TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

Diplomarbeit

zur Erlangung des akademischen Grades
Diplomingenieur für Informationssystemtechnik

A highly-parallel Monte-Carlo-Simulation of X-Ray-Scattering using a Particle-Mesh-Code on GPUs

Alexander Grund
(Geboren am 26. September 1989 in Dresden)

Erstgutachter: Prof. Dr. Wolfgang E. Nagel
Zweitgutachter: Prof. Dr. habil. Wolfgang V. Walter
Betreuer: Dr. Andreas Knüpfer, Dr. Michael Bussmann, Dr. Thomas Kluge

Dresden, 28. September 2016

Hier Aufgabenstellung einfügen!

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Elektrotechnik und Informationstechnik eingereichte Diplomarbeit zum Thema:

A highly-parallel Monte-Carlo-Simulation of X-Ray-Scattering using a Particle-Mesh-Code on GPUs

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 28. September 2016

Alexander Grund

Kurzfassung

Im Rahmen dieser Arbeit wurde eine Softwarelösung entwickelt, welche die Streuung von Röntgenstrahlung in Materie mit einem Monte-Carlo-Ansatz simuliert. Dazu wurde die Anwendung der Röntgen-Kleinwinkelstreuung (SAXS) zur Untersuchung der komplexen Prozesse bei der Interaktion intensiver kurzer Laserpulse mit Festkörpern als Motivation verwendet und beschrieben, wie Fouriertransformationen zur Näherung dieser Streuung verwendet werden können. Darauf aufbauend wurde einerseits die schnelle Fouriertransformation (FFT) als effiziente Implementierung für Computer vorgestellt und andererseits wurde auf die Limitierung dieses Ansatzes zur Beschreibung der Streuprozesse eingegangen. Um diese Limitierungen zu umgehen, wurde ein Modell entwickelt, das die Röntgenstrahlung mittels photonenähnlicher Teilchen beschreibt. Da für eine gute Abbildung der physikalischen Prozesse Milliarden solcher Teilchen benötigt werden, wurde die auf diesem Modell basierende Simulation von Anfang an auf die hoch parallele Struktur moderner Grafikprozessoren ausgelegt, welche es ermöglicht, sehr viele Teilchen gleichzeitig zu simulieren. Der implementierte Algorithmus wurde detailliert beschrieben, wobei gezielt auf die Besonderheiten von Grafikprozessoren eingegangen wurde. Da die richtige Wahl der Datentypen wesentlich für die Geschwindigkeit und Präzision des Algorithmus ist, wurde in einer umfassenden theoretischen Analyse und Tests der numerischen Genauigkeit der Implementation gezeigt, dass sogar mit Berechnungen in geringer Genauigkeit Ergebnisse erzielt werden, die keine wesentlichen Abweichungen von denen der exakten Berechnung aufzeigen. Dadurch können für typische Anwendungen kleinere Datentypen gewählt werden, was die Durchführung umfangreicherer Simulationen auf einer gegebenen Hardware erlaubt. Abschließend konnte die Korrektheit für ausgewählte Beispiele sowie eine gute Skalierbarkeit nachgewiesen werden.

Abstract

In this thesis a software solution was developed that simulates the scattering of X-rays in matter using a Monte Carlo approach. The application of small-angle X-ray scattering in the studies of the complex processes occurring during the interaction of short intense laser pulses in solid matter provides the motivation for this work. Therefore this technique is described and it is shown how Fourier transformation can be used for approximating the scattering results. It is shown how they can be efficiently implemented in computers using the fast Fourier transform (FFT) and why this approach has limitations when describing scattering processes. To circumvent these, a model was developed that uses photon-like particles to describe the X-rays. Billions of such particles are required to provide a good approximation of the physical processes involved, which is why the simulation algorithm described in this work was designed from the ground up to support the massively parallel structure of modern graphic processing units (GPUs) allowing to simulate many particles at once. The implemented algorithm is described focusing on the special methods required to make the most use out of GPUs. As the choice of the appropriate data types is vital for the correctness and precision of the algorithm a comprehensive analysis and test of the numerical accuracy was deployed. It is shown that even reduced precision provides results that are accurate enough for a wide range of applications. Therefore, smaller data types can be used allowing to simulate much larger experiments on a given hardware. Finally the correctness and good scalability of the parallel algorithm are demonstrated.

Inhaltsverzeichnis

Symbolverzeichnis

AoS	Array of Structs
CPU	Central Processing Unit - Hauptprozessor
CUDA	Compute Unified Device Architecture - Plattform und Programmiermodell für NVIDIA Grafikkarten
FFT	Fast Fourier Transformation - Schnelle Fourier Transformation
FFTW	Fastest Fourier Transformation in the West - Populäre, freie Implementierung der FFT
FLOPS	Floating point operations per second, auch mit Vorsatz wie: 1 GFLOPS = 10^9 FLOPS
FMA	Fused Multiply-Add - Ausführung von einer Addition und einer Multiplikation der Form $a \cdot x + b$ ohne Zwischenrundung
GPU	Graphics Processing Unit - Grafikkarte
MQF	Mittlerer quadratischer Fehler
PIConGPU	Framework zur Simulation von Particle-in-Cell Algorithmen auf GPUs
PMMA	Polymethylmethacrylat auch Acrylglas
SAXS	Small Angle X-ray Scattering
SIMD	Single Instruction, Multiple-Data - Meist: Vektoroperationen
SIMT	Single Instruction, Multiple Threads - Ausführungsmodell in CUDA
SoA	Struct of Arrays

1 Einleitung

1.1 Motivation und Zielstellung

Die Wechselwirkung extrem intensiver kurzer Laserpulse mit Festkörpern verspricht einige interessante Anwendungen und Einsichten in grundlegende Fragen der Plasmaphysik. Eine Anwendung besteht darin, schnelle, durch die Laser-Plasma-Wechselwirkung erzeugte Ionen zur schonenderen und zielgerichteteren Behandlung von Krebspatienten zu nutzen, als das mit klassischer Photonen-Strahlentherapie möglich wäre. „Die Ionenstrahlung hat besondere physikalische Eigenschaften, durch die sie der herkömmlichen Photonenbestrahlung überlegen [ist].“ [1] Durch diese kann „[...] die Dosis im Innern des Tumors im Vergleich zur Photonenbestrahlung deutlich [...]“ [1] erhöht werden, während umliegendes gesundes Gewebe weniger geschädigt wird. Insgesamt kann so die Wirksamkeit der Behandlung gesteigert werden, während gleichzeitig weniger Nebenwirkungen auftreten.

Die Ionenstrahlen selbst können experimentell leicht untersucht werden, jedoch bleiben die Prozesse, durch welche sie entstehen, aufgrund der sehr kurzen Zeit- und Raumskalen sowie der Undurchdringlichkeit von Festkörpern für sichtbares Licht nur schwer zugängig. Röntgenstreuexperimente werden als mögliche Lösung gesehen, wobei Röntgenquellen mit extrem hohen Leuchtstärken benötigt werden (vgl. [2]). Zu deren Erzeugung werden Freie-Elektronen-Laser (*FEL*; bei Röntgenstrahlung: *XFEL*) eingesetzt, die stetig weiter verbessert werden (vgl. [3, S. 1 ff.]). Diese Diplomarbeit soll u. a. als Vorbereitung für zukünftige Experimente an dem *European XFEL* dienen, der 2017 in (Nutzer-)Betrieb gehen und Spitzenwerte in der Leuchtstärke liefern wird, die „[...] milliardenfach höher als die der besten herkömmlichen Röntgenquellen[...]“ [4] ist. Auch werden mit diesem XFEL kurze Lichtpulse erzeugt, die eine hohe zeitliche Auflösung ermöglichen. Die vorliegende Arbeit widmet sich der parallelen Computersimulation dieser Experimente, womit diese vorbereitet und später ausgewertet werden können.

Die bislang hauptsächlich eingesetzte Methode zur Analyse der Streuung von Röntgenstrahlen ist eine Näherung, die auf der Fouriertransformation basiert und sich in das Gebiet der Fourieroptik einordnet [5, S. 170 ff., 6]. Dadurch bleiben jedoch wichtige physikalische Prozesse, wie mehrfache Streuung, zeitveränderliche Dichten und Laserprofile unbeachtet.

Im Rahmen dieser Arbeit wird daher eine Softwarelösung entwickelt, in der Propagation und Streuung der Röntgenstrahlung in einer Probe mit Monte-Carlo-Methoden simuliert werden, womit sich prinzipiell die vollen physikalischen Elementarprozesse berücksichtigen lassen. Durch die Nutzung von GPUs und eines skalierbaren Ansatzes auf Basis der Bibliothek *libPMacc*[7] können auch große Proben und Leuchtstärken verarbeitet werden. Da die numerische Genauigkeit eine große Rolle bei der Auswahl der Datentypen spielt, die wiederum die Geschwindigkeit der Simulation wesentlich beeinflussen, wird diese umfassend analysiert. Anhand dieser Analyse werden die jeweils geeignetsten Lösungen vorgestellt, implementiert und getestet.

1.2 Aufbau des Experiments

In einem typischen Anwendungsfall wird eine Folie von einem Hochleistungslaser bestrahlt, wodurch diese ionisiert wird und freie Elektronen an der Vorderseite der Folie entstehen, die durch die starken Felder des Lasers durch die Folie getrieben werden. Sie ionisieren bei ihrem Durchflug die Atome in der Folie, bevor sie an der Rückseite der Folie wieder austreten. Dort bilden sie eine negative Ladungswolke („Debye sheath“ [8]), welche die positiven Ionen auf der Rückseite anzieht und beschleunigt („target normal sheath acceleration“, *TNSA* [8]). Hierdurch entsteht ein Ionenstrahl, dessen Eigenschaften von der genauen Wechselwirkung der schnellen Elektronen mit dem Target abhängt (vgl. [9, 10]). Aufgrund dieser Tatsache erfolgt die Betrachtung der Prozesse, welche die Ionen erzeugen und beeinflussen, über die Analyse der Elektronenverteilung. Hierbei wird ausgenutzt, dass Röntgenstrahlen an Elektronen streuen und unterschiedliche Elektronendichten verschieden starke Streuung verursachen.

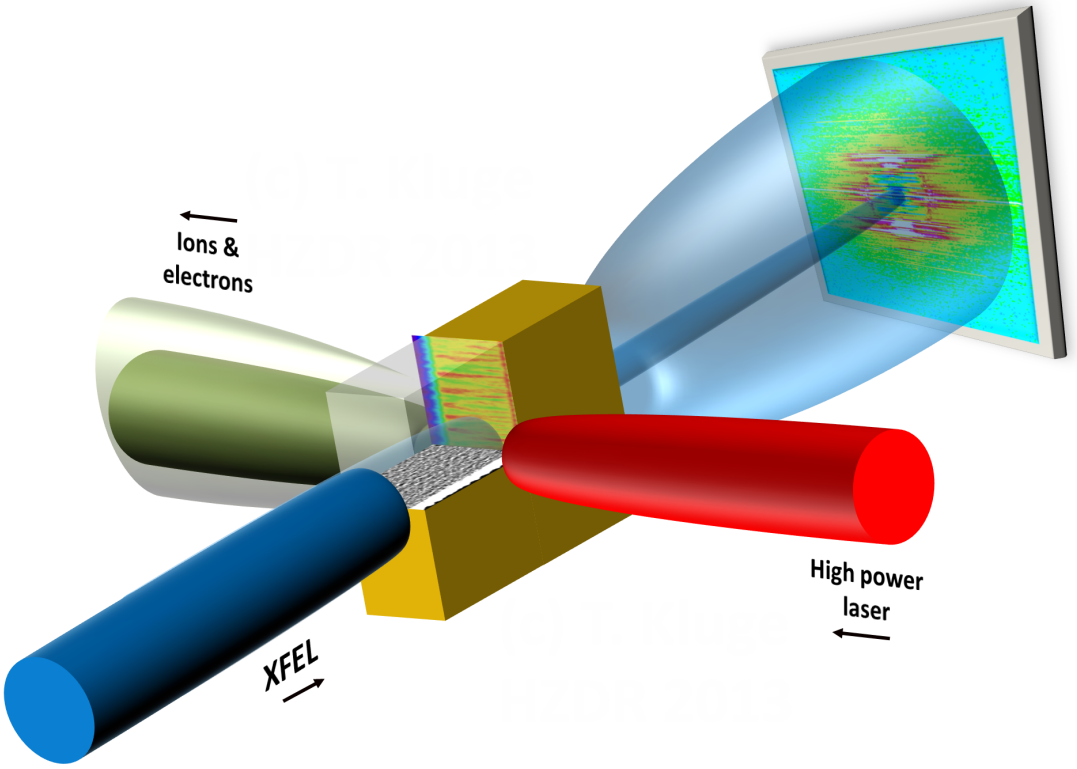


Abbildung 1.1: Aufbau des Experiments (entnommen aus [11])

Den schematischen Aufbau eines solchen Experiments zeigt Abb. 1.1. Der Hochleistungslaser (rot) trifft auf die Folie (golden) und erzeugt dadurch Felder und schnelle Elektronen und Ionen (grün), welche die Folie auf der gegenüberliegenden Seite verlassen. Gleichzeitig wird eine starke, kohärente Röntgenstrahlung (blau), wie sie ein XFEL erzeugt, senkrecht (oder in einem festen Winkel) dazu durch die Folie geschickt. Diese Röntgenstrahlung wird gestreut, wodurch verschiedene Wellen entstehen, die miteinander interferieren (s. Abschnitt 3.2). Das entstehende Muster in der Intensitätsverteilung kann auf einem Detektor, der in XFEL-Richtung hinter der Folie aufgestellt ist, gemessen werden. Da die Streuung von der Elektronendichte abhängt, ist ein Rückschluss von dem Muster auf die Elektronendichte möglich. Diese Technik wird aufgrund der Betrachtung kleiner Streuwinkel als *Small Angle X-Ray Scattering*

(kurz: *SAXS*) bezeichnet und ist bereits länger bekannt, erforscht und eingesetzt (vgl. [12]). Neu sind hier die kurzen Zeitskalen und die hohen Intensitäten, die notwendig sind, um überdichte Plasmen zu durchdringen und ausreichend aufzulösen, was erst durch den *European XFEL* sinnvoll realisiert werden kann. Damit sind zur Zeit noch keine praktischen Experimente dieser Art möglich.

1.3 Aufbau und Einordnung der Arbeit

Die im Rahmen dieser Arbeit entwickelte Computersimulation modelliert die Interaktion von Röntgenstrahlung mit einer gegebenen Dichteverteilung und deren Detektion. Auf das in Abschnitt 1.2 beschriebene Experiment bezogen beschränkt sich die Betrachtung auf den Röntgenlaser, die Elektronendichte und den Detektor. Die Dichte wird dabei als gegeben angesehen und kann z. B. durch andere Simulationen, wie den Particle-In-Cell Code *PIConGPU* [7, 13], vorgegeben werden. Da sich eine Integration der hier erstellten Simulation in PIConGPU geplant ist, wird großer Wert auf die Kompatibilität der verwendeten Strukturen und Datenformate gelegt. Die Integration selbst ist jedoch nicht Teil dieser Arbeit. Der Fokus beim Entwurf des Programms liegt auf der Bereitstellung eines skalierbaren und erweiterbaren Grundgerüsts, welches die grundsätzlichen Prozesse der Bewegung und der Streuung von Photonen beschreibt. Die konkrete Dimensionierung der Größen (Dichte, maximale Streuwinkel und Wahrscheinlichkeiten) kann hier nur am Rand behandelt werden. Es ist möglich, sowohl Einfach- als auch Mehrfachstreuung ohne Energieverlust (Thomson-Streuung, s. Abschnitt 3.2) zu simulieren, wobei auch komplexere physikalische Prozesse bei der Interaktion der Röntgenstrahlung mit der Elektronendichte auf dieser Basis implementierbar sind. Damit ähneln die Ergebnisse denen, die auch über die Fouriertransformation bestimmt werden können, der Ansatz ist jedoch deutlich allgemeiner und kann zusätzliche Effekte berücksichtigen. Die Basis des Algorithmus bildet die Monte-Carlo Methode, nach der das Ergebnis durch zufälliges Sampling mit sehr vielen Samples bestimmt wird. Dieser Ansatz wurde schon mehrfach für Photonen-Propagation und Röntgenstreuung verwendet [14, 15], wobei teilweise Mehrfachstreuung berücksichtigt [16] und auch Grafikarten zur schnelleren Berechnung verwendet wurden [17, 18]. Die Anwendungen sind jedoch verschieden. Bei Badal und Badano [18] werden Röntgenaufnahmen des Menschen mit deutlich größeren Skalen und geringeren Intensitäten betrachtet, als es bei den hier betrachteten Experimenten der Fall ist. Alerstam et al. [17] nehmen hauptsächlich konstante Dichten an, die hier nicht gegeben sind, und auch dort sind die Skalen deutlich größer. Auch die anderen referenzierten Quellen nutzen Annahmen, die für die in dieser Arbeit betrachteten Experimente nicht zutreffen oder deutliche Einschränkungen bilden, obwohl die physikalischen Grundlagen dieselben sind. Die Hauptunterschiede der in dieser Arbeit entwickelten Computersimulation zu den bislang existierenden Lösungen liegen in der sehr allgemeinen Formulierung, mit der sich nahezu beliebige Geometrien und Prozesse beschreiben lassen, in der hohen Skalierbarkeit, wodurch sehr große oder fein aufgelöste Strukturen betrachtet werden können, sowie der hohen Auflösung der Probe.

In den nachfolgenden Kapiteln soll zunächst die Simulation an sich sowie die dazu benötigten Grundlagen erörtert werden. Dazu gehe ich in Kapitel 2 zuerst auf die Fouriertransformationen ein und erkläre wie diese mit Computern effizient berechnet werden können. Außerdem zeige ich an ausgewählten Beispielen die Technik der Template Metaprogrammierung, die in der Simulation verwendet wird. Anschließend folgt in Kapitel 3 die Betrachtung der physikalischen Grundlagen im Zusammenhang mit Wellen, deren Streuung und Interferenz. Hier wird gezeigt, wie die Fouriertransformation aus Kapitel 2

verwendet werden kann, um die Intensitätsverteilung nach der Streuung zu berechnen. Den Abschluss des Kapitels bildet eine Diskussion der Mehrfachstreuung wobei ich näher auf die Motivation eingehe, eine Monte-Carlo-Simulation für diesen Fall zu verwenden. Zu Beginn von Kapitel 4 befasse ich mich speziell mit Grafikkarten und dem Monte-Carlo-Ansatz allgemein. Aufbauend auf den in Kapitel 2 und 3 erläuterten Grundlagen stelle ich in den Kapiteln 4.2 und 4.3 die Simulation, die verwendeten Algorithmen und die Designentscheidungen vor. Danach zeige ich in Abschnitt 4.4 eine wichtige Optimierung des verwendeten Zufallszahlengenerators, welche die Laufzeit enorm reduziert. Schließlich stelle ich die eingesetzten Datenformate in Abschnitt 4.5 vor. Nach der Analyse der numerischen Genauigkeit in Kapitel 5 stelle ich fest, dass für die meisten Anwendungsfälle Berechnungen in einfacher Genauigkeit ausreichen, was ich in Kapitel 6 an ausgewählten Beispielen demonstriere. Abschließend werden die Erkenntnisse der vorliegenden Arbeit in Kapitel 7 zusammengefasst und Anregungen für weiterführende Untersuchungen gegeben.

Diese Arbeit ist in Zusammenarbeit mit dem Helmholtz-Zentrum Dresden-Rossendorf (HZDR) entstanden, das auch den Rechencluster *hypnos* betreibt, der zur Simulation der meisten in Kapitel 6 gezeigten Ergebnisse verwendet wurde. Weitere Simulationen wurden auf dem *Taurus* Cluster des ZIH (Zentrum für Informationsdienste und Hochleistungsrechnen) der TU Dresden durchgeführt.

2 Fast Fourier Transformation

In Abschnitt 3.2 wird gezeigt werden, dass in ausreichender Entfernung von dem Streuprozess die Intensitätsverteilung mit Hilfe von Fouriertransformationen näherungsweise ermittelt werden kann. Da dies auch später zur Validierung der Simulationsergebnisse verwendet wird, werden in diesem Teil der Diplomarbeit Fouriertransformationen und schnelle Fouriertransformationen, üblicherweise als „FFT“ (engl. für „Fast Fourier Transformation“) bezeichnet, betrachtet.

2.1 Mathematische Grundlage

Die Fouriertransformation wird normalerweise im Zusammenhang mit der Transformation einer über der Zeit definierten Funktion in eine über Frequenzen definierte Funktion verwendet. Diese hat spezielle Eigenschaften, die im Weiteren ausgenutzt werden können. Insbesondere ist es möglich aus der einen die jeweils andere Darstellung zu ermitteln, weshalb die mathematische Definition der Fouriertransformation in der Regel mit einem Gleichungspaar erfolgt. Leider ist diese nicht einheitlich und so finden sich in der Literatur verschiedene Angaben, die sich meist nur in den Vorfaktoren unterscheiden. Jedoch gibt es auch Autoren, die statt der Kreisfrequenz ω die Frequenz f verwenden, die über $\omega = 2\pi f$ zusammenhängen. So verwendet Zinth und Zinth [5, S. 323] die Form:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t \tag{2.1}$$

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\mathrm{e}^{\mathrm{i}\omega t}\,\mathrm{d}\omega \tag{2.2}$$

Hierbei ist i die imaginäre Einheit. Die gleiche Form wird auch in Stößel [6, S. 19, 27] verwendet. Es lässt sich zeigen, dass auch die folgende Definition möglich ist, welche die Symmetrie der Vor- und Rücktransformation deutlicher macht:

$$F(\omega) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(t)\mathrm{e}^{-\mathrm{i}\omega t}\,\mathrm{d}t \tag{2.3}$$

$$f(t) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} F(\omega)\mathrm{e}^{\mathrm{i}\omega t}\,\mathrm{d}\omega \tag{2.4}$$

Dies zeigt, dass die Festlegung, welches von beiden die „Hin-“ bzw. „Rücktransformation“ (auch als „Fouriertransformation“ und „inverse Fouriertransformation“ bezeichnet) ist, arbiträr ist. Es ist damit möglich sowohl die Vorzeichen im Exponenten als auch den Vorfaktor in den Gleichungen (2.1) und (2.2) zu tauschen (siehe dazu z. B. die Diskussion von Kaiser [19, S. 32] und Rahman [20, S. 11]).
Auf die vielfältigen Sätze zur Fouriertransformation sowie die notwendigen Voraussetzungen soll in dieser Arbeit nicht weiter eingegangen werden. Hervorzuheben ist jedoch, dass sich die Fouriertransformation in ähnlicher Form auch auf Funktionen über Vektoren anwenden lässt. Außerdem ist eine Rechenregel der Fourieranalyse für diese Arbeit relevant: Die Faltungsregel ermöglicht es, eine Faltung

im Zeitbereich (der Domain von f) durch eine Multiplikation im Frequenzbereich (der Domain von F, auch „Fourierraum“) auszudrücken (vgl. [5, S. 327]). Formal bedeutet das:

$$\mathcal{F}(f(t) \otimes g(t)) = \mathcal{F}(f(t)) \cdot \mathcal{F}(g(t)) \tag{2.5}$$

Dabei bezeichnet $\mathcal{F}$ die Fouriertransformation während die Faltung zweier Funktionen durch den Operator $\otimes$ repräsentiert wird, dessen Definition lautet:

$$f(t) \otimes g(t) = \int_{-\infty}^{\infty} f(\tau) \cdot g(t-\tau)\,\mathrm{d}\tau \tag{2.6}$$

Es ist anzumerken, dass die Formalismen auch gelten, wenn keine zeitabhängige Funktion betrachtet wird, sonder eine, die z. B. vom Ort abhängt. Da die analytische Integration nur für einfache Funktionen bzw. Geometrien sinnvoll machbar ist, werden die Transformationen mit Hilfe von Computern auf diskretisierten Werten durchgeführt und insbesondere nur diskretisierte Frequenzen betrachtet. Dies führt zur Diskreten Fourier Transformation (DFT), welche formal durch das folgende Gleichungspaar[1] (vgl. [20, S. 149]) ausgedrückt wird und periodisch mit der Periodenlänge N ist.

$$F(k) = \sum_{n=0}^{N-1} f(n) \mathrm{e}^{\frac{-2\pi i n k}{N}} \tag{2.7}$$

$$f(n) = \frac{1}{N} \sum_{k=0}^{N-1} F(k) \mathrm{e}^{\frac{2\pi i n k}{N}} \tag{2.8}$$

Das sich ergebende Spektrum $F(k)$ ist an diskreten Punkten k (ganzzahlig) definiert, deren Bereich üblicherweise entweder als $k \in [0, N)$ oder als $k \in [-\frac{N}{2}, \frac{N}{2})$ (für geradzahlige N) angegeben wird. Beide Formen sind durch die Periodizität äquivalent und beide werden sowohl in der Literatur als auch in verschiedenen Softwarebibliotheken verwendet. Das „Sampling“ (Abtastung oder Diskretisierung) erfolgt dabei in gleichen Abständen, sowohl im Fourier- als auch im Realraum.

Die DFT kann als Spezialfall der (kontinuierlichen) Fouriertransformation bezeichnet werden. Dabei lässt sich die eine aus der jeweils anderen herleiten und wichtige Eigenschaften, wie z. B. der Faltungssatz und die Umkehrbarkeit der Transformation, gelten auch für die DFT. Auch gibt es hier mehrere Konventionen bezüglich des Vorzeichens des Exponenten und des Vorfaktors in den Gleichungen (2.7) und (2.8). Die Verwendung von Funktionen über Vektoren ist ebenfalls möglich. Es ist wichtig zu beachten, dass bei der DFT die Funktion $f(n)$ als periodisch oder periodisch fortgesetzt angenommen wird, was bei der kontinuierlichen Fouriertransformation nicht zwingend der Fall ist. Für weitere Informationen sei hier auf die umfangreichen Behandlungen in der Literatur (z. B. [21]) verwiesen, da die diskrete und kontinuierliche Fouriertransformation in vielen Bereichen, wie der Signalanalyse in der Informationstheorie und Elektrotechnik, als auch in vielen physikalischen Untersuchungen eine wichtige Rolle spielt.

[1] Die Notation wurde gegenüber Rahman [20] geringfügig geändert, um F konsistent als (diskrete) Fouriertransformierte für f zu verwenden. Auch wird $F(k)$, F_k oder X_k als Koeffizienten der Fouriertransformierten häufiger verwendet und deshalb u. U. vertrauter.

2.2 Berechnung mittels Computer

Mit den Rechenvorschriften nach Gleichung (2.7) bzw. (2.8) ist die Implementation der diskreten Fouriertransformation in Software sehr leicht möglich. Häufig muss jedoch eine große Anzahl N an Samples verarbeitet werden. Außerdem bildet die Fouriertransformation meist nur den Grundbaustein für komplexe Algorithmen und wird darin häufig aufgerufen, weshalb sie schnell zum limitierenden Faktor für die Geschwindigkeit des Algorithmus wird. Betrachtet man die Komplexität der direkten Implementation von Gleichung (2.7) sieht man, dass für jeden der N Koeffizienten $F(k)$ jeweils einige N Rechenoperationen ausgeführt werden müssen. Das ergibt einen Rechenaufwand der in $O(N^2)$ liegt.
Wenn N sich in (ganzzahlige) Faktoren zerlegen lässt, kann dieser Aufwand durch Anwendung des Divide-And-Conquer (Teile-Und-Herrsche) Prinzips reduziert werden. Das ist besonders effizient, wenn N eine Zweier-Potenz ist (vgl. [22, S. 609 f.]). Der zugrunde liegende Algorithmus wird „Schnelle Fouriertransformation“ (englisch: „Fast Fourier Transformation“, FFT) genannt und wurde u. a. 1965 von Cooley und Tukey [23] beschrieben. Die Komplexität liegt hierbei in $O(N\log(N))$, was eine deutliche Verbesserung gegenüber $O(N^2)$ ist. Es existieren eine Vielzahl von Algorithmen, die auf dem gleichen Prinzip basieren und teils ebenfalls „FFT“ (oft mit Namenszusätzen) genannt werden, wobei sie für bestimmte Fälle optimiert sind (vgl. [22, S. 616]). Einige Optimierungen werden durch Kenntnis der Eingabedaten möglich. So kann z. B. die Symmetrieeigenschaft der Fouriertransformation bei reellen (nicht komplexen) Eingabewerten (vgl. [21, S. 636]) ausgenutzt werden, um den Rechenaufwand zu halbieren, da lediglich $\frac{N}{2}+1$ Werte berechnet werden müssen. Die andere Hälfte kann durch Spiegelung an der Grundfrequenz ($k=0$) über $F(k)=F^*(-k)$ gewonnen werden, wenn nötig.
Es existieren viele Softwarebibliotheken, welche die FFT implementieren. Beispiele hierfür sind FFTW [24], cuFFT [25], Intel MKL [26] und clFFT [27], die als Teil der „clMath“ Bibliothek (vormals APPML) 2013 von AMD veröffentlicht wurde (vgl. [28]). Einige davon sind nur auf bestimmten Architekturen lauffähig oder auf diesen besonders performant. Außerdem unterscheiden sich die Implementationen in ihrem Interface und der Behandlung der Vorfaktoren. Es ist darum nicht (einfach) möglich, eine Bibliothek durch eine andere zu ersetzen, was jedoch für plattformunabhängigen Code wünschenswert ist, der je der nach Architektur eine andere Bibliothek verwenden kann. Eine Lösung dafür soll im folgenden Kapitel behandelt werden.

2.3 Template Metaprogrammierung und ein allgemeines FFT Interface

Gemäß dem Zitat „All problems in computer science can be solved by another level of indirection“, das David Wheeler zugeordnet wird (vgl. [29]), wurde zu Beginn dieser Arbeit eine Bibliothek[2] entwickelt, die eine Abstraktion der zu Grunde liegenden FFT-Bibliothek bietet. Die Motivation war die Schaffung einer Möglichkeit, die auszuführende FFT so zu beschreiben, dass ein allgemeines Interface diese Beschreibung auf Aufrufe einer existierenden FFT-Bibliothek abbilden kann. Später soll die verwendete Datenbeschreibung genutzt werden, um auf ähnliche Weise lineare Operationen (wie Matrixmultiplikationen) abstrakt zu beschreiben und auf spezielle Bibliotheken abzubilden. Auch soll an diesem Beispiel

[2] Im weiteren Verlauf das Kapitels soll „Interface“ für die entwickelte Bibliothek verwendet werden, um den Begriff von der FFT-Bibliothek, die abstrahiert wird, abzugrenzen.

die Template Metaprogrammierung eingeführt werden.
Das entwickelte Interface, was im Folgenden kurz vorgestellt werden soll, ist ausreichend allgemein, um die meisten Use-Cases abzubilden. Dabei ist die Wahl der verwendeten Bibliothek (als „Backend“ bezeichnet), welche die FFT am Ende ausführt, frei. Es muss lediglich eine Abbildung des Interfaces auf das Backend existieren oder geschrieben werden. Der Wechsel zwischen den Bibliotheken erfolgt durch die Änderung eines Parameters anstatt von zahlreichen Änderungen im Nutzer-Code. Das Interface setzt das Adapter Entwurfsmuster[3] um und genügt folgenden Anforderungen:

- Keiner oder sehr geringer Overhead im Vergleich zur direkten Verwendung des Backends
- Identische Verwendung für alle Backends
- Einfache Erweiterbarkeit bezüglich zusätzlicher Backends
- Erkennung von Fehlern und Inkonsistenzen u. a. durch Typsicherheit

Diese Anforderungen konnten mittels C++ Template Metaprogrammierung (TMP) erfüllt werden, welche auch später für die Simulation genutzt wird. Sie nutzt dabei die C++ Templates, um Code zur Compilezeit[4] zu erzeugen und/oder auszuführen. Ein Beispiel dazu ist in Listing 2.1 gegeben, in dem die

```
template<unsigned N>
struct Sum{
  static constexpr unsigned value = N + Sum<N − 1>;
};
template<>
struct Sum<0>{
  static constexpr unsigned value = 0;
};
int main(){
  return Sum<10>::value;
}
```

Listing 2.1: Summation mit Template Metaprogrammierung

Summe der Zahlen von eins (bzw. null) bis N berechnet wird. Zwei wichtige Aspekte werden hier verdeutlicht:

1. Die Form entspricht der, die man sonst in funktionalen Programmiersprachen findet. Darin werden Algorithmen ausschließlich über Funktionen realisiert, die zur Laufzeit ausgewertet werden. Das führt zu einem hohen Grad an Rekursion, was auch in dem Beispiel ersichtlich wird, in dem die Summe über die ersten N Zahlen als Addition von N und der Summe über die ersten $N-1$ Zahlen ausgedrückt wird, wobei $f(0) = 0$ die Abbruchbedingung bildet.
2. Die komplette Summation wird zur Compilezeit ausgeführt, d. h. der Ausdruck *Sum<10>::value* wird vollständig ausgewertet und ist im übersetzten Programm lediglich eine Konstante. Dies trägt wesentlich zu dessen Geschwindigkeit bei, da durch TMP Berechnungen zur Laufzeit eingespart und weitere Optimierungen ermöglicht werden können.

[3] Eine allgemeine Beschreibung des Adapter Entwurfsmusters kann u. a. in [30, S. 241 ff.] gefunden werden.
[4] Hier ist mit „Compilezeit“ die Übersetzung des Programms von einer Programmiersprache wie C++ in Maschinensprache mittels eines Compilers gemeint, wobei sie von der „Laufzeit“, also der eigentlichen Ausführung des übersetzten Programms unterschieden wird.

Eine weitere wichtige Anwendung ist die Definition von „Policies“ (vgl. [31, S. 8 ff.]), womit hier Klassen bezeichnet werden, die ein bestimmtes Verhalten definieren. Ein Beispiel dazu zeigt Listing 2.2, in

```
struct Add{
  int operator()(int a, int b){
    return a + b;
  }
};
struct Mul{
  int operator()(int a, int b){
    return a * b;
  }
};
template<class T_Policy>
struct DoSth{
  int operator()(int a, int b){
    return T_Policy()(a, b);
  }
};
int main(){
  return DoSth<Add>()(5, 6); // Ausgabe: 11
  // Oder:
  return DoSth<Mul>()(5, 6); // Ausgabe: 30
}
```

Listing 2.2: Policies in der Template Metaprogrammierung

dem die Klasse `DoSth` je nach übergebener Policy eine andere Operation ausführt. Im Unterschied zur Fallunterscheidung mittels `if-else` entsteht hier kein Overhead zur Laufzeit, da die Entscheidung, welche Operation ausgeführt wird, bereits zur Compilezeit getroffen wurde. Darüber hinaus wird für die nicht verwendetet Operation gar kein Code generiert, was vielfältige Anwendungsmöglichkeiten bietet. So kann z. B. basierend auf den Eigenschaften eines Objekts ein Algorithmus ausgewählt oder ein angepasster Algorithmus aus solchen „Bausteinen“ zusammen gesetzt werden, der ein bestimmtes Problem löst. Ein weiterer Vorteil gegenüber Laufzeitentscheidungen, durch die sich theoretisch auch eine solche Zusammensetzung lösen lassen würde, ist, dass nicht alle Policies mit allen möglichen Eingabetypen umgehen können müssen. Da kein Code generiert wird, wenn eine Policy nicht verwendet wird, werden in solchen Fällen keine Fehler erzeugt. Eine Lösung, die solche Entscheidungen zur Laufzeit trifft kann dies nicht, oder müsste das Typsystem von C++ umgehen.

Das entwickelte Interface nutzt die Möglichkeiten der Template Metaprogrammierung. Aus Anwendersicht übergibt er das zu nutzende Backend als Policy, welche die eigentliche Abbildung auf die Bibliothek durchführt. Aufgrund der Auflösung dieser Abstraktion zur Compilezeit entsteht kein Overhead zur Laufzeit gegenüber der nativen Nutzung der zugrunde liegenden Funktionen. Außerdem ist es durch die Informationen, die nun zur Compilezeit genutzt werden können, möglich, Fehler schon hier zu erkennen. Beispiele für solche sind unterschiedliche Dimensionalitäten oder Größen der Eingangs- und Ausgangsdaten oder die Verwendung von komplexen Daten an Stellen, wo reelle Daten erwartet werden (oder umgedreht). Jedoch können nicht alle Bedingungen an die Eingabewerte zur Compilezeit geprüft werden und auch eventuell notwendige Änderungen des Datenlayouts erzeugen Overhead, wobei letzterer

auch bei der direkten Verwendung auftritt, wenn das Datenlayout der Eingabedaten fest ist.
Möchte der Nutzer nun eine andere Bibliothek nutzen, muss dieser lediglich die Policy austauschen, was meist nur eine einzige Stelle im Code betrifft. Außerdem ist es möglich, Funktionalität abzubilden, die nur in manchen Backends existiert, oder die Abbildung des Interfaces auf das Backend schrittweise zu implementieren. So könnte man eine Policy schreiben, die nur eine Art der Fouriertransformation abbildet. Solange die nicht implementierten Transformationen nicht verwendet werden bzw. dafür andere Backends, welche die Funktionalität bieten, genutzt werden, führt das Fehlen nicht zu Problemen. Andernfalls erhält man hier entsprechende Meldungen zur Compilezeit, was besser ist, als diese erst zur Laufzeit zu bekommen, wenn z. B. schon längere Berechnungen abgeschlossen sind. Auch das ist ein Vorteil der TMP: Viele Fehler können schon zur Compilezeit erkannt werden. Als Nachteil muss hier erwähnt werden, dass die Komplexität durch die Metaprogrammierung stark steigt. Insbesondere da Fehlermeldungen teilweise sehr komplex und dadurch schwierig zu interpretieren sind und auch kein Debugger zur Verfügung steht, ist die Verwendung eine große Hürde für Einsteiger. Die bereits genannten Vorteile (Typsicherheit, gute Laufzeiten, Modularität, . . .) sind aber nicht zu unterschätzen.
Das hier angesprochene Interface ist unter [32] zu finden und unterstützt zur Zeit die FFTW und cuFFT. Ein Backend für clFFT ist in Vorbereitung und wird sicherlich zeitnah verfügbar sein. Ein Beispiel für

```
void do2D_FFT(const string& inFilePath, const string& outFilePath){
    using namespace LiFFT; using namespace tiffWriter;
    // Festlegung des verwendeten Backends
    // alternativ: using FFT_LIB = libraries::fftw::FFTW<>;
    using FFT_LIB = libraries::cuFFT::CuFFT<>;
    using FFT = FFT_2D_R2C_F<>;
    // Lesen der Eingabe- und Meta-Daten
    auto input = FFT::wrapInput(FloatImage<>(inFilePath));
    // Erzeugung des benötigten Ausgabecontainers basierend auf der auszuführenden FFT
    auto output = FFT::createNewOutput(input);
    // Erzeugen des FFT-Funktors
    auto fft = makeFFT<FFT_LIB>(input, output);
    // Ausführung der FFT
    fft(input, output);
    // Anlegen des Ausgabe-Bildes
    FloatImage<> outImg(outFilePath, input.getBase().getWidth(), input.getBase().getHeight());
    auto fullOutput = types::makeSymmetricWrapper(output, input.getExtents()[1]);
    auto transformAcc = accessors::makeTransposeAccessor(
                            accessors::makeTransformAccessorFor(policies::CalcIntensityFunc(),
                                fullOutput)
                        );
    // Kopieren (mit Konvertierung) der Ergebnisse in das Ausgabe-Bild
    policies::copy(fullOutput, outImg, transformAcc);
    outImg.save();
}
```

Listing 2.3: Fouriertransformation einer als TIFF-Bild gegebenen Dichte und Ausgabe der Intensität im TIFF-Format

die Verwendung ist in Listing 2.3 zu finden, in dem reelle 2D-Daten aus einem Bild geladen und mittels einer 2D-FFT in einfacher Genauigkeit transformiert werden. Wie schon erwähnt wird von der Biblio-

thek nur die Hälfte der N Werte generiert, da das Ergebnis aufgrund der reellwertigen Eingabedaten symmetrisch ist. Mittels einem in dem Interface enthaltenen „Wrapper“ wird dies abstrahiert und so dargestellt, als wären alle Daten vorhanden. Nützliche Zugriffsfunktoren ermöglichen es Umformungen direkt beim Zugriff auf das Element vorzunehmen, was hier genutzt wird um die Intensität (definiert als das Betragsquadrat der komplexen Werte) zu berechnen sowie die Grundfrequenz in die Mitte der Daten zu verschieben. Das Ergebnis wird schließlich als TIFF-Bild im Gleitkommaformat gespeichert und kann zur weiteren Analyse verwendet werden. An diesem Beispiel sieht man die Stärken des Interfaces: Daten und Metadaten, wie Größe, Dimensionalität und Datenlayout, liegen zusammen in sogenannten Containern vor und müssen nicht in jedem Schritt wieder angegeben werden. Die Art der Transformation wird vollständig spezifiziert (in diesem Fall[5]: 2D, Reell zu Komplex, Vorwärtstransformation, einfache Genauigkeit), und die Daten werden vom Interface überprüft, wobei entsprechende Fehlermeldungen erzeugt werden, wenn z. B. versucht wird, komplexe statt reelle Eingabewerte zu übergeben. Der wiederverwendbare `fft`-Funktor abstrahiert die Erzeugung eines „Plans“ (vgl. [33]) bzw. die Initialisierung der FFT Bibliothek und kann beliebig oft verwendet werden. Und schließlich reicht die Änderung der Definition von `FFT_LIB` (im Code angedeutet), um statt der cuFFT die FFTW zu verwenden.
Weitere Details zu dem Interface, dessen Verwendung und den angesprochenen FFT-Bibliotheken können den jeweiligen Dokumentationen entnommen werden.

[5] Hier angegeben durch den vordefinierten Typ `FFT_2D_R2C_F`

3 Streuprozesse

3.1 Mathematische/Physikalische Grundlagen

Im Folgenden sollen die mathematischen Modelle zur Beschreibung der Vorgänge erarbeitet, sowie auf die zu Grunde liegenden physikalischen Prozesse eingegangen werden. Da das Thema sehr komplex ist, soll hier der Schwerpunkt auf den für die Simulation benötigten Formeln gelegt werden. Für weitere Erläuterungen und Hintergründe sei auf Literatur wie [5, 6, 20] verwiesen.
In dem Computerexperiment wird die Ausbreitung von Röntgenstrahlung, was Licht mit einer bestimmten, sehr kleinen Wellenlänge ist, simuliert. Dieses Licht breitet sich als elektromagnetische Welle aus, das heißt es ist ein gekoppeltes elektrisches und magnetisches Feld, das sich über Raum und Zeit gemäß der Wellengleichung verändert (vgl. [5, S. 6]). Aus den Maxwellgleichungen folgt, dass beide Felder senkrecht zur Propagationsrichtung stehen, und das Licht sich damit als Transversalwelle bewegt (vgl. [5, S. 10]). Auch folgt aus ihnen, dass magnetisches und elektrisches Feld senkrecht zueinander stehen und ihre Amplituden sich in einem festen Verhältnis befinden (vgl. [5, S. 9 f.]). „Da die Wechselwirkung zwischen Licht und Materie im Allgemeinen über die elektrische Feldstärke $\vec{E}$ [entspricht $\underline{\boldsymbol{E}}$, Anm. d. Verf.] geschieht und die magnetische Feldstärke B [...] direkt proportional zu $\vec{E}$ ist [...]“ [5, S. 10], soll im Folgenden nur das elektrische oder „E-Feld“ betrachtet werden.
Die Polarisation von Transversalwellen, wie den elektromagnetischen Wellen, gibt die Richtungsänderung der Auslenkung an und wird u. a. von Zinth und Zinth [5, S. 225 ff.] beschrieben. Man bezeichnet eine Welle als „linear polarisiert“, wenn die Auslenkung nur den Betrag und das Vorzeichen ändert, nicht jedoch die Richtung. „Zirkular polarisiert“ ist das Licht, wenn Folgendes gilt: „Der Betrag der Feldstärke ist zeitlich konstant; das Ende des Feldvektors $\vec{E}$ beschreibt in der [Ebene senkrecht zur Ausbreitungsrichtung], eine Kreisbahn[...]“ [5, S. 226]. Es gibt auch noch ellipsenförmige Polarisation, was eine Mischform darstellt und unpolarisierte Wellen, in denen sich die Richtung unregelmäßig ändert, auf die hier aber nicht weiter eingegangen werden soll. In dieser Arbeit sollen nur linear polarisierte Wellen im Vakuum[6] betrachtet werden, was für die gegebenen Anwendungsfälle ausreicht. Darauf aufbauend sind aber auch andere Polarisationen mit wenig Aufwand implementierbar.
Die Ausbreitung einer Transversalwelle als ebene Welle kann über die Amplitudenfunktion mit Gleichung (3.1) beschrieben werden, wobei die Amplitude auch ein Vektor sein kann (vgl. dazu die durch Symmetrie äquivalente Form in [5, S. 8]).

$$A(\underline{\boldsymbol{r}}, t) = A_0 \cos(\underline{\boldsymbol{k}}\,\underline{\boldsymbol{r}} - \omega t + \varphi_0) \tag{3.1}$$

Hierbei gelten die folgenden Definitionen, die im Weiteren verwendet werden:

$\underline{\boldsymbol{r}}$ ... Ort

t ... Zeit

[6] Im Folgenden wird deshalb der Brechungsindex $n = 1$ in den Formeln weggelassen.

$A_0 \ldots$ Amplitude

$\underline{\boldsymbol{k}} \ldots$ Wellenvektor mit $k = |\underline{\boldsymbol{k}}| = \frac{2\pi}{\lambda} \ldots$ Wellenzahl

$\lambda \ldots$ Wellenlänge

$\omega = 2\pi f$ ... Kreisfrequenz oder Phasengeschwindigkeit

$f \ldots$ Frequenz

Die Welle breitet sich dabei in Richtung des Wellenvektors $\underline{\boldsymbol{k}}$ aus (vgl. [5, S. 7]) und es gilt:

$$\lambda f = c \tag{3.2}$$

Wie später ersichtlich wird, ist die Beschreibung der Welle als komplexe Funktion leichter zu behandeln. In Exponentialschreibweise ergibt sich damit Gleichung (3.3), wobei man über die bekannte Eulersche Formel durch Bestimmung des Realteils unmittelbar auf Gleichung (3.1) zurückkommt. Betrachtet man normierte Amplituden ($A_0 = 1$), ist die Funktion ausschließlich über die Phase φ nach Gleichung (3.4) definiert und man kann kurz $\tilde{A} = \mathrm{e}^{\mathrm{i}\varphi}$ verwenden.

$$\tilde{A} = A_0 \mathrm{e}^{\mathrm{i}(\underline{\boldsymbol{k}}\,\underline{\boldsymbol{r}} - \omega t + \varphi_0)} \tag{3.3}$$

$$\varphi = \underline{\boldsymbol{k}}\,\underline{\boldsymbol{r}} - \omega t + \varphi_0 \tag{3.4}$$

$$A = \mathrm{Re}\{\tilde{A}\} \tag{3.5}$$

Da nicht alle Phänomene, die bei der Betrachtung von Licht auftreten, mithilfe des Wellenmodells erklärt werden können, nimmt man an, dass Licht aus Photonen besteht, aber auch Wellencharakter besitzt, was als „Welle-Teilchen-Dualismus“ bekannt ist. „Dieser besagt, dass sich Licht manchmal wie eine Welle und manchmal wie ein Teilchen verhält. Durch diese beiden Modelle lassen sich sämtliche uns bekannten Phänomene des Lichtes beschreiben, wir müssen nur wissen, welches der beiden Modelle wir jeweils anwenden müssen.“ [5, S. 274]
Die Energie $\mathcal{E}_{Phot}$ der Photonen berechnet sich nach Gleichung (3.6) wobei h das Plancksche Wirkungsquantum ist.

$$\mathcal{E}_{Phot} = \mathrm{h}f \tag{3.6}$$

Für monochromatisches, linear polarisiertes Licht mit dem elektrischen Feld E, das sich als ebene Welle nach Gleichung (3.1) ausbreitet, gilt nach Zinth und Zinth [5, S. 11]: Die Intensität ist die Energie pro Zeit und Fläche und ergibt aus der Anzahl der Photonen so, dass für die Photonenflussdichte $N = \frac{\text{Zahl der Photonen}}{\text{Fläche}\cdot\text{Zeiteinheit}}$ Gleichung (3.7) erfüllt ist (vgl. [5, S. 272]), woraus folgt, dass die Anzahl der Photonen proportional zum Quadrat der Amplitude E_0 der elektrischen Feldstärke ist.

$$I = \frac{1}{2} c \epsilon_0 E_0^2 \tag{3.7}$$

3.2 Streuung und Interferenz

Das Licht, als elektromagnetische Welle, wechselwirkt mit anderen Feldern und Ladungen, wie freien Elektronen in Materie. „Die Mehrheit der Elektronen, insbesondere diejenigen leichter Atome und äußerer Schalen schwerer Atome verhalten sich bei der Wechselwirkung mit Röntgenstrahlen [...] wie freie Elektronen [...]“ [34, S. 107]. Deshalb, und weil es sich bei den hier untersuchten Proben meist um

Metallfolien und Plasmen handelt, die von sich aus schon viele freie Elektronen haben, beschreibt man die Strukturen über Elektronendichten, die mit dem Röntgenlaser beobachtet werden können. Die Wechselwirkung findet dabei in Form einer Streuung statt, was im Teilchenbild bedeutet, dass die Photonen von den Elektronen abgelenkt werden, wobei es einen Teil seiner Energie auf das Elektron abgibt. Für kleine Photonenenergien, also Licht mit einer Wellenlänge viel größer als ein Atomradius beziehungsweise der Comptonwellenlänge [7] eines Elektrons ($\lambda_c = \frac{h}{mc}$), ist der Impulsübertrag vom Photon an das Elektron vernachlässigbar. Demzufolge ist für diese sogenannte *Thomson Streuung*, ein Spezialfall der *Compton Streuung*, die Energie des Photons nach dem Stoß gleich der vor dem Stoß, was bedeutet, dass sich die Wellenlänge nicht ändert (s. Gleichung (3.6)). Wie in Abschnitt 1.2 motiviert, hat diese einen wesentlichen Anteil bei der SAXS an lasergetriebenem Plasma, weshalb sich die restliche Arbeit auf die Thomson Streuung als Beispiel beschränkt, und lediglich die Richtungsänderung der Photonen betrachtet werden muss. Analog sind auch weitere Streuprozesse (z. B. Compton und resonante Streuung (vgl. [35])) implementierbar. Auf die näheren Details zur Streuung von Licht an Elektronen- oder Ladungsdichten kann im Rahmen dieser Arbeit nicht eingegangen werden. Erklärungen und Herleitungen können z. B. in Als-Nielsen und McMorrow [3] oder Glinnemann und Schwarzenbach [34] gefunden werden.

