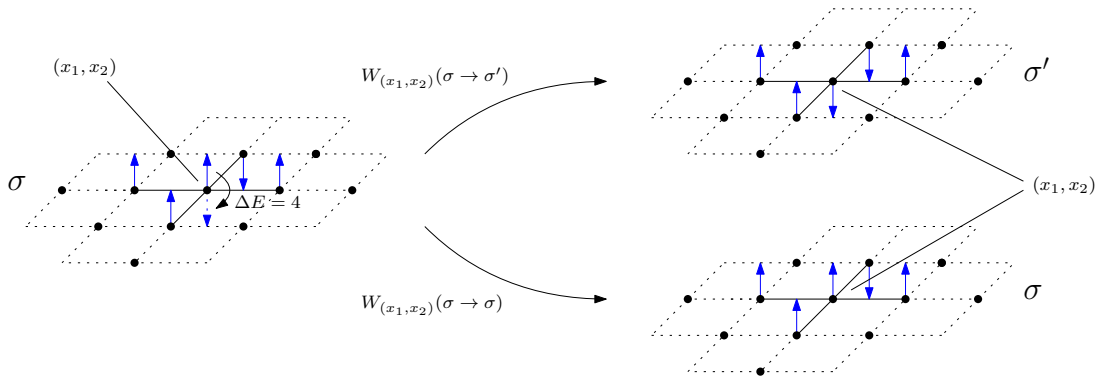


Abbildung 3.2.1: Illustration des Metropolis-Algorithmus für $d = 2$. Hierbei gilt $\Delta E = H(\sigma') - H(\sigma)$.

wobei σ' eine Spinkonfiguration ist für die

$$\forall x \in G^d_{(L_1,\ldots,L_d)} \backslash \{(x_1, \ldots, x_n)\} : \sigma'(x) = \sigma(x) \tag{3.2.3}$$

und

$$\sigma'(x_1, \ldots, x_d) = -\sigma(x_1, \ldots, x_d) \tag{3.2.4}$$

gilt. Dieser Vorgang ist für das zweidimensionale Ising-Modell in Abbildung 3.2.1 illustriert, wobei nur ein Ausschnitt des Gitters gezeigt wird mit der Darstellungsmethode aus Kapitel 2.

Um die Eigenschaft der Ergodizität in Satz 3.1.1 zu erfüllen werden nacheinander für die einzelnen Gitterpunkte $x \in G^d_{(L_1,\ldots,L_d)}$ die Updates ausgeführt. Dies kann in einer beliebigen Reihenfolge geschehen, solange garantiert ist, dass jeder Gitterpunkt innerhalb einer endlichen Anzahl an Schritten für ein Update der Konfiguration ausgewählt wird.

Ferner erkennen wir offensichtlich, dass die *detailed balance* Eigenschaft für die Wahrscheinlichkeit $W_{(x_1,\ldots,x_d)}$ erfüllt ist, da aus

$$\frac{W_{(x_1,\ldots,x_d)}(\sigma \to \sigma')}{W_{(x_1,\ldots,x_d)}(\sigma' \to \sigma)} = \frac{\min\left(1, e^{-\frac{1}{kT}(H(\sigma')-H(\sigma))}\right)}{\min\left(1, e^{-\frac{1}{kT}(H(\sigma)-H(\sigma'))}\right)} \tag{3.2.5}$$

und $H(\sigma') - H(\sigma) = -(H(\sigma) - H(\sigma'))$ sofort

$$\frac{W_{(x_1,\ldots,x_d)}(\sigma \to \sigma')}{W_{(x_1,\ldots,x_d)}(\sigma' \to \sigma)} = e^{-\frac{1}{kT}(H(\sigma')-H(\sigma))} = \frac{e^{-\frac{1}{kT}H(\sigma')}}{e^{-\frac{1}{kT}H(\sigma)}} \tag{3.2.6}$$

folgt, siehe [5]

3.2.2 Demon-Algorithmus

Ein *demon* (englischer Begriff für Dämon) ist ein System, das sich jeweils in einem von k möglichen Zuständen mit der Energie $d_i \in \{d_1, \ldots, d_k\}$ befindet. Ein Demon-System

besteht dabei aus m unabhängigen *demons*, sodass die Hamiltonfunktion durch

$$H_D(d_{i_1}, \ldots, d_{i_m}) = \sum_{j=0}^{m} d_{i_j} \tag{3.2.7}$$

gegeben ist.

Wir betrachten nun ein Gesamtsystem bestehend aus einem d-dimensionalen Spin-System im Rahmen des Ising-Modells mit der Hamiltonfunktion H_S und einem Demon-System, bestehend aus m unabhängigen *demons*, mit der Hamiltonfunktion H_D. Hierbei ergibt sich die Hamiltonfunktion H des Gesamtsystemes zu:

$$H(\sigma, d_{i_1}, \ldots, d_{i_m}) = H_S(\sigma) + H_D(d_{i_1}, \ldots, d_{i_m}) \tag{3.2.8}$$

Eine Kopplung dieses Gesamtsystemes an ein Wärmebad führt zu der Verteilung

$$P(\sigma, d_{i_1}, \ldots, d_{i_m}) = \frac{1}{Z} e^{-\frac{1}{kT} H(\sigma, d_{i_1}, \ldots, d_{i_m})} = \frac{1}{Z_S} e^{-\frac{1}{kT} H_S(\sigma)} \cdot \frac{1}{Z_D} e^{-\frac{1}{kT} H_D(d_{i_1}, \ldots, d_{i_m})} \tag{3.2.9}$$

wobei Z_S die Zustandssumme des Spins-Systems und Z_D die des Demon-Systems ist. Für eine Observable $A(\sigma)$ ergibt sich somit

$$\begin{aligned} \langle A \rangle &= \sum_{\sigma \in G} \sum_{(d_{i_1}, \ldots, d_{i_m})} \frac{1}{Z_S} A(\sigma) e^{-\frac{1}{kT} H_S(\sigma)} \cdot \frac{1}{Z_D} e^{-\frac{1}{kT} H_D(d_{i_1}, \ldots, d_{i_m})} \\ &= \sum_{\sigma \in G} \frac{1}{Z_S} A(\sigma) e^{-\frac{1}{kT} H_S(\sigma)} \end{aligned} \tag{3.2.10}$$

was äquivalent zum Erwartungswert in Gleichung (2.2.3) ist. Folglich können wir die selben Erwartungswerte erhalten, indem wir anstatt das Ising-Modell zu simulieren, das Gesamtsystem, bestehend aus dem Spin-Gitter und den *demons*, an einem Wärmebad gekoppelt, simulieren. Das Konzept der Dämonen führt zu zusätzlichen, statistisch voneinander unabhängigen, Freiheitsgrade. Grundlegende Betrachtungen zu Dämonen im Rahmen des Ising-Modells sind in [18] und [19] erarbeitet.

Beim Demon-Algorithmus geht es um die Simulation des Gesamtsystems, gekoppelt an einem Wärmebad. Hierfür kann ein Algorithmus gemäß [20] und [21] konstruiert werden.

Gemäß Gleichung (3.2.9) können wir schreiben

$$P(E_S, E_D) = \frac{1}{Z} n_S(E_S) n_D(E_D) e^{-\beta(E_S + E_D)} \tag{3.2.11}$$

wobei n_S die Zustandsdichte des Spin-Systems und n_D die Zustandsdichte des Demon-Systems ist.

Im mikrokanonischen Fall folgt für die Wahrscheinlichkeitsverteilung mit der Gesamtenergie $E = E_S + E_D$

$$P_{M,E}(E_S, E_D) = \frac{1}{\Omega(E)} n_S(E_S) n_D(E_D) \tag{3.2.12}$$

woraus sich für die Wahrscheinlichkeit $P_{M,E}(E_S)$ des Spins-Systemes mit der Energie E_S im mikrokanonischen Ensemble der Ausdruck

$$P_{M,E}(E_S) = \frac{1}{\Omega(E_T)} \left(\sum_{E'_D} n_D(E'_D) \right) n_S(E_S) \tag{3.2.13}$$

ergibt. Ferner führen wir die bedingte Wahrscheinlichkeit

$$P_{D,E_S}(E_D) := \frac{1}{Z_D} n_D(E_D) e^{-\beta(E_S+E_D)} \tag{3.2.14}$$

ein. Dies entspricht offensichtlich einer kanonischen Beschreibung des Demon-Systems.

Somit ergibt sich für die Wahrscheinlichkeit das Spin-System mit der Energie E_S , gemäß der mikrokanonischen Verteilung, und das Demon-System mit der Energie E_D unter der Bedingung, dass E_S vorliegt, vorzufinden zu:

$$\begin{aligned} P_{M,E}(E_S)P_{D,E_S}(E_D) &= \frac{1}{\Omega(E_T)} \frac{1}{Z_D} \left(\sum_{E'_D} n_D(E'_D) \right) n_S(E_S) n_D(E_D) e^{-\beta(E_S+E_D)} \\ &\propto P(E_S, E_D) \end{aligned} \tag{3.2.15}$$

Folglich erkennen wir, dass wir die Simulation des Gesamtsystemes aufteilen können in eine mikrokanonische Simulation des Gesamtsystemes und einem kanonischen Update der *demons*. Dies bedeutet ein Update der Gesamtkonfiguration besteht aus zwei Schritten. Zunächst wird ein mikrokanonischer Update ausgeführt und anschließend ein kanonischer Update der *demons*, wozu beide Schritte Markov-Prozesse sein müssen, die jeweils die entsprechenden Bedingungen aus dem Satz 3.1.1 erfüllen, wobei für den mikrokanonischen Fall entsprechend eine Gleichverteilung anstatt des Boltzmann-Faktors verwendet wird. Für genauere Betrachtungen sei auf [20] verwiesen.

Um in der Lage zu sein den folgenden Algorithmus anzugeben, müssen die Energien und Eigenschaften der *demons* genauer spezifiziert werden. Das Demon-System bestehe aus insgesamt $N = L_1 \cdot \ldots \cdot L_d$ Dämonen, wobei jedem Gitterpunkt ein *demon* zugeordnet sei. Die Energien eines *demon* sei $\tilde{d}_i \in \{0, 4, 8, 12, 16, \ldots, \tilde{d}_{max}\}$, wobei es eine maximale Energie $\tilde{d}_{max}$ gebe, und

$$\tilde{d}_i := \frac{d_i}{J_c} \tag{3.2.16}$$

gelte.

Aufgrund der Tatsache, dass die Energien der Dämonen ein Vielfaches von 4 sind, können wir uns vorstellen, dass jeder *demon* Energieniveaus besitzt, die entweder besetzt oder nicht besetzt sind, und die Kombination der besetzten und nicht besetzen Niveaus die Gesamtenergie $\tilde{d}_i$ des *demon* bestimmt. Hierbei ergibt sich die Energie E_i des i-ten Niveaus zu $E_i = 4^i$, $i \geq 1$. Eine Illustration solcher Energieniveaus für ein *demon* ist in Abbildung 3.2.2 vorzufinden.

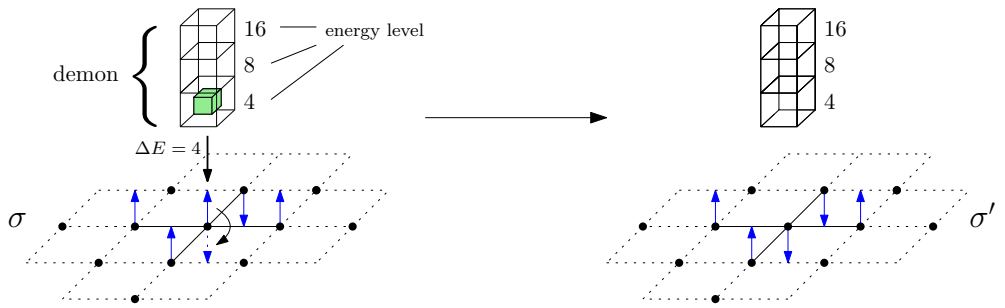


Abbildung 3.2.2: Illustration des Demon-Algorithmus für $d = 2$ und $\tilde{d}_{max} = 16$. Hierbei gilt $\Delta E = \frac{1}{J_c}(H(\sigma') - H(\sigma))$. Der grün gefärbte Würfel stellt die Besetzung des Energieniveaus $E_1 = 4$ dar.

3.2.2.1 Mikrokanonischer Update des Gesamtsystemes

Der mikrokanonische Update des Spin-Gitters besteht aus zwei Schritten. Im ersten Schritt wird zwischen dem Spin-System und Demon-System ein Energieaustausch, bei konstanter Gesamtenergie $E = E_S + E_D$ und im zweiten Schritt eine Verschiebung der Dämonen an neue Positionen ausgeführt.

Der Energieaustausch geschieht lokal für jeden einzelnen Spin-Demon-Paar, wobei die Reihenfolge in der die Spin-Demon-Paare abgearbeitet werden nicht festgelegt ist. Analog zum Metropolis-Algorithmus wird für die aktuelle Spinkonfiguration σ ein Gitterpunkt ausgesucht und vorgeschlagen den Spin zu flippen. Eine Änderung des Spins führt zu einer neuen Konfiguration σ' und einer Änderung der Energie $\Delta E = \frac{1}{J_c}(H_S(\sigma') - H_S(\sigma))$ des Spin-Systemes. Der Vorschlag wird akzeptiert, falls der dem Gitterpunkt zugeordnete *demon* die Energieänderung durch eine Energieabgabe von ΔE kompensieren kann und somit die Gesamtenergie E erhalten bleibt. Anders ausgedrückt, muss $\Delta E \in [0, \tilde{d}_{max}]$ gelten. Für den Fall, dass der Vorschlag akzeptiert wird, geht σ in σ' über und die neue Energie $\tilde{d}'_i$ des *demon* ergibt sich zu $\tilde{d}'_i = \tilde{d}_i - \Delta E$. Falls der Vorschlag nicht akzeptiert wird bleibt die Spinkonfiguration σ erhalten und die Energie des *demon* unverändert, [5]. Der hier dargestellte Energieaustausch eines Spin-Demon-Paares ist in Abbildung 3.2.2 beispielhaft illustriert.

Nachdem die lokalen Updates ausgeführt sind werden den einzelnen Dämonen neue Gitterpunkte zugeordnet, damit für die mikrokanonische Simulation die Ergodizität weitgehend garantiert werden kann.

3.2.2.2 Kanonisches Update des Demon-Systems

Für das kanonische Update der *demon* gibt es mindestens zwei Möglichkeiten dies zu bewerkstelligen.

Zunächst erkennen wir, dass die einzelnen *demons* zueinander und ihre Energieniveaus statistisch unabhängig sind. Die erste Methode ist die naive, wo jedes einzelne Niveau E_i

eines *demon* mit der Wahrscheinlichkeit

$$P^{(1)}(E_i) = \frac{1}{Z^{(1)}} e^{-\beta E_i} = \frac{e^{-\beta E_i}}{1 + e^{-\beta E_i}} \tag{3.2.17}$$

besetzt oder mit der Gegenwahrscheinlichkeit $1 - P^{(1)}(E_i)$ nicht besetzt wird. Offensichtlich müssten in diesem Fall alle *demons* durchgegangen werden, was zu einer Komplexität von $\mathcal{O}(N)$ des Verfahren zum Updaten der Dämonen führt.

Die andere Methode basiert auf der Idee, nacheinander für alle *demons* ein bestimmtes Energieniveau E_i upzudaten. Für das Updaten des i-ten Niveaus wird zunächst die Anzahl n an Dämonen bestimmt, bei denen das Niveau E_i besetzt ist. Anschließend wird über eine Binomialverteilung

$$B(n', E_i) = \binom{N}{n'} (P^{(1)}(E_i))^{n'} (1 - P^{(1)}(E_i))^{N-n'} \tag{3.2.18}$$

eine neue Anzahl n' an besetzten Zuständen des Niveaus E_i bestimmt. Es werden anschließend die Energieniveaus E_i von $|n - n'|$ besetzten *demons* entfernt, falls $n - n' < 0$ gelten sollte, oder von $|n - n'|$ nicht besetzten *demons* besetzt, falls $n - n' > 0$ gelten sollte, [21]. Der Wert $|n - n'|$ liegt hierbei in der Größenordnung der Standardabweichung und wächst mit $\sqrt{n'}$.

Kapitel 4

GPU-Programmierung

In diesem Kapitel widmen wir uns der Programmierung von GPUs (englische Abkürzung für *Graphic Processing Unit*), wobei wir uns weitgehend mit der spezifischen Grafikkartenarchitektur und Programmstruktur des Hersteller *Nvidia* beschäftigen, da auch die im Rahmen dieser Arbeit geschriebenen Programme mit Hilfe von GPUs dieses Herstellers ausgeführt werden.

4.1 Grafikprozessoren

Komplexe, programmierbare Automaten, die durch elektrische Schaltung konstruiert werden sind in der heutigen Welt kaum noch wegzudenken und bilden das Rückgrat der modernen Gesellschaft. Nahezu alle elektrischen Geräte beinhalten Mikrocontroller oder Prozessoren, die durch Ausführen einzelner einfacher Befehle komplexe Probleme lösen oder Vorgänge automatisiert steuern können. Insbesondere lassen sich moderne Prozessoren zur numerischen Berechnung von komplexen physikalischen Problemen benutzen. Dabei können je nach Problem unterschiedliche Prozessorarchitekturen für eine schneller Berechnung von numerischen Aufgaben herangezogen werden.

Jeder heutige Rechner basiert im Prinzip auf der *Von-Neumann-Architektur*, siehe [23]. Bei dieser Architektur besteht der Automat aus einem Rechenwerk, einem Steuerwerk und einem Speicherwerk. Zusätzlich kann auch noch ein Ein-, bzw. Ausgabewerk vorhanden sein. Der Automat besitzt einen grundlegenden Befehlssatz, wie zum Beispiel Addition, Subtraktion, aus Datenspeicher laden. Das auszuführende Programm ist eine Abfolge solcher Befehle, die im Programmspeicher abgespeichert sind. Die Ausführung eines Programmes geschieht durch das Laden der Befehle aus dem Programmspeicher und der anschließenden Ausführung dieser im Rechenwerk. Das Rechenwerk selbst besitzt spezielle Speicherstellen, so genannte Register, die als Zwischenspeicher für Ergebnisse von Berechnungen benutzt werden und Resultate anschließend von diesen aus in den Datenspeicher geschrieben werden können. Bei der Ausführung von Programmen kann grundsätzlich immer nur ein Befehl durch das Steuerwerk geladen und durch das Rechenwerk abgearbeitet werden. Dies bedeutet, dass das Programm sequentiell ausgeführt wird.

Bei heutigen Rechnern ist das Steuerwerk und Rechenwerk typischerweise im Prozessor, auch CPU (englische Abkürzung für *Central Processing Unit*) genant, vereint. Das Speicherwerk wäre dabei der Arbeitsspeicher des Rechners. Es sei jedoch angemerkt, dass dies eine starke Vereinfachung darstellt. Heutige Prozessoren weichen im Allgemeinen von dieser Architektur ab uns besitzen beispielsweise Caches, die häufige Befehle und Daten zwischenspeichern, um den Programmablauf zu beschleunigen oder auch mehrere Kerne, bzw. Rechenwerke womit unterschiedliche Programmabläufe gleichzeitig ausgeführt werden können.

Das Grundprinzip bleibt bei CPUs jedoch das selbe. Die einzelnen Rechenwerke führen immer ein Befehl nach dem anderen aus, was weitgehend zu einem sequentiellen Programmablauf führt. Die Dauer der Ausführung eines Programmes auf einer CPU hängt somit vorwiegend von der Komplexität des verwendeten Algorithmus und der Geschwindigkeit mit der einzelne Befehle in der CPU ausgeführt werden können ab.

Für Probleme und Berechnungen, wo Zwischenergebnisse auf Daten basieren, die durch vorherige Berechnungen ermittelt werden müssen, ist die Architektur von CPUs optimal geeignet. Offensichtlich trifft dies auf eine Vielzahl von Problemen und Berechnungen zu, wie zum Beispiel die Berechnung von Summen, oder Lösungen von Differentialgleichungen im Rahmen vom Runge-Kutta-Verfahren, oder die Ausführung des Betriebssystemes, mit dessen Hilfe diese Dokument entstanden ist.

Es existieren jedoch Probleme, wo viele Daten mit den selben oder ähnlichen, unabhängigen Berechnungen manipuliert werden sollen. Solche Probleme kann man zwar auf der CPU, falls diese mehrere Rechenwerke besitzen sollte, parallelisieren, jedoch ist dies stark limitierte, weil die Anzahl der Kerne auf der CPU typischerweise in der Größenordnung von 1 bis 10 liegt.

Für Probleme dieser Art sind Grafikprozessoren (GPUs) entwickelt worden. Die Architektur einer GPU sieht dabei vor, dass möglichst viele elektrische Verschaltungen und Bauteile für die Verwendung als Rechenwerke benutzt werden,wie in [24] beschrieben. Dies führt dazu, dass GPUs aus vielen Rechenwerken bestehen, die gleichzeitig arbeiten können, jedoch jedes einzelne Rechenwerk weniger Berechnungen pro Zeiteinheit in den meisten Fällen ausführen kann, als ein Rechenwerk auf der CPU. Ferner existieren häufig weitere Restriktionen für GPUs, die im Laufenden genauer beleuchtet werden. Solche Restriktionen basieren zumeist darauf, dass beispielsweise die Rechenwerke nicht vollständig unabhängig voneinander angesteuert werden oder Speicherzugriffe durch Rechenwerke nicht unabhängig von anderen erfolgen können, da entsprechende Kontrolleinheiten und Verschaltungen, wie auf der CPU nicht für jedes einzelne Rechenwerk separat existieren.

Solche Restriktionen werden in Kauf genommen, für die Möglichkeit gleiche oder ähnliche Berechnungen für die Manipulation von größeren Datenmengen effizient zu parallelisieren um Geschwindigkeitsvorteile gegenüber selbigen Programms sequentiell oder parallelisiert auf der CPU, zu erhalten.

Wie aus der Bezeichnung Grafikprozessor interpretiert werden kann, liegt die Hauptanwendung dieser Prozessorarchitektur in der Berechnung von Grafiken bei Anwendungen wie Computerspielen, wo eine schnelle Berechnung vieler Daten gleichzeitig erforderlich ist. Es hat sich jedoch gezeigt, dass es, außerhalb der Anwendung von Grafikberechnungen, numerisch Berechnung gibt, die zu Teil parallelisierbar sind und dessen Teile auf der

GPU effizient implementiert werden können. Mit Hilfe einer solchen Parallelisierung ist es möglich Geschwindigkeitsvorteilen gegenüber einer reinen CPU implementieren zu erhalten. Wann ein Programm genau effizient auf einer GPU ausgeführt werden kann, wird weiter unten bei der Besprechung der GPU-Architektur und Programmstruktur vom Hersteller *Nvidia* besprochen.

4.2 CUDA und GPU-Architektur

In diesem Abschnitt gehen wir auf die Architektur von GPUs, speziell vom Hersteller *Nvidia*, ein und werden ansprechen, wie die Parallelisierung von Programmen auf GPUs dieser Art auf Hardwareebene geschieht. Wir stützen und dabei auf die Angaben von *Nvidia*, in [24] und [25], selber und auf [27]. Ein grundlegendes Verständnis der Architektur ist nötig, um effiziente Programme schreiben zu können. Jedoch beschreiben die folgenden Angaben nur den aktuellen Stand der Technik vom Hersteller *Nvidia*. Es existiert keine Garantie, dass die hier gemachten Erläuterungen in der Zukunft noch zutreffend sein sollten. Dies bedeutet es ist für den verwendeten Grafikprozessor empfohlen, stets die aktuellen Herstellerangaben zu beziehen. Die Angaben an dieser Stelle treffen zumindest für die *Compute Capability* 5.2 Architekturen und weitgehend die vorherigen ab Version 2.0 zu. Für den Begriff der *Compute Capability* siehe Abschnitt 4.2.1.

Unter dem Begriff *CUDA* führt *Nvidia* das Programmmodell und die Prozessorarchitektur spezialisiert für hohe Parallelisierung von Programmabläufen.

Die Grundidee hinter der Parallelisierung von Programmabläufen in *CUDA* besteht in der gleichzeitigen Ausführung von idealerweise identischen oder ähnlichen Programmabläufen auf vielen Rechenkernen (*cores*). Diese Grundidee wird unter dem Begriff *SIMT*-Architektur (englische Abkürzung für *Single-Instruction Multiple Thread*) zusammengefasst.

Im Rahmen dieser Architektur werden vorgegebene Programmabläufe (*threads*) auf die einzelnen *cores* der GPU aufgeteilt und von diesen ausgeführt. Hierbei können die Programmabläufe durch Ablaufbedingungen auf den einzelnen *cores* sich unterscheiden, was jedoch im Allgemeinen zu Gunsten der Ausführungsgeschwindigkeit, vermieden werden sollte. Um eine derartige Aufteilung von Programmabläufen zu erreichen, besteht die GPU aus einzelnen *streaming multiprocessors* (SMs), die unabhängig voneinander eigenen Rechenkerne, Ressourcen, wie Caches oder Register, und Steuereinheiten für die Ansteuerung der Kerne, besitzen. Ein SM ist somit im wesentlichen ein weitgehend eigenständiger Prozessor.

Einzelne *threads* werden in Blöcken, so genannten *thread blocks*, zusammengefasst. Ein solcher Block wird durch genau ein SM bearbeitet, während ein SM je nach verfügbaren Ressourcen mehrere Blöcke bearbeiten kann. Der einem Block zugewiesenen SM ist für die Aufteilung und Ausführung von der einzelnen Programmabläufe auf den Rechenkernen zuständig. Eine Gruppe von *thread blocks* nennt man *grid* (englisch für Gitter). Ein Programm für die GPU wird zunächst in einem sogenannten *kernel* festgelegt. Innerhalb des *kernel* erfolgt die Programmierung der einzelnen *threads*. Wird nun ein *kernel* aufgerufen, so wird ein *grid* mit vorher festgelegten Größen erzeugt und dessen Blöcke an

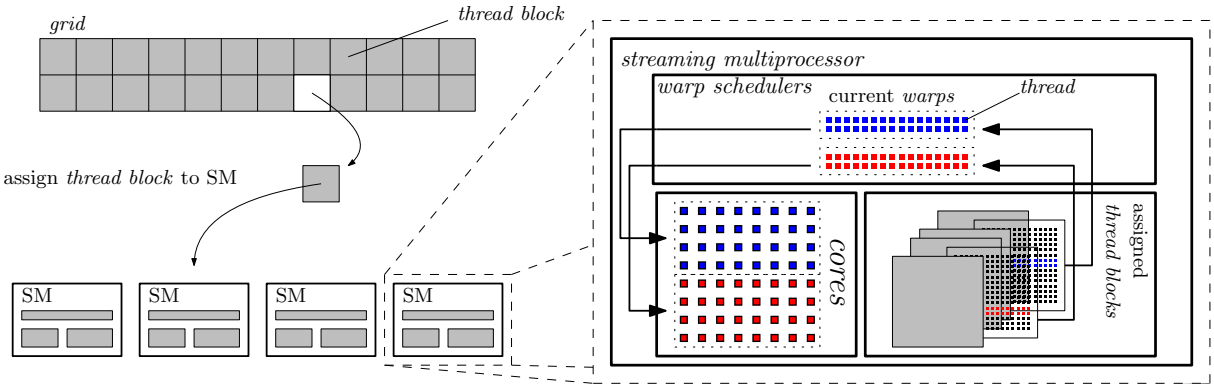


Abbildung 4.2.1: Schematische Illustration der Funktionsweise eines SMs. Gleiche Farben (außer schwarz) der *threads* und der Rechenkerne repräsentieren eine identische auszuführende Instruktion.

die einzelnen SMs übermittelt. Typischerweise ist die GPU in der Lage mehrere *kernels* gleichzeitig zu verwalten und abzuarbeiten.

Das schematisches Vorgehen der hier besprochenen Abarbeitung der Programmabläufe ist in Abbildung 4.2.1 dargestellt. Es sei angemerkt, dass die Darstellung des SMs in der Abbildung 4.2.1 kein gültiges Blockschaltbild dieser ist, sondern nur die Arbeitsweise verdeutlichen soll.

Innerhalb eines *thread blocks* werden je 32 *threads* zusammengefasst zu einem *warp*. Der SM verfügt über sogenannte *warp schedulers*. Deren Aufgabe besteht in der Entscheidung und Aufteilung der aktuellen Instruktionen von den Programmabläufen in den *warps* auf die einzelnen Kerne des SMs. Die 32 *threads* innerhalb eines *warps* werden stets gemeinsam behandelt. Die Programmabläufe eines *warps* beginnen stets zum selben Zeitpunkt an und enden zum selben. Der einfachste Fall ist, dass alle Programmabläufe innerhalb eines *warp* exakt die selben Operationen ausführen. In diesem Fall wird die aktuelle auszuführende Instruktion durch einen *warp scheduler* auf die Kerne des SMs übermittelt und von denen ausgeführt, was in minimaler Anzahl an Taktzyklen geschehen kann. Die genaue Art oder Weise der Ausführung der Instruktionen ist abhängig von der verwendeten Architektur. Man beachte, dass zwar in diesem Fall die selben Instruktionen ausgeführt werden, jedoch im Allgemeinen die Programmabläufe unterschiedliche Daten bearbeiten.

Nichtsdestotrotz besteht die Möglichkeit, dass die Programmabläufe innerhalb eines *warp* nicht die selben Instruktionen, sondern teilweise unterschiedliche Abläufe ausführen. In diesem Fall werden die einzelnen verschiedenen Abläufe serialisiert und nacheinander ausgeführt. Dies bedeutet, dass alle *threads*, deren Instruktionen nicht mit dem aktuellen Ablauf übereinstimmen als inaktiv markiert werden und solange warten, bis deren Ablauf ausgeführt wird. Dieser Sachverhalt von unterschiedlichen Abläufen innerhalb eines *warp* wird unter dem Begriff *warp divergence* geführt und kann zu Verlangsamung des Programmablaufes im Vergleich zum Fall, dass alle *threads* eines *warp* die selben Instruktionen ausführen müssen, führen. Mit Verlangsamung ist gemeint, dass der *warp* mehr Taktzyklen als der optimale Fall braucht um vollständig ausgeführt zu werden, was

gleichbedeutend mit mehr Zeit für die Ausführung des *warps* ist.

Unterschiedliche *warp* können ohne Probleme verschiedene Abläufe ausführen, da diese voneinander unabhängig laufen. Die Reihenfolge, in denen die *warp* bearbeitet werden, ist nicht festgelegt, in dem Sinne, dass vor Laufzeit dies bekannt sei. Die Reihenfolge ist beispielsweise von den verfügbaren Ressourcen, und anderen *warps*, die durch den SM bearbeitet werden, abhängig. Wie viele *warp* gleichzeitig auf den Kernen zu einem bestimmten Zeitpunkt ausgeführt werden hängt von den verfügbaren Ressourcen und Rechenkernen ab. Alle nicht zu einem Zeitpunkt aktiv ausgeführten *warp* warten, bis sie durch die *warp schedulers* und Kerne weiter abgearbeitet werden. Wir sehen somit, dass die naive Vorstellung, dass alle Programmabläufe gleichzeitig auf einer GPU ausgeführt werden, nicht zutrifft und die Ausführung einzelnen Programmabläufe nicht komplett unabhängig voneinander sind, was einer der oben angesprochenen Restriktionen ist.

Genauere Details über Anzahl der Kerne und *warp schedulers* ist spezifisch für die explizit verwendete Architektur. Beispielsweise hat ein SM der *Maxwell*-Architektur, genauer der Grafikprozessor mit der Bezeichnung *GM204* vier *warp schedulers*, wo jeder dieser 32 Rechenkerne verwaltet, siehe [30].

Eine wichtige Bemerkung ist, dass die Ausführung der Programmabläufe innerhalb eines Blockes synchronisiert werden können. Das bedeutet es besteht die Möglichkeit gemeinsame Punkte in den Programmabläufen festzulegen an deren Stelle alle *threads* eines Blockes an gelangen müssen und erst anschließend den Ablauf aller *threads* fortgesetzt wird. Da SMs unabhängig voneinander laufen und es nicht fest definiert ist, welcher Block welchem SM zugeordnet wird, existiert keine Möglichkeit die Blöcke eines *grids* zu synchronisieren.

4.2.1 Compute Capability

Die unterschiedlichen GPU-Architekturen von *Nvidia* haben unterschiedliche Eingenschaften, wie zum Beispiel die Anzahl an Rechenkernen, spezielle Funktionen oder die Details der Speichermodelle, siehe Abschnitt 4.2.2. Je nach solchen Eigenschaften wird der GPU eine *Compute Capability* $x.y$ zugeordnet, wobei x und y natürliche Zahlen sind, die die Version festlegen. Gemäß [24, Seite 16, Abschnitt 2.5], steht hierbei x für die Architektur der GPU und y für eine weitere Klassifizierung innerhalb der vorgegebenen Architektur. Beispielsweise ist für die *Maxwell*-Architektur $x = 5$. Die einzelnen Details zu den *Compute Capabilities* lassen sich unter anderem in [24] nachlesen.

Aufgrund der Tatsache, dass in dieser Arbeit GPUs mit der *Compute Capability* 5.2 Klassifizierung verwendet werden, sind die speziellen Angaben im Folgenden vorwiegend auf die Versionen $5.y$ bezogen. Diese werden jedoch als solche gekennzeichnet und sind unter anderem in [24, Seite 214, Abschnitt G.4.2] angegeben. Die restlichen Angaben sind allgemein gehalten.

4.2.2 Speichermodell der GPU

Der nächste wichtige Punkt für die Programmierung von einer GPU ist das zugrundeliegende Speichermodell, da dieses sich von der eines gewöhnlichen CPUs unterscheidet

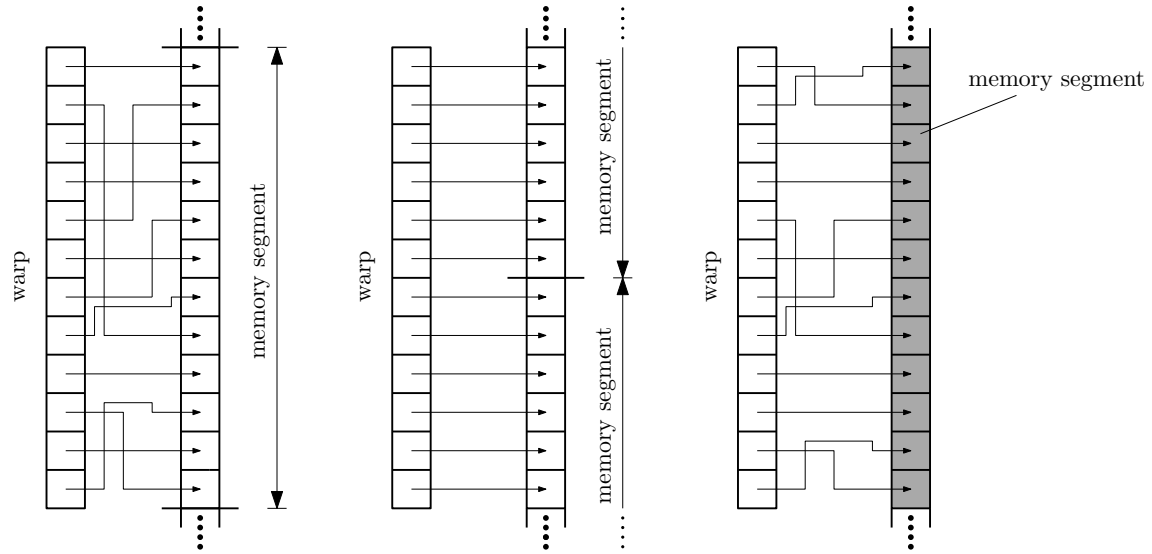


Abbildung 4.2.2: Unterschiedliche Speicherzugriffe auf den *Global Memory*. Die *thread*-Anzahl der *warps* in der Abbildung ist nicht richtig und dient nur der besseren Illustration.

und es gewisse Eigenschaften gibt, die für eine effiziente Programmierung beachtet werden müssen.

Im Grundlegenden gibt es drei verschiedene Arten von Speicher, den *Global Memory*, den *Shared Memory* und die lokalen Speicherregister der SMs, wobei die beiden letzteren Speicherbereiche in den SMs sind und somit direkt auf der GPU liegen.

4.2.2.1 Global Memory

Der größte Speicher ist der *Global Memory*. Dies ist die Analogie des Arbeitsspeichers einer CPU. Der *Global Memory* ist typischerweise mit der GPU zusammen auf der Grafikkarte aufgebracht. Dieser Speicher dient dem allgemeinen Abspeichern von Informationen, wie zum Beispiel den Spin-Gittern in den Programmen dieser Arbeit. Dabei kann auf eine Speicherstelle im *Global Memory* durch alle *threads* eines *grid* zugegriffen werden. Auch *threads* unterschiedlicher *kernels* können auf die selbe Speicherstelle zugreifen, deshalb die Bezeichnung des Speichers als globalen Speicher. Der Zugriff auf den *Global Memory* unterliegt spezifischen Restriktionen. Ein SM kann auf den globalen Speicher zugreifen und Daten in den lokalen des SMs laden. Hierbei können immer nur 32 bytes, 64 bytes oder 128 bytes lange benachbarte Speicherstellen (Segmente), deren Anfangsadresse in bytes relativ zur Adresse 0 ein Vielfaches von 32, 64 oder 128 ist vom globalen Speicher in die lokalen des SMs geladen werden. Dies bedeutet wenn die *threads* eines *warps* mit der aktuellen, auszuführenden Instruktion Daten vom *Global Memory* in ihre lokalen Variablen laden, wird der Datenzugriff in solchen ausgerichteten Speichertransaktionen von Segmenten der Länge 32 bytes, 64 bytes oder 128 bytes aufgeteilt. Welche Speichersegmente für die Datenübertragungen genau verwendet und wie diese im Detail ausgeführt werden ist abhängig von der verwendeten Architektur. Für GPUs der *Compute Capability*

5.y werden Speichertransaktionen von ausgerichteten Segmenten der Länge 32 bytes und 128 bytes verwendet. Der Datentransfer zwischen dem globalen Speicher und der GPU wird dabei stets über einen Zwischenspeicher mit der Bezeichnung *L2-Cache* ausgeführt. Es existiert für den Datentransfer zusätzlich ein weiterer Zwischenspeicher, der sogenannte *L1-Cache*. Wird nur der *L2-Cache* verwendet, so werden bei dieser *Compute Capability* die Speicherzugriffe in das Laden von 32 bytes ausgerichteten Segmenten aufgeteilt. Die Aufteilung von Speicherzugriffen in 128 bytes ausgerichteten Segmenten geschieht, falls zusätzlich der *L1-Cache* verwendet wird.

Im Rahmen der GPU-Programmierung besteht weitgehend das Ziel sein Programmablauf besonders schnell in der Programmausführung zu gestalten. Dementsprechend muss der Zugriff auf den globalen Speicher berücksichtigt werden, um eine möglichst schnelle Ladegeschwindigkeit der Daten zu erreichen. Die höchste Zugriffsgeschwindigkeit auf den globalen Speicher wird erreicht, indem alle Programmabläufe eines *warp* beim gleichzeitigen Laden von Daten nur Daten aus einem ausgerichteten Segment der oben angegeben Längen laden. Dies ist im linken Bild von Abbildung 4.2.2 illustriert. In diesem Fall geschieht das Laden der Daten innerhalb eines Ladevorgangs, was nicht noch weiter beschleunigt werden kann. In dem mittigen Bild von Abbildung 4.2.2 ist der Fall illustriert, wenn der Zugriff nicht optimal geschieht. Hierbei Laden die *threads* des *warp* in der aktuellen, auszuführenden Instruktion die selbe Anzahl an Daten wie in dem linken Fall. Jedoch braucht dieser Speicherzugriff zwei Ladevorgänge, weil zwei Segmente geladen werden müssen. Dies bedeutet, dass die Ladegeschwindigkeit im schlimmsten Fall halb so schnell ist. Das rechte Bild stellt den langsamsten Fall dar. In diesem Fall greifen alle *threads* des *warp* auf unterschiedliche Speichersegmente zu, womit das Laden von den 32 Daten in 32 Ladevorgänge aufgeteilt wird. Dies bedeutet dieser Fall ist um das 32-fache langsamer als der linke Fall in selbiger Abbildung. Das Zugreifen auf Daten wie in dem linken Fall von Abbildung 4.2.2 wird als *coalesced access* bezeichnet, was allgemein die Eigenschaft ausdrückt alle Daten eines ausgerichteten Speichersegmentes zu verwenden, um die Anzahl an nötigen Datentransaktionen zwischen den SMs und dem *Global Memory* zu minimieren.

Im mittigen Fall der Abbildung 4.2.2 muss unter Umständen mit keiner starken Laufzeitverlangsamung im Vergleich zum optimalen Fall gerechnet werden, falls die Segmente kleiner als 128 bytes sein sollten und die GPU-Architektur die Möglichkeit des Zwischenspeicherns von Daten in Caches beim Zugriff auf den globalen Speicher zulässt (siehe [29]). Darauf sollte jedoch bei der Programmentwicklung kein vollständiger Verlass sein und stets zu Gunsten der Laufzeit das *coalesced access* für die Speicherzugriffe eines *warp* eingehalten werden, falls dies möglich sein sollte.

4.2.2.2 Shared Memory

Jeder SM verfügt über ein *Shared Memory*, der als Speicherbereich für Daten dient, auf den alle *threads* eines *thread block* zugreifen können. Dabei können alle *threads* eines Blockes auf die selben Speicherstellen des *Shared Memory* zugreifen. Dies ermöglicht einen Datenaustausch zwischen den einzelnen *threads*. Jedoch können Programmabläufe unterschiedlicher *thread blocks* nicht auf die selben Speicherstellen des *Shared Memory* zugreifen.

Die Reservierung von Speicher im *Shared Memory* gilt jeweils immer nur für einen *thread block*.

Der *Shared Memory* ist in einzelne Segmente, sogenannte *banks*, aufgeteilt. Der Zugriff auf unterschiedliche Segmente geschieht unabhängig voneinander. Hiermit ist gemeint, dass zwei *threads*, die mit ihrer aktuellen, auszuführenden Instruktion Daten aus zwei Speicherstellen laden oder schreiben, die in unterschiedlichen *banks* liegen und jeweils ausschließlich einer der beiden *threads* auf eine der Speicherstellen zugreift, gleichzeitig das Schreiben und Laden für beide Speicherstellen ausgeführt wird. Versuchen jedoch zwei *threads* innerhalb des selben *banks* zu schreiben oder zu lesen, so werden die Zugriffe serialisiert, wobei die Reihenfolge dieser vor der Laufzeit nicht bekannt ist. Diese Eigenschaften des *Shared Memory* können jedoch speziell für die einzelnen Architekturen abweichen, insbesondere die Größe des *Shared Memory*.

Für die *Compute Capability* 5.y beträgt die Größe des *Shared Memory* mindestens 64 kbytes pro SM, wovon jedoch nur 48 kbytes im Rahmen der Programmierung von *threads* verwendet werden können. Der *Shared Memory* ist dabei in 32 *banks* aufgeteilt. Hierbei liegen benachbarte Speicherstellen mit 32-Bit Länge in benachbarten *banks*. Konkret bedeutet das für einen Speicherplatz mit 32-Bit Länge und der Adresse $m \in \mathbb{N}_0$, dass dieser in jenem Segment mit der Nummer $m_b = m \bmod 32$ liegt, wobei die *banks* von 0 bis 31 durchnummeriert werden.

4.2.2.3 Zugriffsgeschwindigkeiten

Die einzelnen Arten des Speichers besitzen unterschiedliche Zugriffsgeschwindigkeiten, die angeben wie viele Daten innerhalb eines bestimmten Zeitraumes geladen oder geschrieben werden können.

Der schnellste Speichertyp sind die Speicherregister der SMs, weil auf diese nahezu ohne Latenz zugegriffen werden kann. Jedoch ist die Anzahl an Speicherregister pro *thread* stark begrenzt, was die Möglichkeit der Programmbeschleunigung über Verwendung von Registern einschränkt.

Der *Global Memory* ist zwar der größte Speicher, jedoch auch der langsamste im Vergleich zu den anderen beiden Speichertypen. Insbesondere ein ineffizienter Datenzugriff kann sich stark in der Laufzeit auswirken, deshalb sollte falls möglich das *Shared Memory*, was bis zu einer Größenordnung mehr Datendurchsatz besitzt, verwendet werden.

4.3 CUDA C/C++ Programmierung

An dieser Stelle gehen wir auf die Programmierung von *kernels* für die GPUs von *Nvidia* mit Hilfe der C/C++ Programmiersprache ein und besprechen die grundlegenden Strukturen. Die Angaben beziehen sich auf die *CUDA* Version 7.5, mit den Beschreibungen von *Nvidia* in [24], [25] und [26]. Die Darstellung soll nicht den Anspruch einer Einführung in die Programmierung von GPUs erheben mit der Intention, dass der Leser ausschließlich mit Hilfe dieser Arbeit die Programmierung von GPUs erlernen kann. Für Lernzwecke sei beispielsweise auf [24] und [25] oder [27] verwiesen. Es soll vorwiegend ein Überblick über

die wichtigsten Elemente geliefert werden, die für das Verständnis der in dieser Arbeit verwendeten Programmen benötigt werden.

Nvidia baut, für die Programmierung seiner GPUs, auf die Syntax von der C/C++ Sprache auf, indem diese durch passende Schlüsselwörter und Programmbibliotheken ergänzt wird. Bei der Programmierung wird dabei zwischen dem *host* und dem *device* unterschieden. Mit dem *device* wird die an einem Rechner angebrachte Grafikkarte, die den Grafikprozessor beherbergt, bezeichnet. Der *host* ist der durch die CPU gesteuerte Rechner, der mit dem Grafikprozessor kommunizieren kann und diesen verwaltet. Das gesamte Programm wird in *host*- und *device*-Code aufgeteilt. Für die Kompilierung des Quellcodes wird das durch *Nvidia* zur Verfügung gestellte Programm `nvcc` verwendet, dass für die unterschiedlichsten Betriebssysteme und Plattformen existiert. Das Programm ist kein eigenständiger Compiler, sondern bedarf immer einen *host*-Compiler, wie zum Beispiel `gcc` auf Linux-Distributionen.

4.3.1 Programmierung von *kernels*

Ein *kernel* wird über folgende Syntax, analog zu einer Funktion in der Programmiersprache C/C++, deklariert und definiert:

```
__global__ void kernelname(T1 arg1, T2 arg2,... ){

  //Hier steht die Definition von kernel

}
```

Mit dem Schlüsselwort `__global__` wird festgelegt, dass es sich um die Deklaration eines *kernel* handelt. Der Datentyp der Rückgabe ist stets `void` und `kernelname` deklariert den Namen des *kernel*.

Innerhalb des Gültigkeitsbereiches der Definition des *kernel* erfolgt die Programmierung der *threads*. Das Programm innerhalb der Definition wird in C/C++ geschrieben und ist jenes, dass die Programmabläufe der *threads* festlegt, dass heißt die Instruktionen innerhalb der Definition sind die von den Programmabläufen der *threads*. Der Aufruf eines *kernel* erfolgt innerhalb des *host*-Codes, der auch die *main*-Funktion beinhaltet, über

```
dim3 threads(blockDimX,... );
dim3 blocks(gridDimX,... );

kernelname<<<threads, blocks>>>(arg1, arg2,... );
```

wobei `threads` die Größe der *thread blocks* und `blocks` die Größe des *grid* festlegt. Die *threads* und *thread blocks* werden in Arrays der Dimension 1, 2 oder 3 angeordnet, je nachdem ob dem Konstruktoren von `threads` oder `blocks` entsprechend 1, 2 oder 3 Argumente des Typs `unsigned int` übergeben werden. Innerhalb der Arrays können die einzelnen *threads*, bzw. *thread blocks* durch Indextupeln passender Dimension identifiziert

werden. Die Indizes der Tupeln können innerhalb der Programmabläufe der *threads* in der Definition vom *kernel* abgerufen werden. Für die Koordinaten der *threads* innerhalb des Arrays eines *thread blocks* geschieht dies über die Variablen `threadIdx.x`, `threadIdx.y` und `threadIdx.z`. Die Koordinaten des *thread block*, in dem sich der *thread* befindet, werden analog über die Variablen `blockIdx.x`, `blockIdx.y` und `blockIdx.z` bestimmt. Hierüber lässt sich ein *thread* innerhalb des *thread block* oder des *grid* eindeutig indizieren und die Programmabläufe abhängig vom *thread* machen. Diese Variablen werden nicht in der Definition von `kernelname` deklariert, da dies außerhalb des Gültigkeitsbereiches von der Definition global in den eingebundenen Headers für die *CUDA*-Programmierung geschieht.

