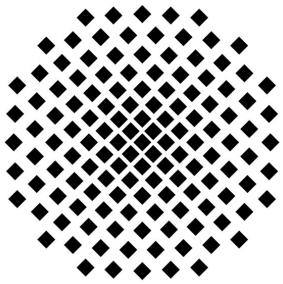

Universität Stuttgart
Institut für Parallele und Verteilte Systeme
Abteilung Parallele Systeme

Diplomarbeit Nr. 3026

GPU-basierte Beschleunigung der diskreten Dipol-Approximation für Streulichtsimulationen

Steffen Kieß

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr.-Ing. Sven Simon
Betreuer:	Tjark Bringewat
Beginn am:	12.04.2010
Beendet am:	17.11.2010
CR-Nummer:	D.1.3, G.1.3, J.2

Zusammenfassung

Die Diskrete Dipol-Approximation (DDA) ist ein numerisches Verfahren zur winkelaufgelösten Simulation von Lichtstreuung an Nanopartikeln. Die Durchführung des Verfahrens ist, insbesondere bei großen Partikeln oder hohen Brechungsindizes, sehr rechenaufwendig.
Diese Arbeit beschleunigt die DDA durch Parallelisierung auf Grafik-Hardware. Hierfür werden die für eine Performance-Steigerung relevanten Operationen identifiziert und auf die Grafikkarten-Architektur angepasst.
Die im Rahmen dieser Arbeit entwickelte parallele Implementierung erreicht gegenüber einer äquivalenten sequentiellen Implementierung eine Beschleunigung um den Faktor drei bis vier.

Inhaltsverzeichnis

1 Einleitung

Die diskrete Dipol-Approximation (englisch „discrete dipole approximation“, DDA) ist ein Verfahren, das dazu dient, die Lichtstreuung an Nanopartikeln zu berechnen. [boh98]
Benötigt werden derartige Streulichtsimulationen beispielsweise, um das Streuverhalten von Zellen für medizinische Zwecke zu berechnen. Andere Einsatzmöglichkeiten sind die Berechnung des Streuverhaltens von interstellarem Staub in der Astronomie sowie die Überwachung von Fertigungsprozessen in der Industrie.
Die DDA nähert das Streuverhalten eines Partikels durch ein Gitter von Dipolen an. Die Interaktionen zwischen den einzelnen Dipolen werden in einer Interaktionsmatrix beschrieben. Mittels dieser Interaktionsmatrix wird ein lineares Gleichungssystem (LGS) aufgestellt. Die Lösung dieses LGS gibt für jeden Dipol die Polarisation an. Aus den Polarisationsvektoren lassen sich dann verschiedene Streucharakteristika wie z.B. das interne elektrische Feld, das gestreute elektrische Feld oder Absorptions- und Auslöschungsquerschnitte berechnen.
Der größte Teil der Rechenzeit wird zur Lösung des LGS benötigt.
Die Interaktionsmatrix ist eine Multilevel-Block-Toeplitz-Matrix. Unter Ausnutzung dieser Eigenschaft kann der Speicherbedarf für die Matrix von $O(n^2)$ auf $O(n)$ reduziert werden. Das LGS wird durch ein iteratives Verfahren gelöst, wobei unter Ausnutzung der Block-Toeplitz-Eigenschaft der Rechenaufwand für eine Matrix-Vektor-Multiplikation von $O(n^2)$ auf $O(n \log n)$ gesenkt wird.
Um das Lösen des LGS zu beschleunigen, wurde im Rahmen dieser Arbeit eine Parallelisierung der DDA auf Grafik-Hardware realisiert. Die Implementierung verwendet als Softwareschnittstelle OpenCL und läuft auf NVIDIA-Grafikkarten ab der GeForce-8-Serie.