Zwei im Weiteren wichtige Größen daraus sollen jedoch hier kurz vorgestellt werden. Die erste ist der totale Wirkungsquerschnitt (im Zusammenhang mit Streuung „Streuquerschnitt“ genannt) für die Thomson Streuung, der wie folgt berechnet wird:

$$\sigma_t = \frac{8\pi}{3} r_0^2 \tag{3.8}$$

Hierbei ist[8] $r_0 = \frac{e^2}{4\pi\epsilon_0 mc^2}$ die Thomson Streulänge, die auch als „klassischer Elektronenradius“ (r_e) bezeichnet wird (vgl. [3, S. 8, 352 f.]). Der Streuquerschnitt ist (hier) ein Maß für die Wahrscheinlichkeit, mit der ein Photon mit einem Elektron wechselwirkt, also gestreut wird. Für die Elektronendichte ρ_e, gegeben als Anzahl der Elektronen pro Raumelement ergibt sich die Streuwahrscheinlichkeit entsprechend zu $P_s(\rho_e) = \sigma_t \rho_e$. Eine wichtige Erkenntnis dabei ist, dass die Streuwahrscheinlichkeit unabhängig von der Wellenlänge bzw. der Photonenenergie ist.

Über die zweite Größe, den differentiellen Streuquerschnitt, lässt sich die Verteilung der gestreuten Photonen (bzw. die Intensität der Strahlung) über den Raumrichtungen bestimmen. Dieser ist in Gleichung (3.9) gegeben, wobei dessen Herleitung z. B. in Als-Nielsen und McMorrow [3, S. 349 ff.] nachgelesen werden kann.

$$\left(\frac{d\sigma}{d\Omega}\right) = r_0^2 \cos^2(\theta) \tag{3.9}$$

Der Winkel θ ist dabei der eingeschlossene Winkel zwischen der Richtung des Photons vor und nach der Streuung. Dieser Zusammenhang zeigt, dass die Streuung der Photonen rotationssymmetrisch um die ursprüngliche Flugrichtung und die Wahrscheinlichkeit für starke Ablenkungen deutlich geringer ist, als für schwache Ablenkung. Auch ist die Rückstreuung (entgegen der Flugrichtung) mit gleicher Wahrscheinlichkeit möglich, was in verschiedenen anderen Experimenten genutzt wird. Hier interessiert hauptsächlich die Vorwärtsstreuung.

Mit diesen Gleichungen kann die Streuung von Röntgenphotonen an Elektronendichten als zweistufiger

[7] Wellenlänge eines Photons gleicher Energie wie die Ruheenergie eines Teilchens

[8] Es ist zu beachten, dass e die Ladung (nicht die eulersche Konstante e) und m die Masse eines Elektrons meint.

Zufallsprozess angesehen werden, was in Unterabschnitt 4.3.3 genutzt wird, um die Propagation als Monte-Carlo-Prozess abzubilden. Der folgende Teil des Abschnitts beschreibt nun die Interaktion der gestreuten Wellen.

Interferenz beschreibt, wie sich die Felder ändern, wenn mehrere Wellen aufeinandertreffen. Die resultierende Amplitude ergibt sich nach dem Superpositionsprinzip aus Beiträgen aller Wellen am Beobachtungspunkt und kann im Fernfeld mit der Fouriertransformation genähert werden, wie gleich gezeigt wird. Um dies hier zu beschreiben, wird eine linear polarisierte, ebene Welle (Licht) der Wellenzahl k betrachtet, die auf eine Blende trifft, welche durch eine Blendenfunktion $A(\underline{\boldsymbol{r}})$ beschrieben werden kann. Diese kann binär sein und Licht stellenweise entweder vollständig durchlassen oder blockieren, oder kontinuierlich, wobei der Funktionswert die Durchlässigkeit an der gegebenen Stelle ist. Im Falle der Streuung von Photonen an Elektronen beschreibt die Blendenfunktion die Elektronendichteverteilung innerhalb des beobachteten Stoffes. Dementsprechend wäre sie sehr groß (stark blockierend) an Stellen mit vielen Elektronen und geht für geringe Elektronendichten gegen null. Zur Vereinfachung und Verdeutlichung des Prinzips, wird $A(\underline{\boldsymbol{r}})$ als konstant in x und z Richtung angenommen, wobei x die Propagationsrichtung des Lichts ist. Außerdem soll sie außerhalb eines endlichen Bereiches verschwinden, was zum Beispiel auch für eine endliche Ausdehnung der eintreffenden Welle (wie etwa bei einem Laser) gegeben ist. Die Blendenfunktion vereinfacht sich damit auf $A(\underline{\boldsymbol{r}}) = A(y)$. Betrachtet werden soll eine endliche Anzahl N von Teilstücken der Breite Δy, die sehr klein gewählt ist, sodass sich der Wert der Blendenfunktion in diesen Intervallen nur unwesentlich ändert. Um die Welle ψ hinter der Blende zu bestimmen, geht man davon aus, dass von jedem Punkt auf der Blende eine Elementarwelle in Form einer Kugelwelle ausgeht, die nach dem Fresnel-Huygensschen Prinzip (vgl. [5, S. 139]) alle phasenrichtig addiert werden müssen:

$$\psi = \sum_{n=0}^{N-1} \psi_n \tag{3.10}$$

Beobachtet man die entstehende Welle an einem beliebigen, aber festen Punkt auf einem Detektor, lässt sich Gleichung (3.11) für die einzelnen Teilwellen ansetzen, wobei der Übersicht halber das Phasenoffset als null angenommen wurde und die Amplitude vollständig von der Blendenfunktion bestimmt ist.

$$\psi_n = A(y_n)\Delta y \cdot \mathrm{e}^{\mathrm{i}(\underline{\boldsymbol{k_n}}\,\underline{\boldsymbol{r_n}} - \omega t)} \tag{3.11}$$

Dabei ist $r_n = |\underline{\boldsymbol{r_n}}|$ der Abstand zwischen Beobachtungspunkt und Ursprungspunkt der n. Teilwelle (auf der Blende) und $\underline{\boldsymbol{k_n}}$ der Wellenvektor mit der gleichen Richtung wie $\underline{\boldsymbol{r_n}}$, da eine Kugelwelle betrachtet wird. Den ortsabhängigen Teil der Phase φ_n der Teilwellen kann man in zwei Teile zerlegen: Einen gemeinsamen Anteil, der für alle Punkte auf der Blende gleich ist, und einen Teil, der von der Position y auf der Blende abhängt. Sei nun θ_n der Beobachtungswinkel [9], $k = |\underline{\boldsymbol{k}}|$ die Wellenzahl der eingehenden Welle und y' eine beliebige Position auf dem Detektor der in einen Abstand L zur Blende steht, ergeben

[9] Winkel zwischen r_n und e_x

sich folgende Beziehungen:

$$\varphi_n = -\omega t + \underline{\boldsymbol{k_n}} \cdot \underline{\boldsymbol{r_n}} \tag{3.12}$$

$$\underline{\boldsymbol{r_n}} = \begin{pmatrix} y_n - y' \\ L \end{pmatrix} = \underline{\boldsymbol{r_0}} + \begin{pmatrix} y \\ 0 \end{pmatrix}, \qquad \underline{\boldsymbol{r_0}} = \begin{pmatrix} -y' \\ -L \end{pmatrix} \tag{3.13}$$

$$\underline{\boldsymbol{k_n}} = k \begin{pmatrix} \sin(\theta_n) \\ \cos(\theta_n) \end{pmatrix} \tag{3.14}$$

$$\varphi_n = -\omega t + \underline{\boldsymbol{k_n}}\,\underline{\boldsymbol{r_0}} + k y_n \sin(\theta_n) \tag{3.15}$$

Für kleine Winkelunterschiede kann man θ_n durch ein gemeinsames θ nähern. Anschaulich bedeutet das, dass sich in ausreichend großer Entfernung eine ebene Welle bildet (s. Abb. 3.1) und die Strahlen demnach parallel sind. Dies ist dann gegeben, wenn der Detektorabstand L groß gegenüber der

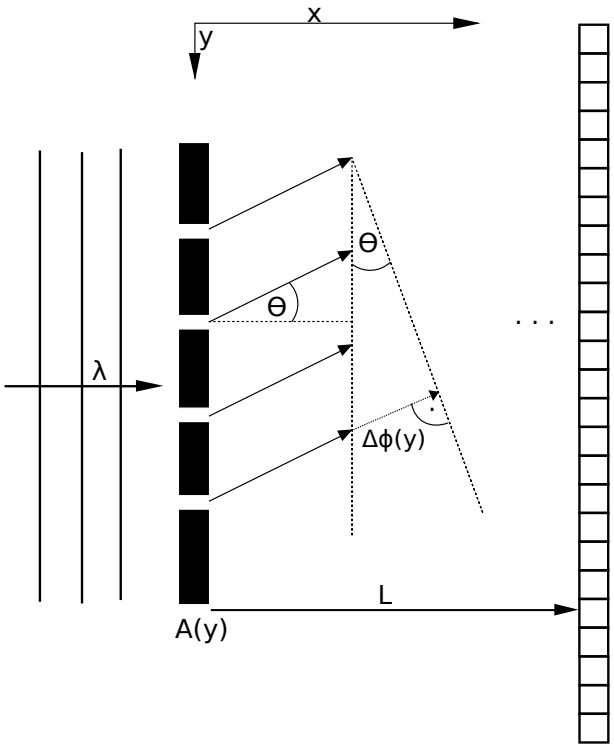


Abbildung 3.1: Welle hinter einer Blende unter dem Beobachtungswinkel θ

Blendengröße ist, wobei man in diesem Fall von „Fraunhoferscher Beugung“ spricht (vgl. [5, S. 148]). Anzumerken ist, dass im Fall einer punktförmigen Lichtquelle auch ein großer Abstand zwischen dieser und der Blende gegeben sein muss, damit die eintreffende Welle als eben angenommen werden kann, was hier vorausgesetzt wurde. Als Maß, ab wann diese *Fernfeldnäherung* angewandt werden darf, kann man einen *Grenzabstand* L_0 definieren, so dass die Fernfeldnäherung gilt, wenn $L \gg L_0$ ist. Ein Näherungsausdruck hierfür ist $L_0\lambda \approx d^2$, wobei d die Blendengröße bezeichnet (vgl. [36, S. 261]), welche in

diesem Fall $d = N\Delta y$ wäre. Damit vereinfachen sich die Summanden der Phase $\varphi_n = \varphi_0 + \Delta\phi(y)$ zu:

$$\varphi_0 = -\omega t + \underline{\boldsymbol{k_0}}\,\underline{\boldsymbol{r_0}} \tag{3.16}$$

$$\Delta\phi(y) = k y_n \sin(\theta) \tag{3.17}$$

Dies eingesetzt in Gleichung (3.10) und (3.11) führt nach Ausklammern des konstanten Faktors zu:

$$\psi(\theta) = \mathrm{e}^{\mathrm{i}\varphi_0} \sum_{n=0}^{N-1} A(y_n)\Delta y \cdot \mathrm{e}^{\mathrm{i}k y_n \sin(\theta)} \tag{3.18}$$

Schließlich kann man zur weiteren Vereinfachung eine Variable q, definiert als $q = k\sin(\theta)$, einführen und auch den Grenzfall $\Delta y \to 0$ bestimmen, für den sich der Integralausdruck in Gleichung (3.20) ergibt.

$$\psi(q) = \mathrm{e}^{\mathrm{i}\varphi_0} \sum_{n=0}^{N-1} A(y_n)\Delta y \cdot \mathrm{e}^{\mathrm{i}q y_n} \tag{3.19}$$

$$\psi(q) = \psi_0 \int_{-\infty}^{\infty} A(y)\mathrm{e}^{\mathrm{i}qy}\,\mathrm{d}y, \qquad \psi_0 = \mathrm{e}^{\mathrm{i}\varphi_0} \tag{3.20}$$

Vergleicht man Gleichung (3.20) mit Gleichung (2.8) auf Seite 10 wird ersichtlich, dass die resultierende Welle ψ der Fouriertransformierten von A bis auf einem Vorfaktor entspricht [10]. Der diskrete Fall in Gleichung (3.19) gleicht entsprechend der diskreten Fouriertransformation (s. Gleichung (2.8)). Demzufolge kann man zur Analyse und Generierung von Streubildern die Fouriertransformation verwenden, worauf das Gebiet der Fourieroptik beruht. Das Streubild entspricht dabei (bis auf Vorfaktoren) dem Betragsquadrat der Fouriertransformierten, da mit realen Detektoren nur die Intensität, nicht aber die Phase gemessen werden kann.

Daraus leitet sich das Phasenproblem ab, das darin besteht, aus einem gegebenen Streubild die Elektronendichte zu rekonstruieren, was über die zu Gleichung (3.20) inverse Fouriertransformation möglich wäre, wenn man auch die Phasen kennen würde. Eine Möglichkeit, aus diesem „Rückwärtsproblem“ ein „Vorwärtsproblem“ zu machen, besteht darin, eine Dichteverteilung zu „raten“ und dafür das Streubild zu simulieren, z. B. durch Näherung mittels Fouriertransformation. Über den Vergleich mit dem gegebenen Streubild und kann man die Schätzung iterativ verbessern. Zu beachten ist hierbei jedoch, dass es eine unendliche Anzahl an Verteilungen gibt, für die ein gegebenes Streubild erzeugt wird, so dass die Abbildung des Streubildes auf eine Elektronendichte nicht eindeutig ist.

Eine weitere Anwendung besteht darin, dass eine Dichteverteilung (beispielsweise innerhalb eines komplizierten Prozesses) simuliert und von dieser (nun bekannten) Verteilung das Streubild erzeugt wird. Über den Vergleich des simulierten, mit einem experimentell erstellten Streubild, lassen sich Rückschlüsse auf die Korrektheit der Simulation ziehen. Dies ist auch die Motivation dieser Arbeit. Mit dem Simulations-Framework PIConGPU [7] kann die Interaktion eines Hochenergielasers mit einer Metallfolie simuliert werden, bei der schnelle Ionen entstehen, die später u. a. zur Krebstherapie verwendet werden können. Da die Ionen spezielle Eigenschaften haben müssen, ist es wichtig, dass der Prozess zu deren Erzeugung genau verstanden wird. Dieser kann über die Elektronendichten beobachtet werden, die durch kurze Impulse eines Röntgenlasers ein Streubild auf einem Detektor indirekt sichtbar werden.

[10] Eine ähnliche Herleitung findet sich auch in Zinth und Zinth [5, S. 144 ff.] und Stößel [6, S. 38 ff.].

Damit hat man hier aber genau das angesprochene Phasenproblem.
Mit der im Rahmen dieser Diplomarbeit entwickelten Simulation kann nun die Streuung der simulierten Elektronendichte bestimmt werden, wobei noch weitere Effekte, wie die im nächsten Kapitel beschriebene Mehrfachstreuung, berücksichtigt werden können, was mit der Fouriertransformation nicht einfach möglich ist.
Abschließend zu diesem Abschnitt sei hier nur erwähnt, dass auch die Herleitung des Streubildes von Blenden bzw. Elektronendichten, die in mehr als einer Richtung variabel sind, in ähnlicher Weise möglich ist. Dies führt zu mehrdimensionalen Fouriertransformationen, die hier nur am Rande betrachtet werden können.

3.3 Mehrfachstreuung

Die bis hierhin beschriebene Analyse behandelte nur Einfachstreuung, das heißt es wird angenommen, dass die Welle nur an einem Punkt gestreut wird und ohne weitere Streuung auf den Detektor trifft. Dies ist eine Näherung, die für Proben mit relativ geringer Elektronendichte (im Vergleich zu der Dichte, bei der Totalreflexion auftritt) sehr gut gilt. Bei höheren Elektronendichten kann der Einfluss der Mehrfachstreuung nicht mehr vernachlässigt werden, da entsprechend des in Abschnitt 3.2 gezeigten Streuquerschnitts die Wahrscheinlichkeit für ein Streuereignis mit der Dichte steigt. Damit ist es auch wahrscheinlicher, dass bereits gestreute Wellen an der restlichen Elektronendichte erneut gestreut werden, was sich auch auf das Streubild auswirkt. In Tsuji et al. [37, S. 447 ff.] wird in einem ähnlichen Experiment gezeigt, wie sich durch zusätzliche Betrachtung der Streuung zweiter bis sechster Ordnung ein Ergebnis erreichen lässt, das deutlich näher an den experimentell gemessenen Werten liegt. Persliden [16] zeigt die Abhängigkeit des Verhältnisses von Mehrfach- zu Einfachstreuung bei der Untersuchung von Streuung in PMMA (Acrylglas) mit einem Compton-Spektrometer. Er findet dabei einen linearen Zusammenhang zwischen diesem Verhältnis und der Dicke der Probe und gibt dafür einen Wert von rund $12\,\%$ für $5\,\mathrm{mm}$ dicke Proben an. Dieses Verhältnis hängt jedoch sehr stark von der bestrahlten Probe und der Energie der Röntgenphotonen ab, wie die Diagramme in Tsuji et al. [37, S. 448] zeigen. In diesen können Energiebereiche (abhängig vom Material) erkannt werden, für die das Streubild durch die Einfachstreuung dominiert wird, sowie andere, in denen fast ausschließlich Mehrfachstreuung das Ergebnis bestimmt.
Die Berechnung der Mehrfachstreuung mittels der oben gezeigten Fouriertransformation ist nicht möglich, da hier die Betrachtung im Fernfeld (im hohen Abstand zum Streupunkt) angenommen wurde. Beim Berechnen der Streuung gestreuter Wellen, befindet man sich noch innerhalb der untersuchten Probe und damit im Allgemeinen im Nahfeld, für welches die obigen Annahmen nicht mehr gelten. Außerdem schreibt Persliden „The behavior of multiply-scattered photons is, however, too complex to solve by analytic methods.“ [16, S. 384] Theoretisch wäre es möglich eine Particle-In-Cell Simulation zu nutzen, bei der das Feld der Welle(n) auf Gitterpunkten diskretisiert wird. Durch die iterative Lösung der Maxwell-Gleichungen kann so die Propagation und auch die Streuung simuliert werden. An dieser Stelle ist anzumerken, dass sich die in Abschnitt 3.2 betrachteten Zusammenhänge für die Interaktion und Interferenz der Wellen natürlich auch direkt aus den Maxwell-Gleichungen herleiten lassen, wobei dies jedoch aufwendiger ist.
Der Ansatz über die Auswertung der Maxwell-Gleichungen wird u. a. in PIConGPU genutzt, um die

Interaktion des Hochenergielasers mit z. B. einer Metallfolie oder Plasma zu simulieren. Die verwendeten Wellenlängen sind dabei in der Größenordnung von 10^{-5} m für die Beispiele „LaserWakefield“ (Kielfeld-Beschleuniger) und „KHI“ (Kelvin-Helmholtz-Instabilität) (vgl. [38]). Entsprechend des bekannten Abtasttheorems (s. [21, S. 103]), müssen die Zellen in der Simulation, welche hier die Abtastung bzw. Diskretisierung der Welle bilden, weniger als halb so groß wie die Wellenlänge sein. Für den gegebenen Hochleistungslaser werden typischerweise Zellgrößen der Größenordnung 10^{-7} m verwendet, was bei einer realistischen Gesamtgröße der Probe zu sehr vielen Zellen führt, wodurch entsprechend auch viel Rechenaufwand notwendig ist. Da der Röntgenlaser eine Wellenlänge in der Größenordnung 10^{-10} m besitzt, müssten die Zellen für eine korrekte Abtastung entsprechend 10.000 mal kleiner sein, was für 3D-Simulationen zu 10^{12} mehr Zellen bei gleicher Probengröße führt. Es ist offensichtlich, dass die Zeit zur Berechnung bei dieser extrem hohen Anzahl von Zellen enorm wäre, weshalb diese Methode hier nicht verwendet werden kann. In Piskozub und McKee [39] ist ein Ansatz beschrieben, in dem die Beiträge jedes Punktes berechnet und anschließend kombiniert werden. Für große Volumina ist jedoch der benötigte Speicher sehr hoch und die Parallelisierung wahrscheinlich nicht sehr performant möglich.

Die Lösung, welche im weiteren Verlauf dieser Arbeit vorgestellt werden soll, ist die Nutzung einer Monte-Carlo-Simulation, in der die in Abschnitt 3.2 beschriebenen Zufallsprozesse mittels photonenähnlicher Teilchen modelliert werden. Die Idee ist, dass für eine unendliche Anzahl von Versuchen (hier: Teilchen) alle möglichen Ereignisse mit ihren Wahrscheinlichkeiten exakt simuliert werden können, aber schon eine endliche Anzahl an Teilchen ausreicht, um die wesentlich zum Streubild beitragenden Ereignisse zu berücksichtigen. Im folgenden Kapitel 4 wird die implementierte Simulation beschrieben. Die Konvergenz des Simulationsergebnisses gegen die analytische Lösung wird in Kapitel 6 „Ergebnisse“ gezeigt.

4 Simulation

4.1 Monte-Carlo Algorithmen auf GPUs

Wie am Ende von Abschnitt 3.3 erläutert, ist die Simulation der Streuung des Röntgenlasers an der Elektronendichte aufgrund der extrem kurzen Wellenlänge zu komplex, um diese exakt und vollständig durchzuführen. Aus diesem Grund wird die Propagation der Welle als Zufallsprozess über die möglichen Wege einzelner Strahlen gesehen. Die zu beobachtende Größe ist die Verteilung über die „Flugzeiten“ der Photonen, die in eine Detektorzelle treffen, da sich über diese Zeiten die Phasen der Photonen berechnen lassen, woraus sich durch Überlagerung die resultierende Welle ergibt. Bezogen auf die Einfachstreuung aus Abschnitt 3.2 ergab sich die Flugzeit aus der Propagation bis zum Streupunkt (bzw. dem Ausgangspunkt der huygensschen Elementarwelle auf der Blende) und dem weiteren Weg bis zum Detektor. Dabei war der erste Teil für alle Photonen gleich und Unterschiede resultierten lediglich aus der unterschiedlichen Entfernung zum Detektor nach der Streuung. Dies gilt hier nicht mehr, da die Photonen durch Mehrfachstreuung im Prinzip beliebige Wege durch das Simulationsvolumen (äquivalent zur allgemeinen, dreidimensionalen Funktion $A(\underline{\boldsymbol{r}})$) nehmen und damit auch stark unterschiedliche Flugzeiten haben können.

Dafür wird nun eine Monte-Carlo Simulation eingesetzt. Das Prinzip besteht darin, durch eine große Anzahl an Photonen sehr viele der möglichen Wege, insbesondere der damit möglichen Flugzeiten, zu betrachten, wobei deren Wahrscheinlichkeitsverteilung erhalten bleiben soll. Das bedeutet, dass wahrscheinlichere Wege entsprechend häufiger ihren Beitrag zum Ergebnis leisten oder alternativ diese stärker gewichtet werden. Nach dem Gesetz der großen Zahlen konvergiert die simulierte Verteilung der Flugzeiten im Mittel gegen die tatsächliche Verteilung. Es handelt sich dabei um einen stochastischen Ansatz, der allerdings viel Rechenzeit braucht, da eine gute Konvergenz nur bei einer großen Anzahl an Zufallsexperimenten gegeben ist. Dies ist einer der Gründe, warum die Monte-Carlo Simulation meist nur eingesetzt wird, wenn die analytische oder anderweitige Berechnung nicht möglich oder zu aufwändig wäre, was hier der Fall ist. Die genaue Betrachtung von Monte-Carlo Simulationen im Allgemeinen ist für den Rahmen dieser Arbeit zu umfangreich. Da solche Simulation u. a. auch in Persliden [16] und Alerstam et al. [17] erfolgreich für sehr ähnliche Probleme genutzt wurden, sind auch hier damit gute Ergebnisse zu erwarten. Einige Hintergrundinformationen zu Monte-Carlo Simulationen können in diesen Werken, sowie deren Referenzen gefunden werden.

Eine Möglichkeit, die Rechenzeit der Simulation gering zu halten, ist die Nutzung der hohen Rechenleistung von Grafikkarten (GPUs), was in diesem Zusammenhang als GPGPU („General Purpose Computation on Graphics Processing Unit“) bezeichnet wird. Im Folgenden werden dabei NVIDIA Grafikkarten und das CUDA Programmiermodell (vgl. [40]) betrachtet, wobei es aktuell Bestrebungen gibt, die hier verwendete Bibliothek und auch das ähnlich aufgebaute PIConGPU mittels einer weiteren Abstraktionsbibliothek namens Alpaka (vgl. [41]) auch auf anderen Plattformen lauffähig zu machen. Um eine möglichst hohe Performance und Skalierbarkeit zu erreichen, sind einige grundlegende Kenntnisse der

Hardware notwendig. Zum Test der Simulation wurden in dieser Arbeit Grafikkarten des Typs K20X verwendet, die auf der *Kepler* Architektur basieren. Da deren Nachfolger *Maxwell* und die zur Zeit aktuellste Architektur *Pascal* nicht grundsätzlich verschieden sind, soll die folgende, etwas allgemeine Betrachtung der Kepler Architektur ausreichen, um auf die für die Performance relevanten Teile einzugehen. Der Grafikprozessor besteht aus mehreren, sogenannten „Streaming Multiprocessors“ (SMX), der in Abb. 4.1 schematisch dargestellt ist. Diese arbeiten weitestgehend unabhängig voneinander und

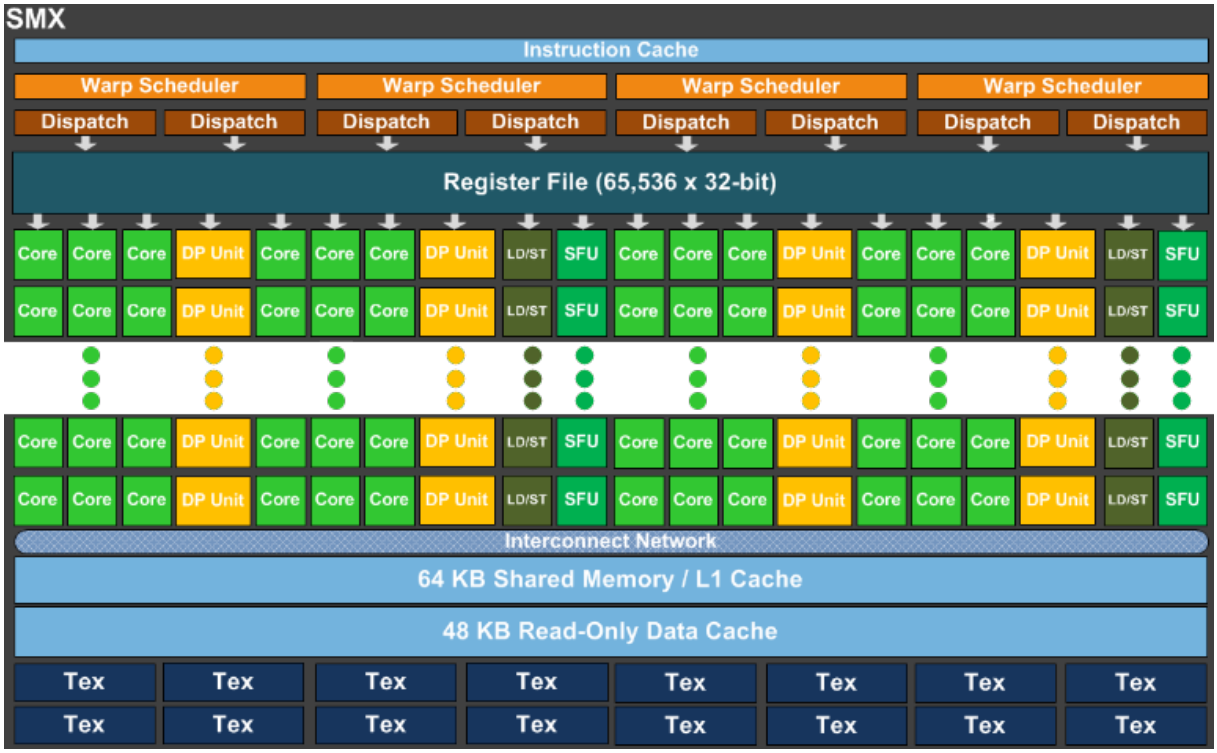


Abbildung 4.1: Schematische Darstellung eines SMX der Kepler Architektur [42] (vgl. [43, S. 8])

beinhalten die „CUDA Cores“, welche für den Großteil der Berechnungen verwendet werden. Wichtig sind auch die getrennten Einheiten für Berechnungen in doppelter Genauigkeit (DP Unit), sowie die „Special Function Units“ (SFU) für die schnelle, näherungsweise Berechnung von z. B. transzendenten Funktionen (vgl. [43, S. 9]).
Daraus ergibt sich, dass der Durchsatz an ausgeführten Rechenoperationen von der Art der Berechnung abhängt. Am wichtigsten ist hier die Anzahl an Gleitzahloperationen, die pro Takt ausgeführt werden können, da der Großteil der Simulation aus solchen besteht. Im Allgemeinen gilt jedoch, dass die Anzahl der Ganzzahloperationen ähnlich oder gleich der der Gleitzahloperationen in einfacher Genauigkeit ist. Bei den hier verwendeten GPUs kann jeder SMX 192 Operationen (Addition, Multiplikation, FMA) in einfacher Genauigkeit ausführen. In doppelter Genauigkeit sind es nur noch 64, was einem Verhältnis von $\mathbf{3} : \mathbf{1}$ entspricht und auch in dem Schema aus Abb. 4.1 zu sehen ist. Dieser Unterschied, der in der Nachfolge-Architektur (Maxwell) mit einem Verhältnis von $\mathbf{32} : \mathbf{1}$ sogar noch größer wird (vgl. [44]), ist Teil der Motivation, so viel wie möglich in einfacher Genauigkeit zu berechnen, da hier ein hoher Geschwindigkeitsgewinn gegenüber doppelter Genauigkeit zu erwarten ist. In Kapitel 5 wird darum durch die Analyse und darauf basierender Tests der numerischen Genauigkeit untersucht, welche Fehler zu erwarten sind, wenn in reduzierter Genauigkeit gerechnet wird, um so zu entscheiden, für welche

Berechnungen dies ausreicht und ob sich durch einen anderen Algorithmus mehr Genauigkeit erreichen lässt.

Ein weiterer wichtiger Aspekt bei der Ausführung von Operationen in einem SMX ist das SIMT (Single Instruction, Multiple Threads) Konzept mit den sogenannten Warps (vgl. [44, SIMT Architecture]). Diese bestehen (zur Zeit) aus 32 Threads, die in jedem Takt die gleiche Operation ausführen. „[S]o full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.“ [44, SIMT Architecture] Dies ist im Wesentlichen vergleichbar mit Vektoroperationen, in der eine Operation gemäß dem SIMD (Single Instruction, Multiple-Data) Prinzip gleichermaßen auf mehrere Daten angewandt wird. Eine weitere Gemeinsamkeit dieser zwei Modelle besteht darin, dass es für beide meist von Vorteil ist, wenn die Daten als Strukturen von Arrays (SoA) statt als Array von Strukturen (AoS) vorliegen. Am Beispiel der Partikeleigenschaften (s. unten) bedeutet das, dass es sinnvoller ist, für jede Eigenschaft der Partikel ein separates Array anzulegen, statt alle Eigenschaften in einem Array zu speichern.

```
#define N ...
struct Particle{
   float3 position;
   float3 direction;
   float phase;
};
Particle AllParticles[N];
```

Listing 4.1: Datenlayout als AoS

```
#define N ...
struct ParticleData{
   float3 position[N];
   float3 direction[N];
   float phase[N];
};
ParticleData AllParticles;
```

Listing 4.2: Datenlayout als SoA

Diese zwei Varianten sind in Listing 4.1 und 4.2 gegenübergestellt, wobei hier nur das Prinzip verdeutlicht werden soll; der echte Code nutzt natürlich keine Arrays fester Größe. Der Vorteil bei der Nutzung von SoA liegt in der besseren Ausnutzung von Caches und Speicheroperationen (Lesen und Schreiben), was sich am Beispiel der Phase wie folgt verdeutlichen lässt: Angenommen alle 32 Threads eines Warps benötigen die Phase je eines der aufeinanderfolgenden[11] Partikel für eine Berechnung. Da diese Threads die jeweilige Operation gleichzeitig ausführen, fordern auch alle die (hier 4 Byte großen) Daten gleichzeitig an. Die Speichertransaktionen sind jeweils 32 bis 128 Byte groß (vgl. [44, device-memory-accesses]), so dass bei jeder Ladeoperation auch Teile umliegenden Speichers mit geladen werden. Im Falle von AoS würden demnach auch die anderen Attribute eines Partikels geladen werden, obwohl davon jeweils nur die Phase benötigt wird, während beim SoA-Layout mit einer Speichertransaktion gleich mehrere Phasen geladen werden. Insbesondere vereinigt die Hardware Zugriffe auf direkt aufeinanderfolgende Daten in einer einzigen Speichertransaktion („Coalescing“ im *CUDA C Programming Guide*). Idealerweise (abhängig von der Ausrichtung (engl. „Alignment“) der Daten) können so alle 4 Byte großen Anfragen eines Warps mit einer einzigen 128 Byte Transaktion bedient werden, was die

[11] Im Allgemeinen organisiert man die Berechnung so, dass jeder Thread einen Datensatz verarbeitet und benachbarten Threads aufeinanderfolgende Datensätze zugeordnet werden.

vorhandene Speicherbandbreite optimal ausnutzt. Auch können mit diesem Datenlayout die Zwischenspeicher (Caches) oft deutlich besser genutzt werden, da nur die Daten, die auch tatsächlich benötigt werden, in die Caches geladen werden. Eine Besonderheit, stellen die Attribute für Position und Bewegungsrichtung dar, die in den obigen Listings als `float3`, und damit als Verbund dreier Einzelwerte, definiert sind. Auch diese könnte man natürlich aufteilen und die einzelnen Werte als Arrays ablegen, jedoch ist zum einen der (Programmier-)Aufwand für SoA höher und zum anderen werden diese Werte i. A. alle gleichzeitig oder kurz nacheinander benötigt, da z. B. bei der Änderung der Position zumeist alle Komponenten verändert werden. Für diesen Zweck besitzt CUDA die Möglichkeit, bis zu 16 Byte große Vektoren (je Thread) mit einer Anweisung zu laden. Diese werden natürlich wieder in maximal 128 Byte großen Speichertransaktionen verarbeitet, so dass dies für sich betrachtet noch keinen großen Vorteil bringt, jedoch wirkt es sich positiv auf die Parallelität innerhalb eines SMX aus. Da die Speicheroperationen (insbesondere wenn die Daten nicht in einem Cache gefunden werden) einige Zeit brauchen, können der betreffende Warp nicht weiterarbeiten und wird „pausiert“. Für diesen Fall kann der SMX jedoch einen anderen Warp zur Ausführung bringen, der keine unerfüllten Abhängigkeiten hat. Sobald die Daten des ersten Warps vorliegen, kann dieser weiter rechnen. Diese Überlappung von Latenzen (sowohl durch Speicher- als auch durch Rechenoperationen, wobei erstere deutlich größer sind) ist entscheidend für die Performance, da diese nur dann maximal ist, wenn in jedem Takt eine (sinnvolle) Operation ausgeführt werden kann. Die vektorisierten Speichertransaktionen helfen hier insofern, dass sie die Anzahl der für die gleiche Aufgabe nötigen Befehle reduzieren. Anstatt also noch einen zweiten und dritten Befehl zum Laden bzw. Schreiben von Speicher auszuführen, können stattdessen Instruktionen anderer Warps ausgeführt werden, so dass deren Ergebnisse entsprechend eher zur Verfügung stehen.

Aus diesen Überlegungen folgt noch ein weiterer Schluss: Um optimale Performance zu erreichen benötigt ein Multiprozessor möglichst viele Warps (bzw. Threads), die ausgeführt werden sollen. Obwohl nur einige (bei den Kepler SMX maximal 192) Threads echt parallel ausgeführt werden können, benötigt man jedoch deutlich mehr, um die angesprochenen Latenzen durch Ausführung von anderen Warps zu verstecken. In CUDA wird deshalb häufig die Occupancy (Belegung) betrachtet, die sich aus dem Verhältnis der tatsächlichen zur maximal möglichen Anzahl der Threads pro GPU bzw. SMX ergibt, und durch Registernutzung, genutzten gemeinsamem Speicher („Shared Memory“) und die gewählten Blockgrößen[12] bestimmt wird. Registernutzung und Shared Memory ist ein klassisches Beispiel für einen Time-Memory Tradeoff, indem durch mehr des einen das jeweils andere reduziert werden kann, wobei auch dem Grenzen gegeben sind. So kann die Registernutzung nur indirekt und sehr wenig beeinflusst werden, jedoch lässt sich mit der Nutzung von einfacher Genauigkeit der Registerverbrauch pro Thread deutlich reduzieren, da für jeden Wert in doppelter Genauigkeit zwei Register verbraucht werden (vgl. [44, multiprocessor-level]). Der Shared Memory, als eine Art frei nutzbarer Cache, ist sinnvoll, um Werte innerhalb eines Blocks wiederzuverwenden oder auszutauschen. Er ist durch seine gute Geschwindigkeit und Flexibilität oft wesentlich für die hohe Performance von GPGPU Anwendungen verantwortlich. Auch hier kann der Verbrauch durch kleinere Datentypen reduziert werden. Die Reduktion der Anzahl der gespeicherten Elemente unterliegt jedoch meist algorithmischen Einschränkungen, wenn zum Beispiel jeder Thread jeweils ein Element in diesem Speicher speichern muss. Zur Blockgröße sei hier schließlich nur gesagt, dass die Nutzung großer Blöcke zwar die Wiederverwendbarkeit von Wer-

[12] “Blöcke“ bezeichnen hier Gruppen von Warps, die garantiert auf einem SMX ausgeführt werden und zwischen denen direkte Synchronisation möglich ist. Weitere Details sind in [44, thread-hierarchy] zu finden.

ten über den Shared Memory verbessert, aber auch die Auslastung eines Multiprozessors einschränken kann, wenn sie zu groß gewählt wird. Das liegt an der Bedingung, dass ein Block immer vollständig auf einem SMX liegen muss, sodass ein Block mit hoher Ressourcennutzung eventuell nicht mehr vollständig auf einen SMX passt und daher Teile davon nicht genutzt werden können, was bei einer Aufteilung in kleinere Blöcke jedoch möglich wäre.
Zusammenfassend ergeben sich damit folgende Aspekte, die beim Entwurf performanter Algorithmen auf GPUs beachtet werden müssen:

- Parallele Formulierung des Algorithmus mit wenigen, datenabhängigen Divergenzen
- Effektive Nutzung der unterschiedlichen Speicher (Register, Caches, gemeinsamer und globaler Speicher)
- Effektive Nutzung der Speicherbandbreite (z. B. über das verwendete Datenlayout)
- Ermöglichung hoher Auslastung der SMX

Damit soll nun die Beschreibung der implementierten Simulation, vor allem auch unter Betrachtung der GPU Nutzung, folgen.

4.2 Aufbau der Simulation

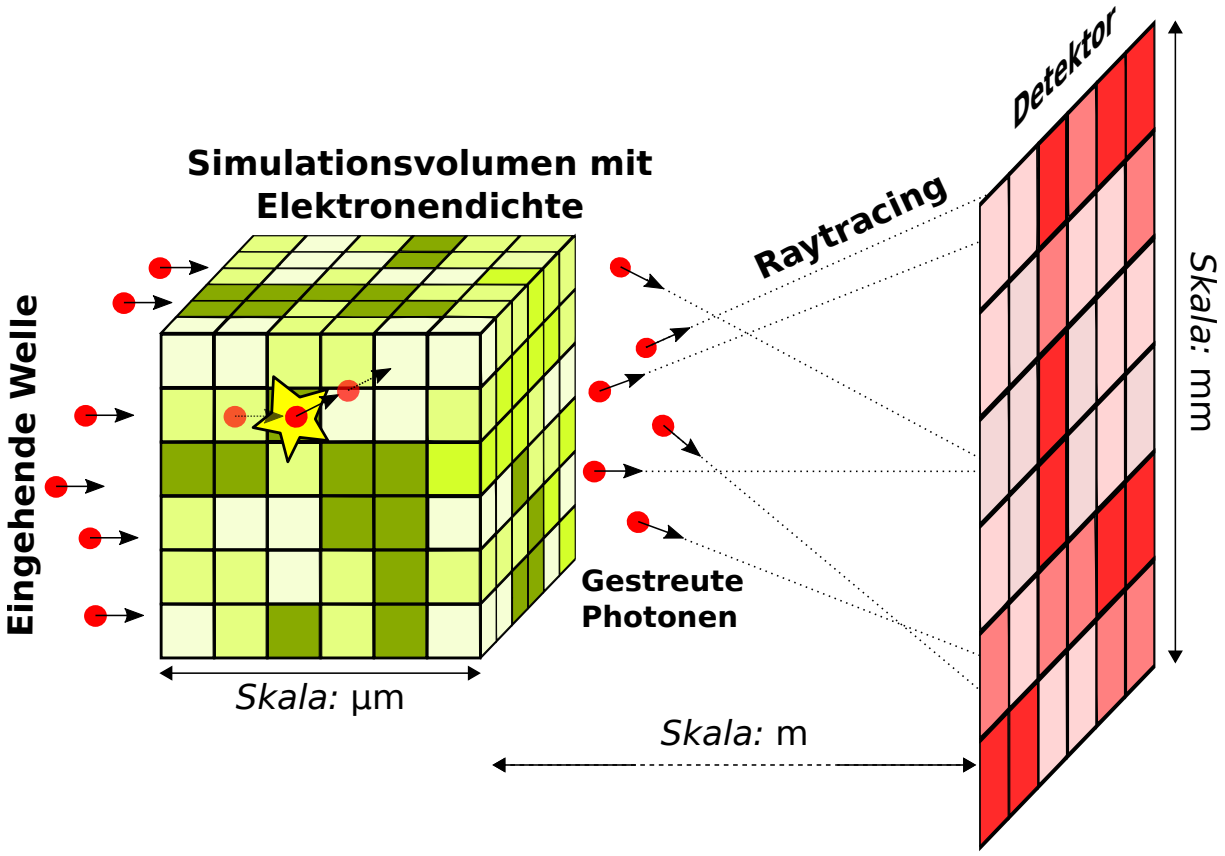


Abbildung 4.2: Schematischer Aufbau der Simulation

Der Röntgenlaser wird als Strom von Photonen betrachtet, die von einer Seite bei $x = 0$ auf eine Probe treffen, sich darin bewegen und gestreut werden können. Beim Verlassen der Probe können sie von

einem Detektor aufgefangen werden, welcher auf der gegenüberliegenden Seite parallel zur y-z-Ebene steht. Das zu simulierende Volumen[13] (die Stoffprobe) wird hierzu in dreidimensionale, quaderförmige Zellen aufgeteilt, denen jeweils eine Elektronendichte zugewiesen wird, was dem Sampling der Dichtefunktion $A(\underline{\boldsymbol{r}})$ aus Abschnitt 3.2 entspricht. Diese Dichte kann dabei aus einer anderen Simulation, wie PIConGPU, stammen oder auch durch eine Funktion beschrieben werden. Für diese Arbeit wird angenommen, dass die Dichte nicht von den Photonen beeinflusst wird, wodurch man nur die Bewegung der Photonen betrachten muss. Der Detektor ist ebenfalls in Zellen aufgeteilt, wobei deren Größe unabhängig von der Zellgröße innerhalb des Simulationsvoluminas ist. Dieser Aufbau ist in Abb. 4.2 schematisch dargestellt, wobei die unterschiedlichen Schattierungen für die verschieden Elektronendichten bzw. Intensitäten stehen. Diese festgelegte Geometrie vereinfacht die weitere Behandlung, stellt jedoch keine übermäßige Einschränkung dar, da dies dem häufig verwendeten Versuchsaufbau entspricht und sich die meisten anderen Konstellation durch geeignete Transformationen abbilden lassen. Im aktuellen Stadium der Implementation ist es aber bereits möglich, die Photonen auf der (im Bild) oberen oder vorderen Ebene zu erzeugen, was für fast alle Anwendungsfälle ausreicht.

Diese Simulation nutzt Methoden der Particle-In-Cell (PIC) Simulationen bzw. Particle-Mesh Algorithmen, um die Photonen durch das Volumen zu propagieren. Die Basis der Implementation ist die Bibliothek libPMacc, die Teil des PIConGPU Repositorys [7] ist und auch von PIConGPU genutzt wird. Diese bietet ein Grundgerüst für hoch skalierbare PIC-Simulationen in C++ und CUDA, wodurch Particle-Mesh-basierte Algorithmen ohne viel Aufwand implementiert und skalierbar auf vielen GPUs gleichzeitig ausgeführt werden können. Dazu wurde die Kommunikation weitestgehend abstrahiert und es stehen Möglichkeiten zur einfachen Darstellung von Partikeldaten im SoA-Layout zur Verfügung. Auch existieren schon Basisroutinen zur Darstellung und Bewegung der Partikel in den Zellen. Die Bibliothek setzt Techniken der Template Metaprogrammierung (vgl. Abschnitt 2.3) in einem hohen Umfang ein, um den Overhead durch die Abstraktionen zu minimieren und die Performance zu maximieren. Ein wichtiger Grund für die Entscheidung, diese Bibliothek zu verwenden, ist die Möglichkeit der zukünftigen Integration dieser Simulation in PIConGPU wodurch Strahlungstransport und Wechselwirkung selbstkonsistent implementiert werden können. Aus diesem Grund wurde bei der Implementation sehr viel Wert darauf gelegt, dass die Funktionsweise so ähnlich wie möglich ist und auch die gleiche Form der Parametrisierung genutzt wird, was die spätere Integration sehr erleichtern wird.

Zur Verwendung von libPMacc ist eine gewisse Grundstruktur für den Algorithmus und die Daten vorgegeben. Kernstück ist das bereits mehrfach erwähnte N-dimensionale Gitter, dass aus Zellen besteht, die mit nullbasierten Indizes referenziert werden können. Der Ursprung, für den der Index der 0-Vektor ist, befindet sich am Beispiel aus Abb. 4.2 an der linken oberen Ecke, in der vorderen Ebene. Für das hier verwendete dreidimensionale Gitter verläuft die x-Achse nach rechts zum Detektor, die y-Achse nach unten und die z-Achse nach hinten. Diese Festlegung ist zwar grundsätzlich arbiträr, wird aber im Weiteren zur Beschreibung verwendet. Da libPMacc auf MPI-parallele Anwendungen ausgelegt ist, kann das globale Gitter in quaderförmige, lokale Gitter zerlegt werden, die den einzelnen Prozessen zugeordnet sind. Deren Größen müssen nicht notwendigerweise gleich sein, da es aus Gründen des Lastausgleichs sinnvoll sein kann, bestimmten Prozessen weniger Zellen zuzuordnen, wenn z. B. zu erwarten ist, dass sich dort mehr Partikel befinden und darum die Rechenzeit für diesen Bereich höher ist. Die kleinste Ein-

[13] Im Folgenden wird „Simulationsvolumen“ oder kurz „Volumen“ für die (diskretisierte) Stoffprobe verwendet, da die Simulation fast ausschließlich darin abläuft.

heit bei der Aufteilung ist eine *Superzelle*, die aus einer festen Anzahl einzelner, zusammenhängender Zellen besteht, deren Bedeutung gleich erläutert werden wird.
LibPMacc unterstützt bereits die Definition von Partikeln, wobei Partikel mit gleichen Eigenschaften einer Spezies zugeordnet werden. Je nachdem, ob sich deren Werte ändern können, sind die Eigenschaften in globale, für alle Partikel einer Spezies geltende *Flags*, sowie instanzabhängige *Attribute* unterteilt. Eine besondere Form der Flags sind dabei bestimmte Algorithmen, die als Policies (vgl. Abschnitt 2.3) den Spezies zugeordnet und an entsprechenden Stellen des Algorithmus verwendet werden können. Instanzen von Partikeln werden eindeutig der Superzelle zugeordnet, deren Volumen die Position des Partikels beinhaltet. Diese Position wird hierarchisch gespeichert und besteht aus drei Teilen: Der Erste ist die Zugehörigkeit zu einer Superzelle, deren Position durch den Index implizit gegeben ist. Den zweiten Teil bildet der Zellindex innerhalb der Superzelle, wobei die Kombination aus diesen beiden Indizes als „globaler Zellindex“ bezeichnet wird. Schließlich wird die Position relativ zu dieser Zelle angegeben, was eine kontinuierliche Repräsentation (im Rahmen der Auflösung von Gleitkommazahlen) ermöglicht. Der letzte Teil wird „Inzell-Position“ genannt und ist in jeder Dimension so definiert, dass ein (Gleitkomma-) Wert von $0{,}0$ den Anfang der Zelle und $1{,}0$ den der nächsten bezeichnet. Demzufolge beträgt der gültige Wertebereich hierfür $[0; 1)$ in jede Richtung. Wenn diese Position sich außerhalb dieses Bereichs befindet, hat das Partikel die jeweilige Zelle verlassen und muss in eine andere verschoben und die Indizes entsprechend angepasst werden. Dies wird von libPMacc transparent übernommen, allerdings mit der Einschränkung, dass das Partikel in einem Zeitschritt nur eine angrenzende Zelle erreichen darf, was sich mit der Wahl eines ausreichend kleinen Zeitschritts Δt sicherstellen lässt. Formal gilt für diesen, die Lichtgeschwindigkeit c und die Zellgröße Δs (in jeder Dimension):

$$c \cdot \Delta t \leq \Delta s \tag{4.1}$$

Damit komme ich zum Ablauf einer Simulation. LibPMacc bearbeitet diese iterativ, indem die Zeit in einzelne, äquidistante und ausreichend kleine Zeitintervalle unterteilt wird und in jeder Iteration die Änderungen eines solchen Zeitschritts berechnet wird. Im allgemeinen Fall bedeutet dies die Lösung der Maxwell-Gleichungen für dieses Intervall aus dem aktuellen Zustand heraus, die Berechnung der Wirkung auf die Partikel, deren eventuelle Bewegung und die Bestimmung der Rückwirkung der Partikel auf die Felder. In dem hier behandelten Fall ist dies hauptsächlich die Erzeugung, Bewegung und Detektion der Photonen. Die Simulation endet, wenn eine bestimmte Anzahl an Zeitschritten erreicht wurde, wobei es theoretisch auch möglich wäre, sie vorher zu beenden.
Nachdem hier der grundlegende Aufbau und die verwendete Bibliothek, sowie deren Besonderheiten vorgestellt wurde, befasst sich das folgende Kapitel mit dem speziellen Ablauf der Simulation. Hauptaugenmerk liegt dabei auf der Funktionsweise, den verwendeten Modellen und eventuellen Optimierungen im Hinblick auf Performance und Korrektheit. Eine umfassende Analyse der verwendeten Formeln und der erreichbaren Genauigkeit findet sich anschließend in Kapitel 5.