Ein typisches Beispiel für die Verwendung der Indizierung von *threads* ist das Laden oder Speichern von Daten aus dem globalen Speicher. Dies kann beispielsweise durch

```
__global__ void metropolis2DPeriodicOdd(unsigned short* spins_d,  int*
   randoms_d, double* local_probabilities_d){

//Hier steht der vorherige Ablauf des thread.

*(spins_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x) =
   local_sub_lattices[_OWN_];

//Hier steht der Rest des Ablaufes.

}
```

gegeben sein, wobei es sich hier um ein Teil eines Programmes aus dieser Arbeit handelt, siehe Abschnitt 8.2.1.5). In dieser Programmzeile wird ein Datum aus dem Array `local_sub_lattices` des Datentyps `unsigned short` in den Speicherbereich des globalen Speichers geschrieben, dessen Anfangsadresse durch den Zeiger `spins_d` des Typs `unsigned short*` referenziert wird. Es werden dabei eindimensionales *thread blocks* mit je 64 *threads* und ein zweidimensionales *grid* verwendet. An dem Codebeispiel ist zu erkennen, dass über die Indizes jeder *thread* des *grids* an genau einer ihm zugeordneten Speicherstelle schreibt. Analog kann man das Lesen von Daten durch die *threads* gestalten. Somit wird deutlich wie unterschiedliche *threads* die selben Berechnungen auf unterschiedlichen Daten ausführen können. Das Programm das innerhalb der Definition des *kernel* angegeben wird lässt sich somit als eine Art Vorlage sehen, die angibt wie die Programmabläufe der einzelnen *threads* explizit durch deren Position im *grid*, bzw. *thread block* zu definieren sind.

Standardmäßig wird nach dem Aufruf des *kernel* der *host*-Code weiter ausgeführt, ohne dass auf die Beendigung des *kernel* gewartet wird. Dies liegt daran, dass der Grafikprozessor unabhängig vom *host*-Programm arbeitet. Werden mehrere *kernels* hintereinander aufgerufen, so werden diese einer Warteschlange (*stream*) zugeordnet, dabei können mehrere solcher Warteschlangen existieren. Falls keine explizite Einstellung erfolgen sollte, werden alle *threads* einem standardmäßigen *stream* zugeordnet, der die Bezeichnung *default stream* trägt. Innerhalb einer solchen Warteschlange werden die zugewiesenen *kernel* nacheinander abgearbeitet, dass heißt für zwei *kernels* A und B mit der Eigenschaft,

dass *kernel* A zeitlich vor *kernel* B dem selben *stream* zugeordnet wäre, wird zunächst *kernel* A abgearbeitet und nach dessen Beendigung *kernel* B. Befänden sich jedoch *kernel* A und *kernel* B in unterschiedlichen Warteschlangen, so ist das obige Verhalten nicht mehr gegeben und beide *kernels* können prinzipiell gleichzeitig auf der GPU abgearbeitet werden. Für die Programme dieser Arbeit wird nur der *default stream* verwendet, womit das Konzept mehrerer Warteschlangen von keiner Relevanz ist.

Falls das *host*-Programm auf die Beendigung eines *kernel* warten soll, kann dies durch den Aufruf der Funktion `cudaDeviceSynchronize()` nach dem Aufruf des *kernel* erreicht werden.

4.3.2 Speicherreservierung

Die unterschiedlichen Arten von Speicher werden über eine unterschiedliche Art und Weise reserviert.

4.3.2.1 Global Memory

Ein Speicherbereich im globalen Speicher wird über folgende Syntax und Funktion innerhalb des *host*-Codes reserviert:

```
T* pointer;

cudaMalloc(&pointer, N*sizeof(T));
```

Zunächst wird der Zeiger `pointer`, der auf ein Element des Datentyps `T` referenziert deklariert. Anschließend wird über den Funktionsaufruf `cudaMalloc(...)` beginnend an der durch `pointer` referenzierten Anfangsadresse `N*sizeof(T)` Bytes an globalen Speicher reserviert. Auf die N Speicherstellen vom Datentyp `T` kann innerhalb der *threads* über `*(pointer + j)` mit $j \in \{0, \ldots, N-1\}$ zugegriffen werden, falls der Zeiger dem *kernel* als Argument übergeben wird. Der Zugriff in dieser Art ist jedoch nicht vom *host* aus möglich, da der Zeiger eine Adresse innerhalb des globalen Speichers beinhaltet, der nicht durch den *host* direkt adressiert oder verwendet werden kann. Genauso ist der Grafikprozessor nicht in der Lage direkt auf den Arbeitsspeicher des *host* zuzugreifen. Nur über passende System-, bzw. Treiberaufrufe können die gewünschten Manipulationen ausgeführt werden (beispielsweise `cudaMemset(...)`, etc.).

Falls der Speicher nicht mehr gebraucht werden sollte muss dieser explizit freigegeben werden über `cudaFree(pointer)`, ansonsten bleibt die Reservierung bestehen, selbst wenn kein Zeiger den Speicher mehr referenzieren sollte.

4.3.2.2 Shared Memory

Der *Shared Memory* wird innerhalb der *kernel*-Definition reserviert. Die Syntax sieht dabei wie folgt aus:

```
__global__ void kernelname(T1 arg1, T2 arg2,... ){
//Hier steht der vorherige Ablauf des thread.

__shared__ T array[SIZE];

//Hier steht der Rest des Ablaufes.
}
```

Es wird ein Array mit der Bezeichnung `array` vom Datentyp `T` der Größe `SIZE` angelegt, wobei `SIZE` als explizite Zahl angegeben werden muss, da es sich um ein statisch angelegtes Array handelt. Das Schlüsselwort `__shared__` gibt dabei an, dass dieses Array im *Shared Memory* angelegt werden soll. Das Anlegen vom Array geschieht hierbei nicht für jeden *thread* sondern immer für einen *thread block*.

Die Speicherreservierung gilt nur innerhalb des Gültigkeitsbereiches der Definition von `kernelname`. Nach Beendigung des *kernel* wird der im *Shared Memory* reservierte Speicher freigegeben.

Eine typische Anwendung des *Shared Memory* liegt in der Optimierung vom Laden und Schreiben von Daten in den globalen Speicher. Wie in Abschnitt 4.2.2.1 besprochen ist es für die Zugriffsgeschwindigkeit auf den globalen Speicher zu empfehlen den *coalesced access* einzuhalten um maximale Performance erreichen zu können. Häufig kann es jedoch sein, dass ein *thread* Zugriffe ausführen müsste, wo der *coalesced access* nicht gegeben ist oder mehrfach Zugriffe geschehen müssen. Mit Hilfe des *Shared Memory* können zunächst die notwendigen Daten aus dem globalen Speicher in diesen geladen werden, wobei versucht wird soweit wie möglich den Speicherzugriff optimal durch den *coalesced access* zu gestalten. Anschließend können die einzelnen *threads* die Daten aus dem *Shared Memory* laden, der weitaus schneller und insbesondere nicht den selben Restriktionen wie dem *Global Memory* unterlegen ist. Diese Methode ist in [28] genauer dargestellt und wird auch in den Programmen dieser Arbeit verwendet.

Kapitel 5

Implementation der Algorithmen

In diesem Kapitel werden die Implementationen der Algorithmen, Metropolis-Algorithmus und Demon-Algorithmus, besprochen. Für letzteren werden die beiden Update-Verfahren für den kanonischen Update der *demons* aus Abschnitt 3.2.2.2 implementiert. Die Programme sind hierbei in der Programmiersprache C/C++ mit dem *CUDA* spezifischen Syntax geschrieben.

5.1 Multispin-Coding

Im Rahmen der Simulation des Ising-Modells wird die aktuelle Spinkonfiguration σ_n im Speicher der verwendeten Rechenmaschine gehalten. Typischerweise kann hierfür ein Array der entsprechenden Dimension d angelegt werden, wo jeder Eintrag entweder den Wert 1 oder -1 annimmt und den entsprechenden Spinzustand repräsentiert. Bei der Reservierung des Speichers für ein Array kann je nach Programmiersprache und System nur bestimmte Datentypen für die einzelnen Speicherstellen des Arrays verwendet werden, wie 4 bytes-Integer. Dies bedeutet, dass üblicherweise mehrere Bytes Speicher verwendet werden, um zwischen zwei Zuständen zu unterscheiden. Ein Spinzustand kann jedoch schon in nur ein Bit b kodiert werden, indem $b = 0$ für den Zustand -1 und $b = 1$ für den Zustand 1 steht. Dies bedeutet, dass beispielsweise für einen 4 bytes-Integer 31 bits keine sinnvolle Information tragen, jedoch trotzdem dieser Speicher reserviert wird.

Die Idee hinter der Mutlispin-Coding-Methode, besteht darin in jedem Bit eines Wortes der Länge n bytes ein Spinzustand zu kodieren. Hierbei unterscheiden wir zwischen zwei Möglichkeiten. Entweder die einzelnen Bits eines Wortes repräsentieren unabhängige Spins, womit wir im Speicher $8 \cdot n$ Spinkonfigurationen gleichzeitig halten oder in einem Wort werden Spins der selben Spinkonfiguration gespeichert. Dies führt nicht nur zu einer effizienteren Speicherverwendung, sondern es können auch Instruktionen für die Implementierung verwendet werden, die einzelne Bits manipulieren und typischerweise besonders schnell auf einer Rechenmaschine ausgeführt werden können, da sie normalerweise Teil des Instruktionssatz der Rechenwerke sind.

Die in dieser Arbeit implementierten Algorithmen verwenden unterschiedlichen Methoden des Multispin-Codings. Für den Demon-Algorithmus werden 32 unabhängige Spins

in einem 4 bytes großen Wort gespeichert, während für den Metropolis-Algorithmus in einem 2 bytes großem Wort Spins der selben Konfiguration gespeichert sind.

5.2 Implementierung des Metropolis-Algorithmus

Für die Implementierung des Metropolis-Algorithmus auf GPUs, insbesondere mit Mutli-Spin-Coding, existieren die Arbeiten in [31] und [32]. Die Implementierung an dieser Stelle basiert teilweise auf Konzepte aus den angegebenen Arbeiten, jedoch ist die Implementierung in dieser Arbeit nicht vollständig identisch mit der aus [31] und [32]. Es existiert sowohl eine zweidimensionale, als auch eine dreidimensionale Implementierung.

Bei den in diesem Kapitel besprochenen Beschreibungen beziehen wir uns auf ein zweidimensionales Gitter. Die Verallgemeinerung auf drei Dimensionen ist grundlegend nicht schwierig, da das Gitter in der $i = 3$ Richtung, mit der Ausdehnung L_3, aufgeteilt werden kann in L_3 zweidimensionale Gitter. Es müssen nur die Wechselwirkungen einer Ebene mit der drüber oder drunter liegenden Ebene berücksichtigt werden. Die dreidimensionale Implementierung ist im Programm mit inkludiert, jedoch spielt sie für die Betrachtungen in dieser Arbeit keine wichtige Rolle.

Der hier programmierte Quellcode ist in 8.2.1 angegeben.

5.2.1 Programmaufbau

Das Programm besteht aus unterschiedlichen *kernel* und Funktionen, die aufgerufen werden müssen, um eine gewünschte Simulation auszuführen. Die Grundlage für die Programmierung einer Simulation ist dabei die Klasse `Lattice`. Die Klasse beinhaltet alle Attribute und Methoden, die für die Simulation des Gitters notwendig sind.

Für eine Simulation wird zunächst ein Objekt `gitter(L1, L2, L3, _PERIODIC_, beta)` der Klasse `Lattice` angelegt, wobei die ersten drei Argumente des Konstruktor die Ausdehnungen des Gitters angeben. `_PERIODIC_` legt fest, dass es sich um periodische Randbedingungen handelt, und `beta` ist die dimensionslose inverse Temperatur, bei der die Simulation ausgeführt werden soll. Falls im Konstruktor das dritte Argument ausgelassen werden sollte, wird ein zweidimensionales Gitter angenommen.

Für die eigentliche Simulation werden die Methoden `gitter.updateSystem()` und `gitter .observeTotalMagnetization()` verwendet. Mit der ersten Methode wird der Update des Gitter gemäß des Metropolis-Algorithmus ausgeführt und mit der zweiten Methode die totale Magnetisierung

$$N \cdot m = \sum_{u \in G} \sigma_n(u) \tag{5.2.1}$$

der aktuellen Spinkonfiguration berechnet.

Eine wichtige Anmerkung ist, dass L_1 und L_2 Vielfache von 32 sein müssen und im dreidimensionalen Fall L_3 einen geraden Wert haben muss.

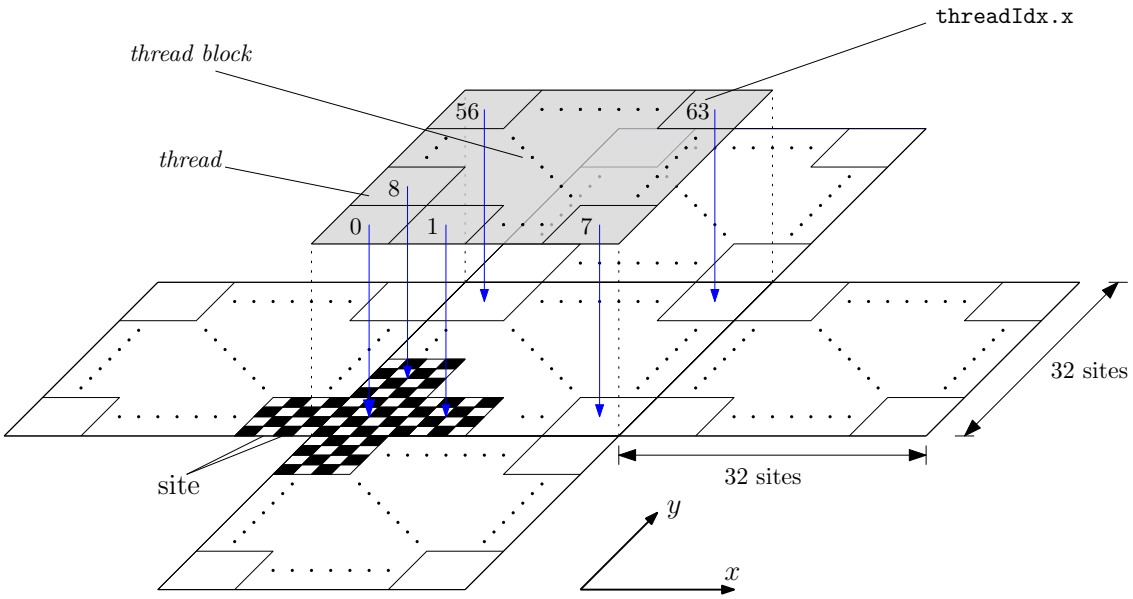


Abbildung 5.2.1: Illustration des Gitteraufbaues für den Metropolis-Algorithmus mit der Aufteilung der Teilgitter auf die einzelnen *threads* eines *thread blocks* für ein zweidimensionales Gitter.

5.2.2 Gitteraufbau

Das Gitter wird in einem Array des Datentyps `unsigned short` gespeichert, wobei die Daten im *Global Memory* lokalisiert sind. Jedes Bit einer Speicherstelle repräsentiert ein Spin des Gitters. Eine Speicherstelle wiederum repräsentiert ein (4×4)-Teilgitter, dass auf der x-y-Ebene, mit den Ausdehnungen L_1 und L_2, liegt. Der grundlegende Gitteraufbau ist in Abbildung 5.2.1 dargestellt.

Innerhalb einer Speicherstelle `sub_lattice` können die einzelnen Spins über den Syntax `(sub_lattice & (1 << (x + y * 4))) >> (x + y * 4)` ausgelesen werden, wobei x und y Koordinaten innerhalb des Teilgitters zwischen 0 und 3 für den entsprechenden Richtungen sind. Dieses Konzept stammt aus [32].

Das Gitter wird auf der x-y-Ebene in 32×32 große Blöcke aufgeteilt. Jeder Block wiederum besteht aus 64 (4×4)-Teilgitter. Dies bedeutet ein Gitterblock wird in 64 Speicherstellen des Arrays abgespeichert, was insgesamt 128 bytes entspricht. Hieran ist es möglich zu erkennen, weshalb die Restriktionen für L_1, bzw. L_2 existieren. Die Restriktionen erlauben das Laden eines Gitterblockes unter Berücksichtigung des *coalesced access* Prinzipes, da benachbarte Teilgitter des selben Gitterblockes nebeneinander im Speicher abgelegt sind und die Anfangsadresse des ersten Teilgitters eine Blockes ein Vielfaches von 128 bytes ist.

5.2.3 Update der Spinkonfiguration

Das Programm speichert stets nur die aktuelle Spinkonfiguration im entsprechenden Array. Für das Updaten der Konfiguration von σ_n zu σ_{n+1} müssen nacheinander die *kernel*

```
dim3 blocks(gridDimX, gridDimY);

metropolis2DPeriodicEven<<<blocks, 64>>>(spins_d, randoms_d,
    local_probabilities_d);
metropolis2DPeriodicOdd<<<blocks, 64>>>(spins_d, randoms_d,
    local_probabilities_d);
```

Programmcode 5.1: Entnommen aus Abschnitt 8.2.1.7

aufgerufen werden, entsprechend `...3D..` für ein dreidimensionales Gitter.

Wie in Abschnitt 3.2.1 dargestellt, muss für die lokalen Updates $W_{(x_1,x_2)}$ der einzelnen Gitterpunkte eine Reihenfolge definiert werden. Aufgrund der Tatsache, dass wir möglichst viele solcher Updates, innerhalb eines *kernel*-Aufrufes, parallel ausführen möchten, wird ein Schachbrettmuster verwendet. Hierbei unterscheiden wir zwischen geraden und ungeraden Gitterpunkten. Ein Gitterpunkt (x_1, x_2) heißt gerade, genau dann, wenn $x_1 + x_2$ gerade ist und ungerade, wenn $x_1 + x_2$ ungerade ist. Für den dreidimensionalen Fall wird entsprechend die Koordinate x_3 in den Summen hinzugefügt. Wenn die Gitterpunkte derartig unterschieden werden, erhalten wir ein Schachbrettmuster, wie in Abbildung 5.2.1, wo dies für die (4×4)-Teilgitter exemplarisch dargestellt ist. In der Abbildung ist es möglich zu erkennen, dass die nächsten Nachbarn eines geraden Gitterpunktes, ungerade sind und die eines ungeraden Gitterpunktes, gerade sind. Die jeweiligen *kernel* die aufgerufen werden, führen entweder, wie die Bezeichnungen nahelegen, lokale Updates für gerade oder ungerade Gitterpunkte aus. Dies bedeutet, die Spins von den nächsten Nachbarn bleiben während des lokalen Updates unberührt, woraus sich offensichtlich ergibt, dass für die Energieänderung $H(\sigma') - H(\sigma)$ angenommen werden darf, dass keine weiter lokalen Updates gleichzeitig ausgeführt werden und folglich sich die Energieänderung nur durch die nächsten Nachbarn eines Spins ergibt.

Für die Ausführung des *kernels* wird ein zweidimensionales, bzw. dreidimensionales, *grid* erstellt, wo jeder *thread block* aus 64 *threads* besteht. Hierbei ist einem *thread block* ein Gitterblock zugeordnet, wie in Abbildung 5.2.1 veranschaulicht. Jeder *thread* eines *thread blocks* ist für das Updaten der geraden oder ungeraden Spins eines (4×4)-Teilgitters verantwortlich, [32]. Dabei werden die einzelnen *threads* nacheinander mit steigendem Index auf die Teilgitter aufgeteilt.

Um diese Aufteilung zu erreichen müssen die Daten aus dem Array geladen werden. Hierzu wird zunächst der Gitterblock selber und die nächst benachbarten Gitterblöcken in das *Shared Memory* geladen, wobei die periodischen Randbedingungen beachtet werden. Nach dem dies geschehen ist, beziehen die einzelnen *threads* aus dem *Shared Memory* das ihnen zugeordnete Teilgitter und jeweils die nächst benachbarten (4×4)-Teilgitter. Jeder *thread* führt nun die lokalen Updates $W_{(x_1,x_2)}$ für die entsprechenden Gitterpunkte aus. Hierbei werden die, für die Berechnungen notwendigen, Spinzustände entweder aus dem eignen Teilgitter oder bei Spins, die am Rand des Teilgitters liegen, aus den nächst benachbarten Teilgittern geladen.

Nachdem die lokalen Updates ausgeführt sind, werden die Resultate ins *Shared Memory* zurück und anschließend in das *Global Memory* geschrieben.

5.2.3.1 Zufallszahlen innerhalb von *threads*

Für die lokalen Updates müssen die einzelnen *threads* entsprechend Zufallszahlen erzeugen. Dies geschieht konzeptionell wie in [32]. Die einzelnen *threads* erzeugen die Zufallszahlen über einen linearen Kongruenzgenerator, der in Abschnitt 3.1.2 beschrieben ist. Hierbei wird die 1. Variation aus dem Abschnitt verwendet. Diese ist in der Funktion `simpleRandomInteger0` implementiert. Die aktuelle Zufallszahl x_n eines Generators wird im globalen Speicher abgelegt, sobald ein *thread* eines *kernel* Zufallszahlen erzeugt hat und seinen Ablauf beenden wird ohne weitere Zufallszahlen zu generieren. Beim Start eines *threads* wird zunächst die letzte im globale Speicher abgelegte Zufallszahl geladen und für die Erzeugung der nächsten Zufallszahlen verwendet. Hierbei müssen die ersten Zufallszahlen x_0 der Generatoren für die einzelnen *threads* initialisiert werden. Dies geschieht über einen Marsaglia-Zaman-Generator und ist in der Funktion `initializeRandoms` implementiert. Die Initialisierung kann prinzipiell zwischen einzelnen *kernel*-Aufrufen platziert werden, um, falls benötigt, eine bessere Statistik zu erhalten und Abhängigkeiten der Zufallsgeneratoren in den einzelnen *threads* zueinander zu minimieren.

5.2.4 Berechnung der totalen Magnetisierung

Das Ziel der Simulation ist es nicht nur Spinkonfigurationen zu erzeugen, sondern auch entsprechende Observablen $A(\sigma)$ zu bestimmen. Die für diese Arbeit vorwiegend benutzen Observablen sind m, m^2 und m^4. Hierzu muss die totale Magnetisierung in Gleichung (5.2.1) aus der aktuellen Spinkonfiguration bestimmt werden.

Dies wird über folgende Programmierung erreicht:

```
dim3 blocks(gridDimX, gridDimY);
totalMagnetizationBlockWide2D <<<blocks, 64 >>>(spins_d,
   magnetization_block_wide_d);
cudaMemcpy(magnetization_block_wide, magnetization_block_wide_d,
   gridDimX*gridDimY*sizeof(int), cudaMemcpyDeviceToHost);

for (int x = 0; x < gridDimX; x++){
  for (int y = 0; y < gridDimY; y++){
    total_magnetization = total_magnetization + *(
   magnetization_block_wide + x + y*gridDimX);
  }
}
```

Programmcode 5.2: Entnommen aus Abschnitt 8.2.1.7

Zunächst wird der *kernel* mit der Bezeichnung `totalMagnetizationBlockWide2D` aufgerufen. Hierbei werden, analog zum Updaten der Spinkonfigurationen, 64 *threads* pro *thread block* verwendet. Ferner ist jedem Gitterblock ein *thread block* zugeordnet. Für die Berechnung der totalen Magnetisierung lädt zunächst, wie in Abbildung 5.2.1, jeder *thread* das ihm zugeordneten (4×4)-Teilgitter in sein lokalen Speicher. Anschließend berechnet

jeder *thread* die totale Magnetisierung des jeweiligen Teilgitters aus und speichert das Ergebnis ins *Shared Memory*. Nun summiert der erste *thread* eines jeden halben *warps*, das heißt jeder 16. *thread*, die totalen Magnetisierungen des Teilgitter im halben *warp* auf. Anschließend summiert der erste *thread*, mit `threadIdx.x==0`, die totalen Magnetisierungen der halben *warps* auf. Das Resultat wird anschließend in den globalen Speicher geschrieben. Die so bestimmten totalen Magnetisierungen pro Gitterblock werden vom globalen Speicher (`magnetization_block_wide_d`) in den *host*-Speicher (`magnetization_block_wide`) kopiert. Im Anschluss werden auf dem *host* die totalen Magnetisierungen pro Gitterblock aufsummiert, was die gesamte totale Magnetisierung liefert.

5.3 Implementierung des Demon-Algorithmus

Im Gegensatz zum Metropolis-Algorithmus wird bei der Implementation des Demon-Algorithmus 32 unabhängige Spinzustände jeweils in einem Wort der Länge 4 bytes gehalten, womit 32 unabhängige Spinkonfigurationen für das selbe Gitter simuliert werden, wobei diese Technik beispielsweise in [21] verwendet wird. Dieses Grundkonzept führt zu einem konzeptionell anderen Aufbau der Implementierung, insbesondere hinsichtlich der Speicherverwaltung.

Der vollständige Quellcode des Programms ist in Abschnitt 8.2.2 vorzufinden.

5.3.1 Programmaufbau

Für die Ausführung von Simulationen des Ising-Modells mit Hilfe der Implementierung des Demon-Algorithmus wird auch hier eine Klasse `Lattice`, dessen Konstruktor sich wie der aus Abschnitt 5.2.1 verhält, verwendet.

Das mikrokanonische Update des Spingitters eines Objektes `gitter` der Klasse wird über den Aufruf `gitter.updateSystem()` ausgeführt. Für das kanonische Update der *demons* sind beide in Abschnitt 3.2.2.2 angegebenen Verfahren implementiert. Das naive Update wird dabei über die Methode `gitter.updateDemonLayersNew()` ausgeführt. Das Update mit Hilfe der Binomialverteilung geschieht über `gitter.updateDemonLayers()`. Die Funktionen zum Updaten bearbeiten hierbei alle 32 Spinkonfigurationen.

Die Messung der totalen Magnetisierung in Gleichung (5.2.1) geschieht über `gitter .observeTotalMagnetization(magn)`, wobei `magn` ein Zeiger auf ein Array mit 32 Einträgen des Datentyps `int` ist, in denen die totale Magnetisierung der einzelnen Konfigurationen gespeichert sind.

Auch bei dieser Implementierung unterliegen die gewählten Ausdehnungen L_1, L_2 und L_3 gewissen Restriktionen. Hierbei muss L_1 ein Vielfaches von 64 und L_2 ein Vielfaches von 32 sein. Ferner muss L_3 eine gerade Zahl sein.

5.3.2 Gitteraufbau

Die 32 Spins der Konfigurationen werden für einen bestimmten Gitterpunkt $x_0 \in G$ in einer Speicherstelle des Datentyps `unsigned int` abgespeichert. Das i-te Bit der Speicher-

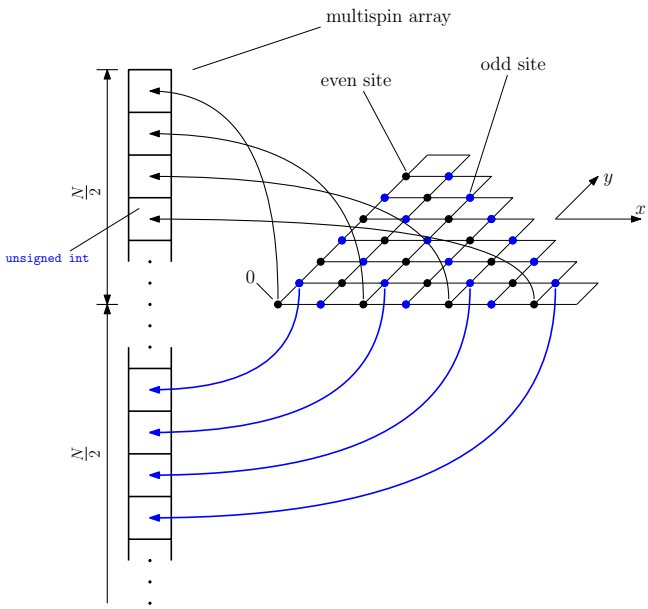


Abbildung 5.3.1: Illustration der Speicherung der Multispins mit der Zuordnung zu den einzelnen Gitterpunkten für ein zweidimensionales Gitter mit $N = L_1 \cdot L_2$ Gitterpunkten.

stelle ist dabei der Spinzustand des Gitterpunktes x_0 der i-ten Spinkonfiguration. Wir bezeichnen ab diesem Punkt eine solche Speicherstelle, die die 32 Spins für einen Gitterpunkt speichert, als ein Multispin.

Die Spinkonfigurationen werden in einem Array des Datentyps `unsigned int` der Länge $N = L_1 \cdot L_2$, entsprechend für drei Dimensionen durch L_3 ergänzt, abgespeichert. Hierbei repräsentiert eine Speicherstelle den Multispin eines Gitterpunktes. Die Multispins aller geraden Gitterpunkte sind in der ersten Hälfte und alle Multispins der ungeraden Gitterpunkte in der zweiten Hälfte des Arrays lokalisiert. Der Sachverhalt ist in Abbildung 5.3.1 illustriert. Hierbei liegen alle geraden, beziehungsweise ungeraden, Gitterpunkte mit der selben y-Koordinate, bezogen auf die Richtung mit Ausdehnung L_2, nebeneinander im Speicher. Konkret liegt, neben einer Speicherstelle eines geraden, beziehungsweise ungeraden, Gitterpunktes mit den Koordinaten (x, y) und der Speicheradresse m in Bytes, der Gitterpunkt mit den Koordinaten $(x + 2, y)$ und der Speicheradresse $m + 4$.

Für die *demons* werden die drei Energieniveaus $E_1 = 4$, $E_2 = 8$ und $E_3 = 16$ gewählt. Die Besetzung eines Energieniveaus kann in einem Bit b kodiert werden. Hierbei bedeutet $b = 0$, dass das Energieniveau nicht besetzt ist und entsprechend $b = 1$, dass es besetzt ist. Für jeden Spinzustand eines Multispins müssen somit drei Bits verwendet werden, um am einem Gitterpunkt für die i-te Spinkonfiguration den *demon* abzuspeichern. Jedem Multispin sind hierfür drei Speicherstellen des Datentyps `unsigned int` zugeordnet. Diese werden als `hdemons`, `mdemons` und `ldemons` bezeichnet und kodieren die Besetzungszustand der Energieniveaus E_3, E_2 und E_1. Das i-te Bit einer der Speicherstellen repräsentiert hierbei die Besetzung des entsprechenden Energieniveaus des *demon* für den Gitterpunkt in der i-ten Spinkonfiguration.

Die Speicherstellen der *demons* werden in einem Array der Länge $N = 3 \cdot L_1 \cdot L_2$ abgespeichert. Hierbei wird das Array, analog zu den Multispins, in zwei Bereichen aufgeteilt. In der ersten Hälfte werden die Zustände der *demons* für die gerade Gitterpunkte und in der zweiten Hälfte für die ungeraden Punkte gespeichert. Jeder dieser Bereiche wiederum wird in drei Teilbereiche aufgeteilt, wobei in den einzelnen Teilbereichen mit steigender Adresse jeweils die Variable `hdemons`, `mdemons` oder `ldemons` für die einzelnen Zustände abgespeichert werden.

Im Rahmen der Implementierung wird das Gitter in der x-y-Ebene aufgeteilt in Gitterblöcke mit 64 Multispins in der x-Richtung mit der Ausdehnung L_1 und 32 Multispins in der y-Richtung mit der Ausdehnung L_2.

5.3.3 Update der Spinkonfigurationen

Für das Updaten der Spinkonfigurationen müssen die beiden Prozesse, mikrokanonisches Update des Gesamtsystemes und kanonisches Update der *demons*, implementiert werden.

5.3.3.1 Mikrokanonisches Update der Spinkonfigurationen

Für das mikrokanonische Update müssen mehrere *kernel* hintereinander aufgerufen werden. Die Programmroutine hierfür ist durch

```
dim3 blocks(this->gridDimX, this->gridDimY);
dim3 blocks_red(this->gridDimX, this->gridDimY / 2);
unsigned int random_offset;


metropolis2DPeriodicEven <<<blocks, 1024 >>>(this->spins_d, this->
   demons_d, this->L1, this->L2);
metropolis2DPeriodicOdd <<<blocks, 1024 >>>(this->spins_d, this->
   demons_d, this->L1, this->L2);

cudaDeviceSynchronize();

random_offset = this->random_generator->getRandomInteger();
shiftDemonsBlockWide2D << <blocks, 1024 >> > (this->demons_d,
   random_offset, this->L1, this->L2);

random_offset = this->random_generator->getRandomInteger();
shiftDemonsGridWide2D << <blocks_red, 1024 >> >(this->demons_d,
   random_offset, this->L1, this->L2);

cudaDeviceSynchronize();
```

Programmcode 5.3: Entnommen aus Abschnitt 8.2.2.8

gegeben. Zunächst werden durch `metropolis2DPeriodicEven` und `metropolis2DPeriodicOdd` die lokalen Updates über die Energieaustausche der Spin-Demon-Paare ausgeführt, entsprechend `...3D..` für ein dreidimensionales Gitter. Hierbei sollte durch die Bezeichnung

`metropolis...` nicht fälschlicherweise angenommen werden, dass es sich um den Metropolis-Algorithmus handelt. Die Bezeichnung ist nur gewählt worden, da auch hier lokale Updates von einzelnen Spins ausgeführt werden. Die Routinen updaten hierbei alle 32 Spinkonfigurationen.

Anschließend werden durch die restlichen Routinen die entsprechenden neuen Zuordnungen der *demons* an Gitterpunkten ausgeführt.

Betrachten wir uns zunächst die lokalen Updates der Spinzustände. Aufgrund der Tatsache, dass die einzelnen Bits eines Multispins unabhängig voneinander sind, können weitgehend alle Manipulierungen der einzelnen Bits der Mutltispins über bitweise Operationen ausgeführt werden und für das Verständnis ist das Verhalten des i-ten Bits ausreichend.

Vergleichen wir den Metropolis-Algorithmus in Abschnitt 3.2.1 mit dem mikrokanonischen Update in Abschnitt 3.2.2.1, so erkennen wir, dass beide Verfahren die Energiedifferenz $H_S(\sigma') - H_S(\sigma)$ verwenden, die sich durch die Änderung des lokalen Spinzustandes ergibt. Somit ist es offensichtlich, dass wir, analog zum Metropolis-Algorithmus, auch in diesem Fall die lokalen Updates einmal für gerade und für ungerade Gitterpunkte ausführen, damit keine Abhängigkeiten der lokalen Manipulationen der Spinzustände zueinander entstehen. Dies erklärt die Existenz der beiden *kernel*, `metropolis2DPeriodicEven` und `metropolis2DPeriodicOdd`.

Für die Ausführung dieser beiden *kernels* werden *thread blocks* mit 1024 *threads* gewählt, wobei jeder *thread* die Ausführung des lokalen Updates für einen Multispins an einem geraden, bzw. ungeraden, Gitterpunkt übernimmt. Ferner wird jeder Gitterblock durch ein *thread block* bearbeitet.

Zu Beginn der Routinen berechnet jeder *thread* die Koordinaten des ihm zugeordneten Multispins. Hierzu wird die Funktion `indexToCoordinates2DEven(threadIdx.x)`, bzw. `indexToCoordinates2DOdd(threadIdx.x)`,verwendet, die entweder, dem Namen der Funktion zu entnehmen, den *thread*-Index auf ein geraden oder ungeraden Gitterpunkt des Gitterblockes abbildet. Der Vorgang ist in Abbildung 5.3.2 veranschaulicht. Es werden beim Blockursprung beginnend, die geraden, bzw. ungeraden, Gitterpunkte über den Wert von `threadIdx.x` in positiver x-Richtung durchnummeriert, bis der äußere Rand des Gitterblocks erreicht ist und anschließend bei der nächst höheren y-Koordinate und $x = 0$ angefangen wird weiter in positiver x-Richtung zu nummerieren.

Am Anfang der Ausführung eines *threads* lädt je nach *kernel* jeder *thread* dem ihn zugeordneten Multispin eines geraden oder ungeraden Gitterpunktes. Um jedoch das mikrokanonische lokale Update der Mutlispins auszuführen, müssen zusätzlich die Multispins der nächst benachbarten Gitterpunkte geladen werden. Dieser Vorgang geschieht mit Hilfe des *Shared Memory*. Zunächst laden alle *threads* eines *thread blocks* die ihnen zugeordneten Multispins in den *Shared Memory* und anschließend werden die restlichen Multispins an den anderen Gitterpunkte, die ungerade oder gerade sind, in den *Shared Memory* geladen. Nach diesem Ladevorgang liegt der gesamte *thread block* im *Shared Memory*. Zusätzlich müssen unter Beachtung der periodischen Randbedingungen die nächste benachbarten Gitterpunkte, die nicht im Gitterblock liegen, in den *Shared Memory* geladen werden. Entsprechend, werden für jeden *thread*, die *demons* des zugeordneten Multispins, die upgedatet werden sollen, aus dem globalen in den lokalen Speicher geladen.

Hieran erkennen wir, weshalb der Gitteraufbau im Speicher und die Restriktionen für

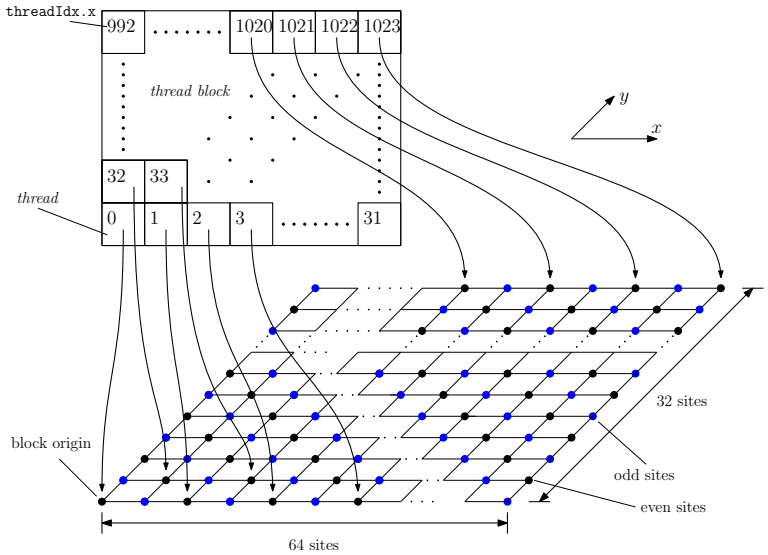


Abbildung 5.3.2: Darstellung der Zuordnung von *threads* eines *thread blocks* auf die einzelnen Gitterpunkte eines Gitterblockes für die Funktion `indexToCoordinates2DEven(threadIdx.x)`

L_1 und L_2 derartige gewählt sind. Es wird damit die *coalesced access* Eigenschaft für das Laden der Daten aus dem globalen Speicher garantiert, weil ein *warp* lädt 128 bytes Multispins an geraden, bzw. ungeraden, Gitterpunkten in den *Shared Memory*, wobei die zu ladenden Daten nebeneinander im Speicher lokalisiert sind. Für die *demons* gilt dies analog.

Nachdem dies geschehen ist, kann ein *thread* jeweils dem ihn zugeordneten Multispin und *demon* und die nächsten Nachbarn in seinen lokalen Speicher laden und anschließend den Energieaustausch zwischen den Spin-Demon-Paaren ausführen. Nachdem der zugeordnete Multispin entsprechend manipuliert worden ist, wird der neue Multispin und *demon* zurück in den globalen Speicher geschrieben.

Der eigentliche Energieaustausch geschieht innerhalb eines *threads* über folgenden Programmcode:

```
/*
compute if spin change is possible.
*/
multint7_t sum_energy;
multint7_t energy_change;
multint7_t demon_energy;

sum_energy = localSpinEnergy2D(own_multispin, neighbour_multispins);
demon_energy = demonEnergy7(own_demons);

energy_change = sum_energy + sum_energy;
//demon energy after spin flip
demon_energy = demon_energy + energy_change;
```

```
unsigned int flip_decision;

// if demon_energy is negative, no flip is allowed, bit is set to 0;
flip_decision = ~(demon_energy.sign);

//if demon energy is bigger than 16+8+4 = 28 (=0011100) no flip allowed.
flip_decision = flip_decision & ~(demon_energy.bit[5]);
flip_decision = flip_decision & (~(demon_energy.bit[4] & demon_energy.
   bit[3] & demon_energy.bit[2]) | (~(demon_energy.bit[1]) & ~(
   demon_energy.bit[0])));

//flip spins
own_multispin = own_multispin ^ flip_decision;


//cast new demon_energy to demons3_t if spin changed.
own_demons.ldemons = ((own_demons.ldemons ^ demon_energy.bit[2]) &
   flip_decision) ^ own_demons.ldemons;
own_demons.mdemons = ((own_demons.mdemons ^ demon_energy.bit[3]) &
   flip_decision) ^ own_demons.mdemons;
own_demons.hdemons = ((own_demons.hdemons ^ demon_energy.bit[4]) &
   flip_decision) ^ own_demons.hdemons;
```

Programmcode 5.4: Entnommen aus Abschnitt 8.2.2.6

Der Code ist weitgehend selbsterklärend. Es ist erkennbar, wie durch die bitweisen Operationen alle 32 Spinzustände gleichzeitig manipuliert werden. Es wird zu keinem Zeitpunkt in diesem Teil des Codes eine sequentielle Bearbeitung der Spinzustände ausgeführt. Um dies zu garantieren ist der Datentyp `multint7_t`, bzw. `multint17_t`, konstruiert worden. Eine Variable dieses Datentypes besteht aus sieben, bzw. 17, Speicherstellen des Datentyps `unsigned int`, wobei die i-ten Bits jeder Speicherstelle zusammen die i-te ganzzahlige, vorzeichenbehaftet, Zahl mit sieben, bzw. 17, Bits repräsentieren. Zwei solcher Variablen des Datentyps können miteinander addiert werden, wobei dies durch bitweise Operationen geschieht und die i-te Zahl des ersten mit der i-ten Zahl des zweiten Summanden addiert wird.

Nachdem die lokalen Updates ausgeführt sind, müssen die *demons* an neue Positionen gebracht werden. Dies geschieht durch die beiden *kernels* mit der Bezeichnung `shiftDemonsBlockWide2D` und `shiftDemonsGridWide2D`. Ferner wird eine Zufallszahl, die durch den Marsaglia-Zaman-Generator generiert wird, je *kernel* gebraucht. Bei der Verschiebung der *demons* werden stets die drei Speicherstellen `hdemons`, `mdemons` und `ldemons` gemeinsam einem neuen Gitterpunkt zugeordnet.

Durch den ersten *kernel* werden die Speicherstellen der *demons* eines Gitterblockes in ein Array im *Shared Memory* geladen und anschließend um einen zufälligen Wert verschoben, wobei Anfang und Ende des Arrays als miteinander verbunden betrachtet werden können.

Mit dem zweiten *kernel* werden die *demons* eines Gitterblockes mit den *demons* eines anderen Blocks vertauscht. Dies ist in Abbildung 5.3.3 exemplarisch für ein zweidimensionales Gitter dargestellt. Es werden zunächst die *demons* der Gitterblöcke der einen

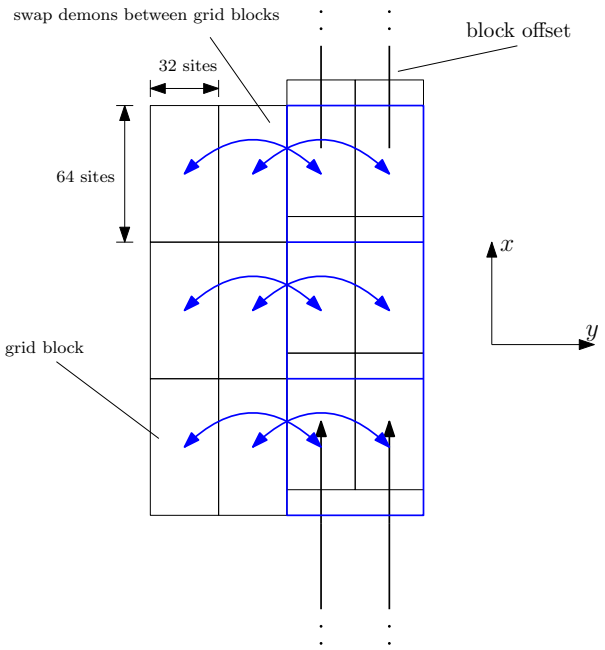


Abbildung 5.3.3: Darstellung der Verschiebung von *demons* über das Gitter hinweg für ein zweidimensionales Gitter. Blau gefärbte Linien stellen den Zustand nach der Verschiebung um einen zufälligen Offset in x-Richtung dar.

Hälfte des Gitters um einen zufälligen Offset in x-Richtung verschoben. Nachdem dies geschehen ist, werden die *demons* der Gitterblöcke mit der selben y Koordinate zwischen den Hälften vertauscht. Für drei Dimensionen wird zusätzlich noch eine Verschiebung mit zufälligen Offset in z-Richtung ausgeführt.

5.3.3.2 Kanonisches Update der Demons

Zunächst besprechen wir das, in der Methode `gitter.updateDemonLayersNew()` implementierte, naive Update-Verfahren. Die Funktion ruft hierbei den *kernel* mit der Bezeichnung `updateDemon2D` auf, indem ein *grid* erstellt wird, wo jedem Gitterblock ein *thread block* mit 1024 *threads* zugeordnet ist. Hierbei lädt jeder *thread* die *demons* an den durch `indexToCoordinates2DEven(threadIdx.x)` und `indexToCoordinates2DOdd(threadIdx.x)` sich ergebenden Gitterpunkten in seine lokalen Speicher. Anschließend wird mit der Wahrscheinlichkeit $P^{(1)}(E_i)$ in Gleichung (3.2.17) jeweils für jeden Bit einer Speicherstelle der beiden *demons* entschieden, ob der Bit auf 1 gesetzt werden soll oder mit der Gegenwahrscheinlichkeit auf 0.

Hierzu werden Zufallszahlen benötigt, wofür der selbe Zufallsgenerator in der Funktion `simpleRandomInteger0` mit der selben Initialisierungsroutine `initializeRandoms`, wie beim Metropolis-Algorithmus verwendet wird.

Das Verfahren, dass mit der Binomialverteilung arbeitet, wird anders implementiert und bedarf der Nutzung der CPU des *host*-Systems. Zunächst müssen wir die neue An-

zahl n' an besetzten Energiezuständen der *demons* gemäß der Verteilung $B(n', E_i)$ in Gleichung (3.2.18) mit $N = 32 \cdot L_1 \cdot L_2$ erzeugen und dies sollte möglichst effizient für große N geschehen. Um dieses Ziel zu erreichen, wird das Verfahren zu Erzeugung von Zufallswerten gemäß der Binomialverteilung $B(n', E_i)$ nach [22, S. 242] implementiert. Dieses Verfahren besitzt eine Komplexität von $\mathcal{O}(\ln(N))$ und ist für dieses Programm auf der CPU implementiert.

Für das Updaten des Energieniveaus E_i der *demons* muss zunächst die aktuelle Anzahl n an besetzten Zuständen des Niveaus bestimmt werden.

Nach der Berechnung von n wird über die Verteilung $B(n', E_i)$ die neue Anzahl an besetzten *demons* bestimmt. Nun werden entsprechend entweder $|n - n'|$ zufällig besetzte oder nicht besetzte *demon*-Zustände entfernt oder besetzt. Hierzu wird ein Array des Datentyps `unsigned int` mit der Länge $L_1 \cdot L_2$ verwendet, wobei das Array im *host*-Speicher liegt und die Bezeichnung `random_pattern` trägt. Das entsprechende Array auf dem *device* wird mit der Bezeichnung `random_pattern_d` geführt.

Folgende Schritte werden zum Updaten des Energieniveaus E_i der *demons* ausgeführt:

1. Es wird die Anzahl an auszuführenden Änderungen $m = |n - n'|$ bestimmt. Für die Speicherstellen des Energieniveaus E_i müssen entweder m zufällige Bits, die den Wert 1 besitzen, entfernt oder m zufällige Bits, die den Wert 0 haben, auf den Wert 1 gesetzt werden. Dies wird über das Vorzeichen von $n - n'$ bestimmt.

2. Das Array `random_pattern` wird mit 0 initialisiert.

3. In diesem Schritt werden über den Marsaglia-Zaman-Generator m zufällige Bits innerhalb des Arrays `random_pattern` bestimmt, deren Wert auf 1 gesetzt wird.

4. Anschließend wird das Array `random_pattern` vom *host* in das Array `random_pattern_d` im globalen Speicher des *device* kopiert.

5. Es wird der *kernel* mit der Bezeichnung `applyPatternSet2D` für das Besetzen oder `applyPatternRemove2D` für das Entfernen von *demon*-Zuständen aufgerufen. Für den dreidimensionalen Fall werden die *kernels* mit `...3D..` verwendet. Beim Aufruf wird ein *grid* mit 1024 *threads* pro *thread block* erzeugt. Jedem *thread block* ist hierbei ein Gitterblock und jedem Gitterpunkt ist eine Speicherstelle aus `random_pattern_d` zugeordnet. Die einzelnen *threads* eines Gitterblocks laden jeweils die *demons* am geraden und ungeraden zugeordneten Gitterpunkt und den Wert aus dem Array `random_pattern_d` in den lokalen Speicher. Das i-te Bit der Speicherstelle für das Energieniveau E_i der *demons* wird je nach *kernel* auf $b = 0$ oder $b = 1$ gesetzt, falls das Bit den Gegenwert $\neg b$ besitzt und die Speicherstelle aus dem Array `radom_pattern_d` am i-ten Bit entsprechend auf 1 gesetzt ist. Anderenfalls findet keine Änderung des i-ten Bits statt.