1.1 Verwandte Arbeiten

Die Grundzüge der DDA wurden von DeVoe 1964 in [dev64] vorgestellt. Hierbei wurden Verzögerungseffekte nicht berücksichtigt, so dass das Verfahren nur für Objekte funktioniert, die verglichen mit der Wellenlänge klein sind [dra94].
Purcell und Pennypacker stellten das Verfahren einschließlich der Verzögerungseffekte 1973 in [pur73] vor. Sie verwendeten das Verfahren, um die Lichtabsorption von interstellarem Staub zu simulieren. Die Zahl der simulierbaren Dipole war hierbei aus Speicher- und Geschwindigkeitsgründen auf etwa 256 beschränkt.
Yung entwickelte 1978 ein Verfahren [yun78], mit dem sich etwa 10 000 Dipole simulieren lassen, allerdings konvergiert das Verfahren bei stark absorbierenden Partikeln nicht [dra88].
Flatau et al. verwenden in [fla90] die Block-Toeplitz-Struktur der Interaktionsmatrix, um den Speicherbedarf zu senken und die Geschwindigkeit der Berechnung zu erhöhen.

Goodman et al. beschleunigen in [goo91] die Berechnung der DDA, indem sie das oben genannte LGS durch eine Faltung repräsentieren, die sich dann mit Hilfe einer schnellen Fourier-Transformation (FFT) sehr effizient evaluieren lässt.
Draine und Flatau untersuchen in [dra94] die Performance und Genauigkeit der DDA.
Penttilä et al. vergleichen in [pen07] verschiedene DDA-Implementierungen mit exakten Techniken.
Yurkin und Hoekstra geben in [yur07] einen Überblick über aktuelle Entwicklungen im Bereich der DDA.

1.2 Existierende Implementierungen

Einige existierende DDA-Implementierungen werden in [pen07] verglichen.

1.2.1 DDSCAT

DDSCAT [dds, dra94, dra10] wurde von Draine und Flatau entwickelt. Das Programm ist in Fortran 90 geschrieben und ist unter der GPL3 frei verfügbar.
DDSCAT besitzt Unterstützung für Clusterbetrieb mit MPI (Message Passing Interface) und kann die Intel Math Kernel Library[1] verwenden.
Es gibt Unterstützung für einfache und doppelte Genauigkeit.

1.2.2 ADDA

ADDA [add, yur07] wurde von Yurkin und Hoekstra an der Universität von Amsterdam entwickelt. ADDA ist in C geschrieben und unter der GPL3 frei verfügbar. Als FFT-Implementierung kann die Temperton FFT oder FFTW verwendet werden.
Um den benötigten Arbeitsspeicher und die benötigte Rechenzeit zu senken zerlegt ADDA die zur Lösung der Matrix-Vektor-Multiplikation notwendige 3D-FFT in drei einzelne 1D-FFTs und nutzt aus, dass bei der Vorwärtstransformation Teile des Vektors mit Nullen besetzt sind und nach der Rücktransformation Teile nicht verwendet werden.
Die in dieser Arbeit vorgestellte DDA-Implementierung übernimmt dieses Prinzip und baut auf ADDA auf.
ADDA benutzt FFTW[2] oder die Temperton FFT als FFT-Implementierung. Der Code ist auf doppelte Genauigkeit ausgelegt, lässt sich jedoch leicht für einfache oder 80-bit-Genauigkeit abändern.
ADDA besitzt ebenfalls Unterstützung für Clusterbetrieb mit MPI.

[1] `http://software.intel.com/en-us/articles/intel-mkl/`
[2] `http://www.fftw.org/`

1.2.3 OpenDDA

OpenDDA [ope, mcd07, mcd09] wurde von Mc Donald 2009 entwickelt.
Ähnlich wie ADDA spaltet OpenDDA die 3D-FFT in drei einzelne 1D-FFTs auf, um die Geschwindigkeit zu steigern und den Speicherbedarf zu senken.
Auch OpenDDA besitzt Unterstützung für Clusterbetrieb mit MPI.

1.3 Problemstellung

Ziel dieser Diplomarbeit ist es, die DDA durch Parallelisierung auf Grafik-Hardware zu beschleunigen. Als Softwareschnittstelle soll dabei OpenCL verwendet werden.
Um den Beschleunigungsfaktor zu überprüfen, soll ADDA als Referenz genommen werden.
Hierbei müssen insbesondere die Schritte der DDA, die einen hohen Anteil der Rechenzeit benötigen, auf ihre Parallelisierbarkeit untersucht und in OpenCL umgesetzt werden.
Nach Abschluss der Arbeit soll eine über die Kommandozeile bedien- und konfigurierbare Anwendung zur Verfügung stehen.