4.3 Ablauf

4.3.1 Überblick über den Algorithmus

Das Programm muss aus drei Komponenten bestehen, die jeweils simuliert werden müssen: Lichtquelle, Propagation der Photonen und Detektion. Daraus folgt der allgemeine Ablauf, der aus drei Schritten besteht, die in jedem Zeitschritt ausgeführt werden:

1. Einfügen von neuen Photonen durch die Lichtquelle in der Eintrittsebene (hier bei $x = 0$)
2. Bewegung aller Partikel entsprechend ihrer aktuellen Richtung und eventuell Richtungsänderung durch Streuung (später: Betrachtung anderer Interaktionen mit der Materie)
3. Simulation der Detektion von Photonen, die das Volumen verlassen haben

Diese Schritte werden sequentiell nacheinander ausgeführt, wobei die Ausführung jedes Schrittes parallel über mehrere oder alle Zellen bzw. Partikel geschieht. Bei der Entscheidung für dieses Design spielten mehrere Faktoren eine Rolle. Grundsätzlich ist es ein Nachteil, sequentielle Teile des Programms zu haben, selbst wenn sie auf einer so hohen Ebene sind, wie hier. Die Aufteilung in die einzelnen Schritte verhindert auch das Vorhalten der Daten eines Partikels in Registern und erfordert, dass alle Daten zwischen diesen Schritten wieder in den Hauptspeicher (oder zumindest in die Caches) zurückgeschrieben werden müssen, was einen negativen Einfluss auf die Performance hat. Untersucht man Implementationen ähnlicher Algorithmen, wie den in Alerstam et al. [17] genutzten (zu finden unter [45]), sieht man einen anderen Ansatz. Die Autoren weisen jedem Thread ein Partikel zu und lassen dieses von der Erzeugung bis zur Detektion von diesem Thread und innerhalb eines Kernels[14] propagieren. Wenn das Partikel durch Absorption oder Detektion entfernt wurde, setzt dieser Thread das Partikel auf den Anfang zurück und berechnet effektiv den Pfad eines neuen Partikels. Dies ist sehr effizient und „[...] the GPU implementation is about 1080× faster than the conventional CPU implementation.“ [17, S. 060504-2] Leider ist diese Technik hier nicht anwendbar, da Alerstam et al. von gleichen Streuwahrscheinlichkeiten bzw. äquivalenter Weise einer konstanten Elektronendichte ausgeht, was hier nicht der Fall ist. Zwar ist es grundsätzlich möglich, die Streuwahrscheinlichkeiten von einer Dichte abhängig zu machen, wodurch der Algorithmus mit nur wenigen Anpassungen auch für die hier gegebene Aufgabe verwendbar ist, doch dann muss man sich fragen, ob der Algorithmus skalierbar ist. Dazu soll der notwendige Speicherbedarf abgeschätzt werden. Angenommen der vollständige Speicher der Grafikkarte könnte für die Dichte verwendet werden und jede Zelle würde dafür 4 Byte benötigen, dann beträgt die maximale Gittergröße für eine Grafikkarte etwa 1280×1024^2 Zellen bei einer K20 (5 GB Speicher [46]) und etwa 3072×1024^2 Zellen auf eine K80 mit 24 GB Speicher, wobei dieser auf zwei GPUs aufgeteilt ist, von denen jede nur 12 GB hat (vgl. [47]). Realistisch liegen die Werte noch deutlich darunter, da Teile des Speichers durch den CUDA Treiber reserviert sind und Zustände der Zufallszahlengeneratoren und die Ergebnisse (der Detektor) gespeichert werden müssen. Auch ohne dies sind die maximalen Gittergrößen zu klein für aktuelle und zukünftige Simulationen. So wurde 2013 PIConGPU verwendet, um ein 8000×768^2 Zellen großes Volumen zu berechnen (vgl. [13]), sodass allein die Dichteverteilung rund 17,6 GB Speicher verbrauchen würde. Die Berechnung mit solchen Größenordnungen auf einer Grafikkarte ist nicht sinnvoll, da eine aufwendige Behandlung der Teile der Dichte und Photonen, die aktuell nicht im Grafikspeicher gehalten werden können, notwendig wäre, was sehr wahrscheinlich auch den Performancevorteil

[14] Funktion, die parallel von einem oder mehreren Threads auf der Grafikkarte ausgeführt wird (vgl. [44]).

zunichte macht. Demnach werden mehrere GPUs benötigt, von denen jede einen Teil des Gitters bearbeitet. Auch muss der Austausch von Teilchen zwischen den Grafikkarten möglich sein, wenn diese das entsprechende Teilgebiet verlassen. Zusammen mit der Überlegung, diese Simulation in PIConGPU zu integrieren, ist die Parallelisierung unter Nutzung von libPMacc die logische Konsequenz.
Abschließend zu diesem Kapitel soll an dieser Stelle auf die Bedeutung der Superzellen eingegangen werden. Deren Größe (in Zellen) wird bei der Übersetzung des Programms festgelegt und ist demnach konstant über die gesamte Laufzeit. Die Partikel werden in Arrays fester Größe gespeichert und den Superzellen zugeordnet, in denen sie sich befinden. Dabei entspricht die Größe eines Arrays (maximale Anzahl an Partikeln) der Anzahl der Zellen in einer Superzelle. Ein Eintrag in einem solchen Array wird über ein Flag als gültiges Partikel markiert, und mehrere solcher Arrays, in libPMacc „Frames“ genannt, werden als verkettete Liste gespeichert, wodurch eine dynamische Anpassung der Partikelanzahl möglich ist. Diese Konvention der Zuordnung ist vergleichbar mit dem räumlichen Sortieren der Partikel und ermöglicht eine hohe Lokalität. Genutzt wird diese, in dem die Superzellen auf CUDA-Blöcke abgebildet werden und z. B. bei der Bewegung jeder Block das zugehörige Dichtefeld effizient in den schnellen Shared Memory lädt. Die Partikel, die diesem Block zugeordnet sind, benötigen lediglich diese wenigen Werte, wodurch ein langsamer, wahlfreier Zugriff auf den Hauptspeicher vermieden wird. Über die Größe der Superzellen kann somit die Granularität der Sortierung, der notwendige Shared Memory und natürlich auch die Blockgröße verändert werden. Schließlich ermöglicht die räumliche Sortierung die einfache Erkennung von Partikeln, die das globale oder GPU-lokale Volumen verlassen haben, und entweder an (MPI-)Nachbarprozesse übergeben oder eventuell vom Detektor erkannt werden müssen. Dazu nutzt libPMacc das Konzept der „Border-“ und „Guard-“Zellen. Letztere sind die Superzellen, die gerade so außerhalb des lokalen Gitters sind. Innerhalb der Simulation werden sie auf die Borderzellen des jeweiligen Nachbarn abgebildet und außerhalb des globalen Gitters zur Behandlung von Partikeln verwendet, die man löschen und eventuell vom Detektor erkennen lassen muss.
Im Folgenden sollen die drei oben genannten Teile eines Zeitschritts der Reihe nach betrachtet werden. Anschließend wird eine spezielle Optimierung und die verwendeten Datenformate betrachtet.

4.3.2 Lichtquelle

Die Lichtquelle muss zu jedem Zeitschritt eine Anzahl an Photonen mit bestimmten Eigenschaften erzeugen und an geeigneten Stellen in das Simulationsvolumen setzen. Dazu müssen zuerst die Eigenschaften eines Partikels, dass ein Photon modellieren soll, definiert werden. Folgende Attribute (vgl. Abschnitt 4.2) sind dazu notwendig:

- Position innerhalb der Simulation (aufgeteilt nach globaler und Inzell-Position)
- Bewegungsrichtung
- Phase (benötigt zur Transformation zwischen Teilchen und Welle)

Wichtig sind außerdem noch konstante Eigenschaften (Flags), die für alle Photonen gleichermaßen gelten:

- Geschwindigkeit (Lichtgeschwindigkeit)
- Wellenlänge (Annahme sind elastische Stöße, damit bleibt die Wellenlänge konstant)
- Art der Streuung (Policies zur Berechnung der Wahrscheinlichkeit und Richtung)

Das Wichtigste hier ist die Phase, welche das Photon von den üblicherweise verwendeten und realen Partikeln unterscheidet und es ermöglicht, aus einer hohen Anzahl an Partikeln wieder eine Welle zu rekonstruieren bzw. die Interferenz zu beschreiben, was mit realen Partikeln nicht möglich ist[15]. Äquivalent dazu kann auch eine Startzeit oder etwas Ähnliches gespeichert werden, womit sich die Flugzeit des Photons bestimmen lässt. Hier die Phase zu nutzen hat jedoch den Vorteil, dass sich in zukünftigen Erweiterungen Prozesse berücksichtigen lassen, welche die Phase verändern. Außerdem ermöglicht es eine Optimierung im Hinblick auf die Genauigkeit, die in Unterabschnitt 5.3.1 beschrieben wird.
Um möglichst viele verschiedene Experimente simulieren zu können, wurde die Beschreibung der Lichtquelle sehr allgemein gehalten. Die einzige Einschränkung ist die Erzeugung der Photonen in einer Ebene von Zellen, was aber meist dem natürlichen Experimentaufbau entspricht, in der eine Probe von einer Seite bestrahlt wird. Um die Lichtquelle zu spezifizieren, muss die Verteilung der Photonen über Raum und Zeit bekannt sein. Damit lassen sich folgende Eigenschaften für Photonen, die in der Ebene bei $x = 0$ erzeugt werden, zusammen mit ihren Abhängigkeiten [16] angeben:

Anzahl	$\leftarrow y, z, t$
Inzell-Position	$\leftarrow y, z, t, n$
Phase	$\leftarrow y, z, t, n$
Bewegungsrichtung	$\leftarrow y, z, t, n$

Wie daraus schon ersichtlich ist, wird für jede Zelle zuerst die Anzahl der Photonen, die in diesem Zeitschritt zu platzieren sind, bestimmt und anschließend für jedes der erzeugten Photonen die jeweilige Eigenschaft gemäß der als Policy gegebenen Funktion gesetzt. Dabei bezeichnet y, z die Position der Zelle, t den Zeitpunkt zu dem die Partikel erzeugt werden und n die Nummer des Partikels, das zu dem gegebenen Zeitpunkt und Position erzeugt wird. Für den häufigen Fall eines Lasers, der als konstante ebene Welle auf die gesamte Fläche trifft, ist Anzahl und Bewegungsrichtung jeweils eine Konstante. Die Phase wird über $-\omega t + \varphi_0$ berechnet (vgl. Gleichung (3.4)) und die Position in der Ebene wird zufällig gewählt, was einerseits quantenmechanisch interpretiert werden kann, wonach von Photonen nur die Aufenthaltswahrscheinlichkeiten, nicht aber die genaue Position bekannt ist. Andererseits entspricht es aber auch dem Monte-Carlo Ansatz, nachdem man alle möglichen Wege mit einer gewissen Wahrscheinlichkeit betrachten will und damit auch alle möglichen Anfangspositionen in der Ebene ermöglicht werden müssen, wobei die Wahrscheinlichkeitsverteilung über diese durch die Anzahl pro Zelle realisiert wird[17].
Mit diesen Formalismen lassen sich die meisten Fälle abdecken. Phasensprünge sind möglich und auch schräg zur Eintrittsebene propagierende Photonen können modelliert werden. Auch gaußförmige Intensitätsverteilung des Lasers sind über die Anzahl möglich, wobei die Verteilung auf Zellgrößen diskretisiert ist. Eine feinere Aufteilung über die Beschränkung der möglichen Inzell-Positionen ist trotzdem möglich, erfordert aber etwas mehr Aufwand. Generell kann durch den Parameter n eine höhere Abtastung realisiert werden, wenn die Auflösung auf Zellen nicht ausreicht.

[15] Die Beschreibung der Interferenz von Partikel „mit sich selbst“ ist jedoch über die quantenmechanische Betrachtung der Wahrscheinlichkeitsfunktion ihres Aufenthaltsortes möglich. Das führt hier aber zu weit und es sei darum auf Literatur wie [48] verwiesen.
[16] $x \leftarrow y$ bedeutet: x ist abhängig von y
[17] Hinweis zu der Formulierung „alle mögliche“: Wege, die nicht betrachtet werden (sollen), können über eine Wahrscheinlichkeit von null ausgeschlossen werden und sind somit nur ein Spezialfall der Betrachtung.

Die Implementation des Kernels, der die Partikel erzeugt, ist relativ geradlinig und zeigt sehr gut die Verwendung der Superzellen. Zur besseren Veranschaulichung des Prinzips dient das Struktogramm in Abb. 4.3. Hauptsächlich interessant bei der Untersuchung von parallelem Code sind Synchronisierung,

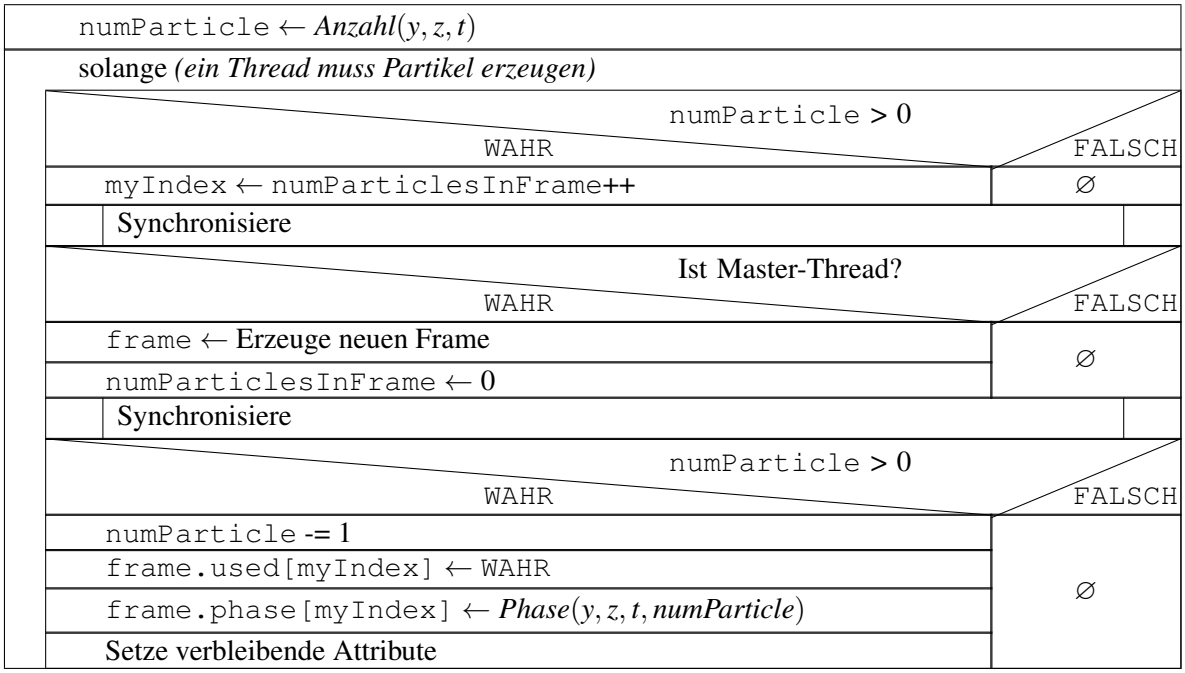


Abbildung 4.3: Struktogramm zum Kernel zur Erzeugung von Partikeln

sequentielle Teile und, insbesondere in Hinblick auf CUDA, Verzweigungen. Kernstück dieses Algorithmus' ist eine Schleife, die solange durchlaufen wird, wie ein Thread noch Partikel erzeugen muss. Der Hauptgrund, dass auch Threads die keine Partikel mehr erzeugen den Schleifenrumpf ausführen müssen, ist die erforderliche Synchronisation, denn der Aufruf der Synchronisationsfunktion „[...] is allowed in conditional code but only if the conditional evaluates identically across the entire thread block[...]“ [44, Synchronization Functions], wozu auch die Schleife zählt. Auch wird der sogenannte „Master-Thread“, ein festgelegter Thread des Blocks, benötigt, um neue Frames (Arrays für Partikeldaten) zu erzeugen. Denn um den Speicher optimal auszunutzen, füllt jeder Block vorhandene Frames vollständig, bevor der nächste Frame angelegt wird. Im Algorithmus werden dazu maximal zwei Frames (ein teilweise gefüllter und ein leerer) gleichzeitig genutzt, wobei der dafür notwendige Code in dem Struktogramm zur Vereinfachung entfernt wurde. Das Erzeugen des neuen Frames übernimmt libPMacc, sodass hier nur ein Funktionsaufruf nötig ist, damit der Speicher alloziert und das neue Array in die verkettete Liste angefügt wird. Zur Erkennung, wann ein neuer Frame benötigt wird, nutzt der Master-Thread die Variable `numParticlesInFrame`, welche, zusammen mit der Variable `frame`, im gemeinsamen Speicher liegt. Die Operation zur Erhöhung muss dabei ein atomares Inkrement sein, damit es hier nicht zu Race Conditions („Wettlaufsituation“) kommt. Als Rückgabewert erhält man dabei den Wert unmittelbar vor der Erhöhung, womit jeder Thread, der dies ausführt einen eigenen Wert erhält, der direkt als Index genutzt werden kann.

Schließlich wird an dieser Stelle noch die Divergenz im Algorithmus betrachtet, also die Situationen, wenn Threads eines Warps (oder Blocks) unterschiedliche Code-Teile ausführen. Es ist offensichtlich, dass die Erzeugung eines Frames zur Divergenz führt, da dies nur von einem einzigen Thread gemacht

wird. Dies lässt sich jedoch nicht vermeiden, wenn man nicht unnötig viele Frames erzeugen möchte, was den knappen Hauptspeicher belasten würde. Auch ist die Nutzung eines Frames für alle Threads des Blocks sinnvoll, da somit in der Regel zusammenhängende Speicherblöcke geschrieben werden. Dies wird vor allem deshalb ermöglicht, weil zur Umsetzung des atomaren Inkrements ein optimierter Algorithmus von Adinetz[49] genutzt wird, der die implizite Synchronisation der Threads in einem Warp ausnutzt, um eine schnellere Inkrement-Operation zu ermöglichen und als Nebeneffekt aufeinanderfolgende Werte für Threads eines Warps erzeugt (vgl. [49]). Außerdem betrifft die Divergenz hier nur einen Warp des Blocks, die anderen Warten am Synchronisationspunkt, sodass der Multiprozessor die Ressourcen für andere Warps nutzen kann. Demnach überwiegen hier klar die Vorteile, sodass diese Divergenz in Kauf genommen werden kann.
Kritischer ist die Situation, wenn die Threads unterschiedlich viele Partikel erzeugen, denn durch die Verzweigungen in der Schleife (`numParticle > 0`) führt dies zur Divergenz innerhalb der Warps und auch über die Warps eines Blocks, da alle Threads die Schleife solange ausführen müssen, bis alle anderen Threads des Blocks ihre Partikel erzeugt haben. In der Praxis ist dies jedoch durch die räumliche Lokalität der Zellen, auf denen die Threads operieren, meist vernachlässigbar. Dazu geht man davon aus, dass die eintreffende Lichtwelle eine einigermaßen glatte Intensitätsverteilung hat, sich diese also nur langsam ändert und so in benachbarten Zellen ähnlich viele Photonen erzeugt werden. Sollte dies nicht der Fall sein, kann man die Zellen verkleinern, was das Sampling erhöht und damit die Verteilung über der gleichen Anzahl an Zellen glättet. Wenn auch das nicht möglich ist, muss dieser Algorithmus überarbeitet werden. Da im Experiment die Elektronendichte möglichst genau und vollständig untersucht werden soll, wird man sicherlich bestrebt sein, die Ausleuchtung möglichst gleichmäßig zu realisieren, womit eine glatte Verteilung gegeben ist. Schließlich lässt sich noch feststellen, dass Divergenz innerhalb eines Warps sich auch hier stärker negativ auf die Performance auswirkt (s. dazu die Betrachtung der SIMT Architektur in Abschnitt 4.1), als die innerhalb eines Blocks, da bei letzterer einige Warps große Teile des Schleifenrumpfs überspringen und so die Rechenkapazitäten nur wenig belastet werden. Jedoch ist auch die Divergenz innerhalb eines Warps weniger wahrscheinlich oder schwächer, da hier die räumliche Lokalität höher ist, als über den ganzen Block.
Zusammenfassend ist zu erwarten, dass für nicht allzu große Blöcke eine gute Performance erreicht wird. Und auch hier kann über die Blockgröße das Verhältnis von zusätzlichen Speicherverbrauch (nicht voll besetzte Frames) zu zusätzlicher Laufzeit (Overhead durch Divergenz) angepasst werden.

4.3.3 Propagation

Die Propagation erfolgt in zwei Schritten: Zuerst wird der Einfluss der Elektronendichte betrachtet und das Photon gegebenenfalls abgelenkt, danach wird es weiter bewegt.
Für die Wirkung des Dichtefeldes auf die Photonen betrachtet man den Streuquerschnitt σ und den differentiellen Wirkungsquerschnitt $\frac{d\sigma}{d\Omega}$, welche in Abschnitt 3.2 erklärt wurden. Die Wahrscheinlichkeit dafür, dass ein Photon in einer Zelle abgelenkt wird, ist damit bis auf einen Vorfaktor durch die Elektronendichte gegeben, welche innerhalb einer Zelle konstant ist. Für starke Dynamiken in der Dichte, also wenn sie sich über benachbarte Zellen stark ändert, kann auch eine Interpolation basierend auf dem Ort des Photons und der Dichte der umgebenden Zellen erfolgen. Hier sollen konstante Dichten je Zelle angenommen werden, um das Prinzip zu erläutern. Zur Entscheidung, ob ein Photon abgelenkt wird oder seine aktuelle Richtung beibehält, muss lediglich in jedem Zeitschritt eine Zufallszahl je Photon

bestimmt und mit der Elektronendichte in der jeweiligen Zelle verglichen werden. Der genaue Wert, ab dem das Photon abgelenkt wird, ist abhängig von der Definition bzw. Skalierung des Dichtefeldes und des Streuquerschnitts aus Gleichung (3.8), wobei im einfachsten Fall das Produkt aus Elektronendichte und Streuquerschnitt gebildet werden muss (vgl. Abschnitt 3.2). Diese Vorgehensweise, nur den Wert der Zelle zu verwenden, in dem sich das Photon befindet, muss allerdings kritisch betrachtet werden. Um ganz exakt zu sein, muss das Photon auf jedem Punkt der Flugbahn abgelenkt werden können, wobei die Wahrscheinlichkeit für einen Ort von der Dichte (bzw. dem Streuquerschnitt) abhängt, die an jedem Punkt des Weges betrachtet werden müsste. Hier wäre das insbesondere beim Wechsel von einer Zelle in die Nachbarzelle notwendig. Aus Gründen der Performance möchte man das vermeiden. Dazu wählt man ausreichend kleine Zellen, so dass sich die Dichte über Nachbarzellen nur wenig ändert, und einen ausreichend kleinen Zeitschritt Δt, mit dem effektiv der Weg eines Photons gesampelt wird. LibPMacc setzt dabei bereits voraus, dass der Zeitschritt so klein ist, dass der zurückgelegte Weg in einem solchen Schritt maximal einer Zellgröße entspricht. Als Konsequenz wird damit bereits die Dichte jeder Zelle, die das Photon durchquert, betrachtet, mit Ausnahme kleiner Teilstücke bei einer Bewegungsrichtung schräg gegenüber der Zellausrichtung. Bei ausreichend geringer Änderung der Dichte zwischen benachbarten Zellen kann man das vernachlässigen, insbesondere, weil der Weganteil, der durch die übersprungene Zelle führt, sehr klein gegenüber dem Gesamtweg ist. Benötigt man dennoch eine höhere Auflösung, kann man immer noch den Zeitschritt verkürzen, was zu Lasten der Simulationszeit geht.

Die Ablenkung selbst wird meist mit Kugelkoordinaten beschrieben was Abb. 4.4 veranschaulicht. Die

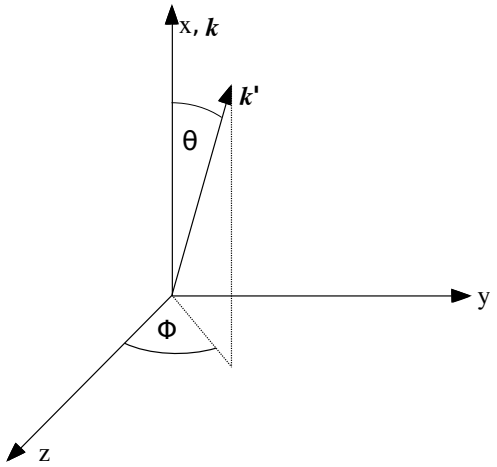


Abbildung 4.4: Visualisierung des Koordinatensystems bei der Ablenkung von Photonen

Achsenbezeichnungen sind dabei so gewählt, dass die x-Achse entlang der Ausbreitungsrichtung des eintreffenden Photons (beschrieben durch $\underline{\boldsymbol{k}}$) verläuft. Natürlich sind diese Bezeichnungen frei wählbar und dienen hier nur zur Definition der Winkel. Mit $\underline{\boldsymbol{k'}}$ als Richtung des gestreuten Strahls ergibt sich der Streuwinkel θ und der Azimutwinkel ϕ. Deren Verteilung lässt sich mittels des differentiellen Wirkungsquerschnitts bestimmen und ist bei der Thomson Streuung proportional zu $\cos^2(\theta)$ und rotationssymmetrisch zur Richtung des eintreffenden Photons (vgl. Abschnitt 3.2).

Die neue Richtung des gestreuten Photons ergibt sich somit, in dem man einen gleich verteilten Winkel ϕ im Intervall $[0, 2\pi)$ und einen Winkel θ im Intervall $[0, \pi]$ mit einer entsprechend des Wirkungsquerschnitts berechneten Wahrscheinlichkeit wählt und den alten Richtungsvektor passend rotiert. Aufgrund

des Aufbaus des Experiments, der Beobachtung, dass hauptsächlich Photonen mit kleinen Ablenkungswinkeln θ vom Detektor aufgefangen werden und der näherungsweise konstanten Wahrscheinlichkeit bei kleinen Winkeln, wird für die Tests eine Gleichverteilung über die möglichen Raumwinkel bei einem kleinen maximalen Ablenkungswinkel θ_{max} verwendet. Dies erhöht die Effizienz, da stark abgelenkte Photonen, die den Detektor nur sehr selten erreichen, nicht berechnet werden müssen. In Kapitel 6 kann gezeigt werden, dass dies zumindest für Einfachstreuung sehr gute Ergebnisse bringt. Die Bestimmung eines zufälligen Streuwinkels θ_r aus dem Intervall $[0, \theta_{max})$, mit dem eine Gleichverteilung auf einer Kugeloberfläche (und damit den Raumwinkeln) realisiert wird, erfolgt aus einer gleichverteilten Zufallszahl r aus dem Intervall $[0, 1)$, die von dem in libPMacc enthaltenem Zufallszahlengenerator (s. Abschnitt 4.4) erzeugt wird. Die verwendete Formel ist dabei:

$$\theta_r = \arccos(r(\cos(0) - \cos(\theta_{max})) + \cos(\theta_{max})) \tag{4.2}$$

Es lässt sich zeigen, dass dies die gewünschte Gleichverteilung erzeugt, was jedoch nicht Teil dieser Arbeit ist. Die Bestimmung des Streuereignisses und der Streuwinkel ist auch über Policies konfigurierbar, so dass dies einfach geändert werden kann.

Der Richtungsvektor liegt in der Simulation als $\frac{\underline{\boldsymbol{k}}}{k} = \begin{pmatrix} x & y & z \end{pmatrix}^T$ und damit als Einheitsvektor vor, wobei hier das Koordinatensystem der Simulation und nicht des Photons verwendet wird. Die Verwendung von Einheitsvektoren bietet mehrere Vorteile. Zum Einen muss bei der Bewegung der Betrag des Richtungsvektors nicht neu berechnet werden, was eine langsame Wurzeloperation benötigen würde. Zum Anderen lässt sich auch die zur Richtungsänderung notwendige Rotationsmatrix vereinfachen, was schließlich in Gleichung (4.3) resultiert. Diese Formel erhält dabei die Eigenschaft des Einheitsvektors, so dass diese nicht zusätzlich wiederhergestellt werden muss. Aufgrund numerischer Ungenauigkeiten kann es jedoch trotzdem notwendig sein, was in Unterabschnitt 5.3.2 auf Seite 59 betrachtet wird. Die Herleitung von Gleichung (4.3) ist etwas länglich und soll an dieser Stell nur kurz skizziert werden: Ist ein Einheitsvektor $\underline{\boldsymbol{a}}$ gegeben, lässt sich eine Rotationsmatrix $\underline{\boldsymbol{R_a}}$ finden, die den Einheitsvektor $\underline{\boldsymbol{e_x}} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$ auf $\underline{\boldsymbol{a}}$ abbildet, was eine Rotation um den Vektor $\underline{\boldsymbol{a}} \times \underline{\boldsymbol{e_x}}$ mit dem Winkel $\arccos(\underline{\boldsymbol{a}} \cdot \underline{\boldsymbol{e_x}})$ darstellt. Entsprechende Vereinfachungen mit Hilfe der Eigenschaft $|\underline{\boldsymbol{a}}| = 1 \Leftrightarrow a_x^2 + a_y^2 + a_z^2 = 1$ helfen die Matrix kompakter darzustellen. Es lässt sich außerdem leicht zeigen, dass der um θ und ϕ rotierte Einheitsvektor $\underline{\boldsymbol{e_x}}$ sich zu $\underline{\boldsymbol{b}} = \underline{\boldsymbol{R_b}}\,\underline{\boldsymbol{e_x}} = \begin{pmatrix} \cos(\theta) & \sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) \end{pmatrix}^T$ ergibt. Damit kann man diesen rotierten Vektor in das Koordinatensystem von $\underline{\boldsymbol{a}}$ rotieren, indem man ihn linksseitig mit $\underline{\boldsymbol{R_a}}$ multipliziert. Dies führt zu der Form in Gleichung (4.3).

$$\underline{\boldsymbol{k'}} = \begin{pmatrix} x\cos(\theta) - y\sin(\phi)\sin(\theta) - z\cos(\phi)\sin(\theta) \\ y\cos(\theta) + (x + \frac{z^2}{1+x})\sin(\phi)\sin(\theta) - \frac{yz}{1+x}\cos(\phi)\sin(\theta) \\ z\cos(\theta) + (x + \frac{y^2}{1+x})\cos(\phi)\sin(\theta) - \frac{yz}{1+x}\sin(\phi)\sin(\theta) \end{pmatrix} \tag{4.3}$$

Der Fall $x = -1$ muss auf Grund der Divisionen gesondert betrachtet werden. Durch die Voraussetzung, dass ein Einheitsvektor gegeben ist, also $\sqrt{x^2 + y^2 + z^2} = 1$ gilt, ist für diesen Fall $y = z = 0$ und die Berechnung ist wie folgt:

$$\underline{\boldsymbol{k'}} = \begin{pmatrix} x\cos(\theta) \\ x\sin(\phi)\sin(\theta) \\ x\cos(\phi)\sin(\theta) \end{pmatrix} \tag{4.4}$$

Im Folgenden sei mit $\underline{\boldsymbol{k}'}$ die aktuelle Richtung des Photons mit $|\underline{\boldsymbol{k}'}| = 1$ bezeichnet, was über $\underline{\boldsymbol{k}} = k \cdot \underline{\boldsymbol{k}'}$ im Zusammenhang mit dem aktuellen Wellenvektor steht. Außerdem soll die für Zuweisungen übliche Notation der Form $\underline{\boldsymbol{k}'} \leftarrow f(\underline{\boldsymbol{k}'})$ als Kurzform für $\underline{\boldsymbol{k}'_{\text{neu}}} \leftarrow f(\underline{\boldsymbol{k}'_{\text{alt}}})$ verwendet werden, wobei f die Rotation beschreibt. Die Unterscheidung zwischen eintreffenden und ausgehenden Wellenvektor ergibt sich aus dem Kontext der Verwendung oder wird explizit angegeben.
Wurde die Richtung bestimmt, kann die neue Position des Partikels aus der alten und dem um die Geschwindigkeit skalierten Richtungsvektor bestimmt werden, welche bei Photonen die Lichtgeschwindigkeit c ist. Demzufolge gilt für die Position $\underline{\boldsymbol{p}}$:

$$\underline{\boldsymbol{p}} \leftarrow \underline{\boldsymbol{p}} + \frac{\boldsymbol{k}}{k} c\Delta t = \underline{\boldsymbol{p}} + \underline{\boldsymbol{k}'} c\Delta t \tag{4.5}$$

Die Implementation als Kernel ist wieder sehr geradlinig: Für jedes Partikel wird eine Zufallszahl ermittelt und abhängig davon bestimmt, ob es abgelenkt wird. Wenn ja, wird die neue Richtung ermittelt. In jedem Fall wird danach die Position entsprechend der (eventuell geänderten) Richtung berechnet. Gegebenenfalls werden Partikel neuen Zellen oder Superzellen zugeordnet, indem die Inzell-Position komponentenweise mit dem Intervall $[0, 1)$ verglichen und der lokale Zellindex (relativ zum Ursprung der Superzelle) angepasst wird. Ein weiterer Kernel wird gestartet, wenn mindestens ein Partikel eine Superzelle verlassen hat und in einen Frame einer benachbarten Superzelle verschoben werden muss. Dieser Teil des Algorithmus ist fast vollständig aus PIConGPU übernommen bzw. in libPMacc enthalten, sodass der hier nicht weiter betrachtet werden soll. Interessant für die Funktionsweise ist jedoch die Behandlung der Partikel, die im Folgenden skizziert werden soll.
Wie in der Einführung zu den Superzellen in Unterabschnitt 4.3.1 beschrieben, werden wieder Blöcke der Größe einer Superzelle gestartet. Jeder Thread des Blocks lädt einen Wert des Dichtefeldes in den gemeinsamen Speicher, sodass er von allen Threads des Blocks genutzt werden kann. Anschließend findet ein Domainwechsel statt, in der die Zuordnung der Threads zu Gitterzellen aufgehoben wird und sie stattdessen Partikeln zugeordnet werden. Dazu iterieren die Threads über die verkettete Liste der Frames und verarbeiten in jeder Iteration jeweils ein Partikel, wobei es nur im letzten Frame sein kann, dass ein Thread kein gültiges Partikel hat und auf die anderen warten muss. Genau hierfür wird immer sichergestellt, dass jeder Frame von null beginnend lückenlos gefüllt ist (gültige Partikel enthält), weshalb durch „fehlende“ Partikel erzeugte Divergenzen innerhalb eines Warps auf maximal einen Warp je Block beschränkt sind. Die Dichten, die zur Bestimmung der Streuwahrscheinlichkeit notwendig sind, liegen für jedes dieser Partikel bereits im schnellen gemeinsamen Speicher. Die Neuberechnung, inklusive dem Lesen und Schreiben von Position und Richtung erfolgt durch die Zuordnung aufeinanderfolgender Threads zu aufeinanderfolgenden Partikeln (bzw. Einträgen in den Frames) effizient parallel und unter optimaler[18] Ausnutzung der Bandbreite zum Hauptspeicher. Damit ist die einzige Schwachstelle die Streuung selbst, da hier hohe Divergenzen bei der Bestimmung der neuen Richtung auftreten können, wenn nur wenige Partikel des Warps gestreut werden. Hier wurde entschieden, das in Kauf zu nehmen, um die allgemeine Form des Algorithmus und der Datenaufteilung aus Kompatibilität zu PIConGPU beizubehalten, anstatt zum Beispiel die Threads umzusortieren, sodass solche, welche die Streuung berechnen müssen, zusammenhängend sind. Auch würde das deutlich mehr Aufwand bei der Entwicklung und Synchronisierung zur Laufzeit erfordern. Stattdessen wurde die Formel zur Rotation des Bewegungsvektors so weit wie

[18] Amortisiert betrachtet für eine ausreichend große Anzahl an Partikeln.

möglich vereinfacht, um Rechenoperationen zu sparen. Auch wurde der Algorithmus so entworfen, dass er nach dem Lesen der Dichte keine Synchronisierung braucht, sodass die Divergenz auf Warps begrenzt bleibt. Die Basis dazu bildet die feste Zuordnung von Threads zu Einträgen in den Frames und die Nutzung eines nachfolgenden Kernels für die Verschiebung von Partikels zwischen Superzellen und der damit verbundenen Modifizierung der verketteten Liste. Dadurch ist keine Kommunikation zwischen den Threads notwendig und die Liste selbst wird nicht verändert, sodass sie parallel gelesen werden kann.
Im Anschluss an die eventuelle Neuzuordnung der Partikel auf Superzellen findet die notwendige Kommunikation statt, in der Partikel, die das lokale Volumen einer GPU verlassen und in den Guard-Zellen gelandet sind, auf die entsprechenden Nachbarzellen verteilt werden. Dazu wird von libPMacc für jede der 26 Richtungen (alle, auch schräg, angrenzende Zellen) ein unabhängiger Task gestartet, die parallel ablaufen können. Diese Kommunikation ist notwendig und lässt sich auch nur begrenzt durch anderweitige Berechnungen überdecken. Mögliche wäre es, die neuen Partikel bereits während der Kommunikation zu erzeugen, jedoch betrifft das nur einen Teil der GPUs. Deshalb wird dieser Datenaustausch die Performance negativ beeinflussen. Aufgrund der festen Anzahl an Kommunikation ist jedoch zu erwarten, dass das Weak Scaling davon nur unwesentlich beeinflusst wird, da in dem Fall die ausgetauschte Datenmenge gleich bleibt. Im Fall des Strong Scalings wird jedoch der Kommunikationsoverhead (moderat) wachsen.

4.3.4 Detektion

Dieser Teil der Simulation wird ausgeführt, wenn Photonen das Simulationsvolumen verlassen, was dann der Fall ist, wenn sie sich in den Guard-Zellen befinden, die GPU aber in der entsprechenden Richtung keine Nachbar-GPU hat. Geprüft werden diese Zellen in jedem Zeitschritt zusammen mit allen anderen Guard-Zellen, wobei je nachdem, ob eine Nachbar-GPU vorhanden ist, eine von zwei Policies verwendet wird. Aufgrund der Trennung in die verschiedenen Richtungen entsteht dabei keine Divergenz innerhalb eines Kernels, da pro Richtung ein Kernel gestartet wird (s. Ende des letzten Kapitels). In dieser Simulation tauscht die erste Policy Daten mit dem Nachbarn aus und die zweite leitet die Partikel an den Detektor weiter. Zu bemerken ist hier, dass in der Standardeinstellung Partikel außerhalb der Simulation gelöscht werden, wobei auch andere Behandlungen möglich sind. Eine Variante ist das Zählen der Partikel, um z. B. Verluste zu berechnen, oder das Reflektieren an der Grenze, wobei die Partikel in die Simulation zurückgeführt werden. Diese Flexibilität zeigt die Stärke der Template Metaprogrammierung, da durch die hier verwendeten Policies sehr einfach Algorithmen ausgetauscht werden können, ohne dass durch die Abstraktion Overhead zur Laufzeit entsteht.
Für jedes der Photonen außerhalb des Simulationsgebietes muss geprüft werden, ob und wenn ja wo und wann sie auf den Detektor treffen. Dafür wird ein Raytracing Ansatz verwendet und der Schnittpunkt des ausgehenden Strahls, gegeben durch Ort und Flugrichtung des Photons, mit der Detektor-Ebene berechnet. Wenn dieser existiert und in Vorwärtsrichtung des Strahls liegt, kann über die Distanz und Geschwindigkeit die Zeit des Auftreffens berechnet und dessen Effekt auf den Detektor simuliert werden. Dieser wird als zweidimensionale Anordnung von Zellen betrachtet (vgl. Abb. 4.2), von denen jede nach dem gleichen Prinzip und unabhängig von den anderen agiert. Die genaue Funktionsweise hängt von der Art des Detektors ab. Häufige Varianten sind das Zählen von Photonen und das Integrieren des Feldes über der Zeit. Beide sind implementiert und können mittels entsprechender Policies aktiviert werden.
Im Folgenden wird ein integrierender Detektor betrachtet. Dieser misst die Intensität des Lichtes, die

nach Gleichung (3.7) proportional zum Betragsquadrat des E-Feldes ist. Da sich die Amplitude über der Zeit ändern kann, betrachtet man die Intensität als den zeitlichen Mittelwert des elektrischen Feldes über einer Periode T, wobei der Zusammenhang in Zinth und Zinth [5, S. 10 f.] erläutert wird und in Gleichung (4.6) resultiert. Dazu ist zu sagen, dass der Brechungsindex n hier wie bisher als eins angenommen wird, die spitzen Klammern die Bildung des zeitlichen Mittelwertes beschreiben und hier nicht nur eine Schwingungsperiode als T bezeichnet wird, sondern die gesamte Messzeit, wie es auch in Stößel [6, S. 49 f.] der Fall ist. Diese Periode ist dabei im einfachsten Fall die komplette Simulationsdauer.

$$I = \epsilon_0 cn\langle|\underline{\boldsymbol{E}}|^2\rangle \tag{4.6}$$

Man kann dies auch als Messung der (mittleren) Energie auffassen, denn die „[...] Strahlungsflussdichte oder Lichtintensität I [...] ist also die mittlere Lichtenergie pro Zeit und pro (senkrecht zur Ausbreitungsrichtung stehender) Fläche." [5, S. 11] Sie ist durch die Wellengleichung abhängig von Zeit und Ort, wobei für die Zeit feste Intervalle angenommen werden. Das heißt, nach der Mittelung über eine Periode T an einem Ort $\underline{\boldsymbol{r}}$ steht die Intensität zum Zeitpunkt $t = mT$ fest, wobei m eine Ganzzahl ist. Damit muss an einem Ort auf dem Detektor über das Quadrat der Wellengleichung (3.1), bzw. dem Betragsquadrat der komplexen Wellengleichung Gleichung (3.3) integriert werden. Es wird angenommen, dass die Detektorzellen klein genug sind, damit die Energieunterschiede auf der Fläche einer Zelle vernachlässigt werden können, andernfalls lässt sich die Verteilung nicht exakt abbilden oder man muss zusätzlich innerhalb der Zellen sampeln, was hier nicht betrachtet werden soll.

In der Simulation bedeutet das, dass zwei Summen benötigt werden. Die erste Summe über alle Photonen, die zu einem Zeitpunkt t an einem Punkt $\underline{\boldsymbol{p_d}}$ auf dem Detektor ankommen, drückt die Interferenz der durch die Photonen repräsentierten huygensschen Elementarwellen aus. Die zweite summiert die Betragsquadrate der interferierten Wellen über mehrere Zeitschritte, um auf die Intensität zu kommen, was der diskretisierten Integration entspricht. Fasst man die Vorfaktoren, wie die Periodendauer T und diejenigen aus Gleichung (4.6), in einen gemeinsamen Vorfaktor p_c zusammen ergibt sich für die Intensität:

$$I(\underline{\boldsymbol{p_d}}, T) = p_c \sum_{t=1}^{T} \Big| \sum_{n=1}^{N_t(\underline{\boldsymbol{p_d}})} \mathrm{e}^{\mathrm{i}\varphi_n}\Big|^2 \tag{4.7}$$

Aus der Feststellung, dass nur Photonen, die zum gleichen Zeitpunkt am selben Ort sind, miteinander interferieren ergibt sich ein Problem: Aufgrund der unterschiedlichen Austrittswinkel und Position der Photonen und der dadurch verschiedenen Weglängen zum Detektor, kommen selbst Photonen eines Zeitschrittes nicht zum gleichen Zeitpunkt auf dem Detektor an. Auch sind die Auftreffpunkte innerhalb einer Detektorzelle, wie man sie durch das Raytracing unter Nutzung des Richtungsvektors beim Austritt des Photons ermittelt, unterschiedlich und können so um bis zu eine Zellgröße voneinander abweichen. Da die Phase des Photons auf dem Detektor u. a. von der zurückgelegten Distanz abhängt, wird dadurch der Beitrag des Photons wesentlich verändert.

Zur Lösung des Ortsproblems wird nur ein Punkt $\underline{\boldsymbol{p_d}}$ in der Detektorzelle als Auftreffpunkt erlaubt. Das heißt, Photonen können nur an diesem Punkt in der Zelle landen und miteinander interagieren, was äquivalent zu der obigen Festlegung ist, kein Sampling innerhalb einer Detektorzelle zu nutzen. Ein Nachweis, dass dies das Ergebnis nicht wesentlich beeinflusst, soll hier mit Hilfe von Abb. 4.5 skizziert werden. Das Photon repräsentiert eine ebene Welle entlang seiner Flugbahn, deren Phase demnach

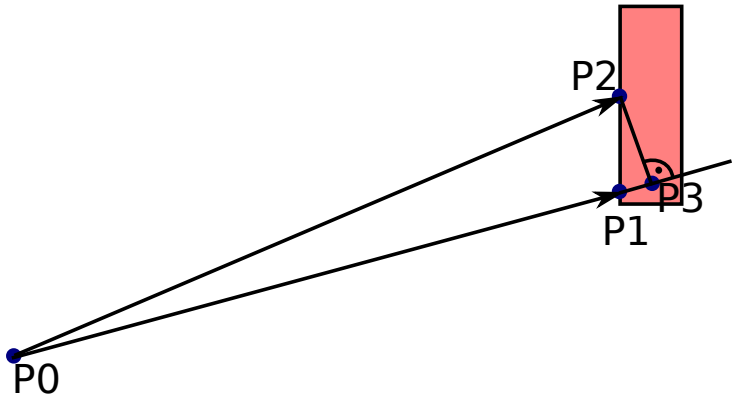


Abbildung 4.5: Modifizierte Flugbahn eines Photons vom Punkt P0 in eine Detektorzelle (roter Kasten)

konstant an Ebenen senkrecht zu dieser Flugbahn ist. Verlängert man nun die Strecke vom Streupunkt P0 des Photons zum Schnittpunkt P1 mit der Detektorebene so weit, dass sich der festgelegte Punkt P2 (äquivalent zu obigen $\underline{\boldsymbol{p_d}}$) in einer solchen Ebene mit dem neuen Endpunkt P3 befindet, erhält man in unendlicher Entfernung die Beziehung $\overline{\text{P0P3}} = \overline{\text{P0P2}}$, da die Strahlen im Unendlichen parallel sind. Dies gilt in der Simulation näherungsweise, da das Fernfeld angenommen wird, d. h. der Abstand vom Streupunkt zum Detektor ist groß und die Detektorzellen sind klein. Verstärken lässt sich das Argument bei Betrachtung der bedingten Wahrscheinlichkeit einen speziellen Punkt in einer Detektorzelle zu treffen, wenn diese schon erreicht wird. Für kleine Detektorzellen erhält man (näherungsweise) eine Gleichverteilung. Zusammen mit der Betrachtung der Fehler, die sich durch Annahme von $\overline{\text{P0P3}} \approx \overline{\text{P0P2}}$ im Endlichen ergeben, lässt sich zeigen, dass sich diese für ausreichend viele Partikel gegenseitig aufheben. Mit dieser Argumentation lässt sich auch zeigen, dass für eine sehr große Anzahl an Partikeln und ausreichend kleinen Detektorzellen (die Intensität innerhalb einer solchen ändert sich nicht), auch die direkte Strecke (in diesem Beispiel) $\overline{\text{P0P1}}$ betrachtet werden kann. Die Beschränkung auf einen diskreten Punkt auf der Zelle erlaubt jedoch eine Optimierung der Genauigkeit, die in Unterabschnitt 5.3.3 auf Seite 63 erläutert wird und ermöglicht eine bessere Konvergenz bei weniger Partikeln, da die Varianz der Streckenlängen reduziert wird.

Nun wird die Lösung des zeitlichen Problems betrachtet, wobei die Phase des Photons ausgenutzt wird. Mit dem Abstand d_n zwischen dem Ort des Photons beim Austritt aus der Simulation und $\underline{\boldsymbol{p_d}}$ lässt sich die Zeitdauer Δt_n bestimmen, die es bis zum Detektor braucht:

$$\Delta t_n = \frac{d_n}{c} \tag{4.8}$$

Die Phase φ'_n des Photons beim Verlassen des Volumens kann man aus der Startphase φ_{n0} und der Flugzeit $\Delta t'_n$ durch die Simulation berechnen (vgl. Gleichung (3.4)):

$$\varphi'_n = \varphi_{n0} - \omega \Delta t'_n \tag{4.9}$$

Nimmt man nun eine unendlich ausgedehnte Welle an, dann wäre deren aktuelle Phase am Ort $\underline{\boldsymbol{p_d}}$:

$$\varphi_n = \frac{\omega}{c} d_n + \varphi'_n \tag{4.10}$$

Hierbei kann der erste Term auch durch $\omega\Delta t_n$ ausgedrückt werden, wobei Δt_n als zeitliche Verschiebung von Gleichung (3.4) anzusehen ist und daher mit negativen Vorzeichen eingeht. Dies zeigt, dass die Phase auf dem Detektor linear abhängig von der Flugzeit vom Ursprung des Photons bis zur Detektorzelle ist ($\varphi_n = \omega(\Delta t_n + \Delta t'_n)$), weshalb Alerstam et al. [17] nur diese Flugzeiten betrachtet. Im Folgenden wird genutzt, dass die Photonen Repräsentanten huygensscher Elementarwellen sind und außerdem angenommen, dass sich diese über den betrachteten Zeitraum nicht ändern. Dies gilt, wenn für diesen der eingehende Laser eine zeitlich konstante Amplitude hat und sich auch die Elektronendichte nicht ändert. Nach einer kurzen Einschwingphase haben alle Elementarwellen den Punkt $\underline{\boldsymbol{p_d}}$ erreicht, wodurch sich hier eine Schwingung mit konstanter Amplitude bildet, welche ich an dieser Stelle herleiten werde. Wenn φ_{n1} die Phase zum Zeitpunkt t_1 in einer Detektorzelle (nach Gleichung (4.10)) ist, dann ist die Phase φ_{n2} zum Zeitpunkt t_2 gemäß Gleichung (3.4):

$$\varphi_{n2} = -\omega(t_2 - t_1) + \varphi_{n1} \tag{4.11}$$

Definiert man nun einen Referenzzeitpunkt t_R, kann man den Beitrag φ_{nR} eines Photons, das zum Zeitpunkt t_n das Volumen mit der Phase φ'_n verlässt und auf eine Detektorzelle im Abstand d_n trifft wie folgt berechnen:

$$\varphi_{nR} = -\omega(t_R - t_n) + \frac{\omega}{c} d_n + \varphi'_n \tag{4.12}$$

$$\varphi_{n0} = k d_n + \omega t_n + \varphi'_n \tag{4.13}$$

Dabei wurde für φ_{n0} die Referenzzeit $t_R := 0$ definiert. Die Formel entspricht, erwartungsgemäß, genau der Form, die sich aus der Gleichung (3.4) ergibt, wenn man sie um t_n verschiebt.