6. Es wird die Anzahl m' an ausgeführten Änderungen bestimmt.

7. Der Wert m wird auf $m = m - m'$ gesetzt.

8. Falls $m > 0$ sein sollte wird das Verfahren ab dem 2. Schritt ausgeführt, ansonsten terminiert das Vorgang.

Wir erkennen, dass die gesetzten Bits in `random_pattern` zufällige *demon*-Bits vorschlagen, die falls möglich entweder gesetzt oder entfernt werden sollen. Dieses Grundverfahren zeigt jedoch gewisse Schwächen, die das Verfahren verlangsamen. Der Fall der hierbei auftreten kann ist, dass die Anzahl an gesetzten, bzw. nicht gesetzten Bits für das Entfernen, bzw. Setzen von Bits viel kleiner als die Gesamtanzahl an Bits ist, womit eine sehr kleine Akzeptanzwahrscheinlichkeit für die Vorschläge in `random_pattern` entsteht, was zu einem häufigeren Ausführen der Schritte 2 bis 8 führt. Dieser Umstand wird in der Implementierung verbessert. Wir betrachten und dies für den Fall, dass gesetzte Bits entfernt werden sollen. Im anderen Fall gelten die Argumentationen entsprechend.

Ist für das Energieniveau E_i das Verhältnis $p = \frac{N_b}{N}$ zwischen gesetzten Bits N_b und der Gesamtanzahl N an *demon*-Bits, größer oder gleich 0.5, so wird das Verfahren wie oben ausgeführt. In dem Fall, dass das Verhältnis p geringer als 0.5 sein sollte, werden im 2. Schritt nicht m Bits des `random_pattern` gesetzt, sondern $\left\lfloor \frac{m}{p} \right\rfloor$ Bits, womit die Wahrscheinlichkeit, dass mehr Vorschläge im 5. Schritt akzeptiert werden erhöht wird. Der Wert für p bleibt jedoch in jeder Ausführung von Schritt 2 bis 8 für ein Update der selbe und entspricht dem Wert, der sich beim ersten Durchlauf ergibt.

Somit können zwar die Anzahl an Durchläufen der Schritte 2 bis 8 verringert werden, jedoch kann es sein, dass nach Beendigung des Verfahrens zu viele Bits entfernt wurden und $m < 0$ gilt. Dies wird korrigiert, indem entsprechend mit dem Grundverfahren $-m$ Bits hinzugefügt werden. Für dieses Hinzufügen ist das Verhältnis größer als 0.5. Es sei noch angemerkt, dass der Faktor $\frac{1}{p}$ willkürlich gewählt ist und es auch möglich ist andere Faktoren zu wählen, wobei für diese gelten sollte, dass kleiner p einen größeren Faktor bewirken.

5.3.4 Berechnung der totalen Magnetisierung

Die totale Magnetisierung wird aus den Spinkonfigurationen mit selber Methode, wie für den Metropolis-Algorithmus, bestimmt. Das Aufrufschema der einzelnen Funktionen und *kernel* ist identisch. Die entsprechenden *kernel* sind nur anders programmiert, jedoch berechnen in beiden Implementierungen die *kernel* die Magnetisierungen zunächst für die einzelnen Gitterblöcke aus und auf dem *host*-System werden die einzelnen totalen Magnetisierungen aufsummiert. Der *kernel* mit der Bezeichnung `totalMagnetizationBlockWide2D` summiert dabei analog, wie beim Metropolis-Algorithmus, die einzelnen totalen Magnetisierungen innerhalb jedes *warps* und anschließend über die Werte der einzelnen *warps* auf.

Kapitel 6

Analyse und Vergleich der Implementierungen

In diesem Kapitel werden die einzelnen Implementierungen bezogen auf ihre Laufzeit verglichen und anschließend die Funktionalität der Programme durch Simulationen eines zweidimensionalen Gitters verifiziert. Hierbei bezeichnet die Gitterausdehnung L die Ausdehnung eines Gitters in beide Richtungen.

6.1 Laufzeitanalyse der Implementierungen

In diesem Abschnitt soll die Laufzeit der einzelnen Algorithmen, bzw. Implementierungen, zueinander verglichen werden. Hierzu werden die einzelnen Zeiten für die Berechnung von Spinkonfigurationen bei unterschiedlichen Gittergrößen gemessen. Konkret werden die Update-Verfahren für die Erzeugung von N Spinkonfigurationen bei gegebenen Anfangswerten jeweils ausgeführt und dabei die Zeit t zur Ausführung der Updates ermittelt. Der Durchsatz an Spinkonfigurationen ergibt sich aus dem Verhältnis $\frac{N}{t}$. Beim Demon-Algorithmus muss beachtet werden, dass dort bei jeder Ausführung eines vollständigen Updates der Multispins 32 Spinkonfigurationen erzeugt werden. In den Programmen werden hierbei keine Zwischenberechnungen ausgeführt, sondern nur der Anzahl entsprechend die notwendigen Update-Methoden für ein Objekt der Klasse `Lattice` aufgerufen.

Ein derartiger Vergleich von Algorithmen zueinander ist stets kritisch zu betrachten und mit Vorsicht zu genießen. Dies liegt daran, dass die ermittelten Zeiten vom verwendeten System abhängig sind. Hierbei beeinflussen fast alle Komponenten des Systems und alle auf dem System parallel laufenden Prozesse, das Betriebssystem mit eingeschlossen, die Werte für die Zeiten. Auf einen anderen System kann die selbe Implementierung mit den selben Anfangswerten unterschiedliche Zeiten liefern, womit die Reproduzierbarkeit der Laufzeitanalysen eingeschränkt ist. Zusätzlich hängen die bestimmten Zeiten von der Implementierung selber und von der Umsetzung des Compilers ab. Es besteht keine Garantie, dass die hier getroffenen Implementierungen die bestmögliche Umsetzung der Algorithmen sind.

Nichtsdestotrotz können wir eine qualitativen Vergleich, bezogen auf die Laufzeit,

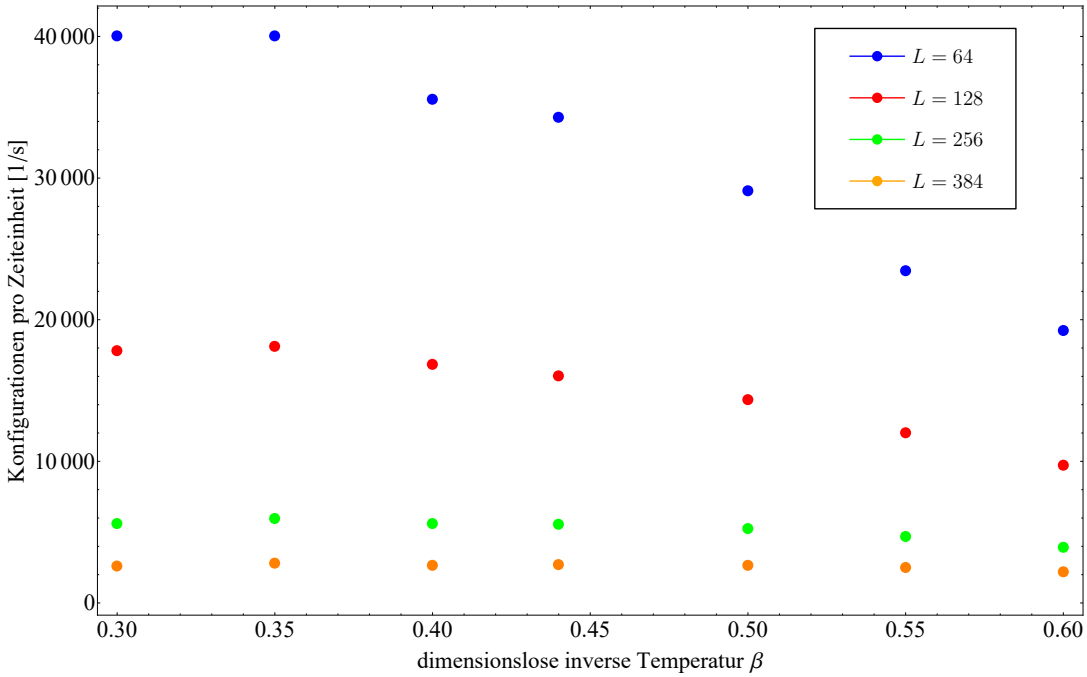


Abbildung 6.1.1: Es sind die berechneten Konfigurationen pro Zeiteinheit in Abhängigkeit der dimensionslosen inversen Temperatur β für unterschiedliche Gitterausdehnungen L für den Demon-Algorithmus angegeben. Hierbei wird der kanonische Update der *demons* mit Hilfe der Binomialverteilung verwendet.

zwischen den Implementierungen ausführen. Die Ausführung der Programme geschieht hierbei auf der GPU mit der Bezeichnung *GTX 980 Ti* vom Hersteller *Nvidia* und der CPU mit der Bezeichnung *Intel Core i7-5820K*. Ferner werden die Simulationen auf einem zweidimensionalen Gitter ausgeführt.

Zunächst vergleichen wir die beiden Implementierungen des Demon-Algorithmus mit den unterschiedlichen Update-Verfahren für die *demons*. Die Resultate des kanonischen Updates der *demons* mit Hilfe der Binomialverteilung ist in Abbildung 6.1.1 und mit Hilfe des naiven Verfahrens in Abbildung 6.1.2 dargestellt. Der erste Unterschied zwischen beiden Implementierungen ist, dass die Anzahl an Konfigurationen pro Zeiteinheit für das naive Verfahren unabhängig von der dimensionslosen inversen Temperatur β ist, während sie für das Verfahren mit Hilfe der Binomialverteilung offensichtlich temperaturabhängig ist und für steigende Werte von β die Anzahl an Konfigurationen anscheinend abnimmt. Das naive Update-Verfahren wiederum ist unabhängig von der Temperatur, da stets die selbe Anzahl an Instruktionen ausgeführt wird.

Ferner können wir erkennen, dass das naive Verfahren einen wesentlich höheren Durchsatz an Konfigurationen pro Zeiteinheit besitzt und insbesondere für größere Gitter der Durchsatz für das Verfahren mit Hilfe der Binomialverteilung in die Größenordnung von 1000 gelangt. Diese rapide Abnahme des Durchsatz ist mit hoher Wahrscheinlichkeit mit der Implementierung des Verfahrens begründet. Das Update-Verfahren mit Hilfe der Bi-

nomialverteilung benutzt sowohl die CPU als auch die GPU für die Berechnungen und überträgt Daten zwischen *host* und *device* über die Funktion `cudaMemcpy(...)`. Hierbei werden für das Update eines Energieniveaus der *demons* m Arrays mit der Größe des Gitters vom *host* zum *device* übertragen. Zwar dauert jede einzelnen Übertragung nicht sonderlich lange, jedoch führt der Faktor m zu einer entsprechenden Aufsummierung der Zeiten. Es muss hierzu berücksichtigt werden, dass der Datentransfer zwischen *host* und *device* die langsamste Art an Speichertransaktionen für die GPU-Programmierung darstellt. Das Resultat steht im Einklang mit der Tatsache, dass für größere β bei gleichem Gitter der Durchsatz abnimmt, da je geringer die Temperatur ist umso weniger sind die einzelnen Energieniveaus der *demons* besetzt, womit die Akzeptanzwahrscheinlichkeit für die Vorschläge der Änderung der *demons* abnimmt und somit m im Schnitt größere wird.

Es bleibt die Frage offen, weshalb die Implementierung des kanonischen Update-Verfahrens mit Hilfe der Binomialverteilung nicht vollständig auf der GPU ausgeführt wird, um die Transaktionen zwischen *host* und *device* zu minimieren. Das Problem hierbei ist, dass die Manipulation eines *demon* in diesem Verfahren abhängig von der Besetzung des *demon* ist und insbesondere zufällige *demons* mit einer bestimmten Besetzung ausgesucht werden müssen, die wiederum statistisch verteilt sind. Diese Tatsache führt zur der Problematik, dass die Umsetzung auf die GPU erschwert wird, weil die einzelnen *threads* müssten zufällig auf entsprechende *demons*-Zustände zugreifen, wobei der Zugriff unter Umständen außerhalb des zugeordneten Gitterblockes geschehen müsste. Dies kann zu einer ineffizienten Benutzung der *threads* aufgrund von unterschiedlichen Programmabläufen im Rahmen der *warp divergence* führen. Zusammengefasst ist das Problem, dass auf bestimmte an zufälligen Stellen liegenden Daten zugegriffen werden muss und von diesen wiederum zufällig entschieden werden muss welche manipuliert werden. Aufgrund der Konstruktion einer GPU, ist diese vor allem auf Berechnungen spezialisiert, wo selbe Instruktionen parallel auf unterschiedliche Daten angewendet werden. Ein ähnliches Problem ergibt sich bei dem Versuch Cluster-Algorithmen für das Ising-Modell auf die GPU umzusetzen. Bei diesem Typ an Algorithmen werden zunächst zwischen zwei benachbarten Gitterpunkten entweder Kanten gesetzt oder entfernt und jene Gitterpunkte, die jeweils über einen Pfad von Kanten miteinander verbunden sind, zu einem Cluster zusammengefasst. Anschließend werden für die Spins eines Clusters entschieden, ob diese geflippt werden sollen oder nicht. Für genauere Betrachtungen sei beispielsweise auf [5] verwiesen. Im Rahmen dieser Arbeit bestand der Versuch auch einen Cluster-Algorithmus auf der GPU zu programmieren. Jedoch hat sich gezeigt, dass dies mit ähnlichen Schwierigkeiten verbunden ist, wie oben. Das Setzen der Kanten lässt sich vergleichsweise effizient implementieren, jedoch besteht das Problem in der Identifizierung der Cluster, denn die Kanten sind zufällig verteilt und insbesondere müssen Gitterpunkte entlang von Pfaden, bestehend aus mehreren Kanten, dem selben Cluster zugeordnet werden. Es bestand der Versuch mit Hilfe der Verfahren zur Identifikation von Clustern auf der GPU in [33] eine Variante der Cluster-Algorithmen zu implementieren. Die Versuche scheiterten jedoch an der vergleichsweise langsam Zeit für die Identifizierung der Cluster, die je nach Besetzung des Gitters mehrere Sekunden betragen konnte. Somit ist die Idee eines Cluster-Algorithmus für die GPU verworfen worden.

Insgesamt können wir erkennen, dass das kanonische Update der *demons* gemäß [20]

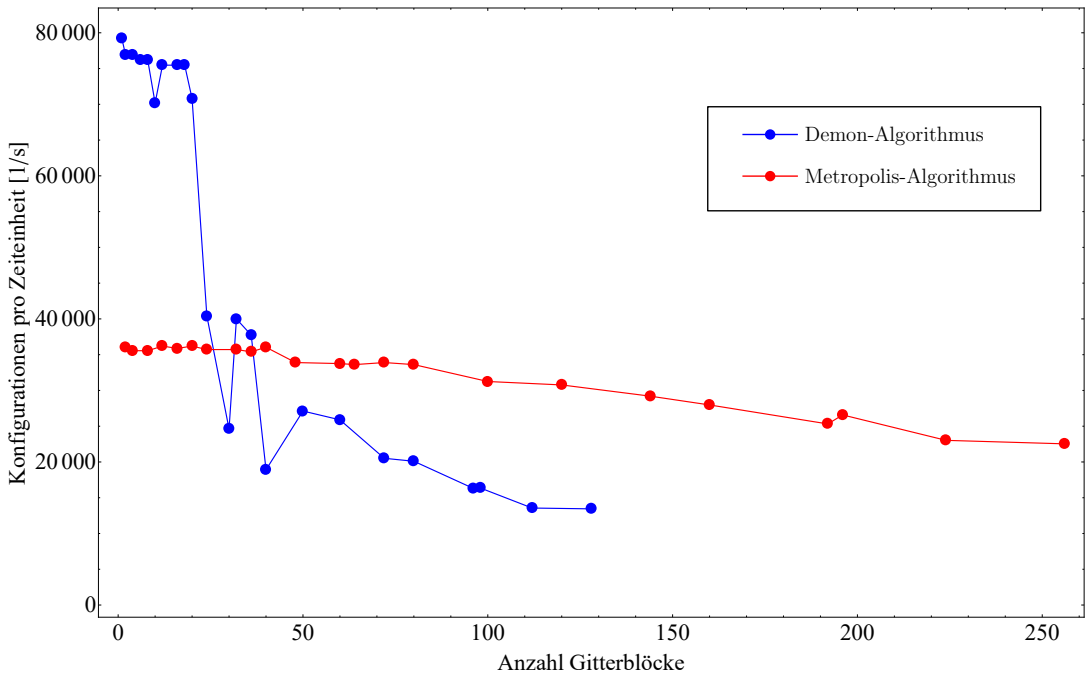


Abbildung 6.1.2: Berechneten Konfigurationen pro Zeiteinheit in Abhängigkeit der Anzahl an Gitterblöcken. Die Größe eines Gitterblocks beträgt für den Demon-Algorithmus 64×32 und für den Metropolis-Algorithmus 32×32 Gitterpunkte pro Block.

und [21] nicht ohne weiteres auf die GPU umgesetzt werden kann, sondern das naive Verfahren sich als, unter Umständen, schneller für die Gesamtsimulation in der Anzahl an Konfigurationen pro Zeiteinheit erweist. Abschließend vergleichen wir den Metropolis-Algorithmus mit dem Demon-Algorithmus, wobei das naive kanonischen Update der *demons* verwendet wird. Die Resultate des Laufzeitvergleiches sind in Abbildung 6.1.2 angegeben. Hierbei werden die Anzahl an Spinkonfigurationen pro Zeiteinheit gegen die Anzahl an Gitterblöcken, die jeweils durch ein *thread block* bearbeitet werden, aufgetragen. Beim Demon-Algorithmus beträgt die Größe der Gitterblöcke 64×32, während sie für den Metropolis-Algorithmus 32×32 beträgt. Wir können erkennen, dass für kleine Gitter die Implementierung des Demon-Algorithmus durchaus mehr als doppelt so viele Spinkonfigurationen pro Zeiteinheit liefert, jedoch für größer werdende Anzahl an Gitterblöcken ab ca. 30 Blöcken der Durchsatz an Spinkonfigurationen rapide abfällt und den Metropolis-Algorithmus in diesen Bereichen schneller ist. Vor allem ist zu erkennen, dass der Durchsatz des Metropolis-Algorithmus weniger stark bei steigender Anzahl an Gitterblöcken abnimmt. Die Ursache für diese Verlangsamung des Demon-Algorithmus, kann aller Wahrscheinlichkeit nach auf das naive kanonische Update-Verfahren zurück geführt werden. Dies lässt sich mit den Resultaten in Abbildung 6.1.3 begründen, wo die Ergebnisse des Durchsatz an Spinkonfigurationen für den Demon-Algorithmus ohne kanonischen Update dargestellt sind. In der Abbildung ist zu erkennen, dass der reine mikrokanonische Update um mehr als eine Größenordnung höheren Durchsatz besitzt und erst ab circa 500

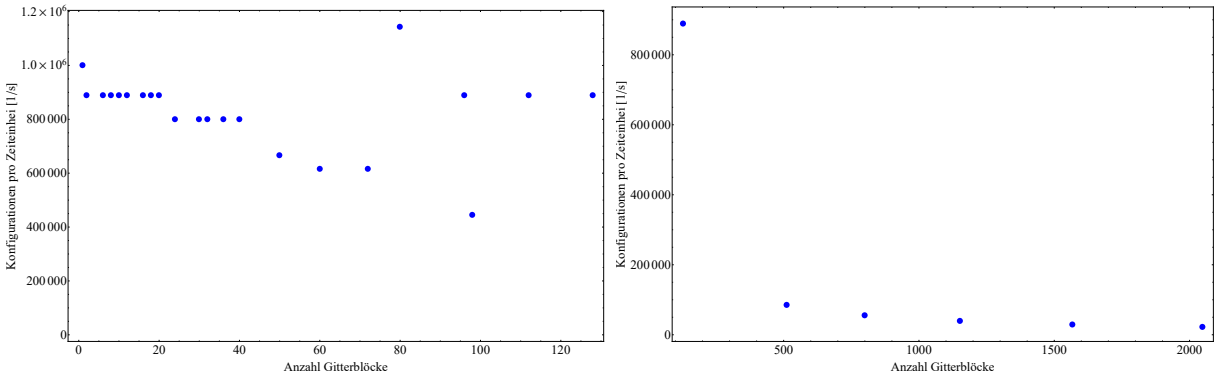


Abbildung 6.1.3: Anzahl an Spinkonfigurationen pro Zeiteinheit in Abhängigkeit der Anzahl der Gitterblöcke für den mikrokanonischen Update des Demon-Algorithmus.

Gitterblöcken vom Durchsatz her in die Größenordnung des Metropolis-Algorithmus und des Demon-Algorithmus in Abbildung 6.1.2 gelangt.

Die Verringerung des Durchsatz vom Demon-Algorithmus mit kanonischen Update ist mit hoher Wahrscheinlichkeit auf die Tatsache, dass im entsprechenden *kernel* der aufgerufen wird die einzelnen Bits eines *demon* nacheinander und nicht wie beim mikorkanonischen Update durch bitweise Operationen gleichzeitig manipuliert werden, zurückzuführen.

Es sollte jedoch beachtet werden, dass auch andere Faktoren eine Rolle spielen. Beispielsweise bestehen die *thread blocks* der Implementierung des Metropolis-Algorithmus aus jeweils 64 *threads*, während die der Implementierung des Demon-Algorithmus aus 1024 *threads* bestehen. Ferner brauchen die einzelnen *threads* der beiden Implementierungen unterschiedlich viele Ressourcen. Aus diesen Eigenschaften ergibt sich, wie viele *warps* gleichzeitig in einem SM abgearbeitet werden können.

In der Zusammenfassung dieser Analysen zeigt sich, dass der Metropolis-Algorithmus unter Umständen eine bessere Stabilität bezogen auf den Durchsatz an Spinkonfigurationen bei steigender Anzahl an Gitterblöcken besitzt und somit unter Umständen für kanonische Simulationen verwendet werden sollte. Jedoch erkennen wir, dass für die mikrokanonische Simulation, der Demon-Algorithmus besonders schnell zu seien scheint und unter Umständen für solche Betrachtungen verwendet werden kann. Insbesondere könnte die mikrokanonische Simulation benutzt werden, um für ausreichend große Gitterausdehnungen L den thermodynamischen Limes zu nähern und somit auch das kanonische Ensemble, aufgrund der Äquivalenz der Ensemble für große Systeme zu erhalten, [2]. Dies erfordert jedoch tiefgehende Analysen, insbesondere muss festgestellt werden ab welchem L die Näherung gelten sollte und in wie weit das hier implementierte mikrokanonische Update-Verfahren für eine reine mikrokanonische Simulation verhält.

6.2 Simulation des zweidimensionalen Gitters

Im vorherigen Abschnitt haben wir die Performance der Implementierungen miteinander verglichen. In diesem Abschnitt sollen nummerische Ergebnisse von Simulationen für ein zweidimensionales Gitter betrachtet und mit den theoretischen Erkenntnissen aus Kapitel 2 verglichen werden, um auf der einen Seite eine praktische Ausführung von Analysen in Rahmen des Ising-Modells mit verbunden Problematiken darzustellen und zum Anderen gleichzeitig die Funktionalität der implementierten Verfahren zu verifizieren.

Hierbei werden die Analysen für die Implementierung des Metropolis-Algorithmus und des Demon-Algorithmus ausgeführt. Aufgrund der Tatsache, dass das naive kanonische Update-Verfahren sich für den Demon-Algorithmus als schneller erwiesen hat im Vergleich zu der Implementierung mit der Binomialverteilung, wird das naive Update-Verfahren für die folgenden Analysen gewählt.

6.2.1 Qualitative Untersuchung der Implementierungen

Zunächst betrachten wir die Simulationen der mittleren Magnetisierung $\langle |m| \rangle_{h=0}$, der Suszeptibilität χ_L und der Binderkumulante U. Hierbei sei der Fokus zunächst auf die qualitativen Verhalten der Größen gelegt, anders ausgedrückt es wird zunächst keine Fehlerbetrachtung und keine Simulationen mit hoher Anzahl an Messwerten ausgeführt.

Es sind für unterschiedliche Temperaturen $\frac{kT}{J_c}$ jeweils die einzelnen Größen bestimmt. Hierbei ergibt sich an den jeweiligen Messpunkten der Wert der entsprechenden Größe über den arithmetischen Mittelwert von den aus den Konfigurationen berechneten Werten. Für den Metropolis-Algorithmus werden für jeden Messpunkt insgesamt 32000 Messwerte bestimmt, wobei die ersten 2000 verworfen werden, während für den Demon-Algorithmus 96000 Messwerte für die Mittlung jeweils verwendet und die ersten 2000 Konfigurationen von Multispins verworfen werden. Die Simulationen sind hierbei auf der GPU mit der Bezeichnung *GTX 970* vom Hersteller *Nvidia* und unterschiedlichen Gittergrößen ausgeführt worden. Die Resultate sind in Abbildung 6.2.1 bis 6.2.4 dargestellt.

Zunächst seien die Resultate für die mittlere Magnetisierung $\langle |m| \rangle_{h=0}$ betrachtet. In Abbildung 6.2.1 erkennen wir das Verhalten der Magnetisierung für unterschiedlich Gittergrößen. Die eingezeichnete schwarz gefärbte Kurve stellt hierbei den theoretischen Verlauf im thermodynamischen Limes aus Gleichung (2.2.20) dar. Die Linien mit denen die einzelnen Messwerte miteinander verbunden sind haben keine Bedeutung und dienen nur der bessern Unterscheidung der Kurven zueinander.

Die einzelnen Kurven für beide Implementierungen zeigen hierbei das typische Verhalten für $\langle |m| \rangle_{h=0}$ bei endlicher Gittergröße. Der Verlauf der Kurven folgt qualitativ dem des thermodynamischen Limes, wobei am Phasenübergang bei endlichen Gittern ein glatter Übergang von $T > T_c$ nach $T < T_c$ stattfindet, während im thermodynamischen Limes die Magnetisierung erst nach dem Phasenübergang ungleich null wird. Ferner erkennen wir, dass für größer werdende Gitterausdehnungen L die Kurven sich dem thermodynamischen Limes nähern, was zu erwarten ist. Insbesondere erkennen wir dass die berechneten Verläufe bei gleicher Gitterausdehnung L für beide Implementierungen weitgehend identisch sind. Dies spricht für eine Funktionalität der Implementierungen, da beide auf

unterschiedliche Verfahren aufbauen.

Der Verlauf der Suszeptibilität χ_L für die einzelnen Implementierungen ist in Abbildung 6.2.2 dargestellt. Auch hier können wir das typische Verhalten für endliche Gitter erkennen. In der Nähe der kritischen Temperatur T_c nimmt die Suszeptibilität χ_L für die einzelnen Kurven ein Maximum an und für steigende Gitterausdehnungen L nimmt der Wert des Maximums zu. Auf Basis des Finite-Size-Scaling-Gesetzes in Gleichung (2.2.27) können wir in der Nähe des Phasenübergangs auf die Gesetzmäßigkeit

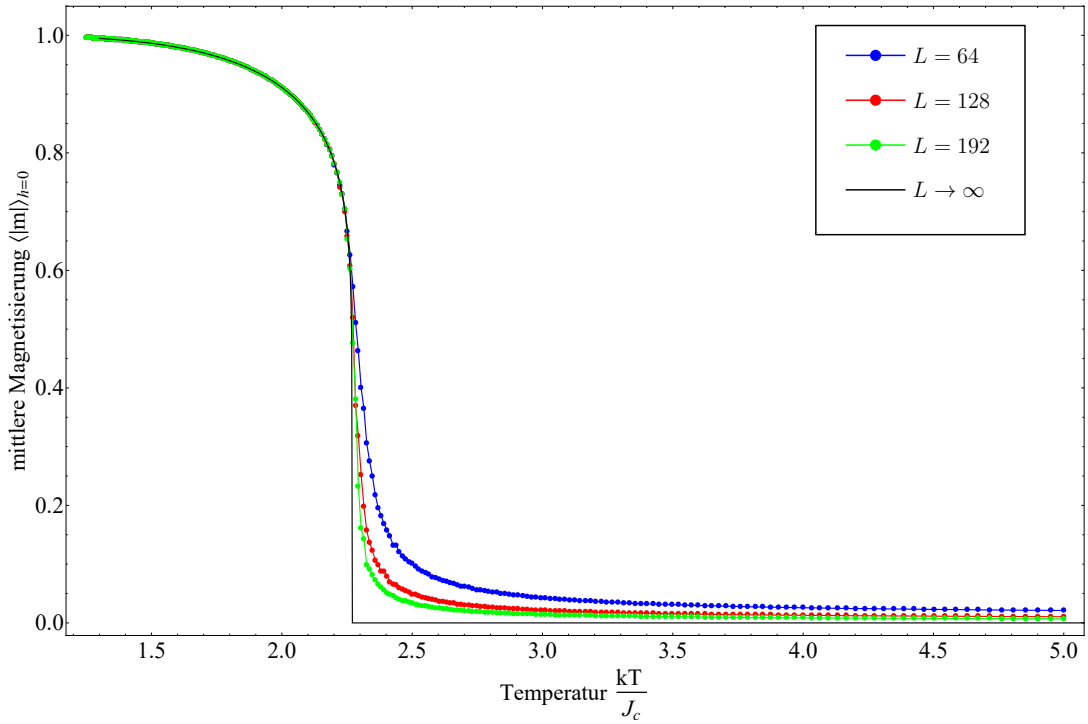
$$\frac{\chi_L}{L^{\frac{\gamma}{\nu}}} = g(tL^{\frac{1}{\nu}}) \tag{6.2.1}$$

schließen. Hieraus können wir folgern, dass, wenn wir $\frac{\chi_L}{L^{\frac{\gamma}{\nu}}}$ gegen $tL^{\frac{1}{\nu}}$ in einem Diagramm auftragen, einen Kurvenverlauf erhalten, der unabhängig vom Wert L ist. Anders ausgedrückt, müssen die einzelnen Werte, die sich aus den Simulationen der unterschiedlichen Gittergrößen ergeben, auf einer gemeinsamen Kurve liegen. Dieses Verhalten lässt sich in Abbildung 6.2.3, für die beiden Implementierungen, erkennen. Weiterhin ist es möglich zu erkennen, dass die Resultate für beide Implementierungen näherungsweise identisch sind, was in diesem Fall zusätzlich als Verifikation der Verfahren dient.

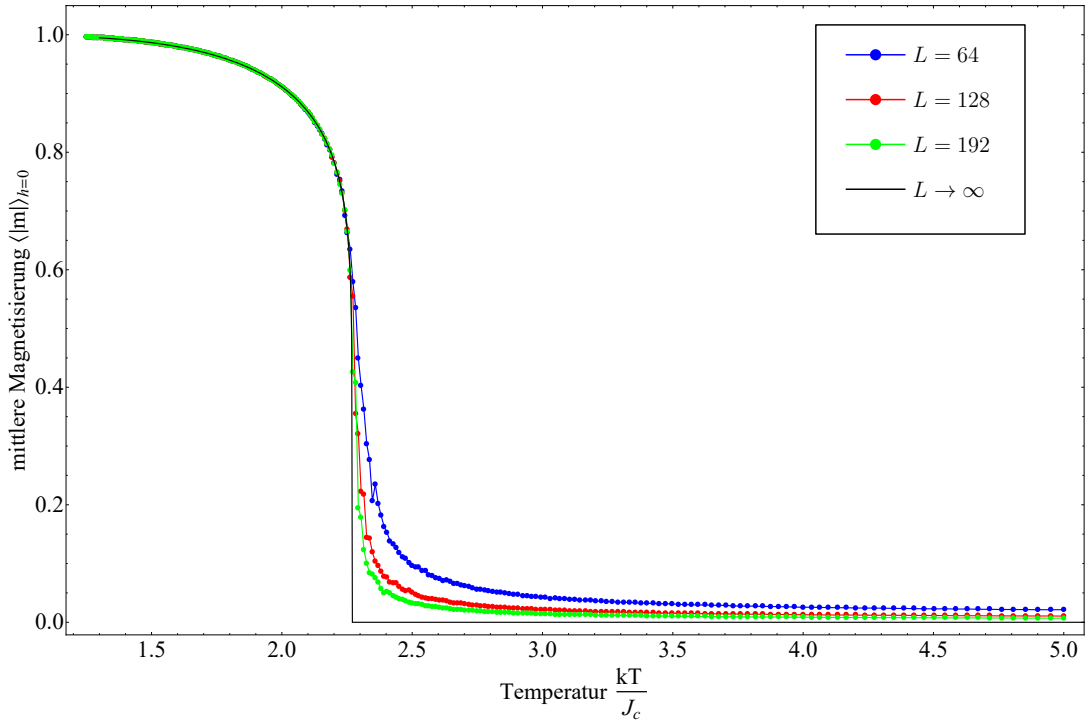
Die Resultate für die Binderkumulante U sind in Abbildung 6.2.4 dargestellt. In den Diagrammen ist zu erkennen, dass am Phasenübergang die Werte der Binderkumulante U der einzelnen Verläufe näherungsweise dem Wert U^* entsprechen, wobei jedoch relativ starke Schwankungen zu erkennen sind. Eine genauere Betrachtung der Binderkumulante folgt im nächsten Abschnitt.

Insgesamt reproduzieren die Implementierungen qualitativ das nicht triviale Verhalten des Ising-Modell für endliche zweidimensionale Gitter, was zu einer gewissen Sicherheit über die Funktionalität der Implementierungen führt.

Im folgenden Abschnitt werden zusätzliche quantitative Betrachtungen ausgeführt, um eine möglichst gute Verifikation der Implementierungen zu erhalten.

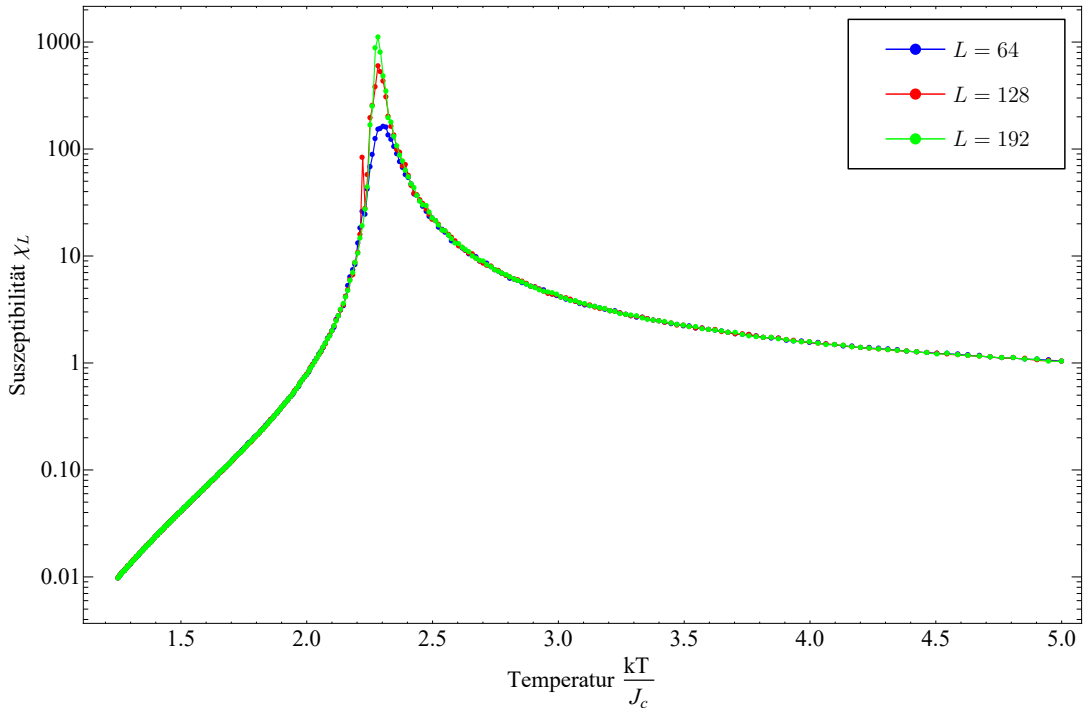


(a) Demon-Algorithmus

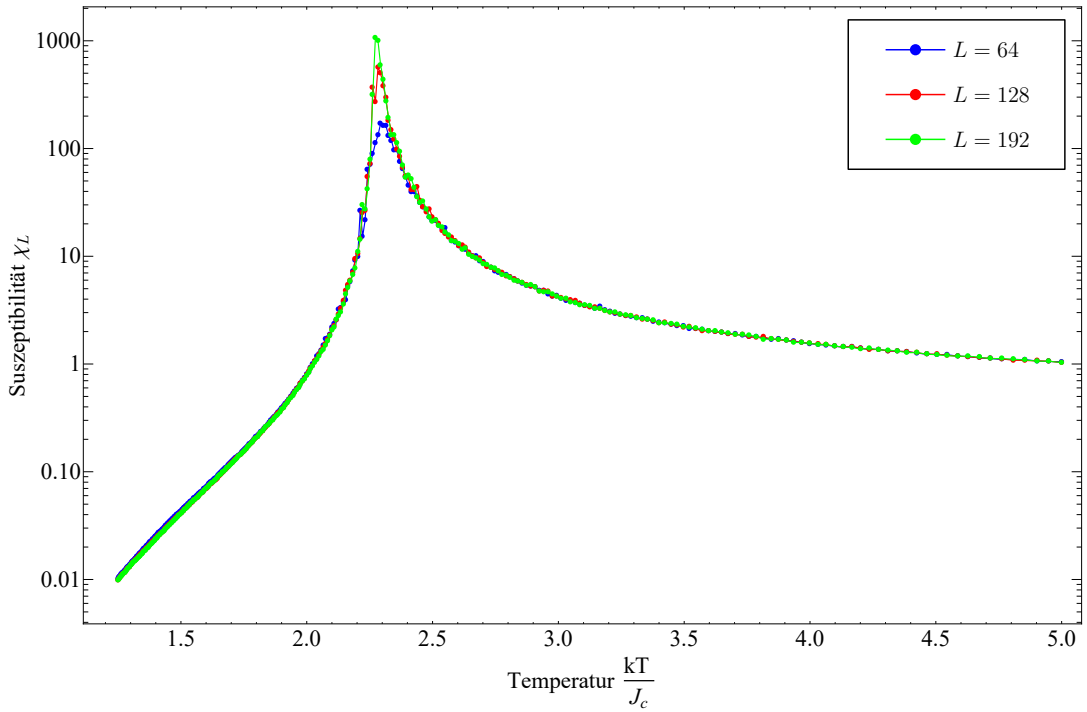


(b) Metropolis-Algorithmus

Abbildung 6.2.1: Mittlere Magnetisierung $\langle|m|\rangle_{h=0}$ in Abhängigkeit der Temperatur $\frac{kT}{J_c}$ für unterschiedliche zweidimensionale Gitterausdehnungen L, jeweils über die beiden Implementierungen bestimmt. In beiden Diagrammen ist jeweils der theoretische Verlauf aus Gleichung (2.2.20) dargestellt.

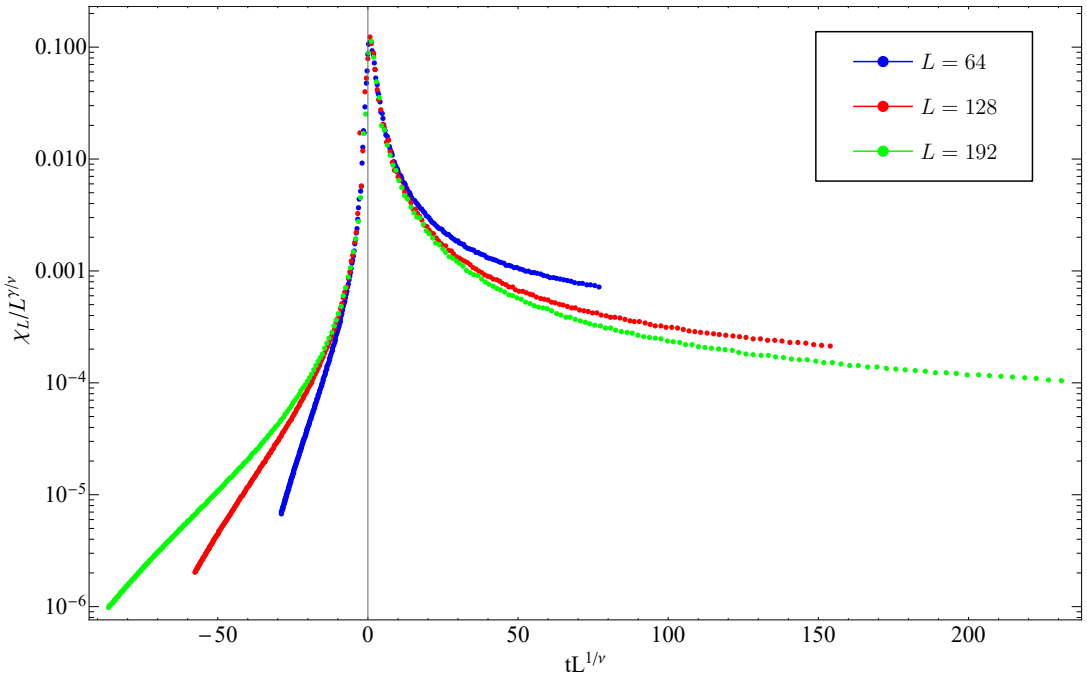


(a) Demon-Algorithmus

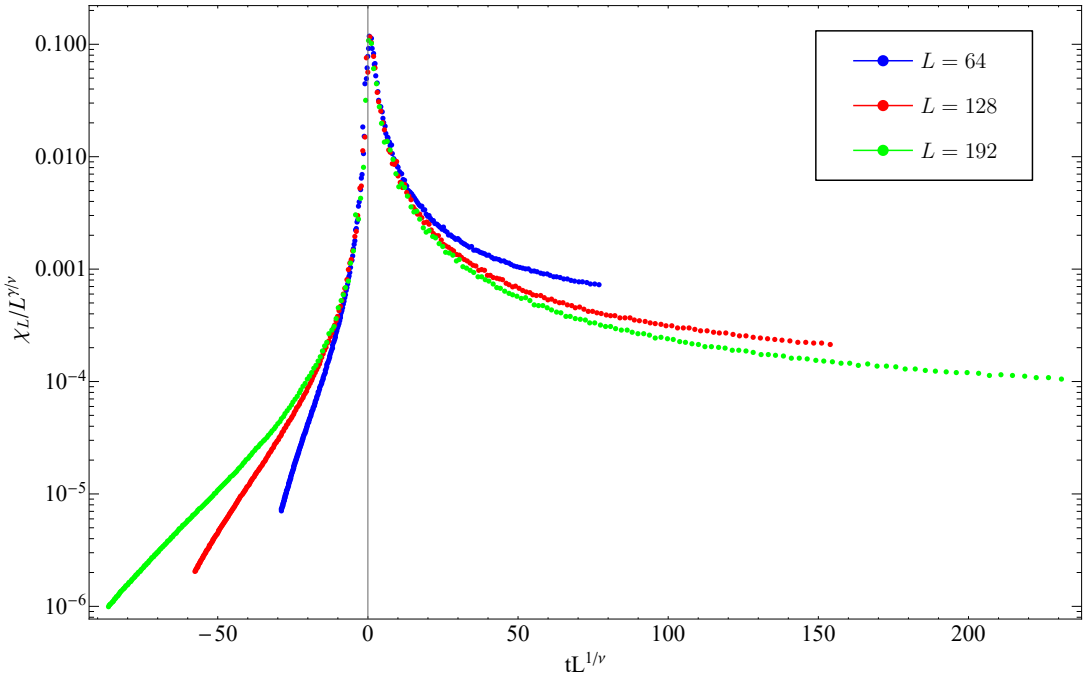


(b) Metropolis-Algorithmus

Abbildung 6.2.2: Suszeptibilität χ_L in Abhängigkeit der Temperatur $\frac{kT}{J_c}$ für unterschiedliche zweidimensionale Gitterausdehnungen L, jeweils über die beiden Implementierungen bestimmt.

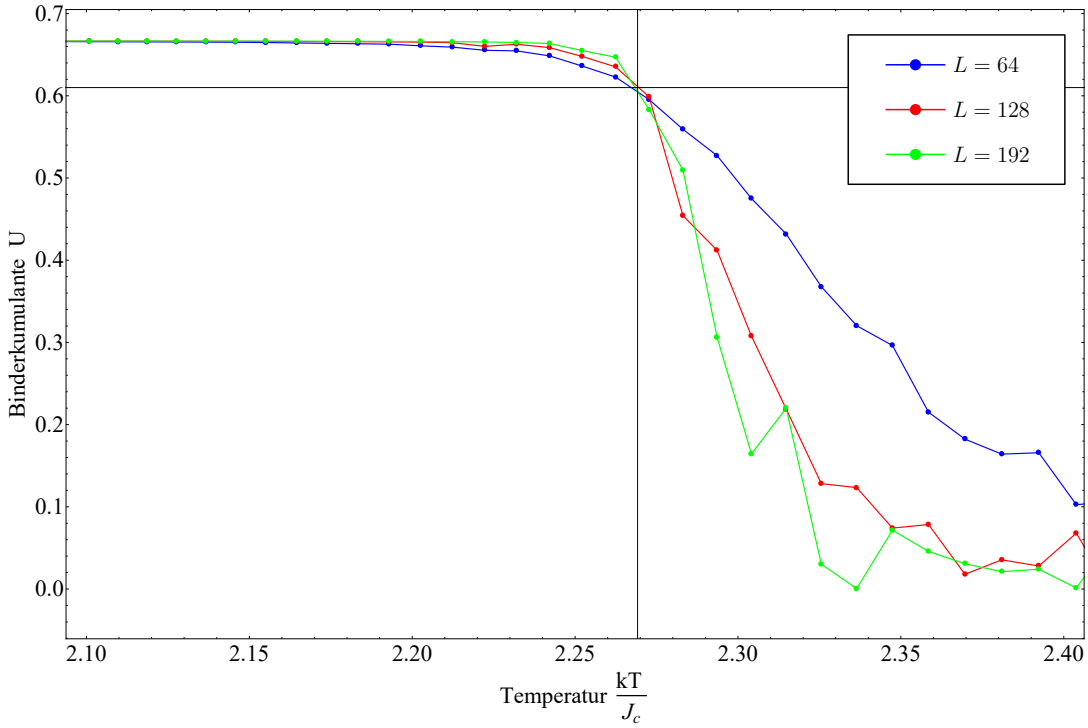


(a) Demon-Algorithmus

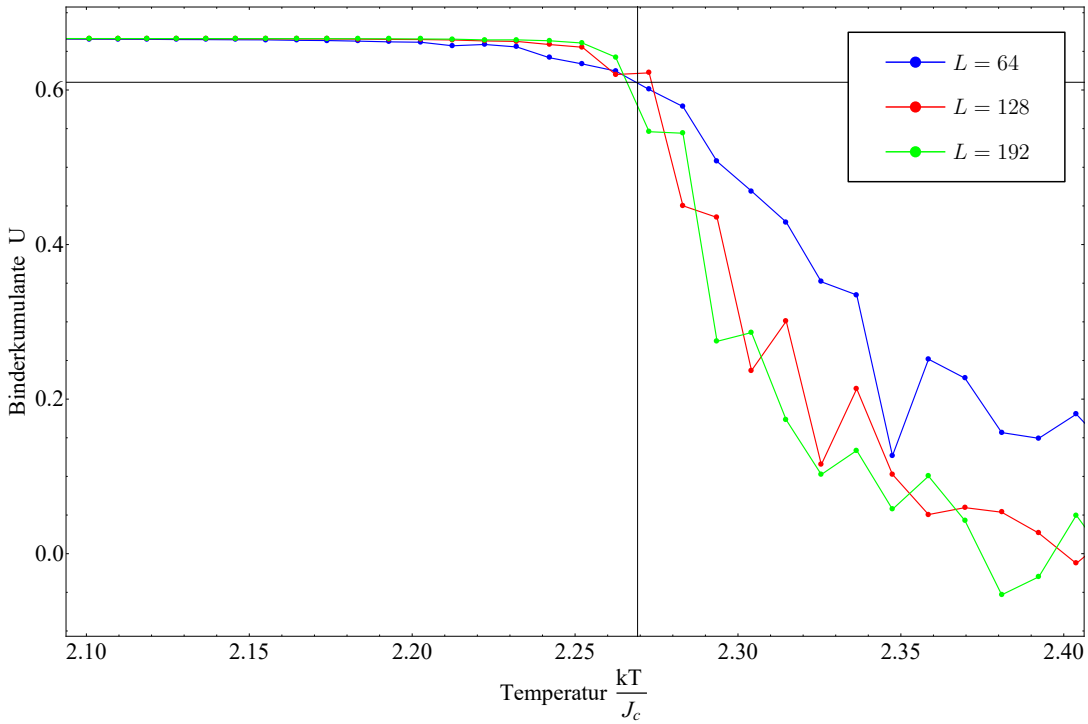


(b) Metropolis-Algorithmus

Abbildung 6.2.3: $\chi_L/L^{\frac{\gamma}{\nu}}$ in Abhängigkeit von $tL^{\frac{1}{\nu}}$ für unterschiedliche zweidimensionale Gitterausdehnungen L, jeweils über die beiden Implementierungen bestimmt.



(a) Demon-Algorithmus



(b) Metropolis-Algorithmus

Abbildung 6.2.4: Binderkumulante U in Abhängigkeit der Temperatur $\frac{kT}{J_c}$ für unterschiedliche zweidimensionale Gitterausdehnungen L, jeweils über die beiden Implementierungen bestimmt. Ferner ist in beiden Diagrammen der Kreuzungspunkt bei der kritischen Temperatur mit dem Wert $U^* \approx 0.61067$ eingezeichnet.

6.2.2 Quantitative Untersuchungen der Implementierungen

In diesem Abschnitt werden mit Hilfe der beiden Implementierungen das Verhalten der Binderkumulante U genauer betrachtet und zusätzlich der kritische Exponent γ nummerisch bestimmt und mit dem bekannten theoretischen Wert $\gamma = \frac{7}{4}$ verglichen. Hierzu werden auch grundlegende Fehlerbetrachtungen für die statistischen Größen ausgeführt.

6.2.2.1 Statistischer Fehler und Autokorrelation

Aufgrund der Tatsache, dass im Rahmen der Monte-Carlo-Simulationen statistisch verteilte Messwerte gemittelt werden, unterliegen diese einer statistischen Schwankung, die in Form einer Fehlerangabe zu berücksichtigen ist.

Gegeben sei ein Obersvable $A(\sigma)$. Im Rahmen der nummerischen Simulationen wird bei gegebener dimensionsloser inverser Temperatur β eine Messreihe von N Werten $A_i = A(\sigma_i)$ der einzelnen Konfigurationen ausgeführt. Für ausreichend große N wissen wir, dass die Näherung

$$\bar{A}_N = \frac{1}{N}\sum_{i=1}^{N} A_i \approx \langle A \rangle \tag{6.2.2}$$

zutrifft. Hierbei tritt jedoch das Problem auf, dass der Mittelwert $\bar{A}_N$ eine statistische Größe ist und somit mit einer gewissen Wahrscheinlichkeit in einem bestimmten Bereich um den Wert $\langle A \rangle$ auftrifft. Für ausreichend große N ist der Mittelwert $\bar{A}_N$ um den Erwartungswert normal verteilt, mit der Standardabweichung $\sigma(\bar{A}_N)$, siehe [6], womit diese als sinnvolle Fehlerangabe verwendet werden kann, da sie eine Sicherheit darüber gibt mit welcher Wahrscheinlichkeit der berechnete Mittelwert sich in einer bestimmten Umgebung des Erwartungswertes befindet. Für den Fall, dass die einzelnen Messungen der Werte A_i statistisch unabhängig voneinander sind ergibt sich der einfache Zusammenhang (siehe [15])

$$\sigma^2(\bar{A}_N) = \frac{\sigma^2(A)}{N} \tag{6.2.3}$$

wobei für ausreichend große N der Wert $\sigma(A)$ abgeschätzt werden kann zu (siehe [6])

$$\sigma^2(A) \approx \frac{1}{N-1}\sum_{i=0}^{N}(A_i - \bar{A}_N)^2 =: S_{(A_i)}(\bar{A}_N) \tag{6.2.4}$$

Bei den hier ausgeführten Simulationen werden jedoch Markov-Prozesse verwendet, womit die einzelnen Konfigurationen in der Kette $(\sigma_1, \ldots, \sigma_N)$ nicht zueinander statistisch unabhängig sind und somit auch nicht die Werte A_i. In diesem Fall muss diese Autokorrelation der Werte berücksichtigt werden, was zu einer Korrektur der Form

$$\sigma^2(\bar{A}_N) = \frac{\tau_{int.}}{N}\sigma^2(A) \tag{6.2.5}$$

führt, wobei $\tau_{int.}$ die sogenannte integrierte Autokorrelationszeit ist. Für genauere Betrachtungen sei beispielsweise auf [15] verwiesen. Die Aufgabe ist es somit aus den Messwerten A_i den Wert $\bar{A}_N$ und eine Abschätzung von $\sigma(\bar{A}_N)$ zu berechnen.

In den folgenden Betrachtungen wird der Fehler nicht über die direkte Ermittelung von $\tau_{int.}$ bestimmt, sondern über die Binning-Methode. Die Idee hinter der Methode ist es benachbarte Messwerte A_i in einzelnen Bereichen (*bins*) einzuteilen und die Mittelwerte dieser zu berechnen und als Messreihe A'_i zu interpretieren. Hierbei ergibt sich der Wert A'_k zu (siehe [6])

$$A'_k = \frac{1}{N_B} \sum_{i=N_B(k-1)+1}^{kN_B} A_i \tag{6.2.6}$$

wobei N_B die Anzahl an benachbarten Messwerten pro Bereich ist. Für ausreichend große N_B werden die einzelnen A'_k näherungsweise zueinander statistisch unabhängig, womit über Gleichung (6.2.3) der Fehler $\sigma(\bar{A}_N)$ abgeschätzt werden kann, indem die Werte A'_k verwendet werden. Generell gilt hierbei die Beziehung (siehe [15]):

$$\tau_{int.} = \lim_{N\to\infty} \tau_N = \lim_{N\to\infty} \frac{S_{(A'_k)}(\bar{A}_N)}{S_{(A_k)}(\bar{A}_N)} \tag{6.2.7}$$

wobei die Anzahl $n = \frac{N}{N_B}$ an *bins* konstant gehalten wird. Die Schwierigkeit bei der Bestimmung des Fehlers ist es zu entscheiden, ob die Anzahl n an *bins* und ob die Anzahl N an Messwerten ausreichend gut gewählt ist, sodass der Wert $\bar{A}_N$ und $\sigma(\bar{A}_N)$ zuverlässige ist. Es existieren hierzu keine allgemeinen Verfahren die angewendet werden können. Vielmehr kann beispielsweise nachträglich die integrierte Autokorrelationszeit für die gewählte *bin*-Größe n über Gleichung (6.2.7) abgeschätzt und mit gewissen Grundregeln verglichen werden, um eine plausible Begründung für den bestimmten Fehler zu erhalten. Gemäß [15] sollte hierbei als Grundregel:

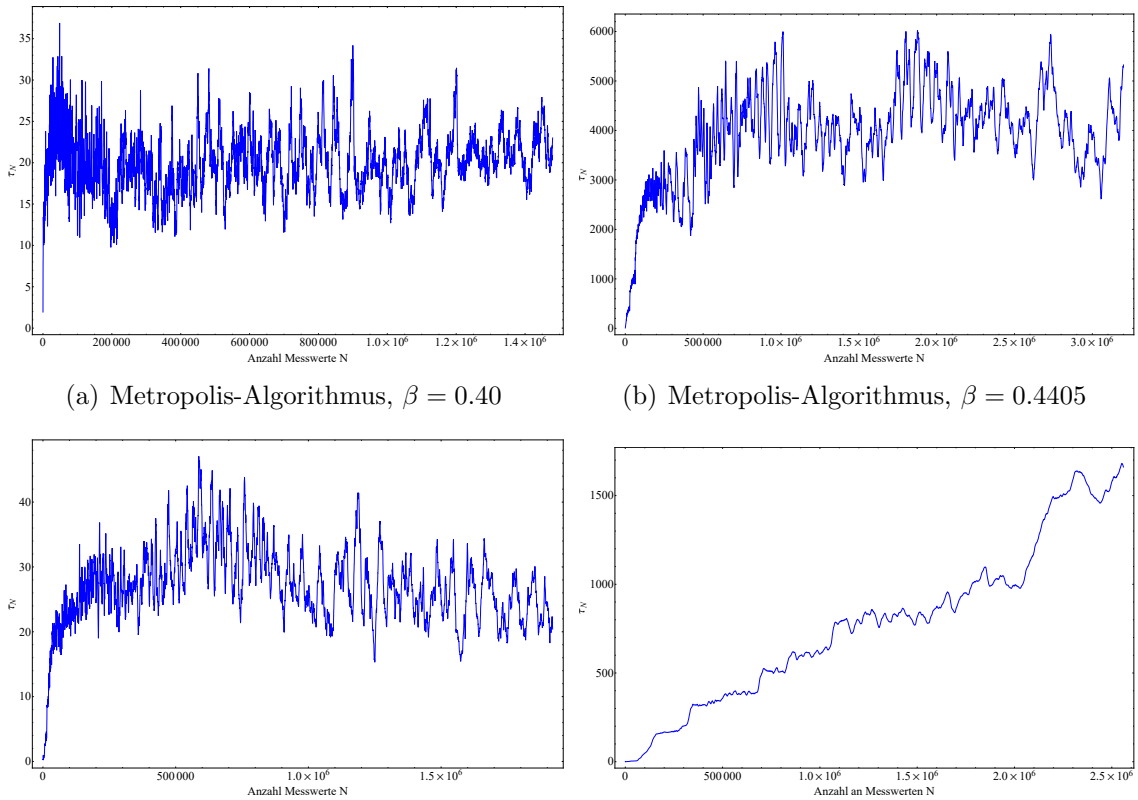
$$N_D \gg \tau_{int.} \tag{6.2.8}$$

$$N > n\tau_{int.} \tag{6.2.9}$$

$$n \geq 16 \tag{6.2.10}$$

beachtet werden, wobei N_D die Anzahl an zu Beginn verworfenen Anfangskonfigurationen der Markov-Kette ist.

Diese Methode ist nicht ohne Probleme und schlussendlich bleibt es eine Abschätzung des Fehlers, der entweder als ausreichend oder als nicht ausreichend angesehen wird. An dieser Stelle gehen wir auf ein paar Probleme der Bestimmung des Fehlers auf Basis der Binning-Methode im Rahmen der hier getroffenen Implementierungen ein. In Abbildung 6.2.5 sind die Werte τ_N für die Observable $|m(\sigma)|$ in Abhängigkeit der Anzahl der Messungen N für die Simulation des zweidimensionalen Gitters mit $L = 192$ bei unterschiedlichen dimensionslosen inversen Temperaturen β dargestellt. Wir können erkennen, dass die Werte τ_N starke Schwankungen aufweisen, jedoch für die Resultate des Metropolis-Algorithmus und für den Demon-Algorithmus bei $\beta = 0.40$ anscheinend die Tendenz zeigen um einen mittleren Wert zu schwanken, gegen den sie unter Umständen für größere N konvergieren. Dies bietet die Möglichkeit eine grobe Abschätzung für $\tau_{int.}$ auszuführen, um im Nachhinein aus den Messwerten auf die Erfüllung der Regeln in Gleichung (6.2.8) bis Gleichung (6.2.10) zu schließen, was durchaus ein plausible Begründung



(a) Metropolis-Algorithmus, $\beta = 0.40$

(b) Metropolis-Algorithmus, $\beta = 0.4405$

(c) Demon-Algorithmus, $\beta = 0.40$

(d) Demon-Algorithmus, $\beta = 0.4405$

Abbildung 6.2.5: τ_N für die Magnetisierung $|m|$ in Abhängigkeit der Anzahl der Messwerte N. Hierbei beträgt die Gitterausdehnung jeweils $L = 192$ und die Anzahl an *bins* beträgt $n = 32$.

sein kann für den bestimmten Fehler. Wir erkennen jedoch beispielsweise beim Demon-Algorithmus, dass beim Punkt $\beta = 0.4405$ in der Nähe des Phasenübergangs der Wert für τ_N keine derartige Schwankung zeigt, die ein Abschätzen von $\tau_{int.}$ ermögliche. In so einem Fall gibt es die Möglichkeit die Anzahl n der *bins* zu verringern, sodass N_B vergrößert wird. Jedoch muss beachtet werden, dass n nicht beliebig verringert werden kann, weil sonst der Wert $S_{(A'_k)}(\bar{A}_N)$ keine ausreichende Näherung mehr für $\sigma(A)$ darstellt. Bei den ausgeführten Simulationen in der Nähe des Phasenüberganges hat sich für die Implementierung des Demon-Algorithmus häufig eine nicht eindeutige Schwankung um einen Wert für $\tau_{int.}$ gezeigt. Dies erschwert die Fehlerbestimmung.

Insbesondere handelt es sich bei den beiden Algorithmen um Verfahren, die lokale Updates ausführen. Dies führt dazu, dass am Phasenübergang die integrierte Autokorrelationszeit $\tau_{int.}$ besonders große Werte annimmt, da die Korrelationslänge ξ für endliche Gitter in der Nähe des kritischen Punkts ein Maximum besitzt und somit die einzelnen Spins stärker miteinander korreliert sind. Dies wiederum führt zu einer stärkeren Korrelation der einzelnen aufeinander folgenden Konfigurationen $\sigma_n, \sigma_{n+1}, \ldots$. Hierdurch wird es notwendig am Phasenübergang mehr Konfigurationen zu erzeugen, damit die Aussa-

$tL^{\frac{1}{\nu}}$	γ_F (Metropolis-Algorithmus)	γ_F (Demon-Algorithmus)
0.1	1.73 ± 0.12	1.81 ± 0.13
0.02	1.79 ± 0.13	1.75 ± 0.15

Tabelle 6.2.1: Fit-Ergebnisse für die Messpunkte in Abbildung 6.2.7

gekraft der ermittelten Mittelwerte und Fehler erhalten bleibt. Zusätzlich nimmt die Autokorrelationszeit $\tau_{int.}$ mit der Gitterausdehnung L zu. Dieses Verhalten wird als *Critical slowing down* bezeichnet, siehe [6].

Es stellt sich somit heraus, dass eine möglichst genau Fehleranalyse mit einem entsprechenden Aufwand in der Erzeugung von vielen Messwerten liegt und genauere statistische Analysen notwendig sind. Für die Zwecke in dieser Arbeit schätzen wir den Fehler durch die Binning-Methode ein, indem $n = 32$ gewählt wird. Dies führt zumindest zu einer groben Abschätzung der ermittelten Größen. Jedoch sollten die bestimmten Standardabweichungen, insbesondere in der Nähe des Phasenübergangs, kritisch behandelt und ihnen kein allzu große Aussagekraft zugesprochen werden. Zumindest für die Verifikation der Funktionalität der Implementierungen sollten die Einschätzungen ausreichend sein.

6.2.2.2 Binderkumulante und kritischer Exponent

Für die jeweiligen Implementierungen sind, für unterschiedliche dimensionslosen Temperaturen β, Simulationen ausgeführt, wobei für die einzelnen Konfigurationen $|m|$, m^2 und m^4 berechnet und entsprechend erfasst sind. Aus diesen Werte können anschließend die gewünschten Werte für die Binderkumulante und Suszeptibilität bestimmt werden mit entsprechender Standardabweichung als Fehler der Größen, wobei die Binning-Methode mit $n = 32$ verwendet wird. Die genauen Parameter der Simulationen sind im Anhang 8.1 vorzufinden. Die Programme sind hierbei auf der GPU mit der Bezeichnung *GTX 970* ausgeführt worden.

Die Resultate für die Binderkumulante U sind, für beide Implementierungen, in Abbildung 6.2.6 dargestellt. Im Vergleich zu Abbildung 6.2.4 ist es möglich in den hier vorliegenden Resultaten den näherungsweise gemeinsamen Schnittpunkt bei U^* am kritischen Punkt zu erkennen.

Für die Bestimmung des kritischen Exponenten γ wird das Finite-Size-Scaling-Gesetz in Gleichung (2.2.27) benutzt. Aus diesem ergibt sich für zwei Gitter mit der jeweiligen Ausdehnung L_1 und L_2 und den dazugehörigen Suszeptibilitäten χ_1 und χ_2 beim selben Wert für $tL^{\frac{1}{\nu}}$ die Gesetzmäßigkeit:

$$\frac{\chi_1}{\chi_2} = \left(\frac{L_1}{L_2}\right)^{\frac{\gamma}{\nu}} \tag{6.2.11}$$

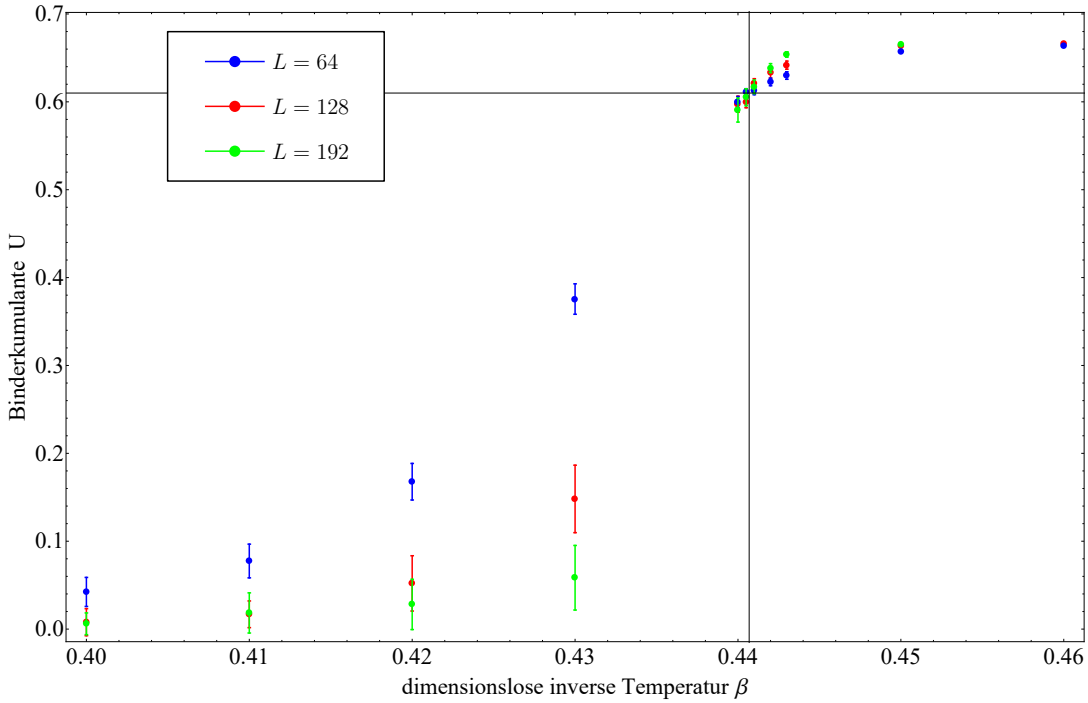
Die hier ausgeführten Simulationen, mit Hilfe der jeweiligen Implementierungen bei selben Werten $tL^{\frac{1}{\nu}}$, liefern als Resultat Abbildung 6.2.7. Für die so bestimmten Punkte wird jeweils ein Fit mit der Modellfunktion,

$$\left(\frac{\chi_1}{\chi_2}\right)_F = \left(\frac{L_1}{L_2}\right)^{\gamma_F} \tag{6.2.12}$$

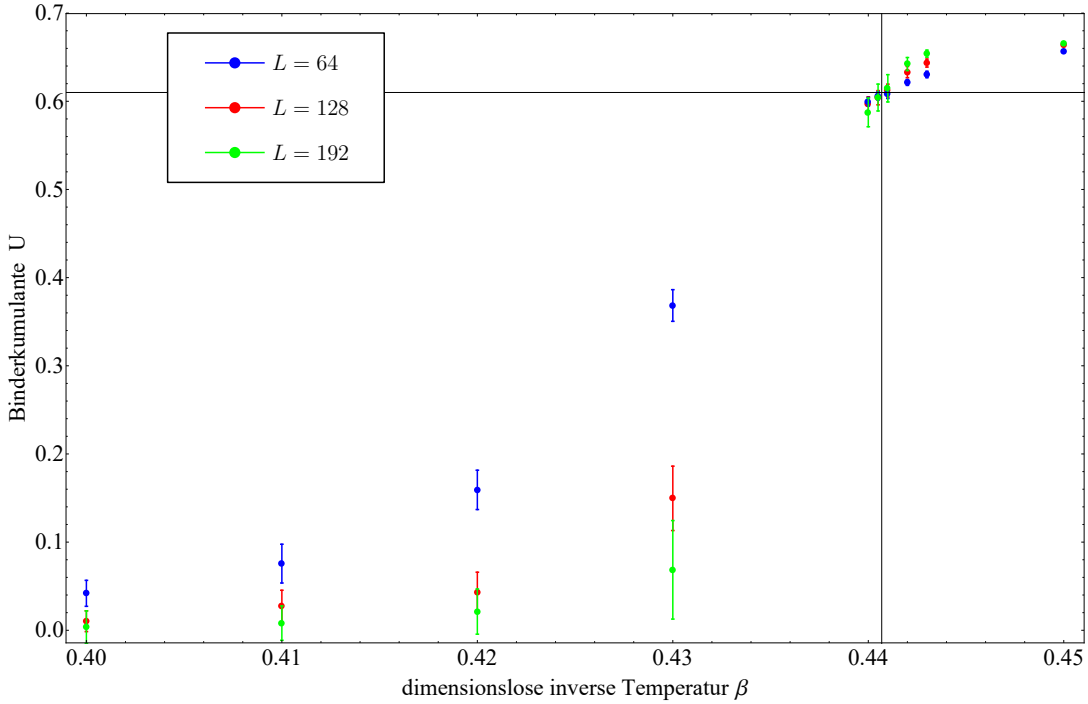
ausgeführt, wobei γ_F der Fit-Parameter ist. Das Fitten wird mit Hilfe der Funktion *NonlinearModelFit* des Programmes *Mathematica 11.0* umgesetzt. Ferner wird für den kritischen Punkt $\beta_c \approx 0.4406868$ verwendet. Das Ergebnisse der Fits sind in Tabelle 6.2.1 angegeben. Hierbei wird der Fehler über die Standardabweichungen der einzelnen Messwerte abgeschätzt.

Vergleichen wir die Werte mit den theoretischen Wert $\gamma = 1.75$, so können wir eine gute Übereinstimmung im Rahmen des Fehlers erkennen und qualitativ betrachtet liegen die Messpunkte in Abbildung 6.2.7 nahezu auf der Fit-Kurve.

Alle bis hierhin angegebenen Resultate lassen begründet den Schluss ziehen, dass die Funktionalität der Implementationen verifiziert ist.

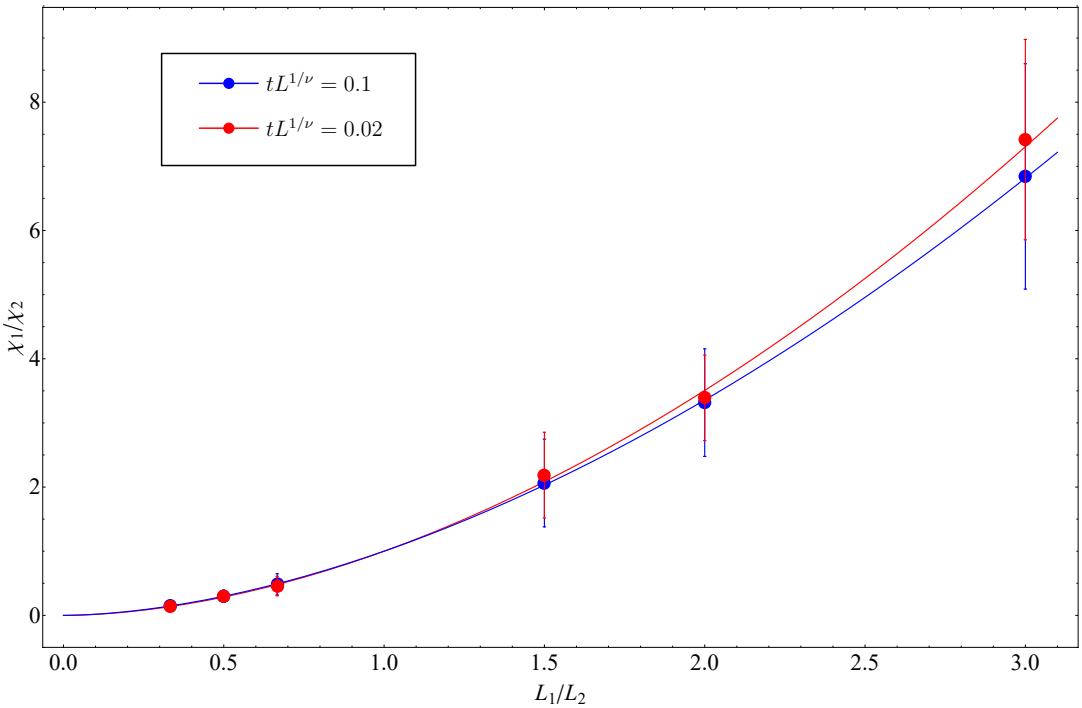


(a) Demon-Algorithmus

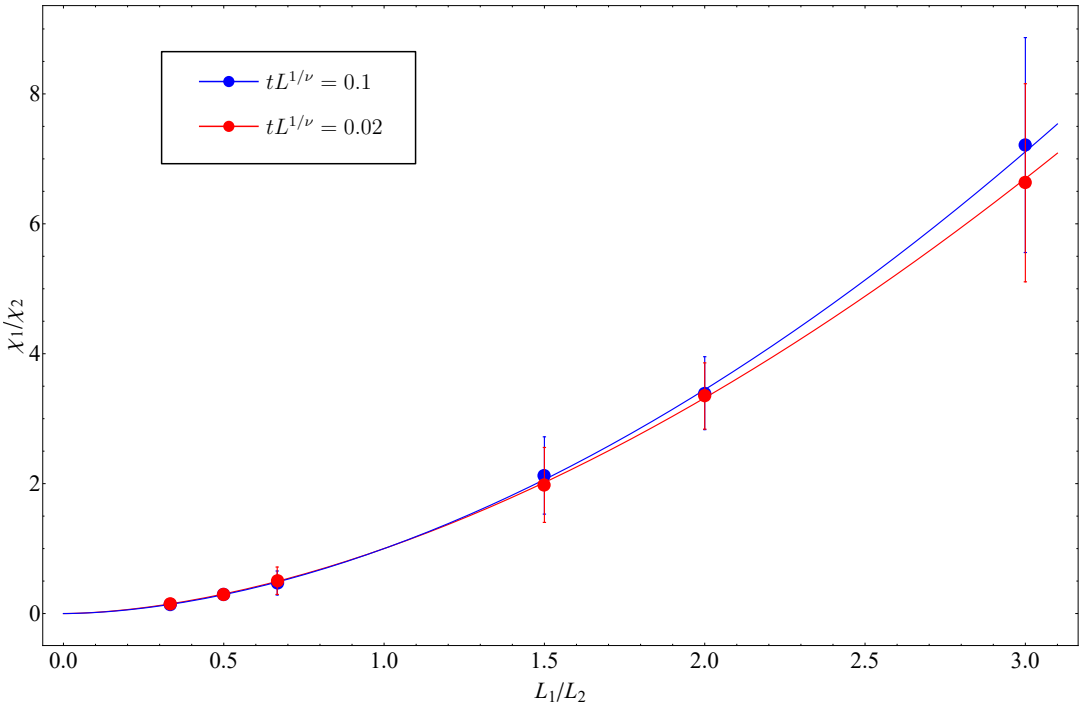


(b) Metropolis-Algorithmus

Abbildung 6.2.6: Binderkumulante U, mit Fehlerbetrachtung, in Abhängigkeit der Temperatur $\frac{kT}{J_c}$ für unterschiedliche zweidimensionale Gitterausdehnungen L, jeweils über die beiden Implementierungen bestimmt. Ferner ist in beiden Diagrammen der Kreuzungspunkt bei der kritischen Temperatur mit dem Wert $U^* \approx 0.61067$ eingezeichnet.



(a) Demon-Algorithmus



(b) Metropolis-Algorithmus

Abbildung 6.2.7: Verhältnis der Suszeptibilitäten $\frac{\chi_1}{\chi_2}$ in Abhängigkeit des Verhältnis der Gitterausdehnungen $\frac{L_1}{L_2}$ für das zweidimensionale Gitter, jeweils über die beiden Implementierungen bestimmt. Hierbei ist χ_i die Suszeptibilität des Gitters mit der Ausdehnung L_i. Die durchgezogenen Linien, sind entsprechend der Farbe die Fit-Ergebnisse für die einzelnen Messpunkte.

Kapitel 7

Zusammenfassung und Ausblick

Wir können zusammenfassen, dass die Implementierung des Demon-Algorithmus mit Multispin-Coding nach [21] für die GPU mit gewissen Problemen verbunden ist und das kanonische Update-Verfahren mit der Binomialverteilung nicht ohne weiteres auf die GPU umgesetzt werden kann und die Zuhilfenahme der CPU zu einer entsprechenden Verlangsamung führt. Es zeigt sich, dass das naive kanonische Update-Verfahren schneller ist. Jedoch ist die Implementierung des Metropolis-Algorithmus für größere Gitter schneller in der Anzahl an erzeugten Konfigurationen pro Zeiteinheit. Folglich ist die Implementierung des Metropolis-Algorithmus mit der Multispin-Coding-Methode auf Basis von [32] für Simulationen des Ising-Modells auf GPUs anscheinend besser geeignet als der Demon-Algorithmus mit der anderen Multispin-Coding-Methode.

Allerdings haben wir gesehen, dass die reine mikrokanonische Simulation mit Hilfe des Demon-Algorithmus einen hohen Durchsatz an Konfigurationen erreichen kann. Auf Basis dieser Tatsache ist es durchaus interessant mikrokanonische Betrachtungen des Ising-Modells auszuführen und der Fragestellung nachzugehen in wie weit der thermodynamische Limes durch die mikrokanonische Beschreibung eines endlichen Gitters nachgestellt werden kann oder bei welcher Gitterausdehnung L die Äquivalenz zum kanonischen Ensemble näherungsweise zutrifft und ob dies bei einem L geschieht, bei der die Anzahl an Konfiguration pro Zeiteinheit höher ist als die kanonischen Verfahren. Derartige Untersuchungen können beispielsweise für weiterführende Betrachtungen für die Implementierung des Demon-Algorithmus ausgeführt werden. Jedoch müssten hierfür unterschiedliche Faktoren genauer beachtet werden. Beispielsweise müsste genauer untersucht werden, ob die Ergodizität des Verfahrens ausreichend gewährleistet ist und wie stark die einzelnen Spinkonfigurationen zueinander korreliert sind.

Kapitel 8

Anhang

8.1 Simulationsparameter für Abschnitt 6.2.2.2

Hier sind, die für die Simulationen in Abschnitt 6.2.2.2 verwendeten Simulationsparameter, angegebenen. Das Programm hat hierbei die Observablen $|m|$, m^2 und m^4 erfasst.

β	L	N (Metro)	N_D (Metro)	N (Demon)	N_D (Demon)
0.40	64	1600000	10000	50000	10000
0.41	64	1600000	10000	50000	10000
0.42	64	1600000	10000	50000	10000
0.43	64	1600000	10000	50000	10000
0.44	64	2592000	32000	60000	10000
0.4405	64	2592000	320000	100000	20000
0.441	64	2592000	320000	60000	10000
0.442	64	2592000	32000	60000	10000
0.443	64	2592000	320000	60000	10000
0.45	64	2592000	32000	60000	10000
0.46	64	k.A.	k.A.	60000	10000
0.40	128	2592000	32000	70000	20000
0.41	128	2592000	320000	70000	20000
0.42	128	2592000	32000	80000	20000
0.43	128	2592000	320000	80000	20000
0.44	128	2592000	320000	80000	20000
0.4405	128	3230000	320000	120000	20000
0.441	128	2592000	320000	100000	20000
0.442	128	2592000	32000	100000	20000
0.443	128	2592000	32000	100000	20000
0.450	128	1600000	10000	80000	20000
0.46	128	k.A.	k.A.	80000	20000
0.40	192	1600000	32000	80000	20000
0.41	192	1600000	100000	80000	20000
0.42	192	1700000	10000	80000	20000
0.43	192	1700000	100000	80000	20000
0.44	192	3520000	320000	100000	20000
0.4405	192	3520000	320000	100000	20000
0.441	192	2592000	32000	100000	20000
0.442	192	2592000	32000	100000	20000
0.443	192	2592000	32000	100000	20000
0.45	192	2592000	320000	100000	20000
0.46	192	k.A.	k.A.	80000	20000
0.44399993011 ($x = 0.1$)	64	672000	320000	100000	20000
0.4403427822 ($x = 0.1$)	128	672000	320000	100000	20000
0.4404573951 ($x = 0.1$)	192	672000	320000	100000	20000
0.4405491284 ($x = 0.02$)	64	672000	320000	100000	20000
0.4406179534 ($x = 0.02$)	128	672000	320000	100000	20000
0.4406408999 ($x = 0.02$)	192	672000	320000	100000	20000

Tabelle 8.1.1: Simulationsparameter für die quantitativen Betrachtungen des Metropolis-Algorithmus und Demon-Algorithmus in Abschnitt 6.2.2.2. Hierbei ist N die Gesamtanzahl an berechneten Konfigurationen, L die Gitterausdehnung und N_D die Anzahl an anfänglich verworfenen Konfigurationen.

8.2 Dokumentation der Quellcodes

In diesem Abschnitt folgen die Quellcodes der in dieser Arbeit verwendeten Programme.

8.2.1 Metropolis-Algorithmus

Die Implementierung des Algorithmus besteht aus vier Header- und drei Source-Dateien.

8.2.1.1 routines2D.hpp

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#ifndef ROUTINES2D_HPP
#define ROUTINES2D_HPP


/*
  metropolis update step for even coordinates.
  invoke this kernel with:

  dim3 blocks(gridDimX, gridDimY);
  metropolis2DPeriodicEven<<blocks, 64>>>(spins_d, randoms_d, local_probabilities_d);

*/
__global__ void metropolis2DPeriodicEven(unsigned short* spins_d,int* randoms_d, double*
    local_probabilities_d);

/*

metropolis update step for odd coordinates.
invoke this kernel with:

dim3 blocks(gridDimX, gridDimY);
metropolis2DPeriodicOdd<<blocks, 64>>>(spins_d, randoms_d, local_probabilities_d);

*/
__global__ void metropolis2DPeriodicOdd(unsigned short* spins_d, int* randoms_d, double*
    local_probabilities_d);

/*
  sum up magnetization block wide

  result for each block is saved in magnetization_block_wide_d

  invoke kernel with
  dim3 blocks(gridDimX, gridDimY);

  totalMagnetizationBlockWide2D<<<blocks, 64>>>(int* spins_d, int*
    magnetization_block_wide_d);
*/
__global__ void totalMagnetizationBlockWide2D(unsigned short* spins_d, int*
    magnetization_block_wide_d);

#endif
```

8.2.1.2 routines3D.hpp

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#ifndef ROUTINES3D_HPP
#define ROUTINES3D_HPP


/*
  metropolis update step for even coordinates.

  invoke this kernel with:

  dim3 blocks(gridDimX, gridDimY, gridDimZ);
  metropolis3DPeriodicEven<<blocks, 64>>>(spins_d, randoms_d, local_probabilities_d);

*/
__global__ void metropolis3DPeriodicEven(unsigned short* spins_d,  int* randoms_d, double
    * local_probabilities_d);



/*
  metropolis update step for odd coordinates.

  invoke this kernel with:

  dim3 blocks(gridDimX, gridDimY, gridDimZ);
  metropolis3DPeriodicOdd<<blocks, 64>>>(spins_d, randoms_d, local_probabilities_d);

*/
__global__ void metropolis3DPeriodicOdd(unsigned short* spins_d,  int* randoms_d, double*
     local_probabilities_d);


/*
  sum up magnetization block wide

  result for each block is saved in magnetization_block_wide_d

  invoke kernel with
  dim3 blocks(gridDimX, gridDimY, gridDimZ);

  totalMagnetizationBlockWide3D<<<blocks, 64>>>(int* spins_d, int*
    magnetization_block_wide_d);
*/
__global__ void totalMagnetizationBlockWide3D(unsigned short* spins_d, int*
    magnetization_block_wide_d);
#endif
```

8.2.1.3 rng.hpp

```
#ifndef RNG_HPP
#define RNG_HPP
#define PI 3.14159265358979323846264338327 95
#define PI2 6.28318530717958647692528676655 90
#define SQRTPI2 2.50662827463100050241576528481 10

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <math.h>

```

```
11
12 __host__ __device__ inline void simpleRandomInteger0(int& current_random){
13 current_random = 16807 * (current_random % 127773) - 2836 * (current_random / 127773);
14 if (current_random < 0){
15   current_random += 2147483647;
16 }
17 }
18
19 __host__ __device__ inline void simpleRandomInteger1(int& current_random){
20 current_random = 48721 * (current_random % 44488) - 3399 * (current_random / 44488);
21 if (current_random < 0){
22   current_random += 2147483647;
23 }
24 }
25
26 /*
27 MARSAGLIA-ZAMAN generator (substract-with-borrow generator).
28 generates random integers between 0 and 2^32-5 OR random doubles between 0 and 1.
29
30 r=43; s=22; b = 2^32 - 5
31 */
32 class MarsagliaZamanGenerator{
33 private:
34
35 unsigned int values[43];
36 int lower_index = 0; //n-43
37 int higher_index = 21; //n-22
38 int c;
39 public:
40
41
42 MarsagliaZamanGenerator(unsigned int* start_values, int c){
43
44 for (int i = 0; i < 43; i++){
45   this->values[i] = start_values[i];
46 }
47
48 this->c = c;
49
50 }
51
52 /*
53 default init.
54 */
55 MarsagliaZamanGenerator(){
56 int current_random = 3141592;
57 for (int i = 0; i < 43; i++){
58   simpleRandomInteger0(current_random);
59   this->values[i] = current_random;
60 }
61
62 this->c = 0;
63 }
64 unsigned int getRandomInteger(){
65
66 if (values[higher_index] >= values[lower_index] + c){
67   values[lower_index] = (values[higher_index] - values[lower_index]) - c;
68   c = 0;
69 }
70 else{
71   values[lower_index] = 4294967291 - ((values[lower_index] + c) - values[higher_index]);
72   c = 1;
73 }
74 unsigned int rvalue = values[lower_index];
75
76
```

```
lower_index = (lower_index + 1) % 43;
higher_index = (higher_index + 1) % 43;

return rvalue;
}

/*
compute random double between 0...1
*/
double getRandomDouble(){

if (values[higher_index] >= values[lower_index] + c){
  values[lower_index] = (values[higher_index] - values[lower_index]) - c;
  c = 0;
}
else{
  values[lower_index] = 4294967291 - ((values[lower_index] + c) - values[higher_index]);
  c = 1;
}
int rvalue = values[lower_index];


lower_index = (lower_index + 1) % 43;
higher_index = (higher_index + 1) % 43;

//this is necessary to suppress the sign for the result of the return value.
rvalue = rvalue & ~(1 << 31);

return (double)(rvalue / ((double)~(1 << 31)));

}

/*
generate random standard normal variates using the box-muller-transformation.
(see  G. E. P. Box and Mervin E. Muller, Ann. Math. Statist. Volume 29, Number 2 (1958),
    610-611.)
*/
double getRandomStandardNormal(){

double value1 = getRandomDouble();
double value2 = getRandomDouble();

return sqrt(-2 * log(value1))*cos(PI2*value2);

}
};
#endif
```

8.2.1.4 Lattice.hpp

```
#define _ISING_2D_ 2
#define _ISING_3D_ 3
#define _PERIODIC_ 1
#define _ANTIPERIODIC_ -1


#define _OWN_ 0
#define _FRONT_ 1 //positive x-driection
#define _BACK_  2 //negative x-direction
#define _LEFT_  3 //positive y-direction
#define _RIGHT_ 4 //negative y-direction
#define _UPPER_ 5 //positive z-direction
```

```
13 #define _LOWER_ 6 //negative z-direction
14
15
16 #include "cuda_runtime.h"
17 #include "device_launch_parameters.h"
18 #include <math.h>
19 #include <iostream>
20
21 using namespace std;
22
23 #ifndef LATTICE_HPP
24 #define LATTICE_HPP
25
26 #include "rng.hpp"
27
28 class Lattice{
29
30 private:
31 int dimension;
32 int boundary;
33
34 int L1;
35 int L2;
36 int L3;
37
38 int gridDimX;
39 int gridDimY;
40 int gridDimZ;
41
42 double beta; //inverse temperature
43
44 MarsagliaZamanGenerator* random_generator;
45
46 unsigned short* spins_d;  //multispin array, on device memory
47 int* randoms_d; //random number for kernels. (on device)
48 int* randoms_h; //on host
49
50 int* magnetization_block_wide_d; //on device
51 int* magnetization_block_wide; // on host
52
53 double* local_probabilities_d; //save probabilities for energy change.
54 public:
55 //2D constructor
56 Lattice(int L1, int L2, int boundary, double beta){
57 this->L1 = L1;
58 this->L2 = L2;
59
60 this->dimension = _ISING_2D_;
61
62 this->boundary = boundary;
63
64 this->beta = beta;
65
66 this->gridDimX = L1 / 32;
67 this->gridDimY = L2 / 32;
68
69 cudaMalloc(&spins_d, 64 * gridDimX*gridDimY*sizeof(short));
70 cudaMemset(spins_d, 0, 64 * gridDimX*gridDimY*sizeof(short));
71
72 cudaMalloc(&randoms_d, 64 * gridDimX*gridDimY*sizeof(int));
73 cudaMallocHost(&randoms_h, 64 * gridDimX*gridDimY*sizeof(int));
74
75 cudaMalloc(&magnetization_block_wide_d, gridDimX*gridDimY*sizeof(int));
76 cudaMallocHost(&magnetization_block_wide, gridDimX*gridDimY*sizeof(int));
77
78 double local_probabilities[3];
```

```
local_probabilities[0] = exp(-2 * beta);
local_probabilities[1] = exp(-4 * beta);
local_probabilities[2] = exp(-8 * beta);

cudaMalloc(&local_probabilities_d, 3 * sizeof(double));
cudaMemcpy(local_probabilities_d, local_probabilities, 3 * sizeof(double),
    cudaMemcpyHostToDevice);

random_generator = new MarsagliaZamanGenerator();

initializeRandoms();

}
Lattice(int L1, int L2, int L3, int boundary, double beta){
this->L1 = L1;
this->L1 = L2;
this->L3 = L3;

this->dimension = _ISING_3D_;

this->boundary = boundary;

this->beta = beta;

this->gridDimX = L1 / 32;
this->gridDimY = L2 / 32;
this->gridDimZ = L3;

cudaMalloc(&spins_d, 64 * gridDimX*gridDimY*gridDimZ*sizeof(short));
cudaMemset(spins_d, 0, 64 * gridDimX*gridDimY*gridDimZ*sizeof(short));

cudaMalloc(&randoms_d, 64 * gridDimX*gridDimY*gridDimZ*sizeof( int));
cudaMallocHost(&randoms_h, 64 * gridDimX*gridDimY*gridDimZ*sizeof(int));

cudaMalloc(&magnetization_block_wide_d, gridDimX*gridDimY*gridDimZ*sizeof(int));
cudaMallocHost(&magnetization_block_wide, gridDimX*gridDimY*gridDimZ*sizeof(int));

double local_probabilities[4];
local_probabilities[0] = exp(-2 * beta);
local_probabilities[1] = exp(-4 * beta);
local_probabilities[2] = exp(-8 * beta);
local_probabilities[3] = exp(-12 * beta);

cudaMalloc(&local_probabilities_d, 4 * sizeof(double));
cudaMemcpy(local_probabilities_d, local_probabilities, 4 * sizeof(double),
    cudaMemcpyHostToDevice);

random_generator = new MarsagliaZamanGenerator();

initializeRandoms();


}


~Lattice(){

cudaFree(spins_d);

cudaFree(randoms_d);
cudaFreeHost(randoms_h);

cudaFree(magnetization_block_wide_d);
cudaFreeHost(magnetization_block_wide);

cudaFree(local_probabilities_d);
```

```


delete random_generator;
}

/*
metropolis update of the spin system
*/
void updateSystem();


void initializeRandoms();

int observeTotalMagnetization();


};

#endif
}
```

8.2.1.5 routines2D.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "routines2D.hpp"
#include "Lattice.hpp"
#include "rng.hpp"



struct coordinates2D_t{
  int x;
  int y;
};

/*
computes block origin for own or neighbour blocks.

local_block is element of {_OWN_, _LEFT_, _RIGHT_, _FRONT_, _BACK_, _UPPER_, _LOWER_}.
the value of local_block specifies the block relative to  own block.
*/
__device__ inline coordinates2D_t getBlockOrigin(int local_block){

coordinates2D_t local_block_position;
local_block_position.x = blockIdx.x;
local_block_position.y = blockIdx.y;

int x = blockIdx.x;
int y = blockIdx.y;

switch (local_block){

case _FRONT_:
  //regard boundary conditions
  if (x + 1 > gridDim.x - 1){
    local_block_position.x = 0;
  }
  else{
    local_block_position.x = x + 1;
  }
```

```
break;

case _BACK_:
  //regard boundary conditions
  if (x - 1 < 0){
    local_block_position.x = gridDim.x - 1;
  }
  else{
    local_block_position.x = x - 1;
  }
break;

case _LEFT_:
  //regard boundary conditions
  if (y + 1 > gridDim.y - 1){
    local_block_position.y = 0;
  }
  else{
    local_block_position.y = y + 1;
  }
break;

case _RIGHT_:
  //regard boundary conditions
  if (y - 1 < 0){
    local_block_position.y = gridDim.y - 1;
  }
  else{
    local_block_position.y = y - 1;
  }
break;
}

return local_block_position;
}




__global__ void metropolis2DPeriodicEven(unsigned short* spins_d,  int* randoms_d, double
    * local_probabilities_d){



//each short represents an 4x4 sublattice.
__shared__ unsigned short spin_chunk[7][64];
__shared__ double local_probabilities[4];
int current_random;

coordinates2D_t block_origin;
//load spins in shared memory
for (int i = 0; i < 5; i++){

  block_origin = getBlockOrigin(i);
  spin_chunk[i][threadIdx.x] = *(spins_d + 64 * (block_origin.x + block_origin.y*gridDim.
    x) + threadIdx.x);
}
//load random seeds in shared memory
current_random = *(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x);

//load spin flip probabilities to local memory
if (threadIdx.x < 3){
  local_probabilities[threadIdx.x] = *(local_probabilities_d + threadIdx.x);
}

```

```
104 __syncthreads();
105 //load 4x4 sub lattices to local memory
106 coordinates2D_t own_sub_lattice;
107 own_sub_lattice.y = threadIdx.x / 8;
108 own_sub_lattice.x = threadIdx.x % 8;
109
110 unsigned short local_sub_lattices[7];
111
112 //load own sub lattice
113 local_sub_lattices[_OWN_] = spin_chunk[_OWN_][own_sub_lattice.x + own_sub_lattice.y * 8];
114
115 //load front sub lattice
116 if (own_sub_lattice.x + 1 < 8){
117 local_sub_lattices[_FRONT_] = spin_chunk[_OWN_][own_sub_lattice.x + 1 + own_sub_lattice.y
      * 8];
118 }
119 else{
120 local_sub_lattices[_FRONT_] = spin_chunk[_FRONT_][own_sub_lattice.y * 8];
121 }
122
123 //load back sub lattice
124 if (own_sub_lattice.x > 0){
125   local_sub_lattices[_BACK_] = spin_chunk[_OWN_][own_sub_lattice.x - 1 + own_sub_lattice.
      y * 8];
126 }
127 else{
128   local_sub_lattices[_BACK_] = spin_chunk[_BACK_][7 + own_sub_lattice.y * 8];
129 }
130
131 //load left sub lattice
132 if (own_sub_lattice.y + 1 < 8){
133   local_sub_lattices[_LEFT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y +
      1) * 8];
134 }
135 else{
136   local_sub_lattices[_LEFT_] = spin_chunk[_LEFT_][own_sub_lattice.x];
137 }
138
139 //load right sub lattice
140 if (own_sub_lattice.y > 0){
141   local_sub_lattices[_RIGHT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y
      - 1) * 8];
142 }
143 else{
144   local_sub_lattices[_RIGHT_] = spin_chunk[_RIGHT_][own_sub_lattice.x + 56];
145 }
146
147 //load upper and lower sub lattice
148
149 coordinates2D_t sub_lattice_origin; //origin of own sublattice.
150 sub_lattice_origin.x = blockIdx.x * 32 + own_sub_lattice.x * 4;
151 sub_lattice_origin.y = blockIdx.y * 32 + own_sub_lattice.y * 4;
152
153 //calculate spin flips for all even points in own sub lattice
154
155 int x_offset;
156 //is sub lattice origin even ?
157 if (((sub_lattice_origin.x + sub_lattice_origin.y) % 2) == 0){
158   //begin with first element in sub lattice
159   x_offset = 0;
160 }
161 else{
162   //begin with second element in sub lattice
163   x_offset = 1;
164 }
165
```

```

unsigned short own_spin;
unsigned short neighbour_spins[4];
unsigned short parallel;
short current_energy;
short energy_change;
double probability;
unsigned short flip_decision;
int x;
for (int y = 0; y < 4; y++){
  for (int i = 0; i < 2; i++){
    x = 2 * i + x_offset;
    own_spin = (local_sub_lattices[_OWN_] & (1 << (x + y * 4))) >> (x + y * 4);

    //load front spin
    if (x + 1 < 4){
      neighbour_spins[0] = (local_sub_lattices[_OWN_] & (1 << (x + 1 + y * 4))) >> (x + 1
     + y * 4);
    }
    else{
      neighbour_spins[0] = (local_sub_lattices[_FRONT_] & (1 << (y * 4))) >> (y * 4);
    }

    //load back spin
    if (x > 0){
      neighbour_spins[1] = (local_sub_lattices[_OWN_] & (1 << (x - 1 + y * 4))) >> (x - 1
     + y * 4);
    }
    else{
      neighbour_spins[1] = (local_sub_lattices[_BACK_] & (1 << (3 + y * 4))) >> (3 + y *
    4);
    }