Mit diesem Ergebnis kann man Gleichung (4.7) auch wie folgt ausdrücken, wobei die Indizes in i_x, i_y verwendet werden, um zu zeigen, dass es sich um diskrete Punkte (Zellen) auf dem Detektor handelt.

$$I'(i_x, i_y, T) = p_c \left| \sum_{t=1}^{T} \sum_{n=1}^{N_t(i_x,i_y)} \mathrm{e}^{\mathrm{i}\varphi_{n0}} \right|^2 \tag{4.14}$$

Das bedeutet, dass zur Berechnung der Intensität auf dem Detektor in jeder Zelle alle Photonen entsprechend ihrer modifizierten Phase komplex addiert werden und erst zur Ausgabe das Betragsquadrat gebildet werden muss. Damit ist eine speicherplatzeffiziente Implementation möglich, da pro Zelle nur ein (komplexer) Wert gespeichert werden muss, statt mehrere, Werte für die einzelnen Auftreffzeiten, die je nach Auflösung u. U. sehr viele sein können.

An dieser Stelle ist noch anzumerken, welche Bedeutung die Position des Betragszeichen hat, weshalb die drei Möglichkeiten hier gegenübergestellt werden.

$$I'(i_x, i_y, T) = p_c |\sum_{t=1}^{T} \sum_{n=1}^{N_t(i_x,i_y)} \mathrm{e}^{\mathrm{i}\varphi_{n0}}|^2 \tag{4.14}$$

$$I(i_x, i_y, T) = p_c \sum_{t=1}^{T} |\sum_{n=1}^{N_t(i_x,i_y)} \mathrm{e}^{\mathrm{i}\varphi_{n}}|^2 \tag{4.15}$$

$$I''(i_x, i_y, T) = p_c \sum_{t=1}^{T} \sum_{n=1}^{N_t(i_x,i_y)} |\mathrm{e}^{\mathrm{i}\varphi_{n}}|^2 \tag{4.16}$$

Die erste ist Gleichung (4.14), deren Bedeutung gerade erläutert wurde, während die zweite auf Gleichung (4.7) basiert, in der nur Photonen miteinander wechselwirken, die zum gleichen Zeitpunkt auf den Detektor treffen. Als letztes werden mit Gleichung (4.16) einzelne Photonen gezählt, ein Prinzip, dass ebenfalls schon erwähnt wurde und über eine Policy in der Simulation nutzbar ist.

In der Implementation wird Gleichung (4.14) verwendet und beinhaltet die Annahme, dass die Kohärenzzeit des Lichts länger ist als die Simulationszeit. Diese Größe wurde bisher nicht betrachtet und soll hier nur am Rande erwähnt werden: Die Kohärenzzeit gibt hier die Zeitdauer an, in der das Licht interferenzfähig ist (vgl. [5, S. 174]) und man kann sie (für bestimmte Laser) „[...] heute mit großem technischen Aufwand [...] praktisch bis in den Sekundenbereich ausdehnen." [5, S. 175] Insbesondere da z. B. die üblicherweise genutzten (Röntgen-)Lichtblitze sehr kurz sind (für den kommenden *European XFEL* beispielsweise unter $100\,\mathrm{fs}$ (vgl. [4])), ist die Annahme von kohärenten Licht während der Simulationsdauer gerechtfertigt. Begrenzte Kohärenzzeiten kann man abbilden, in dem man jeweils eine Simulation für jedes Zeitintervall startet, das der Kohärenzzeit entspricht und in einer Nachverarbeitung die Intensitäten akkumuliert.

Beim Verlassen des Simulationsvolumens tritt ein Sprung in den zu betrachtenden Größenordnungen auf, da die Position der Photonen bisher relativ zu den (kleinen) Zellen betrachtet wurde, die Distanz zum Detektor aber sehr groß dem gegenüber ist. Deshalb muss hier die erreichbare Genauigkeit geprüft werden. Ein einfacher Vergleich der Größenordnungen des Abstands und der Wellenlänge zeigt bereits, dass die Nutzung einfacher Genauigkeit nicht ausreicht. In Unterabschnitt 5.3.3 wird dies genauer analysiert und eine Lösung vorgestellt, die auf der Fernfeldnäherung (s. Abschnitt 3.2) und der Betrachtung der Phasenunterschiede beruht, wobei die Phase nicht vollständig (und damit fehlerbehaftet) berechnet werden muss. An dieser Stelle soll lediglich die Änderung am Detektor betrachtet werden, die sich aus der ermittelten Phase ergibt.

Hier müssen komplexe Zahlen addiert werden, was am einfachsten in der kartesischen Darstellung $c = a + \mathrm{i}b = \cos(\varphi) + \mathrm{i}\sin(\varphi)$ möglich ist. Hat man den Real- und Imaginärteil bestimmt, addiert man diese auf die aktuellen Werte der Detektorzelle, in der das Photon gelandet ist. Da aber über die Photonen parallelisiert wurde, ist dazu eine atomare Addition notwendig, die insbesondere bei vielen Photonen zur teilweisen Serialisierung des Kernels führt. Außerdem entstehen durch die vielen Additionen Rundungsfehler, die mit steigendem Betrag des Ergebnisses zunehmen. In der Implementation wird für jede Grafikkarte ein eigenes Detektorfeld verwendet, was den Algorithmus resistenter gegen diese beiden Probleme macht, da nun nicht mehr alle Photonen auf einem Detektor landen. Dies verringert einerseits die Häufigkeit des parallelen Zugriffs auf eine Detektorzelle und die damit verbundene Serialisierung und andererseits wachsen die Werte in den Zellen nicht so stark an, was den numerischen Fehler

gering hält (vgl. [50, S. 80 ff.]). Zur Ausgabe werden die Teilergebnisse der Detektoren über eine MPI-weite Reduktion zu einem kombiniert, wobei hier in der Regel ähnliche große Werte addiert werden, was ebenfalls der Genauigkeit zugute kommt (s. Kapitel 5). Außerdem wird dadurch zusätzliche Kommunikation zwischen den Grafikkarten vermieden, da diese nur noch zur Ausgabe des Detektors und nicht in jedem Zeitschritt notwendig ist.

4.4 Zufallszahlengenerator

Bei der Analyse der Laufzeiten fiel auf, dass der Pseudozufallszahlengenerator sehr viel Zeit in Anspruch nahm, weshalb ich auf diesen zum Abschluss des Kapitels näher eingehen möchten. Durch die Natur der Monte-Carlo Simulationen wird eine große Menge an Zufallszahlen benötigt, um die Prozesse damit modellieren zu können. Hier sind das je Partikel und Zeitschritt aktuell zwischen einer und drei Zufallszahlen für die Streuwahrscheinlichkeit und beide Winkel. Zur Generierung kommen dafür meist Pseudozufallszahlengeneratoren (PRNGs) zum Einsatz, die aus einem Zustand über eine Berechnungsvorschrift einen Folgezustand und eine Zahl ermitteln, deren Verteilung über viele auf solche Weise erzeugte Zahlen „zufällig genug“ ist, d. h. bestimmte statistische Tests auf Zufälligkeit besteht. Unter anderem ist es wünschenswert, dass die erzeugte Folge einen möglichst großen Zyklus hat, nachdem sie sich wiederholt[19]. Die Theorie hinter solchen Methoden ist nicht trivial, da die schlechte (oder zufällige) Wahl von Algorithmen oder Parametern zu sehr kurzen Zyklen oder starken Abhängigkeiten der erzeugten Zahlen führt (vgl. [51, Chapter Three]). Zusätzlich werden für diese Simulation viele parallel nutzbare PRNGs benötigt, da jeder Thread mehrere Zufallszahlen braucht und die Nutzung eines einzigen, gemeinsam genutzten PRNGs und die damit notwendige Synchronisierung aufgrund der hohen Anzahl von Threads nicht sinnvoll machbar ist. Wichtig ist hierbei, dass die einzelnen Pseudozufallszahlengeneratoren unabhängig voneinander sind, da Abhängigkeiten zu ungewollter Musterbildung führen können.

LibPMacc verwendet deshalb den in CUDA enthaltenen „XORWOW“-Generator [52] aufgrund seiner hohen Geschwindigkeit, geringem Speicherverbrauch und der Verwendbarkeit für hoch parallele Anwendungen. Die Periode von diesem ist größer als 2^{190} [52], und bei der Initialisierung kann zusätzlich zum Seed (Startwert) eine Sequenznummer angegeben werden, wobei die große Periode in einzelne Teilsequenzen unterteilt wird. Diese Initialisierung ist vor der Erzeugung von Zufallszahlen notwendig, und setzt den internen Zustand auf einen Startwert. Aufgrund der Empfehlung von NVIDIA („For the highest quality parallel pseudorandom number generation, [...] each thread of computation should be assigned a unique sequence number.“ [52]) und der Zuordnung von Threads auf Zellen, wurde der Wrapper von libPMacc so verwendet, dass der Seed aus dem jeweiligen Kernel, dem Zeitschritt und dem MPI-Rank abgeleitet und der (lokale) Zellindex als Sequenznummer verwendet wurde. Diese Implementierung ist zwar speicherplatzeffizient, aber durch die Initialisierung in jedem Zeitschritt sehr langsam, wie Abb. 4.6 zeigt. Mit steigender Simulationsgröße dauerte die Initialisierung mehrere Sekunden pro Zeitschritt, wohingegen die eigentliche Erzeugung selbst für vier Millionen Zellen (bzw. Zufallszahlen) nur rund drei Millisekunden in Anspruch nahm.

Deshalb wurde eine neue Abstraktion entworfen, die ebenso einfach zu verwenden ist, wie die ursprüngli-

[19] Zyklen können nicht gänzlich verhindert werden, da der Zustandsraum endlich ist und somit irgendwann ein Zustand wieder auftritt wodurch sich die Folge wiederholt.

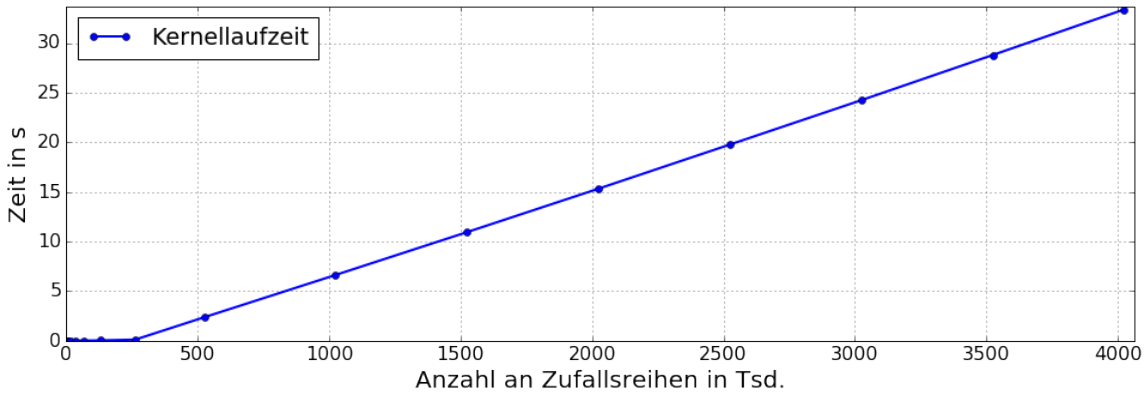


Abbildung 4.6: Zeit zum parallelen Initialisieren der XORWOW PRNGs

che, in libPMacc implementierte Version, aber die Zustände nur einmal erzeugt und im globalen Speicher vorhält. Für jede Zelle der Simulation existiert somit schon ein Zustand, der jeweils von einem Thread verwendet wird. Zum direkten Vergleich für das in dieser Arbeit erstellte Programm, wurden beide Versionen in der ansonsten gleichen Simulation verwendet. In dieser wurde ein Doppelspalt als Dichte und 524288 Zellen auf einer GPU verwendet. Trotz dieser noch relativ kleinen Simulation ist die neue Implementation mit 96 Sekunden Gesamtlaufzeit mehr als 53 mal schneller, als die alte Implementation (5144 Sekunden). Bei einer vier mal größeren Simulation (rund zwei Mio. Zellen) war sie sogar rund 310 mal schneller: $137\,\mathrm{s}$ statt $42\,458\,\mathrm{s}$. Das liegt neben dem Zusatzaufwand für die Berechnung auch daran, dass die Kernel durch die Initialisierung der PRNGs i. d. R. mehr Register verbrauchen, was die effektive Auslastung des SMX (s. Abschnitt 4.1) verringern kann, wodurch Latenzen schlechter ausgeglichen werden können.

Ein Nachteil der neuen Implementation ist der erhöhte Speicherverbrauch, da jeder Zustand 44 Byte[20] verbraucht. Da dieser jedoch meist nicht, oder nicht vollständig, im Register gehalten werden kann und deshalb in den Hauptspeicher ausgelagert wird, verbraucht auch die alte Implementation Speicher, wobei dies dort üblicherweise weniger ist. Außerdem konnte durch den Verzicht auf die (hier nicht notwendige) Generierung von normalverteilten Zufallszahlen die Größe der Zustände auf jeweils 24 Byte reduziert werden, indem dafür notwendige Teile der Zustände entfernt werden. Da dazu allerdings auf nicht dokumentierte Implementationsdetails zugegriffen werden musste, kann es sein, dass dies nicht mit zukünftigen CUDA Versionen kompatibel ist. Es ist jedoch anzunehmen, dass sich diese Struktur nicht ändern wird. Außerdem wird die Kompatibilität zur Compilezeit geprüft und eine Änderung an dieser Stelle kann somit erkannt werden, bevor sie sich auf das Ergebnis auswirkt. Der Speicherverbrauch bei vier Mio. Zellen beträgt lediglich 96 MB, was für den zu erwartenden Performancevorteil sehr wenig ist, weshalb ausschließlich die neue Implementation verwendet wird.

[20] Durch die Ausrichtung der Struktur kann es eventuell mehr sein.

4.5 Genutzte Datenformate

Die Simulation erzeugt Ausgabedaten in verschiedenen Formaten und kann auch verschiedenartige Eingaben annehmen. Die Ausgabe erfolgt über Plugins, wobei das zugehörige System in libPMacc implementiert ist und auch von PIConGPU verwendet wird. Jedes der Plugins wird in konfigurierbaren, periodischen Intervallen aufgerufen und kann auf die benötigen Daten zugreifen und diese ausgeben. Auch kann jedes Plugin Kommandozeilenparameter definieren, die der Nutzer beim Start angeben kann. So können Partikeleigenschaften zu Debug-Zwecken als Text auf der Konsole ausgegeben werden. Zur einfachen Verwendung lässt sich das Dichtefeld ebenenweise als monochromes PNG oder TIFF Bild und die Intensität ebenfalls als TIFF Bild ausgeben, wobei letzteres gewählt wurde, da es Gleitkommawerte akzeptiert und dieses somit direkt weiterverarbeitet werden kann. Ebenfalls zu Testzwecken stehen verschiedene „Generatoren“ zur Verfügung, die bestimmte Dichteverteilungen (z. B. Doppelspalt-Muster) erzeugen können. Diese können erweitert werden womit man sehr einfach Verteilungen durch eine Funktion des Ortes ausdrücken kann. Auch ist es möglich, eine Gruppe von TIFF Bildern als Schnittebenen einer dreidimensionalen Dichte aufzufassen und einzulesen. Als wichtigstes Datenformat dient HDF5 [53], welches über libSplash [54] (eine objektorientierte Bibliothek, welche die Verwendung von HDF5 in Simulationen vereinfacht) gelesen und geschrieben wird. Dabei kommt das openPMD (Meta-)Datenformat zum Einsatz. „[O]penPMD provides naming and attribute conventions that allow to exchange particle and mesh based data from scientific simulations and experiments.“ [55] Standardmäßig wird die Intensität auf dem Detektor als HDF5 Datei und als TIFF Bild ausgegeben, sofern die entsprechenden Bibliotheken beim Konfigurieren mit CMake gefunden werden. Es wird somit nicht vorausgesetzt, beide zur Verfügung zu haben und die Ausgabe in jedes der Formate lässt sich mit einem passenden Kommandozeilenparameter deaktivieren. Die Implementation unterstützt auch das Erstellen und Laden von Sicherungspunkten („Checkpoints“) in der gleichen Art und Weise wie sie auch von PIConGPU verwendet werden. Damit ist es möglich, eine abgebrochene oder beendete Simulation an einem dieser Punkte fortzusetzen, oder (bei Einhaltung des Formats) diese als Eingaben zu verwenden. Im Hinblick auf die Verwendung zeitveränderlicher Dichten wurde außerdem eine Funktionalität entwickelt, die einen Satz von Dichteverteilungen im (openPMD konformen) HDF5 Format annimmt, wobei jeder Datensatz einem Zeitpunkt entspricht. In der Simulation wird die Dichte zum aktuellen Zeitschritt aus den zwei zeitlich nächsten Datensätzen interpoliert, was parallel zur Ausführung des Kernels eines vorhergehenden Zeitschritts möglich ist.

Die Simulationsparameter sind aufgeteilt in Laufzeitparameter, die über die Kommandozeile beim Start übergeben werden und Compilezeit Parameter, welche in speziellen C++ Header-Dateien stehen. Letztere ermöglichen es, einige Berechnungen (wie Umrechnungen in andere Einheiten) bereits zur Compilezeit durchzuführen und erlauben auch weitere Optimierungen. Zu dieser Kategorie gehören u. a. die Zellgrößen von Simulationsvolumen und Detektor, Abstände, Zeitschrittlänge und physikalische Konstanten wie die Lichtgeschwindigkeit, aber auch die Eigenschaften der Partikel-Spezies, wozu sowohl Werte als auch Policies gehören (vgl. Abschnitt 4.2). Dieses Vorgehen erhöht zwar den Aufwand beim Ändern der Parameter, da dies eine Neuübersetzung erfordert, ermöglicht aber das Vermeiden von Overhead zur Laufzeit durch dynamische Typen oder Abfragen. Insbesondere bei hoch parallelen Anwendungen ist es sinnvoller, eine längere Zeit zum Übersetzen aufzuwenden, was auf einer Maschine abläuft, als längere Laufzeiten auf potentiell tausenden, parallel arbeitenden Rechenknoten in Kauf zu nehmen.

Die Kommandozeilenparameter dienen schließlich zur Angabe von Gesamtgrößen des Simulationsvolumens und Detektors (in Zellen), Ausgabeperioden, Dateipfade für Ein- und Ausgabedateien sowie weiteren Optionen für die Plugins und dem Aktivieren von Sicherungspunkten bzw. deren Nutzung. Auch wird hier die Anzahl und Aufteilung der GPUs festgelegt. So lässt sich zum Beispiel die Anwendung mit einer Grundkonfiguration erzeugen, in der die Mesh-Geometrien und Größen festgelegt sind, und anschließend für verschiedene Dichteverteilungen nutzen, die auch unterschiedliche Ausdehnungen haben können. Generell ist die Aufteilung in Compile- und Laufzeit-Parameter mit dem Abwägen von Flexibilität zu möglichem Performancevorteil verbunden. Die hier getroffene Aufteilung orientiert sich stark an PIConGPU um auch damit eine hohe Kompatibilität und leichte Integrationsmöglichkeit zu bieten.

5 Numerische Genauigkeit

In diesem Kapitel soll die numerische Genauigkeit der Berechnungen in der Simulation geprüft werden, mit dem Hintergrund, so weit wie möglich einfache Genauigkeit zu verwenden, da dies für die Laufzeit der Simulation mehrere Vorteile bietet. Einerseits lassen sich durch den geringeren Speicherbedarf von vier statt acht Byte mehr Photonen gleichzeitig speichern (und verarbeiten), andererseits ist die Berechnung auch schneller. Die Gründe dafür sind folgende (s. Abschnitt 4.1): Die Anzahl von Recheneinheiten für einfache und doppelte Genauigkeit unterscheidet sich um einen Faktor von drei[21] (Kepler Architektur) bis 32 (Maxwell Architektur). Der Registerverbrauch ist niedriger, da für Werte in einfacher Genauigkeit nur ein statt zwei Register gebraucht werden, was sich wiederum positiv auf die Auslastung der SMX und der Fähigkeit, Latenzen auszugleichen, auswirkt. Und schließlich wird auch weniger Bandbreite benötigt, um Daten von und zum Grafikspeicher zu transferieren.

5.1 Bedeutung

Gleitkommazahlen In Computern werden für Berechnungen nicht-ganzzahliger Werte Gleitkommazahlen nach IEEE-754 verwendet, welche u. a. Festlegungen für binäre und dezimale Gleitkommazahlen enthält. Die folgende Betrachtung wird sich auf die binären beschränken.
Eine Gleitkommazahl F wird dabei mittels des Exponenten e und der ganzzahligen Mantisse m als $F = S \cdot m \cdot 2^{e-t}$ dargestellt, wobei $S = \pm 1$ das Vorzeichen und t die Genauigkeit (in Bit) angibt. In der binären Repräsentation wird der Exponent immer so klein wie möglich gewählt, wodurch die (binäre) Mantisse normalisierter Zahlen stets mit einer 1 beginnt. Diese wird gemäß der „hidden bit“-Konvention nicht gespeichert, wodurch man ein zusätzliches Bit an Genauigkeit gewinnt. Vom Exponent wird die sogenannte Charakteristik gespeichert, welche man erhält, indem ein fester Biaswert auf den Exponenten addiert wird. Dies dient u. a. der einfachen Sortierbarkeit, da somit nur das Bitmuster betrachtet werden muss. Die minimalen und maximalen Werte der Charakteristik sind reserviert zur Markierung von Sonderfällen, wie der Null, $\pm\infty$ und denormalisierten Zahlen. Letztere füllen die Lücke zwischen null und der kleinsten, normalisierten Zahl, haben aber eine geringere Genauigkeit. Der Standard umfasst unter anderem die typischerweise verwendeten Formate für einfache und doppelte Genauigkeit mit 32 bzw. 64 Bit Größe, die in C/C++ üblicherweise durch `float` bzw. `double` repräsentiert werden. Dabei ist festgelegt, wie viele Bits jeweils für die Charakteristik und die Mantisse verwendet werden, wodurch sich der maximal darstellbare Zahlenbereich (abhängig von den möglichen Exponenten) und die maximale Genauigkeit (abhängig von der Mantissenlänge) ergibt. Weitere, im Folgenden wichtige Festlegungen in IEEE-754 sind Vorschriften zur Rundung von Werten und verschiedene Operationen wie Addition, Subtraktion, Multiplikation, Division, das Wurzelziehen und die Berechnung des Restes bei der Division mit einem ganzzahligem Ergebnis (*Remainder-Operation*). Letzteres ist definiert als $r(x,y) = x - \mathrm{rnd}(x/y)y$, wobei $\mathrm{rnd}()$ die Rundung zur nächsten Ganzzahl bedeutet. Im Folgenden soll das der Einfachheit halber

[21] Die Pascal Architektur verbessert das Verhältnis von Berechnungen in einfacher zu doppelter Genauigkeit auf 2:1 (vgl. [56]).

als Modulo-Operation mit dem Formelzeichen $\%$ mit $x \,\%\, y \mathrel{\hat{=}} r(x, y)$ bezeichnet werden. Für die genannten Operationen gilt, dass das Ergebnis dem Wert entsprechen muss, der sich aus der exakten (unendlich genauen) Berechnung durch anschließende Rundung ergibt, was als „correctly rounded“ oder „korrekt gerundet“ bezeichnet wird. Im Standard sind außerdem einige andere Operationen, wie den trigonometrischen Funktionen (Sinus, Kosinus, Tangens und deren Umkehrfunktionen), empfohlen, die korrekt gerundet sein sollen. Der Standard-Rundungsmodus ist „Round-To-Nearest-Even“, das auf die nächstmögliche darstellbare Zahl rundet, wobei Werte exakt zwischen zwei Zahlen auf die gerade Zahl gerundet werden. Es sind aber auch drei weitere Rundungsmodi gefordert: Aufrunden (gegen $+\infty$), Abrunden (gegen $-\infty$) und Runden zur betragsmäßig kleineren Zahl (gegen 0).

Rundungsfehler Durch die Rundung ergibt sich i. d. R. ein Fehler. Dabei unterscheidet man zwischen dem absoluten Fehler der Approximation $\hat{x}$ von einer Zahl x gegeben als $E_{abs}(\hat{x}) = |x - \hat{x}|$ und dem relativen Fehler $E_{rel}(\hat{x}) = |\frac{x-\hat{x}}{x}|$ bzw. der alternativen Repräsentation $\hat{x} = x(1 + \epsilon)$ mit $|\epsilon| = E_{rel}(\hat{x})$ ([50, S. 3]). Im Allgemeinen interessiert man sich für den maximalen Fehler (absolut oder relativ) um eine minimale Genauigkeit eines Ergebnisses angeben zu können. Das auch verwendete Maß, das die Anzahl korrekter signifikanter Stellen angibt (signifikante Stellen sind alle Ziffern einer Zahl ab der ersten, die nicht null ist), soll im Weiteren nicht verwendet werden, da eine einheitliche Definition nicht gegeben ist, es von der verwendeten Basis abhängt und es weniger genau als der relative Fehler ist (vgl. dazu die Diskussion in [50, S. 3 f.]).

Da auch die Zwischenergebnisse von einzelnen Operationen dargestellt werden müssen, wird nach jeder Operation gerundet. Dadurch gilt zwar die Kommutativität ($a + b = b + a$) allerdings nicht unbedingt die Assoziativität ($(a + b) + c = a + (b + c)$) oder Distributivität ($(a + b) \cdot c = a \cdot c + b \cdot c$). Zwei wichtige Effekte in diesem Zusammenhang sind die Absorption und Auslöschung: Werden Zahlen, die sich in ihrer Größenordnung stark unterscheiden, addiert, kann es zur Absorption der kleineren kommen. Es gilt so z. B. $a + \epsilon = a$ und $(a + \epsilon) - a = 0$ für ϵ ausreichend klein gegenüber a. Umgekehrt tritt die Auslöschung (englisch „Cancellation“) bei der Subtraktion ähnlich großer Werte auf. „There are two kinds of cancellation: catastrophic and benign. Catastrophic cancellation occurs when the operands are subject to rounding errors.“ [57] Katastrophal bezieht sich hierbei auf den „catastrophic loss of precision“ wie er auch in Harris [58] beschrieben ist, und dadurch entsteht, dass die höchstwertigen Bits der Mantisse, die gleichzeitig den geringsten Fehler haben, durch die Subtraktion ausgelöscht werden und lediglich die niederwertigen, teilweise fehlerbehafteten Bits übrig bleiben. Im Extremfall werden zwei Werte subtrahiert, die auf den gleichen Wert gerundet wurden und damit die gleiche Mantisse haben, wodurch das Ergebnis ausschließlich aus den Rundungsfehlern besteht. Ein Beispiel hierfür ist die numerische Ableitung von Funktionen: Die Formel zur numerischen Berechnung der Ableitung an einer Stelle x ist $f'(x) = \frac{f(x+h)-f(x)}{h}$, wobei h sehr klein (größer 0) gewählt wird. Dadurch ergibt sich eine Differenz im Zähler von zwei (gerundeten) Werten, die fast gleich sind, was dazu führt, dass nur sehr wenige (im Extremfall gar keine) signifikante Stellen übrig bleiben. Da diese am stärksten fehlerbehaftet sind ist demzufolge das Ergebnis stark fehlerbehaftet. Ein besseres Ergebnis lässt sich mit einem etwas größerem h erreichen, wodurch mehr signifikante Stellen übrig bleiben, auch wenn dadurch der Fehler für stark nichtlineare Funktionen im betrachteten Bereich größer wird. Wie in Goldberg [57] ausgeführt wird, tritt auch bei $x^2 - y^2$ katastrophale Auslöschung auf, wenn x^2 und y^2 nah beieinander liegen. Das lässt sich jedoch in eine harmlose (englisch „benign“) Auslöschung umwandeln, in dem die Berechnung

in die (mathematisch) äquivalente Darstellung $(x+y)(x-y)$ umgeformt wird. Unter der Annahme, dass die Eingabegrößen exakt sind, erfolgt hier keine Subtraktion gerundeter Werte und das Ergebnis ist exakt bis auf einen kleinen relativen Fehler. Dies zeigt, wie durch Umformung von Termen in mathematisch äquivalente Darstellungen bestimmte Fehler vermieden und die erreichten Genauigkeiten erhöht werden können. An dieser Stelle sei bemerkt, dass nicht jede (katastrophale) Auslöschung um jeden Preis vermieden werden muss, da sie nicht immer das Endergebnis maßgeblich beeinflusst. So ist die mögliche Auslöschung in der Gleichung $x + (y - z)$ kein Problem, wenn x ausreichend groß gegenüber y und z ist (vgl. [50, S. 9 f.]). Auch sei bemerkt, dass die Subtraktion zweier Gleitkommawerte $x - y$ unter bestimmten Bedingungen exakt ist. So gilt dies z.,B. für $y/2 \leq x \leq 2y$ nach dem Theorem von Sterbenz [59], unter der Bedingung, dass das Ergebnis im darstellbaren Bereich liegt (kein „Unterlauf“).

Stabilität und Kondition Bei der Fehleranalyse eines Algorithmus betrachtet man die Stabilität mittels Vorwärts- und Rückwärtsanalyse, die hier erklärt werden. Dazu sei eine Funktion $y = f(x)$ gegeben, die durch einen Algorithmus $\hat{y} = \hat{f}(x)$ implementiert ist, der analysiert werden soll.

Typischerweise betrachtet man meist zuerst den Vorwärtsfehler (engl. „forward error“), was dem Fehler zwischen $f(x)$ und $\hat{f}(x)$ (absolut oder relativ) entspricht. Kann man für diesen eine Grenze festlegen, die klein genug über den kompletten Definitionsbereich ist, nennt man den Algorithmus „vorwärtsstabil“ (engl. „forward stable“).

Bei der Rückwärtsanalyse bestimmt man das betragsmäßig[22] kleinste ∂x mit $\hat{f}(x) = f(x + \partial x)$ (möglicherweise auch relativ zu $|x|$), was als Rückwärtsfehler (engl. „backward error“) bezeichnet wird (vgl. [50, S. 6]). Es geht dabei also darum, den Bereich um den tatsächlichen Eingabewert zu bestimmen, für den das Ergebnis gilt. Kann man ein solches ∂x angeben, das für alle x gilt und klein genug ist, nennt man den Algorithmus „rückwärtsstabil“ (engl. „backward stable“). Hintergrund davon ist, dass die Eingabegrößen meist selbst gerundete Werte sind. Wenn mit der Rückwärtsanalyse nachgewiesen werden kann, dass der Fehler kleiner ist, als der Rundungsfehler der Eingabewerte, dann ist die Berechnung genau genug. Daher auch das „klein genug“ in der vorhergehenden Definition. Die Rückwärtsstabilität ist hierbei das stärkere Kriterium, da sich nachweisen lässt, dass sie die Vorwärtsstabilität impliziert (vgl. [50, S. 9]). Häufig wird eine etwas schwächere Variante verwendet, um einen Algorithmus als numerisch stabil zu klassifizieren: Der gemischte Vorwärts-Rückwärts Fehler (engl. „mixed forward-backward error“), den Higham [50] definiert als:

$$\hat{y} + \partial y = f(x + \partial x), |\partial y| \leq \epsilon |y|, |\partial x| \leq \eta |x|$$

mit ausreichend kleinen ϵ und η bzw. $|\partial y|$ und $|\partial x|$. Das heißt die algorithmische Lösung liegt nur unwesentlich neben der realen Lösung für eine nur unwesentlich veränderte Eingabe. Dies wird z. B. für die Algorithmen zur Berechnung der transzendenten Funktionen verwendet, die der strikten Rückwärtsstabilität nicht genügen. Es ist leicht ersichtlich, dass ein rückwärtsstabiler Algorithmus auch numerisch stabil im Sinne des gemischten Vorwärts-Rückwärts Fehlers ist.

Im Zusammenhang mit der Stabilität muss man auch die Kondition eines Problems betrachten. Diese gibt an, wie stark sich die Ausgabe für kleine Änderungen in der Eingabe ändert. Über eine Taylorent-

[22] Für nicht-skalare Werte, wie Vektoren oder Matrizen, wird statt dem Betrag meist eine Norm verwendet. Die Vorgehensweise ist aber die gleiche.

wicklung an der Stelle x kommt man (in obiger Notation) und mit $\tilde{y} = f(x + \partial x)$ über die Gleichung

$$\frac{\tilde{y} - y}{y} = \left(\frac{xf'(x)}{f(x)} \right) \frac{\partial x}{x} + O((\partial x)^2)$$

zur relativen Konditionszahl

$$c(x) = \left| \frac{xf'(x)}{f(x)} \right|$$

die den Zusammenhang zwischen einer kleinen, relativen Änderung in x und der relativen Änderung in y ausdrückt. Entsprechend gilt bei konsistenter Definition der Größen: Vorwärtsfehler $\lesssim$ Konditionszahl $\cdot$ Rückwärtsfehler (vgl. [50, S. 8 f.]). Eine wichtige Erkenntnis hieraus ist, dass Rundungsfehler insbesondere an Stellen, wo die Ableitung sehr groß ist (z. B. an Singularitäten), zu großen Fehlern im Ergebnis führen können.

5.2 Modell zur Analyse von Gleitkommaoperationen

Bei der Betrachtung der Genauigkeit von Gleitkommaoperationen benötigt man eine Grenze, durch die der maximale Fehler limitiert ist. Im Folgenden soll angenommen werden, dass Eingabewerte und (Zwischen-)Ergebnisse im Bereich der darstellbaren Gleitkommazahlen liegen, also weder Unterlauf noch Überlauf auftreten, da diese Sonderfälle gegebenenfalls gesondert betrachtet werden müssen. Häufig verwendet wird die „unit in the last place“ (kurz „ulp“) (vgl. [57]), die dem Wert entspricht welche durch die letzte Stelle der Mantisse dargestellt werden kann. Dieser ist gegeben durch $ulp(x) = 2^{e-t}$, wobei e der Exponent der normalisierten Repräsentation von x und t die Mantissenlänge in Bit (inklusive des „hidden bit“) ist. Damit ist die ulp zwar ein Maß für den absoluten Fehler, die Angabe „ein Ergebnis ist auf n ulps genau“ jedoch ein relativer Wert, da er die Kenntnis des Exponenten des Ergebnisses erfordert. Für korrekt gerundete Ergebnisse beträgt der Fehler immer weniger als eine ulp, da immer auf die nächste darstellbare Zahl gerundet wird. Bei Verwendung des Rundungsmodus Round-To-Nearest, was im Folgenden implizit angenommen werden soll, ist der maximale Fehler auf $0{,}5$ ulps beschränkt (vgl. [57]). Betrachtet man den relativen Fehler, nutzt man meist die Maschinengenauigkeit. In der Literatur werden in dem Zusammenhang häufig die Begriffe „Rundungseinheit“ (engl. „unit roundoff“) und Maschinenepsilon (engl. „machine epsilon“ oder auch „macheps“) mit teilweise wechselnden Bedeutungen verwendet). In [60] wird beispielsweise das Maschinenepsilon gleich der Rundungseinheit angenommen.

Hier sei basierend auf Higham [50, S. 37 f.] das Maschinenepsilon als ϵ_M und die Rundungseinheit als $\mathbf{u}$ definiert, wobei der Begriff „Maschinengenauigkeit“ je nach Kontext, i. d. R. jedoch für die Rundungseinheit genutzt wird. Das Maschinenepsilon „is the distance ϵ_M from 1.0 to the next larger floating point number“ [50, S. 37] und entspricht $\epsilon_M = 2^{1-t}$. Die Rundungseinheit $\mathbf{u}$ ist die wichtigere Größe, da sie ein direktes Maß für den relativen Fehler ist. Sie ist abhängig vom Rundungsmodus, ergibt sich für Round-To-Nearest zu $\mathbf{u} = \frac{1}{2}2^{1-t} = 2^{-t}$ und ist damit gleich der Hälfte des Maschinenepsilon.

Damit lässt sich jetzt der maximale Fehler δ bei der Rundung eines Gleitkommawertes bzw. des Ergeb-

nisses einer Gleitkommaoperation wie folgt beschreiben:

$$fl(x) = x(1+\delta), \qquad |\delta| \leq \mathbf{u} \tag{5.1}$$

$$fl(x) = \frac{x}{1+\delta}, \qquad |\delta| \leq \mathbf{u} \tag{5.2}$$

Hierbei wurde die Notation von Higham [50] verwendet, in der $fl(x)$ die korrekt gerundete Gleitkommazahl zu x bezeichnet. Die erste Gleichung ergibt sich unmittelbar aus der Definition des relativen Fehlers (s. Abschnitt 5.1 auf Seite 50), die zweite Darstellung ist oftmals hilfreich und folgt aus dem absoluten Fehler relativ zu $fl(x)$. Für einen Beweis der Gültigkeit sei auf Higham [50] verwiesen. Die gleichen Notationen sollen für arithmetische Operationen $(+, -, \cdot, /)$ auf zwei (exakt repräsentierte) Operanden x und y verwendet werden (vgl. [50, S. 40]):

$$fl(x \operatorname{op} y) = (x \operatorname{op} y)(1+\delta), \qquad |\delta| \leq \mathbf{u} \tag{5.3}$$

$$fl(x \operatorname{op} y) = \frac{x \operatorname{op} y}{1+\delta}, \qquad |\delta| \leq \mathbf{u} \tag{5.4}$$

Dies gilt auch für die Operation des Wurzelziehens.

An dieser Stelle soll kurz auf den Zusammenhang von „ulp“ und ϵ_M bzw. $\mathbf{u}$ hingewiesen werden, da z. B. der *CUDA C Programming Guide* die Genauigkeit einiger Funktionen als „[...] the absolute value of the difference in ulps between a correctly rounded [...] result and the result returned by the CUDA library function“ [44] angibt. Betrachtet werden soll eine Zahl x deren normalisierte, binäre Repräsentation den Exponent e habe und damit $ulp(x) = 2^{e-t} = \epsilon_M 2^{e-1}$ (im Folgenden: ulp_x) gilt. Durch die Normalisierung, gilt für x: $2^{e-1} \leq |x| < 2^e$. Für einen maximalen absoluten Fehler von N ulps bei der Rundung von x gilt dann $|\frac{fl(x)-x}{x}| \leq N\frac{ulp_x}{|x|}$ und da wegen $|x| \geq 2^{e-1}$ die Ungleichung $N\frac{\epsilon_M 2^{e-1}}{|x|} \leq N\epsilon_M$ erfüllt ist, gilt $|\frac{fl(x)-x}{x}| \leq N\epsilon_M$. Für den Rundungsmodus Round-To-Nearest gilt somit auch $|\frac{fl(x)-x}{x}| \leq 2N\mathbf{u}$ und bei einem Fehler von maximal $0{,}5$ ulps folgt damit unmittelbar Gleichung (5.1). Bei einer absoluten Differenz von M ulps zum korrekt gerundeten Wert ergibt sich demnach ein relativer Fehler von maximal $(2M+1)\mathbf{u}$.

Bei der Analyse von Berechnungen kommt man mit Gleichung (5.3) und (5.4) häufig auf Produkte und Brüche von mehreren $(1+\delta_i)$. Zu deren Vereinfachung können folgende Regeln aus Higham [50, S. 63, 67 ff.] verwendet werden:

$$\prod_{i=i}^{n}(1+\delta_i)^{p_i} = 1+\theta_n, \qquad |\theta_n| \leq \frac{n\mathbf{u}}{1-n\mathbf{u}} =: \gamma_n \tag{5.5}$$

$$(1+\theta_k)(1+\theta_j) = 1+\theta_{k+j} \tag{5.6}$$

$$\frac{1+\theta_k}{1+\theta_j} = \begin{cases} 1+\theta_{k+j}, & j \leq k \\ 1+\theta_{k+2j}, & j > k \end{cases} \tag{5.7}$$

Hierbei gilt gemäß obiger Definition $|\delta_i| \leq \mathbf{u}$ und $p_i \pm 1$. Häufig verwendet man auch eine Variante von Gleichung (5.5), in der man die Terme höherer Ordnung ignoriert (s.dazu Einarsson [61, S. 65]):

$$\prod_{i=i}^{n}(1+\delta_i) = 1+\theta'_n, \qquad |\theta'_n| \leq n\mathbf{u} \tag{5.8}$$

Dies hilft häufig die komplizierten Fehlergrenzen zu vereinfachen und ist trotzdem noch ausreichend exakt, da der tatsächliche Fehler meist deutlich unter der berechneten Grenze liegt, weil das Modell z. B. Fehler, die sich gegenseitig aufheben, nicht berücksichtigen kann. Higham [50] schreibt dazu mit Verweis auf [62]: „[...] if the bound is $f(n)\mathbf{u}$, the rule of thumb says that the error is typically of order $\sqrt{f(n)}\mathbf{u}$ [...]“ [50, S. 48].

Für die Addition von Zahlen mit gleichem Vorzeichen ist $(1+\theta_n)$ distributiv, d. h. $z_1(1+\theta_n)+z_2(1+\theta_n) = (z_1 + z_2)(1 + \theta_n)$, was aber darüber hinaus nicht generell gilt. Der Beweis dazu ist analog zu dem in [63, S. 24 f.], indem dies für $\hat{\prod}_j$ bewiesen wird, was als $\hat{\prod}_j = (1 + \theta_j)$ definiert ist. Daraus folgt die Distributivität bezüglich der Addition von Zahlen gleichen Vorzeichens auch für $(1 + \delta)$. Hierbei steht δ ohne Index, analog zu θ_n, für ein beliebiges δ_i, das i. A. verschieden zu anderen δ bzw. δ_j ist und für das deshalb die Distributivität nicht generell gilt.

Eine wichtige Einschränkung des Modells ist, „[that it] does not require that $\delta = 0$ when x op $y \in F$ – a condition which obviously does hold for the rounded answer [...]“ [50, S. 40], wobei er mit F die Menge der exakt darstellbaren Gleitkommazahlen meint. Diese wichtige Ausnahme kann bei der Analyse helfen, den Fehler nicht zu sehr zu überschätzen, wenn die entsprechenden Werte bekannt sind.

Mithilfe dieses Modells und unter zusätzlicher Berücksichtigung möglicher Auslöschung und Absorption sollen im Folgenden die mathematischen Operationen in der Simulation genauer betrachtet und hinsichtlich ihrer Genauigkeit untersucht werden. Das Ziel ist es, so weit wie möglich auf Berechnungen in doppelter Genauigkeit zu verzichten um einerseits Speicher und andererseits Rechenzeit zu sparen. Am Ende dieses Kapitels sollen Vorhersagen über die möglichen Fehler sowie Grenzen für die Eingabegrößen gefunden werden, für die eine gegebene Genauigkeit noch erreicht wird. Im Folgenden wird von einer ebenen Welle mit zeit- und ortsunabhängiger Amplitude sowie einer ebenfalls zeitinvarianten Elektronendichte ausgegangen. Die Simulation wird durch folgende Größen parametrisiert:

λ ...	Wellenlänge [m]
s ...	Simulationsgröße [m]
L ...	Detektorabstand [m]
θ_{max} ...	Maximaler Ablenkungswinkel [rad]
D ...	(Halbe) Detektorgröße [m]
t ...	Simulationszeit [s]
Δt ...	Länge eines Zeitschritts [s]
Δs ...	Zellgröße in der Simulation [m]
ΔD ...	Größe der Detektorzellen [m]
N_s, N_D ...	Anzahl der Zellen (je Richtung) in Simulation bzw. Detektor

Dabei sei hier von quadratischen bzw. würfelförmigen Geometrien ausgegangen, sodass nur jeweils eine Größe für die Zellen $(\Delta s, \Delta D)$ bzw. Gesamtmaße (s, D) nötig ist. Ebenfalls sei angenommen, dass die Größe des Detektors und der Simulation ganzzahlige Vielfache ihrer jeweiligen Zellgrößen sind, sodass $s = \Delta s \cdot N_s$ und $D = \Delta D \cdot N_D$ gilt.

Da diese Größen teilweise voneinander abhängen, genügt es nur einige davon als Eingaben zu betrachten. Der maximale Ablenkungswinkel in alle Richtungen soll gleich groß sein, womit für die halbe Detektorgröße $\tan(\theta_{max}) \cdot L = D$ gilt. Andere Detektorgrößen sind in dieser Betrachtung nicht sinnvoll, da sich für größere Werte bei konstanter Zellenanzahl die Auflösung verschlechtert und für kleinere Werte

Informationen verloren gehen. Aus der Strukturgröße und der Wellenlänge ergibt sich der maximale Ablenkungswinkel, der umgekehrt proportional zur Strukturgröße ist. Da (maximale) Verstärkung auftritt, wenn der Weglängenunterschied ein ganzzahliges Vielfaches der Wellenlänge ist, kann der maximal benötigte Winkel zur Beobachtung von n Maxima aus der minimalen Strukturgröße bestimmt werden. Ist dies die Zellgröße Δs ergibt sich, unter Verwendung der Fernfeldnäherung $\sin(\theta_{max}) \cdot \Delta s = n\lambda$ als der maximal benötigte Winkel. Die Zeit soll hier aufgrund der Zeitinvarianz von Laser und Elektronendichte lediglich zur Erhöhung des Samplings verwendet werden. Im Allgemeinen ist sie relevant für zeitabhängige Dichten und Laser und durch die Kohärenzzeit t_C beschränkt, wobei hier angenommen werden soll, dass $t < t_C$ oder $t_C \to \infty$. Der Parameter ΔD gibt die Auflösung des Detektors an und ist bei realistischer Wahl deutlich größer als Δs und damit unkritisch für die Genauigkeit, was in Unterabschnitt 5.3.3 kurz gezeigt wird. Damit reicht es s, L, θ_{max}, t und λ als Eingaben zu betrachten, wobei letzterem eine besondere Rolle zukommt: Am Ende werden Längenunterschiede zwischen Wegen gemessen, die in der Größenordnung weniger λ liegen. Maximale Verstärkung tritt bei Unterschieden von Vielfachen der Wellenlänge auf, während Auslöschung bei ungeraden Vielfachen der halben Wellenlänge auftritt. Das heißt die absoluten Fehler in Berechnungen der Länge müssen viel kleiner als $\frac{\lambda}{2}$ sein, damit das Ergebnis noch ausreichend genau ist. Teilweise werden die Längen mittels der Wellenzahl k, die von der Wellenlänge abhängt, in Phasen-Winkel umgerechnet. Diese müssen entsprechend auf π genau bestimmt werden, da eine Wellenlänge auf 2π abgebildet wird. Aufgrund der Periodizität der Winkelfunktionen sind diese Phasenfehler äquivalent modulo 2π (z. B. gilt: $-0{,}1 \equiv 2\pi - 0{,}1$). Die Wellenlänge ist beschränkt auf den Bereich zwischen $2{,}4\,\text{pm}$ (Compton-Wellenlänge des Elektrons) und $250\,\text{pm}$ (obere Grenze für Röntgenstrahlung), da Thomson-Streuung von Röntgenstrahlung angenommen wird (vgl. Abschnitt 3.2). Die Maschinengenauigkeit in einfacher ($\mathbf{u}_{SP}$) bzw. doppelter ($\mathbf{u}_{DP}$) Genauigkeit beträgt für die verwendeten IEEE-754 Formate:

$$\mathbf{u}_{SP} = 2^{-24} \approx 5{,}96 \cdot 10^{-8} \tag{5.9}$$

$$\mathbf{u}_{DP} = 2^{-53} \approx 1{,}11 \cdot 10^{-16} \tag{5.10}$$

5.3 Mathematische Operationen in der Simulation

Im Folgenden werden die mathematischen Operationen in den einzelnen Schritten der Simulation (s. Abschnitt 4.3) auf ihre Fehlergrenzen hin untersucht. Um diese zu validieren, wurden einzelne Funktionen in äquivalenter Form in Python implementiert, wobei die zu untersuchenden Operationen mittels der bigfloat Bibliothek [64] ausgeführt werden, die arbiträre Genauigkeit und IEEE-754 Arithmetik erlaubt. Zur Bestimmung des tatsächlichen Fehlers wurde jeweils eine Berechnung mit der Genauigkeit ausgeführt, die in der Simulation möglich ist (einfach oder doppelt). Anschließend wurde der selbe Algorithmus mit den gleichen Werten (vorher auf die selbe Genauigkeit gerundet) in sehr hoher Genauigkeit (512 Bit Mantissenlänge) ausgeführt und die Differenz dieser zwei Ergebnisse gebildet. Für die hier verwendeten Operationen kann der auf diese Art bestimmte Fehler als exakt angesehen werden. Dieser wird in passenden Diagrammen für verschiedene Eingabewerte dargestellt, wobei der Fokus auf den Grenzbereichen liegt, in dem die reduzierte Genauigkeit gerade noch ausreicht. Teilweise wurden mehrere Eingabeparameter variiert, um Abhängigkeiten des simulierten Fehlers von speziellen Eingabewerten zu vermeiden.

5.3.1 Lichtquelle / Erzeugung der Photonen

Die Berechnung der Anzahl der Photonen ist unkritisch, da sie sich lediglich auf die Abtastung des Monte-Carlo-Prozesses auswirkt und damit zu viele Photonen das Ergebnis sogar positiv beeinflussen. Die Position in x-Richtung ist null (der Anfang der Zelle) für alle Photonen, da dies die Ebene ist, an der die Photonen in die Simulation eintreten. In den übrigen Dimensionen (y, z) wird die Position zufällig gesetzt, um Musterbildung zu vermeiden. Diese ist bezogen auf die Zelle und liegt im Intervall $[0, 1)$ (s. Abschnitt 4.2), sodass eventuelle Rundungsfehler hier keinen relevanten Einfluss haben. Die Richtung ist für alle Photonen konstant und im Allgemeinen $\underline{\boldsymbol{e_x}}$, da sich die Welle gleichmäßig in x-Richtung ausbreitet.