    //load left spin
    if (y + 1 < 4){
      neighbour_spins[2] = (local_sub_lattices[_OWN_] & (1 << (x + (y + 1) * 4))) >> (x +
     (y + 1) * 4);
    }
    else{
      neighbour_spins[2] = (local_sub_lattices[_LEFT_] & (1 << x)) >> x;
    }

    //load right spin
    if (y > 0){
      neighbour_spins[3] = (local_sub_lattices[_OWN_] & (1 << (x + (y - 1) * 4))) >> (x +
     (y - 1) * 4);
    }
    else{
      neighbour_spins[3] = (local_sub_lattices[_RIGHT_] & (1 << (x + 12))) >> (x + 12);
    }



    //calculate energy change
    energy_change = 0;
    for (int i = 0; i < 4; i++){
    //bit 0: spins antiparallel; bit 1: spins parallel
    parallel = ~(neighbour_spins[i] ^ own_spin);

    //arithmetical shift!
    parallel = (parallel << 31) >> 31;

    current_energy = 1;
    current_energy = current_energy ^ (parallel << 1);

    energy_change = energy_change + current_energy;
```

```
227     }
228
229     energy_change = -2 * energy_change;
230
231     //flip own spin?
232     if (energy_change <= 0){
233       probability = 1;
234     }
235     else{
236       probability = local_probabilities[energy_change / 4];
237     }
238
239     simpleRandomInteger0(current_random);
240
241
242     if ((current_random / 2147483646.0) <= probability){
243       flip_decision = 1;
244     }
245     else{
246       flip_decision = 0;
247     }
248     local_sub_lattices[_OWN_] = local_sub_lattices[_OWN_] ^ (flip_decision << (x + y * 4)
        );
249
250   }
251   x_offset = x_offset ^ 1;
252 }
253
254 //write results back to global memory
255 *(spins_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x) = local_sub_lattices[
        _OWN_];
256
257 //write last random number back to global memory
258 *(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x) = current_random;
259
260 }
261 __global__ void metropolis2DPeriodicOdd(unsigned short* spins_d,  int* randoms_d, double*
         local_probabilities_d){
262
263 //each short represents an 4x4 sublattice.
264 __shared__ unsigned short spin_chunk[7][64];
265 __shared__ double local_probabilities[4];
266 int current_random;
267
268 coordinates2D_t block_origin;
269 //load spins in shared memory
270 for (int i = 0; i < 5; i++){
271
272   block_origin = getBlockOrigin(i);
273     spin_chunk[i][threadIdx.x] = *(spins_d + 64 * (block_origin.x + block_origin.y*
        gridDim.x) + threadIdx.x);
274 }
275 //load random seeds in shared memory
276 current_random = *(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x);
277
278 if (threadIdx.x < 3){
279   local_probabilities[threadIdx.x] = *(local_probabilities_d + threadIdx.x);
280 }
281
282 __syncthreads();
283 //load 4x4 sub lattices to local memory
284 coordinates2D_t own_sub_lattice;
285 own_sub_lattice.y = threadIdx.x / 8;
286 own_sub_lattice.x = threadIdx.x % 8;
287
288 unsigned short local_sub_lattices[7];
```

```
289
290 //load own sub lattice
291 local_sub_lattices[_OWN_] = spin_chunk[_OWN_][own_sub_lattice.x + own_sub_lattice.y * 8];
292
293 //load front sub lattice
294 if (own_sub_lattice.x + 1 < 8){
295   local_sub_lattices[_FRONT_] = spin_chunk[_OWN_][own_sub_lattice.x + 1 + own_sub_lattice
      .y * 8];
296 }
297 else{
298   local_sub_lattices[_FRONT_] = spin_chunk[_FRONT_][own_sub_lattice.y * 8];
299 }
300
301 //load back sub lattice
302 if (own_sub_lattice.x > 0){
303   local_sub_lattices[_BACK_] = spin_chunk[_OWN_][own_sub_lattice.x - 1 + own_sub_lattice.
      y * 8];
304 }
305 else{
306   local_sub_lattices[_BACK_] = spin_chunk[_BACK_][7 + own_sub_lattice.y * 8];
307 }
308
309 //load left sub lattice
310 if (own_sub_lattice.y + 1 < 8){
311   local_sub_lattices[_LEFT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y +
      1) * 8];
312 }
313 else{
314   local_sub_lattices[_LEFT_] = spin_chunk[_LEFT_][own_sub_lattice.x];
315 }
316
317 //load right sub lattice
318 if (own_sub_lattice.y > 0){
319   local_sub_lattices[_RIGHT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y
      - 1) * 8];
320 }
321 else{
322   local_sub_lattices[_RIGHT_] = spin_chunk[_RIGHT_][own_sub_lattice.x + 56];
323 }
324
325 //load upper and lower sub lattice
326
327 coordinates2D_t sub_lattice_origin; //origin of own sublattice.
328 sub_lattice_origin.x = blockIdx.x * 32 + own_sub_lattice.x * 4;
329 sub_lattice_origin.y = blockIdx.y * 32 + own_sub_lattice.y * 4;
330
331 //calculate spin flips for all even points in own sub lattice
332
333 int x_offset;
334 //is sub lattice origin even ?
335 if (((sub_lattice_origin.x + sub_lattice_origin.y) % 2) == 0){
336   //begin with first element in sub lattice
337   x_offset = 1;
338 }
339 else{
340   //begin with second element in sub lattice
341   x_offset = 0;
342 }
343
344
345 unsigned short own_spin;
346 unsigned short neighbour_spins[4];
347 unsigned short parallel;
348 short current_energy;
349 short energy_change;
350 double probability;
```

```
351 unsigned short flip_decision;
352 int x;
353 for (int y = 0; y < 4; y++){
354   for (int i = 0; i < 2; i++){
355     x = 2 * i + x_offset;
356     own_spin = (local_sub_lattices[_OWN_] & (1 << (x + y * 4))) >> (x + y * 4);
357
358     //load front spin
359     if (x + 1 < 4){
360       neighbour_spins[0] = (local_sub_lattices[_OWN_] & (1 << (x + 1 + y * 4))) >> (x + 1
     + y * 4);
361     }
362     else{
363       neighbour_spins[0] = (local_sub_lattices[_FRONT_] & (1 << (y * 4))) >> (y * 4);
364     }
365
366     //load back spin
367     if (x > 0){
368       neighbour_spins[1] = (local_sub_lattices[_OWN_] & (1 << (x - 1 + y * 4))) >> (x - 1
     + y * 4);
369     }
370     else{
371       neighbour_spins[1] = (local_sub_lattices[_BACK_] & (1 << (3 + y * 4))) >> (3 + y *
     4);
372     }
373
374     //load left spin
375     if (y + 1 < 4){
376       neighbour_spins[2] = (local_sub_lattices[_OWN_] & (1 << (x + (y + 1) * 4))) >> (x +
     (y + 1) * 4);
377     }
378     else{
379       neighbour_spins[2] = (local_sub_lattices[_LEFT_] & (1 << x)) >> x;
380     }
381
382     //load right spin
383     if (y > 0){
384       neighbour_spins[3] = (local_sub_lattices[_OWN_] & (1 << (x + (y - 1) * 4))) >> (x +
     (y - 1) * 4);
385     }
386     else{
387       neighbour_spins[3] = (local_sub_lattices[_RIGHT_] & (1 << (x + 12))) >> (x + 12);
388     }
389
390
391
392     //calculate energy change
393     energy_change = 0;
394     for (int i = 0; i < 4; i++){
395       //bit 0: spins antiparallel; bit 1: spins parallel
396       parallel = ~(neighbour_spins[i] ^ own_spin);
397
398       //arithmetical shift!
399       parallel = (parallel << 31) >> 31;
400
401       current_energy = 1;
402       current_energy = current_energy ^ (parallel << 1);
403
404       energy_change = energy_change + current_energy;
405     }
406
407     energy_change = -2 * energy_change;
408
409     //flip own spin?
410     if (energy_change <= 0){
411       probability = 1;
```

```
412     }
413     else{
414       probability = local_probabilities[energy_change / 4];
415     }
416
417     simpleRandomInteger0(current_random);
418
419
420     if ((current_random / 2147483646.0) <= probability){
421       flip_decision = 1;
422     }
423     else{
424       flip_decision = 0;
425     }
426     local_sub_lattices[_OWN_] = local_sub_lattices[_OWN_] ^ (flip_decision << (x + y * 4)
        );
427
428   }
429   x_offset = x_offset ^ 1;
430 }
431
432 //write results back to global memory
433 *(spins_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x) = local_sub_lattices[
        _OWN_];
434
435 //write last random number back to global memory
436 *(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x) = current_random;
437 }
438
439 __global__ void totalMagnetizationBlockWide2D(unsigned short* spins_d, int*
        magnetization_block_wide_d){
440
441
442 __shared__ int magnetization[64];
443 unsigned short own_sub_lattice;
444 own_sub_lattice = *(spins_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x) + threadIdx.x);
445
446
447 //sum up magnetization of own sub lattice.
448 int current_magn = 0;
449 int sign;
450 for (int i = 0; i < 16; i++){
451   sign = ((~((own_sub_lattice & (1 << i)) >> i) << 31) >> 31);
452   current_magn = current_magn + (1 | sign);
453 }
454 magnetization[threadIdx.x] = current_magn;
455 __syncthreads();
456 //half warps sums up magnetization
457 if (threadIdx.x % 16 == 0){
458   for (int i = 1; i < 16; i++){
459   magnetization[threadIdx.x] = magnetization[threadIdx.x] + magnetization[threadIdx.x + i
        ];
460   }
461 }
462 __syncthreads();
463 if (threadIdx.x == 0){
464   for (int i = 1; i < 4; i++){
465   magnetization[0] = magnetization[0] + magnetization[i * 16];
466   }
467
468   //write result to global memory
469   *(magnetization_block_wide_d + blockIdx.x + blockIdx.y*gridDim.x) = magnetization[0];
470   }
471
472 }
```

8.2.1.6 routines3D.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "routines3D.hpp"
#include "Lattice.hpp"
#include "rng.hpp"

struct coordinates3D_t{
int x;
int y;
int z;
};

/*
computes block origin for own or neighbour blocks.

local_block is element of {_OWN_, _LEFT_, _RIGHT_, _FRONT_, _BACK_, _UPPER_, _LOWER_}.
the value of local_block specifies the block relative to  own block.
*/
__device__ inline coordinates3D_t getBlockOrigin(int local_block){

coordinates3D_t local_block_position;
local_block_position.x = blockIdx.x;
local_block_position.y = blockIdx.y;
local_block_position.z = blockIdx.z;

int x = blockIdx.x;
int y = blockIdx.y;
int z = blockIdx.z;

switch (local_block){

case _FRONT_:
  //regard boundary conditions
  if (x + 1 > gridDim.x - 1){
    local_block_position.x = 0;
  }
  else{
    local_block_position.x = x + 1;
  }
break;

case _BACK_:
  //regard boundary conditions
  if (x - 1 < 0){
    local_block_position.x = gridDim.x - 1;
  }
  else{
    local_block_position.x = x - 1;
  }
break;

case _LEFT_:
  //regard boundary conditions
  if (y + 1 > gridDim.y - 1){
    local_block_position.y = 0;
  }
  else{
```

```
59      local_block_position.y = y + 1;
60    }
61  break;
62
63  case _RIGHT_:
64    //regard boundary conditions
65    if (y - 1 < 0){
66      local_block_position.y = gridDim.y - 1;
67    }
68    else{
69      local_block_position.y = y - 1;
70    }
71  break;
72
73  case _UPPER_:
74    //regard boundary conditions
75    if (z + 1 > gridDim.z - 1){
76      local_block_position.z = 0;
77    }
78    else{
79      local_block_position.z = z + 1;
80    }
81  break;
82
83  case _LOWER_:
84    //regard boundary conditions
85    if (z - 1 < 0){
86      local_block_position.z = gridDim.z - 1;
87    }
88    else{
89      local_block_position.z = z - 1;
90    }
91  break;
92  }
93
94  return local_block_position;
95  }
96
97
98
99
100
101 __global__ void metropolis3DPeriodicEven(unsigned short* spins_d,  int* randoms_d, double
      * local_probabilities_d){
102
103
104
105 //each short represents an 4x4 sublattice.
106 __shared__ unsigned short spin_chunk[7][64];
107 __shared__ double local_probabilities[4];
108 int current_random;
109
110 coordinates3D_t block_origin;
111 //load spins in shared memory
112 for (int i = 0; i < 7; i++){
113
114   block_origin = getBlockOrigin(i);
115   spin_chunk[i][threadIdx.x] = *(spins_d + 64 * (block_origin.x + block_origin.y*gridDim.
      x + block_origin.z*gridDim.x*gridDim.y) + threadIdx.x);
116 }
117 //load random seeds in shared memory
118 current_random = *(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
      gridDim.x*gridDim.y) + threadIdx.x);
119
120 if (threadIdx.x < 4){
121   local_probabilities[threadIdx.x] = *(local_probabilities_d + threadIdx.x);
```

```
122 }
123
124 __syncthreads();
125 //load 4x4 sub lattices to local memory
126 coordinates3D_t own_sub_lattice;
127 own_sub_lattice.z = blockIdx.z;
128 own_sub_lattice.y = threadIdx.x / 8;
129 own_sub_lattice.x = threadIdx.x % 8;
130
131 unsigned short local_sub_lattices[7];
132
133 //load own sub lattice
134 local_sub_lattices[_OWN_] = spin_chunk[_OWN_][own_sub_lattice.x + own_sub_lattice.y * 8];
135
136 //load front sub lattice
137 if (own_sub_lattice.x + 1 < 8){
138   local_sub_lattices[_FRONT_] = spin_chunk[_OWN_][own_sub_lattice.x + 1 + own_sub_lattice
      .y * 8];
139 }
140 else{
141   local_sub_lattices[_FRONT_] = spin_chunk[_FRONT_][own_sub_lattice.y * 8];
142 }
143
144 //load back sub lattice
145 if (own_sub_lattice.x > 0){
146   local_sub_lattices[_BACK_] = spin_chunk[_OWN_][own_sub_lattice.x - 1 + own_sub_lattice.
      y * 8];
147 }
148 else{
149   local_sub_lattices[_BACK_] = spin_chunk[_BACK_][7 + own_sub_lattice.y * 8];
150 }
151
152 //load left sub lattice
153 if (own_sub_lattice.y + 1 < 8){
154   local_sub_lattices[_LEFT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y +
      1) * 8];
155 }
156 else{
157   local_sub_lattices[_LEFT_] = spin_chunk[_LEFT_][own_sub_lattice.x];
158 }
159
160 //load right sub lattice
161 if (own_sub_lattice.y > 0){
162   local_sub_lattices[_RIGHT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y
      - 1) * 8];
163 }
164 else{
165   local_sub_lattices[_RIGHT_] = spin_chunk[_RIGHT_][own_sub_lattice.x + 56];
166 }
167
168 //load upper and lower sub lattice
169 local_sub_lattices[_UPPER_] = spin_chunk[_UPPER_][own_sub_lattice.x + own_sub_lattice.y *
      8];
170 local_sub_lattices[_LOWER_] = spin_chunk[_LOWER_][own_sub_lattice.x + own_sub_lattice.y *
      8];
171
172 coordinates3D_t sub_lattice_origin; //origin of own sublattice.
173 sub_lattice_origin.x = blockIdx.x* 32 + own_sub_lattice.x * 4;
174 sub_lattice_origin.y = blockIdx.y* 32 + own_sub_lattice.y * 4;
175 sub_lattice_origin.z = own_sub_lattice.z;
176
177 //calculate spin flips for all even points in own sub lattice
178
179 int x_offset;
180 //is sub lattice origin even ?
181 if (((sub_lattice_origin.x + sub_lattice_origin.y + sub_lattice_origin.z) % 2) == 0){
```

```
182   //begin with first element in sub lattice
183   x_offset = 0;
184 }
185 else{
186   //begin with second element in sub lattice
187   x_offset = 1;
188 }
189
190
191 unsigned short own_spin;
192 unsigned short neighbour_spins[6];
193 unsigned short parallel;
194 short current_energy;
195 short energy_change;
196 double probability;
197 unsigned short flip_decision;
198 int x;
199 for (int y = 0; y < 4; y++){
200   for (int i = 0; i < 2; i++){
201     x = 2*i + x_offset;
202     own_spin = (local_sub_lattices[_OWN_] & (1 << (x + y * 4))) >> (x + y * 4);
203
204     //load front spin
205     if (x + 1 < 4){
206       neighbour_spins[0] = (local_sub_lattices[_OWN_] & (1 << (x + 1 + y * 4))) >> (x + 1
      + y * 4);
207     }
208     else{
209       neighbour_spins[0] = (local_sub_lattices[_FRONT_] & (1 << (y * 4))) >> (y * 4);
210     }
211
212     //load back spin
213     if (x > 0){
214       neighbour_spins[1] = (local_sub_lattices[_OWN_] & (1 << (x - 1 + y * 4))) >> (x - 1
      + y * 4);
215     }
216     else{
217       neighbour_spins[1] = (local_sub_lattices[_BACK_] & (1 << (3 + y * 4))) >> (3 + y *
     4);
218     }
219
220     //load left spin
221     if (y + 1 < 4){
222       neighbour_spins[2] = (local_sub_lattices[_OWN_] & (1 << (x + (y + 1) * 4))) >> (x +
      (y + 1) * 4);
223     }
224     else{
225       neighbour_spins[2] = (local_sub_lattices[_LEFT_] & (1 << x)) >> x;
226     }
227
228     //load right spin
229     if (y > 0){
230       neighbour_spins[3] = (local_sub_lattices[_OWN_] & (1 << (x + (y - 1) * 4))) >> (x +
      (y - 1) * 4);
231     }
232     else{
233       neighbour_spins[3] = (local_sub_lattices[_RIGHT_] & (1 << (x + 12))) >> (x + 12);
234     }
235
236     //load upper spin
237     neighbour_spins[4] = (local_sub_lattices[_UPPER_] & (1 << (x + y * 4))) >> (x + y *
     4);
238     neighbour_spins[5] = (local_sub_lattices[_LOWER_] & (1 << (x + y * 4))) >> (x + y *
     4);
239
240     //calculate energy change
```

```
241     energy_change = 0;
242     for (int i = 0; i < 6; i++){
243       //bit 0: spins antiparallel; bit 1: spins parallel
244       parallel = ~(neighbour_spins[i] ^ own_spin);
245
246       //arithmetical shift!
247       parallel = (parallel << 31) >> 31;
248
249       current_energy = 1;
250       current_energy = current_energy ^ (parallel << 1);
251
252       energy_change = energy_change + current_energy;
253     }
254
255     energy_change = -2 * energy_change;
256
257     //flip own spin?
258     if (energy_change <= 0){
259       probability = 1;
260     }
261     else{
262       probability = local_probabilities[energy_change / 4];
263     }
264
265     simpleRandomInteger0(current_random);
266
267
268     if ((current_random / 2147483647.0) <= probability){
269       flip_decision = 1;
270     }
271     else{
272       flip_decision = 0;
273     }
274     local_sub_lattices[_OWN_] = local_sub_lattices[_OWN_]  ^ (flip_decision << (x + y *
        4));
275
276   }
277   x_offset = x_offset ^ 1;
278 }
279
280 //write results back to global memory
281 *(spins_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*gridDim.y) +
        threadIdx.x) = local_sub_lattices[_OWN_];
282
283 //write last random number back to global memory
284 *(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*gridDim.y) +
         threadIdx.x) = current_random;
285
286 }
287 __global__ void metropolis3DPeriodicOdd(unsigned short* spins_d, int* randoms_d, double*
        local_probabilities_d){
288
289 //each short represents an 4x4 sublattice.
290 __shared__ unsigned short spin_chunk[7][64];
291 __shared__ double local_probabilities[4];
292 int current_random;
293
294 coordinates3D_t block_origin;
295 //load spins in shared memory
296 for (int i = 0; i < 7; i++){
297
298   block_origin = getBlockOrigin(i);
299   spin_chunk[i][threadIdx.x] = *(spins_d + 64 * (block_origin.x + block_origin.y*gridDim.
        x + block_origin.z*gridDim.x*gridDim.y) + threadIdx.x);
300 }
301 //load random seeds in shared memory
```

```
302 current_random = *(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
      gridDim.x*gridDim.y) + threadIdx.x);
303
304 if (threadIdx.x < 4){
305   local_probabilities[threadIdx.x] = *(local_probabilities_d + threadIdx.x);
306 }
307 __syncthreads();
308
309 //load 4x4 sub lattices to local memory
310 coordinates3D_t own_sub_lattice;
311 own_sub_lattice.z = blockIdx.z;
312 own_sub_lattice.y = threadIdx.x / 8;
313 own_sub_lattice.x = threadIdx.x % 8;
314
315 unsigned short local_sub_lattices[7];
316
317 //load own sub lattice
318 local_sub_lattices[_OWN_] = spin_chunk[_OWN_][own_sub_lattice.x + own_sub_lattice.y * 8];
319
320 //load front sub lattice
321 if (own_sub_lattice.x + 1 < 8){
322   local_sub_lattices[_FRONT_] = spin_chunk[_OWN_][own_sub_lattice.x + 1 + own_sub_lattice
      .y * 8];
323 }
324 else{
325   local_sub_lattices[_FRONT_] = spin_chunk[_FRONT_][own_sub_lattice.y * 8];
326 }
327
328 //load back sub lattice
329 if (own_sub_lattice.x > 0){
330   local_sub_lattices[_BACK_] = spin_chunk[_OWN_][own_sub_lattice.x - 1 + own_sub_lattice.
      y * 8];
331 }
332 else{
333   local_sub_lattices[_BACK_] = spin_chunk[_BACK_][7 + own_sub_lattice.y * 8];
334 }
335
336 //load left sub lattice
337 if (own_sub_lattice.y + 1 < 8){
338   local_sub_lattices[_LEFT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y +
      1) * 8];
339 }
340 else{
341   local_sub_lattices[_LEFT_] = spin_chunk[_LEFT_][own_sub_lattice.x];
342 }
343
344 //load right sub lattice
345 if (own_sub_lattice.y > 0){
346   local_sub_lattices[_RIGHT_] = spin_chunk[_OWN_][own_sub_lattice.x + (own_sub_lattice.y
      - 1) * 8];
347 }
348 else{
349   local_sub_lattices[_RIGHT_] = spin_chunk[_RIGHT_][own_sub_lattice.x + 56];
350 }
351
352 //load upper and lower sub lattice
353 local_sub_lattices[_UPPER_] = spin_chunk[_UPPER_][own_sub_lattice.x + own_sub_lattice.y *
      8];
354 local_sub_lattices[_LOWER_] = spin_chunk[_LOWER_][own_sub_lattice.x + own_sub_lattice.y *
      8];
355
356 coordinates3D_t sub_lattice_origin; //origin of own sublattice.
357 sub_lattice_origin.x = blockIdx.x * 32 + own_sub_lattice.x * 4;
358 sub_lattice_origin.y = blockIdx.y * 32 + own_sub_lattice.y * 4;
359 sub_lattice_origin.z = own_sub_lattice.z;
360
```

```
361 //calculate spin flips for all even points in own sub lattice
362
363 int x_offset;
364 //is sub lattice origin even ?
365 if (((sub_lattice_origin.x + sub_lattice_origin.y + sub_lattice_origin.z) % 2) == 0){
366   //begin with first element in sub lattice
367   x_offset = 1;
368 }
369 else{
370   //begin with second element in sub lattice
371   x_offset = 0;
372 }
373
374
375 unsigned short own_spin;
376 unsigned short neighbour_spins[6];
377 unsigned short parallel;
378 short current_energy;
379 short energy_change;
380 double probability;
381 unsigned short flip_decision;
382 int x;
383 for (int y = 0; y < 4; y++){
384   for (int i = 0; i < 2; i++){
385     x = 2 * i + x_offset;
386     own_spin = (local_sub_lattices[_OWN_] & (1 << (x + y * 4))) >> (x + y * 4);
387
388     //load front spin
389     if (x + 1 < 4){
390       neighbour_spins[0] = (local_sub_lattices[_OWN_] & (1 << (x + 1 + y * 4))) >> (x + 1
     + y * 4);
391     }
392     else{
393       neighbour_spins[0] = (local_sub_lattices[_FRONT_] & (1 << (y * 4))) >> (y * 4);
394     }
395
396     //load back spin
397     if (x > 0){
398       neighbour_spins[1] = (local_sub_lattices[_OWN_] & (1 << (x - 1 + y * 4))) >> (x - 1
     + y * 4);
399     }
400     else{
401       neighbour_spins[1] = (local_sub_lattices[_BACK_] & (1 << (3 + y * 4))) >> (3 + y *
     4);
402     }
403
404     //load left spin
405     if (y + 1 < 4){
406       neighbour_spins[2] = (local_sub_lattices[_OWN_] & (1 << (x + (y + 1) * 4))) >> (x +
     (y + 1) * 4);
407     }
408     else{
409       neighbour_spins[2] = (local_sub_lattices[_LEFT_] & (1 << x)) >> x;
410     }
411
412     //load right spin
413     if (y > 0){
414       neighbour_spins[3] = (local_sub_lattices[_OWN_] & (1 << (x + (y - 1) * 4))) >> (x +
     (y - 1) * 4);
415     }
416     else{
417       neighbour_spins[3] = (local_sub_lattices[_RIGHT_] & (1 << (x + 12))) >> (x + 12);
418     }
419
420     //load upper spin
```

```
    neighbour_spins[4] = (local_sub_lattices[_UPPER_] & (1 << (x + y * 4))) >> (x + y *
    4);
    neighbour_spins[5] = (local_sub_lattices[_LOWER_] & (1 << (x + y * 4))) >> (x + y *
    4);

    //calculate energy change
    energy_change = 0;
    for (int i = 0; i < 6; i++){
      //bit 0: spins antiparallel; bit 1: spins parallel
      parallel = ~(neighbour_spins[i] ^ own_spin);

      //arithmetical shift!
      parallel = (parallel << 31) >> 31;

      current_energy = 1;
      current_energy = current_energy ^ (parallel << 1);

      energy_change = energy_change + current_energy;
    }

    energy_change = -2 * energy_change;

    //flip own spin?
    if (energy_change <= 0){
      probability = 1;
    }
    else{
      probability = local_probabilities[energy_change / 4];
    }

    simpleRandomInteger0(current_random);


    if ((current_random / 2147483647.0) <= probability){
      flip_decision = 1;
    }
    else{
      flip_decision = 0;
    }
    local_sub_lattices[_OWN_] = local_sub_lattices[_OWN_] ^ (flip_decision << (x + y * 4)
    );

  }
  x_offset = x_offset ^ 1;
}

//write results back to global memory
*(spins_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*gridDim.y) +
    threadIdx.x) = local_sub_lattices[_OWN_];

//write last random number back to global memory
*(randoms_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*gridDim.y) +
     threadIdx.x) = current_random;
}

__global__ void totalMagnetizationBlockWide3D(unsigned short* spins_d, int*
    magnetization_block_wide_d){


__shared__ int magnetization[64];
unsigned short own_sub_lattice;
own_sub_lattice = *(spins_d + 64 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
    gridDim.x*gridDim.y) + threadIdx.x);


//sum up magnetization of own sub lattice.
```

```
int current_magn = 0;
int sign;
for (int i = 0; i < 16; i++){
  sign = ((~((own_sub_lattice & (1 << i)) >> i) << 31) >> 31);
  current_magn = current_magn + (1 | sign);
}
magnetization[threadIdx.x] = current_magn;
__syncthreads();
//half warps sums up magnetization
if (threadIdx.x % 16 == 0){
  for (int i = 1; i < 16; i++){
    magnetization[threadIdx.x] = magnetization[threadIdx.x] + magnetization[threadIdx.x +
     i];
  }
}
__syncthreads();
if (threadIdx.x == 0){
  for (int i = 1; i < 4; i++){
    magnetization[0] = magnetization[0] + magnetization[i * 16];
  }

  //write result to global memory
  *(magnetization_block_wide_d + blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x
    *gridDim.y) = magnetization[0];
}

}
```

8.2.1.7 Lattice.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "Lattice.hpp"
#include "routines3D.hpp"
#include "routines2D.hpp"

#include <iostream>

//#include "routines2D.hpp"

using namespace std;

void Lattice::updateSystem(){

if (dimension == _ISING_2D_ && boundary == _PERIODIC_){
  dim3 blocks(gridDimX, gridDimY);

  metropolis2DPeriodicEven<<<blocks, 64 >>>(spins_d, randoms_d, local_probabilities_d);
  metropolis2DPeriodicOdd<<<blocks, 64 >>>(spins_d, randoms_d, local_probabilities_d);
  cudaDeviceSynchronize();
}
else if (dimension == _ISING_3D_ && boundary == _PERIODIC_){
  dim3 blocks(gridDimX, gridDimY, gridDimZ);

  metropolis3DPeriodicEven<<<blocks, 64>>>(spins_d, randoms_d, local_probabilities_d);
  metropolis3DPeriodicOdd<<<blocks, 64>>>(spins_d, randoms_d, local_probabilities_d);
  cudaDeviceSynchronize();
}

}

```

```
33 void Lattice::initializeRandoms(){
34
35
36 if(dimension == _ISING_2D_){
37   for (int i = 0; i < 64 * gridDimX*gridDimY; i++){
38     do{
39       randoms_h[i] = (int) ((random_generator->getRandomInteger()) >> 1);
40     } while (randoms_h[i] == 0);
41   }
42   cudaMemcpy(randoms_d, randoms_h, 64 * gridDimX*gridDimY*sizeof(unsigned int),
       cudaMemcpyHostToDevice);
43
44 } else if (dimension == _ISING_3D_){
45   for (int i = 0; i < 64 * gridDimX*gridDimY*gridDimZ; i++){
46   do{
47     randoms_h[i] = (int) ((random_generator->getRandomInteger()) >> 1);
48   } while (randoms_h[i] == 0);
49   }
50   cudaMemcpy(randoms_d, randoms_h, 64 * gridDimX*gridDimY*gridDimZ*sizeof(unsigned int),
       cudaMemcpyHostToDevice);
51 }
52
53 }
54
55 int Lattice::observeTotalMagnetization(){
56 int total_magnetization = 0;
57 if (dimension == _ISING_2D_){
58   dim3 blocks(gridDimX, gridDimY);
59   totalMagnetizationBlockWide2D <<<blocks, 64 >>>(spins_d, magnetization_block_wide_d);
60   cudaMemcpy(magnetization_block_wide, magnetization_block_wide_d, gridDimX*gridDimY*
       sizeof(int), cudaMemcpyDeviceToHost);
61
62   for (int x = 0; x < gridDimX; x++){
63     for (int y = 0; y < gridDimY; y++){
64       total_magnetization = total_magnetization + *(magnetization_block_wide + x + y*
       gridDimX);
65     }
66   }
67
68 }
69 else if (dimension == _ISING_3D_){
70   dim3 blocks(gridDimX, gridDimY, gridDimZ);
71   totalMagnetizationBlockWide3D<<<blocks, 64>>>(spins_d, magnetization_block_wide_d);
72   cudaMemcpy(magnetization_block_wide, magnetization_block_wide_d, gridDimX*gridDimY*
       gridDimZ*sizeof(int), cudaMemcpyDeviceToHost);
73
74   for (int x = 0; x < gridDimX; x++){
75     for (int y = 0; y < gridDimY; y++){
76       for (int z = 0; z < gridDimZ; z++){
77         total_magnetization = total_magnetization + *(magnetization_block_wide + x + y*
       gridDimX + z*gridDimX*gridDimY);
78       }
79     }
80   }
81
82 }
83 return total_magnetization;
84
85 }
```

8.2.2 Demon-Algorithmus

Die Implementierung des Demon-Algorithmus besteht aus insgesamt fünf Header- und drei Source-Dateien.

8.2.2.1 multint.hpp

```
#ifndef MULTINT_HPP
#define MULTINT_HPP
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
/*
32 7-bit signed integers.
i-th bits represents an integer in the range from
-64 to 63
*/
struct multint7_t{
  unsigned int bit[6];
  unsigned int sign;
};

/*
32 16-bit signed integers.
i-th bits represents an integer in the range from
-65536 to 65535
*/
struct multint17_t{
  unsigned int bit[16];
  unsigned int sign;
};

__device__ const struct multint7_t multint7_0 = { { 0, 0, 0, 0, 0, 0 }, 0 };

__device__ const struct multint17_t multint17_0 = { { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0 }, 0 };

__host__ __device__ inline multint7_t operator+ (multint7_t& a, multint7_t& b){

multint7_t result;
unsigned int carry = 0;

for (int i = 0; i < 6; i++){
  result.bit[i] = (a.bit[i] ^ b.bit[i]) ^ carry;
  carry = (a.bit[i] & b.bit[i]) | (a.bit[i] & carry) | (b.bit[i] & carry);
}
result.sign = (a.sign ^ b.sign) ^ carry;

return result;
}



__host__ __device__ inline multint17_t operator+ (multint17_t& a, multint17_t& b){

multint17_t result;
unsigned int carry = 0;

for (int i = 0; i < 16; i++){
  result.bit[i] = (a.bit[i] ^ b.bit[i]) ^ carry;
  carry = (a.bit[i] & b.bit[i]) | (a.bit[i] & carry) | (b.bit[i] & carry);
}
result.sign = (a.sign ^ b.sign) ^ carry;

return result;
```

```
}


__host__ __device__ inline multint17_t castMultint(const multint7_t& a){

multint17_t result = { { a.bit[0], a.bit[1], a.bit[2], a.bit[3], a.bit[4], a.bit[5], a.
    sign, a.sign, a.sign, a.sign, a.sign, a.sign, a.sign, a.sign, a.sign, a.sign }, a.
    sign };

return result;
}


__host__ __device__ inline int getNumber(multint7_t& a, int index){
unsigned int digit_mask = 1 << index;
unsigned int bit_mask = ~63; // 1.....11000000

int number = (a.sign & digit_mask) >> index;
/*
arithmetical right shift for signed variable,
maybe not the best solution, could depend on used c++ system.
*/
number = (number << 31) >> 31;
number = number & bit_mask;

for (int i = 0; i < 6; i++){
  number = number ^ (((a.bit[i] & digit_mask) >> index) << i);
}
return number;
}

__host__ __device__ inline int getNumber(multint17_t& a, int index){
unsigned int digit_mask = 1 << index;
unsigned int bit_mask = ~65535; // 1...110000000000000000

int number = (a.sign & digit_mask) >> index;
/*
arithmetical right shift for signed variable,
maybe not the best solution, could depend on used c++ system.
*/
number = (number << 31) >> 31;
number = number & bit_mask;

for (int i = 0; i < 16; i++){
  number = number ^ (((a.bit[i] & digit_mask) >> index) << i);
}
return number;
}


#endif
```

8.2.2.2 rng.hpp

```
#ifndef RNG_HPP
#define RNG_HPP
#define PI 3.14159265358979323846264338327 95
#define PI2 6.28318530717958647692528676655 90
#define SQRTPI2 2.50662827463100050241576528481 10

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
```

```
9  #include <math.h>
10 /*
11 linear congruential generator with
12 x_{n+1} = (a*x_n + b) % m
13 and x_0 = seed
14
15 for the double values the integervalue is divided with m.
16 */
17
18
19 __host__ __device__ inline void simpleRandomInteger0(int& current_random){
20 current_random = 16807 * (current_random % 127773) - 2836 * (current_random / 127773);
21 if (current_random < 0){
22   current_random += 2147483647;
23 }
24 }
25
26
27 __host__ __device__ inline void simpleRandomInteger1(int& current_random){
28 current_random = 48721 * (current_random % 44488) - 3399 * (current_random / 44488);
29 if (current_random < 0){
30   current_random += 2147483647;
31 }
32 }
33
34 /*
35 MARSAGLIA-ZAMAN generator (substract-with-borrow generator).
36 generates random integers between 0 and 2^32-5 OR random doubles between 0 and 1.
37
38 r=43; s=22; b = 2^32 - 5
39 */
40 class MarsagliaZamanGenerator{
41 private:
42
43 unsigned int values[43];
44 int lower_index = 0; //n-43
45 int higher_index = 21; //n-22
46 int c;
47 public:
48
49
50 MarsagliaZamanGenerator(unsigned int* start_values, int c){
51
52 for (int i = 0; i < 43; i++){
53   this->values[i] = start_values[i];
54 }
55 this->c = c;
56
57 }
58
59 /*
60 default init.
61 */
62 MarsagliaZamanGenerator(){
63 int current_random = 3141592;
64 for (int i = 0; i < 43; i++){
65   simpleRandomInteger0(current_random);
66   this->values[i] = current_random;
67 }
68
69 this->c = 0;
70 }
71 unsigned int getRandomInteger(){
72
73 if (values[higher_index] >= values[lower_index] + c){
74   values[lower_index] = (values[higher_index] - values[lower_index]) - c;
```

```
  c = 0;
}
else{
  values[lower_index] = 4294967291 - ((values[lower_index] + c) - values[higher_index]);
  c = 1;
}
unsigned int rvalue = values[lower_index];


lower_index = (lower_index + 1) % 43;
higher_index = (higher_index + 1) % 43;

return rvalue;
}

/*
compute random double between 0...1
*/
double getRandomDouble(){

if (values[higher_index] >= values[lower_index] + c){
  values[lower_index] = (values[higher_index] - values[lower_index]) - c;
  c = 0;
}
else{
  values[lower_index] = 4294967291 - ((values[lower_index] + c) - values[higher_index]);
  c = 1;
}
int rvalue = values[lower_index];


lower_index = (lower_index + 1) % 43;
higher_index = (higher_index + 1) % 43;

//this is necessary to suppress the sign for the result of the return value.
rvalue = rvalue & ~(1 << 31);

return (double)(rvalue / ((double)~(1 << 31)));

}

/*
generate random standard normal variates using the box-muller-transformation.
(see  G. E. P. Box and Mervin E. Muller, Ann. Math. Statist. Volume 29, Number 2 (1958), 610-611.)
*/
double getRandomStandardNormal(){

double value1 = getRandomDouble();
double value2 = getRandomDouble();

return sqrt(-2 * log(value1))*cos(PI2*value2);

}
};


/*
binomial random number generator. Using the Ahrens and Dieters Algorithm BB( Computing 12, 223-246 (1974))
*/
class BinomialGenerator{
private:
MarsagliaZamanGenerator* random_generator;
int n;
double p;
```

```
/*
beta distribution with a=b > 1.5
*/
inline double betaDist(double a){
double A = a - 1;
double t = sqrt(A + A);
double x;
double s;
double s4;
double u;
do{
  do{
    do{
      s = random_generator->getRandomStandardNormal();
      s4 = s*s*s*s;
      x = 0.5*(1 + (s / t));
    } while (x < 0 || x>1);
    u = random_generator->getRandomDouble();
    if (u <= 1 - s4 / (8 * a - 12)){
      return x;
    }

  } while (u >= 1 - s4 / (8 * a - 8) + 0.5*(s4 / (8 * a - 8))*(s4 / (8 * a - 8)));

} while (log(u) > A*log(4 * x*(1 - x)) + (s*s) / 2);
return x;
}
public:
BinomialGenerator(MarsagliaZamanGenerator* random_generator, int n, double p){
this->random_generator = random_generator;
this->n = n;
this->p = p;

}

unsigned int  getBinomialInteger(){

int m = n;
int k = 0;
double a;
double s;
double q = p;

do{
  if (m % 2 == 0){
  m = m - 1;

  if (random_generator->getRandomDouble() <= q){
    k = k + 1;
  }
}

a = (double)(m + 1) / 2.0;
s = betaDist(a);
if (s <= q){
  k = k + a;
  q = (q - s) / (1 - s);
}
else{
  q = q / s;
}
m = (int) a - 1;
} while (m > 32);

int m2 = 0;
int j = 0;
```

```
double u;
do{
  u = random_generator->getRandomDouble();
  if (u <= q){
  j = j + 1;
}
m2 = m2 + 1;
} while (m2 < m);
return k + j;
}

};

#endif
```

8.2.2.3 routines2D.hpp

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"


#ifndef ROUTINES2D_HPP
#define ROUTINES2D_HPP

#include "Lattice.hpp"

struct coordinates2D_t{
int x;
int y;
};

/*******************************************************
device functions and cuda kernels for a 2D lattice.
*******************************************************/

/*
calculates thread index to even 2D coordinates. (even means: (x+y)%2 == 0)
index is always the thread index

for a increasing index the even coordinates along the positive direction of the x-axis
    are counted.
reaching 32 counts in positive x-direction, a jump in the positive y-direction occurs.
*/
__device__ inline coordinates2D_t indexToCoordinates2DEven(int index);


/*
calculates thread index to odd 2D coordinates. (odd means: (x+y)%2 != 0)
index is always the thread index.

for a increasing index the odd coordinates along the positive direction of the x-axis are
     counted.
reaching 32 counts in positive x-direction, a jump in the positive y-direction occurs.
*/
__device__ inline coordinates2D_t indexToCoordinates2DOdd(int index);


__device__ inline coordinates2D_t getNeighbourPosition2D(coordinates2D_t own_position,
    int neighbour, int L1, int L2);


//load multispin at position from spins_d
```

```
43 __device__ inline unsigned int getMultispin2D(unsigned int* spins_d, const
       coordinates2D_t& position, int L1, int L2);
44
45 __device__ inline void writeMultispin2D(unsigned int multispin, unsigned int* spins_d,
       const coordinates2D_t& position, int L1, int L2);
46
47 //load demons at position from demons_d
48 __device__ inline demons3_t getDemons2D(unsigned int* demons_d, const coordinates2D_t&
       position, int L1, int L2);
49
50 __device__ inline void writeDemons2D(demons3_t demons, unsigned int* demons_d, const
       coordinates2D_t& position, int L1, int L2);
51
52 /*
53 calculates the energy between the local energy between own_multispin and
       neighbour_multispins.
54 */
55 __device__ inline multint7_t localSpinEnergy2D(unsigned int own_multispin, unsigned int*
       neighbour_multispins);
56
57 /*
58 transforms thread position to even lattice site and loads own multispin and
59 neighbouring multispins including own demons.
60
61
62 neighbour_multispins[0]: front multispin
63 neighbour_multispins[1]: back multispin
64 neighbour_multispins[2]: left multispin
65 neighbour_multispins[3]: right multispin
66 */
67 __device__ inline void getEnvironment2DEven(unsigned int* spins_d, unsigned int* demons_d
       , coordinates2D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
       unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2);
68
69 /*
70 transforms thread position to odd lattice site and loads own multispin and
71 neighbouring multispins including own demons.
72 */
73 __device__ inline void getEnvironment2DOdd(unsigned int* spins_d, unsigned int* demons_d,
        coordinates2D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
       unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2);
74
75 /*
76 metropolis demon algorithm in 2D for even coordinates.
77
78 for execution:
79 an (L1/2)*L2 sublattice of even points is splitted in blocks of 32x32 lattice planes in
       the x-y-plane.
80
81 invoke this kernel with:
82 dim3 blocks(L1/64, L2/32);
83 metropolis2DPeriodicEven<<<blocks, 1024>>>(...)
84 */
85 __global__ void metropolis2DPeriodicEven(unsigned int* spins_d, unsigned int* demons_d,
       int L1, int L2);
86
87 /*
88 metropolis demon algorithm in 2D for odd coordinates.
89
90 for execution:
91 an (L1/2)*L2 sublattice of even points is splitted in blocks of 32x32 lattice planes in
       the x-y-plane.
92
93 invoke this kernel with:
94 dim3 blocks(L1/64, L2/32);
95 metropolis32PeriodicOdd<<<blocks, 1024>>>(...)
```

```
96  */
97  __global__ void metropolis2DPeriodicOdd(unsigned int* spins_d, unsigned int* demons_d,
        int L1, int L2);
98
99
100 /*
101 sums up the total energy of the combined (periodic) system block wide.
102 one block consists of 1024 threads.
103
104 invoke this kernel with:
105 dim3 blocks(L1/64, L2/32);
106 totalEnergyBlockWide2DPeriodic<<<blocks, 1024>>>(...)
107
108 for each block the total sum of the energies are saved in 32 successive integers in
        global memory.
109
110 */
111 __global__ void totalEnergyBlockWide2DPeriodic(unsigned int* spins_d, unsigned int*
        demons_d, int* energies_block_wide, int L1, int L2);
112
113
114
115 /*
116 sums up the total energy of the spin (periodic) system block wide.
117 one block consists of 1024 threads.
118
119 invoke this kernel with:
120 dim3 blocks(L1/64, L2/32);
121 totalSpinEnergyBlockWide2DPeriodic<<<blocks, 1024>>>(...)
122
123 for each block the total sum of the energies are saved in 32 successive integers in
        global memory.
124
125 */
126 __global__ void totalSpinEnergyBlockWide2DPeriodic(unsigned int* spins_d, unsigned int*
        demons_d, int* energies_block_wide, int L1, int L2);
127
128
129 /*
130 sums up the total energy of the demon system block wide.
131 one block consists of 1024 threads.
132
133 invoke this kernel with:
134 dim3 blocks(L1/64, L2/32);
135 totalSpinEnergyBlockWide3DPeriodic<<<blocks, 1024>>>(...)
136
137 for each block the total sum of the energies are saved in 32 successive integers in
        global memory.
138
139 */
140 __global__ void totalDemonEnergyBlockWide2D(unsigned int* demons_d, int*
        energies_block_wide, int L1, int L2);
141
142
143 /*
144 sums up the total magnetization of the spin system block wide.
145 one block consists of 1024 threads.
146
147 invoke this kernel with:
148 dim3 blocks(L1/64, L2/32, L3);
149 totalMagnetizationBlockWide2D(unsigned int* spins_d, int* magnetization_block_wide, int
        L1, int L2);
150 <<<blocks, 1024>>>(...)
151
152 for each block the total sum of the energies are saved in 32 successive integers in
        global memory.
```

```

*/
__global__ void totalMagnetizationBlockWide2D(unsigned int* spins_d, int*
    magnetization_block_wide, int L1, int L2);

/*
shift demons on an 64x32 sublattice on the x-y-plane blockwide with a random_offset.

invoke this kernel with:
dim3 blocks(L1/64, L2/32);
shiftDemonsBlockWide<<<blocks, 1024>>>(...)
*/
__global__ void shiftDemonsBlockWide2D(unsigned int* demons_d, unsigned int random_offset
    , int L1, int L2);

/*
shift demons block wide with a random offset.

invoke this kernel with:
dim3 blocks(L1/64, (L2/32)/2);
shiftDemonsBlockWide<<<blocks, 1024>>>(...)
*/
__global__ void shiftDemonsGridWide2D(unsigned int* demons_d, unsigned int random_offset,
     int L1, int L2);

__global__ void demonLayersOccupationBlockWide2D(unsigned int* demons_d, unsigned int*
    demon_block_attributes_d, int L1, int L2);

__global__ void applyPatternRemove2D(unsigned int* demons_d, unsigned int*
    random_pattern_d, int layer, int L1, int L2);

__global__ void applyPatternSet2D(unsigned int* demons_d, unsigned int* random_pattern_d,
     int layer, int L1, int L2);

__global__ void updateDemons2D(unsigned int* demons_d, unsigned int* randoms, double
    set_probabilities[], int L1, int L2);
#endif
```

8.2.2.4 routines3D.hpp

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"



#ifndef ROUTINES3D_HPP
#define ROUTINES3D_HPP

#include "Lattice.hpp"

struct coordinates3D_t{
int x;
int y;
int z;
};

/*******************************************************
device functions and cuda kernels for a 3D lattice.
*******************************************************/

/*
calculates thread index to even 3D coordinates. (even means: (x+y+z)%2 == 0)
```

```
index is always the thread index

for a increasing index the even coordinates along the positive direction of the x-axis
    are counted.
reaching 32 counts in positive x-direction, a jump in the positive y-direction occurs.
*/
__device__ inline coordinates3D_t indexToCoordinates3DEven(int index);


/*
calculates thread index to odd 3D coordinates. (odd means: (x+y+z)%2 != 0)
index is always the thread index.

for a increasing index the odd coordinates along the positive direction of the x-axis are
     counted.
reaching 32 counts in positive x-direction, a jump in the positive y-direction occurs.
*/
__device__ inline coordinates3D_t indexToCoordinates3DOdd(int index);


__device__ inline coordinates3D_t getNeighbourPosition3D(coordinates3D_t own_position,
    int neighbour, int L1, int L2, int L3);

//load multispin at position from spins_d
__device__ inline unsigned int getMultispin3D(unsigned int* spins_d, const
    coordinates3D_t& position, int L1, int L2, int L3);


__device__ inline void writeMultispin3D(unsigned int multispin, unsigned int* spins_d,
    const coordinates3D_t& position, int L1, int L2, int L3);

//load demons at position from demons_d
__device__ inline demons3_t getDemons3D(unsigned int* demons_d, const coordinates3D_t&
    position, int L1, int L2, int L3);

__device__ inline void writeDemons3D(demons3_t demons, unsigned int* demons_d, const
    coordinates3D_t& position, int L1, int L2, int L3);


/*
calculates the energy between the local energy between own_multispin and
    neighbour_multispins.
*/
__device__ inline multint7_t localSpinEnergy3D(unsigned int own_multispin, unsigned int*
    neighbour_multispins);


/*
transforms thread position to even lattice site and loads own multispin and
neighbouring multispins including own demons.


neighbour_multispins[0]: front multispin
neighbour_multispins[1]: back multispin
neighbour_multispins[2]: left multispin
neighbour_multispins[3]: right multispin
neighbour_multispins[4]: upper multispin
neighbour_multispins[5]: lower multispin

*/
__device__ inline void getEnvironment3DEven(unsigned int* spins_d, unsigned int* demons_d
    , coordinates3D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
    unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2, int L3)
    ;


```

```
/*
transforms thread position to odd lattice site and loads own multispin and
neighbouring multispins including own demons.
*/
__device__ inline void getEnvironment3DOdd(unsigned int* spins_d, unsigned int* demons_d,
     coordinates3D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
    unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2, int L3)
    ;

/*
metropolis demon algorithm in 3D for even coordinates.

for execution:
an (L1/2)*L2*L3 sublattice of even points is splitted in blocks of 32x32 lattice planes
    in the x-y-plane.