Es verbleibt noch die Phase, welche sich nach Gleichung (3.4) berechnet. Die Ortsabhängigkeit ist für die ebene Welle auf einer Ebene senkrecht zur Ausbreitungsrichtung (wie in diesem Fall) konstant und kann mit dem Offset φ_0 kombiniert werden, was als null definiert wird. D. h. alle Phasen sind relativ zur Anfangsphase zum Zeitpunkt $t = 0$. Damit verbleibt $\varphi(t) = -\omega t = -\omega \Delta t \cdot t_n$ mit Δt als Zeitschrittlänge und $t_n = 0, 1, 2, \ldots$ als aktueller Zeitschritt. Da sich hier relevante Ungenauigkeiten ergeben können, muss dies genauer analysiert werden. Die Größen Δt und t_n können als exakte Eingabegrößen angenommen werden, da Δt eine Größe der Simulation ist, die in dieser (eventuell gerundeter) Darstellung konsistent weiter verwendet wird, und t_n eine Ganzzahl ist, die sich für $t_n <= 2^{24}$ exakt in einfacher Genauigkeit[23] darstellen lässt. Zur Vereinfachung sei $\hat{\omega} = fl(\frac{2\pi c}{\lambda}) \approx \omega$ ebenfalls als exakt angenommen, da bei konsistenter Verwendung von ω statt λ in der Simulation, ein Ergebnis für eine Wellenlänge $\hat{\lambda}$ berechnet wird, die nur geringfügig von der tatsächlichen Wellenlänge abweicht. Dies lässt sich wie folgt überprüfen (es gilt gemäß dem Modell aus Abschnitt 5.2 auf Seite 52: $|\delta_i| \leq \mathbf{u}$ für alle i):

$$\omega = fl(\frac{2\pi c}{\lambda}) \tag{5.11}$$

$$= \frac{fl(2\pi c)}{fl(\lambda)}(1 + \delta_1) \tag{5.12}$$

$$= \frac{(2\, fl(\pi)c)(1 + \delta_2))}{fl(\lambda)}(1 + \delta_1) \tag{5.13}$$

$$= \frac{(2\pi c)(1 + \delta_2)(1 + \delta_3)}{\lambda/(1 + \delta_4)}(1 + \delta_1) \tag{5.14}$$

$$= \frac{2\pi c}{\lambda} \prod_{i=1}^{4}(1 + \delta_i) \tag{5.15}$$

$$= \frac{2\pi c}{\hat{\lambda}}, \qquad \hat{\lambda} = \lambda(1 + \theta_4),\ |\theta_4| \leq \frac{4\mathbf{u}}{1 - 4\mathbf{u}} \tag{5.16}$$

Hierbei wurde ausgenutzt, dass sowohl die Zwei als auch $2 \cdot c$ exakt in einfacher Genauigkeit darstellbar sind. Der relative Fehler in $\hat{\lambda}$ beträgt demnach maximal $2{,}4 \cdot 10^{-7}$, was für alle praktischen Anwendungen ausreicht.

[23] Basierend auf 23 Mantissenbits + 1 Hidden Bit

In ähnlicher Weise wird nun der Vorwärtsfehler für $\varphi(t)$ bestimmt:

$$\hat{\varphi}(t_n) = -\,fl(\omega\Delta t \cdot t_n) \tag{5.17}$$

$$= \omega\Delta t \cdot t_n(1+\theta_2), \qquad |\theta_2| \leq \frac{2\mathbf{u}}{1-2\mathbf{u}} \tag{5.18}$$

$$= \varphi(t_n) + \partial\varphi, \qquad \partial\varphi = \varphi(t_n)\cdot\theta_2 \tag{5.19}$$

Setzt man nun die Bedingung $\partial\varphi \ll \pi$ an, ergibt sich für die Eingabegrößen eine Einschränkung von

$$\frac{t}{\lambda} \ll \begin{cases} 1{,}4\cdot 10^{-2}, & \text{einfache Genauigkeit} \\ 7{,}5\cdot 10^{6}, & \text{doppelte Genauigkeit} \end{cases} \tag{5.20}$$

Wie dieser Fehler in einer Beispiel-Simulation mit $\lambda = 100\,\text{pm}$ und $\Delta t = 13{,}3\,\text{as}$ tatsächlich aussieht, ist in Abb. 5.1 dargestellt. Gezeigt wird das laufende Maximum[24] des Fehlers über dem Zeitschritt für zwei mögliche Berechnungen des Produkts, sowie der berechnete maximale Fehler (rot) gemäß Gleichung (5.19). Man sieht, dass die theoretische Grenze sehr gut mit der tatsächlichen übereinstimmt,

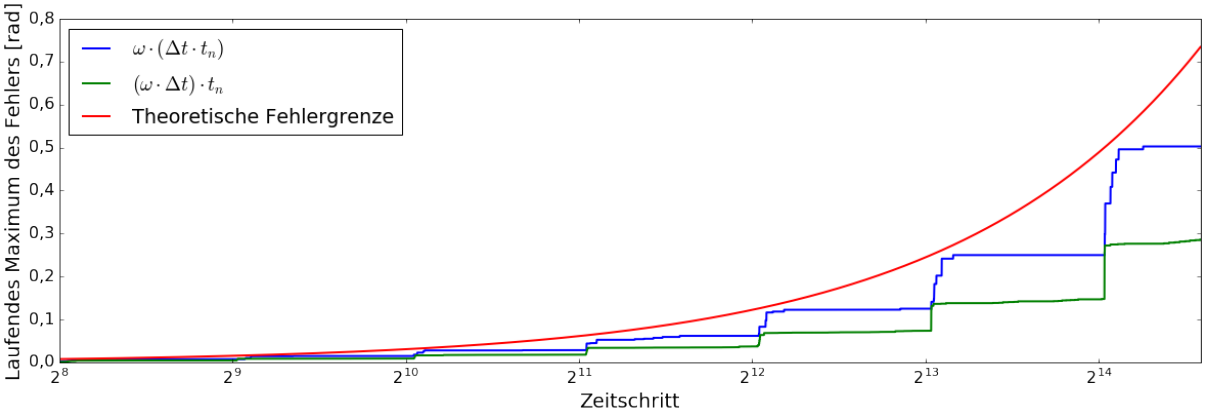


Abbildung 5.1: Laufender max. Fehler bei Berechnung von $\omega \cdot \Delta t \cdot t_n$ in einfacher Genauigkeit

und dass Gleitkommaoperationen tatsächlich nicht assoziativ sind, da sich der Fehler zwischen den zwei Algorithmen um den Faktor ~ 2 unterscheidet. Es lässt sich folgern, dass die zwei Rundungsfehler aus Gleichung (5.19) in dem einen Fall unterschiedliche Vorzeichen haben und sich so gegenseitig abschwächen, während dies in der anderen Variante nicht der Fall ist, was man daran erkennt, dass das Maximum teilweise erreicht wird. Dieser Effekt ist abhängig von den Eingabegrößen, womit keiner der beiden Algorithmen generell genauer ist, als der andere, wie Abb. 5.2 zeigt, in der die Fehler für einen festen Zeitschritt von $t_n = 2^{14} + 2^{13}$ (rechter Rand des letzten Diagramms) aufgetragen sind. Diese, mit einfacher Genauigkeit erreichte Präzision ist nicht ausreichend für eine hohe Anzahl an Zeitschritten insbesondere bei kleineren Wellenlängen: Die in den Diagrammen verwendete Wellenlänge liegt immer noch zwei Größenordnungen unter der minimal angesetzten Wellenlänge, wodurch der Fehler entsprechend noch um 2 Größenordnungen steigen kann.

Die Lösung ist die Verwendung der Periodizität der Winkelfunktionen, um größere Eingabebereiche

[24] Das „laufende Maximum“ ist der maximale Funktionswert (hier: Fehler) über alle Argumente (hier: Zeitschritte) kleiner gleich dem aktuellen Argument.

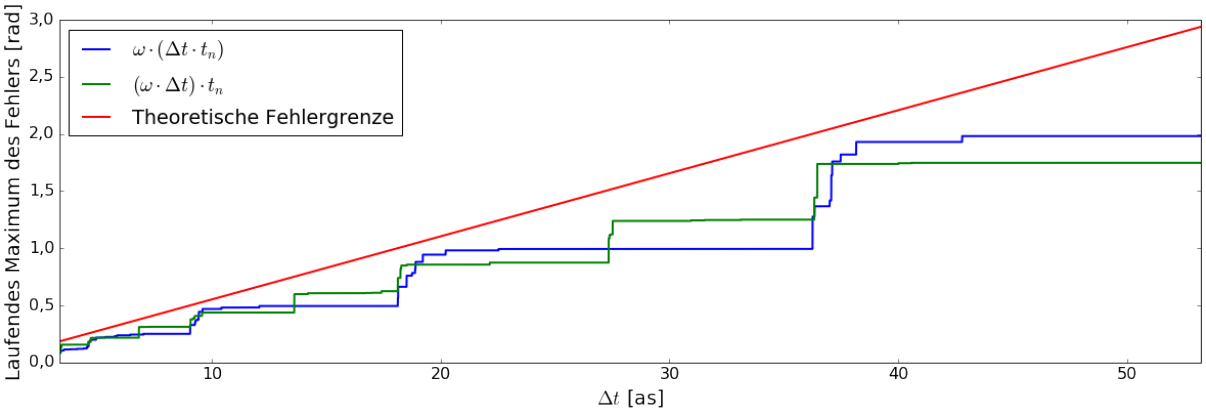


Abbildung 5.2: Laufender max. Fehler bei Berechnung von $\omega \cdot \Delta t \cdot t_n$ in einfacher Genauigkeit über Δt

zu ermöglichen. Dabei müssen die Photonen mit der gleichen Startphase initialisiert werden, was bei einer ebenen Welle der Fall ist. Damit kann man die Phase auf der CPU-Seite in hoher (doppelter) Genauigkeit für alle Photonen eines Zeitschritts berechnen, das Ergebnis in einfacher Genauigkeit auf die GPU kopieren und so weiter verwenden. Die „teure“ Berechnung in doppelter Genauigkeit und die Umwandlung in einfache Genauigkeit muss somit nur einmal gemacht werden, statt redundant für jedes Photon. Das allein reicht jedoch noch nicht aus, da damit das (große) Ergebnis in einfache Genauigkeit gerundet wird, wobei zwar nur ein Rundungsfehler auftritt, jedoch der Fehler lediglich halbiert wird. Man kann jedoch die Phase in doppelter Genauigkeit auf den Bereich $[0, 2\pi)$ bringen und anschließend auf `float` runden, was demzufolge nur noch einen Fehler relativ zu diesem Wert und damit maximal $2\pi\mathbf{u}$ enthält. Dies ist möglich, da auch die Modulo-Operation gemäß IEEE-754 Standard korrekt gerundet ist (vgl. Abschnitt 5.1 auf Seite 49). Die genaue Analyse des Fehlers nach der Modulo-Operation gestaltet sich durch die Rundungsoperation etwas schwieriger, jedoch soll im Folgenden gezeigt werden, dass sich der absolute Fehler nicht wesentlich verändert. Gegeben sei ein fehlerbehafteter Wert $\hat{x} = x(1+\theta_n)$ und ein ebenfalls gerundeter Modul $\hat{y} = y(1+\delta_1)$. Dabei sei angenommen, dass $x\theta_n \ll \hat{y}$, was in diesem Fall gegeben ist, da der Fehler in $\omega\Delta t \cdot t_n$ deutlich kleiner ist, als 2π, wenn Gleichung (5.20) beachtet wird. Dann ist mit der Definition der Remainder-Operation gemäß IEEE die Herleitung des Fehlers wie folgt:

$$
\begin{aligned}
fl(\hat{x} \,\%\, \hat{y}) &= (\hat{x} - \mathrm{rnd}(\frac{\hat{x}}{\hat{y}})y)(1+\delta_2) && && (5.21)\\
&= (\hat{x} - \mathrm{rnd}(\frac{x}{y} + \frac{x\theta_{n+1}}{y})\hat{y})(1+\delta_2) && && (5.22)\\
&= (x + x\theta_n - \mathrm{rnd}(\frac{x}{y})y - \mathrm{rnd}(\frac{x}{y})y\delta_1 + \sigma\hat{y})(1+\delta_2), && \sigma \in \{-1,0,1\} && (5.23)\\
&= (x \,\%\, y + x\theta_n + (x \,\%\, y - x)\delta_1 + \sigma\hat{y})(1+\delta_2) && && (5.24)\\
&\approx x \,\%\, y + x\theta_n + x \,\%\, y\delta_1 - x\delta_1 + x \,\%\, y\delta_2 + \sigma\hat{y}\delta_2 && && (5.25)\\
&\approx x \,\%\, y + x\theta_{n+1} + (x \,\%\, y)2\delta_3 + y\delta_4, && |\delta_4| = |\sigma\delta_2| \leq \mathbf{u} && (5.26)\\
&\approx x \,\%\, y + x\theta_{n+1} + 3y\delta_5 && && (5.27)
\end{aligned}
$$

Der Term $\sigma\hat{y}$ in Gleichung (5.23) bildet hierbei den Fehler in der Rundung (rnd) ab, der durch die Rundungsfehler in $\hat{x}$ und $\hat{y}$ verursacht wird. Mit der Bedingung, dass $x\theta_{n+1} < y/2$ gilt, was durch die vorher

getroffenen Annahmen der Fall ist, kann bei der Rundung zur nächsten Ganzzahl ein maximaler Fehler von ± 1 entstehen, was in der Definition von σ berücksichtigt wurde. Anschließend wurde die Definition der Remainder-Operation rückwärts verwendet, um die zweite Rundungsoperation durch eine Modulo-Operation auszudrücken. Die erste Näherung (Gleichung (5.25)) entsteht durch Ausmultiplizieren und der Vernachlässigung der Fehlerterme höherer Ordnung (Produkte aus θ_i und δ_j), sowie dem Produkt $\sigma\hat{y}$. Letzteres kann weggelassen werden, da es einen absoluten „Fehler“ vom Betrage her exakt $\hat{y}$ erzeugen würde. Aufgrund der Bedingung, dass $|\hat{x} \,\%\, \hat{y}| < \hat{y}$ gilt, bedeutet das lediglich, dass das Ergebnis zwar ein anderes Vorzeichen hat, jedoch äquivalent modulo $\hat{y}$ ist. In der vorletzten Näherung wurde verwendet, dass $|\theta_n \pm \delta| < |\theta_{n+1}|$ sowie $|\delta_i + \delta_j| < 2|\delta|$ gilt, was aus der folgenden Betrachtung bzw. der Dreiecksungleichung folgt:

$$|\theta_n \pm \delta| \leq \frac{n\mathbf{u} + \mathbf{u} - n\mathbf{u}^2}{1 - n\mathbf{u}} < \frac{(n+1)\mathbf{u}}{1 - (n+1)\mathbf{u}} \tag{5.28}$$

$$|\delta_i + \delta_j| < 2\mathbf{u} \tag{5.29}$$

Somit lässt sich die Fehlerdarstellung einfacher ausdrücken, weshalb auch in Gleichung (5.27) die letzten beiden Terme mit Hilfe von $|x \,\%\, y| < y$ zusammengefasst wurden. Dies hat außerdem den Vorteil, dass diese Überschätzung des Fehlers die Unterschätzung in der vorhergehenden Näherung ungefähr ausgleicht, sodass die sich ergebende obere Schranke hinreichend genau ist.

Aus dieser Betrachtung folgt, dass der absolute Fehler durch die Modulo-Operation nicht wesentlich größer wird. Für die Berechnung der reduzierten Phase gilt demzufolge

$$fl(\varphi_{red}) = fl((\omega \cdot \Delta t \cdot t_n) \,\%\, 2\pi) \tag{5.30}$$

$$\approx \varphi_{red} + (\omega \cdot \Delta t \cdot t_n)\theta_3 \tag{5.31}$$

wobei auch der von 2π abhängige Fehlerterm vernachlässigt werden konnte, da er in der Regel mehrere Größenordnungen unter dem anderen liegt. Auch die anschließende Rundung in einfache Genauigkeit, nach der Berechnung der reduzierten Phase in doppelter Genauigkeit, ändert den Wert nicht wesentlich, wie in Abb. 5.3 zu sehen ist (die Kurven für die Berechnung mit und ohne anschließender Rundung liegen fast übereinander). Die Werte für ω und Δt entsprechen denen in Abb. 5.1, man beachte jedoch die deutlich größeren Zeitschritte. Damit lässt sich die Phase exakt berechnen, wenn $\frac{t}{\lambda} \ll 5{,}0 \cdot 10^6$ gilt.

5.3.2 Propagation

Bei der Erzeugung der Zufallsgrößen wird eine Zufallszahl aus dem Intervall $[0, 1)$ erzeugt und auf den gewünschten Bereich hochskaliert. Danach wird (im Fall von Streuung) die Richtung geändert und die Position $\underline{\boldsymbol{p}}$ des Photons berechnet nach $\underline{\boldsymbol{p}} \leftarrow \underline{\boldsymbol{p}} + c\underline{\boldsymbol{k}'}$ mit $\underline{\boldsymbol{k}'}$ als Einheitsvektor der Richtung. Die Berechnung bzw. Skalierung der Zufallsgröße ist unkritisch, da eventuelle Rundungsfehler sich nicht direkt auf das Ergebnis auswirken, sondern lediglich Einfluss auf die Verteilung haben, die bei dem Monte-Carlo-Prozess erzeugt wird. Da diese hauptsächlich von der Elektronendichte abhängt, die als simulierte oder gemessene Größe ebenfalls mit Rundungsfehlern behaftet ist, ändert dieser zusätzliche relative Fehler im Bereich der Maschinengenauigkeit das Ergebnis deutlich weniger, als die Ungenauigkeit bei der Messung bzw. Berechnung.

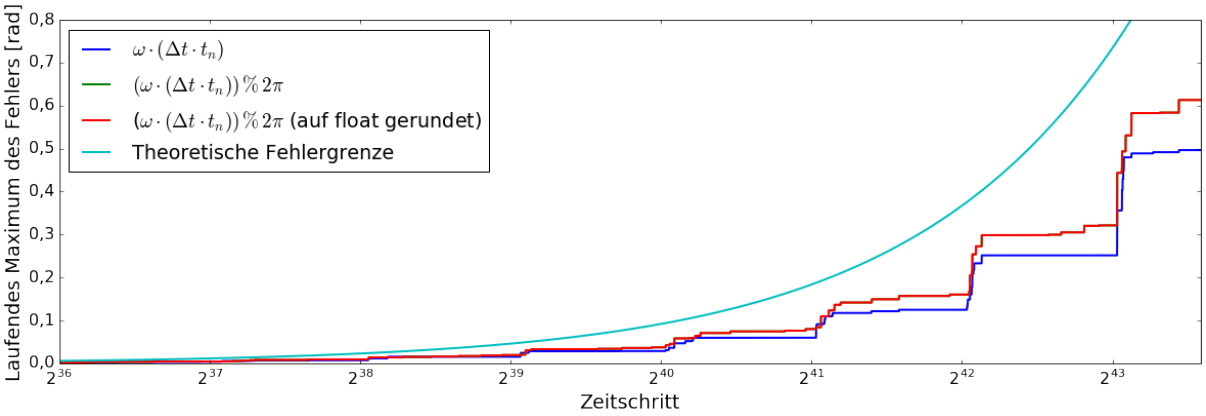


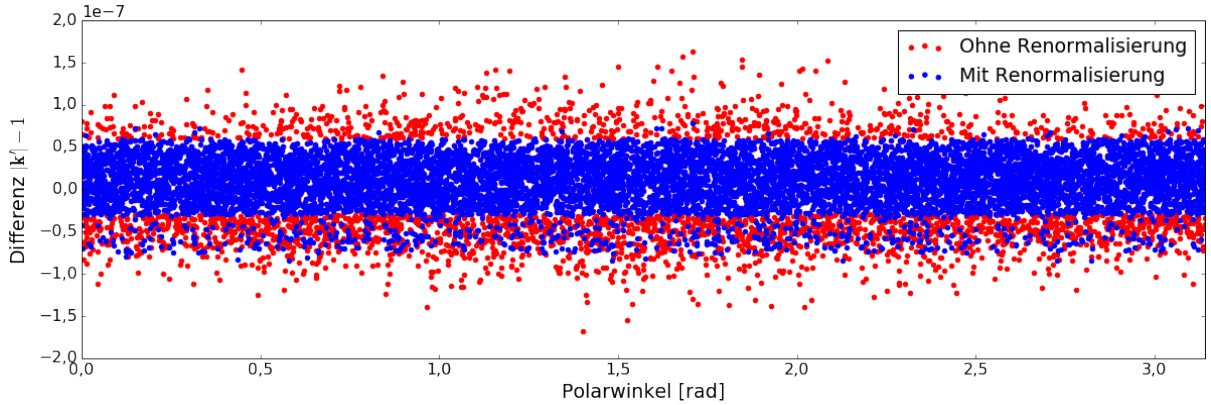
Abbildung 5.3: Fehler bei Berechnung der Phase in doppelter Genauigkeit sowie bei der anschließenden Reduzierung auf 2π

Eventuell kritisch ist die Berechnung der Richtung nach Gleichung (4.3), da diese durch die Singularität an $x = -1$ in diesem Bereich schlecht konditioniert ist. In der Praxis ist dies jedoch nicht relevant, da dies den Fall bezeichnet, in dem sich das Photon entgegen der Ausbreitungsrichtung der eintreffenden Welle bewegt. Dies ist einerseits durch die hohe Wahrscheinlichkeit für kleine Ablenkungswinkel an sich schon unwahrscheinlich, und andererseits ist es noch unwahrscheinlicher, dass ein solches Photon den Detektor erreicht und damit überhaupt einen Beitrag zu dem Ergebnis leistet. Insgesamt folgt aus der Wahrscheinlichkeitsbetrachtung, dass Ungenauigkeiten solcher Photonen das Ergebnis nicht wesentlich beeinflussen. Sollten dennoch Simulationen betrachtet werden, in dem solche großen Ablenkungswinkel in relevanter Anzahl auftreten, ist es möglich, die Rotationsmatrix so umzuschreiben, dass die Singularität stattdessen z. B. bei $x = 1$ auftritt und entsprechend zur Laufzeit die jeweils bessere Variante auszuwählen.

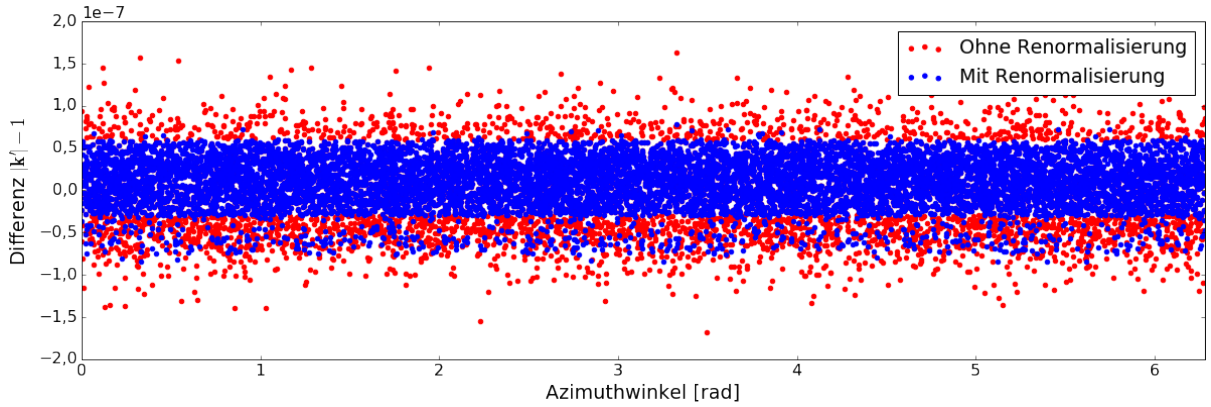
Durch die auftretenden Additionen und Subtraktionen, wobei die eine durch Vorzeichenwechsel in der Winkelfunktion zur anderen werden kann, sind Absorption bzw. Auslöschung möglich. Diese Effekte lassen sich minimieren, in dem zuerst die kleinen Werte zusammengefasst werden. Für den Fall kleiner Ablenkungswinkel θ wäre dafür ein möglicher Algorithmus:

$$\underline{\boldsymbol{k}'} = \underline{\boldsymbol{k}} \cos(\theta) + \begin{pmatrix} -y \sin(\phi) - z \cos(\phi) \\ (x + \frac{z^2}{1+x}) \sin(\phi) - \frac{yz}{1+x} \cos(\phi) \\ (x + \frac{y^2}{1+x}) \cos(\phi) - \frac{yz}{1+x} \sin(\phi) \end{pmatrix} \sin(\theta) \tag{5.32}$$

Dabei wird der aktuelle Richtungsvektor mit dem Faktor $\cos(\theta)$ (für kleine Winkel nahe eins) gewichtet und anschließend gemäß der Winkel um einen kleinen Wert (allein schon durch die Multiplikation mit $\sin(\theta)$) verändert. Unter der Annahme, dass die Photonen sich hauptsächlich in x-Richtung bewegen, gilt außerdem $x \ll y$ bzw. $x \ll z$, was eine mögliche Absorption in u. a. $x + \frac{z^2}{1+x}$ zur Folge hat. Dies lässt sich durch folgende Formulierung verbessern, in der die quadratischen Terme und die mit den Produkt



(a) Fehler über dem Polarwinkel



(b) Fehler über dem Azimutwinkel

Abbildung 5.4: Scatter-Plot der Fehler in der Länge bei der Berechnung der neuen Richtung über 10 Tsd. zufällige Kombinationen von ursprünglicher Richtung und Streuwinkeln

$y \cdot z$ zuerst addiert/subtrahiert werden:

$$\underline{\boldsymbol{k'}} = \begin{pmatrix} x\cos(\theta) \\ x\sin(\phi)\sin(\theta) \\ x\cos(\phi)\sin(\theta) \end{pmatrix} + \begin{pmatrix} -y\sin(\phi)\sin(\theta) - z\cos(\phi)\sin(\theta) \\ y\cos(\theta) + (z^2\sin(\phi) - yz\cos(\phi))\frac{\sin(\theta)}{1+x} \\ z\cos(\theta) + (y^2\cos(\phi) - yz\sin(\phi))\frac{\sin(\theta)}{1+x} \end{pmatrix} \tag{5.33}$$

Abgesehen von diesen möglichen Umformulierungen, soll der genaue Fehler bei der Berechnung des neuen Richtungsvektors nicht weiter analysiert werden, da Fehler hier, genau wie schon bei der Berechnung der Zufallsvariable, lediglich Einfluss auf das Sampling haben, nicht aber (direkt) auf das Ergebnis. Damit sind kleine Abweichungen bei der berechneten Richtung tolerierbar. Wichtig ist jedoch, dass die Invariante $|\underline{\boldsymbol{k}}| = |\underline{\boldsymbol{k'}}| = 1$ erhalten bleibt, da andernfalls eine Renormalisierung notwendig ist. In Abb. 5.4 ist der Fehler $|\underline{\boldsymbol{k'}}| - 1$ über verschiedene Winkel und ursprüngliche Richtungen bei Verwendung einfacher Genauigkeit aufgetragen. Der Fehler ohne Renormalisierung beträgt weniger als $3\mathbf{u}$, was ausreichen sollte. Wenn ein Photon sehr oft gestreut wird ist es jedoch notwendig, den Richtungsvektor neu zu normalisieren, da der Fehler steigen kann. Bei doppelter Genauigkeit scheint es nicht notwendig zu sein, da der Fehler dort im Bereich von 10^{-16} liegt und so das Ergebnis nicht wesentlich beeinflusst.

Durch die endliche Darstellung der Elemente des Richtungsvektor ist auch der mögliche Winkelbereich limitiert. Für einfache Genauigkeit ist dieser bei der kleinstmöglichen Ablenkung $\begin{pmatrix} 1 & 2^{-126} & 0 \end{pmatrix}^T$, was

für alle praktischen Zwecke ausreichen sollte. Da jedoch der Richtungsvektor auf eine Position addiert wird besteht für kleine Winkel die Gefahr der Absorption. Beide Vektoren liegen (elementweise) zwischen null und eins, wodurch der minimale effektive Winkel, für den eine Positionsänderung in dem jeweiligen Element garantiert ist, dem Wert $\arctan(\epsilon_M) \approx 1{,}19 \cdot 10^{-7}$ rad entspricht.

Generell ist die Propagation selbst (nach der eventuellen Richtungsberechnung) die problematische Operation. Durch die Trennung in einen ganzzahligen Zellindex[25] und die Position relativ zum Ursprung der aktuellen Zelle wird bereits eine hohe Genauigkeit erreicht, da bei der Berechnung und Speicherung dieser (Inzell-)Position der Rundungsfehler relativ zur Zellgröße ist.

Im Folgenden soll die geradlinige Propagation eines Photons betrachtet werden, wobei der Fehler in der zurückgelegten Strecke ermittelt werden soll. Eine eventuelle Abweichung in der Richtung sei hier als Teil der Zufallsvariable angesehen (s. o.) und wird deshalb nicht weiter betrachtet. Basierend auf Gleichung (4.5) ergibt sich folgende Gleichung, die in jedem Schritt ausgewertet wird:

$$\underline{\boldsymbol{pos}} \leftarrow (\underline{\boldsymbol{pos}} + \underline{\boldsymbol{k'}} \frac{c\Delta t}{\Delta s}) \,\%\, 1 \tag{5.34}$$

Hierbei ist die modulo-Operation als elementweise Normalisierung auf den Bereich $[0, 1)$ zu verstehen, was in der Implementation durch die Addition von ± 1 erreicht wird, wenn der Bereich verlassen wurde. Die Konstante $\frac{c\Delta t}{\Delta s}$ ist gemäß der Voraussetzung aus Gleichung (4.1) maximal eins, sodass das Ergebnis der Bewegung vor der Renormalisierung elementweise im Bereich $(-1, 2]$ liegt.

Damit lässt sich die Propagation als Summe der Form $((\underline{\boldsymbol{a}} + \underline{\boldsymbol{b}}) \,\%\, 1 + \underline{\boldsymbol{b}}) \,\%\, 1 + \ldots$ ausdrücken. Durch die Reduktion auf eins ist jeder neue Fehler auch relativ zu eins was diesen gering hält. Als grobe Abschätzung des Fehlers nach N Zeitschritten kann $\widehat{\underline{\boldsymbol{pos}}}\,(N) \approx \underline{\boldsymbol{pos}}\,(0) + N(\underline{\boldsymbol{k'}} \frac{c\Delta t}{\Delta s})(1 + \theta_2)$ verwendet werden, was sich aus den zwei Operationen (Addition und Remainder) ergibt. Jetzt lässt sich der Fehler in der zurückgelegten Strecke $d = |\underline{\boldsymbol{pos}}\,(N) - \underline{\boldsymbol{pos}}\,(0)|$ abschätzen und man erhält $\partial d \approx 2N\delta$ unter Verwendung von $|\underline{\boldsymbol{k'}} \frac{c\Delta t}{\Delta s}| \leq 1$. Numerische Experimente unterstützen diese Grenze. Als Beispiel sind in Abb. 5.5 die Fehler in der zurückgelegten Strecke für verschiedene Ablenkungswinkel dargestellt.

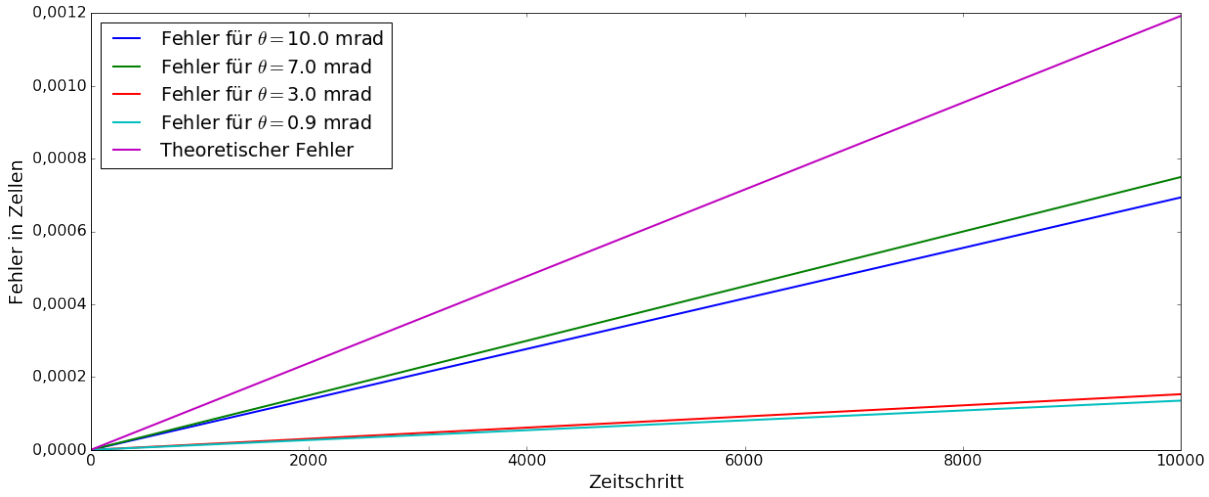


Abbildung 5.5: Fehler in der Länge bei der schrittweisen Propagation

[25] Tatsächlich ist der Zellindex nochmal in einen globalen und einen lokalen Offset unterteilt, was aber vorerst als ein (gemeinsamer) Offset betrachtet werden soll.

Dabei ist die Zellgröße $\Delta s = 4\,\text{nm}$ und der Zeitschritt $\Delta t = 13{,}3\,\text{as}$, was den Größen in den Tests aus Kapitel 6 entspricht.
Es ist zu anzumerken, dass diese Abschätzung auch für mehrere Ablenkungen entlang des Weges gilt. Dafür werden die Fehler in den zurückgelegten Strecken zwischen den einzelnen Streupunkten jeweils separat berechnet und anschließend addiert, wodurch man den Fehler in der zurückgelegten Gesamtstrecke erhält. Daraus wird ersichtlich, dass auch hier die Grenze von $|\partial d| \leq 2N\mathbf{u}$ gilt.
Damit ergibt sich hier eine recht kleine Grenze für N und Δs von $2 \cdot N\Delta s \cdot \mathbf{u} \ll \frac{\lambda}{2}$, was effektiv eine Grenze für die Gesamtgröße des Volumens bzw. dessen Durchmesser darstellt. Mit der Simulationsgröße s und der Beobachtung, dass die Anzahl N der Zeitschritte, die nötig sind, um diese Größe zu durchqueren, dem Term $\frac{s}{c\Delta t}$ entspricht, erhält man $2s\frac{1}{q}\mathbf{u} \ll \frac{\lambda}{2}$ mit $q = \frac{c\Delta t}{\Delta s}$ (s. oben). Der Faktor ist dabei eine Simulationskonstante, die sich aus der Dimensionierung des Zeitschrittes und der Zellgröße ergibt, und kleiner eins sein muss, da sich Partikel in einem Zeitschritt maximal eine Zelle weit bewegen dürfen. Zu kleine Werte sind jedoch auch nicht sinnvoll, da dies die Anzahl der Zeitschritte und damit die benötigte Simulationszeit erhöht. Nimmt man hier einen Wert von $0{,}5 <= q <= 1$ an, ergibt sich die Grenze für die Simulationsgröße zu $s \ll \frac{\lambda}{8\mathbf{u}}$. Bei einer Wellenlänge von $0{,}1\,\text{nm}$, Verwendung von einfacher Genauigkeit und einem Faktor von 10 für die „viel kleiner“ Relation ergibt sich damit eine maximale Simulationsgröße von rund $21\,\mu\text{m}$, was für die Simulation aktuell relevanter Experimente mit Durchmessern von nur wenigen Mikrometern (Proben sind dünne Folien, s. [65, 9]) ausreicht, aber eventuell bei zukünftigen Anwendungen mit dickeren Proben beachtet werden muss.

5.3.3 Detektion

In diesem Abschnitt soll die Erkennung eines Photons auf dem Detektor betrachtet werden, was zum Zeitpunkt t_e das Volumen auf der rechten Seite (gegenüber der Eintrittsfläche) verlassen und nun die Position $\underline{\boldsymbol{p}}$ eingenommen hat, wobei zur einfachen Verdeutlichung die x-Position und alle folgenden x-Koordinaten relativ zur Austrittsfläche sind, da der Detektorabstand als Abstand zu dieser angegeben wird. Es soll angenommen werden, dass das Photon auf den Detektor trifft, was insbesondere für den Bewegungsvektor $\underline{\boldsymbol{k}}$ impliziert, dass $k_x > 0$ gilt, da sich das Photon sonst nicht in Richtung des Detektors bewegt. Dieser befindet sich bei $x = L$ und ist parallel zur Austrittsfläche. Damit berechnet sich die Position $\underline{\boldsymbol{p_d}}$, an der das Photon auf den Detektor trifft, nach

$$\underline{\boldsymbol{p_d}} = \begin{pmatrix} \frac{L-p_x}{k_x}k_z + p_z \\ \frac{L-p_x}{k_x}k_y + p_y \end{pmatrix} \tag{5.35}$$

Hier ist bei der Umwandlung von 3D Simulationskoordinaten in 2D Detektorkoordinaten zu beachten, dass die y-Koordinaten gleich sind, jedoch die x-Dimension des Detektors entlang der z-Dimension der Simulation verläuft. Aus der Position und der Detektorzellgröße lässt sich deren Index ermitteln (und auch, ob es auf dem Detektor landet). Da nur Photonen direkt miteinander interagieren können, die am gleichen Ort sind, die Detektorzellen jedoch eine Ausdehnung haben, die groß gegenüber der Wellenlänge ist, muss noch eine Anpassung des Auftreffpunktes vorgenommen werden. Dazu wird je Zelle ein fester Punkt gewählt, auf dem alle Photonen auftreffen müssen. Das entspricht der Diskretisierung der Fouriertransformierten bei der Fourieroptik. Faktisch wird damit der Streuwinkel des Photons angepasst, was dessen Wahrscheinlichkeitsverteilung ändert. Für den Fall kleiner Winkel und einen geringen Gradi-

enten der Wahrscheinlichkeitsfunktion können die Winkel in einer Zelle als gleich-verteilt angenommen werden, wodurch sich der Fehler im Mittel zu null ergibt (vgl. Unterabschnitt 4.3.4). Für die diskretisierte Position auf dem Detektor $\underline{\boldsymbol{p'_d}}$ ist der Abstand gegeben als $d_n = |\underline{\boldsymbol{p'_d}} - \underline{\boldsymbol{p}}|$ und der Zeitpunkt t'_e des Auftreffens:

$$t'_e = t_e + \frac{d_n}{c} \tag{5.36}$$

Damit lässt sich der Phasenbeitrag des Photons auf dem Detektor berechnen:

$$\begin{aligned} \varphi_{n0} &= kd_n + \omega t_e + \varphi'_n && (5.37)\\ &= \frac{\omega}{c} d_n + \omega t_e + \varphi'_n && (5.38)\\ &= \omega t'_n + \varphi'_n && (5.39) \end{aligned}$$

Die Berechnung des Feldes auf dem Detektor erfolgt anschließend durch Aufsummieren der komplexen Werte. Lässt man den Skalierungsfaktor eines Photons außer Acht (kann als konstanter Faktor aus der Summe gezogen und separat betrachtet werden) ergibt sich:

$$\begin{aligned} D(y,z) &= \sum_{n=1}^{N(y,z)} \mathrm{e}^{\varphi_{n0}} && (5.40)\\ &= \sum_{n=1}^{N(y,z)} \cos(\varphi_{n0}) + i \sum_{n=1}^{N(y,z)} \sin(\varphi_{n0}) && (5.41) \end{aligned}$$

Die zweite Darstellung entspricht dabei der kartesischen Form der komplexen Zahl, für die sich die Addition leichter ausführen lässt, weshalb sie im folgenden verwendet wird.

Durch die Diskretisierung auf jeweils einen Punkt pro Zelle, kann der Prozess der Detektion in zwei Phasen unterteilt werden. In der ersten wird der Index der getroffenen Detektorzelle ermittelt und in der zweiten der Beitrag des Photons für den Punkt in dieser Zelle berechnet und addiert. Da beide Phasen unterschiedliche Genauigkeitsanforderungen haben, sollen diese getrennt voneinander betrachtet werden.

Die Bestimmung der Detektorzelle muss offensichtlich auf eine Zellgröße genau erfolgen. Es ist leicht zu erkennen, dass sich Gleichung (5.35) hierfür nicht eignet, da die Projektion der kleinen Abweichungen von der Hauptrichtung auf die üblicherweise große Distanz zum Detektor zu einer Verstärkung des Fehlers führt und insbesondere die Addition anfällig für Absorption ist. Darum werden die Detektorzellen nicht gleich groß gewählt, sondern so, dass sie gleiche Winkelbereiche abdecken, was auch die direkte Vergleichbarkeit mit der Fouriertransformierten vereinfacht, in der die „Zellen“ proportional zum Sinus des Winkels sind und für kleine Winkel, wie sie hier betrachtet werden, $\sin(\alpha) \approx \alpha$ gilt. Zusätzlich kann man zeigen, dass für kleine Winkel die Detektorzellen immer noch annähernd gleich groß sind und sich die resultierende Intensitätsverteilung über Winkelbereiche auch auf feste Abstände umrechnen lässt, was einen gewissen Freiraum in der Definition des von einer Detektorzelle abgedeckten Winkelbereichs $\Delta D'$ lässt, solange dieser fest und dokumentiert ist. Für die Bestimmung dieses Winkelbereichs $\Delta D'$ aus der tatsächlichen Zellgröße kann z. B. der mittlere Winkelbereich über alle Zellen gewählt werden, was hier aber bewusst nicht gemacht wurde. Stattdessen wird er definiert zu $\Delta D' = \arctan(\frac{\Delta D}{D})$, damit die Größen der inneren Detektorzellen am wenigsten verändert werden, sodass dieser, häufig interessantere Bereich, meist schon ohne Umrechnung auf Längen genutzt werden kann. Damit ergibt sich der

Detektorzellindex $\underline{\boldsymbol{i_D}}$ zu

$$\underline{\boldsymbol{i_D}} = \left\lfloor \begin{pmatrix} \arctan(\frac{k_z}{k_x}) \\ \arctan(\frac{k_y}{k_x}) \end{pmatrix} \frac{1}{\Delta D'} + \begin{pmatrix} p_z - \frac{s}{2} \\ p_y - \frac{s}{2} \end{pmatrix} \cdot \frac{1}{\Delta D'} \right\rfloor + \begin{pmatrix} N_D \\ N_D \end{pmatrix} \tag{5.42}$$

Der erste Term entspricht einem „Binning“ des Winkels, wie es auch bei der Erstellung von Histogrammen üblich ist, der zweite einer (häufig sehr kleinen) Korrektur, da die aus dem ersten Term resultierende Detektorzelle für Photonen gilt, die das Simulationsvolumen mittig verlassen, und demzufolge das Offset des Photons zur Mitte noch mit einfließen muss. Anschließend wird die halbe Detektorgröße addiert, um die Mitte des Detektors gegenüber der Mitte des Simulationsvolumens zu platzieren. Die Addition vor der Rundung ist unter der sinnvollen Annahme, dass das Simulationsvolumen maximal einige wenige Detektorzellen groß ist, anfällig für Absorption, insbesondere für größere Winkel und Photonen nahe der Mitte der Austrittsfläche, was zwar nur kleine Fehler zur Folge hat, aber teilweise dazu führt, dass Photonen einer Nachbarzelle zugeordnet wird, was insbesondere in den künstlichen Tests der Implementation zu unschönen Effekten führte. Die implementierte Lösung trennt daher den ganzzahligen Anteil aus dem ersten Term ab, und addiert den verbleibenden Teil und den zweiten Term. Beide Teilergebnisse werden anschließend in Ganzzahlarithmetik zusammengefügt, was sich algorithmisch (Pseudocode) je Komponente wie folgt darstellt:

$$\begin{aligned} \text{float fIdx} &\leftarrow \arctan(k_z/k_x)/\Delta D \\ \text{int iIdx} &\leftarrow \lfloor \text{ fIdx } \rfloor \\ \text{fIdx} &\text{ -= iIdx} \\ \text{fIdx} &\text{ += } (p_z - \tfrac{s}{2})/\Delta D \\ \text{iIdx} &\text{ += } \lfloor \text{ fIdx } \rfloor + N_D \end{aligned}$$

Das ist eine ähnliche Technik, wie sie auch der Kahan-Summe (beschrieben von Kahan [66]) zugrunde liegt, da in der dritten Zeile eine Art Korrektur- oder Fehlerterm berechnet wird. Anschließend werden Terme ähnlicher Größe addiert, was den absoluten Fehler gering hält. Die anschließende Addition in Ganzzahlarithmetik ist dann wieder exakt für alle praktisch relevanten Bereiche. Auffällig ist die Vernachlässigung von p_x, was aber aufgrund von $|p_x| \leq \Delta s \ll \Delta D$ nur einen vernachlässigbaren Fehler erzeugt, was sich leicht durch Berechnung der Werte bei einem Winkel von $45°$ (max. Fehler) nachprüfen lässt. Es ist leicht zu zeigen, dass die berechnete Zelle hinreichend genau bestimmt wird, da der Fehler kleiner als $c_e \mathbf{u} \Delta D$ ist, mit c_e als kleine Konstante, deren genaue Bestimmung über die Fehleranalyse möglich ist. Der maximale Fehler ist weniger als eine Detektorzelle, der dann auftritt, wenn das Photon in den Randbereich einer Zelle trifft. Dieser entspricht ungefähr dem, der durch die Diskretisierung durch das Binning verursacht wird und ist damit unkritisch.

Damit komme ich nun zur Betrachtung des Beitrags zum Wert auf der Detektorzelle. Die Berechnung der Distanz d_n stellt sich sehr schnell als zu ungenau heraus, da darin ein Term der Form $L^2 + \epsilon^2$, mit $\epsilon = f(\Delta s, \Delta D)$ vorkommt, in dem durch $L \gg \Delta s$ sehr schnell so starke Absorption auftritt, dass effektiv $d_n = L$ berechnet wird, was offensichtlich einen Fehler größer der Wellenlänge erzeugt. Der Fall totaler Absorption, für den $fl(L^2 + \epsilon^2) = L^2$ gilt, tritt auf für ca.[26] $L^2 \mathbf{u} \geq \epsilon^2$, was den verwendbaren Bereich zu stark einschränkt. Selbst ohne die Absorption ist der maximale Abstand des Detektors, für den einfache Genauigkeit ausreicht, zu gering. Dies zeigt sich aus der Betrachtung von $d_n = L + \Delta L$, mit $\Delta L \geq 0$ als

[26] Durch Rundungsfehler bei der Quadrierung ist die tatsächliche Grenze um einen Faktor kleiner $\mathbf{u}$ verschieden.

zusätzlich zurückgelegter Weg durch die Ablenkung, wodurch sich selbst mit der groben Abschätzung $fl(d_n) \approx L(1+\delta)$ und damit ein Fehler $|\partial d_n| \leq L\mathbf{u}$ ergibt. Mit der Begrenzung des Fehlers auf die halbe Wellenlänge ist der Abstand L auf $L \ll \frac{\lambda}{2\mathbf{u}}$ begrenzt und damit sogar für die längste Wellenlänge im Millimeter Bereich. Das schließt die Berechnung der exakten Entfernungen in einfacher Genauigkeit aus, wie Abb. 5.6 sehr gut zeigt: Selbst für sehr kleine Winkel, und damit kleinen d_n, ist der Fehler deutlich zu groß für eine Wellenlänge von z. B. $0{,}1$ nm.

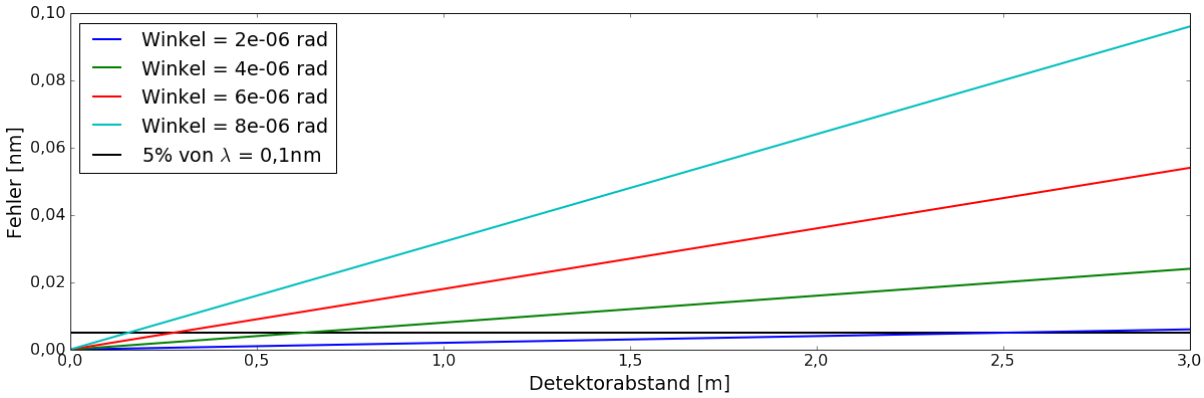


Abbildung 5.6: Absoluter Fehler bei der Abstandsberechnung über $\sqrt{L^2+\epsilon^2}$ in einfacher Genauigkeit

Eine Lösung bietet sich aus der Erkenntnis, dass die Intensität von Phasenoffsets unbeeinflusst ist. Demzufolge reicht es aus, die Längendifferenz zu einer festen Länge zu berechnen, was lediglich einen Phasenoffset erzeugt, der, da er für alle Photonen gleich ist, am Ende in einen Phasenoffset des komplexen Detektorwertes resultiert und die Intensität nicht beeinflusst. Insbesondere ist es möglich, diesen, möglicherweise in höherer Genauigkeit, einmalig zu berechnen und damit das Ergebnis zu korrigieren, wenn die Phaseninformation erhalten bleiben soll. Alternativ kann dieser auch bei der Startphase als Teil des φ_0 betrachtet werden. Die augenscheinlich einfachste Variante, $d'_n = d_n - L$ zu berechnen, ist nur möglich, wenn d_n nicht direkt berechnet wird, da sonst durch katastrophale Auslöschung nur ein zu ungenaues Ergebnis herauskommt. Jedoch ist eine Äquivalenzumformung der Form $\sqrt{L^2+\epsilon^2} - L^2 = \frac{\epsilon^2}{\sqrt{L^2+\epsilon^2}+L^2}$ möglich, was im Falle der Absorption in $\frac{\epsilon^2}{2L^2}$ übergeht. Dies entspricht dem ersten Glied der Laurent-Reihe (Reihenentwicklung für $L \to \infty$), welche einen Fehler [27] kleiner $\frac{\epsilon^4}{8L^3}$ hat. Dieser ist im logarithmischen Plot in Abb. 5.7 für einen festen Abstand L dargestellt und zeigt eine deutliche Verbesserung gegenüber der Berechnung der vollständigen Distanz für kleine ϵ, da für den Fall der Absorption der Fehler durch den der Laurent-Reihe dominiert wird, welcher gerade für diesen Bereich sehr klein ist. Trotz dieser Verbesserung ist der numerische Fehler für größere Detektorabstände und Ablenkungswinkel immer noch zu groß, wie aus Abb. 5.8 ersichtlich ist. Ein Teil dieses Fehlers liegt immer noch an der Größe des Ergebnisses, da dieses vom Abstand, von der Größe der Simulation und der des Detektors abhängt. Zusätzlich ist immer noch die Berechnung von Termen mit L kritisch, da diese anfällig für Absorption sind, obwohl dieser Effekt zumindest verringert wurde.