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
metropolis3DPeriodicEven<<<blocks, 1024>>>(...)
*/
__global__ void metropolis3DPeriodicEven(unsigned int* spins_d, unsigned int* demons_d,
    int L1, int L2, int L3);


/*
metropolis demon algorithm in 3D for odd coordinates.

for execution:
an (L1/2)*L2*L3 sublattice of odd points is splitted in blocks of 32x32 lattice planes in
     the x-y-plane.

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
metropolis3DPeriodicEven<<<blocks, 1024>>>(...)
*/
__global__ void metropolis3DPeriodicOdd(unsigned int* spins_d, unsigned int* demons_d,
    int L1, int L2, int L3);

/*
sums up the total energy of the combined (periodic) system block wide.
one block consists of 1024 threads.

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
totalEnergyBlockWide3DPeriodic<<<blocks, 1024>>>(...)

for each block the total sum of the energies are saved in 32 successive integers in
    global memory.

*/
__global__ void totalEnergyBlockWide3DPeriodic(unsigned int* spins_d, unsigned int*
    demons_d, int* energies_block_wide, int L1, int L2, int L3);


/*
sums up the total energy of the spin (periodic) system block wide.
one block consists of 1024 threads.

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
totalSpinEnergyBlockWide3DPeriodic<<<blocks, 1024>>>(...)

for each block the total sum of the energies are saved in 32 successive integers in
    global memory.

*/
```

```
__global__ void totalSpinEnergyBlockWide3DPeriodic(unsigned int* spins_d, unsigned int*
    demons_d, int* energies_block_wide, int L1, int L2, int L3);

/*
sums up the total energy of the demon system block wide.
one block consists of 1024 threads.

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
totalDemonEnergyBlockWide3D<<<blocks, 1024>>>(...)

for each block the total sum of the energies are saved in 32 successive integers in
    global memory.

*/
__global__ void totalDemonEnergyBlockWide3D(unsigned int* demons_d, int*
    energies_block_wide, int L1, int L2, int L3);

/*
sums up the total magnetization of the spin system block wide.
one block consists of 1024 threads.

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
totalMagnetizationBlockWide3D(unsigned int* spins_d, int* magnetization_block_wide, int
    L1, int L2, int L3)<<<blocks, 1024>>>(...)

for each block the total sum of the energies are saved in 32 successive integers in
    global memory.

*/
__global__ void totalMagnetizationBlockWide3D(unsigned int* spins_d, int*
    magnetization_block_wide, int L1, int L2, int L3);

/*
shift demons on an 64x32 sublattice on the x-y-plane blockwide with a random_offset.

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
shiftDemonsBlockWide<<<blocks, 1024>>>(...)
*/
__global__ void shiftDemonsBlockWide3D(unsigned int* demons_d, unsigned int random_offset
    , int L1, int L2, int L3);

/*
shift demons block wide with a random offset.

invoke this kernel with:
dim3 blocks(L1/64, (L2/32)/2, L3);
shiftDemonsBlockWide<<<blocks, 1024>>>(...)
*/
__global__ void shiftDemonsGridWide3D(unsigned int* demons_d, unsigned int random_offset,
     int L1, int L2, int L3);


/*
calcualtes the number of occupied demons in the i-th layer, block wide.
the result is saved in demon_block_attributes_d, for each block.
for each block the results are saved in 3 successive integers in global memory. (first
    layer with energy 4, second layer with energy 8, etc.)

invoke this kernel with:
dim3 blocks(L1/64, L2/32, L3);
demonLayerOccupationBlockWide3D(unsigned int* spins_d, int* magnetization_block_wide, int
     L1, int L2, int L3)<<<blocks, 1024>>>(...)
*/
```

```
__global__ void demonLayersOccupationBlockWide3D(unsigned int* demons_d, unsigned int*
    demon_block_attributes_d, int L1, int L2, int L3);


__global__ void applyPatternRemove3D(unsigned int* demons_d, unsigned int*
    random_pattern_d, int layer, int L1, int L2, int L3);

__global__ void applyPatternSet3D(unsigned int* demons_d, unsigned int* random_pattern_d,
     int layer, int L1, int L2, int L3);

__global__ void updateDemons3D(unsigned int* demons_d, unsigned int* randoms, double
    set_probabilities[], int L1, int L2, int L3);

#endif
```

8.2.2.5 Lattice.hpp

```
#define _ISING_2D_ 2
#define _ISING_3D_ 3
#define _PERIODIC_ 1
#define _ANTIPERIODIC_ -1

#define _OWN_ 0
#define _FRONT_ 1 //positive x-driection
#define _BACK_  2 //negative x-direction
#define _LEFT_  3 //positive y-direction
#define _RIGHT_ 4 //negative y-direction
#define _UPPER_ 5 //positive z-direction
#define _LOWER_ 6 //negative z-direction

#define DEMON_LAYERS 3
#define DEMON_LAYER_LOW 0
#define DEMON_LAYER_MID 1
#define DEMON_LAYER_HIGH 2



#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <math.h>
#include <stdio.h>

#ifndef LATTICE_HPP
#define LATTICE_HPP

#include "multint.hpp"
#include "rng.hpp"

/*
3bit demons data type

bit order (from most significant bit to lowest) :    hdemons mdemons ldemons
energy level:                      16    8   4
*/
struct demons3_t{
unsigned int hdemons;
unsigned int mdemons;
unsigned int ldemons;
};

/*
calculates the energy of demons and returns the result as multint7_t
```

```
46 */
47 __device__ inline multint7_t demonEnergy7(demons3_t& demons){
48 //cast demons3_t to multiint7_t
49 multint7_t demon_energy = multint7_0;
50
51 multint7_t current_demon_level = multint7_0;
52
53 //demon energy 4 (ldemons)
54 current_demon_level.bit[2] = demons.ldemons;
55 demon_energy = demon_energy + current_demon_level;
56 current_demon_level = multint7_0;
57
58 //demon energy 8 (mdemons)
59 current_demon_level.bit[3] = demons.mdemons;
60 demon_energy = demon_energy + current_demon_level;
61 current_demon_level = multint7_0;
62
63 //demon energy 16 (hdemons)
64 current_demon_level.bit[4] = demons.hdemons;
65 demon_energy = demon_energy + current_demon_level;
66
67 return demon_energy;
68 }
69
70 /*
71 calculates the energy of demons and returns the result as multint17_t
72 */
73 __device__ inline multint17_t demonEnergy17(demons3_t& demons){
74 //cast demons3_t to multiint17_t
75 multint17_t demon_energy = multint17_0;
76
77 multint17_t current_demon_level = multint17_0;
78
79 //demon energy 4 (ldemons)
80 current_demon_level.bit[2] = demons.ldemons;
81 demon_energy = demon_energy + current_demon_level;
82 current_demon_level = multint17_0;
83
84 //demon energy 8 (mdemons)
85 current_demon_level.bit[3] = demons.mdemons;
86 demon_energy = demon_energy + current_demon_level;
87 current_demon_level = multint17_0;
88
89 //demon energy 16 (hdemons)
90 current_demon_level.bit[4] = demons.hdemons;
91 demon_energy = demon_energy + current_demon_level;
92
93 return demon_energy;
94 }
95
96 /*
97 class contains all needed informations for the simulation of a Ising-Lattice
98
99 options:
100
101 int dimension:
102 2D lattice: _ISING_2D_
103 3D lattice: _ISING_3D_
104
105 int boundary:
106 periodic boundary conditions: _PERIODIC_
107 antiperiodic boundary conditions: _ANTIPERIODIC_ (not implemented yet)
108
109 create an object of the lattice class with
110
111 2D Lattice: Lattice a(L1, L2, boundary, beta);
```

```
3D Lattice: Lattice a(L1, L2, L3, boundary, beta);

where
L1: size of the lattice in x-direction
L2: size of the lattice in y-direction
L3: size of the lattice in z-direction
beta: inverse temperature of the heat bath.

for a correct execution of the algorithms consider following restrictions:

a.) 2D and 3D lattice: L1 has to be a multiple of 64. L2 has to be a multiple of 32
b.) 3D lattice: L3 > 0 && L3 % 2 == 0.
c.) the bit operators >>, << have to be arithmetical shifts, otherwise adjustments in
    getNumber (multint.hpp) are required.

If this restrictions are not fullfilled, an undefined behaviour will be expected.

*/
class Lattice{

private:
int dimension;
int boundary;

int L1;
int L2;
int L3;

int gridDimX;
int gridDimY;
int gridDimZ;

double beta; //inverse temperature


unsigned int* spins_d;  //multispin array, on device memory
unsigned int* demons_d; //demons array, on device memory
unsigned int* demons_h; //demons array, on host

int* energies_block_wide_d; //on device
int* energies_block_wide; //on host

int* magnetization_block_wide_d; //on device
int* magnetization_block_wide; // on host

unsigned int* random_pattern;
unsigned int* random_pattern_d;
/*
for each block the attributes are saved in 8 successive integers
structure:
.....|first layer occupation|second layer occupation|third layer occupation|first layer
    change|second layer change|third layer change|random seed|random seed|....
*/
unsigned int* demon_block_attributes; //on host
unsigned int* demon_block_attributes_d; //on device


MarsagliaZamanGenerator* random_generator;

double demon_occupation_probabilities[DEMON_LAYERS];
BinomialGenerator* demon_layer_occupation_dist[DEMON_LAYERS];

void demonRandomChangesSet3D(int &changes, int layer, int pattern_fact);
void demonRandomChangesRemove3D(int &changes, int layer, int pattern_fact);

void demonRandomChangesSet2D(int &changes, int layer, int pattern_fact);
```

```
176 void demonRandomChangesRemove2D(int &changes, int layer, int pattern_fact);
177
178 double set_probabilities[3];
179 double* set_probabilities_d;
180
181 public:
182 //2D constructor
183 Lattice(int L1, int L2, int boundary,double beta){
184 this->dimension = _ISING_2D_;
185 this->boundary = boundary;
186
187 this->L1 = L1;
188 this->L2 = L2;
189
190 this->beta = beta;
191
192 this->gridDimX = (this->L1) / 64;
193 this->gridDimY = (this->L2) / 32;
194 this->gridDimZ = (this->L3);
195
196
197 random_generator = new MarsagliaZamanGenerator();
198
199 demon_occupation_probabilities[DEMON_LAYER_LOW] = exp(-4 * beta)/(1+exp(-4*beta));
200 demon_layer_occupation_dist[DEMON_LAYER_LOW] = new BinomialGenerator(random_generator, 32
      * L1*L2, demon_occupation_probabilities[DEMON_LAYER_LOW]);
201
202 demon_occupation_probabilities[DEMON_LAYER_MID] = exp(-8 * beta) / (1 + exp(-8 * beta));
203 demon_layer_occupation_dist[DEMON_LAYER_MID] = new BinomialGenerator(random_generator, 32
      * L1*L2, demon_occupation_probabilities[DEMON_LAYER_MID]);
204
205 demon_occupation_probabilities[DEMON_LAYER_HIGH] = exp(-16 * beta) / (1 + exp(-16 * beta)
    );
206 demon_layer_occupation_dist[DEMON_LAYER_HIGH] = new BinomialGenerator(random_generator,
    32 * L1*L2, demon_occupation_probabilities[DEMON_LAYER_HIGH]);
207
208 cudaMalloc(&(this->spins_d), L1*L2*sizeof(unsigned int));
209 cudaMalloc(&(this->demons_d), 3 * L1*L2*sizeof(unsigned int));
210 cudaMallocHost(&(this->demons_h), 3 * L1*L2*sizeof(unsigned int));
211
212 set_probabilities[0] = exp(-4 * beta) / (1 + exp(-4 * beta));
213 set_probabilities[1] = exp(-8 * beta) / (1 + exp(-8 * beta));
214 set_probabilities[2] = exp(-16 * beta) / (1 + exp(-16 * beta));
215
216
217 cudaMalloc(&set_probabilities_d, 3 * sizeof(double));
218 cudaMemcpy(set_probabilities_d, set_probabilities, 3 * sizeof(double),
    cudaMemcpyHostToDevice);
219 unsigned int* init_array = new unsigned int[L1*L2 * 3];
220 for (int i = 0; i < L1*L2; i++){
221 //init_array[i] = random_generator->getRandomInteger();
222 init_array[i] = 0;
223 }
224 cudaMemcpy(spins_d, init_array, L1*L2*sizeof(unsigned int), cudaMemcpyHostToDevice);
225
226 for (int i = 0; i < 3 * L1*L2; i++){
227 //init_array[i] = random_generator->getRandomInteger();
228 init_array[i] = 0;
229 }
230 cudaMemcpy(demons_d, init_array, 3 * L1*L2*sizeof(unsigned int), cudaMemcpyHostToDevice);
231
232 cudaMalloc(&(this->energies_block_wide_d), (32 * (L1*L2) / 2048) * sizeof(int));
233 cudaMallocHost(&(this->energies_block_wide), (32 * (L1*L2) / 2048) * sizeof(int));
234
235 cudaMalloc(&(this->demon_block_attributes_d), (4 * (L1*L2) / 2048)*sizeof(unsigned int));
236 cudaMallocHost(&demon_block_attributes, (4 * (L1*L2) / 2048)*sizeof(unsigned int));
```

```
237
238
239 cudaMemcpy(demon_block_attributes_d, demon_block_attributes, (4 * (L1*L2) / 2048)*sizeof(
        unsigned int), cudaMemcpyHostToDevice);
240
241
242
243 cudaMalloc(&random_pattern_d, L1*L2*sizeof(unsigned int));
244 //cudaMallocHost(&random_pattern, L1*L*sizeof(unsigned int));//allocate pinned memory
245 random_pattern = demons_h;
246
247 initializeRandoms();
248
249 magnetization_block_wide_d = energies_block_wide_d;
250 magnetization_block_wide = energies_block_wide;
251
252 delete[] init_array;
253 }
254 Lattice(int L1, int L2, int L3, int boundary, double beta){
255 this->L1 = L1;
256 this->L2 = L2;
257 this->L3 = L3;
258
259 this->dimension = _ISING_3D_;
260 this->boundary = boundary;
261 this->beta = beta;
262
263 this->gridDimX = (this->L1) / 64;
264 this->gridDimY = (this->L2) / 32;
265 this->gridDimZ = (this->L3);
266
267
268 random_generator = new MarsagliaZamanGenerator();
269
270 demon_occupation_probabilities[DEMON_LAYER_LOW] = exp(-4 * beta) / (1.0 + (double) exp(-4
        * beta));
271 demon_layer_occupation_dist[DEMON_LAYER_LOW] = new BinomialGenerator(random_generator, 32
        * L1*L2*L3, demon_occupation_probabilities[DEMON_LAYER_LOW]);
272
273 demon_occupation_probabilities[DEMON_LAYER_MID] = exp(-8 * beta) / (1.0 + (double) exp(-8
        * beta));
274 demon_layer_occupation_dist[DEMON_LAYER_MID] = new BinomialGenerator(random_generator, 32
        * L1*L2*L3, demon_occupation_probabilities[DEMON_LAYER_MID]);
275
276 demon_occupation_probabilities[DEMON_LAYER_HIGH] = exp(-16 * beta) / (1.0 + (double) exp
        (-16 * beta));
277 demon_layer_occupation_dist[DEMON_LAYER_HIGH] = new BinomialGenerator(random_generator,
        32 * L1*L2*L3, demon_occupation_probabilities[DEMON_LAYER_HIGH]);
278
279 cudaMalloc(&(this->spins_d), L1*L2*L3*sizeof(unsigned int));
280 cudaMalloc(&(this->demons_d), 3 * L1*L2*L3*sizeof(unsigned int));
281 cudaMallocHost(&(this->demons_h), 3 * L1*L2*L3*sizeof(unsigned int));
282
283
284 set_probabilities[0] = exp(-4 * beta) / (1 + exp(-4 * beta));
285 set_probabilities[1] = exp(-8 * beta) / (1 + exp(-8 * beta));
286 set_probabilities[2] = exp(-16 * beta) / (1 + exp(-16 * beta));
287
288 cudaMalloc(&set_probabilities_d, 3 * sizeof(double));
289 cudaMemcpy(set_probabilities_d, set_probabilities, 3 * sizeof(double),
        cudaMemcpyHostToDevice);
290 unsigned int* init_array = new unsigned int[L1*L2*L3 * 3];
291
292 for (int i = 0; i < L1*L2*L3; i++){
293 //init_array[i] = random_generator->getRandomInteger();
294 init_array[i] = 0;
```

```
295 }
296
297 cudaMemcpy(spins_d, init_array, L1*L2*L3*sizeof(unsigned int), cudaMemcpyHostToDevice);
298
299 for (int i = 0; i < 3 * L1*L2*L3; i++){
300
301 //init_array[i] = random_generator->getRandomInteger();
302 init_array[i] = 0;
303 }
304
305 cudaMemcpy(demons_d, init_array, 3 * L1*L2*L3*sizeof(unsigned int),
        cudaMemcpyHostToDevice);
306
307 cudaMalloc(&(this->energies_block_wide_d), (32 * (L1*L2*L3) / 2048) * sizeof(int));
308 cudaMallocHost(&(this->energies_block_wide), (32 * (L1*L2*L3) / 2048) * sizeof(int));
309
310 cudaMalloc(&(this->demon_block_attributes_d), (4 * (L1*L2*L3) / 2048)*sizeof(unsigned int
        ));
311 cudaMallocHost(&demon_block_attributes, (4 * (L1*L2*L3) / 2048)*sizeof(unsigned int));
312
313
314 cudaMemcpy(demon_block_attributes_d, demon_block_attributes, (4 * (L1*L2*L3) / 2048)*
        sizeof(unsigned int), cudaMemcpyHostToDevice);
315
316 cudaMalloc(&random_pattern_d, L1*L2*L3*sizeof(unsigned int));
317 //cudaMallocHost(&random_pattern, L1*L2*L3*sizeof(unsigned int));//allocate pinned memory
318 random_pattern = demons_h;
319
320 initializeRandoms();
321
322 magnetization_block_wide_d = energies_block_wide_d;
323 magnetization_block_wide   = energies_block_wide;
324
325 delete[] init_array;
326 }
327
328
329 ~Lattice(){
330
331 cudaFree(spins_d);
332 cudaFree(demons_d);
333 cudaFree(energies_block_wide_d);
334 cudaFree(random_pattern_d);
335 //cudaFreeHost(random_pattern);
336 cudaFreeHost(demons_h);
337 cudaFreeHost(demon_block_attributes);
338 cudaFreeHost(energies_block_wide);
339
340 delete random_generator;
341
342 }
343 /*
344 updates the combined system (spins+demons) microcanonical and shifts
345 the demons.
346 */
347 void updateSystem();
348
349 /*
350 updates the i-th demon layer trough an heat bath with the inverse tempreature beta.
351 */
352 void updateDemonLayers();
353
354
355
356 void initializeRandoms();
357
```

```
358
359
360
361 void updateDemonLayersNew();
362 /*
363 obeserves the total energy of all 32 combined systems configurations. The result is
364 written in energies.
365
366 energies has to be allocated (energies = new int[32]) outside the function.
367 */
368 void observeTotalEnergies(int* energies);
369
370 /*
371 obeserves the total spin energy of all 32 spin configurations. The result is
372 written in energies.
373
374 energies has to be allocated (energies = new int[32]) outside the function.
375 */
376 void observeTotalSpinEnergies(int* energies);
377
378 /*
379 obeserves the total demon energy of all 32 demon configurations. The result is
380 written in energies.
381
382 energies has to be allocated (energies = new int[32]) outside the function.
383 */
384 void observeTotalDemonEnergies(int* energies);
385
386
387 /*
388 obeserves the total magnetization of all 32 spin configurations. The result is
389 written in energies.
390
391 energies has to be allocated (energies = new int[32]) outside the function.
392 */
393 void observeTotalMagnetization(int* magnetization);
394
395
396 };
397
398 #endif
```

8.2.2.6 routines2D.cu

```
1 #include "cuda_runtime.h"
2 #include "device_launch_parameters.h"
3
4 #include "routines2D.hpp"
5 #include <stdio.h>
6
7 __device__ inline coordinates2D_t indexToCoordinates2DEven(int index){
8 coordinates2D_t position;
9
10 //origin of the 32x32 sub lattice.
11 int pointX = blockIdx.x * 64;
12 int pointY = blockIdx.y * 32;
13 /*
14 calculate position in sub lattice 32x32
15 i.e.: transform index to coordinates of the full lattice
16 */
17 position.y = index / 32 + pointY;
18 position.x = index % 32;
```

```
19
20
21 //expand coordinates to all even points in the lattice.
22
23 //first coordinate in x-direction (0,y,z) is even
24 if ((pointX + position.y) % 2 == 0){
25   position.x = 2 * position.x + pointX;
26 }
27 //first coordinate in x-direction (0,y,z) is odd
28 else if ((pointX + position.y) % 2 != 0){
29   position.x = 2 * position.x + 1 + pointX;
30 }
31
32 return position;
33 }
34
35
36
37 __device__ inline coordinates2D_t indexToCoordinates2DOdd(int index){
38 coordinates2D_t position;
39
40 int pointX = blockIdx.x * 64;
41 int pointY = blockIdx.y * 32;
42
43 //calculate position in sub lattice 32x32
44 position.y = index / 32 + pointY;
45 position.x = index % 32;
46
47 //expand coordinates to all even points in the lattice.
48
49 //first coordinate in x-direction (0,y,z) is odd
50 if ((pointX + position.y) % 2 != 0){
51   position.x = 2 * position.x + pointX;
52 }
53 //first coordinate in x-direction (0,y,z) is even
54 else if ((pointX + position.y) % 2 == 0){
55   position.x = 2 * position.x + 1 + pointX;
56 }
57
58 return position;
59 }
60
61
62 __device__ inline coordinates2D_t getNeighbourPosition2D(coordinates2D_t own_position,
     int neighbour, int L1, int L2) {
63
64
65 int x = own_position.x;
66 int y = own_position.y;
67
68 switch (neighbour){
69
70 case _FRONT_:
71   //regard boundary conditions
72   if (x + 1 > L1 - 1){
73     own_position.x = 0;
74   }
75   else{
76     own_position.x = x + 1;
77   }
78 break;
79
80 case _BACK_:
81   //regard boundary conditions
82   if (x - 1 < 0){
83     own_position.x = L1 - 1;
```

```
  }
  else{
  own_position.x = x - 1;
  }
break;

case _LEFT_:
  //regard boundary conditions
  if (y + 1 > L2 - 1){
    own_position.y = 0;
  }
  else{
    own_position.y = y + 1;
  }
break;

case _RIGHT_:
  //regard boundary conditions
  if (y - 1 < 0){
    own_position.y = L2 - 1;
  }
  else{
    own_position.y = y - 1;
  }
break;
}

return own_position;
}


__device__ inline unsigned int getMultispin2D(unsigned int* spins_d, const
    coordinates2D_t& position, int L1, int L2){

int x = position.x;
int y = position.y;

int sub_L1 = L1 / 2;

//coordinates even?
if ((x + y) % 2 == 0){
  //reverse transformation in indexToCoordinates2DEven
  if (y % 2 == 0){
    x = x / 2;
  }
  else{
    x = (x - 1) / 2;
  }
  return *(spins_d + x + y*sub_L1);
}
else{
  //reverse transformation in indexToCoordinates2DOdd
  if (y % 2 != 0){
    x = x / 2;
  }
  else{
    x = (x - 1) / 2;
  }
  return *(spins_d + x + y*sub_L1 + sub_L1 * L2);
}

}

__device__ inline void writeMultispin2D(unsigned int multispin, unsigned int* spins_d,
    const coordinates2D_t& position, int L1, int L2){

```

```
148 int x = position.x;
149 int y = position.y;
150
151 int sub_L1 = L1 / 2;
152
153 //coordinates even?
154 if ((x + y) % 2 == 0){
155   //reverse transformation in indexToCoordinates2DEven
156   if (y % 2 == 0){
157     x = x / 2;
158   }
159   else{
160     x = (x - 1) / 2;
161   }
162   *(spins_d + x + y*sub_L1) = multispin;
163 }
164 else{
165 //reverse transformation in indexToCoordinates2DOdd
166   if (y % 2 != 0){
167     x = x / 2;
168   }
169   else{
170     x = (x - 1) / 2;
171   }
172   *(spins_d + x + y*sub_L1 + sub_L1 * L2) = multispin;
173 }
174 }
175
176
177 __device__ inline demons3_t getDemons2D(unsigned int* demons_d, const coordinates2D_t&
      position, int L1, int L2){
178 int x = position.x;
179 int y = position.y;
180
181 int sub_L1 = L1 / 2;
182
183 demons3_t demons;
184
185 //coordinates even?
186 if ((x + y) % 2 == 0){
187   //reverse transformation in indexToCoordinates2DEven
188   if (y % 2 == 0){
189     x = x / 2;
190   }
191   else{
192     x = (x - 1) / 2;
193   }
194   demons.hdemons = *(demons_d + x + y*sub_L1);
195   demons.mdemons = *(demons_d + x + y*sub_L1 + sub_L1*L2);
196   demons.ldemons = *(demons_d + x + y*sub_L1 + 2 * sub_L1*L2);
197 }
198 else{
199   //reverse transformation in indexToCoordinates2DOdd
200   if (y % 2 != 0){
201     x = x / 2;
202   }
203   else{
204     x = (x - 1) / 2;
205   }
206   demons.hdemons = *(demons_d + x + y*sub_L1 + 3 * sub_L1*L2);
207   demons.mdemons = *(demons_d + x + y*sub_L1 + 4 * sub_L1*L2);
208   demons.ldemons = *(demons_d + x + y*sub_L1 + 5 * sub_L1*L2);
209 }
210
211 return demons;
212 }
```

```
__device__ inline void writeDemons2D(demons3_t demons, unsigned int* demons_d, const
    coordinates2D_t& position, int L1, int L2){
int x = position.x;
int y = position.y;

int sub_L1 = L1 / 2;

//coordinates even?
if ((x + y) % 2 == 0){
  //reverse transformation in indexToCoordinates2DEven
  if (y % 2 == 0){
    x = x / 2;
  }
  else{
    x = (x - 1) / 2;
  }
  *(demons_d + x + y*sub_L1) = demons.hdemons;
  *(demons_d + x + y*sub_L1 + sub_L1*L2) = demons.mdemons;
  *(demons_d + x + y*sub_L1 + 2 * sub_L1*L2) = demons.ldemons;
}
else{
  //reverse transformation in indexToCoordinates2DOdd
  if (y % 2 != 0){
    x = x / 2;
  }
  else{
    x = (x - 1) / 2;
  }
  *(demons_d + x + y*sub_L1 + 3 * sub_L1*L2) = demons.hdemons;
  *(demons_d + x + y*sub_L1 + 4 * sub_L1*L2) = demons.mdemons;
  *(demons_d + x + y*sub_L1 + 5 * sub_L1*L2) = demons.ldemons;

}
}


__device__ inline multint7_t localSpinEnergy2D(unsigned int own_multispin, unsigned int*
    neighbour_multispins){

multint7_t sum_energy = multint7_0;
multint7_t current_edge_energy;
unsigned int parallel;

for (int i = 0; i < 4; i++){

  //bit 0: spins antiparallel; bit 1: spins parallel
  parallel = ~(neighbour_multispins[i] ^ own_multispin);

  //if antiparallel set current_edge_energy = 1, else -1
  current_edge_energy.bit[0] = ~0;
  for (int i = 1; i < 6; i++){
    current_edge_energy.bit[i] = parallel;
  }
  current_edge_energy.sign = parallel;

  //add edge energy to total energy.
  sum_energy = sum_energy + current_edge_energy;

}

return sum_energy;
}
```

```
__device__ inline void getEnvironment2DEven(unsigned int* spins_d, unsigned int* demons_d
    , coordinates2D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
    unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2){


own_position = indexToCoordinates2DEven(threadIdx.x);

coordinates2D_t block_origin;
block_origin.x = blockIdx.x * 64;
block_origin.y = blockIdx.y * 32;

int x = own_position.x;
int y = own_position.y;



coordinates2D_t neighbour_position;

own_multispin = getMultispin2D(spins_d, own_position, L1, L2);
own_demons = getDemons2D(demons_d, own_position, L1, L2);

/*
neighbour_multispins[0]: front multispin
neighbour_multispins[1]: back multispin
neighbour_multispins[2]: left multispin
neighbour_multispins[3]: right multispin
*/


//load neigbhour multispins on the x-y-plane in spin_chunk
coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
spin_chunk[threadIdx.x] = getMultispin2D(spins_d, odd_position, L1, L2);
__syncthreads(); //IMPORTANT, prevents race condition in shared memory


/*
load front and back multispins to local variables.
*/
//is the first point in the row (block.origin.x, y, z) even ?
if ((block_origin.x + y) % 2 == 0){

  //write front multispin
  neighbour_multispins[0] = spin_chunk[threadIdx.x];
  if (x > block_origin.x){
    //write back multispin
    neighbour_multispins[1] = spin_chunk[threadIdx.x - 1];
  }
  else{
    neighbour_position = getNeighbourPosition2D(own_position, _BACK_, L1, L2);
    //write back multispin
    neighbour_multispins[1] = getMultispin2D(spins_d, neighbour_position, L1, L2);
  }
}
else{

  //write back multispin
  neighbour_multispins[1] = spin_chunk[threadIdx.x];

  if (x < block_origin.x + 63){

    //write front multispin
    neighbour_multispins[0] = spin_chunk[threadIdx.x + 1];
  }
  else{
    neighbour_position = getNeighbourPosition2D(own_position, _FRONT_, L1, L2);
    //write front multispin
```

```
    neighbour_multispins[0] = getMultispin2D(spins_d, neighbour_position, L1, L2);
}
}

/*
load left and right multispins to local variables.
*/
//is the first point in the column (x, block_origin.y, z) even?
if ((x + block_origin.y) % 2 == 0){

  //write left multispin
  neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];

  if (y > block_origin.y){
    //write right multispin
    neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];
  }
else{
  neighbour_position = getNeighbourPosition2D(own_position, _RIGHT_, L1, L2);
  //write right multispin
  neighbour_multispins[3] = getMultispin2D(spins_d, neighbour_position, L1, L2);
}
}
else{

  //write right multispin
  neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];

  if (y < block_origin.y + 31){
    //write left multispin
    neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];
  }
  else{
    neighbour_position = getNeighbourPosition2D(own_position, _LEFT_, L1, L2);
    //write left multispin
    neighbour_multispins[2] = getMultispin2D(spins_d, neighbour_position, L1, L2);
  }
}
}


__device__ inline void getEnvironment2DOdd(unsigned int* spins_d, unsigned int* demons_d,
     coordinates2D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
    unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2){



own_position = indexToCoordinates2DOdd(threadIdx.x);

coordinates2D_t block_origin;
block_origin.x = blockIdx.x * 64;
block_origin.y = blockIdx.y * 32;

int x = own_position.x;
int y = own_position.y;



coordinates2D_t neighbour_position;

own_multispin = getMultispin2D(spins_d, own_position, L1, L2);
own_demons = getDemons2D(demons_d, own_position, L1, L2);

/*
neighbour_multispins[0]: front multispin
neighbour_multispins[1]: back multispin
```

```
neighbour_multispins[2]: left multispin
neighbour_multispins[3]: right multispin
*/


//load neigbhour multispins on the x-y-plane in spin_chunk
coordinates2D_t even_position = indexToCoordinates2DEven(threadIdx.x);
spin_chunk[threadIdx.x] = getMultispin2D(spins_d, even_position, L1, L2);
__syncthreads(); //IMPORTANT, prevents race condition in shared memory


/*
load front and back multispins to local variables.
*/
//is the first point in the row (block.origin.x, y, z) odd ?
if ((block_origin.x + y) % 2 != 0){

  //write front multispin
  neighbour_multispins[0] = spin_chunk[threadIdx.x];
  if (x > block_origin.x){
    //write back multispin
    neighbour_multispins[1] = spin_chunk[threadIdx.x - 1];
  }
  else{
    neighbour_position = getNeighbourPosition2D(own_position, _BACK_, L1, L2);
    //write back multispin
    neighbour_multispins[1] = getMultispin2D(spins_d, neighbour_position, L1, L2);
  }
}
else{

  //write back multispin
  neighbour_multispins[1] = spin_chunk[threadIdx.x];

  if (x < block_origin.x + 63){

    //write front multispin
    neighbour_multispins[0] = spin_chunk[threadIdx.x + 1];
  }
  else{
    neighbour_position = getNeighbourPosition2D(own_position, _FRONT_, L1, L2);
    //write front multispin
    neighbour_multispins[0] = getMultispin2D(spins_d, neighbour_position, L1, L2);
  }
}

/*
load left and right multispins to local variables.
*/
//is the first point in the column (x, block_origin.y, z) even?
if ((x + block_origin.y) % 2 != 0){

  //write left multispin
  neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];

  if (y > block_origin.y){
    //write right multispin
    neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];
  }
  else{
    neighbour_position = getNeighbourPosition2D(own_position, _RIGHT_, L1, L2);
    //write right multispin
    neighbour_multispins[3] = getMultispin2D(spins_d, neighbour_position, L1, L2);
  }
}
else{
```

```
470
471   //write right multispin
472   neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];
473
474   if (y < block_origin.y + 31){
475     //write left multispin
476     neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];
477   }
478   else{
479     neighbour_position = getNeighbourPosition2D(own_position, _LEFT_, L1, L2);
480     //write left multispin
481     neighbour_multispins[2] = getMultispin2D(spins_d, neighbour_position, L1, L2);
482   }
483 }
484 }
485
486
487 __global__ void metropolis2DPeriodicEven(unsigned int* spins_d, unsigned int* demons_d,
        int L1, int L2){
488
489 coordinates2D_t own_position;
490 unsigned int own_multispin;
491 demons3_t own_demons;
492 unsigned int neighbour_multispins[4];
493
494 __shared__ unsigned int spin_chunk[1024];
495
496 getEnvironment2DEven(spins_d, demons_d, own_position, own_multispin, own_demons,
        neighbour_multispins, spin_chunk, L1, L2);
497
498 /*
499 compute if spin change is possible.
500 */
501 multint7_t sum_energy;
502 multint7_t energy_change;
503 multint7_t demon_energy;
504
505 sum_energy = localSpinEnergy2D(own_multispin, neighbour_multispins);
506 demon_energy = demonEnergy7(own_demons);
507
508 energy_change = sum_energy + sum_energy;
509 //demon energy after spin flip
510 demon_energy = demon_energy + energy_change;
511
512 unsigned int flip_decision;
513
514 // if demon_energy is negative, no flip is allowed, bit is set to 0;
515 flip_decision = ~(demon_energy.sign);
516
517 //if demon energy is bigger than 16+8+4 = 28 (=0011100) no flip allowed.
518 flip_decision = flip_decision & ~(demon_energy.bit[5]);
519 flip_decision = flip_decision & (~(demon_energy.bit[4] & demon_energy.bit[3] &
        demon_energy.bit[2]) | (~(demon_energy.bit[1]) & ~(demon_energy.bit[0])));
520
521 //flip spins
522 own_multispin = own_multispin ^ flip_decision;
523
524
525 //cast new demon_energy to demons3_t if spin changed.
526 own_demons.ldemons = ((own_demons.ldemons ^ demon_energy.bit[2]) & flip_decision) ^
        own_demons.ldemons;
527 own_demons.mdemons = ((own_demons.mdemons ^ demon_energy.bit[3]) & flip_decision) ^
        own_demons.mdemons;
528 own_demons.hdemons = ((own_demons.hdemons ^ demon_energy.bit[4]) & flip_decision) ^
        own_demons.hdemons;
529
```

```
530
531 writeMultispin2D(own_multispin, spins_d, own_position, L1, L2);
532 writeDemons2D(own_demons, demons_d, own_position, L1, L2);
533 }
534
535
536 __global__ void metropolis2DPeriodicOdd(unsigned int* spins_d, unsigned int* demons_d,
        int L1, int L2){
537
538 coordinates2D_t own_position;
539 unsigned int own_multispin;
540 demons3_t own_demons;
541 unsigned int neighbour_multispins[4];
542
543 __shared__ unsigned int spin_chunk[1024];
544
545 getEnvironment2DOdd(spins_d, demons_d, own_position, own_multispin, own_demons,
        neighbour_multispins, spin_chunk, L1, L2);
546
547 /*
548 compute if spin change is possible.
549 */
550 multint7_t sum_energy;
551 multint7_t energy_change;
552 multint7_t demon_energy;
553
554 sum_energy = localSpinEnergy2D(own_multispin, neighbour_multispins);
555 demon_energy = demonEnergy7(own_demons);
556
557 energy_change = sum_energy + sum_energy;
558 //demon energy after spin flip
559 demon_energy = demon_energy + energy_change;
560
561 unsigned int flip_decision;
562
563 // if demon_energy is negative, no flip is allowed, bit is set to 0;
564 flip_decision = ~(demon_energy.sign);
565
566 //if demon energy is bigger than 16+8+4 = 28 (=0011100) no flip allowed.
567 flip_decision = flip_decision & ~(demon_energy.bit[5]);
568 flip_decision = flip_decision & (~(demon_energy.bit[4] & demon_energy.bit[3] &
        demon_energy.bit[2]) | (~(demon_energy.bit[1]) & ~(demon_energy.bit[0])));
569
570 //flip spins
571 own_multispin = own_multispin ^ flip_decision;
572
573
574 //cast new demon_energy to demons3_t if spin changed.
575 own_demons.ldemons = ((own_demons.ldemons ^ demon_energy.bit[2]) & flip_decision) ^
        own_demons.ldemons;
576 own_demons.mdemons = ((own_demons.mdemons ^ demon_energy.bit[3]) & flip_decision) ^
        own_demons.mdemons;
577 own_demons.hdemons = ((own_demons.hdemons ^ demon_energy.bit[4]) & flip_decision) ^
        own_demons.hdemons;
578
579
580 writeMultispin2D(own_multispin, spins_d, own_position, L1, L2);
581 writeDemons2D(own_demons, demons_d, own_position, L1, L2);
582 }
583
584
585 __global__ void totalEnergyBlockWide2DPeriodic(unsigned int* spins_d, unsigned int*
        demons_d, int* energies_block_wide, int L1, int L2){
586
587 coordinates2D_t own_position;
588 unsigned int own_multispin;
```

```
589 demons3_t own_demons;
590
591 unsigned int neighbour_multispins[4];
592
593
594 extern  __shared__ int chunk[];
595 unsigned int* spin_chunk = (unsigned int*)&chunk[0];
596
597 getEnvironment2DEven(spins_d, demons_d, own_position, own_multispin, own_demons,
        neighbour_multispins, spin_chunk, L1, L2);
598
599 __syncthreads();
600
601 demons3_t odd_demons;
602
603
604 //load neigbhour demons on the x-y-plane in demon_chunk
605 coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
606
607 odd_demons = getDemons2D(demons_d, odd_position, L1, L2);
608
609 //add up energies
610 multint17_t sum_energy;
611 multint17_t demon_energy;
612 multint17_t demon_energy_odd;
613
614 sum_energy = castMultint(localSpinEnergy2D(own_multispin, neighbour_multispins));
615
616
617 demon_energy = demonEnergy17(own_demons);
618 demon_energy_odd = demonEnergy17(odd_demons);
619 demon_energy = demon_energy + demon_energy_odd;
620
621 sum_energy = sum_energy + demon_energy;
622
623 multint17_t* energy_chunk = (multint17_t*)&chunk[0];
624
625 // first 512 threads loads energies in chunk
626 if (threadIdx.x < 512){
627   energy_chunk[threadIdx.x] = sum_energy;
628 }
629
630 //next 512 threads add there own energies.
631 __syncthreads();
632 if (threadIdx.x >= 512){
633   energy_chunk[threadIdx.x - 512] = energy_chunk[threadIdx.x - 512] + sum_energy;
634 }
635
636 //add energies warp wide
637 __syncthreads();
638 if (threadIdx.x < 512){
639   if (threadIdx.x % 32 == 0){
640     for (int i = 1; i < 32; i++){
641       energy_chunk[threadIdx.x] = energy_chunk[threadIdx.x] + energy_chunk[threadIdx.x +
        i];
642     }
643   }
644 }
645
646 //add energies block wide
647 __syncthreads();
648 if (threadIdx.x == 0){
649   for (int i = 1; i < 16; i++){
650     energy_chunk[0] = energy_chunk[0] + energy_chunk[32 * i];
651   }
652 }
```

```
653
654 //first 32 threads writes results in global memory
655 __syncthreads();
656 if (threadIdx.x < 32){
657   *(energies_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*
      gridDim.y) + threadIdx.x) = getNumber(energy_chunk[0], threadIdx.x);
658 }
659
660 }
661
662
663 __global__ void totalSpinEnergyBlockWide2DPeriodic(unsigned int* spins_d, unsigned int*
      demons_d, int* energies_block_wide, int L1, int L2){
664
665 coordinates2D_t own_position;
666 unsigned int own_multispin;
667 demons3_t own_demons;
668
669 unsigned int neighbour_multispins[4];
670
671
672 extern  __shared__ int chunk[];
673 unsigned int* spin_chunk = (unsigned int*)&chunk[0];
674
675 getEnvironment2DEven(spins_d, demons_d, own_position, own_multispin, own_demons,
      neighbour_multispins, spin_chunk, L1, L2);
676
677 __syncthreads();
678
679
680 //load neigbhour demons on the x-y-plane in demon_chunk
681
682 //add up energies
683 multint17_t sum_energy;
684
685 sum_energy = castMultint(localSpinEnergy2D(own_multispin, neighbour_multispins));
686
687
688 multint17_t* energy_chunk = (multint17_t*)&chunk[0];
689
690 // first 512 threads loads energies in chunk
691 if (threadIdx.x < 512){
692   energy_chunk[threadIdx.x] = sum_energy;
693 }
694
695 //next 512 threads add there own energies.
696 __syncthreads();
697 if (threadIdx.x >= 512){
698   energy_chunk[threadIdx.x - 512] = energy_chunk[threadIdx.x - 512] + sum_energy;
699 }
700
701 //add energies warp wide
702 __syncthreads();
703 if (threadIdx.x < 512){
704   if (threadIdx.x % 32 == 0){
705     for (int i = 1; i < 32; i++){
706       energy_chunk[threadIdx.x] = energy_chunk[threadIdx.x] + energy_chunk[threadIdx.x +
      i];
707     }
708   }
709 }
710
711 //add energies block wide
712 __syncthreads();
713 if (threadIdx.x == 0){
714   for (int i = 1; i < 16; i++){
```

```
715     energy_chunk[0] = energy_chunk[0] + energy_chunk[32 * i];
716   }
717 }
718
719 //first 32 threads writes results in global memory
720 __syncthreads();
721 if (threadIdx.x < 32){
722   *(energies_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*
      gridDim.y) + threadIdx.x) = getNumber(energy_chunk[0], threadIdx.x);
723 }
724
725 }
726
727 __global__ void totalDemonEnergyBlockWide2D(unsigned int* demons_d, int*
      energies_block_wide, int L1, int L2){
728
729
730 coordinates2D_t own_position = indexToCoordinates2DEven(threadIdx.x);
731 coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
732
733 demons3_t own_demons = getDemons2D(demons_d, own_position, L1, L2);
734 demons3_t odd_demons = getDemons2D(demons_d, odd_position, L1, L2);
735
736 multint17_t sum_energy;
737 multint17_t sum_energy_odd;
738
739 sum_energy = demonEnergy17(own_demons);
740 sum_energy_odd = demonEnergy17(odd_demons);
741 sum_energy = sum_energy + sum_energy_odd;
742
743 extern __shared__ int chunk[];
744 multint17_t* energy_chunk = (multint17_t*)&chunk[0];
745
746 // first 512 threads loads energies in chunk
747 if (threadIdx.x < 512){
748   energy_chunk[threadIdx.x] = sum_energy;
749 }
750
751 //next 512 threads add there own energies.
752 __syncthreads();
753 if (threadIdx.x >= 512){
754   energy_chunk[threadIdx.x - 512] = energy_chunk[threadIdx.x - 512] + sum_energy;
755 }
756
757 //add energies warp wide
758 __syncthreads();
759 if (threadIdx.x < 512){
760   if (threadIdx.x % 32 == 0){
761     for (int i = 1; i < 32; i++){
762       energy_chunk[threadIdx.x] = energy_chunk[threadIdx.x] + energy_chunk[threadIdx.x +
      i];
763     }
764   }
765 }
766
767 //add energies block wide
768 __syncthreads();
769 if (threadIdx.x == 0){
770   for (int i = 1; i < 16; i++){
771     energy_chunk[0] = energy_chunk[0] + energy_chunk[32 * i];
772   }
773 }
774
775 //first 32 threads writes results in global memory
776 __syncthreads();
777 if (threadIdx.x < 32){
```

```
  *(energies_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*
    gridDim.y) + threadIdx.x) = getNumber(energy_chunk[0], threadIdx.x);
}

}

__global__ void totalMagnetizationBlockWide2D(unsigned int* spins_d, int*
    magnetization_block_wide, int L1, int L2){

//add up energies
multint17_t local_magnetization;

coordinates2D_t own_position = indexToCoordinates2DEven(threadIdx.x);
coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
unsigned int own_multispin = getMultispin2D(spins_d, own_position, L1, L2);
unsigned int odd_multispin = getMultispin2D(spins_d, odd_position, L1, L2);


unsigned int sign = ~own_multispin;
multint17_t current_magnetization = multint17_0;

current_magnetization.bit[0] = ~0;

for (int i = 1; i < 16; i++){
  current_magnetization.bit[i] = sign;
}
current_magnetization.sign = sign;
local_magnetization = current_magnetization;
current_magnetization = multint17_0;

sign = ~odd_multispin;
current_magnetization.bit[0] = ~0;
for (int i = 1; i < 16; i++){
  current_magnetization.bit[i] = sign;
}
current_magnetization.sign = sign;

local_magnetization = local_magnetization + current_magnetization;

extern __shared__ int chunk[];
multint17_t* magn_chunk = (multint17_t*)&chunk[0];

// first 512 threads loads energies in chunk
if (threadIdx.x < 512){
  magn_chunk[threadIdx.x] = local_magnetization;
}

//next 512 threads add there own energies.
__syncthreads();
if (threadIdx.x >= 512){
  magn_chunk[threadIdx.x - 512] = magn_chunk[threadIdx.x - 512] + local_magnetization;
}

//add energies warp wide
__syncthreads();
if (threadIdx.x < 512){
  if (threadIdx.x % 32 == 0){
    for (int i = 1; i < 32; i++){
      magn_chunk[threadIdx.x] = magn_chunk[threadIdx.x] + magn_chunk[threadIdx.x + i];
    }
  }
}

//add energies block wide
__syncthreads();
if (threadIdx.x == 0){
```

```
842   for (int i = 1; i < 16; i++){
843     magn_chunk[0] = magn_chunk[0] + magn_chunk[32 * i];
844   }
845 }
846
847 //first 32 threads writes results in global memory
848 __syncthreads();
849 if (threadIdx.x < 32){
850   *(magnetization_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
      gridDim.x*gridDim.y) + threadIdx.x) = getNumber(magn_chunk[0], threadIdx.x);
851 }
852 }
853
854
855 __global__ void shiftDemonsBlockWide2D(unsigned int* demons_d, unsigned int random_offset
      , int L1, int L2){
856
857 coordinates2D_t even_position = indexToCoordinates2DEven(threadIdx.x);
858 coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
859
860 __shared__ demons3_t demon_chunk[2048];
861
862 //load even demons
863 demon_chunk[threadIdx.x] = getDemons2D(demons_d, even_position, L1, L2);
864
865 //load odd demons
866 demon_chunk[threadIdx.x + 1024] = getDemons2D(demons_d, odd_position, L1, L2);
867
868 __syncthreads();
869
870 int new_index_even = (threadIdx.x + random_offset) % 2048;
871 int new_index_odd = (threadIdx.x + 1024 + random_offset) % 2048;
872
873 writeDemons2D(demon_chunk[new_index_even], demons_d, even_position, L1, L2);
874 writeDemons2D(demon_chunk[new_index_odd], demons_d, odd_position, L1, L2);
875 }
876
877
878 __global__ void shiftDemonsGridWide2D(unsigned int* demons_d, unsigned int random_offset,
       int L1, int L2){
879
880 unsigned int offset_x = (random_offset & 65535);
881
882 coordinates2D_t even_position = indexToCoordinates2DEven(threadIdx.x);
883 coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
884
885 coordinates2D_t block_origin;
886 block_origin.x = blockIdx.x * 64;
887 block_origin.y = blockIdx.y * 32;
888
889 coordinates2D_t swap_block_origin;
890 swap_block_origin.x = ((blockIdx.x + offset_x) % gridDim.x) * 64;
891 swap_block_origin.y = (blockIdx.y + gridDim.y) * 32;
892
893
894 coordinates2D_t swap_even_position;
895 swap_even_position.x = (even_position.x - block_origin.x) + swap_block_origin.x;
896 swap_even_position.y = (even_position.y - block_origin.y) + swap_block_origin.y;
897
898 coordinates2D_t swap_odd_position;
899 swap_odd_position.x = (odd_position.x - block_origin.x) + swap_block_origin.x;
900 swap_odd_position.y = (odd_position.y - block_origin.y) + swap_block_origin.y;
901
902 __shared__ demons3_t demon_chunk[2048];
903
904 /*
```

```
905 swap even demons3_t
906 */
907
908 demon_chunk[threadIdx.x] = getDemons2D(demons_d, even_position, L1, L2);
909 demon_chunk[threadIdx.x + 1024] = getDemons2D(demons_d, swap_even_position, L1, L2);
910
911 __syncthreads();
912
913 writeDemons2D(demon_chunk[threadIdx.x + 1024], demons_d, even_position, L1, L2);
914 writeDemons2D(demon_chunk[threadIdx.x], demons_d, swap_even_position, L1, L2);
915
916 __syncthreads();
917
918 /*
919 swap
920 odd demons3_t
921 */
922 demon_chunk[threadIdx.x] = getDemons2D(demons_d, odd_position, L1, L2);
923 demon_chunk[threadIdx.x + 1024] = getDemons2D(demons_d, swap_odd_position, L1, L2);
924
925 __syncthreads();
926
927 writeDemons2D(demon_chunk[threadIdx.x + 1024], demons_d, odd_position, L1, L2);
928 writeDemons2D(demon_chunk[threadIdx.x], demons_d, swap_odd_position, L1, L2);
929 }
930
931
932 __global__ void demonLayersOccupationBlockWide2D(unsigned int* demons_d, unsigned int*
        demon_block_attributes_d, int L1, int L2){
933
934 demons3_t own_demons;
935 demons3_t odd_demons;
936
937 coordinates2D_t own_position = indexToCoordinates2DEven(threadIdx.x);
938 coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
939
940 own_demons = getDemons2D(demons_d, own_position, L1, L2);
941 odd_demons = getDemons2D(demons_d, odd_position, L1, L2);
942
943 //add up occupations
944 unsigned int own_layer_occupation_low = 0; //first layer
945 unsigned int own_layer_occupation_mid = 0; // second layer
946 unsigned int own_layer_occupation_high = 0; //third layer
947
948 unsigned int demon_mask;
949 for (int i = 0; i < 32; i++){
950   demon_mask = (1 << i);
951
952   own_layer_occupation_low = own_layer_occupation_low + ((own_demons.ldemons & demon_mask
        ) >> i) + ((odd_demons.ldemons & demon_mask) >> i);
953   own_layer_occupation_mid = own_layer_occupation_mid + ((own_demons.mdemons & demon_mask
        ) >> i) + ((odd_demons.mdemons & demon_mask) >> i);
954   own_layer_occupation_high = own_layer_occupation_high + ((own_demons.hdemons &
        demon_mask) >> i) + ((odd_demons.hdemons & demon_mask) >> i);
955
956 }
957 __shared__ unsigned int occupation_chunk[3 * 1024];
958
959 occupation_chunk[3 * threadIdx.x] = own_layer_occupation_low;
960 occupation_chunk[3 * threadIdx.x + 1] = own_layer_occupation_mid;
961 occupation_chunk[3 * threadIdx.x + 2] = own_layer_occupation_high;
962
963 __syncthreads();
964
965 if (threadIdx.x % 32 == 0){
966   for (int i = 1; i < 32; i++){
```

```
967     occupation_chunk[3 * threadIdx.x] = occupation_chunk[3 * threadIdx.x] +
        occupation_chunk[3 * threadIdx.x + 3 * i];
968     occupation_chunk[3 * threadIdx.x + 1] = occupation_chunk[3 * threadIdx.x + 1] +
        occupation_chunk[3 * threadIdx.x + 3 * i + 1];
969     occupation_chunk[3 * threadIdx.x + 2] = occupation_chunk[3 * threadIdx.x + 2] +
        occupation_chunk[3 * threadIdx.x + 3 * i + 2];
970   }
971 }
972
973 //add energies block wide
974 __syncthreads();
975 if (threadIdx.x == 0){
976   for (int i = 1; i < 32; i++){
977     occupation_chunk[0] = occupation_chunk[0] + occupation_chunk[32 * 3 * i];
978     occupation_chunk[1] = occupation_chunk[1] + occupation_chunk[32 * 3 * i + 1];
979     occupation_chunk[2] = occupation_chunk[2] + occupation_chunk[32 * 3 * i + 2];
980   }
981   //write results in memory
982   *(demon_block_attributes_d + 4 * (blockIdx.x + blockIdx.y*gridDim.x)) =
        occupation_chunk[0];
983   *(demon_block_attributes_d + 4 * (blockIdx.x + blockIdx.y*gridDim.x) + 1) =
        occupation_chunk[1];
984   *(demon_block_attributes_d + 4 * (blockIdx.x + blockIdx.y*gridDim.x) + 2) =
        occupation_chunk[2];
985 }
986
987
988 }
989
990 __global__ void applyPatternRemove2D(unsigned int* demons_d, unsigned int*
        random_pattern_d, int layer, int L1, int L2){
991
992 coordinates2D_t own_position = indexToCoordinates2DEven(threadIdx.x);
993 coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
994
995 demons3_t own_demons = getDemons2D(demons_d, own_position, L1, L2);
996 demons3_t odd_demons = getDemons2D(demons_d, odd_position, L1, L2);
997
998 unsigned int random_pattern_own;
999 unsigned int random_pattern_odd;
1000
1001
1002 unsigned int layer_demons_own;
1003 unsigned int layer_demons_odd;
1004
1005 unsigned int applied_changes_own;
1006 unsigned int applied_changes_odd;
1007
1008 unsigned int own_change_count = 0;
1009
1010 __shared__ unsigned int block_wide_changes[1024];
1011
1012 if (layer == DEMON_LAYER_LOW){
1013
1014   layer_demons_own = own_demons.ldemons;
1015   layer_demons_odd = odd_demons.ldemons;
1016
1017 }
1018 else if (layer == DEMON_LAYER_MID){
1019
1020   layer_demons_own = own_demons.mdemons;
1021   layer_demons_odd = odd_demons.mdemons;
1022
1023 }
1024 else if (layer == DEMON_LAYER_HIGH){
1025
```

```
1026   layer_demons_own = own_demons.hdemons;
1027   layer_demons_odd = odd_demons.hdemons;
1028
1029 }
1030
1031 random_pattern_own = *(random_pattern_d + own_position.x + own_position.y*L1);
1032 random_pattern_odd = *(random_pattern_d + odd_position.x + odd_position.y*L1);
1033
1034 applied_changes_own = layer_demons_own;
1035 applied_changes_odd = layer_demons_odd;
1036
1037 //remove demons if possible
1038 layer_demons_own = layer_demons_own & (~random_pattern_own);
1039 layer_demons_odd = layer_demons_odd & (~random_pattern_odd);
1040
1041 /*
1042 bit 1: change applied (demon removed)
1043 bit 0: no change occurred
1044 */
1045 applied_changes_own = (applied_changes_own & random_pattern_own);
1046 applied_changes_odd = (applied_changes_odd & random_pattern_odd);
1047
1048 //compute number of changes
1049 unsigned int demon_mask;
1050 for (int i = 0; i < 32; i++){
1051   demon_mask = (1 << i);
1052   own_change_count = own_change_count + ((applied_changes_own & demon_mask) >> i) + ((
       applied_changes_odd & demon_mask) >> i);
1053 }
1054
1055 block_wide_changes[threadIdx.x] = own_change_count;
1056
1057 __syncthreads();
1058 //sum up changes warp wide
1059 if (threadIdx.x % 32 == 0){
1060
1061   for (int i = 1; i < 32; i++){
1062     block_wide_changes[threadIdx.x] = block_wide_changes[threadIdx.x] +
       block_wide_changes[threadIdx.x + i];
1063   }
1064
1065 }
1066 __syncthreads();
1067
1068 //thread 0 sums up rest and saves informations in random_pattern_d
1069 if (threadIdx.x == 0){
1070
1071   for (int i = 1; i < 32; i++){
1072     block_wide_changes[0] = block_wide_changes[0] + block_wide_changes[32 * i];
1073   }
1074   *(random_pattern_d + blockIdx.x * 64 + blockIdx.y * 32 * L1) = block_wide_changes[0];
1075 }
1076
1077 //write back new demons to memory
1078
1079 if (layer == DEMON_LAYER_LOW){
1080
1081   own_demons.ldemons = layer_demons_own;
1082   odd_demons.ldemons = layer_demons_odd;
1083
1084 }
1085 else if (layer == DEMON_LAYER_MID){
1086
1087   own_demons.mdemons = layer_demons_own;
1088   odd_demons.mdemons = layer_demons_odd;
1089
```

```
1090 }
1091 else if (layer == DEMON_LAYER_HIGH){
1092
1093   own_demons.hdemons = layer_demons_own;
1094   odd_demons.hdemons = layer_demons_odd;
1095 }
1096
1097 writeDemons2D(own_demons, demons_d, own_position, L1, L2);
1098 writeDemons2D(odd_demons, demons_d, odd_position, L1, L2);
1099 }
1100
1101 __global__ void applyPatternSet2D(unsigned int* demons_d, unsigned int* random_pattern_d,
     int layer, int L1, int L2){
1102
1103 coordinates2D_t own_position = indexToCoordinates2DEven(threadIdx.x);
1104 coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);
1105
1106 demons3_t own_demons = getDemons2D(demons_d, own_position, L1, L2);
1107 demons3_t odd_demons = getDemons2D(demons_d, odd_position, L1, L2);
1108
1109 unsigned int random_pattern_own;
1110 unsigned int random_pattern_odd;
1111
1112
1113 unsigned int layer_demons_own;
1114 unsigned int layer_demons_odd;
1115
1116 unsigned int applied_changes_own;
1117 unsigned int applied_changes_odd;
1118
1119 unsigned int own_change_count = 0;
1120
1121 __shared__ unsigned int block_wide_changes[1024];
1122
1123 if (layer == DEMON_LAYER_LOW){
1124
1125   layer_demons_own = own_demons.ldemons;
1126   layer_demons_odd = odd_demons.ldemons;
1127
1128 }
1129 else if (layer == DEMON_LAYER_MID){
1130
1131   layer_demons_own = own_demons.mdemons;
1132   layer_demons_odd = odd_demons.mdemons;
1133
1134 }
1135 else if (layer == DEMON_LAYER_HIGH){
1136
1137   layer_demons_own = own_demons.hdemons;
1138   layer_demons_odd = odd_demons.hdemons;
1139
1140 }
1141
1142 random_pattern_own = *(random_pattern_d + own_position.x + own_position.y*L1);
1143 random_pattern_odd = *(random_pattern_d + odd_position.x + odd_position.y*L1);
1144
1145 applied_changes_own = layer_demons_own;
1146 applied_changes_odd = layer_demons_odd;
1147
1148 //set demons if possible
1149 layer_demons_own = layer_demons_own | random_pattern_own;
1150 layer_demons_odd = layer_demons_odd | random_pattern_odd;
1151
1152 /*
1153 bit 1: change applied (demon set)
1154 bit 0: no change occurred
```

```
1155 */
1156 applied_changes_own = (applied_changes_own ^ random_pattern_own) & random_pattern_own;
1157 applied_changes_odd = (applied_changes_odd ^ random_pattern_odd) & random_pattern_odd;
1158
1159 //compute number of changes
1160 unsigned int demon_mask;
1161 for (int i = 0; i < 32; i++){
1162   demon_mask = (1 << i);
1163   own_change_count = own_change_count + ((applied_changes_own & demon_mask) >> i) + ((
       applied_changes_odd & demon_mask) >> i);
1164 }
1165
1166 block_wide_changes[threadIdx.x] = own_change_count;
1167
1168 __syncthreads();
1169 //sum up changes warp wide
1170 if (threadIdx.x % 32 == 0){
1171
1172   for (int i = 1; i < 32; i++){
1173     block_wide_changes[threadIdx.x] = block_wide_changes[threadIdx.x] +
       block_wide_changes[threadIdx.x + i];
1174   }
1175
1176 }
1177 __syncthreads();
1178
1179 //thread 0 sums up rest and saves informations in random_pattern_d
1180 if (threadIdx.x == 0){
1181
1182   for (int i = 1; i < 32; i++){
1183     block_wide_changes[0] = block_wide_changes[0] + block_wide_changes[32 * i];
1184   }
1185
1186   *(random_pattern_d + blockIdx.x * 64 + blockIdx.y * 32 * L1) = block_wide_changes[0];
1187 }
1188
1189 //write new demons back to memory
1190
1191 if (layer == DEMON_LAYER_LOW){
1192
1193   own_demons.ldemons = layer_demons_own;
1194   odd_demons.ldemons = layer_demons_odd;
1195
1196 }
1197 else if (layer == DEMON_LAYER_MID){
1198
1199   own_demons.mdemons = layer_demons_own;
1200   odd_demons.mdemons = layer_demons_odd;
1201
1202 }
1203 else if (layer == DEMON_LAYER_HIGH){
1204
1205   own_demons.hdemons = layer_demons_own;
1206   odd_demons.hdemons = layer_demons_odd;
1207 }
1208
1209 writeDemons2D(own_demons, demons_d, own_position, L1, L2);
1210 writeDemons2D(odd_demons, demons_d, odd_position, L1, L2);
1211 }
1212
1213 __global__ void updateDemons2D(unsigned int* demons_d, unsigned int* randoms, double
       set_probabilities[], int L1, int L2){
1214 demons3_t own_demons;
1215 demons3_t odd_demons;
1216
1217 coordinates2D_t own_position = indexToCoordinates2DEven(threadIdx.x);
```

```
coordinates2D_t odd_position = indexToCoordinates2DOdd(threadIdx.x);

own_demons = getDemons2D(demons_d, own_position, L1, L2);
odd_demons = getDemons2D(demons_d, odd_position, L1, L2);

int current_random = (int) *(randoms + 1024*(blockIdx.x + blockIdx.y*gridDim.x)  +
    threadIdx.x);
//printf("%i \n", current_random);
double set_prob[3];
set_prob[0] = set_probabilities[0];
set_prob[1] = set_probabilities[1];
set_prob[2] = set_probabilities[2];

for (int i = 0; i < 32; i++){
  //for low demons
  simpleRandomInteger0(current_random);
  if ((current_random / 2147483646.0) <= set_prob[0]){
    own_demons.ldemons = own_demons.ldemons | (1 << i);
  }
else{
  own_demons.ldemons = own_demons.ldemons & ~(1 << i);
}

simpleRandomInteger0(current_random);
if ((current_random / 2147483646.0) <= set_prob[0]){
  odd_demons.ldemons = odd_demons.ldemons | (1 << i);
}
else{
  odd_demons.ldemons = odd_demons.ldemons & ~(1 << i);
}

simpleRandomInteger0(current_random);
if ((current_random / 2147483646.0) <= set_prob[1]){
  own_demons.mdemons = own_demons.mdemons | (1 << i);
}
else{
  own_demons.mdemons = own_demons.mdemons & ~(1 << i);
}

simpleRandomInteger0(current_random);
if ((current_random / 2147483646.0) <= set_prob[1]){
  odd_demons.mdemons = odd_demons.mdemons | (1 << i);
}
else{
  odd_demons.mdemons = odd_demons.mdemons & ~(1 << i);
}

simpleRandomInteger0(current_random);
if ((current_random / 2147483646.0) <= set_prob[2]){
  own_demons.hdemons = own_demons.hdemons | (1 << i);
}
else{
  own_demons.hdemons = own_demons.hdemons & ~(1 << i);
}

simpleRandomInteger0(current_random);
if ((current_random / 2147483646.0) <= set_prob[2]){
  odd_demons.hdemons = odd_demons.hdemons | (1 << i);
}
else{
  odd_demons.hdemons = odd_demons.hdemons & ~(1 << i);
}

}
//write back randoms
```

```
*(randoms + 1024*(blockIdx.x + blockIdx.y*gridDim.x)  + threadIdx.x) = (unsigned int)
    current_random;

//write back demons

writeDemons2D(own_demons, demons_d, own_position, L1, L2);
writeDemons2D(odd_demons, demons_d, odd_position, L1, L2);
}
```

8.2.2.7 routines3D.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "routines3D.hpp"
#include <stdio.h>
__device__ inline coordinates3D_t indexToCoordinates3DEven(int index){
coordinates3D_t position;

//origin of the 32x32 sub lattice.
int pointX = blockIdx.x * 64;
int pointY = blockIdx.y * 32;
/*
calculate position in sub lattice 32x32
i.e.: transform index to coordinates of the full lattice
*/
position.z = blockIdx.z;
position.y = index / 32 + pointY;
position.x = index % 32;


//expand coordinates to all even points in the lattice.

//first coordinate in x-direction (0,y,z) is even
if ((pointX + position.y + position.z) % 2 == 0){
  position.x = 2 * position.x + pointX;
}
//first coordinate in x-direction (0,y,z) is odd
else if ((pointX + position.y + position.z) % 2 != 0){
  position.x = 2 * position.x + 1 + pointX;
}

return position;
}


__device__ inline coordinates3D_t indexToCoordinates3DOdd(int index){
coordinates3D_t position;

int pointX = blockIdx.x * 64;
int pointY = blockIdx.y * 32;

//calculate position in sub lattice 32x32
position.z = blockIdx.z;
position.y = index / 32 + pointY;
position.x = index % 32;

//expand coordinates to all even points in the lattice.

//first coordinate in x-direction (0,y,z) is odd
if ((pointX + position.y + position.z) % 2 != 0){
  position.x = 2 * position.x + pointX;
```

```
52 }
53 //first coordinate in x-direction (0,y,z) is even
54 else if ((pointX + position.y + position.z) % 2 == 0){
55   position.x = 2 * position.x + 1 + pointX;
56 }
57
58 return position;
59 }
60
61 __device__ inline coordinates3D_t getNeighbourPosition3D(coordinates3D_t own_position,
       int neighbour, int L1, int L2, int L3) {
62
63
64 int x = own_position.x;
65 int y = own_position.y;
66 int z = own_position.z;
67
68 switch (neighbour){
69
70 case _FRONT_:
71 //regard boundary conditions
72 if (x + 1 > L1 - 1){
73   own_position.x = 0;
74 }
75 else{
76   own_position.x = x + 1;
77 }
78 break;
79
80 case _BACK_:
81 //regard boundary conditions
82 if (x - 1 < 0){
83   own_position.x = L1 - 1;
84 }
85 else{
86   own_position.x = x - 1;
87 }
88 break;
89
90 case _LEFT_:
91 //regard boundary conditions
92 if (y + 1 > L2 - 1){
93   own_position.y = 0;
94 }
95 else{
96   own_position.y = y + 1;
97 }
98 break;
99
100 case _RIGHT_:
101 //regard boundary conditions
102 if (y - 1 < 0){
103   own_position.y = L2 - 1;
104 }
105 else{
106   own_position.y = y - 1;
107 }
108 break;
109
110 case _UPPER_:
111 //regard boundary conditions
112 if (z + 1 > L3 - 1){
113   own_position.z = 0;
114 }
115 else{
116   own_position.z = z + 1;
```

```
117 }
118 break;
119
120 case _LOWER_:
121 //regard boundary conditions
122 if (z - 1 < 0){
123   own_position.z = L3 - 1;
124 }
125 else{
126   own_position.z = z - 1;
127 }
128 break;
129 }
130
131 return own_position;
132 }
133
134
135 __device__ inline unsigned int getMultispin3D(unsigned int* spins_d, const
        coordinates3D_t& position, int L1, int L2, int L3){
136
137 int x = position.x;
138 int y = position.y;
139 int z = position.z;
140
141 int sub_L1 = L1 / 2;
142
143 //coordinates even?
144 if ((x + y + z) % 2 == 0){
145   //reverse transformation in indexToCoordinates3DEven
146   if ((y + z) % 2 == 0){
147     x = x / 2;
148   }
149   else{
150     x = (x - 1) / 2;
151   }
152 return *(spins_d + x + y*sub_L1 + z*sub_L1*L2);
153 }
154 else{
155   //reverse transformation in indexToCoordinates3DOdd
156   if ((y + z) % 2 != 0){
157     x = x / 2;
158   }
159   else{
160     x = (x - 1) / 2;
161   }
162 return *(spins_d + x + y*sub_L1 + z*sub_L1*L2 + sub_L1 * L2 * L3);
163 }
164
165 }
166
167
168 __device__ inline void writeMultispin3D(unsigned int multispin, unsigned int* spins_d,
        const coordinates3D_t& position, int L1, int L2, int L3){
169
170 int x = position.x;
171 int y = position.y;
172 int z = position.z;
173
174 int sub_L1 = L1 / 2;
175
176 //coordinates even?
177 if ((x + y + z) % 2 == 0){
178   //reverse transformation in indexToCoordinates3DEven
179   if ((y + z) % 2 == 0){
180     x = x / 2;
```

```
181   }
182   else{
183     x = (x - 1) / 2;
184   }
185   *(spins_d + x + y*sub_L1 + z*sub_L1*L2) = multispin;
186 }
187 else{
188   //reverse transformation in indexToCoordinates3DOdd
189   if ((y + z) % 2 != 0){
190     x = x / 2;
191   }
192   else{
193     x = (x - 1) / 2;
194   }
195   *(spins_d + x + y*sub_L1 + z*sub_L1*L2 + sub_L1 * L2 * L3) = multispin;
196 }
197 }
198
199
200 __device__ inline demons3_t getDemons3D(unsigned int* demons_d, const coordinates3D_t&
      position, int L1, int L2, int L3){
201 int x = position.x;
202 int y = position.y;
203 int z = position.z;
204
205 int sub_L1 = L1 / 2;
206
207 demons3_t demons;
208
209 //coordinates even?
210 if ((x + y + z) % 2 == 0){
211   //reverse transformation in indexToCoordinates3DEven
212   if ((y + z) % 2 == 0){
213     x = x / 2;
214   }
215   else{
216     x = (x - 1) / 2;
217   }
218   demons.hdemons = *(demons_d + x + y*sub_L1 + z*sub_L1*L2);
219   demons.mdemons = *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + sub_L1*L2*L3);
220   demons.ldemons = *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 2 * sub_L1*L2*L3);
221 }
222 else{
223   //reverse transformation in indexToCoordinates3DOdd
224   if ((y + z) % 2 != 0){
225     x = x / 2;
226   }
227   else{
228     x = (x - 1) / 2;
229   }
230   demons.hdemons = *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 3 * sub_L1*L2*L3);
231   demons.mdemons = *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 4 * sub_L1*L2*L3);
232   demons.ldemons = *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 5 * sub_L1*L2*L3);
233 }
234
235 return demons;
236 }
237
238 __device__ inline void writeDemons3D(demons3_t demons, unsigned int* demons_d, const
      coordinates3D_t& position, int L1, int L2, int L3){
239 int x = position.x;
240 int y = position.y;
241 int z = position.z;
242
243 int sub_L1 = L1 / 2;
244
```

```
245 //coordinates even?
246 if ((x + y + z) % 2 == 0){
247   //reverse transformation in indexToCoordinates3DEven
248   if ((y + z) % 2 == 0){
249     x = x / 2;
250   }
251   else{
252     x = (x - 1) / 2;
253   }
254   *(demons_d + x + y*sub_L1 + z*sub_L1*L2) = demons.hdemons;
255   *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + sub_L1*L2*L3) = demons.mdemons;
256   *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 2 * sub_L1*L2*L3) = demons.ldemons;
257 }
258 else{
259   //reverse transformation in indexToCoordinates3DOdd
260   if ((y + z) % 2 != 0){
261     x = x / 2;
262   }
263   else{
264     x = (x - 1) / 2;
265   }
266   *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 3 * sub_L1*L2*L3) = demons.hdemons;
267   *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 4 * sub_L1*L2*L3) = demons.mdemons;
268   *(demons_d + x + y*sub_L1 + z*sub_L1*L2 + 5 * sub_L1*L2*L3) = demons.ldemons;
269
270 }
271 }
272
273 __device__ inline multint7_t localSpinEnergy3D(unsigned int own_multispin, unsigned int*
      neighbour_multispins){
274
275 multint7_t sum_energy = multint7_0;
276 multint7_t current_edge_energy;
277 unsigned int parallel;
278
279 for (int i = 0; i < 6; i++){
280
281   //bit 0: spins antiparallel; bit 1: spins parallel
282   parallel = ~(neighbour_multispins[i] ^ own_multispin);
283
284   //if antiparallel set current_edge_energy = 1, else -1
285   current_edge_energy.bit[0] = ~0;
286   for (int i = 1; i < 6; i++){
287     current_edge_energy.bit[i] = parallel;
288   }
289   current_edge_energy.sign = parallel;
290
291   //add edge energy to total energy.
292   sum_energy = sum_energy + current_edge_energy;
293
294 }
295
296 return sum_energy;
297 }
298
299
300
301 __device__ inline void getEnvironment3DEven(unsigned int* spins_d, unsigned int* demons_d
      , coordinates3D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
      unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2, int L3)
      {
302
303
304 own_position = indexToCoordinates3DEven(threadIdx.x);
305
306 coordinates3D_t block_origin;
```

```
307 block_origin.x = blockIdx.x * 64;
308 block_origin.y = blockIdx.y * 32;
309 block_origin.z = blockIdx.z;
310
311 int x = own_position.x;
312 int y = own_position.y;
313 int z = own_position.z;
314
315
316
317 coordinates3D_t neighbour_position;
318
319 own_multispin = getMultispin3D(spins_d, own_position, L1, L2, L3);
320 own_demons = getDemons3D(demons_d, own_position, L1, L2, L3);
321
322 /*
323 neighbour_multispins[0]: front multispin
324 neighbour_multispins[1]: back multispin
325 neighbour_multispins[2]: left multispin
326 neighbour_multispins[3]: right multispin
327 neighbour_multispins[4]: upper multispin
328 neighbour_multispins[5]: lower multispin
329 */
330
331 //load upper and lower mutlispins
332 neighbour_position = getNeighbourPosition3D(own_position, _UPPER_, L1, L2, L3);
333 neighbour_multispins[4] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
334
335 neighbour_position = getNeighbourPosition3D(own_position, _LOWER_, L1, L2, L3);
336 neighbour_multispins[5] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
337
338
339 //load neigbhour multispins on the x-y-plane in spin_chunk
340 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
341 spin_chunk[threadIdx.x] = getMultispin3D(spins_d, odd_position, L1, L2, L3);
342 __syncthreads(); //IMPORTANT, prevents race condition in shared memory
343
344
345 /*
346 load front and back multispins to local variables.
347 */
348 //is the first point in the row (block.origin.x, y, z) even ?
349 if ((block_origin.x + y + z) % 2 == 0){
350
351   //write front multispin
352   neighbour_multispins[0] = spin_chunk[threadIdx.x];
353   if (x > block_origin.x){
354     //write back multispin
355     neighbour_multispins[1] = spin_chunk[threadIdx.x - 1];
356   }
357   else{
358     neighbour_position = getNeighbourPosition3D(own_position, _BACK_, L1, L2, L3);
359     //write back multispin
360     neighbour_multispins[1] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
361   }
362 }
363 else{
364
365   //write back multispin
366   neighbour_multispins[1] = spin_chunk[threadIdx.x];
367
368   if (x < block_origin.x + 63){
369
370     //write front multispin
371     neighbour_multispins[0] = spin_chunk[threadIdx.x + 1];
372   }
```

```
373   else{
374     neighbour_position = getNeighbourPosition3D(own_position, _FRONT_, L1, L2, L3);
375     //write front multispin
376     neighbour_multispins[0] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
377   }
378 }
379
380 /*
381 load left and right multispins to local variables.
382 */
383 //is the first point in the column (x, block_origin.y, z) even?
384 if ((x + block_origin.y + z) % 2 == 0){
385
386   //write left multispin
387   neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];
388
389   if (y > block_origin.y){
390     //write right multispin
391     neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];
392   }
393   else{
394     neighbour_position = getNeighbourPosition3D(own_position, _RIGHT_, L1, L2, L3);
395     //write right multispin
396     neighbour_multispins[3] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
397   }
398 }
399 else{
400
401   //write right multispin
402   neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];
403
404   if (y < block_origin.y + 31){
405     //write left multispin
406     neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];
407   }
408   else{
409   neighbour_position = getNeighbourPosition3D(own_position, _LEFT_, L1, L2, L3);
410   //write left multispin
411   neighbour_multispins[2] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
412   }
413 }
414 }
415
416
417
418 __device__ inline void getEnvironment3DOdd(unsigned int* spins_d, unsigned int* demons_d,
      coordinates3D_t& own_position, unsigned int& own_multispin, demons3_t& own_demons,
     unsigned int* neighbour_multispins, unsigned int* spin_chunk, int L1, int L2, int L3)
     {
419
420
421
422 own_position = indexToCoordinates3DOdd(threadIdx.x);
423
424 coordinates3D_t block_origin;
425 block_origin.x = blockIdx.x * 64;
426 block_origin.y = blockIdx.y * 32;
427 block_origin.z = blockIdx.z;
428
429 int x = own_position.x;
430 int y = own_position.y;
431 int z = own_position.z;
432
433
434
435 coordinates3D_t neighbour_position;
```

```
436
437 own_multispin = getMultispin3D(spins_d, own_position, L1, L2, L3);
438 own_demons = getDemons3D(demons_d, own_position, L1, L2, L3);
439
440 /*
441 neighbour_multispins[0]: front multispin
442 neighbour_multispins[1]: back multispin
443 neighbour_multispins[2]: left multispin
444 neighbour_multispins[3]: right multispin
445 neighbour_multispins[4]: upper multispin
446 neighbour_multispins[5]: lower multispin
447 */
448
449 //load upper and lower mutlispins
450 neighbour_position = getNeighbourPosition3D(own_position, _UPPER_, L1, L2, L3);
451 neighbour_multispins[4] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
452
453 neighbour_position = getNeighbourPosition3D(own_position, _LOWER_, L1, L2, L3);
454 neighbour_multispins[5] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
455
456
457 //load neigbhour multispins on the x-y-plane in spin_chunk
458 coordinates3D_t even_position = indexToCoordinates3DEven(threadIdx.x);
459 spin_chunk[threadIdx.x] = getMultispin3D(spins_d, even_position, L1, L2, L3);
460 __syncthreads(); //IMPORTANT, prevents race condition in shared memory
461
462
463 /*
464 load front and back multispins to local variables.
465 */
466 //is the first point in the row (block.origin.x, y, z) odd ?
467 if ((block_origin.x + y + z) % 2 != 0){
468
469   //write front multispin
470   neighbour_multispins[0] = spin_chunk[threadIdx.x];
471   if (x > block_origin.x){
472     //write back multispin
473     neighbour_multispins[1] = spin_chunk[threadIdx.x - 1];
474   }
475   else{
476     neighbour_position = getNeighbourPosition3D(own_position, _BACK_, L1, L2, L3);
477     //write back multispin
478     neighbour_multispins[1] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
479   }
480 }
481 else{
482
483   //write back multispin
484   neighbour_multispins[1] = spin_chunk[threadIdx.x];
485
486   if (x < block_origin.x + 63){
487
488     //write front multispin
489     neighbour_multispins[0] = spin_chunk[threadIdx.x + 1];
490   }
491   else{
492     neighbour_position = getNeighbourPosition3D(own_position, _FRONT_, L1, L2, L3);
493     //write front multispin
494     neighbour_multispins[0] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
495   }
496 }
497
498 /*
499 load left and right multispins to local variables.
500 */
501 //is the first point in the column (x, block_origin.y, z) even?
```

```
502 if ((x + block_origin.y + z) % 2 != 0){
503
504   //write left multispin
505   neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];
506
507   if (y > block_origin.y){
508     //write right multispin
509     neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];
510   }
511   else{
512     neighbour_position = getNeighbourPosition3D(own_position, _RIGHT_, L1, L2, L3);
513     //write right multispin
514     neighbour_multispins[3] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
515   }
516 }
517 else{
518
519   //write right multispin
520   neighbour_multispins[3] = spin_chunk[threadIdx.x - 32];
521
522   if (y < block_origin.y + 31){
523     //write left multispin
524     neighbour_multispins[2] = spin_chunk[threadIdx.x + 32];
525   }
526   else{
527     neighbour_position = getNeighbourPosition3D(own_position, _LEFT_, L1, L2, L3);
528     //write left multispin
529     neighbour_multispins[2] = getMultispin3D(spins_d, neighbour_position, L1, L2, L3);
530   }
531 }
532 }
533
534
535 __global__ void metropolis3DPeriodicEven(unsigned int* spins_d, unsigned int* demons_d,
        int L1, int L2, int L3){
536
537 coordinates3D_t own_position;
538 unsigned int own_multispin;
539 demons3_t own_demons;
540 unsigned int neighbour_multispins[6];
541
542 __shared__ unsigned int spin_chunk[1024];
543
544 getEnvironment3DEven(spins_d, demons_d, own_position, own_multispin, own_demons,
        neighbour_multispins, spin_chunk, L1, L2, L3);
545
546 /*
547 compute if spin change is possible.
548 */
549 multint7_t sum_energy;
550 multint7_t energy_change;
551 multint7_t demon_energy;
552
553 sum_energy = localSpinEnergy3D(own_multispin, neighbour_multispins);
554 demon_energy = demonEnergy7(own_demons);
555
556 energy_change = sum_energy + sum_energy;
557 //demon energy after spin flip
558 demon_energy = demon_energy + energy_change;
559
560 unsigned int flip_decision;
561
562 // if demon_energy is negative, no flip is allowed, bit is set to 0;
563 flip_decision = ~(demon_energy.sign);
564
565 //if demon energy is bigger than 16+8+4 = 28 (=0011100) no flip allowed.
```

```
flip_decision = flip_decision & ~(demon_energy.bit[5]);
flip_decision = flip_decision & (~(demon_energy.bit[4] & demon_energy.bit[3] &
    demon_energy.bit[2]) | (~(demon_energy.bit[1]) & ~(demon_energy.bit[0])));

//flip spins
own_multispin = own_multispin ^ flip_decision;


//cast new demon_energy to demons3_t if spin changed.
own_demons.ldemons = ((own_demons.ldemons ^ demon_energy.bit[2]) & flip_decision) ^
    own_demons.ldemons;
own_demons.mdemons = ((own_demons.mdemons ^ demon_energy.bit[3]) & flip_decision) ^
    own_demons.mdemons;
own_demons.hdemons = ((own_demons.hdemons ^ demon_energy.bit[4]) & flip_decision) ^
    own_demons.hdemons;


writeMultispin3D(own_multispin, spins_d, own_position, L1, L2, L3);
writeDemons3D(own_demons, demons_d, own_position, L1, L2, L3);
}



__global__ void metropolis3DPeriodicOdd(unsigned int* spins_d, unsigned int* demons_d,
    int L1, int L2, int L3){


coordinates3D_t own_position;
unsigned int own_multispin;
demons3_t own_demons;
unsigned int neighbour_multispins[6];

__shared__ unsigned int spin_chunk[1024];

getEnvironment3DOdd(spins_d, demons_d, own_position, own_multispin, own_demons,
    neighbour_multispins, spin_chunk, L1, L2, L3);

/*
compute if spin change is possible.
*/
multint7_t sum_energy;
multint7_t energy_change;
multint7_t demon_energy;

sum_energy = localSpinEnergy3D(own_multispin, neighbour_multispins);
demon_energy = demonEnergy7(own_demons);

energy_change = sum_energy + sum_energy;
//demon energy after spin flip
demon_energy = demon_energy + energy_change;

unsigned int flip_decision;

// if demon_energy is negative, no flip is allowed, bit is set to 0;
flip_decision = ~(demon_energy.sign);

//if demon energy is bigger than 16+8+4 = 28 (=0011100) no flip allowed.
flip_decision = flip_decision & ~(demon_energy.bit[5]);
flip_decision = flip_decision & (~(demon_energy.bit[4] & demon_energy.bit[3] &
    demon_energy.bit[2]) | (~(demon_energy.bit[1]) & ~(demon_energy.bit[0])));

//flip spins
own_multispin = own_multispin ^ flip_decision;


//cast new demon_energy to demons3_t if spin changed.
```

```
own_demons.ldemons = ((own_demons.ldemons ^ demon_energy.bit[2]) & flip_decision) ^
    own_demons.ldemons;
own_demons.mdemons = ((own_demons.mdemons ^ demon_energy.bit[3]) & flip_decision) ^
    own_demons.mdemons;
own_demons.hdemons = ((own_demons.hdemons ^ demon_energy.bit[4]) & flip_decision) ^
    own_demons.hdemons;


writeMultispin3D(own_multispin, spins_d, own_position, L1, L2, L3);
writeDemons3D(own_demons, demons_d, own_position, L1, L2, L3);

}


__global__ void totalEnergyBlockWide3DPeriodic(unsigned int* spins_d, unsigned int*
    demons_d, int* energies_block_wide, int L1, int L2, int L3){

coordinates3D_t own_position;
unsigned int own_multispin;
demons3_t own_demons;

unsigned int neighbour_multispins[6];


extern  __shared__ int chunk[];
unsigned int* spin_chunk = (unsigned int*)&chunk[0];

getEnvironment3DEven(spins_d, demons_d, own_position, own_multispin, own_demons,
    neighbour_multispins, spin_chunk, L1, L2, L3);

__syncthreads();

demons3_t odd_demons;


//load neigbhour demons on the x-y-plane in demon_chunk
coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);

odd_demons = getDemons3D(demons_d, odd_position, L1, L2, L3);

//add up energies
multint17_t sum_energy;
multint17_t demon_energy;
multint17_t demon_energy_odd;

sum_energy = castMultint(localSpinEnergy3D(own_multispin, neighbour_multispins));

demon_energy = demonEnergy17(own_demons);
demon_energy_odd = demonEnergy17(odd_demons);
demon_energy = demon_energy + demon_energy_odd;

sum_energy = sum_energy + demon_energy;

multint17_t* energy_chunk = (multint17_t*)&chunk[0];

// first 512 threads loads energies in chunk
if (threadIdx.x < 512){
  energy_chunk[threadIdx.x] = sum_energy;
}

//next 512 threads add there own energies.
__syncthreads();
if (threadIdx.x >= 512){
  energy_chunk[threadIdx.x - 512] = energy_chunk[threadIdx.x - 512] + sum_energy;
}