[27] Fehler der Reihe selbst, ohne Betrachtung numerischer Abweichungen

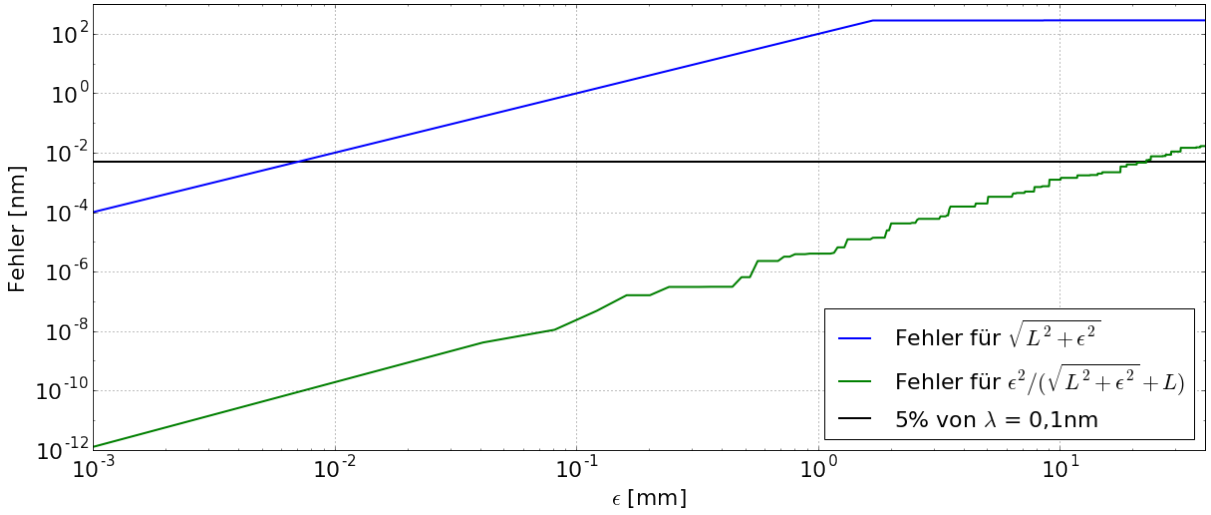


Abbildung 5.7: Laufender max. Fehler bei Berechnung des (reduzierten) Abstands in einfacher Genauigkeit für $L = 5\,\mathrm{m}$

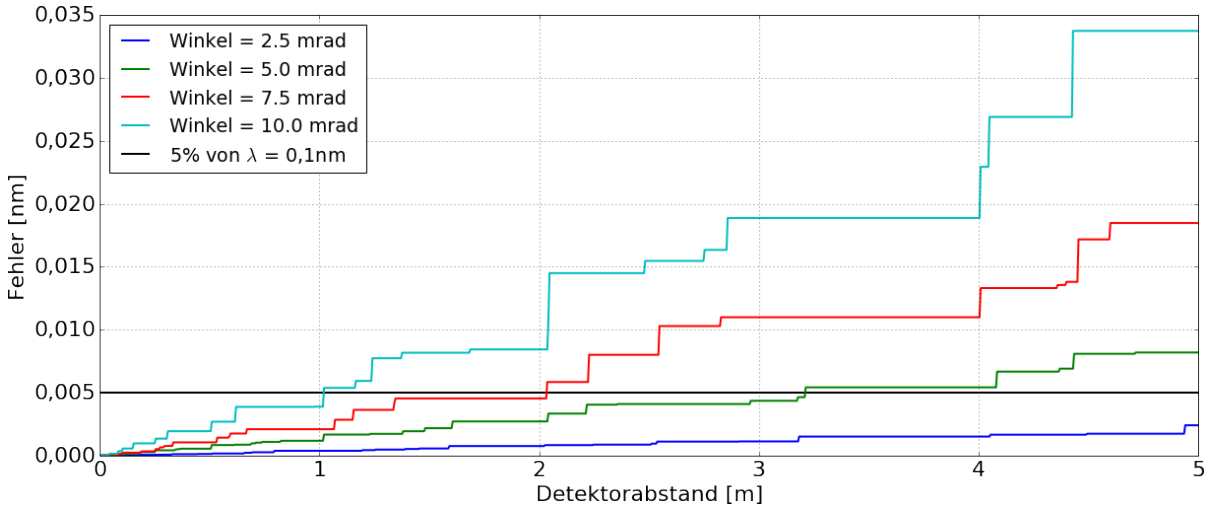


Abbildung 5.8: Laufender max. Fehler bei Berechnung von $\frac{\epsilon^2}{\sqrt{L^2+\epsilon^2}+L^2}$ in einfacher Genauigkeit

Die in der Implementation genutzte Lösung berechnet nur Längenunterschiede bezüglich des Strahls, der von der Mitte der Austrittsfläche in die jeweilige Detektorzelle verläuft. Damit ist die maximal mögliche Differenz so klein wie möglich und befindet sich in der Größenordnung von einigen wenigen Wellenlängen, was die Rundung des Ergebnisses auch in einfacher Genauigkeit unkritisch macht. Über die Fernfeldnäherung (s. Abschnitt 3.2) kann dieser Längenunterschied nur mit Hilfe der aktuellen Richtung und des Abstands zur Mitte der Austrittsfläche (allgemeiner: zum Ursprung des Referenzstrahls) bestimmt werden. Der Abstand zum Detektor wird hier nicht mehr (direkt) benötigt und die Winkeldifferenz $\Delta\varphi$ kann mit Gleichung (3.17) auf Seite 22 bestimmt werden, wobei diese jedoch nur auf den zweidimensionalen Fall direkt anwendbar ist. Im vorliegenden, dreidimensionalen Fall muss die Projektion über Vektoren verwendet werden. Die entsprechende Formel ist dann:

$$\Delta\varphi = \underline{\boldsymbol{k_n}} \cdot \underline{\boldsymbol{p_s}} = k\frac{\underline{\boldsymbol{dir}} \cdot \underline{\boldsymbol{p_s}}}{|\underline{\boldsymbol{dir}}|} \tag{5.43}$$

Dabei bezeichnet $\underline{\boldsymbol{dir}}$ die Richtung zur Detektorzelle, in der das Photon landet. Seien die Winkel zu dieser Zelle $\alpha_y = i_y \cdot \Delta D'$ und $\alpha_z = i_z \cdot \Delta D'$, wobei $i_{y,z}$ die Zellindizes nach Gleichung (5.42) bezeichnet, dann ist die Richtung $\underline{\boldsymbol{dir}} = \left(1, \tan(\alpha_y), \tan(\alpha_z)\right)^T$. Der Vektor $\underline{\boldsymbol{p_s}} = \underline{\boldsymbol{p}} - \left(0, \frac{s}{2}, \frac{s}{2}\right)^T$ ist der Abstand von der Mitte der Austrittsfläche und besteht aus zwei Teilen: Der Position $\underline{\boldsymbol{i_c}}$ in der Simulationszelle und dem Abstand $\underline{\boldsymbol{i_g}}$ zur Mitte der Simulation in Zellen. Letzteres lässt sich aus dem Zellindex des Photons (vgl. Abschnitt 4.2) und der Gesamtzahl der Zellen in der Simulation exakt bestimmen. Um Absorption zu vermeiden, werden diese Teile gemäß folgender Herleitung getrennt:

$$\Delta\varphi = k\frac{\underline{\boldsymbol{dir}} \cdot \underline{\boldsymbol{p_s}}}{|\underline{\boldsymbol{dir}}|} \tag{5.44}$$

$$= k\frac{\underline{\boldsymbol{dir}} \cdot (\underline{\boldsymbol{i_c}} + \underline{\boldsymbol{i_m}})\Delta s}{|\underline{\boldsymbol{dir}}|} \tag{5.45}$$

$$= k\frac{\underline{\boldsymbol{dir}} \cdot (\underline{\boldsymbol{i_c}}\,\Delta s)}{|\underline{\boldsymbol{dir}}|} + k\frac{\underline{\boldsymbol{dir}} \cdot (\underline{\boldsymbol{i_g}}\,\Delta s)}{|\underline{\boldsymbol{dir}}|} \tag{5.46}$$

Das hat den Vorteil, dass nun aufgrund der Periodizität, die auch schon in Unterabschnitt 5.3.1 verwendet wurde, die Summanden vor der Addition mit Hilfe der remainder-Operation auf den Bereich von $\pm 2\pi$ gebracht werden kann, sodass der maximale Fehler bei der Addition relativ zu 4π und demzufolge klein genug ist, damit er selbst bei Verwendung einfacher Genauigkeit vernachlässigt werden kann. Ähnliches gilt für die Addition der Komponenten aus dem Skalarprodukt, sodass sich die Berechnung von $\Delta\varphi$ letztlich als Summe der Form $\Delta\varphi = \sum \frac{k}{|\underline{\boldsymbol{dir}}|} dir_\sigma i_\varsigma$ darstellt. Hierfür können allgemeine Techniken zur Fehlerreduktion in Summen verwendet werden, wie sie u. a. Higham [50, S. 80 ff.] beschreibt. Als Kriterium zur Auswahl bzw. Entwurf einer entsprechenden Methode, sollten die Beträge der Zwischenergebnisse minimiert werden (vgl. [50, S. 82]). Das wird erreicht, in dem einerseits die Summanden in Gleichung (5.46) modulo 2π reduziert werden und andererseits ausgenutzt wird, dass man kleine Winkel betrachtet und demnach i. A. die Beiträge aus den y- und z- Komponenten (im ersten Summand) betragsmäßig kleiner sind, als der aus den x-Komponenten. Deshalb werden diese Beiträge zuerst summiert und anschließend auf den aus den x-Komponenten addiert. Im zweiten Summand ist die x-Komponente meist null, weshalb die Auswirkung auf den Fehler durch unterschiedliche Reihenfolge der Additionen hier vernachlässigt werden kann.

Damit folgt nun die Vorwärtsanalyse des Fehlers in $\Delta\varphi$. Mit der Absicht, die Fehler bei einigen Eingabegrößen so weit wie möglich vernachlässigen zu können und möglichst kleine Fehlergrenzen zu erhalten, soll zuerst festgestellt werden, dass die Berechnung des Richtungsvektors $\underline{\boldsymbol{dir}}$ als praktisch exakt angesehen werden kann. Hierbei muss die Berechnung von $\tan(\alpha)$ für die y und z Komponente betrachtet werden. Der Winkel ergibt sich aus dem Zellindex i_D relativ zur Mitte des Detektors ($id_d - N_D$) und dem Winkel je Zelle zu $\alpha = (i_D - N_D)\Delta D'$. Für kleine Winkel kann der Tangens mit $\tan(\alpha) \approx \alpha + \frac{x^3}{3}$ genähert werden. Die Genauigkeit der (Hardware-)Implementation dieser Funktion ist im *CUDA C Programming Guide* als maximal 4 ulps Differenz zum korrekt gerundeten Ergebnis angegeben, womit der relative Fehler maximal $9\mathbf{u}$ ist (vgl. Abschnitt 5.2 auf Seite 53). Damit gilt:

$$
\begin{aligned}
fl(\tan(\alpha)) &= \tan(\alpha(1+\delta_1))(1+9\delta_2) && (5.47)\\
&\approx \alpha(1+\delta_1)(1+9\delta_2) + \frac{(\alpha(1+\delta_1))^3}{3}(1+9\delta_2) && (5.48)\\
&\approx \alpha(1+10\delta_3) + \frac{(\alpha(1+\delta_1)(1+3\delta_4)^3}{3} && (5.49)\\
&\approx \tan(\alpha(1+10\delta_3)) = \tan((i_D - N_D)\Delta D'(1+10\delta_3) && (5.50)
\end{aligned}
$$

Hierbei wurden wieder Fehlerterme höherer Ordnung vernachlässigt, und eben diese Approximation verwendet, um ein δ_4 zu finden, sodass $(1+\delta_4)^3 \approx (1+9\delta_2)$ gilt, wobei der maximale Fehler dabei sogar überschätzt wird. Aus der weiteren (Über-)Abschätzung, dass $(1+\delta_1)(1+3\delta_4) \leq (1+10\delta_3)$ folgt die letzte Näherung. Damit wird die Richtung für einen Winkel berechnet, dessen maximale relativer Fehler $10\mathbf{u} \approx 6 \cdot 10^{-7}$ beträgt, was in der Regel unterhalb der Messgenauigkeit liegt. Als Konsequenz kann die berechnete Richtung $\underline{\boldsymbol{dir}}$ als exakt angenommen werden, wobei das Ergebnis dann für einen unwesentlich verschiedenen Winkel gilt.

Der relative Fehler von $fl(|\underline{\boldsymbol{dir}}|)$ kann mit θ_4 abgeschätzt werden. Die Herleitung sei hier kurz skizziert und nutzt u. a. dass $|\underline{\boldsymbol{dir}}|$ mindestens eins aber aufgrund der kleinen Winkel auch nicht deutlich größer ist, um den zweiten Summand in Gleichung (5.54) zu $|\underline{\boldsymbol{dir}}|\theta_3$ abzuschätzen. Diese Abschätzung kann mittels Bestimmung der minimal und maximal möglichen Werte des Summanden erfolgen, was zu diesem Ergebnis führt. Die anderen Umformungen und Abschätzungen basieren auf den bereits verwendeten Techniken nach dem Modell aus Abschnitt 5.2.

$$
\begin{aligned}
fl(|\underline{\boldsymbol{dir}}|) &= \sqrt{(1 + dir_y^2(1+\theta_2) + dir_z^2(1+\theta_2))(1+\delta_1)}(1+\delta_2) && (5.51)\\
&= \sqrt{1 + dir_y^2 + dir_z^2 + (dir_y^2 + dir_z^2)\theta_3 + \delta_1}(1+\delta_2) && (5.52)\\
&\approx (|\underline{\boldsymbol{dir}}| + \frac{(dir_y^2 + dir_z^2)\theta_3 + \delta_1}{2|\underline{\boldsymbol{dir}}|}(1+\delta_2) && (5.53)\\
&\approx (|\underline{\boldsymbol{dir}}| + \frac{(|\underline{\boldsymbol{dir}}|^2 - 1)\theta_3 + \delta_1}{2|\underline{\boldsymbol{dir}}|}(1+\delta_2) && (5.54)\\
&\approx |\underline{\boldsymbol{dir}}|(1+\theta_3)(1+\delta_2) && (5.55)\\
&\approx |\underline{\boldsymbol{dir}}|(1+\theta_4) && (5.56)
\end{aligned}
$$

Damit kann schließlich der Fehler $\partial\Delta\varphi$ in Gleichung (5.46) abgeschätzt werden. Nun wird zuerst der vom Abstand $\underline{\boldsymbol{i_g}} = \begin{pmatrix} i_x & i_y & i_z \end{pmatrix}^T$ abhängige Summand φ_g betrachtet. Zur kompakten Darstellung der

folgenden Herleitung werden folgende Größen definiert:

$$\kappa_x = i_x dir_x \Delta s \qquad (\kappa_y, \kappa_z \text{ analog dazu}) \tag{5.57}$$

$$\varpi = \frac{\omega}{c|\underline{\boldsymbol{dir}}|} \tag{5.58}$$

Folgende Größen werden als exakt angenommen: $c, \Delta s, \omega$ als Eingabegrößen (vgl. vorherige Kapitel), $\underline{\boldsymbol{dir}}$ gemäß obiger Argumentation, sowie $\underline{\boldsymbol{i_g}}$, da dieser Fehler und dessen Einfluss später betrachtet wird.

$$fl(\varphi_g) = \Big(\kappa_x(1+\theta_2) + (\kappa_y(1+\theta_2) + \kappa_z(1+\theta_2))(1+\delta_1)\Big)(1+\delta_2)\varpi\frac{1+\theta_2}{1+\theta_4} \tag{5.59}$$

$$= \Big(\kappa_x(1+\theta_{13}) + \kappa_y(1+\theta_{14}) + \kappa_z(1+\theta_{14})\Big)\varpi \tag{5.60}$$

$$= \varphi_g + (\kappa_x\theta_{13} + \kappa_y\theta_{14} + \kappa_z\theta_{14})\varpi \tag{5.61}$$

$$|\partial\varphi_g| \leq (|\kappa_x|\theta_{13} + |\kappa_y|\theta_{14} + |\kappa_z|\theta_{14})\varpi \tag{5.62}$$

$$\leq |\varphi_g|\theta_{14} \tag{5.63}$$

Die Schwierigkeit bei der Herleitung rührt u. a. daher, dass nicht davon ausgegangen werden kann, das die κ-Terme alle das gleiche Vorzeichen haben, wodurch das Ausklammern der θ-Terme möglich gewesen wäre. Stattdessen wurde bei der Bestimmung des Fehlers die Dreiecksungleichung verwendet und anschließend ausgenutzt, dass der Betrag der Phase maximal wird, wenn die κ-Terme das gleiche Vorzeichen haben, wodurch der Betrag der Phase nach oben abgeschätzt werden konnte. Die Fehlergrenze könnte noch etwas niedriger angesetzt werden, wenn die Reihenfolge der Additionen vollständig ausgenutzt werden würde, die in dem Unterschied der θ-Indizes resultierte. Um ein kompaktes Ergebnis zu erhalten, wurde dies jedoch nicht verwendet. Der Fehler im zweiten Summand von Gleichung (5.46), der als φ_c bezeichnet werden soll, lässt sich analog dazu bestimmen. Diese beiden Summanden werden mittels der remainder-Operation auf 2π reduziert und anschließend addiert. Wie bereits in Abschnitt 5.3.1 auf Seite 58 gezeigt wurde, bleibt der absolute Fehler dabei im Wesentlichen unverändert, erhält aber eine Erhöhung des θ-Indexes.

Nun kann der Beitrag des Photons basierend auf Gleichung (4.13) berechnet werden, wobei kd_n durch das Ergebnis nach Gleichung (5.46) ersetzt wird, was effektiv einen weiteren, von der Position der Detektorzelle abhängigen Phasenoffset erzeugt. Dieser ist aber, wie bereits diskutiert, für die Intensität unerheblich. Damit ergibt sich:

$$\varphi'_{n0} = -\varphi_g \,\%\, 2\pi - \varphi_c \,\%\, 2\pi + \omega t_e \,\%\, 2\pi + \varphi'_n \tag{5.64}$$

Die negativen Vorzeichen ergeben sich dabei aus der genutzten Konvention für die Richtung der Winkel α_y, α_z, nach der größere Winkel einen höheren Zellindex auf dem Detektor zur Folge haben. Die Additionen erzeugen einen absoluten Fehler von maximal $6\mathbf{u}\cdot\pi$, wenn auch die Zwischenergebnisse gegebenenfalls reduziert werden. Dieser lässt sich noch halbieren, indem die Summanden auf den Bereich $\pm\pi$ reduziert, statt auf $\pm 2\pi$. Im Allgemeinen ist er aber deutlich kleiner als die absoluten Fehler, die hier summiert werden. Man beachte, dass der Term $\omega t_e \,\%\, 2\pi$, wie schon in Unterabschnitt 5.3.1 beschrieben, in hoher Genauigkeit berechnet werden kann. Um den absoluten Fehler $\partial\varphi'_{n0}$ zu erhalten, müssen nun lediglich die absoluten Fehler der Summanden addiert werden. Zusätzlich kommt ein Beitrag durch die

Berechnung der Position dazu, der durch den Fehler in der zurückgelegten Distanz $\partial d = \hat{d} - d$ bestimmt ist, dessen Abschätzung in Unterabschnitt 5.3.2 betrachtet wurde. Dieser Beitrag wird dann maximal, wenn der Fehler in der Distanz entlang der Flugrichtung liegt und sich dann zu $k\partial d$ ergibt. Damit erhält man für die Startzeit t_s des Photons und unter Verwendung von γ^{DP} zur Kennzeichnung der Nutzung höherer Genauigkeit:

$$|\partial\varphi'_{n0}| \leq |\varphi_g|\gamma_{15} + |\varphi_c|\gamma_{15} + \omega t_e \gamma_3^{DP} + \omega t_s \gamma_3^{DP} + 2kN\mathbf{u} \tag{5.65}$$

$$\leq (|\varphi_g| + |\varphi_c|)\gamma_{15} + \omega(t_s + t_e)\gamma_3^{DP} + 2kN\Delta s u \tag{5.66}$$

Um hieraus nun Grenzen für die möglichen Eingabeparameter zu erhalten, wird der zurückgelegte Weg mit $2s$ abgeschätzt, was groß genug sein sollte, um alle relevanten Streuprozesse zu berücksichtigen. Sollten Photonen mehr als diesen Weg zurücklegen, müssen sie mehrfach und eventuell mit großen Winkeln abgelenkt worden sein, was deren Wahrscheinlichkeit, am Ende auf dem Detektor aufzutreffen stark verringert. Die Anzahl solcher Teilchen, ist demnach deutlich kleiner, als die derer, die nur wenig abgelenkt wurden, wodurch sie nur einen vernachlässigbaren Einfluss auf das Ergebnis haben. Durch diese Festlegung ergibt sich $N = \frac{2s}{c\Delta t}$ und $t_e = t_s + \frac{2s}{c}$. Der Term $|\varphi_g| + |\varphi_c|$ entspricht hier der maximalen Phasendifferenz, die bei dem maximal möglichen Winkel $\alpha_{max} \approx \arctan(\frac{D}{L})$ auftritt und rund $k \sin\alpha_{max} \frac{s}{2} = k\frac{Ds}{2\sqrt{L^2+D^2}} \approx k\frac{Ds}{2L}$ beträgt. Dabei wurde der Abstand in Propagationsrichtung vernachlässigt, da dieser für die meisten Photonen nahe null ist. Außerdem wird wieder $q = \frac{c\Delta t}{\Delta s}$ mit $0{,}5 \leq q \leq 1$ verwendet (vgl. Abschnitt 5.3.2 auf Seite 63). Einsetzen dieser Beziehungen in Gleichung (5.66) führt dann zu

$$|\partial\varphi'_{n0}| \leq \frac{Ds}{2L}\gamma_{15} + \frac{4\pi}{\lambda}(t_s c + s)\gamma_3^{DP} + \frac{16\pi s}{\lambda}\mathbf{u} \tag{5.67}$$

Dies kann mit der Bedingung $|\partial\varphi'_{n0}| \ll \pi$ genutzt werden, um zu entscheiden, ob die gewählte Genauigkeit für ein gegebenes Setup ausreicht. Der kritische Term hier ist offensichtlich der letzte, welcher aus der Propagation resultiert. Daraus folgt, dass einfache Genauigkeit bei der Berechnung und Speicherung der (Inzell-)Position der Photonen für größere Simulationsgrößen nicht ausreicht. Auch ist es sehr wahrscheinlich, dass der Richtungsvektor für die Bewegung ($\underline{\boldsymbol{k'}}$) ebenfalls in höherer Genauigkeit gespeichert werden muss, da sonst z. B. die Bedingung des Einheitsvektors verletzt sein könnte, was zu einem additiven Fehler je Zeitschritt führt. Dies lässt sich durch einen Korrekturfaktor verbessern, der zusätzlich gespeichert wird, wodurch der Speicherverbrauch auf 16 Byte steigt, was aber immer noch eine Verbesserung gegenüber den 24 Byte für die Speicherung in doppelter Genauigkeit ist. Dagegen spricht ein erhöhter Aufwand bei der Konvertierung, Berechnung und Implementation, sodass dies vorerst nicht umgesetzt wurde.

6 Ergebnisse

In diesem Kapitel werden zuerst Tests mit einfachen Dichteverteilungen durchgeführt um die simulierten Intensitätsverteilungen mit analytischen Lösungen vergleichen zu können. Dabei wird besonders die Konvergenz betrachtet, also in wie weit sich die simulierte Lösung der analytischen annähert, da eine exakte Übereinstimmung nur mit unendlich vielen Partikeln möglich wäre. Als Metrik kommt die mittlere quadratische Abweichung, auch mittlerer quadratischer Fehler (MQF) oder MSE (engl. „Mean squared error“) genannt, zum Einsatz. Dessen Definition ist:

$$MQF(I_1, I_2) = \frac{1}{N} \sum_{n=1}^{N} (I_1[n] - I_2[n])^2 \tag{6.1}$$

Hierbei sind I_1 und I_2 (diskrete) Dichteverteilungen mit jeweils N Werten, wobei zweidimensionale Verteilungen gegebenenfalls linearisiert[28] werden. Soweit nicht anders angegeben, ist der MQF einer simulierten Verteilung I_{sim} immer bezogen auf die analytische Lösung I_{ana}, und demnach $MQF(I_{sim}) = MQF(I_{sim}, I_{ana})$.
Im zweiten Teil dieses Kapitels wird das Weak und Strong Scaling betrachtet und ich gehe kurz auf deren Bedeutung ein. Für die Tests auf dem *hypnos* Cluster des HZDR wurden K20X GPUs mit CUDA 7.5 und GCC 4.8.2 als Hostcompiler verwendet. Die MPI Bibliothek war OpenMPI 1.8.4 mit CUDA 7.5 Unterstützung. Auf dem *Taurus* des ZIH kamen K80 GPUs mit CUDA 7.5, GCC 4.8.0 und bullxMPI 1.2.8.4 zum Einsatz. Ausgewählte Rohdaten der Ergebnisse sind auf der beiliegenden DVD zu finden.

6.1 Tests

Als Standardtest während der Entwicklung diente ein Doppelspaltexperiment (basierend auf dem Youngschen Doppelspalt), da der Aufbau und die Lösung sehr einfach sind und dieses Experiment sehr bekannt ist, da es sehr häufig als Beispiel für den Wellencharakter des Lichts und auch von Teilchen verwendet wird. So beschreibt Walther und Walther [67, S. 92 ff.] die Interferenz von Photonen am Doppelspalt „mit sich selbst“ und in Feynman et al. [68, S. 1-5 f.] wird die Interferenzerscheinung von Elektronen hinter einem Doppelspalt beschrieben. Bemerkenswert hierbei ist, dass in der Praxis die Kenntnis des Weges, den ein Photon genommen hat, das Interferenzmuster zerstört. „Die Existenz der Interferenzstreifen beruht einzig auf der Unvorhersagbarkeit des genauen Weges des Photons von der Quelle zum Detektor.“ [67, S. 95] In der Simulation sind jedoch die genauen Wege der Photonen bekannt, wobei nur ein Modell verwendet wird und die Wege nur Instanzen (oder Samples) der Zufallsprozesse sind. Gerade deshalb bietet sich das Doppelspaltexperiment an, um auf einfache Weise zu überprüfen, ob die komplexen Vorgänge ausreichend gut abgebildet werden können.

[28] Für ein zweidimensionales I mit r Zeilen und c Spalten ist die linearisierte Verteilung $I'[n] = I[i][j]$ mit $n = i \cdot c + j$ und $N = r \cdot c$.

Zu bestimmen ist zuerst die analytische Lösung eines Doppelspalts mit einer Spaltbreite b und einem Abstand a zwischen den Mittelpunkten (bzw. Startpunkten) der Spalte, welcher von Licht mit der Wellenzahl k gleichmäßig beleuchtet wird. Über die Fouriertransformation erhält man für die Intensität den Ausdruck:

$$I(\theta) = \frac{1}{I(0)} \cos^2\left(\frac{ka}{2}\sin(\theta)\right) \mathrm{si}^2\left(\frac{kb}{2\pi}\sin(\theta)\right) \tag{6.2}$$

Die Funktion $\mathrm{si}(x)$ ist die Spaltfunktion, die als $\mathrm{si}(x) = \frac{\sin(x)}{x}$ definiert ist. Mit der Normierung ist es möglich, die Ergebnisse unabhängig von den Vorfaktoren zu vergleichen, was die folgende Betrachtung einfacher macht. In der Simulation wird der Doppelspalt in eine Ebene entlang der Propagationsrichtung mit einer Dicke von einer Zelle gesetzt, um Mehrfachstreuung zu vermeiden. Der Schnitt entlang dieser Ebene ist in Abb. 6.1 (nicht maßstäblich) dargestellt. Die Grenzen des Simulationsvolumens sind als

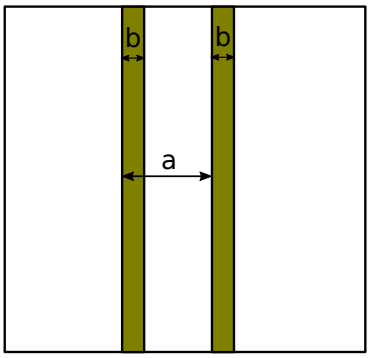


Abbildung 6.1: Schematische Darstellung des Doppelspalts (Sicht in Richtung des Detektors)

schwarzer Rahmen angedeutet während die farblich markierten Flächen Dichten mit konstantem Wert sind. Außerhalb dieser Regionen ist die Dichte konstant null. Diese augenscheinlich inverse Formulierung begründet sich darin, dass die Photonen an vorhandener Dichte gestreut werden und um demnach die Streuung von den Spalten ausgehend zu simulieren, müssen die Spalte eine hohe Elektronendichte haben. Es lässt sich zeigen, dass das Interferenzmuster für die inverse Dichteverteilung qualitativ gleich ist. Hier wurde diese Formulierung verwendet, da sie sich einfacher nachvollziehen lässt. Für die notwendige zweidimensionale Fouriertransformation lässt sich zeigen, dass die Dichteverteilung in eine Dimension (hier Winkel) der Gleichung (6.2) entspricht und für die andere Dimension außerhalb von null verschwindet. Dies kann auch sehr einfach mit dem Programm ImageJ [69] überprüft werden, das die Berechnung des Leistungsspektrums (Betragsquadrat der Fouriertransformierten) per Knopfdruck erledigt. Deshalb wird für die folgenden Diagramme jeweils nur der Schnitt entlang einer Achse auf dem Detektor gezeigt.

Die nach Gleichung (6.2) berechnete Intensitätsverteilung (normiert auf $I(0)$) ist in Abb. 6.2 dargestellt, wobei die Farben auch den Teilen der Gleichung entsprechen. Dazu wurde eine Spaltbreite b von 12 nm und ein Spaltabstand a von 32 nm verwendet. Das Licht hat eine Wellenlänge λ von 0,1 nm und der Detektor steht im Abstand von 5 m zum Ende der Simulation mit einer Größe von rund[29] 12 µm pro Zelle und 1024 Zellen Ausdehnung pro Richtung. In der Simulation wurde eine Zellgröße Δs von 4 nm bei einer Abmessung von 128×128 Zellen als Querschnitt und 256 Zellen in Propagationsrichtung des

[29] Die Detektorzellen werden zum Rand hin größer, da sie proportional zum mittleren Beobachtungswinkel sind (s. Unterabschnitt 5.3.3)

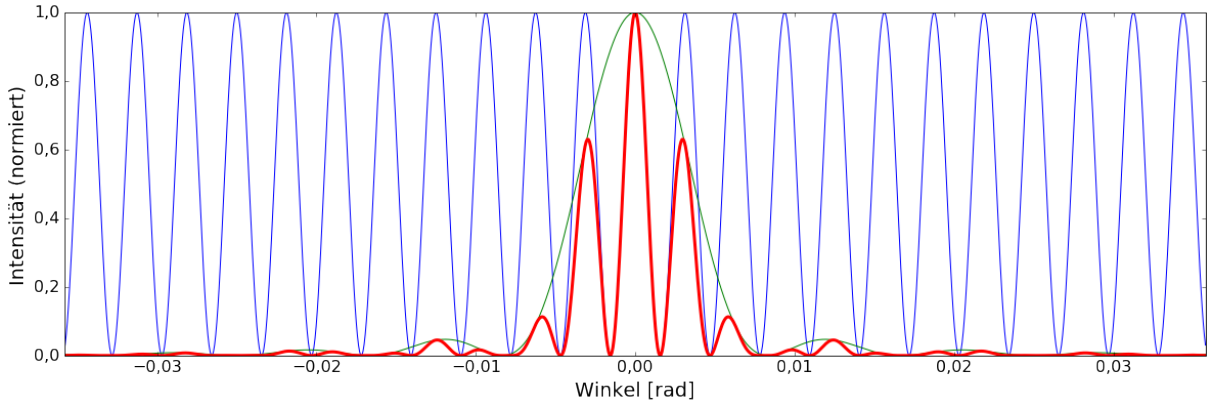


Abbildung 6.2: rot: Analytische Lösung der Intensität hinter einem Doppelspalt; grün: Streubild eines Einzelspalts; blau: Vom Abstand a abhängiger Beitrag

Lichtes verwendet. Zur Varianzreduktion wurde die Streuwahrscheinlichkeit innerhalb des Doppelspalts auf eins gesetzt (es wird immer gestreut) und der maximal mögliche Ablenkungswinkel auf $0{,}1$ rad festgelegt, da bei dem gegebenen Aufbau bereits Streuwinkel oberhalb von rund $0{,}4$ rad nicht mehr erkannt werden. Dadurch konvergiert die Lösung schneller bei nur unwesentlicher Änderung des Ergebnisses. Der Röntgenlaser wurde mit 10 Photonen je Zeitschritt und Zelle (in der Eintrittsebene) simuliert, die zufällig platziert wurden. Im Interferenzmuster zeigen sich bereits nach 10 Tsd. Zeitschritte das Haupt-

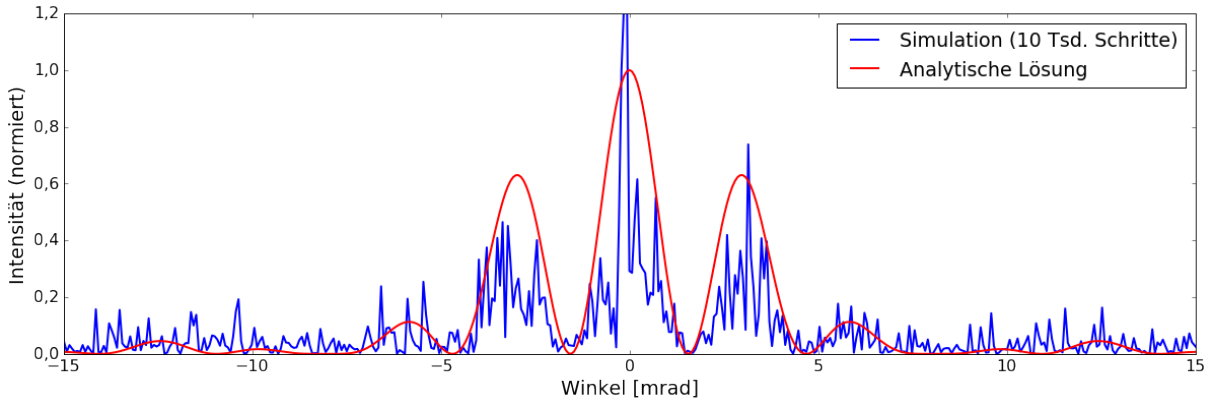


Abbildung 6.3: Simulierte Intensität auf dem Detektor nach 10 Tsd. Zeitschritten (doppelte Genauigkeit) im Vergleich mit der theoretischen Lösung (Ausschnitt)

und die ersten Nebenmaxima, wie aus Abb. 6.3 ersichtlich wird. Zum Vergleich wurde die simulierte Intensität so skaliert, dass deren Mittelwert mit dem der theoretischen Lösung (nach Gleichung (6.2)) übereinstimmt. Auch muss beachtet werden, dass die Simulation einen sogenannten „Beamstop“ implementiert, der nicht abgelenkte Partikel aus der Mitte des Detektors entfernt, um einen sinnvollen Kontrast zu erreichen. Andernfalls wäre an dieser Stelle ein extremer Peak der Intensität, der nicht durch das Streumuster verursacht wurde und damit für die Betrachtung nicht weiter relevant ist. Natürlich ist auch dies deaktivierbar. In den Diagrammen wurden der Darstellung halber diese null-Werte durch interpolierte Werte ersetzt. Dies betrifft hier nur die mittleren zwei Zellen und deren Werte werden auch für die Mittelwertbildung und Fehlerbestimmung nicht verwendet. In Abb. 6.4 ist die Intensitätsverteil-

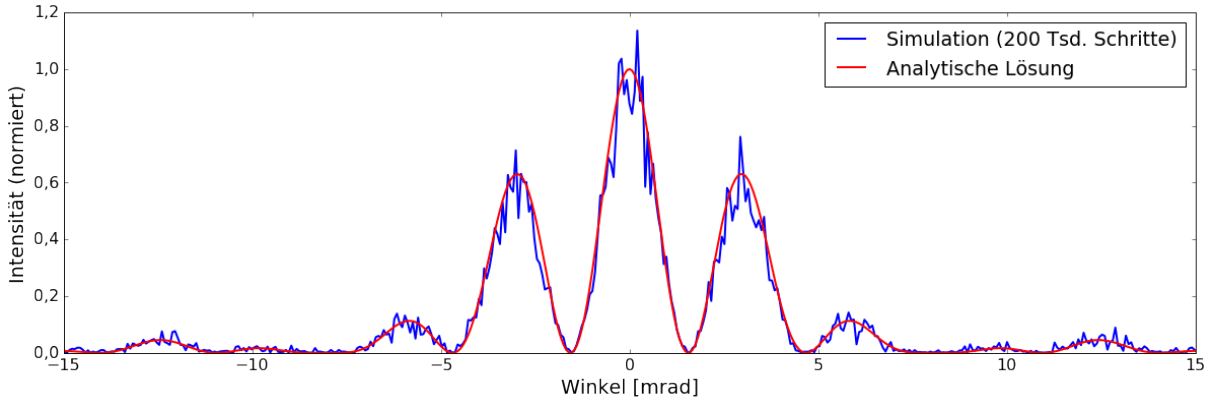


Abbildung 6.4: Simulierte Intensität auf dem Detektor nach 200 Tsd. Zeitschritten (doppelte Genauigkeit) im Vergleich mit der theoretischen Lösung (Ausschnitt)

lung nach 200 Tsd. Zeitschritten dargestellt. Wie man sieht, stimmt sie mit der berechneten Lösung sehr gut überein. Abb. 6.5 zeigt die Verteilung bei der Verwendung von einfacher Genauigkeit. Auch die-

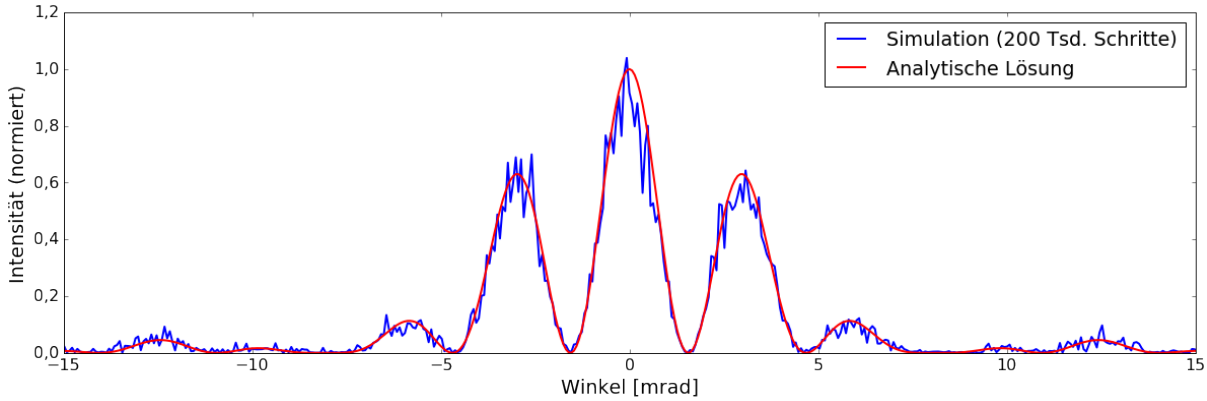
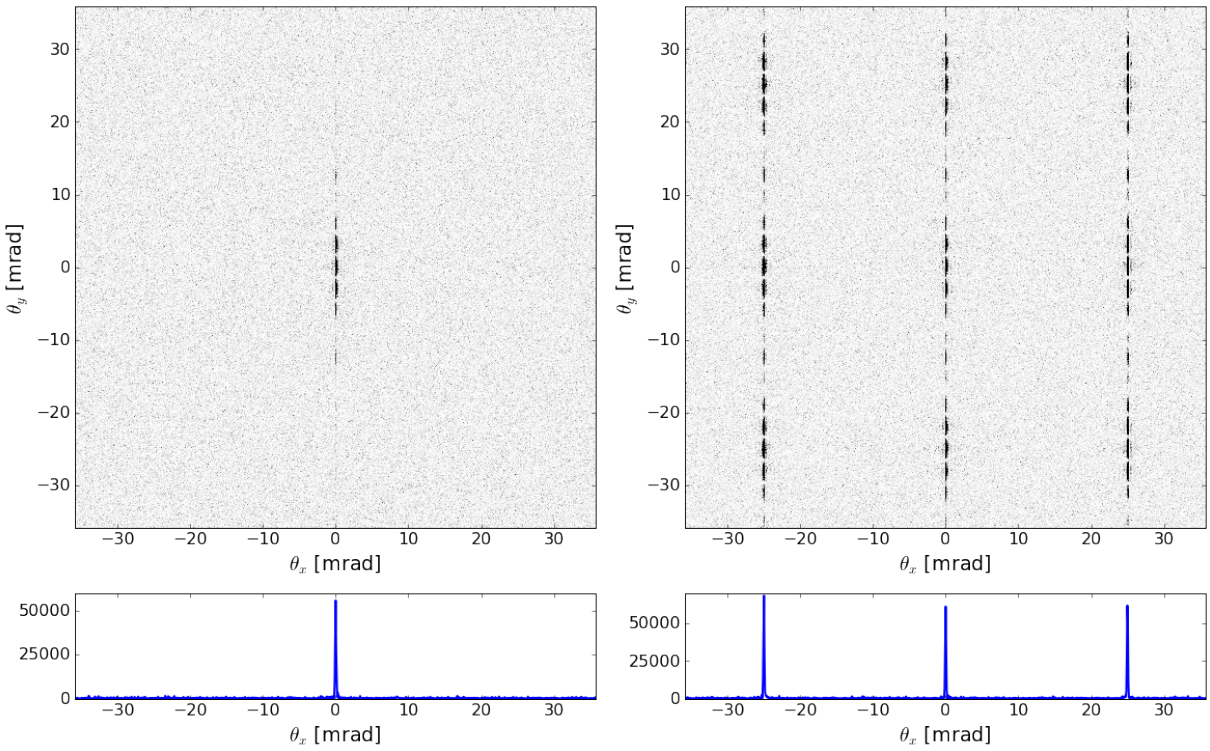


Abbildung 6.5: Simulierte Intensität auf dem Detektor nach 200 Tsd. Zeitschritten (einfache Genauigkeit) im Vergleich mit der theoretischen Lösung (Ausschnitt)

se unterscheidet sich nicht wesentlich von der berechneten Intensität, wodurch gezeigt ist, dass diese Genauigkeit zumindest für dieses Beispiel ausreicht. Dies zeigt sich auch beim Vergleich der mittleren quadratischen Fehler (MQF), die bei 10 Tsd. Zeitschritten $1{,}219 \cdot 10^{-2}$ bzw. $1{,}224 \cdot 10^{-2}$ und nach 200 Tsd. Zeitschritten $6{,}193 \cdot 10^{-4}$ bzw. $6{,}309 \cdot 10^{-4}$ für doppelte und einfache Genauigkeit betragen. Das vollständige Streubild ist in Abb. 6.6a mit überhöhtem Kontrast dargestellt. Deutlich zu erkennen ist das angesprochene Interferenzmuster bei $\theta_x = 0$, welches, wie erwartet, bereits in kleiner Entfernung davon praktisch nicht mehr vorhanden ist (s. Querschnitt darunter). Das Rauschen wird durch geringe numerische Ungenauigkeiten und die begrenzte Photonenzahl erzeugt, ist hier jedoch deutlich kleiner als das Signal und damit unproblematisch. Setzt man die Photonen nicht zufällig verteilt in die Simulation, sondern an eine konstante Position innerhalb der Zellen, so dass sie einen festen Abstand zueinander haben, erzeugt dies eine zyklische Wiederholung des Interferenzmusters wie in Abb. 6.6b zu sehen ist. Auch dies entspricht der Erwartung und lässt sich über die Fouriertransformation herleiten: Die in der Theorie kontinuierliche Photonenverteilung wird hier diskretisiert, was einer Multiplikation mit



(a) Zufällige Platzierung der Photonen

(b) Feste Startpositionen

Abbildung 6.6: Simulierte Intensität auf dem Detektor nach 200 Tsd. Zeitschritten und Schnitt entlang $\theta_y \approx 0$ (Intensität in arb. Einheiten)

einer Dirac-Impulsfolge (auch Dirac-Kamm genannt [6, S. 75]) entspricht, deren Fouriertransformierte ebenfalls eine Dirac-Impulsfolge ist (vgl. [21, S. 98]). Gemäß dem Faltungssatz (s. Gleichung (2.5) auf Seite 10) muss die Fouriertransformierte des Doppelspalts mit dem Dirac-Kamm gefaltet werden, wodurch sich die periodische Wiederholung ergibt (vgl. [21, S. 98 f.]). Aus der Periode g lässt sich über $g = \frac{2\pi}{d}$ (vgl. [6, S. 75]) der Abstand der Photonen aus dem der Streumuster ermitteln, bzw. umgekehrt. In diesem Fall ergibt sich mit der Periode $g = k\sin(\Delta\theta)$ (über $q = k\sin(\theta)$, s. Abschnitt 3.2), der Winkeldifferenz $\Delta\theta$ von rund $25\,\mu\text{rad}$ (aus dem Bild) und obiger Wellenlänge ein Abstand d von $4{,}00\,\text{nm}$, was dem Zellabstand Δs entspricht. Dieses Beispiel dient einerseits der Verifizierung der Simulation, andererseits zeigt es auch, dass eine bestimmte Auflösung für die Position (insbesondere die Anfangsposition) für die Photonen benötigt wird, um Überlagerungen mit sekundären Streubildern zu vermeiden. Zu bemerken sei hier der enge Zusammenhang mit dem Nyquist-Shannon-Theorem (s. [21, S. 103]), da um Überlagerungen auf dem Detektor mit einem maximalen Beobachtungswinkel θ_{max} zu vermeiden, die Photonen mit einem Abstand d_{max} von höchstens $d_{max} = \frac{\lambda}{\sin(\theta_{max})}$ aufgelöst werden müssen. Mit der zufälligen Platzierung wird für ausreichend viele Photonen die notwendige Auflösung erreicht, da der mittlere Abstand bei einfacher Genauigkeit $2^{-24} \cdot \Delta s$ beträgt[30]. Das reicht für alle praktischen Fälle aus, da $\Delta s < 1\,\text{mm}$ angenommen werden kann, womit alle Winkel ohne Überlagerungen abbildbar sind. Im Folgenden wird deshalb ausschließlich die zufällige Platzierung der Photonen betrachtet.

Zu erwähnen ist hier, dass die Intensitätsverteilung in x-Richtung nicht ideal steil abfällt, und außerhalb von (exakt) $\theta_x = 0$ verschwindet, sondern das Interferenzmuster eine geringe Ausdehnung in diese Rich-

[30] Für diese Gleitkommazahlen im Bereich von null bis eins (Wertebereich der Inzell-Position) ist der maximale Abstand 2^{-24} bei Zahlen nahe eins.

tung aufweist. Das liegt hauptsächlich an der endlichen Ausdehnung des Doppelspalts und kann ebenfalls über die Fouriertransformation erklärt werden. Ein endlich ausgedehnter Doppelspalt lässt sich aus einem idealen (unendlichen) Doppelspalt konstruieren, indem dessen Beschreibungsfunktion mit einer Rechteckfunktion multipliziert wird, wodurch sich die Intensität aus der Faltung der Doppelspaltintensität mit einer si-Funktion ergibt[31]. Die Breite des mittleren Peaks der si-Funktion ist dabei umgekehrt proportional zur Breite der Rechteckfunktion und geht im Grenzfall in den unendlich schmalen Dirac-Impuls über. Ausdehnung des Streumusters in x-Richtung auf Abb. 6.6 ist demzufolge durch die si-Funktion verursacht, deren Ursprung wiederum die Multiplikation mit der Rechteckfunktion mit einer großen, aber endlichen Ausdehnung ist. Demnach verhält sich die Simulation auch hier wie erwartet.
Es folgt nun die Konvergenz der Intensitätsverteilung gegen die analytische Lösung. Sinnvollerweise sollte die Betrachtung über der Anzahl der Photonen erfolgen, da diese den Beitrag zum Ergebnis leisten. Für die hier gegebene statische Elektronen- und Photonendichte soll jedoch die Konvergenz über der Anzahl der Zeitschritte für verschiedene Mengen an Photonen, die pro Zelle erzeugt werden, untersucht werden. Die Gesamtzahl der Partikel ergibt sich demnach aus der Fläche, auf der die Photonen erzeugt werden und der simulierten Zeit. Aus dieser Betrachtung kann ebenfalls eine Aussage über die Konvergenz mit wachsender Anzahl der Photonen getroffen werden, gleichzeitig erlaubt es aber auch die direkte Beurteilung wie viele Partikel in einer (Super-)Zelle sind und welcher Rechenaufwand damit pro GPU-Thread bzw. Block verbunden ist. Abb. 6.7 zeigt die Konvergenz im Bereich von null bis 200 Tsd. Zeitschritten bei Verwendung doppelter Genauigkeit und ausgewählte Werte sind in Tab. 6.1 zu finden. Auch hier wurde jedes Detektor-Bild einzeln so skaliert, dass der Mittelwert mit dem der ana-

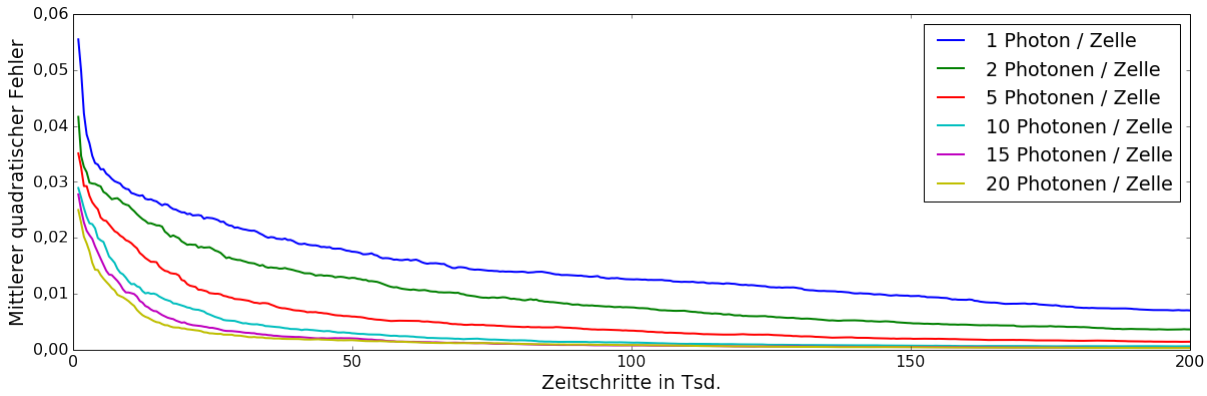


Abbildung 6.7: Mittlerer quadratischer Fehler über der Photonenzahl pro Zelle

lytischen Lösung übereinstimmt und nur das eigentliche Muster bei $\theta_x = 0$ verwendet. Die Konvergenz des Fehlers gegen null ist gut zu erkennen und auch, dass er von der Gesamtzahl der Partikel abhängt, sodass bei einer Verdopplung der Photonenzahl pro Zelle nur noch etwa halb so viele Zeitschritte notwendig sind (s. Tab. 6.1). Unterstützt wird diese Beobachtung durch die ermittelte Schätzfunktion, die für den Zeitschritt t_s und die Anzahl N_c an Photonen pro Zelle den MQF für dieses Beispiel abschätzt. Dazu wurden Funktionen der Form $f(x) = Ax^B$ und $g(x) = \frac{A}{1+Bx}$ untersucht, wobei der Parameter x als $x = t_s$ oder $x' = t_s \cdot N_c$ angesetzt wurde. Die Form $f(x)$ erlaubt dabei die Darstellung von Abhängigkeiten wie $\frac{1}{\sqrt{x}}$ und $\frac{1}{x}$, während die zweite Form $g(x)$ eine Verzögerung berücksichtigen kann, was z. B.

[31] Die Fouriertransformierte der Rechteckfunktion ($\mathrm{rect}(x) = 1$, wenn $|x| \leq \frac{1}{2}$, sonst 0) ist die si-Funktion (vgl. [6, S. 247]).

Tabelle 6.1: Mittlerer quadratischer Fehler für das Doppelspaltexperiment

	Photonen / Zelle					
Zeitschritt	1	2	5	10	15	20
25000	0,023 50	0,018 04	0,010 07	0,006 07	0,003 91	0,002 91
50000	0,017 52	0,012 85	0,005 92	0,002 93	0,002 03	0,001 71
75000	0,014 11	0,009 33	0,004 33	0,001 80	0,001 10	0,001 15
100000	0,012 56	0,007 57	0,003 41	0,001 28	0,000 78	0,000 83
125000	0,011 15	0,005 91	0,002 66	0,000 87	0,000 57	0,000 54
150000	0,009 58	0,004 75	0,001 98	0,000 72	0,000 47	0,000 45
175000	0,007 93	0,004 21	0,001 67	0,000 61	0,000 38	0,000 37
200000	0,007 02	0,003 63	0,001 42	0,000 62	0,000 32	0,000 30

in Commons et al. [70, S. 58] beschrieben wird. Zur Ermittlung der Parameter kam die Python Funktion `curve_fit` [71] zum Einsatz, die auf der Methode der kleinsten Quadrate beruht und auch Varianzen der Parameter zurück gibt. Die Funktion wurde auf jedes N_c einzeln angewandt und die Parameter anschließend verglichen, um einen möglichst allgemeinen Ausdruck für alle N_c und t_s zu erhalten. Lediglich mit der Form $g(x')$ konnten Parameter ermittelt werden, die über alle Werte aus Tab. 6.1 gute Näherungswerte lieferte, sodass der MQF mit

$$g(t_s, N_c) = \frac{3{,}5 \cdot 10^{-2}}{1 + 2{,}1 \cdot 10^{-5} N_c t_s} \tag{6.3}$$

genähert werden kann, was für zwei Beispiele in Abb. 6.8 dargestellt ist.

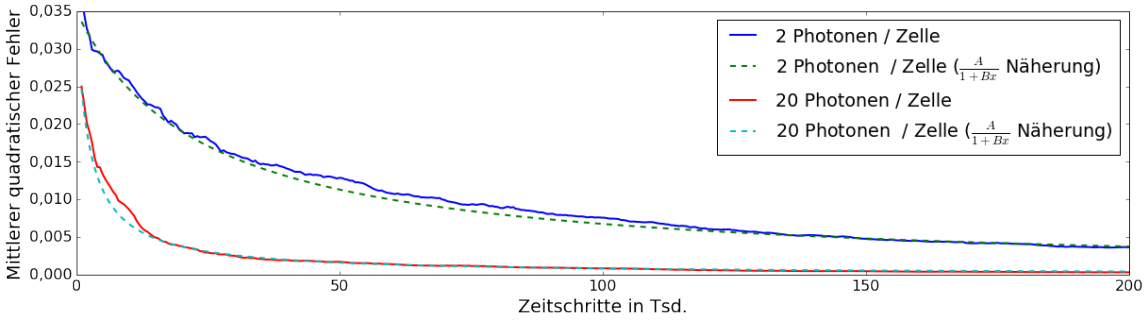


Abbildung 6.8: MQF und dessen Approximation der Intensität hinter einem Doppelspalt

Damit ist gezeigt, dass die Simulation tatsächlich gegen die analytische Lösung konvergiert, und die Anzahl der Photonen linear (mit Verzögerung) in den MQF eingeht, woraus auch abgeleitet werden kann, dass die mittlere absolute Abweichung näherungsweise mit der Wurzel der Gesamtzahl der Partikel abfällt.

Als nächstes folgen die Ergebnisse bei der Verwendung von einfacher Genauigkeit. In Abschnitt 5.3 wurden Grenzen hergeleitet bei denen diese Genauigkeit verwendet werden kann, ohne dass deutliche Fehler in der Intensitätsverteilung auftreten. Die kritischste Bedingung war die Gesamtgröße der Simulation (s. Unterabschnitt 5.3.2). Deshalb wurde obiger Doppelspalt mit 9984×224^2 Zellen[32], fünf

[32] Die Größen ergeben sich hauptsächlich aus dem maximal verfügbaren Grafikspeicher, müssen aber auch Vielfache der GPU-Aufteilung und Superzellgrößen sein.

Photonen je Zelle und 200 Tsd. Zeitschritten auf 64 GPUs ausgeführt, wobei je einmal einfache und doppelte Genauigkeit verwendet wurde. Der Doppelspalt war somit fast doppelt so lang. Die anderen Simulationsparameter wurden nicht geändert, sodass sich das Streubild nur unwesentlich [33] ändern sollte. Der Fehler wurde ebenfalls als MQF berechnet und für beide Simulationen, zusammen mit der Abschät-

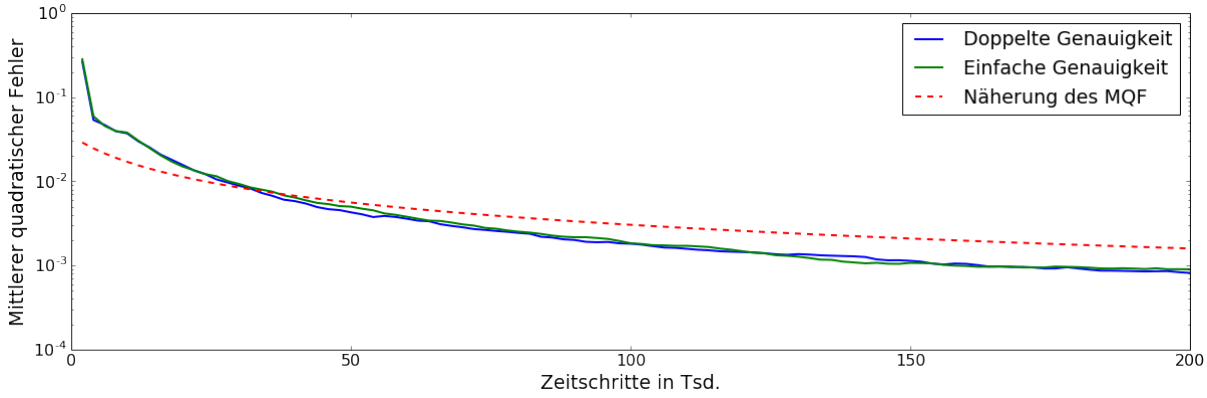


Abbildung 6.9: MQF der Intensität hinter einem Doppelspalt; Größe in Propagationsrichtung: $\sim 40\,\mu m$

zung nach Gleichung (6.3), in Abb. 6.9 als halblogarithmischer Plot dargestellt, um die geringen Abweichungen zu verdeutlichen. Die Konvergenz entspricht der Erwartung, auch wenn die Schätzfunktion hier nicht steil genug ist, was durch die halblogarithmische Darstellung verstärkt wird. Beim Vergleich der zwei Simulation lassen sich selbst bei dieser Darstellung keine wesentlichen Unterschiede erkennen. Das wirft die Frage auf, warum das trotz einer Simulationsgröße von $40\,\mu m$ (doppelt so groß, wie die in Unterabschnitt 5.3.2 ermittelte Grenze bei der Verwendung von einfacher Genauigkeit) möglich ist. Die Antwort liegt in den Beschränkungen des verwendeten Modells, denn die ermittelten Fehlergrenzen sind maximale Fehler. So wird u. a nicht berücksichtigt, dass sich Fehler gegenseitig auslöschen können. Hier ist es nicht nur möglich, dass sich Fehler bei der Positionsberechnung eines Photons im Verlauf der Simulation gegenseitig auslöschen, sondern es ist auch möglich, dass die jeweiligen Abweichungen in der Position zweier interferierender Partikel (annähernd) gleich sind, wodurch sich letztendlich der selbe Phasenunterschied ergibt, wie bei exakter Berechnung. Dies ist insbesondere bei Einfachstreuung durch die lange Propagation in eine Richtung möglich, da hier der Fehler i. d. R. linear wächst (s. Abb. 5.5) und die Photonen annähernd den gleichen Weg (längenmäßig) durch das Volumen haben. Es lässt sich durch die Analyse und den Tests der numerischen Genauigkeit demnach nur sagen, dass der Fehler für bestimmte Eingabewerte sicher kleiner ist, als ein Grenzwert, nicht jedoch, dass er für andere Eingabewerte größer ist. Nach der Faustregel aus Abschnitt 5.2 auf Seite 54 ist auch zu erwarten, dass deutlich größere Volumina möglich sind, solange viele Zellen vorhanden sind, demnach die Zellgröße ausreichend klein ist. Das folgt direkt aus der Anwendung der Faustregel, da der Fehler bei der Propagation von der Anzahl N_s der Zellen (indirekt über die Zeitschritte) abhängt und damit ein Faktor von $O(N_s)$ auf die maximale Simulationsgröße aufgerechnet werden kann.

Ein weitere Erkenntnis hat sich aus Tests mit größeren Simulationszellen ergeben, wobei hier ebenfalls der Doppelspalt und eine Größe von 256×128^2 Zellen verwendet wurde. Durch die größeren Strukturen sind die Streumuster entsprechend kleiner, d. h. es müssen kleinere Winkel beobachtet werden, was

[33] Die Beobachtungswinkel sind durch den größeren Abstand etwas größer, wodurch das Streubild größer erscheint.

durch eine Verringerung der Detektorzellgröße erreicht werden kann. Speziell wurde $\Delta s = 0{,}1772\,\mu\text{m}$, $\Delta D = 30\,\mu\text{m}$ und $L = 10\,\text{m}$ gewählt, wodurch sich ein Winkel $\Delta D'$ von $3\,\mu\text{rad}$ pro Zelle ergibt. Da jedoch der maximale Ablenkungswinkel weiterhin bei $0{,}1\,\text{rad}$ lag, kamen trotz gleicher Laufzeit (in Zeitschritten) und der selben Anzahl erzeugter Photonen nur sehr wenige davon auf dem Detektor an, wodurch sich ein Verhalten ähnlich eines Photonenzählers ergab. Außerdem waren deutliche Ringe von Partikeln in der Nähe der Detektormitte zu erkennen und in einem Kreis direkt um die Mitte befanden sich gar keine Partikel (abgesehen von den, die durch den Beamstop entfernt wurden), was daran erkennbar war, dass die Intensitäten exakt null waren. Wie die vorherigen Versuche gezeigt haben, ist selbst in Bereichen, in denen analytisch keine Intensität zu erwarten ist, immer noch ein Rauschen zu erkennen, was hier nicht der Fall ist. In Abb. 6.10 ist dieses Muster als Ausschnitt um die Detektormitte

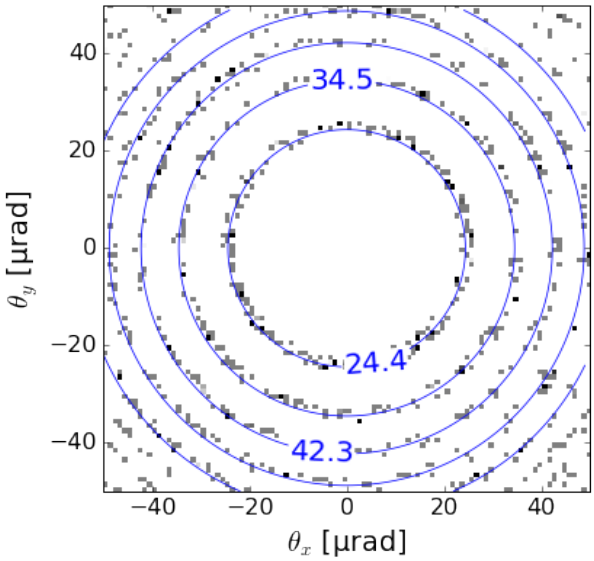


Abbildung 6.10: Durch die Auflösung der Zufallszahlen erzeugte, ringförmige Muster auf dem Detektor (qualitativ); blau: Kennzeichnung der Winkel $\theta_r(n)$ für $n = 1 \ldots 5$

zu sehen, wobei die Ringe zusätzlich markiert und mit den zugehörigen Winkeln bezeichnet wurden. Weil die Ringe auch bei Verwendung doppelter Genauigkeit auftraten, ist es unwahrscheinlich, dass sie von Fehlern während der Propagation verursacht werden, da die Partikel auch noch mit einem Winkel von $0{,}119\,\mu\text{rad}$ bei einfacher Genauigkeit (s. Unterabschnitt 5.3.2) bewegt werden könnten. Als Ursache wurde die Erzeugung der Streuwinkel ausgemacht, die auf Zufallszahlen zwischen null und eins basiert, welche wiederum aus Gründen der Performance in einfacher Genauigkeit ermittelt werden, was für alle bisherigen Anwendungsfälle ausgereicht hat. Bei der Analyse der verwendeten Gleichung (4.2) zur Berechnung des zufälligen Streuwinkels θ_r wird ersichtlich, dass kleine Winkel für große Zufallszahlen (nahe Eins) erreicht werden. Der Abstand benachbarter Gleitkommawerte in dem Bereich beträgt für einfache Genauigkeit 2^{-24}, sodass die ersten n möglichen Winkel durch Einsetzen von $r = 1 - n2^{-24}$ ($n \in \mathbb{N}; 0 < n < 2^{24}$) in Gleichung (4.2) bestimmt werden können. Durch triviale Umformung erhält man

$$\theta_r(n) = \arccos\left(1 - n \cdot 2^{-24}(1 - \cos(\theta_{max}))\right) \tag{6.4}$$

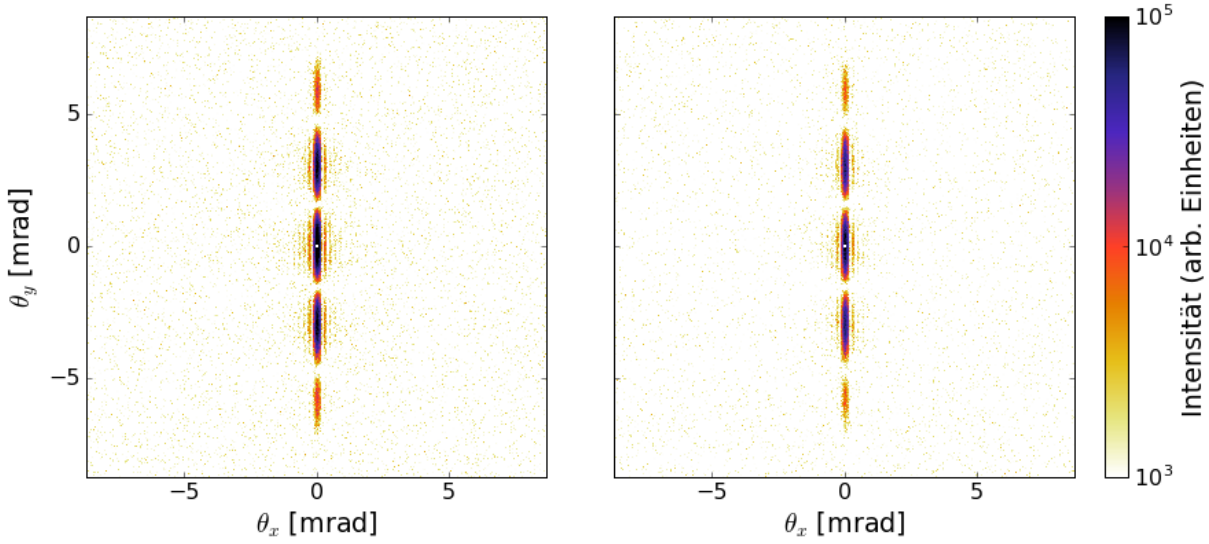
was sich unter Nutzung von $1 - \cos(x) = 2\sin^2(\frac{x}{2})$ und der Approximation von $\arccos(1 - x) \approx \sqrt{2x}$

für kleine x vereinfachen lässt:

$$\theta_r(n) \approx \frac{\sqrt{n}}{2^{11}} \sin(\frac{\theta_{max}}{2}) \tag{6.5}$$

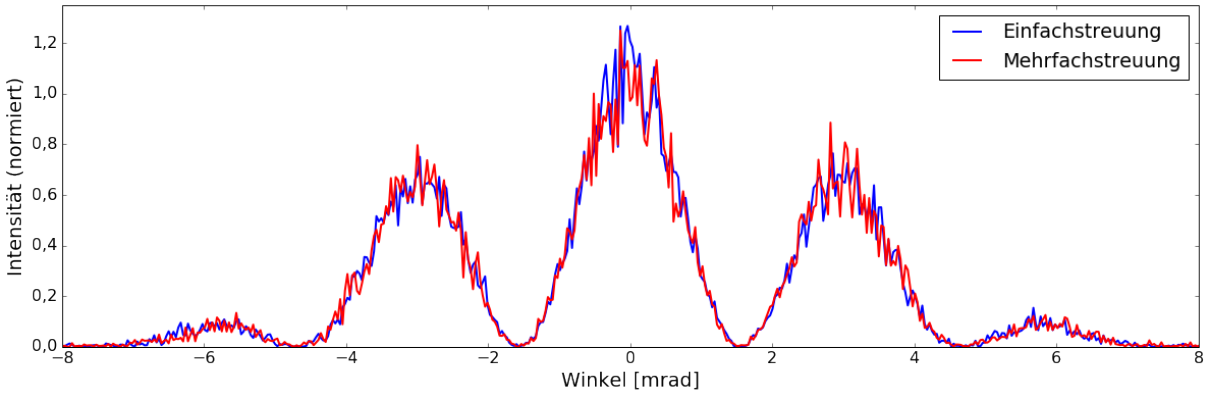
Diese Gleichung wurde auch verwendet, um die blauen Markierungen in Abb. 6.10 für $n = 1 \ldots 5$ zu bestimmen. Die Übereinstimmung mit den kreisförmigen Photonenverteilungen zeigt, dass diese Diskretisierung des Streuwinkels die Ursache für die Muster sind. Das dennoch einige Photonen neben den inneren Kreisen liegen, erklärt sich durch die Ausdehnung des Simulationsvolumens, da die Winkel auf die Mitte der Simulation bezogen sind, und Photonen am Rand demnach etwas neben denen aus der Mitte landen können. Auch können numerische Ungenauigkeiten, besonders bei der Berechnung der Winkelfunktionen eine Rolle spielen. Da in der Praxis die aufzulösenden Strukturen kleiner sind, wodurch größere Winkel betrachtet werden müssen ist die begrenzte Auflösung des Zufallszahlengenerators hier kein Problem. Jedoch ist zu beachten, dass bei der Verwendung von einfacher Genauigkeit bei der Berechnung des Arguments vom $\arccos$ der maximale Wert $1 - 2^{-24}$ ist, wodurch sich ein minimaler Winkel von rund $345{,}3\,\mu$rad ergibt, der größer ist, als der mit Gleichung (6.5) berechnete Wert von $\theta_r(1) \approx 24{,}4\,\mu$rad, da bei der Herleitung von $\theta_r(n)$ unbegrenzte Genauigkeit angenommen wurde. In guter Näherung gilt für kleine n bei einfacher Genauigkeit $\theta_{r,SP}(n) \approx \sqrt{n}2^{-11}$, d. h. die Winkelauflösung für kleine Winkel kann hier nicht durch θ_{max} beeinflusst werden. Für die hier vorgestellten Tests wurde das Argument und der $\arccos$ in doppelter Genauigkeit berechnet und anschließend in die Genauigkeit der Simulation gerundet, unabhängig von der sonst verwendeten Einstellung, was jedoch nur einen unwesentlichen Teil der Rechenzeit ausmacht und auch keinen Grafikspeicher verbraucht. Möchte man kleinere Winkel auflösen, muss die Zufallszahl r in doppelter Genauigkeit bestimmt werden. Die Basis des Zufallszahlengenerators liefert einen 32 bit Wert, wodurch die Abschätzung von Gleichung (6.5) auf $\theta_{r,DP}(n) \approx \frac{\sqrt{n}}{2^{15}} \sin(\frac{\theta_{max}}{2})$ verbessert werden kann, ohne die Performance wesentlich zu beeinflussen. Noch weiter erhöhen lässt sich die Auflösung nur durch Generierung von zwei 32 bit Zufallswerten, wodurch der Divisor von 2^{15} auf $2^{25{,}5}$ steigt. Es ist fraglich, ob so eine hohe Auflösung tatsächlich notwendig ist, da gerade bei einem Streuwinkel über den ganzen Raum ($\theta_{max} = \pi$) eine extrem hohe Anzahl an Photonen gebraucht werden, wenn Winkelbereiche dieser Größenordnung abzubilden sind.

Abschließend wurden auch Versuche zur Mehrfachstreuung durchgeführt. Dazu wurde der bisher verwendete Doppelspalt entlang der Propagationsrichtung über das komplette Volumen ausgedehnt, wobei 512×128^2 Zellen, 70 Partikel je Zelle und 500 Tsd. Zeitschritte simuliert wurden. Die Dichte auf dem Spalt wurde so gewählt, dass $0{,}1\,\%$ der Partikel in einer Zelle gestreut wurden. Zum Vergleich von Einfach- zu Mehrfachstreuung wurde ein Schalter implementiert, durch den bereits gestreute Partikel nicht noch einmal gestreut werden. Die Intensität bei $\theta_x = 0$ ist in Abb. 6.11c (mit Normierung wie bisher) und die zweidimensionale Intensitätsverteilung in den Abbildungen 6.11a und 6.11a gezeigt. Bis auf ein immer noch deutliches Rauschen ist hier qualitativ kein Unterschied zwischen Einfach- und Mehrfachstreuung festzustellen, quantitativ ist die Intensität bei Mehrfachstreuung etwa $40\,\%$ niedriger, was darauf hindeutet, dass mehr Partikel so stark abgelenkt werden, dass sie nicht mehr auf dem Detektor landen. Als Ursache, dass nicht mehr Unterschiede erkennen sind, lässt sich hier nur vermuten, dass der Effekt durch die Mehrfachstreuung bei diesem Experiment zu gering ist, um deutlich sichtbar zu werden, oder er durch das Rauschen überlagert wird. Letzteres lässt sich eventuell mit noch mehr simulierten Photonen lösen, was jedoch nicht mehr getestet werden konnte. Trotzdem kann die geringere Intensität als Resultat eines Effekts bei Mehrfachstreuung angesehen werden, der sich wie ein höherer Streuquerschnitt auswirkt.



(a) Nur Einfachstreuung

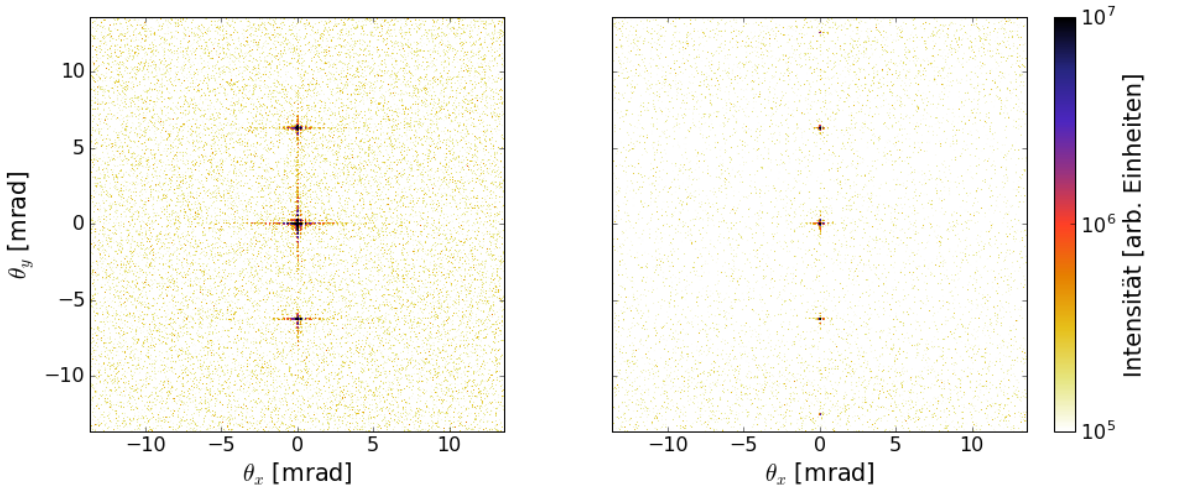
(b) Mit Mehrfachstreuung



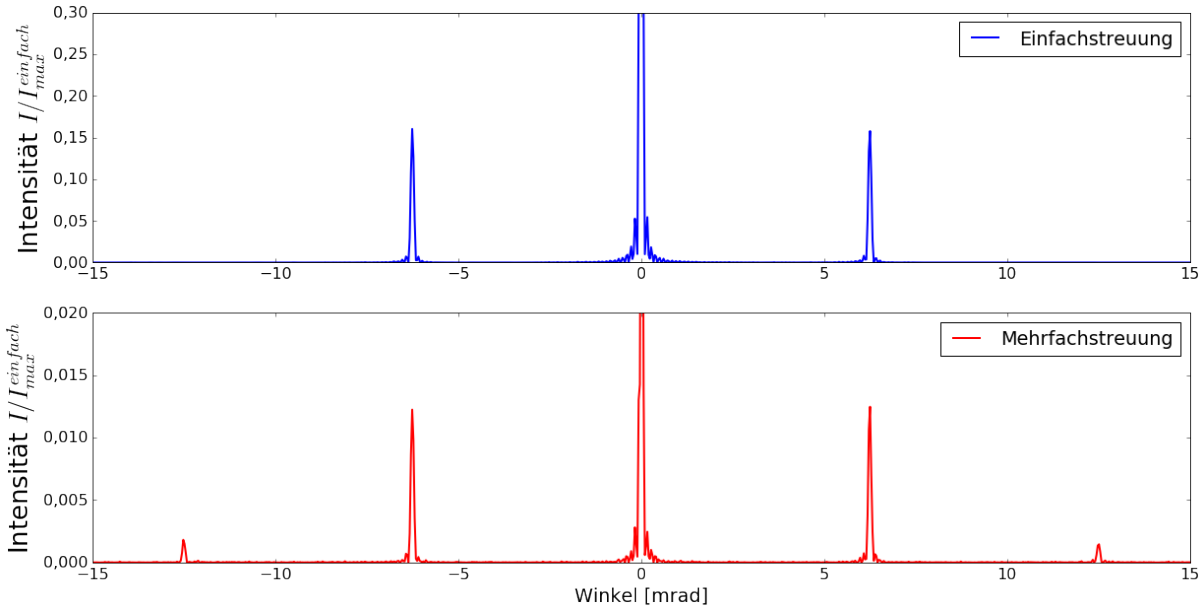
(c) Intensität bei $\theta_x = 0$

Abbildung 6.11: Simulierte Intensität auf dem Detektor bei einem ausgedehnten Doppelspalt

Ein weiteres Beispiel, in dem die Mehrfachstreuung und auch der Vorteil dieses Algorithmus' deutlich wird, ist die Simulation zweier versetzter Gitter, deren Ergebnis in Abb. 6.12 auf Seite 85 veranschaulicht wird. Die Dichteverteilung wurde so gewählt, dass 8 nm breite Streifen von konstanter Dichte von ebenso breiten Streifen leeren Raumes (ohne Dichte) gefolgt werden. Diese Verteilung erstreckt sich über das halbe Volumen in Propagationsrichtung und wird anschließend um 8 nm versetzt bis zum Ende des Volumens fortgesetzt. Bei der zweidimensionalen Fouriertransformation betrachtet man die Projektion dieser Dichte in Propagationsrichtung, was hier eine konstante Fläche wäre. Dafür erhält man nur eine Intensität in der Mitte des Detektors, auf dem Rest wäre sie verschwindend gering. In der Simulation ähnelt die Intensitätsverteilung für die Einfachstreuung (s. Abb. 6.12a) der eines idealen Gitters. Bei der Mehrfachstreuung kommen zusätzliche Maxima hinzu, wie Abb. 6.12b zeigt. Da diese sehr schmal sind, ist auch hier der Schnitt entlang $\theta_x = 0$ in Abb. 6.12c gezeigt, wobei auf die maximale Intensität der Einfachstreuung normiert und der durch den Beamstop entfernte Bereich interpoliert wurde. Zu erkennen sind die zusätzlichen Peaks bei der Mehrfachstreuung und die um eine Größenordnung geringere Intensität.
In diesem Experiment wurde die Dichte so gewählt, dass 1 % der Partikel in einer Zelle gestreut werden, wodurch der Effekt der Mehrfachstreuung bereits sichtbar wird. Bei kleineren Werten ist der Einfluss der Mehrfachstreuung deutlich geringer. Daran sieht man, dass bei kleinen Streuwahrscheinlichkeiten und Volumina die Einfachstreuung eine gute Näherung ist.



(a) Nur Einfachstreuung (b) Mit Mehrfachstreuung

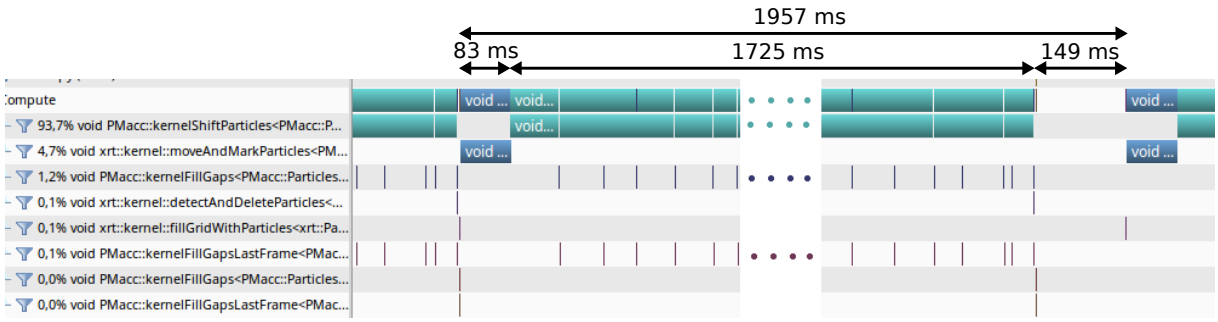


(c) Intensität bei $\theta_x = 0$

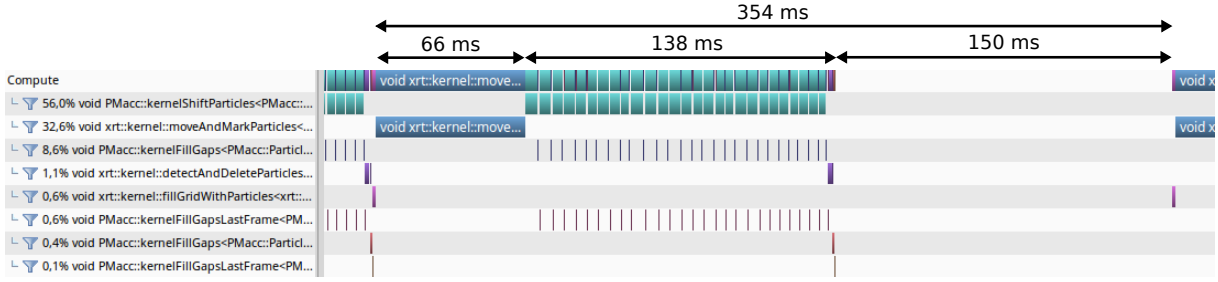
Abbildung 6.12: Simulierte Intensität auf dem Detektor bei zwei ausgedehnten, versetzten Gittern

6.2 Performance und Skalierung

In diesem Kapitel möchte ich zunächst ausgewählte Profile, die mit dem NVIDIA Profiler *nvprof* erstellt wurden, auswerten und dabei auf Performance-relevante Teile eingehen. Anschließend sollen die Laufzeiten bei der Verwendung von einfacher und doppelter Genauigkeit verglichen werden und zu guter Letzt gehe ich auf das Skalierungsverhalten der Simulation ein.
Zuerst soll die Zeitleiste für die Doppelspalt-Simulation in doppelter Genauigkeit in Abb. 6.13a betrachtet werden. Die Größe der Simulation beträgt $160 \times 128 \times 256$ Zellen, wobei der Doppelspalt entlang der 256 Zellen verläuft, sie hatte zehn Photonen je Zelle, und lief auf einer Grafikkarte um den Kommunikationsoverhead zwischen den GPUs vorerst nicht mit zu messen. Zu sehen sind Teile von drei Zeitschritten, die jeweils ca. bei dem zweiten Kernel von oben beginnen. Der mittlere Zeitschritte wurde zur besseren Darstellung gekürzt. Eindeutig dominant ist der Kernel `kernelShiftParticles`, der 93,7 % der Zeit ausmacht, gefolgt von `moveAndMarkParticles` mit 4,7 %. Ersterer ist aus der libPMacc Bibliothek und verschiebt die Partikel zwischen den Frames benachbarter Superzellen, die ihre aktuelle Superzelle verlassen haben. Um Race Conditions zu vermeiden muss dieser mit Blöcken im Doppel-Schachbrettmuster aufgerufen werden. Das bedeutet, dass pro Kernel-Aufruf nur jede dritte Superzelle in allen drei Dimensionen betrachtet wird, damit Partikel ohne weitere Synchronisation in eine benachbarte Zelle verschoben werden können. Ohne diese Aufteilung müsste man Zugriffe auf die Frames benachbarter Superzellen synchronisieren, was über Blockgrenzen hinweg schwierig und möglicherweise nicht performant ist. Um so alle Superzellen abzudecken müssen $3^3 = 27$ Kernel aufgerufen werden, was in Abb. 6.13a gekürzt wurde. Nach jedem solchen Kernel werden zwei weitere aufgerufen, welche die durch entfernte Partikel entstandenen Lücken in den Frames füllen, was einerseits den Speicherverbrauch durch Fragmentation senkt und andererseits Divergenzen innerhalb eines Blocks reduziert (s. Unterabschnitt 4.3.3).



(a) Doppelte Genauigkeit (gekürzt)



(b) Einfache Genauigkeit

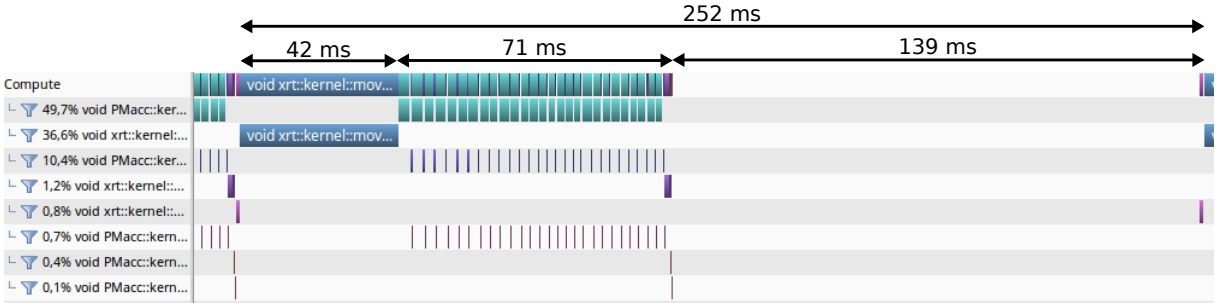
Abbildung 6.13: Zeitleiste der Doppelspalt-Simulation im Bereich der Ausgabe (~ 5 Mio. Zellen / GPU)

Der zweite Kernel ist für die Propagation der Partikel zuständig. Er scheint ausreichend effizient zu sein, da er deutlich weniger Zeit in Anspruch nimmt, als die Verschiebung der Partikel in libPMacc, was vor Allem dem neuen Zufallszahlengenerator zu verdanken ist, ohne den dieser Teil deutlich stärker ins Gewicht fallen würde. Die Funktionen zur Erzeugung der Partikel (`fillGridWithParticles`) und deren Erkennung auf dem Detektor (`detectAndDeleteParticles`) sind beide so schnell, dass deren Laufzeit vernachlässigt werden kann. Die Lücke auf der rechten Seite, nach dem zweiten Zeitschritt wird durch die Ausgabe des Detektors (inkl. globaler MPI-Kommunikation und Schreiben auf Festplatte) verursacht, deren Laufzeit der des Kernels zur Propagation (`moveAndMarkParticles`) ähnelt, jedoch nur selten aufgerufen wird. Die Häufigkeit wird durch den entsprechenden Laufzeitparameter bestimmt, und sollte für das Beispiel dieses Doppelspalts mindestens Tausend bis einige Tsd. Zeitschritte betragen, weil sich dazwischen die Intensität auf dem Detektor nicht wesentlich ändert (s. die Konvergenzbetrachtungen im vorangegangenen Abschnitt). Damit kann man auch diese Laufzeit vernachlässigen.

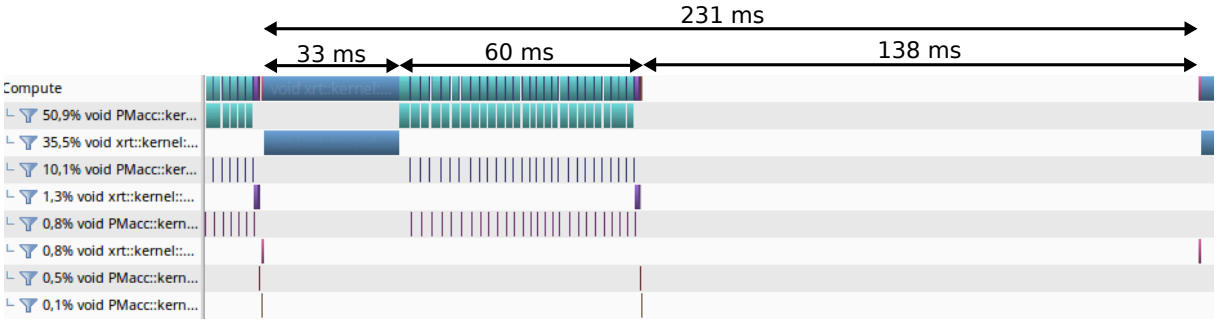
Effektiv ist der Algorithmus durch die Bandbreite und Latenz zum Grafikspeicher beschränkt, weil der Kernel `kernelShiftParticles` hauptsächlich Partikel von einem Frame in einen anderen schiebt. Zu erwarten ist demnach, dass die Geschwindigkeit durch die Verwendung von einfacher Genauigkeit, und damit nur noch rund halb so großen Partikeln, um den Faktor zwei steigt. Bevor ich die absoluten Zeiten analysiere, soll noch das Profil betrachtet werden, das Abb. 6.13b für den gleichen Bereich der Simulation mit der selben Konfiguration, jedoch in einfacher Genauigkeit darstellt. Es ist zu erkennen, dass die Reihenfolge der Kernel (nach deren Gesamtlaufzeit) die gleiche ist, jedoch nimmt `kernelShiftParticles` nur noch $56\,\%$ der Zeit in Anspruch. Auch die Lücke, in der die Ausgabe des Detektors erfolgt, ist im Vergleich dazu deutlich größer und dauert damit fast so lang, wie ein Zeitschritt. Da der Detektor in beiden Fällen Werte in doppelter Genauigkeit speichert, ist die Zeit für dessen Ausgabe annähernd konstant und für beide Simulation gleich mit etwa $150\,\mathrm{ms}$. Aus dem Verhältnis der Balken lässt sich bereits erkennen, dass die Simulation mit einfacher Genauigkeit deutlich schneller ist, als die in doppelter Genauigkeit.

Die in den vorangegangenen beiden Simulationen verwendeten Größen sind so gewählt, dass der Grafikspeicher (bei Verwendung doppelter Genauigkeit) zu einem Großteil gefüllt wird, wodurch die GPU gut ausgelastet wird. Als Vergleich soll das Profil in Abb. 6.14a für eine etwas kleinere Simulation mit $128 \times 64 \times 256$ Zellen pro GPU (um ein Faktor $2{,}5$ kleiner) und ansonsten gleicher Konfiguration mit doppelter Genauigkeit dienen. Deutlich ist der geringere Anteil von `kernelShiftParticles` an der Zeitlinie, obwohl er diese immer noch dominiert und das größere Verhältnis der Laufzeiten von Ausgabe und Zeitschritt.

Da der Kernel stark von Speicheroperationen abhängt, ist eine hohe SMX-Belegung vorteilhaft, die den Ausgleich von Latenzen ermöglicht. Laut dem Profiler beträgt die maximal mögliche Auslastung nur $50\,\%$, was laut Tatourian [72] (der wiederum auf Daten von NVIDIA verweist) ausreichen kann, um die Latenzen zu verstecken. Eine Erhöhung kann jedoch die Performance verbessern. Laut dem *CUDA Occupancy Calculator* aus dem CUDA Toolkit ist der Kernel durch die genutzten 52 Register/Thread limitiert und eine Verringerung dieser auf maximal 48 würde die Auslastung auf $63\,\%$ erhöhen. Da die Registerallokation vom Compiler vorgenommen wird, lässt sich dies nicht direkt ändern. Indirekt beeinflussen lässt sie sich durch Annotation des Kernels mit `__launch_bounds__` unter Angabe von `maxThreadsPerBlock` (vgl. [44]), was jedoch in libPMacc erfolgen müsste und die Kenntnis der maximalen Threads je Block für alle Anwendungen erfordert und damit nicht praktikabel ist. Die ma-



(a) Doppelte Genauigkeit



(b) Einfache Genauigkeit

Abbildung 6.14: Zeitleiste der Doppelspalt-Simulation (∼2 Mio. Zellen / GPU)

ximale Registeranzahl für alle Kernel lässt sich auch durch das Compilerflag `maxregcount` festlegen, was jedoch andere Kernel negativ beeinflussen könnte, die dadurch Werte aus Registern in den langsamen globalen Speicher auslagern müssen. Sinnvoller ist da die Entfernung der Fehlerausgabe mittels `printf` auf der GPU, für den Fall, dass kein neuer Frame alloziert werden konnte. Die Umsetzung steht noch aus, ein vorläufiger Test, in dem der Befehl vollständig entfernt wurde, zeigt jedoch, dass in diesem Fall nur noch 48 Register benötigt werden, was vielversprechend ist. Obwohl zu erwarten wäre, dass der Registerverbrauch bei einfacher Genauigkeit geringer ist, zeigt der Profiler dafür die gleiche Anzahl an Registern. Zu vermuten ist, dass der Verbrauch durch die Verwaltung der Frames bestimmt wird, die in beiden Fällen sehr ähnlich ist, exakt überprüfen lässt sich dies jedoch nur Analyse des erzeugten Maschinencodes, was aufgrund des hohen Aufwands hier nicht getan wurde. Wünschenswert wären an dieser Stelle Tools von NVIDIA, welche diese Analyse erleichtern.

Die Analyse des Propagations-Kernels zeigt ebenfalls eine maximale Auslastung von 50 % bei 62 Registern und 2 KiB Shared Memory, was ebenfalls zur Limitierung durch die Register führt. Auch hier müssten diese, um eine höhere Auslastung zu erreichen, laut CUDA Occupancy Calculator auf 48 reduziert werden, was sicherlich zu mehr Speicheroperationen führt und damit eher langsamer wird. Dieser Kernel profitiert jedoch sehr von der Nutzung einfacher Genauigkeit, womit der Registerverbrauch auf 37 reduziert und die Auslastung auf 75 % erhöht wird, was ausreichen sollte, um die Latenzen auszugleichen. Diese Werte stammen aus dem in Abb. 6.14b dargestellten Profil, indem eine zumindest geringfügige Verbesserung der Laufzeit zu erkennen ist.

Abb. 6.15 zeigt die Laufzeiten von Simulationen mit einfacher und doppelter Genauigkeit für sechs Beispiele (ohne bestimmte Reihenfolge), die nun verglichen und ausgewertet werden sollen. Test 1 zeigt die Laufzeiten für Simulationen, wie sie in Abschnitt 6.1 verwendet wurden, um u. a. die Abbildungen 6.4 und 6.5 zu erzeugen, wobei hier Wert auf eine möglichst geringe Gesamtlaufzeit gelegt wurde. Dass sich

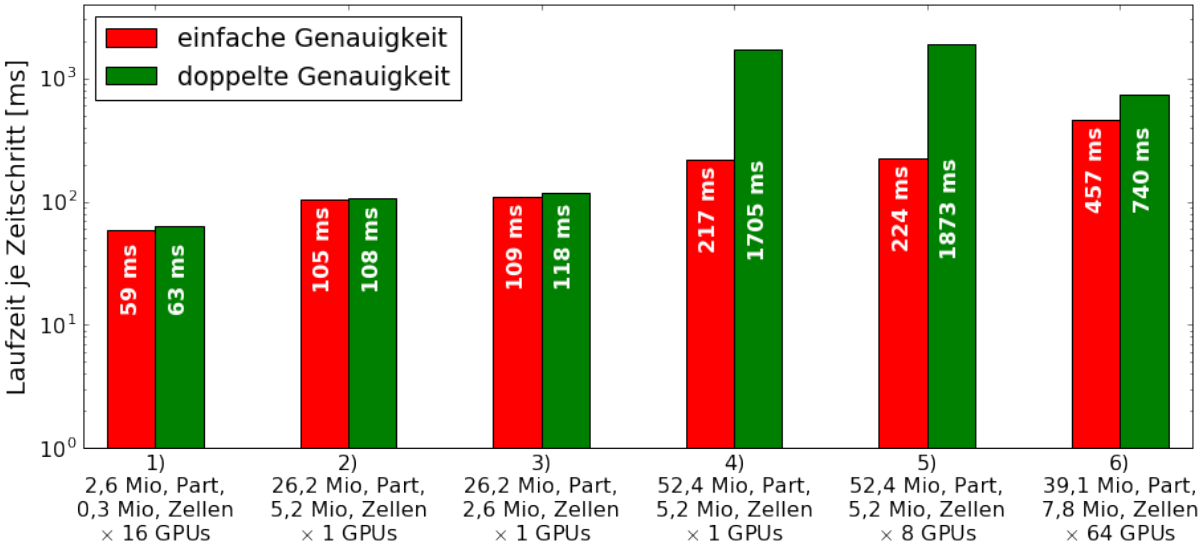


Abbildung 6.15: Laufzeiten für Simulation mit einfacher und doppelter Genauigkeit. Angaben der Partikelanzahlen und Zellen sind je GPU und auf eine Nachkommastelle gerundet

die Zeiten hier nicht wesentlich unterscheiden, deutet darauf hin, dass die GPU nicht ausreichend ausgelastet war und die Laufzeit durch die Latenzen bestimmt ist, die nicht durch Ausführung anderer Warps ausgeglichen werden konnten. Das zweite Beispiel sieht ähnlich aus, trotz dass hier 10 mal mehr Partikel vorhanden sind. Dies lässt sich nicht mehr mit der Auslastung der SMX erklären. Dazu soll der ähnliche Test 3 betrachtet werden, dessen Profile in Abb. 6.14 dargestellt sind, bei denen man ebenfalls die fast gleichen Laufzeiten bei einfacher und doppelter Genauigkeit sieht. Bei der verwendeten Simulationsgröße werden 10240 Blöcke des Propagations-Kernels gestartet, was auch mit dem Profiler kontrolliert wurde und ausreicht, um die 14 SMX der K20X zu füllen, die jeweils maximal 16 Blöcke (vgl. [43]) gleichzeitig verarbeiten können. Der Kernel (zumindest bei einfacher Genauigkeit) ist demnach nicht durch zu geringe Auslastung limitiert. Eine Abschätzung der erreichten Bandbreite aus der Anzahl der Partikel und der Größe der verwendeten Eigenschaften inkl. des Zustands des PRNG und der Laufzeit von rund 33,3 ms zeigt, dass diese mit rund $38\,\mathrm{GB\,s^{-1}}$ deutlich geringer ist, als die maximal mögliche Bandbreite von $250\,\mathrm{GB\,s^{-1}}$ (vgl. [46]). Demnach ist der Kernel auch nicht durch die Bandbreite limitiert. Ähnlich gilt dies für die Rechenkapazität, wobei diese Abschätzung durch den PRNG schwieriger ist. Eine Skalierung der darin verwendeten Ganzzahloperationen auf die Performance von Gleitzahloperationen nach [44] ergibt etwa[34] 26 GFLOPS, was zwei Größenordnungen unter der maximalen Performance von 3,52 TFLOPS (vgl. [46]) ist. Es ist zu vermuten, dass bei dieser Partikeldichte der Overhead durch das Laden des Feldes und die Verwaltung der Partikel in den Frames die Geschwindigkeit limitiert und da dieser bei einfacher und doppelter Genauigkeit näherungsweise[35] gleich ist, ergibt sich auch kein wesentlicher Unterschied bei der Verwendung geringerer Genauigkeit. Vorteil davon ist trotzdem, dass durch den geringeren Speicherverbrauch mehr Partikel auf eine GPU passen, was für größere Simulationen vorteilhaft ist. Tests 4 zeigt nun die Laufzeiten der Simulationen, die in den Profilen auf Abb. 6.13 betrachtet wurden. Hier wird nun der Performancevorteil bei der Verwendung einfacher Genauigkeit

[34] Annahme: 3 Multiply-Add für die Position plus ~ 10 Ganzzahloperationen mit minimal $\frac{1}{3}$ des Durchsatzes für den PRNG $\hat{=}$ 33 Gleitzahloperationen je Partikel

[35] Das Laden des Feldes kann für einfache Genauigkeit schneller sein.

deutlich, der einen Faktor von rund 7,8 bei der Laufzeit ausmacht. Vergleicht man die Tests 2 und 4 zwischen denen die Zahl der Partikel verdoppelt wurde, beobachtet man eine Verdopplung der Zeit bei einfacher Genauigkeit, während sich die Zeit bei doppelter Genauigkeit um den Faktor 17 erhöht. Noch etwas deutlicher ist der Performance Unterschied in Test 5 mit einem Faktor von rund 8,4 in der Laufzeit zwischen einfacher und doppelter Genauigkeit. Dieser Test ist prinzipiell der gleiche, wie Test 4 (gleiche Größen pro GPU, s. Weak Scaling), wird jedoch auf acht GPUs ausgeführt, was auf dem verwendeten Rechencluster zwei Rechenknoten sind. Dadurch wird Kommunikation in und zwischen den Knoten mit gemessen, die bei doppelter Genauigkeit mehr ins Gewicht fällt, da hier mehr Daten übertragen werden müssen. Schließlich zeigt Test 6 die Zeiten für die 40 µm lange Simulation, deren Konvergenz in Abb. 6.9 dargestellt wurde. Obwohl weniger Partikel verwendet werden, ist die Laufzeit bei einfacher Genauigkeit länger, als bei Test 4 und 5, stattdessen ist die Laufzeit bei doppelter Genauigkeit gesunken. Der Laufzeitunterschied in Test 6 ist zwar immer noch ein Faktor von 1,6, jedoch geringer als man es anhand der möglichen Rechenleistung und Speicherbandbreite erwarten würde.