```

```
686 //add energies warp wide
687 __syncthreads();
688 if (threadIdx.x < 512){
689   if (threadIdx.x % 32 == 0){
690     for (int i = 1; i < 32; i++){
691       energy_chunk[threadIdx.x] = energy_chunk[threadIdx.x] + energy_chunk[threadIdx.x +
      i];
692     }
693   }
694 }
695
696 //add energies block wide
697 __syncthreads();
698 if (threadIdx.x == 0){
699   for (int i = 1; i < 16; i++){
700     energy_chunk[0] = energy_chunk[0] + energy_chunk[32 * i];
701   }
702 }
703
704 //first 32 threads writes results in global memory
705 __syncthreads();
706 if (threadIdx.x < 32){
707   *(energies_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*
      gridDim.y) + threadIdx.x) = getNumber(energy_chunk[0], threadIdx.x);
708 }
709
710 }
711
712 __global__ void totalSpinEnergyBlockWide3DPeriodic(unsigned int* spins_d, unsigned int*
      demons_d, int* energies_block_wide, int L1, int L2, int L3){
713
714 coordinates3D_t own_position;
715 unsigned int own_multispin;
716 demons3_t own_demons;
717
718 unsigned int neighbour_multispins[6];
719
720
721 extern  __shared__ int chunk[];
722 unsigned int* spin_chunk = (unsigned int*)&chunk[0];
723
724 getEnvironment3DEven(spins_d, demons_d, own_position, own_multispin, own_demons,
      neighbour_multispins, spin_chunk, L1, L2, L3);
725
726 __syncthreads();
727
728
729 //add up energies
730 multint17_t sum_energy;
731
732 sum_energy = castMultint(localSpinEnergy3D(own_multispin, neighbour_multispins));
733
734 multint17_t* energy_chunk = (multint17_t*)&chunk[0];
735
736 // first 512 threads loads energies in chunk
737 if (threadIdx.x < 512){
738   energy_chunk[threadIdx.x] = sum_energy;
739 }
740
741 //next 512 threads add there own energies.
742 __syncthreads();
743 if (threadIdx.x >= 512){
744   energy_chunk[threadIdx.x - 512] = energy_chunk[threadIdx.x - 512] + sum_energy;
745 }
746
747 //add energies warp wide
```

```
748 __syncthreads();
749 if (threadIdx.x < 512){
750   if (threadIdx.x % 32 == 0){
751     for (int i = 1; i < 32; i++){
752       energy_chunk[threadIdx.x] = energy_chunk[threadIdx.x] + energy_chunk[threadIdx.x +
      i];
753     }
754   }
755 }
756
757 //add energies block wide
758 __syncthreads();
759 if (threadIdx.x == 0){
760   for (int i = 1; i < 16; i++){
761     energy_chunk[0] = energy_chunk[0] + energy_chunk[32 * i];
762   }
763 }
764
765 //first 32 threads writes results in global memory
766 __syncthreads();
767 if (threadIdx.x < 32){
768   *(energies_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*
      gridDim.y) + threadIdx.x) = getNumber(energy_chunk[0], threadIdx.x);
769 }
770
771 }
772
773 __global__ void totalDemonEnergyBlockWide3D(unsigned int* demons_d, int*
      energies_block_wide, int L1, int L2, int L3){
774
775
776 demons3_t own_demons;
777 demons3_t odd_demons;
778
779 coordinates3D_t own_position = indexToCoordinates3DEven(threadIdx.x);
780 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
781
782 own_demons = getDemons3D(demons_d, own_position, L1, L2, L3);
783 odd_demons = getDemons3D(demons_d, odd_position, L1, L2, L3);
784
785 //add up energies
786 multint17_t sum_energy;
787 multint17_t sum_energy_odd;
788 sum_energy = demonEnergy17(own_demons);
789 sum_energy_odd = demonEnergy17(odd_demons);
790 sum_energy = sum_energy + sum_energy_odd;
791
792
793 extern __shared__ int chunk[];
794 multint17_t* energy_chunk = (multint17_t*)&chunk[0];
795
796 // first 512 threads loads energies in chunk
797 if (threadIdx.x < 512){
798   energy_chunk[threadIdx.x] = sum_energy;
799 }
800
801 //next 512 threads add there own energies.
802 __syncthreads();
803 if (threadIdx.x >= 512){
804   energy_chunk[threadIdx.x - 512] = energy_chunk[threadIdx.x - 512] + sum_energy;
805 }
806
807 //add energies warp wide
808 __syncthreads();
809 if (threadIdx.x < 512){
810   if (threadIdx.x % 32 == 0){
```

```
811     for (int i = 1; i < 32; i++){
812     energy_chunk[threadIdx.x] = energy_chunk[threadIdx.x] + energy_chunk[threadIdx.x + i
        ];
813     }
814   }
815 }
816
817 //add energies block wide
818 __syncthreads();
819 if (threadIdx.x == 0){
820   for (int i = 1; i < 16; i++){
821     energy_chunk[0] = energy_chunk[0] + energy_chunk[32 * i];
822   }
823 }
824
825 //first 32 threads writes results in global memory
826 __syncthreads();
827 if (threadIdx.x < 32){
828   *(energies_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*
        gridDim.y) + threadIdx.x) = getNumber(energy_chunk[0], threadIdx.x);
829 }
830
831 }
832
833 __global__ void totalMagnetizationBlockWide3D(unsigned int* spins_d, int*
        magnetization_block_wide, int L1, int L2, int L3){
834
835
836 //add up energies
837 multint17_t local_magnetization;
838
839 coordinates3D_t own_position = indexToCoordinates3DEven(threadIdx.x);
840 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
841 unsigned int own_multispin = getMultispin3D(spins_d, own_position, L1, L2, L3);
842 unsigned int odd_multispin = getMultispin3D(spins_d, odd_position, L1, L2, L3);
843
844
845 unsigned int sign = ~own_multispin;
846 multint17_t current_magnetization = multint17_0;
847
848 current_magnetization.bit[0] = ~0;
849
850 for (int i = 1; i < 16; i++){
851   current_magnetization.bit[i] = sign;
852 }
853 current_magnetization.sign = sign;
854 local_magnetization = current_magnetization;
855 current_magnetization = multint17_0;
856
857 sign = ~odd_multispin;
858 current_magnetization.bit[0] = ~0;
859 for (int i = 1; i < 16; i++){
860   current_magnetization.bit[i] = sign;
861 }
862 current_magnetization.sign = sign;
863
864 local_magnetization = local_magnetization + current_magnetization;
865
866 extern __shared__ int chunk[];
867 multint17_t* magn_chunk = (multint17_t*)&chunk[0];
868
869 // first 512 threads loads energies in chunk
870 if (threadIdx.x < 512){
871   magn_chunk[threadIdx.x] = local_magnetization;
872 }
873
```

```
874 //next 512 threads add there own energies.
875 __syncthreads();
876 if (threadIdx.x >= 512){
877   magn_chunk[threadIdx.x - 512] = magn_chunk[threadIdx.x - 512] + local_magnetization;
878 }
879
880 //add energies warp wide
881 __syncthreads();
882 if (threadIdx.x < 512){
883   if (threadIdx.x % 32 == 0){
884     for (int i = 1; i < 32; i++){
885       magn_chunk[threadIdx.x] = magn_chunk[threadIdx.x] + magn_chunk[threadIdx.x + i];
886     }
887   }
888 }
889
890 //add energies block wide
891 __syncthreads();
892 if (threadIdx.x == 0){
893   for (int i = 1; i < 16; i++){
894     magn_chunk[0] = magn_chunk[0] + magn_chunk[32 * i];
895   }
896 }
897
898 //first 32 threads writes results in global memory
899 __syncthreads();
900 if (threadIdx.x < 32){
901   *(magnetization_block_wide + 32 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
      gridDim.x*gridDim.y) + threadIdx.x) = getNumber(magn_chunk[0], threadIdx.x);
902 }
903 }
904
905 __global__ void shiftDemonsBlockWide3D(unsigned int* demons_d, unsigned int random_offset
      , int L1, int L2, int L3){
906
907 coordinates3D_t even_position = indexToCoordinates3DEven(threadIdx.x);
908 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
909
910 __shared__ demons3_t demon_chunk[2048];
911
912 //load even demons
913 demon_chunk[threadIdx.x] = getDemons3D(demons_d, even_position, L1, L2, L3);
914
915 //load odd demons
916 demon_chunk[threadIdx.x + 1024] = getDemons3D(demons_d, odd_position, L1, L2, L3);
917
918 __syncthreads();
919
920 int new_index_even = (threadIdx.x + random_offset) % 2048;
921 int new_index_odd = (threadIdx.x + 1024 + random_offset) % 2048;
922
923 writeDemons3D(demon_chunk[new_index_even], demons_d, even_position, L1, L2, L3);
924 writeDemons3D(demon_chunk[new_index_odd], demons_d, odd_position, L1, L2, L3);
925 }
926
927 __global__ void shiftDemonsGridWide3D(unsigned int* demons_d, unsigned int random_offset,
       int L1, int L2, int L3){
928
929 unsigned int offset_x = (random_offset & 65535);
930 unsigned int offset_z = (random_offset & ~65535) >> 16;
931
932 coordinates3D_t even_position = indexToCoordinates3DEven(threadIdx.x);
933 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
934
935 coordinates3D_t block_origin;
936 block_origin.x = blockIdx.x * 64;
```

```
937 block_origin.y = blockIdx.y * 32;
938 block_origin.z = blockIdx.z;
939
940 coordinates3D_t swap_block_origin;
941 swap_block_origin.x = ((blockIdx.x + offset_x) % gridDim.x) * 64;
942 swap_block_origin.y = (blockIdx.y + gridDim.y) * 32;
943 swap_block_origin.z = (blockIdx.z + offset_z) % gridDim.z;
944
945
946 coordinates3D_t swap_even_position;
947 swap_even_position.x = (even_position.x - block_origin.x) + swap_block_origin.x;
948 swap_even_position.y = (even_position.y - block_origin.y) + swap_block_origin.y;
949 swap_even_position.z = (even_position.z - block_origin.z) + swap_block_origin.z;
950
951 coordinates3D_t swap_odd_position;
952 swap_odd_position.x = (odd_position.x - block_origin.x) + swap_block_origin.x;
953 swap_odd_position.y = (odd_position.y - block_origin.y) + swap_block_origin.y;
954 swap_odd_position.z = (odd_position.z - block_origin.z) + swap_block_origin.z;
955
956 __shared__ demons3_t demon_chunk[2048];
957
958 /*
959 swap even demons3_t
960 */
961
962 demon_chunk[threadIdx.x] = getDemons3D(demons_d, even_position, L1, L2, L3);
963 demon_chunk[threadIdx.x + 1024] = getDemons3D(demons_d, swap_even_position, L1, L2, L3);
964
965 __syncthreads();
966
967 writeDemons3D(demon_chunk[threadIdx.x + 1024], demons_d, even_position, L1, L2, L3);
968 writeDemons3D(demon_chunk[threadIdx.x], demons_d, swap_even_position, L1, L2, L3);
969
970 __syncthreads();
971
972 /*
973 swap
974 odd demons3_t
975 */
976 demon_chunk[threadIdx.x] = getDemons3D(demons_d, odd_position, L1, L2, L3);
977 demon_chunk[threadIdx.x + 1024] = getDemons3D(demons_d, swap_odd_position, L1, L2, L3);
978
979 __syncthreads();
980
981 writeDemons3D(demon_chunk[threadIdx.x + 1024], demons_d, odd_position, L1, L2, L3);
982 writeDemons3D(demon_chunk[threadIdx.x], demons_d, swap_odd_position, L1, L2, L3);
983 }
984
985
986 __global__ void demonLayersOccupationBlockWide3D(unsigned int* demons_d, unsigned int*
        demon_block_attributes_d, int L1, int L2, int L3){
987
988 demons3_t own_demons;
989 demons3_t odd_demons;
990
991 coordinates3D_t own_position = indexToCoordinates3DEven(threadIdx.x);
992 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
993
994 own_demons = getDemons3D(demons_d, own_position, L1, L2, L3);
995 odd_demons = getDemons3D(demons_d, odd_position, L1, L2, L3);
996
997 //add up occupations
998 unsigned int own_layer_occupation_low = 0; //first layer
999 unsigned int own_layer_occupation_mid = 0; // second layer
1000 unsigned int own_layer_occupation_high = 0; //third layer
1001
```

```
1002 unsigned int demon_mask;
1003 for (int i = 0; i < 32; i++){
1004   demon_mask = (1 << i);
1005
1006   own_layer_occupation_low = own_layer_occupation_low + ((own_demons.ldemons & demon_mask
       ) >> i) + ((odd_demons.ldemons & demon_mask) >> i);
1007   own_layer_occupation_mid = own_layer_occupation_mid + ((own_demons.mdemons & demon_mask
       ) >> i) + ((odd_demons.mdemons & demon_mask) >> i);
1008   own_layer_occupation_high = own_layer_occupation_high + ((own_demons.hdemons &
       demon_mask) >> i) + ((odd_demons.hdemons & demon_mask) >> i);
1009
1010 }
1011 __shared__ unsigned int occupation_chunk[3 * 1024];
1012
1013 occupation_chunk[3 * threadIdx.x] = own_layer_occupation_low;
1014 occupation_chunk[3 * threadIdx.x + 1] = own_layer_occupation_mid;
1015 occupation_chunk[3 * threadIdx.x + 2] = own_layer_occupation_high;
1016
1017 __syncthreads();
1018
1019 if (threadIdx.x % 32 == 0){
1020   for (int i = 1; i < 32; i++){
1021     occupation_chunk[3 * threadIdx.x] = occupation_chunk[3 * threadIdx.x] +
       occupation_chunk[3 * threadIdx.x + 3 * i];
1022     occupation_chunk[3 * threadIdx.x + 1] = occupation_chunk[3 * threadIdx.x + 1] +
       occupation_chunk[3 * threadIdx.x + 3 * i + 1];
1023     occupation_chunk[3 * threadIdx.x + 2] = occupation_chunk[3 * threadIdx.x + 2] +
       occupation_chunk[3 * threadIdx.x + 3 * i + 2];
1024   }
1025 }
1026
1027 //add energies block wide
1028 __syncthreads();
1029 if (threadIdx.x == 0){
1030   for (int i = 1; i < 32; i++){
1031     occupation_chunk[0] = occupation_chunk[0] + occupation_chunk[32 * 3 * i];
1032     occupation_chunk[1] = occupation_chunk[1] + occupation_chunk[32 * 3 * i + 1];
1033     occupation_chunk[2] = occupation_chunk[2] + occupation_chunk[32 * 3 * i + 2];
1034   }
1035   //write results in memory
1036   *(demon_block_attributes_d + 4 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
       gridDim.x*gridDim.y)) = occupation_chunk[0];
1037   *(demon_block_attributes_d + 4 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
       gridDim.x*gridDim.y)+1) = occupation_chunk[1];
1038   *(demon_block_attributes_d + 4 * (blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*
       gridDim.x*gridDim.y)+2) = occupation_chunk[2];
1039 }
1040
1041
1042 }
1043
1044
1045 __global__ void applyPatternRemove3D(unsigned int* demons_d, unsigned int*
       random_pattern_d, int layer, int L1, int L2, int L3){
1046
1047 coordinates3D_t own_position = indexToCoordinates3DEven(threadIdx.x);
1048 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
1049
1050 demons3_t own_demons = getDemons3D(demons_d, own_position, L1, L2, L3);
1051 demons3_t odd_demons = getDemons3D(demons_d, odd_position, L1, L2, L3);
1052
1053 unsigned int random_pattern_own;
1054 unsigned int random_pattern_odd;
1055
1056
1057 unsigned int layer_demons_own;
```

```
1058 unsigned int layer_demons_odd;
1059
1060 unsigned int applied_changes_own;
1061 unsigned int applied_changes_odd;
1062
1063 unsigned int own_change_count = 0;
1064
1065 __shared__ unsigned int block_wide_changes[1024];
1066
1067 if (layer == DEMON_LAYER_LOW){
1068
1069   layer_demons_own = own_demons.ldemons;
1070   layer_demons_odd = odd_demons.ldemons;
1071
1072 }
1073 else if (layer == DEMON_LAYER_MID){
1074
1075   layer_demons_own = own_demons.mdemons;
1076   layer_demons_odd = odd_demons.mdemons;
1077
1078 }
1079 else if (layer == DEMON_LAYER_HIGH){
1080
1081   layer_demons_own = own_demons.hdemons;
1082   layer_demons_odd = odd_demons.hdemons;
1083
1084 }
1085
1086 random_pattern_own = *(random_pattern_d + own_position.x + own_position.y*L1 +
       own_position.z*L1*L2);
1087 random_pattern_odd = *(random_pattern_d + odd_position.x + odd_position.y*L1 +
       odd_position.z*L1*L2);
1088
1089 applied_changes_own = layer_demons_own;
1090 applied_changes_odd = layer_demons_odd;
1091
1092 //remove demons if possible
1093 layer_demons_own = layer_demons_own & (~random_pattern_own);
1094 layer_demons_odd = layer_demons_odd & (~random_pattern_odd);
1095
1096 /*
1097 bit 1: change applied (demon removed)
1098 bit 0: no change occurred
1099 */
1100 applied_changes_own = (applied_changes_own & random_pattern_own);
1101 applied_changes_odd = (applied_changes_odd & random_pattern_odd);
1102
1103 //compute number of changes
1104 unsigned int demon_mask;
1105 for (int i = 0; i < 32; i++){
1106   demon_mask = (1 << i);
1107   own_change_count = own_change_count + ((applied_changes_own & demon_mask) >> i) + ((
       applied_changes_odd & demon_mask) >> i);
1108 }
1109
1110 block_wide_changes[threadIdx.x] = own_change_count;
1111
1112 __syncthreads();
1113 //sum up changes warp wide
1114 if (threadIdx.x % 32 == 0){
1115
1116   for (int i = 1; i < 32; i++){
1117     block_wide_changes[threadIdx.x] = block_wide_changes[threadIdx.x] +
       block_wide_changes[threadIdx.x + i];
1118   }
1119
```

```
1120 }
1121 __syncthreads();
1122
1123 //thread 0 sums up rest and saves informations in random_pattern_d
1124 if (threadIdx.x == 0){
1125
1126   for (int i = 1; i < 32; i++){
1127     block_wide_changes[0] = block_wide_changes[0] + block_wide_changes[32 * i];
1128   }
1129   *(random_pattern_d + blockIdx.x * 64 + blockIdx.y * 32 * L1 + blockIdx.z*L1*L2) =
       block_wide_changes[0];
1130 }
1131
1132 //write back new demons to memory
1133
1134 if (layer == DEMON_LAYER_LOW){
1135
1136   own_demons.ldemons = layer_demons_own;
1137   odd_demons.ldemons = layer_demons_odd;
1138
1139 }
1140 else if (layer == DEMON_LAYER_MID){
1141
1142   own_demons.mdemons = layer_demons_own;
1143   odd_demons.mdemons = layer_demons_odd;
1144
1145 }
1146 else if (layer == DEMON_LAYER_HIGH){
1147
1148   own_demons.hdemons = layer_demons_own;
1149   odd_demons.hdemons = layer_demons_odd;
1150 }
1151
1152 writeDemons3D(own_demons, demons_d, own_position, L1, L2, L3);
1153 writeDemons3D(odd_demons, demons_d, odd_position, L1, L2, L3);
1154 }
1155
1156 __global__ void applyPatternSet3D(unsigned int* demons_d, unsigned int* random_pattern_d,
       int layer, int L1, int L2, int L3){
1157
1158 coordinates3D_t own_position = indexToCoordinates3DEven(threadIdx.x);
1159 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
1160
1161 demons3_t own_demons = getDemons3D(demons_d, own_position, L1, L2, L3);
1162 demons3_t odd_demons = getDemons3D(demons_d, odd_position, L1, L2, L3);
1163
1164 unsigned int random_pattern_own;
1165 unsigned int random_pattern_odd;
1166
1167
1168 unsigned int layer_demons_own;
1169 unsigned int layer_demons_odd;
1170
1171 unsigned int applied_changes_own;
1172 unsigned int applied_changes_odd;
1173
1174 unsigned int own_change_count = 0;
1175
1176 __shared__ unsigned int block_wide_changes[1024];
1177
1178 if (layer == DEMON_LAYER_LOW){
1179
1180   layer_demons_own = own_demons.ldemons;
1181   layer_demons_odd = odd_demons.ldemons;
1182
1183 }
```

```
1184 else if (layer == DEMON_LAYER_MID){
1185
1186   layer_demons_own = own_demons.mdemons;
1187   layer_demons_odd = odd_demons.mdemons;
1188
1189 }
1190 else if (layer == DEMON_LAYER_HIGH){
1191
1192   layer_demons_own = own_demons.hdemons;
1193   layer_demons_odd = odd_demons.hdemons;
1194
1195 }
1196
1197 random_pattern_own = *(random_pattern_d + own_position.x + own_position.y*L1 +
       own_position.z*L1*L2);
1198 random_pattern_odd = *(random_pattern_d + odd_position.x + odd_position.y*L1 +
       odd_position.z*L1*L2);
1199
1200 applied_changes_own = layer_demons_own;
1201 applied_changes_odd = layer_demons_odd;
1202
1203 //set demons if possible
1204 layer_demons_own = layer_demons_own | random_pattern_own;
1205 layer_demons_odd = layer_demons_odd | random_pattern_odd;
1206
1207 /*
1208 bit 1: change applied (demon set)
1209 bit 0: no change occurred
1210 */
1211 applied_changes_own = (applied_changes_own ^ random_pattern_own) & random_pattern_own;
1212 applied_changes_odd = (applied_changes_odd ^ random_pattern_odd) & random_pattern_odd;
1213
1214 //compute number of changes
1215 unsigned int demon_mask;
1216 for (int i = 0; i < 32; i++){
1217   demon_mask = (1 << i);
1218   own_change_count = own_change_count + ((applied_changes_own & demon_mask) >> i) + ((
       applied_changes_odd & demon_mask) >> i);
1219 }
1220
1221 block_wide_changes[threadIdx.x] = own_change_count;
1222
1223 __syncthreads();
1224 //sum up changes warp wide
1225 if (threadIdx.x % 32 == 0){
1226
1227   for (int i = 1; i < 32; i++){
1228     block_wide_changes[threadIdx.x] = block_wide_changes[threadIdx.x] +
       block_wide_changes[threadIdx.x + i];
1229 }
1230
1231 }
1232 __syncthreads();
1233
1234 //thread 0 sums up rest and saves informations in random_pattern_d
1235 if (threadIdx.x == 0){
1236
1237   for (int i = 1; i < 32; i++){
1238   block_wide_changes[0] = block_wide_changes[0] + block_wide_changes[32 * i];
1239   }
1240
1241   *(random_pattern_d + blockIdx.x*64+ blockIdx.y*32*L1 + blockIdx.z*L1*L2) =
       block_wide_changes[0];
1242 }
1243
1244 //write new demons back to memory
```

```
1245
1246 if (layer == DEMON_LAYER_LOW){
1247
1248   own_demons.ldemons = layer_demons_own;
1249   odd_demons.ldemons = layer_demons_odd;
1250
1251 }
1252 else if (layer == DEMON_LAYER_MID){
1253
1254   own_demons.mdemons = layer_demons_own;
1255   odd_demons.mdemons = layer_demons_odd;
1256
1257 }
1258 else if (layer == DEMON_LAYER_HIGH){
1259
1260   own_demons.hdemons = layer_demons_own;
1261   odd_demons.hdemons = layer_demons_odd;
1262 }
1263
1264 writeDemons3D(own_demons, demons_d, own_position, L1, L2, L3);
1265 writeDemons3D(odd_demons, demons_d, odd_position, L1, L2, L3);
1266 }
1267
1268
1269 __global__ void updateDemons3D(unsigned int* demons_d, unsigned int* randoms, double
     set_probabilities[], int L1, int L2, int L3){
1270 demons3_t own_demons;
1271 demons3_t odd_demons;
1272
1273 coordinates3D_t own_position = indexToCoordinates3DEven(threadIdx.x);
1274 coordinates3D_t odd_position = indexToCoordinates3DOdd(threadIdx.x);
1275
1276 own_demons = getDemons3D(demons_d, own_position, L1, L2, L3);
1277 odd_demons = getDemons3D(demons_d, odd_position, L1, L2, L3);
1278
1279 int current_random = (int) *(randoms + 1024*(blockIdx.x + blockIdx.y*gridDim.x + blockIdx
     .z*gridDim.x*gridDim.y) + threadIdx.x);
1280 double set_prob[3];
1281 set_prob[0] = set_probabilities[0];
1282 set_prob[1] = set_probabilities[1];
1283 set_prob[2] = set_probabilities[2];
1284
1285 for (int i = 0; i < 32; i++){
1286   //for low demons
1287   simpleRandomInteger0(current_random);
1288   if ((current_random / 2147483646.0) <= set_prob[0]){
1289     own_demons.ldemons = own_demons.ldemons | (1 << i);
1290   }
1291   else{
1292     own_demons.ldemons = own_demons.ldemons & ~(1 << i);
1293   }
1294
1295   simpleRandomInteger0(current_random);
1296   if ((current_random / 2147483646.0) <= set_prob[0]){
1297     odd_demons.ldemons = odd_demons.ldemons | (1 << i);
1298   }
1299   else{
1300     odd_demons.ldemons = odd_demons.ldemons & ~(1 << i);
1301   }
1302
1303   simpleRandomInteger0(current_random);
1304   if ((current_random / 2147483646.0) <= set_prob[1]){
1305     own_demons.mdemons = own_demons.mdemons | (1 << i);
1306   }
1307   else{
1308     own_demons.mdemons = own_demons.mdemons & ~(1 << i);
```

```
1309    }
1310
1311    simpleRandomInteger0(current_random);
1312    if ((current_random / 2147483646.0) <= set_prob[1]){
1313      odd_demons.mdemons = odd_demons.mdemons | (1 << i);
1314    }
1315    else{
1316      odd_demons.mdemons = odd_demons.mdemons & ~(1 << i);
1317    }
1318
1319    simpleRandomInteger0(current_random);
1320    if ((current_random / 2147483646.0) <= set_prob[2]){
1321      own_demons.hdemons = own_demons.hdemons | (1 << i);
1322    }
1323    else{
1324      own_demons.hdemons = own_demons.hdemons & ~(1 << i);
1325    }
1326
1327    simpleRandomInteger0(current_random);
1328    if ((current_random / 2147483646.0) <= set_prob[2]){
1329      odd_demons.hdemons = odd_demons.hdemons | (1 << i);
1330    }
1331    else{
1332      odd_demons.hdemons = odd_demons.hdemons & ~(1 << i);
1333    }
1334  }
1335
1336  //write back randoms
1337  *(randoms + 1024*(blockIdx.x + blockIdx.y*gridDim.x + blockIdx.z*gridDim.x*gridDim.y) +
        threadIdx.x) = (unsigned int) current_random;
1338
1339  //write back demons
1340
1341  writeDemons3D(own_demons, demons_d, own_position, L1, L2, L3);
1342  writeDemons3D(odd_demons, demons_d, odd_position, L1, L2, L3);
1343  }
```

8.2.2.8 Lattice.cu

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include "Lattice.hpp"
5  #include "routines3D.hpp"
6  #include "routines2D.hpp"
7  #include <iostream>
8
9  using namespace std;
10
11 void Lattice::demonRandomChangesSet3D(int &changes, int layer, int pattern_fact){
12 unsigned int total_changes;
13 unsigned int random_site;
14 dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
15 while (changes > 0){
16
17   //reinitialize random pattern
18   cudaMemset(random_pattern_d, 0, L1*L2*L3*sizeof(unsigned int));
19   cudaDeviceSynchronize();
20   cudaMemcpy(random_pattern, random_pattern_d, L1*L2*L3*sizeof(unsigned int),
       cudaMemcpyDeviceToHost);
21
22
```

```
23    //create random pattern
24    for (int k = 0; k < pattern_fact*changes; k++){
25      //random site
26      random_site = random_generator->getRandomInteger() % (L1*L2*L3);
27      random_pattern[random_site] = random_pattern[random_site] | (1 << (random_generator->
        getRandomInteger() % 32));
28    }
29
30    cudaMemcpy(random_pattern_d, random_pattern, L1*L2*L3*sizeof(unsigned int),
        cudaMemcpyHostToDevice);
31    applyPatternSet3D << <blocks, 1024 >> >(demons_d, random_pattern_d, layer, L1, L2, L3);
32    cudaMemcpy(random_pattern, random_pattern_d, L1*L2*L3*sizeof(unsigned int),
        cudaMemcpyDeviceToHost);
33
34    //calculate total changes
35    total_changes = 0;
36    for (int x = 0; x < L1; x = x + 64){
37      for (int y = 0; y < L2; y = y + 32){
38        for (int z = 0; z < L3; z++){
39
40          total_changes = total_changes + *(random_pattern + x + y*L1 + z*L1*L2);
41
42        }
43      }
44    }
45    changes = changes - total_changes;
46  }
47  }
48
49  void Lattice::demonRandomChangesRemove3D(int &changes, int layer, int pattern_fact){
50  unsigned int total_changes;
51  unsigned int random_site;
52  dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
53  while (changes > 0){
54
55    //reinitialize random pattern
56    cudaMemset(random_pattern_d, 0, L1*L2*L3*sizeof(unsigned int));
57    cudaDeviceSynchronize();
58    cudaMemcpy(random_pattern, random_pattern_d, L1*L2*L3*sizeof(unsigned int),
        cudaMemcpyDeviceToHost);
59
60
61    //create random pattern
62    for (int k = 0; k < pattern_fact*changes; k++){
63      //random site
64      random_site = random_generator->getRandomInteger() % (L1*L2*L3);
65      random_pattern[random_site] = random_pattern[random_site] | (1 << (random_generator->
        getRandomInteger() % 32));
66    }
67
68    cudaMemcpy(random_pattern_d, random_pattern, L1*L2*L3*sizeof(unsigned int),
        cudaMemcpyHostToDevice);
69    applyPatternRemove3D << <blocks, 1024 >> >(demons_d, random_pattern_d, layer, L1, L2,
        L3);
70    cudaMemcpy(random_pattern, random_pattern_d, L1*L2*L3*sizeof(unsigned int),
        cudaMemcpyDeviceToHost);
71
72    //calculate total changes
73    total_changes = 0;
74    for (int x = 0; x < L1; x = x + 64){
75      for (int y = 0; y < L2; y = y + 32){
76        for (int z = 0; z < L3; z++){
77
78          total_changes = total_changes + *(random_pattern + x + y*L1 + z*L1*L2);
79
80        }
```

```
81      }
82    }
83    changes = changes - total_changes;
84  }
85
86  }
87
88  void Lattice::demonRandomChangesSet2D(int &changes, int layer, int pattern_fact){
89  unsigned int total_changes;
90  unsigned int random_site;
91  dim3 blocks(this->gridDimX, this->gridDimY);
92  while (changes > 0){
93
94    //reinitialize random pattern
95    cudaMemset(random_pattern_d, 0, L1*L2*sizeof(unsigned int));
96    cudaDeviceSynchronize();
97    cudaMemcpy(random_pattern, random_pattern_d, L1*L2*sizeof(unsigned int),
        cudaMemcpyDeviceToHost);
98
99
100   //create random pattern
101   for (int k = 0; k < pattern_fact*changes; k++){
102     //random site
103     random_site = random_generator->getRandomInteger() % (L1*L2);
104     random_pattern[random_site] = random_pattern[random_site] | (1 << (random_generator->
        getRandomInteger() % 32));
105   }
106
107   cudaMemcpy(random_pattern_d, random_pattern, L1*L2*sizeof(unsigned int),
        cudaMemcpyHostToDevice);
108   applyPatternSet2D << <blocks, 1024 >> >(demons_d, random_pattern_d, layer, L1, L2);
109   cudaMemcpy(random_pattern, random_pattern_d, L1*L2*sizeof(unsigned int),
        cudaMemcpyDeviceToHost);
110
111   //calculate total changes
112   total_changes = 0;
113   for (int x = 0; x < L1; x = x + 64){
114     for (int y = 0; y < L2; y = y + 32){
115
116       total_changes = total_changes + *(random_pattern + x + y*L1);
117
118     }
119   }
120   changes = changes - total_changes;
121 }
122
123 }
124
125 void Lattice::demonRandomChangesRemove2D(int &changes, int layer, int pattern_fact){
126 unsigned int total_changes;
127 unsigned int random_site;
128 dim3 blocks(this->gridDimX, this->gridDimY);
129 while (changes > 0){
130
131   //reinitialize random pattern
132     cudaMemset(random_pattern_d, 0, L1*L2*sizeof(unsigned int));
133     cudaDeviceSynchronize();
134     cudaMemcpy(random_pattern, random_pattern_d, L1*L2*sizeof(unsigned int),
        cudaMemcpyDeviceToHost);
135
136
137     //create random pattern
138     for (int k = 0; k < pattern_fact*changes; k++){
139       //random site
140       random_site = random_generator->getRandomInteger() % (L1*L2);
```

```
141       random_pattern[random_site] = random_pattern[random_site] | (1 << (random_generator
      ->getRandomInteger() % 32));
142     }
143
144     cudaMemcpy(random_pattern_d, random_pattern, L1*L2*sizeof(unsigned int),
      cudaMemcpyHostToDevice);
145     applyPatternRemove2D <<<blocks, 1024 >>>(demons_d, random_pattern_d, layer, L1, L2);
146     cudaMemcpy(random_pattern, random_pattern_d, L1*L2*sizeof(unsigned int),
      cudaMemcpyDeviceToHost);
147
148     //calculate total changes
149     total_changes = 0;
150     for (int x = 0; x < L1; x = x + 64){
151       for (int y = 0; y < L2; y = y + 32){
152
153         total_changes = total_changes + *(random_pattern + x + y*L1);
154
155       }
156     }
157     changes = changes - total_changes;
158 }
159
160 }
161
162 void Lattice::updateSystem(){
163 if (dimension == _ISING_3D_ && boundary == _PERIODIC_){
164   //metropolis update
165   dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
166   dim3 blocks_red(this->gridDimX, this->gridDimY / 2, this->gridDimZ);
167   unsigned int random_offset;
168
169
170   metropolis3DPeriodicEven <<<blocks, 1024 >>>(this->spins_d, this->demons_d, this->L1,
      this->L2, this->L3);
171   metropolis3DPeriodicOdd <<<blocks, 1024 >>>(this->spins_d, this->demons_d, this->L1,
      this->L2, this->L3);
172
173   cudaDeviceSynchronize();
174
175   random_offset = this->random_generator->getRandomInteger();
176   shiftDemonsBlockWide3D <<<blocks, 1024 >>> (this->demons_d, random_offset, this->L1,
      this->L2, this->L3);
177
178   random_offset = this->random_generator->getRandomInteger();
179   shiftDemonsGridWide3D <<<blocks_red, 1024 >>>(this->demons_d, random_offset, this->L1,
      this->L2, this->L3);
180
181   cudaDeviceSynchronize();
182 }
183 else if (dimension == _ISING_2D_ && boundary == _PERIODIC_){
184   //metropolis update
185   dim3 blocks(this->gridDimX, this->gridDimY);
186   dim3 blocks_red(this->gridDimX, this->gridDimY / 2);
187   unsigned int random_offset;
188
189
190   metropolis2DPeriodicEven <<<blocks, 1024 >>>(this->spins_d, this->demons_d, this->L1,
      this->L2);
191   metropolis2DPeriodicOdd <<<blocks, 1024 >>>(this->spins_d, this->demons_d, this->L1,
      this->L2);
192
193   cudaDeviceSynchronize();
194
195   random_offset = this->random_generator->getRandomInteger();
196   shiftDemonsBlockWide2D << <blocks, 1024 >> > (this->demons_d, random_offset, this->L1,
      this->L2);
```

```
197
198   random_offset = this->random_generator->getRandomInteger();
199   shiftDemonsGridWide2D << <blocks_red, 1024 >> >(this->demons_d, random_offset, this->L1
        , this->L2);
200
201   cudaDeviceSynchronize();
202 }
203
204
205 }
206
207 void Lattice::updateDemonLayers(){
208
209 if (dimension == _ISING_3D_){
210
211   dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
212   demonLayersOccupationBlockWide3D<<<blocks, 1024>>>(this->demons_d, this->
        demon_block_attributes_d, this->L1, this->L2, this->L3);
213
214   cudaMemcpy(demon_block_attributes, demon_block_attributes_d, (4 * (L1*L2*L3) / 2048)*
        sizeof(unsigned int), cudaMemcpyDeviceToHost);
215
216   //compute total occupation for each demon layer.
217   int total_occupation[DEMON_LAYERS];
218
219   total_occupation[DEMON_LAYER_LOW] = 0;
220   total_occupation[DEMON_LAYER_MID] = 0;
221   total_occupation[DEMON_LAYER_HIGH] = 0;
222
223   for (int i = 0; i < (L1*L2*L3) / 2048; i++){
224
225     //sum up total number of occupied demon energies.
226     total_occupation[DEMON_LAYER_LOW] = total_occupation[DEMON_LAYER_LOW] +
        demon_block_attributes[4 * i];
227     total_occupation[DEMON_LAYER_MID] = total_occupation[DEMON_LAYER_MID] +
        demon_block_attributes[4* i + 1];
228     total_occupation[DEMON_LAYER_HIGH] = total_occupation[DEMON_LAYER_HIGH] +
        demon_block_attributes[4 * i + 2];
229
230   }
231
232
233
234   //calculate changes for the demon layer
235   int demon_layer_change[DEMON_LAYERS];
236
237   demon_layer_change[DEMON_LAYER_LOW] = total_occupation[DEMON_LAYER_LOW] -
        demon_layer_occupation_dist[DEMON_LAYER_LOW]->getBinomialInteger();
238   demon_layer_change[DEMON_LAYER_MID] = total_occupation[DEMON_LAYER_MID] -
        demon_layer_occupation_dist[DEMON_LAYER_MID]->getBinomialInteger();
239   demon_layer_change[DEMON_LAYER_HIGH] = total_occupation[DEMON_LAYER_HIGH] -
        demon_layer_occupation_dist[DEMON_LAYER_HIGH]->getBinomialInteger();
240
241
242   double pattern_fact;
243
244   for (int i = 0; i < DEMON_LAYERS; i++){
245     if (demon_layer_change[i] < 0){
246
247       demon_layer_change[i] = -demon_layer_change[i];
248       pattern_fact = ((32 * L1*L2*L3 - total_occupation[i]) / ((double)32 * L1*L2*L3));
249
250       if (pattern_fact >= 0.5){
251         demonRandomChangesSet3D(demon_layer_change[i], i, 1);
252       }
253       else{
```

```
254
255         pattern_fact = 1 / pattern_fact;
256         demonRandomChangesSet3D(demon_layer_change[i], i, (int) pattern_fact);
257
258         if (demon_layer_change[i] < 0){
259
260           demon_layer_change[i] = -demon_layer_change[i];
261           demonRandomChangesRemove3D(demon_layer_change[i], i, 1);
262
263         }
264       }
265
266     }
267     else{
268       pattern_fact = total_occupation[i] / ((double)32 * L1*L2*L3);
269
270       if (((double)total_occupation[i])/((double)32*L1*L2*L3) >= 0.5){
271         demonRandomChangesRemove3D(demon_layer_change[i], i, 1);
272       }
273       else{
274
275         pattern_fact =  1 / pattern_fact;
276         demonRandomChangesRemove3D(demon_layer_change[i], i, (int) pattern_fact);
277
278         if (demon_layer_change[i] < 0){
279
280           demon_layer_change[i] = -demon_layer_change[i];
281           demonRandomChangesSet3D(demon_layer_change[i], i, 1);
282
283         }
284       }
285     }
286   }
287
288 }
289 else if (dimension == _ISING_2D_){
290   dim3 blocks(this->gridDimX, this->gridDimY);
291   demonLayersOccupationBlockWide2D << <blocks, 1024 >> >(this->demons_d, this->
      demon_block_attributes_d, this->L1, this->L2);
292
293   cudaMemcpy(demon_block_attributes, demon_block_attributes_d, (4 * (L1*L2) / 2048)*
      sizeof(unsigned int), cudaMemcpyDeviceToHost);
294
295   //compute total occupation for each demon layer.
296   int total_occupation[DEMON_LAYERS];
297
298   total_occupation[DEMON_LAYER_LOW] = 0;
299   total_occupation[DEMON_LAYER_MID] = 0;
300   total_occupation[DEMON_LAYER_HIGH] = 0;
301
302   for (int i = 0; i < (L1*L2) / 2048; i++){
303
304     //sum up total number of occupied demon energies.
305     total_occupation[DEMON_LAYER_LOW] = total_occupation[DEMON_LAYER_LOW] +
        demon_block_attributes[4 * i];
306     total_occupation[DEMON_LAYER_MID] = total_occupation[DEMON_LAYER_MID] +
        demon_block_attributes[4 * i + 1];
307     total_occupation[DEMON_LAYER_HIGH] = total_occupation[DEMON_LAYER_HIGH] +
        demon_block_attributes[4 * i + 2];
308
309
310   }
311
312
313
314   //calculate changes for the demon layer
```

```
315   int demon_layer_change[DEMON_LAYERS];
316
317   demon_layer_change[DEMON_LAYER_LOW] = total_occupation[DEMON_LAYER_LOW] -
        demon_layer_occupation_dist[DEMON_LAYER_LOW]->getBinomialInteger();
318   demon_layer_change[DEMON_LAYER_MID] = total_occupation[DEMON_LAYER_MID] -
        demon_layer_occupation_dist[DEMON_LAYER_MID]->getBinomialInteger();
319   demon_layer_change[DEMON_LAYER_HIGH] = total_occupation[DEMON_LAYER_HIGH] -
        demon_layer_occupation_dist[DEMON_LAYER_HIGH]->getBinomialInteger();
320
321   double pattern_fact;
322
323   for (int i = 0; i < DEMON_LAYERS; i++){
324     if (demon_layer_change[i] < 0){
325
326       demon_layer_change[i] = -demon_layer_change[i];
327       pattern_fact = ((32 * L1*L2 - total_occupation[i]) / ((double)32 * L1*L2));
328
329     if (pattern_fact >= 0.5){
330       demonRandomChangesSet2D(demon_layer_change[i], i, 1);
331     }
332     else{
333
334       pattern_fact = 1 / pattern_fact;
335       demonRandomChangesSet2D(demon_layer_change[i], i, (int)pattern_fact);
336
337       if (demon_layer_change[i] < 0){
338
339         demon_layer_change[i] = -demon_layer_change[i];
340         demonRandomChangesRemove2D(demon_layer_change[i], i, 1);
341
342       }
343     }
344
345   }
346   else{
347     pattern_fact = total_occupation[i] / ((double)32 * L1*L2);
348
349     if (((double)total_occupation[i]) / ((double)32 * L1*L2) >= 0.5){
350       demonRandomChangesRemove2D(demon_layer_change[i], i, 1);
351     }
352     else{
353
354       pattern_fact = 1 / pattern_fact;
355       demonRandomChangesRemove2D(demon_layer_change[i], i, (int)pattern_fact);
356
357       if (demon_layer_change[i] < 0){
358
359         demon_layer_change[i] = -demon_layer_change[i];
360         demonRandomChangesSet2D(demon_layer_change[i], i, 1);
361
362       }
363     }
364   }
365 }
366
367 }
368 }
369
370
371 void Lattice::updateDemonLayersNew(){
372 if (dimension == _ISING_2D_){
373   dim3 blocks(gridDimX, gridDimY);
374   updateDemons2D <<<blocks, 1024 >>>(demons_d, random_pattern_d, set_probabilities_d, L1,
        L2);
375   cudaDeviceSynchronize();
376
```

```
377 }
378 else if (dimension == _ISING_3D_){
379   dim3 blocks(gridDimX, gridDimY, gridDimZ);
380   updateDemons3D <<<blocks, 1024 >>>(demons_d, random_pattern_d, set_probabilities_d, L1,
      L2, L3);
381   cudaDeviceSynchronize();
382 }
383 }
384
385 void Lattice::observeTotalEnergies(int* energies){
386
387 if (dimension == _ISING_3D_ && boundary == _PERIODIC_){
388
389   dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
390
391   totalEnergyBlockWide3DPeriodic << <blocks, 1024, 34816 >> >(this->spins_d, this->
      demons_d, this->energies_block_wide_d, this->L1, this->L2, this->L3);
392   cudaMemcpy(this->energies_block_wide, this->energies_block_wide_d, (32 * (L1*L2*L3) /
      2048) * sizeof(int), cudaMemcpyDeviceToHost);
393
394   cudaDeviceSynchronize();
395
396   //sum over block energies.
397   for (int i = 0; i < 32; i++){
398     energies[i] = 0;
399
400     //sum over blockwide energies
401     for (int x = 0; x < L1 / 64; x++){
402       for (int y = 0; y < L2 / 32; y++){
403         for (int z = 0; z < L3; z++){
404           energies[i] = energies[i] + *(energies_block_wide + 32 * (x + y*(this->gridDimX
      ) + z*(this->gridDimX)*(this->gridDimY)) + i);
405         }
406       }
407     }
408   }
409 }
410 else if (dimension == _ISING_2D_ && boundary == _PERIODIC_){
411
412   dim3 blocks(this->gridDimX, this->gridDimY);
413
414   totalEnergyBlockWide2DPeriodic << <blocks, 1024, 34816 >> >(this->spins_d, this->
      demons_d, this->energies_block_wide_d, this->L1, this->L2);
415   cudaMemcpy(this->energies_block_wide, this->energies_block_wide_d, (32 * (L1*L2) /
      2048) * sizeof(int), cudaMemcpyDeviceToHost);
416
417   cudaDeviceSynchronize();
418
419   //sum over block energies.
420   for (int i = 0; i < 32; i++){
421     energies[i] = 0;
422
423     //sum over blockwide energies
424     for (int x = 0; x < L1 / 64; x++){
425       for (int y = 0; y < L2 / 32; y++){
426         energies[i] = energies[i] + *(energies_block_wide + 32 * (x + y*(this->gridDimX))
       + i);
427       }
428     }
429   }
430 }
431 }
432
433 void Lattice::initializeRandoms(){
434 if (dimension == _ISING_2D_){
435   for (int i = 0; i < L1 *L2; i++){
```

```
436     do{
437       random_pattern[i] = (random_generator->getRandomInteger()) >> 1;
438       //random_pattern[i] = i+1;
439     } while (random_pattern[i] == 0);
440   }
441   //cudaMemset(random_pattern_d, 0, L1*L2*sizeof(unsigned int));
442   cudaMemcpy(random_pattern_d, random_pattern, L1*L2*sizeof(unsigned int),
      cudaMemcpyHostToDevice);
443
444 }
445 else if (dimension == _ISING_3D_){
446   for (int i = 0; i < L1 *L2*L3; i++){
447     do{
448       random_pattern[i] = (random_generator->getRandomInteger()) >> 1;
449     } while (random_pattern[i] == 0);
450   }
451   //cudaMemset(random_pattern_d, 0, L1*L2*L3*sizeof(unsigned int));
452   cudaMemcpy(random_pattern_d, random_pattern, L1*L2*L3*sizeof(unsigned int),
      cudaMemcpyHostToDevice);
453 }
454 }
455
456 void Lattice::observeTotalSpinEnergies(int* energies){
457 if (dimension == _ISING_3D_ && boundary == _PERIODIC_){
458
459   dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
460
461   totalSpinEnergyBlockWide3DPeriodic <<<blocks, 1024, 34816 >> >(this->spins_d, this->
      demons_d, this->energies_block_wide_d, this->L1, this->L2, this->L3);
462   cudaMemcpy(this->energies_block_wide, this->energies_block_wide_d, (32 * (L1*L2*L3) /
      2048) * sizeof(int), cudaMemcpyDeviceToHost);
463
464   cudaDeviceSynchronize();
465
466   //sum over block energies.
467   for (int i = 0; i < 32; i++){
468     energies[i] = 0;
469
470     //sum over blockwide energies
471     for (int x = 0; x < L1 / 64; x++){
472       for (int y = 0; y < L2 / 32; y++){
473         for (int z = 0; z < L3; z++){
474           energies[i] = energies[i] + *(energies_block_wide + 32 * (x + y*(this->gridDimX
      ) + z*(this->gridDimX)*(this->gridDimY)) + i);
475         }
476       }
477     }
478   }
479 }
480 else if (dimension == _ISING_2D_ && boundary == _PERIODIC_){
481
482   dim3 blocks(this->gridDimX, this->gridDimY);
483
484   totalSpinEnergyBlockWide2DPeriodic <<<blocks, 1024, 34816 >> >(this->spins_d, this->
      demons_d, this->energies_block_wide_d, this->L1, this->L2);
485   cudaMemcpy(this->energies_block_wide, this->energies_block_wide_d, (32 * (L1*L2) /
      2048) * sizeof(int), cudaMemcpyDeviceToHost);
486
487   cudaDeviceSynchronize();
488
489   //sum over block energies.
490   for (int i = 0; i < 32; i++){
491     energies[i] = 0;
492
493     //sum over blockwide energies
494     for (int x = 0; x < L1 / 64; x++){
```

```
495        for (int y = 0; y < L2 / 32; y++){
496          energies[i] = energies[i] + *(energies_block_wide + 32 * (x + y*(this->gridDimX))
       + i);
497        }
498      }
499    }
500  }
501  }
502
503  void Lattice::observeTotalDemonEnergies(int* energies){
504  if (dimension == _ISING_3D_){
505
506    dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
507
508    totalDemonEnergyBlockWide3D <<<blocks, 1024, 34816 >> >(this->demons_d, this->
       energies_block_wide_d, this->L1, this->L2, this->L3);
509    cudaMemcpy(this->energies_block_wide, this->energies_block_wide_d, (32 * (L1*L2*L3) /
       2048) * sizeof(int), cudaMemcpyDeviceToHost);
510
511    cudaDeviceSynchronize();
512
513    //sum over block energies.
514    for (int i = 0; i < 32; i++){
515      energies[i] = 0;
516
517      //sum over blockwide energies
518      for (int x = 0; x < L1 / 64; x++){
519        for (int y = 0; y < L2 / 32; y++){
520            for (int z = 0; z < L3; z++){
521
522            energies[i] = energies[i] + *(energies_block_wide + 32 * (x + y*(this->gridDimX
       ) +     z*(this->gridDimX)*(this->gridDimY)) + i);
523          }
524        }
525      }
526    }
527  }
528  else if (dimension == _ISING_2D_){
529
530    dim3 blocks(this->gridDimX, this->gridDimY);
531
532    totalDemonEnergyBlockWide2D <<<blocks, 1024, 34816 >>>(this->demons_d,this->
       energies_block_wide_d, this->L1, this->L2);
533    cudaMemcpy(this->energies_block_wide, this->energies_block_wide_d, (32 * (L1*L2) /
       2048) * sizeof(int), cudaMemcpyDeviceToHost);
534
535    cudaDeviceSynchronize();
536
537    //sum over block energies.
538    for (int i = 0; i < 32; i++){
539      energies[i] = 0;
540
541      //sum over blockwide energies
542      for (int x = 0; x < L1 / 64; x++){
543        for (int y = 0; y < L2 / 32; y++){
544          energies[i] = energies[i] + *(energies_block_wide + 32 * (x + y*(this->gridDimX))
       + i);
545        }
546      }
547    }
548  }
549
550  }
551
552  void Lattice::observeTotalMagnetization(int* magnetization){
553  if (dimension == _ISING_3D_){
```

```
554
555   dim3 blocks(this->gridDimX, this->gridDimY, this->gridDimZ);
556
557   totalMagnetizationBlockWide3D <<<blocks, 1024, 34816 >>>(this->spins_d, this->
        magnetization_block_wide_d, this->L1, this->L2, this->L3);
558   cudaMemcpy(this->magnetization_block_wide, this->magnetization_block_wide_d, (32 * (L1*
        L2*L3) / 2048) * sizeof(int), cudaMemcpyDeviceToHost);
559
560   cudaDeviceSynchronize();
561
562   //sum over block energies.
563   for (int i = 0; i < 32; i++){
564     magnetization[i] = 0;
565
566   //sum over blockwide energies
567   for (int x = 0; x < L1 / 64; x++){
568     for (int y = 0; y < L2 / 32; y++){
569       for (int z = 0; z < L3; z++){
570         magnetization[i] = magnetization[i] + *(magnetization_block_wide + 32 * (x + y*(
        this->gridDimX) + z*(this->gridDimX)*(this->gridDimY)) + i);
571       }
572     }
573   }
574 }
575 }
576 else if (dimension == _ISING_2D_){
577
578   dim3 blocks(this->gridDimX, this->gridDimY);
579
580   totalMagnetizationBlockWide2D <<<blocks, 1024, 34816 >>>(this->spins_d, this->
        magnetization_block_wide_d, this->L1, this->L2);
581   cudaMemcpy(this->magnetization_block_wide, this->magnetization_block_wide_d, (32 * (L1*
        L2) / 2048) * sizeof(int), cudaMemcpyDeviceToHost);
582
583   cudaDeviceSynchronize();
584
585   //sum over block energies.
586   for (int i = 0; i < 32; i++){
587     magnetization[i] = 0;
588
589     //sum over blockwide energies
590     for (int x = 0; x < L1 / 64; x++){
591       for (int y = 0; y < L2 / 32; y++){
592         magnetization[i] = magnetization[i] + *(magnetization_block_wide + 32 * (x + y*(
        this->gridDimX)) + i);
593       }
594     }
595   }
596 }
597 }
```

Literaturverzeichnis

[1] *Finite Size Scaling Analysis of Ising Model Block Distribution Functions*, K. Binder, 1981, Zeitschrift für Physik B - Condensed Matter, Vol. 43, S. 119-140

[2] *Statistische Mechanik*, 2.Auflage, F. Schwabl, 2004, Springer-Verlag Berlin Heidelberg

[3] *Statistische Physik*, 2. Auflage, T. Fließbach, 1995, Spektrum Akademischer Verlag Heidelberg

[4] *Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transistion*, L. Onsager, 1943, Physical Review, Vol. 65, No. 3 & 4, S. 117-149

[5] *Monte Carlo Studies of the three dimensional Ising Model*, Habilitationsschrift, M. Hasenbusch, Institut für Physik Humboldt Universität zu Berlin

[6] *Monte Carlo Simulation in der statistischen Physik*, Vorlesungsskript, M. Hasenbusch, Institut für Physik Humboldt Universität zu Berlin

[7] *The Spontaneous Magnetization of a Two-Dimensional Ising-Model*, C. N. Yang, 1951, Physical Review Vol. 85, No. 5, S. 808-816

[8] *Critical Phenomena and Renormalization-Group Theory*, A. Pelissetto und E. Vicari, 2002, Physics Report 368, S. 549

[9] *Renormalization group theory: Its basis and formulation in statistical physics*, E. Fisher, 1998, Reviews of Modern Physics, Vol. 70, No. 2, S. 653-681

[10] *Finite-size tests of hyperscaling*, K.Binder, M. Nauenberg, V. Privman, A. P. Young, 1985, Physical Review B, Vol. 31, No. 3, S. 1498-1502

[11] *Introduction to Monte Carlo methods for an Ising Model of a Ferromagnet*, J. Kotze, 2008, arXiv:0803.0217v1 [cond-mat.stat-mech]

[12] *Universal ratio of magnetization moments in two-dimensional Ising models*, G. Kamieniarz, 1992, Journal of Physics A: Mathematical and General, Vol. 26, S. 201-212

[13] *Beitrag zur Theorie des Ferromagnetismus*, E. Ising, 1925, Zeitschrift für Physik, Vol. 31, S. 253-258

[14] *Basic Concepts in Computational Physics*, 2. Auflage, B. A. Stickler und E. Schachinger, 2014, Springer Verlag

[15] *Markov Chain Monte Carlo Simulations and Their Statistical Analysis*, B. A. Berg, 2004, World Scientific Publishing Co. Pte. Ltd.

[16] *A NEW CLASS OF RANDOM NUMBER GENERATORS*, G. Margsaglia und A. Zaman, 1991, The Annals of Applied Probability, Vol. 1, No. 3, S. 462-480

[17] *Numerical Recipes in C - The Art of Scientific Computing*, 2. Auflage, W. H. Press, S. A. Teukolsky, W. T. Vetterling und B. P. Flannery, 1992, Cambridge University Press

[18] *Microcanonical Monte Carlo Simulation*, M. Creutz, 1983, Physical Review Letters, Vol. 50, No. 19, S. 1411-1414

[19] *MICROCANONICAL SIMULATION OF ISING SYSTEMS*, G. Bhanot, M. Creutz und H. Neuberger, 1984, Nuclear Physics B235, S. 417-434

[20] *Multicanonical cluster algorithm and the two-dimensional 7-state Potts model*, K. Rummukainen, 1993, Nuclear Physics B390, S. 621-636

[21] *Comparison of Monte Carlo results for the 3D Ising interface tension and interface energy with (extrapolated) series expansions*, M. Hasenbusch und K. Pinn, 1994, Physica A 203, S. 189-213

[22] *Computer Methods for Sampling from Gamma, Beta, Poisson and Binomial Distributions*, J.H. Ahrens und U. Dieter, 1974, Computing 12, S. 223-246

[23] *First Draft of a Report on the EDVAC*, J. von Neumann, 1993, IEEE Annals of the History of Computing, Vol. 15, No. 4, S. 27-43

[24] *CUDA C PROGRAMMING GUIDE*, Version: PG-02829-001_v7.5, 2015, herausgegeben durch den Hersteller *Nvidia*

[25] *CUDA C BEST PRACTICES GUIDE*, Version: DG-05603-011_v7.5, 2015, herausgegeben durch den Hersteller *Nvidia*

[26] *CUDA RUNTIME API*, Version: v7.5, 2015, herausgegeben durch den Hersteller *Nvidia*

[27] *CUDA application design and development*, R. Farber, 2012, Waltham, MA: Morgan Kaufmann

[28] *Using Shared Memory in CUDA C/C++*, M. Harris, 2013,
Adresse: *https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/*,
Letzter Aufruf: 26.08.2016

[29] *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*, M. Harris, 2013, Adresse: *https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/*, Letzter Aufruf: 27.08.2016

[30] *NVIDIA GeForce GTX 980*, Whitepaper herausgegeben durch den Hersteller *Nvidia*, Adresse: *http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF*, Letzter Aufruf: 28.08.2016

[31] *GPU accelerated Monte Carlo simulation of the 2D ans 3D Ising model*, T. Preis, P. Virnau, W. Paul und J. J. Schneider, 2009, Journal of Computational Physics 228, S. 4468-4477

[32] *Multi-GPU accelerated mutli-spin Monte Carlo simulations of the 2D Ising model*, B. Block, P. Virnau und T. Preis, 2010, Computer Physics Communications 181, S. 1549-1556

[33] *Parallel graph component labelling with GPUs and CUDA*, K. A. Hawick, A. Leist und D. P. Playne, 2010, Parallel Computing 36, S. 655-678

Selbstständigkeitserklärung

Hiermit versichere ich, die vorgelegte Thesis selbstständig und ohne unerlaubte fremde Hilfe und nur mit den Hilfen angefertigt zu haben, die ich in der Thesis angegeben habe. Alle Textstellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen sind, und alle Angaben die auf mündlichen Auskünften beruhen, sind als solche kenntlich gemacht. Bei den von mir durchgeführten und in der Thesis erwähnten Untersuchungen habe ich die Grundsätze guter wissenschaftlicher Praxis, wie sie in der ‚Satzung der Justus-Liebig-Universität zur Sicherung guter wissenschaftlicher Praxis' niedergelegt sind, eingehalten. Gemäß § 25 Abs. 6 der Allgemeinen Bestimmungen für modularisierte Studiengänge dulde ich eine Überprüfung der Thesis mittels Anti-Plagiatssoftware.

Datum	Unterschrift