Aus den Beispielen ist zu Erkennen, dass einfache Genauigkeit bei vielen Photonen und besonders bei hohen Photonendichten deutlich bessere Laufzeiten von bis zu einer Größenordnung bringt. Da auch der Speicherverbrauch geringer ist, lassen sich mehr Partikel in einer gegebenen GPU-Konfiguration berechnen, was wichtig wird, wenn größere Streuwinkel (z. B. für Mehrfachstreuung) betrachtet werden sollen, was wiederum deutlich mehr Photonen erfordert, um gute Intensitätsverteilungen auf dem Detektor zu erhalten.

Schließlich komme ich zum Skalierungsverhalten der Simulation. Dazu nutze ich den Doppelspalt von oben und betrachte zuerst das Verhalten bei einer festen Größe von 2 Mio. Zellen und 16 Mio. Partikeln pro GPU, was in Abb. 6.16 gezeigt ist.

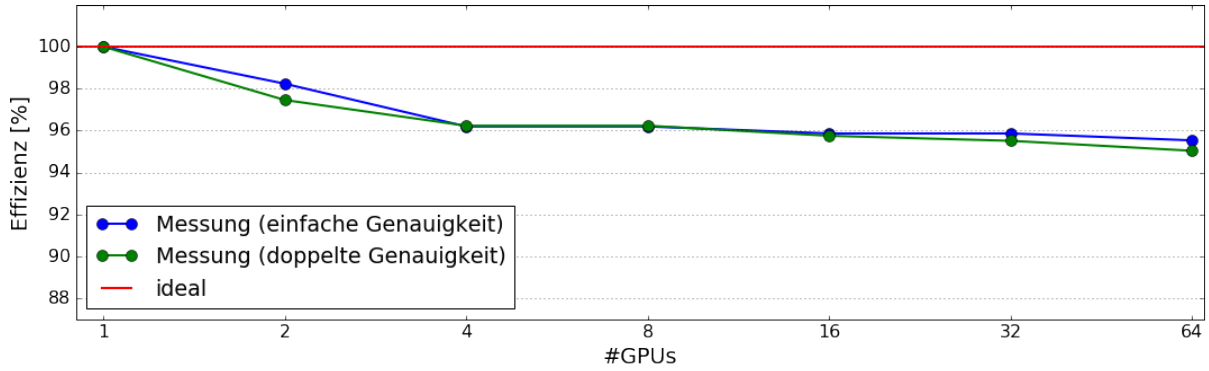


Abbildung 6.16: Weak Scaling: Doppelspalt (2 Mio. Zellen, 16 Mio. Partikeln pro GPU)

Skaliert wurde die Simulation in Richtung des Doppelspalts (z-Achse), wodurch die Anzahl der erzeugten und gestreuten Partikel ebenfalls mit skaliert. Es wird bis 64 GPUs eine Effizienz (Verhältnis von der Zeit auf einer GPU zu der auf N GPUs) von über 95 % erreicht, was zeigt, dass die Kommunikation zwischen den GPUs nur sehr wenig Zeit in Anspruch nimmt. Erwartungsgemäß ist das Weak Scaling bei doppelter Genauigkeit (geringfügig) schlechter, da hier mehr Daten ausgetauscht werden müssen. Interessant ist der annähernd lineare Abfall der Effizienz von einer zu zwei und zu vier GPUs sowie das anschließende Plateau zu acht GPUs, was in sofern erwartet wurde, dass bei zwei GPUs Kommunikation in eine Richtung notwendig ist, während bei vier GPUs in zwei Richtungen kommuniziert werden muss,

wodurch sich der Overhead verdoppelt. Da nur in eine Dimension skaliert wurde, bleibt der Overhead pro GPU danach konstant. Bei 16 GPUs sind bei dem verwendeten Rechencluster vier Rechenknoten notwendig, was die erneute (geringfügige) Reduzierung der Effizienz erklärt, wobei diese durch die Kommunikation mit nur einem Nachbarknoten bei 8 GPUs nicht wesentlich beeinflusst wird. Über weitere Rechenknoten erhöht sich der Kommunikationsoverhead danach nur noch wenig (man beachte die halblogarithmische Skalierung der x-Achse), er ist jedoch bei doppelter Genauigkeit durch die größeren Datenmengen deutlicher erkennbar. Stärker wird der Unterschied beim Weak Scaling, wenn eine größere Simulation betrachtet wird. Verwendet man die selbe Konfiguration wie Test 4 aus Abb. 6.15 ergibt sich das in Abb. 6.17 gezeigte Bild. Während sich die Weak Scaling Effizienz bei einfacher Genauigkeit auf 96 % einpendelt, ist sie bei doppelter Genauigkeit nur noch 90 %, was jedoch immer noch akzeptable Werte sind.

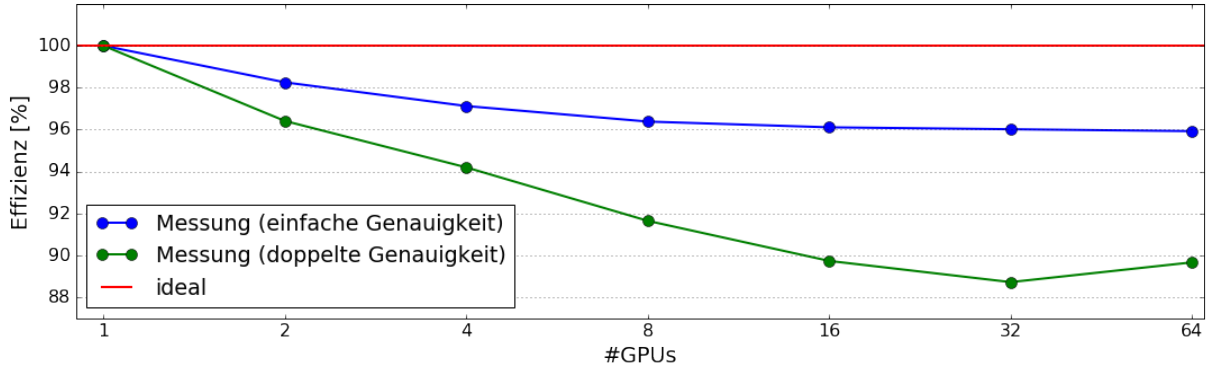


Abbildung 6.17: Weak Scaling: Doppelspalt (5,2 Mio. Zellen, 52,4 Mio. Partikeln pro GPU)

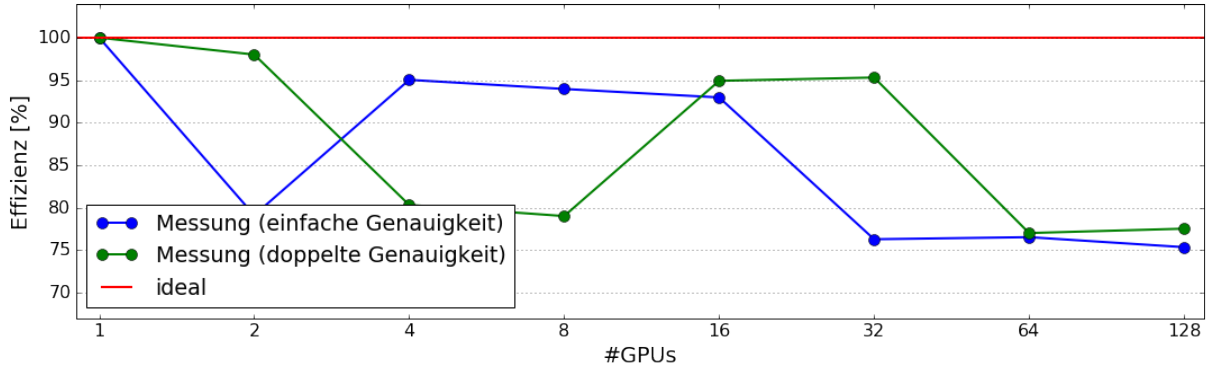


Abbildung 6.18: Weak Scaling (Taurus): Doppelspalt (5,2 Mio. Zellen, 52,4 Mio. Partikeln pro GPU)

Einen interessanten Effekt sieht man bei dem Weak Scaling auf dem Taurus Cluster, das in Abb. 6.18 gezeigt ist. Hierbei wurden K80 Grafikkarten verwendet, die über zwei Grafikprozessoren verfügen [73]. Demzufolge entsprechen zwei GPUs in dem Graphen einer K80 Grafikkarte. Auf der Abbildung sieht man, dass sich die Effizienz des Weak Scalings bei einer Vergrößerung von 64 auf 128 GPUs nicht wesentlich ändert, jedoch fallen auch zwei unterschiedliche Niveaus auf bei rund 95 % und 75 % auf. Eine Erklärung bedarf weiterer Untersuchungen, wahrscheinlich ist jedoch, dass die physische Verbindung der Knoten stark unterschiedliche Latenzen und/oder Geschwindigkeiten aufweist. Durch weitere Optimierungen lässt sich möglicherweise der Einfluss der höheren Latenz reduzieren. So sind in dieser Simulation die Größen der Kommunikationspuffer sehr klein gewählt gewesen, was viele Übertragungen pro Zeitschritt und Richtung zur Folge hatte. Durch eine Vergrößerung dieser Puffer kann dies auf eine

Übertragung pro Zeitschritt und Richtung beschränkt werden. Auch kann die Geometrie der Superzellen angepasst werden, die aktuell auf $4 \times 8 \times 8$ Zellen gesetzt ist, was für die hauptsächliche Bewegung in x-Richtung eventuell nicht optimal ist. Vorläufige Tests zeigten hier jedoch eine langsamere Laufzeit, sodass auch hier noch weitere Analysen notwendig sind. Zu den Tests auf den K80 Grafikkarten ist noch zu bemerken, dass die Laufzeit in doppelter Genauigkeit deutlich kürzer war (etwa um den Faktor Drei) als auf den K20X, was hauptsächlich an dem größeren Speicher liegt.
Die Aussage dieses Weak Scalings ist es, dass sich größere Simulationen mit mehr Partikeln und/oder höherer Auflösung bzw. größeren Volumina durch das Hinzufügen zusätzlicher Rechenressourcen ohne viel zusätzlichen Zeitaufwand berechnen lassen, wenn die Rechenkapazitäten in der gleichen Größenordnung steigen, mit der das Problem wächst. Das ist nicht selbstverständlich, da durch viel oder ineffiziente Kommunikation, der dadurch verursachte Overhead so stark wachsen kann, dass eine (schwache) Skalierung ab einer bestimmten Größe nicht mehr sinnvoll ist. In der hier implementierten Lösung beschränkt sich die Kommunikation auf die direkten Nachbarn, wodurch der erzeugte Overhead ab einer bestimmten Größe praktisch konstant ist, was auch dadurch erreicht wurde, dass der jeder Prozess eine Instanz des Detektors hat, der den Beitrag dieses Prozesses entspricht. Kommunikation zwischen allen Prozessen ist nur zur Ausgabe des Detektors notwendig, was entsprechend selten geschehen sollte, um die Performance nicht zu stark zu beeinflussen. Dabei wird eine Reduktion verwendet, die es je nach Implementierung des MPI-Protokolls vermeidet, dass ein Prozess mit allen kommunizieren muss, in dem diese zuerst Teilergebnisse über mehrere Prozesse bildet, die anschließend rekursiv zusammengeführt werden. In Abb. 6.16 wurde der Detektor alle 3000 Zeitschritte ausgegeben. Dies kann deutlich erhöht werden (s. Abschnitt 6.1), was die Effizienz noch steigern sollte.
Nun folgt das Strong Scaling, mit dem man beschreibt, wie sich die Laufzeit ändert, wenn bei gleicher Problemgröße die Anzahl der Recheneinheiten erhöht werden. Das ist besonders deshalb wichtig, da in bestimmten Fällen, wie Echtzeitsystemen, Anforderungen an die Zeit bis zur Lösung eines Problems bzw. einer Simulation (engl. „Time-to-Solution“) gestellt sind. Vorstellbar ist eine experimentbegleitende Simulation, die dafür verwendet werden kann, die Parameter des Experiments basierend auf den Simulationsergebnissen zu verändern. Das setzt voraus, dass die Simulation in einer vernünftigen (nicht zu großen) Zeit ausgeführt werden kann. Bei einer gegebenen Problemgröße lässt sich das durch zusätzliche Rechenkapazitäten für die parallele Anwendung ermöglichen, wobei die maximale Reduktion der Laufzeit maßgeblich von dem parallel ausführbaren Anteil r_p des Programms abhängt. Das heißt bei einem seriellen Anteil r_s (mit $r_s + r_p = 1$) und n parallelen Einheiten ist der Speedup Faktor S (Verhältnis der Laufzeit von serieller zu paralleler Ausführung) gegeben als $S = (r_s + \frac{r_p}{n})^{-1}$ (sogenanntes Amdahlsches Gesetz, vgl. [74]), womit sich ein maximaler Faktor von r_s^{-1} ergibt.
Abb. 6.19 zeigt das Verhältnis der Laufzeit t_n auf n GPUs zu der Laufzeit t_2 auf zwei GPUs bei Verwendung einfacher und doppelter Genauigkeit. Die Simulation mit doppelter Genauigkeit auf einer GPU dauerte mehr als $36\times$ so lange, wie auf zwei GPUs, während es bei einfacher Genauigkeit nur etwa zweimal so lange war. Um das ansonsten sehr ähnliche Verhalten zu zeigen, wurde die Laufzeit auf einer GPU für die Darstellung nicht verwendet. Eine Erklärung dieses Verhaltens ist möglicherweise, dass die verwendete Bibliothek mallocMC zur parallelen Speicherallokation [75] bei einer hohen Speicherauslastung sehr langsam wird. In dem Fall, sind auch die Ergebnisse aus der Betrachtung von Abb. 6.15 auf Seite 89 entsprechend anders zu interpretieren. Die Aussage, dass die Verwendung von einfacher Genauigkeit bei großen Simulationen kürzere Laufzeiten ermöglicht, behält jedoch ihre Gültigkeit.

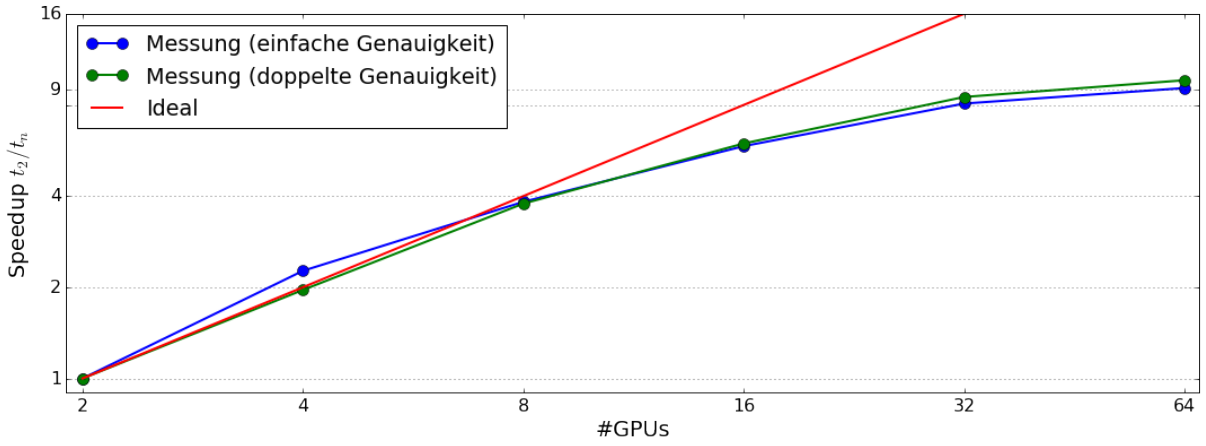


Abbildung 6.19: Strong Scaling: Doppelspalt mit insgesamt 8,4 Mio. Zellen und 67 Mio. Partikeln; Verhältnis der Laufzeit auf n GPUs zu der auf zwei GPUs

Eine grobe Analyse des Speicherverbrauchs zeigt, dass in Test 4 aus Abb. 6.15 25 % des Speichers (68 Tsd. von 276 Tsd. Slots[36]) bei doppelter und 60 % (320 Tsd. von 528 Tsd. Slots) bei einfacher Genauigkeit frei war. Sollte dies bereits der Grund für die obigen Performance-Unterschiede sein, wäre es eine wesentliche Einschränkung bei der Nutzung von mallocMC. Bei diesem Skalierungstest ist die Speicherallokation jedoch sehr sicher der Grund für die lange Laufzeit Die Anzahl der freien Slots beträgt hier bei doppelter Genauigkeit nur etwa 14 von 279 Tsd. Slots (5 %), wobei jeweils nach einem Zeitschritt gemessen wurde. Während des Zeitschritts kann der Kernel, der die Partikel neuen Superzellen zuordnet, mehr Slots benötigen. Wenn je Superzelle ein neuer Frame benötigt würde, währen das bei einer Superzellgröße von 256 Zellen fast 33 Tsd. Slots. Durch die Struktur des Kernels (doppeltes Schachbrettmuster) und der Hauptbewegungsrichtung der Partikel ist es zwar praktisch weniger, trotzdem kann es zwischenzeitlich zur Verwendung aller Slots kommen, was zu der längeren Laufzeit oder sogar zum Abbruch der Simulation führt. Eine genauere Analyse der Kernel und der Allokation und eine eventuelle Optimierung der Allokationen konnte im Rahmen dieser Arbeit nicht gemacht werden, weshalb ich hier diese hohe Laufzeit auf einer GPU als Ausreißer betrachte und außer Acht lasse.

An Abb. 6.19 ist zu erkennen, dass die Simulation bis 16 GPUs sehr gut skaliert und danach gegen einen Faktor von etwas über neun konvergiert. Das heißt, egal wie viele GPUs man noch hinzufügt, man kann nicht mehr als rund 18 mal schneller gegenüber der Ausführung auf einer GPU werden[37]. Andersherum betrachtet ergibt sich eine minimale Größe der Simulation, die noch sinnvoll ist. Soll noch beispielsweise mindestens 60 % parallele Effizienz ($\frac{t_1}{t_n n}$ bzw. hier unter Vermeidung von t_1 und mit t_2 als Referenz: $\frac{t_2 \cdot 2}{t_n n}$) erreicht werden, dürfen für diese Simulationsgröße nicht mehr als 16 GPUs verwendet werden, was ca. 500 Tsd. Zellen und 4 Mio. Partikeln pro GPU entspricht.

Die vollständige Verteilung der Effizienz ist in Abb. 6.20 zu finden. Sehr deutlich ist hier die höhere Effizienz bei vier GPUs zu sehen, die auf einen superlinearen Speedup hinweist, der auch in der vorherigen Abbildung sichtbar ist. Als Ursache ist hier die bessere Ausnutzung der Caches zu vermuten, da bei mehr Partikeln pro GPU die Wahrscheinlichkeit für die Verdrängung einer benötigen Cachezeile steigt. Um das zu bestätigen wären jedoch weitere Analysen notwendig.

[36] Slots sind Speichereinheiten fester Größe, die mit mallocMC alloziert werden können.

[37] Hierbei wurde angenommen, dass die Ausführung auf einer GPU doppelt so lange dauert wie auf zwei GPUs, was bei diesem Test nur bei einfacher Genauigkeit der Fall war.

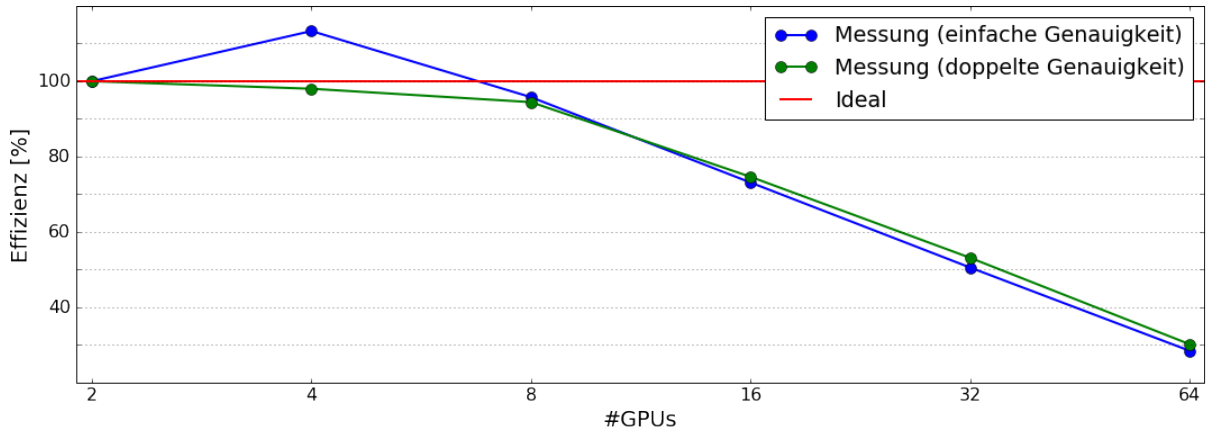


Abbildung 6.20: Strong Scaling Effizienz $\frac{t_2 \cdot 2}{t_n \cdot n}$: Doppelspalt (Gesamt: 8,4 Mio. Zellen, 67 Mio. Partikel)

Zum Schluss möchte ich das Skalierungsverhalten bei einer Erhöhung der Problemgröße zeigen, wozu die Doppelspalt-Simulation auf einer GPU ausgeführt und die Anzahl der Zellen erhöht wurde. Dabei wurden 10 Partikel je Zelle erzeugt und 5000 Zeitschritte simuliert. Das Ergebnis zeigt Abb. 6.21, wobei als Referenz die „ideale Skalierung“ eingezeichnet wurde, die man durch Skalierung der Laufzeit pro Zelle von der kleinsten, betrachteten Simulation mit der Simulationsgröße erhält. Da sich die Zeiten für diese Simulation bei verschiedener Genauigkeit nicht wesentlich unterscheiden, ist nur die ideale Skalierung für einfache Genauigkeit gezeigt, die andere ist nur unwesentlich größer. Zu sehen ist, dass die gemessene Laufzeit, bis auf das letzte Stück, unter der idealen liegt, was darauf hindeutet, dass die kleinste Simulation die GPU nicht voll ausgelastet hat. Ansonsten skaliert die Laufzeit der Simulation sehr gut bis knapp acht Mio. Zellen bei einfacher und gut vier Mio. Zellen bei doppelter Genauigkeit. Danach scheint der begrenzte Speicher die Allokationen und damit die Simulation zu verlangsamen, wofür auch spricht, dass der Effekt mit einfacher Genauigkeit bei der doppelten Simulationsgröße wie mit doppelter Genauigkeit eintritt. Bei 7,9 Mio. Zellen wurde die Simulation in doppelter Genauigkeit wegen vollem Speichers abgebrochen, die in einfacher Genauigkeit erst bei 13 Mio. Zellen.

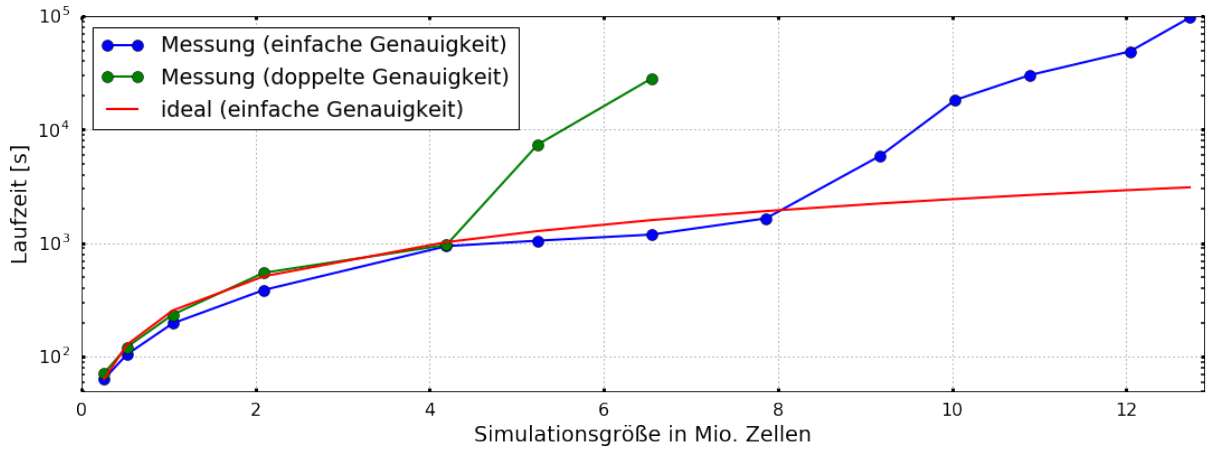


Abbildung 6.21: blau, grün: Skalierungsverhalten bei Änderung der Problemgröße; rot: Gesamtzeit bei angenommener konstanter Laufzeit pro Zelle

7 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde eine Monte-Carlo-Simulation entworfen und implementiert, welche die Röntgenstreuung in einer Probe simuliert und die Intensitätsverteilung auf einem Detektor berechnet. Dazu kann die Dichteverteilung in der Probe durch eine Funktion, eine Reihe von TIFF-Bildern oder in Form einer HDF5-Datei spezifiziert werden. Auch die weiteren Konfigurationsparameter, wie Größen und Auflösungen, sind durch einfach aufgebaute Parameterdateien leicht zu ändern. Zusätzlich besteht die Möglichkeit, durch die Entwicklung weiterer Policy-Klassen Teile des Verhaltens zu verändern, um weitere physikalische Effekte zu berücksichtigen, ohne dass die Struktur der Simulation geändert werden muss. Die Verwendung der Bibliothek libPMacc vereinfachte einerseits die parallele und hoch skalierbare Implementation, was beispielhaft gezeigt wurde, andererseits ermöglicht sie auch die einfache zukünftige Integration dieser Simulation in PIConGPU. Dadurch wird es in Zukunft nicht nur möglich sein, die Röntgenstreuung *in situ* bei der Simulation der Laser-Plasma-Interaktion zu betrachten, sondern auch Rückwirkungen von dem Röntgenlaser auf die Elektronen mit zu berücksichtigen.

In Tests an ausgewählten Beispielen wurde nachgewiesen, dass die berechneten Ergebnisse der Simulation gut mit der analytischen Lösung übereinstimmen. Auch konnte gezeigt werden, dass die Simulation bis zu 80 Mio. Partikel gleichzeitig effizient auf einer K20X GPU berechnen kann und darüber hinaus ein gutes Weak Scaling bei einer Effizienz von über $95\,\%$ in dem getesteten Bereich aufweist. Leider konnte der erwartete Performancevorteil bei der Verwendung einfacher Genauigkeit nicht nachgewiesen werden, wenn die Speicherauslastung gering genug ist. Insbesondere wenn das Verhältnis von Recheneinheiten für einfache und doppelte Genauigkeit in der neuesten GPU-Architektur auf 1:2 steigt, spricht nur noch wenig gegen die Verwendung von doppelter Genauigkeit. Eine Möglichkeit, die sich dadurch bietet, ist die Fernfeldnäherung großteils fallen zu lassen, und die Phasenunterschiede auf dem Detektor direkt zu berechnen. Bei kleineren Abständen zum Detektor können so eventuell Fehler vermieden werden, was jedoch weiteren Untersuchungen bedarf. Allerdings ließ sich bei Verwendung einfacher Genauigkeit ein besseres Weak Scaling nachweisen. Auch die Möglichkeit, die Problemgröße (Anzahl der Zellen und/oder Partikel) zu verdoppeln ist nicht außer acht zu lassen, gerade weil sehr viele Partikel für den Monte-Carlo Ansatz benötigt werden. Noch wichtiger wird der geringere Speicherbedarf bei der Integration in PIConGPU, wo der verfügbare Speicher durch zusätzliche Elemente sehr limitiert ist. Es konnten ebenfalls Effekte der Mehrfachstreuung gezeigt werden, wobei auch hier noch weitere Analysen notwendig sind, wie der Vergleich mit analytischen Lösungen. Diese sind jedoch deutlich komplizierter, als bei der Einfachstreuung.

Durch die Analyse und Tests der numerischen Genauigkeit konnte ich zeigen, dass für alle hier relevanten Größen auch mit einfacher Genauigkeit ausreichend exakte Ergebnisse berechnet werden können. Es konnten sogar Grenzen ermittelt werden, bis zu denen dies garantiert ist, was gerade deshalb wichtig ist, da die in Abschnitt 1.2 motivierten Experimente sich größenmäßig nahe dieser Grenzen befinden. Da der erwartete Geschwindigkeitsvorteil ausblieb, sind noch weitere Tests notwendig, um den Grund dafür zu finden. Das kann sich durchaus als lohnenswert erweisen, da eine Optimierung an der Stelle

eventuell libPMacc und damit auch PIConGPU zugute kommt. Als Fazit aus der Analyse der numerischen Genauigkeit ziehe ich, dass in den allermeisten Fällen eine grobe Abschätzung des numerischen Fehlers ausreicht, die eventuell mit Unterstützung durch numerische Experimente erfolgen kann. Die bigfloat und NumPy Python Bibliotheken haben sich dafür als sehr nützlich erwiesen. Jedoch ist es hilfreich, die hier verwendeten Methoden zur Analyse der Numerik und die Besonderheiten, die bei der Verwendung von Gleitkommazahlen auftreten, zu kennen, um kritische Stellen leichter identifizieren und eventuell vermeiden zu können. Gegenstand weiterer Analysen in dieser Richtung kann die Summation auf dem Detektor sein. Die atomare Addition ist mittels atomaren Compare-And-Swap implementiert, da für Gleitkommazahlen in doppelter Genauigkeit keine passendere Funktion in CUDA vorhanden ist. Auch können bei hohen Intensitäten numerische Fehler durch Absorption auftreten, die hier zwar durch Verwendung doppelter Genauigkeit und lokaler Detektoren so weit wie möglich vermieden wird, jedoch ab bestimmten Größen relevant werden kann. Eventuell lässt sich auch noch das Rauschen verringern, wenn die Ursache nicht in den statistischen Abweichungen durch die Zufallsprozesse liegt. In dem Zusammenhang ist es sinnvoll, auch quasi-zufällige Verteilungen zu berücksichtigen, welche z. B. Flächen gleichmäßiger abtasten können, als das mit pseudo-zufälligen Verteilungen möglich ist. Dabei ist jedoch auch die zusätzlich möglichen Korrelationen zu untersuchen und es wird mehr Speicher durch die Zustände verbraucht. Auch besteht die Möglichkeit, den aktuellen Zustand des Detektors periodisch zu speichern und diese Zwischenergebnisse am Ende zu kombinieren. Dadurch kann der numerische Fehler bei der Addition der komplexen Zahlen verringert werden, was insbesondere bei deutlich höheren Photonenzahlen notwendig werden kann.

Zukünftige Untersuchungen sollten sich mit dem Einfluss der Wahrscheinlichkeiten und Streubereiche beschäftigen. Zwar sind die physikalischen Größen dafür bekannt, jedoch ist es u. U. nicht sinnvoll alle Streuwinkel zu berücksichtigen, da dies eine langsamere Konvergenz zur Folge hat. Ob, und welche Bereich hier im Sinne der Varianzreduktion zu vernachlässigen sind, sollte untersucht werden. Auch die Verwendung „gewichteter“ Photonen (ähnlich zu den Makropartikeln in PIConGPU) und die Erzeugung weiterer Photonen bei einem Streuprozess ist denkbar, um die Anzahl der explizit zu simulierenden Partikel zu reduzieren.

Literatur

[1] Debus, J.; Wannenmacher, M.: *Mit schweren Ionen gegen Krebs*. In: *Ruperto Carola* 3 (1999). URL: http://www.uni-heidelberg.de/presse/ruca/ruca99_3/debus.html (besucht am 12. 09. 2016).

[2] Altarelli, M.: *Opportunities for resonant elastic X-ray scattering at X-ray free-electron lasers*. In: *European physical journal special topics* 208 (2012), S. 351–357. ISSN: 1951-6355. DOI: 10.1140/epjst/e2012-01629-8. URL: http://bib-pubdb1.desy.de/record/97430.

[3] Als-Nielsen, J.; McMorrow, D.: *Elements of modern X-ray physics*. 2. Aufl. Wiley, 2011. ISBN: 9780470973950.

[4] URL: http://www.xfel.eu/ueberblick/zahlen_und_fakten/ (besucht am 10. 09. 2016).

[5] Zinth, W.; Zinth, U.: *Optik: Lichtstrahlen - Wellen - Photonen*. 4. Aufl. München: Oldenbourg, 2013. ISBN: 9783486721362.

[6] Stößel, W.: *Fourieroptik: eine Einführung*. Berlin Heidelberg: Springer, 1993. ISBN: 3540532870.

[7] *PIConGPU Github Repository*. URL: https://github.com/ComputationalRadiationPhysics/picongpu (besucht am 02. 09. 2016).

[8] Passoni, M.; Bertagna, L.; Zani, A.: *Target normal sheath acceleration: theory, comparison with experiments and future perspectives*. In: *New Journal of Physics* 12.4 (2010), S. 045012. URL: http://stacks.iop.org/1367-2630/12/i=4/a=045012.

[9] Schreiber, J.: *Teilchenbeschleunigung mit Licht*. Forschungsbericht. Garching: Max-Planck-Institut für Quantenoptik, 2013. URL: https://www.mpg.de/6806457/MPQ_JB_2013 (besucht am 12. 09. 2016).

[10] Sauerbrey, R.; Schramm, U.: *Kompakte Beschleuniger für die Medizin*. In: *Welt der Physik* (30. Aug. 2007). URL: http://www.weltderphysik.de/gebiet/atome/laserteilchenbeschleunigung/kompakte-beschleuniger-fuer-die-medizin/ (besucht am 12. 09. 2016).

[11] Kluge, T.; Gutt, C.; Huang, L.; Metzkes, J.; Schramm, U.; Bussmann, M.; Cowan, T. E.: *Using X-ray free-electron lasers for probing of complex interaction dynamics of ultra-intense lasers with solid matter*. In: *Physics of Plasmas* 21.3 (März 2014), S. 33110.

[12] Glatter, O.; Kratky, O.: *Small Angle X-Ray Scattering*. London: Academic Press, Dez. 1982. ISBN: 9780122862809.

[13] Bussmann, M.; Burau, H.; Cowan, T. E.; Debus, A.; Huebl, A.; Juckeland, G.; Kluge, T.; Nagel, W. E.; Pausch, R.; Schmitt, F.; Schramm, U.; Schuchart, J.; Widera, R.: *Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability*. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013, 5:1–5:12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2504564.

[14] Antyufeev, V. S.: *Monte Carlo method for solving inverse problems of radiation transfer*. Zeist: VSP, 2000. ISBN: 978-3-11-092030-7.

[15] Choi, M.; Ghammraoui, B.; Badal, A.; Badano, A.: *Monte Carlo X-ray transport simulation of small-angle X-ray scattering instruments using measured sample cross sections*. In: *Journal of Applied Crystallography* 49.1 (Feb. 2016), S. 188–194. DOI: 10.1107/S1600576715023924.

[16] Persliden, J.: *Monte Carlo Simulation of Multiple Scattering in Compton Spectroscopy*. In: *Acta Radiologica* 33.4 (1992), S. 384–387. ISSN: 0284-1851. DOI: 10.1080/02841859209173199.

[17] Alerstam, E.; Svensson, T.; Andersson-Engels, S.: *Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration*. In: *Journal of Biomedical Optics* 13.6 (2008). DOI: 10.1117/1.3041496.

[18] Badal, A.; Badano, A.: *Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit*. In: *Medical Physics* 36.11 (2009), S. 4878–4880. DOI: 10.1118/1.3231824.

[19] Kaiser, G.: *A Friendly Guide to Wavelets*. Modern Birkhäuser Classics. Birkhäuser Boston, 2010. ISBN: 9780817681111.

[20] Rahman, M.: *Applications of Fourier Transforms to Generalized Functions*. WIT Press, 2011. ISBN: 9781845645649.

[21] Oppenheim, A. V.; Schafer, R. W.: *Zeitdiskrete Signalverarbeitung*. 3. Aufl. München, Wien: Oldenbourg, 1999. ISBN: 3-486-24145-1.

[22] Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P.: *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3. Aufl. New York, NY, USA: Cambridge University Press, 2007. ISBN: 9780521880688.

[23] Cooley, J. W.; Tukey, J. W.: *An algorithm for the machine calculation of complex Fourier series*. In: *Mathematics of Computation* 19 (1965), S. 297–301. ISSN: 0025-5718. URL: http://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/ (besucht am 01. 09. 2016).

[24] *Fastest Fourier Transform in the West*. URL: http://www.fftw.org/ (besucht am 01. 09. 2016).

[25] *cuFFT*. URL: https://developer.nvidia.com/cufft (besucht am 01. 09. 2016).

[26] *The Intel® Math Kernel Library and its Fast Fourier Transform Routines*. URL: https://software.intel.com/en-us/articles/the-intel-math-kernel-library-and-its-fast-fourier-transform-routines (besucht am 01. 09. 2016).

[27] *clFFT Github Repository*. URL: https://github.com/clMathLibraries/clFFT (besucht am 01. 09. 2016).

[28] Knox, K.: *AMD releases APPML source code, creates clMath library*. 13. Aug. 2013. URL: http://developer.amd.com/community/blog/2013/08/13/amd-releases-appml-source-code-creates-clmath-library/ (besucht am 01. 09. 2016).

[29] Spinellis, D.: *Another Level of Indirection*. In: *Beautiful Code: Leading Programmers Explain How They Think*. Hrsg. von Oram, A.; Wilson, G. Sebastopol, CA: O'Reilly und Associates, 2007, S. 279–291. ISBN: 0-596-51004-7. URL: http://www.dmst.aueb.gr/dds/pubs/inbook/beautiful_code/html/Spi07g.html (besucht am 09. 02. 2016).

[30] Freeman, E.; Freeman, E.: *Entwurfsmuster von Kopf bis Fuß*. Ein Buch zum Mitmachen und Verstehen. O'Reilly, 2006. ISBN: 9783897214217.

[31] Alexandrescu, A.: *Modern C++ Design*. 14. Aufl. Dez. 2006. ISBN: 0201704315.

[32] Grund, A.: *Library for library independant FFTs*. URL: https://github.com/ComputationalRadiationPhysics/liFFT (besucht am 09. 02. 2016).

[33] *Using-Plans*. URL: http://www.fftw.org/fftw3_doc/Using-Plans.html#Using-Plans (besucht am 09. 02. 2016).

[34] Glinnemann, J.; Schwarzenbach, D.: *Kristallographie*. Springer Berlin Heidelberg, 2013. ISBN: 9783642595301.

[35] Kluge, T.; Bussmann, M.; Chung, H.-K.; Gutt, C.; Huang, L.; Zacharias, M.; Schramm, U.; Cowan, T.: *Nanoscale femtosecond imaging of transient hot solid density plasmas with elemental and charge state sensitivity using resonant coherent diffraction*. In: *Physics of Plasmas* 23.3 (2016), S. 033103.

[36] Crawford, F. S.: *Schwingungen und Wellen*. 3. Aufl. Braunschweig: Vieweg, 1989.

[37] Tsuji, K.; Injuk, J.; Van Grieken, R.: *X-Ray Spectrometry: Recent Technological Advances*. Wiley, 2005. ISBN: 9780470020425.

[38] *Laser Konfigurationsdatei für LaserWakefield Beispiel in PIConGPU*. URL: https://github.com/ComputationalRadiationPhysics/picongpu/blob/release-0.2.0/examples/LaserWakefield/include/simulation_defines/param/laserConfig.param (besucht am 09. 03. 2016).

[39] Piskozub, J.; McKee, D.: *Effective scattering phase functions for the multiple scattering regime*. In: *Opt. Express* 19.5 (Feb. 2011), S. 4786–4794. DOI: 10.1364/OE.19.004786.

[40] *Parallele Berechnungen mit CUDA*. URL: http://www.nvidia.de/object/cuda-parallel-computing-de.html (besucht am 03. 09. 2016).

[41] Zenker, E.; Worpitz, B.; Widera, R.; Huebl, A.; Juckeland, G.; Knüpfer, A.; Nagel, W. E.; Bussmann, M.: *Alpaka - An Abstraction Library for Parallel Kernel Acceleration*. In: IEEE Computer Society, 23. Mai 2016. arXiv: 1602.08477. URL: http://arxiv.org/abs/1602.08477.

[42] Grund, A.: *Leistungsvergleich plattformunabhängiger Anwendungen für Hardwarebeschleuniger*. Großer Beleg. TU Dresden, 2014.

[43] NVIDIA: *NVIDIA Kepler GK110 Architecture Whitepaper*. 2012. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (besucht am 03. 09. 2016).

[44] *CUDA C Programming Guide. CUDA 7.5*. 1. Sep. 2015. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (besucht am 28. 08. 2016).

[45] *GPU Monte Carlo*. URL: http://www.atomic.physics.lu.se/biophotonics/research/monte-carlo-simulations/gpu-monte-carlo/ (besucht am 06. 09. 2016).

[46] NVIDIA: *Tesla K Produktfamilie – Übersicht*. URL: http://www.nvidia.de/content/PDF/kepler/Tesla-KSeries-Overview-LR.pdf (besucht am 06. 09. 2016).

[47] NVIDIA: *Tesla K80 Grafikprozessor*. URL: http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-k80-overview.pdf (besucht am 06. 09. 2016).

[48] Phillips, A.: *Introduction to Quantum Mechanics*. Manchester Physics Series. Wiley, 2013. ISBN: 9781118723258.

[49] Adinetz, A.: *CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics*. 1. Okt. 2014. URL: http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[50] Higham, N. J.: *Accuracy and Stability of Numerical Algorithms*. 2. Aufl. Philadelphia, PA, USA: Society for Industrial und Applied Mathematics, 2002. ISBN: 0898715210.

[51] Knuth, D.: *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Pearson Education, 2014. ISBN: 9780321635761.

[52] *cuRand Documentation*. 1. Sep. 2015. URL: http://docs.nvidia.com/cuda/curand/ (besucht am 11. 09. 2016).

[53] *HDF5 Homepage*. URL: https://www.hdfgroup.org/HDF5/ (besucht am 11. 09. 2016).

[54] *LibSplash Github repository*. URL: https://github.com/ComputationalRadiationPhysics/libSplash (besucht am 11. 09. 2016).

[55] *OpenPMD Standard*. URL: https://github.com/openPMD/openPMD-standard (besucht am 11. 09. 2016).

[56] URL: http://www.nvidia.de/object/gpu-architecture-de.html (besucht am 12. 09. 2016).

[57] Goldberg, D.: *What Every Computer Scientist Should Know About Floating-point Arithmetic*. In: *ACM Comput. Surv.* 23.1 (März 1991), S. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163. (Besucht am 09. 02. 2016).

[58] Harris, R.: *You're Going To Have To Think!* In: *Overload Online* 99 (Okt. 2010), S. 4–9. URL: http://accu.org/var/uploads/journals/overload99.pdf (besucht am 09. 02. 2016).

[59] Sterbenz, P. H.: *Floating point computation*. 1974.

[60] *Basic Issues in Floating Point Arithmetic and Error Analysis*. URL: https://people.eecs.berkeley.edu/~demmel/cs267/lecture21/lecture21.html (besucht am 12. 08. 2016).

[61] Einarsson, B.: *Accuracy and Reliability in Scientific Computing*. SIAM e-books. Society for Industrial und Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2005. ISBN: 9780898718157.

[62] Wilkinson, J. H.: *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994. ISBN: 0486679993.

[63] Castaldo, A. M.: *Error analysis of various forms of floating point dot products*. Masterarbeit. The university of Texas at San Antonio, Aug. 2007. URL: http://www.csc.lsu.edu/~whaley/theses/castaldo_ms.pdf (besucht am 08. 29. 2016).

[64] URL: https://pypi.python.org/pypi/bigfloat/ (besucht am 12. 09. 2016).

[65] Schwoerer, H.; Pfotenhauer, S.; Jäckel, O.; Amthor, K.-U.; Liesfeld, B.; Ziegler, W.; Sauerbrey, R.; Ledingham, K. W. D.; Esirkepov, T.: *Laser-plasma acceleration of quasi-monoenergetic protons from microstructured targets*. In: *Nature* 439.7075 (Jan. 2006), S. 445–448. ISSN: 0028-0836. DOI: 10.1038/nature04492.

[66] Kahan, W.: *Pracniques: Further Remarks on Reducing Truncation Errors*. In: *Commun. ACM* 8.1 (Jan. 1965), S. 40 ff. ISSN: 0001-0782. DOI: 10.1145/363707.363723.

[67] Walther, T.; Walther, H.: *Was ist Licht?: von der klassischen Optik zur Quantenoptik*. Beck Reihe. Beck, 1999. ISBN: 9783406447228.

[68] Feynman, R. P.; Leighton, R. B.; Sands, M. L.: *The Feynman lectures on physics, Volume III: Quantum mechanics*. Basic Books, 2010. ISBN: 9780465024179.

[69] URL: http://imagej.net/ (besucht am 12. 09. 2016).

[70] Commons, M.; Mazur, J.; Nevin, J.; Rachlin, H.: *The Effect of Delay and of Intervening Events on Reinforcement Value: Quantitative Analyses of Behavior*. Quantitative Analyses of Behavior Series Bd. 5. Taylor & Francis, 2013. ISBN: 9781317838074.

[71] URL: http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html (besucht am 15. 09. 2016).

[72] Tatourian, A.: *NVIDIA GPU Architecture & CUDA Programming Environment*. URL: https://code.msdn.microsoft.com/vstudio/NVIDIA-GPU-Architecture-45c11e6d (besucht am 10. 09. 2016).

[73] *NVIDIA TESLA K80*. URL: http://www.nvidia.de/object/tesla-k80-de.html (besucht am 10. 09. 2016).

[74] Amdahl, G. M.: *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, S. 483–485. DOI: 10.1145/1465482.1465560.

[75] *mallocMC Github repository*. URL: https://github.com/ComputationalRadiationPhysics/mallocMC (besucht am 11. 09. 2016).

Abbildungsverzeichnis

Danksagung

An dieser Stelle möchte ich mich bei Dr. Andreas Knüpfer von der TU Dresden für die Betreuung dieser Arbeit sowie bei Dr. Michael Bussmann vom Helmholtz-Zentrum Dresden-Rossendorf (HZDR) für das sehr interessante Thema und seine Unterstützung bedanken. Großer Dank gebührt auch Dr. Thomas Kluge, der mich als Betreuer seitens des HZDR in vielen physikalischen Fragen unterstützt hat, sowie dem HZDR selbst, dass u. a. den Rechencluster *hypnos* betreibt, der für die vielen Simulationen im Rahmen dieser Arbeit verwendet wurde und ohne diesen es zumindest sehr viel schwerer gewesen wäre. Danken möchte ich auch meinen Kollegen am Institut für Strahlenphysik des HZDR, mit denen ich mich während meiner Arbeit über viele interessante Themen austauschen konnte und Unterstützung bei Schwierigkeiten erhalten habe. Auch wäre diese Diplomarbeit ohne die Unterstützung der Professoren Dr. Wolfgang E. Nagel und Dr. Wolfgang V. Walter nicht möglich gewesen.
Schließlich geht mein Dank auch an meine Familie und Freunde für die fortwährende Unterstützung im Laufe des Studiums und an Franziska Görlitz, die mich besonders während der letzten Phase des Diploms viel unterstützt und aufgemuntert hat.