1.4 Gliederung der Arbeit

In Kapitel 1 wird die DDA erläutert und existierende Forschungsarbeiten und DDA-Implementierungen werden betrachtet.
In Kapitel 2 werden die theoretische Grundlagen der DDA dargestellt und die einzelnen Schritte der DDA erläutert.
In Kapitel 3 wird die im Rahmen dieser Arbeit entwickelte DDA-Implementierung, die auf Grafikkarten läuft, im Detail vorgestellt. Hierbei werden insbesondere die Parallelisierung und GPU-spezifische Optimierungen beschrieben.
In Kapitel 4 wird die Performance der Implementierung untersucht und mit der Performance von ADDA verglichen.
In Kapitel 5 wird die Arbeit zusammengefasst und es wird ein Ausblick auf weitere Optimierungsmöglichkeiten gegeben.
In Anhang A werden die verwendeten Symbole erklärt.
In Anhang B wird für Begriffe, bei denen oft der englische Begriff verwendet wird, in dieser Arbeit aber ein deutscher Begriff, eine englische Übersetzung angegeben.
In Anhang C wird die Bedienung der für diese Arbeit entwickelten Anwendung erläutert.

2 Theoretische Grundlagen der DDA

Dieses Kapitel gibt einen Überblick über die Herleitung der DDA und ihre Implementierung, wie sie in der im Rahmen dieser Arbeit entstandenen Anwendung verwendet wird.
Bei der DDA wird Lichtstreuung an einem Partikel simuliert, indem das Partikel durch eine Anzahl von Dipolen angenähert wird [mis00]. Jeder Dipol hat dabei eine Position und einen Polarisierbarkeits-Tensor.
Die folgende Darstellung beschränkt sich auf die für das grundlegende Verständnis des DDA-Verfahrens wesentlichen Informationen. Für eine ausführlichere Betrachtung des hier Wiedergegebenen siehe [dra88, dra94, mis02].

2.1 Voraussetzungen für die DDA

Um die DDA auf ein Streuproblem anwenden zu können müssen bestimmte Voraussetzungen erfüllt sein.

- Das Partikel und das Medium müssen nichtleitend sein, da sich ansonsten kein Dipol aufbaut.
- Das Partikel und das Medium müssen optisch näherungsweise linear sein, d.h. die Polarisation eines Dipols muss linear zur Stärke des angelegten Feldes sein.

  Diese Voraussetzung ist in den meisten Fällen erfüllt, nichtlineare Effekte treten nur bei sehr hohen Lichtleistungen auf.
- Für die hier vorgestellte Implementierung muss die magnetische Permeabilität des Partikels ungefähr 1 sein bzw. die magnetische Suszeptibilität ungefähr 0 sein, d.h. das Partikel darf weder stark diamagnetisch noch stark ferromagnetisch sein.

  Diese Einschränkung lässt sich bei Bedarf umgehen siehe [lak92.a, lak92.b], hierbei verliert α die Eigenschaft einer Diagonalmatrix.

  Bei den meisten Materialien ist dies nicht aber notwendig (siehe [dra94] 2.A.).
- Das einfallende Licht muss monochromatisch sein. Für polychromatisches Licht muss die Berechnung für jede Wellenlänge getrennt durchgeführt werden.

2.2 Fundamentale Gleichung

N ist die Anzahl der Dipole. j und l sind Indizes für Dipole, d.h. $j, l \in \mathbb{N}$ und $1 \leq j \leq N$, $1 \leq l \leq N$.
f ist die Frequenz und $\omega = 2\pi f$ die Kreisfrequenz des einfallenden Lichtstrahls.

Alle im folgenden genannten elektrischen Felder und Polarisationen schwingen mit der Frequenz f. Die Felder und Polarisationen werden als komplexe Zahlen dargestellt, wobei der Betrag der komplexen Zahl die Amplitude und das Argument der komplexen Zahl den negativen Nullphasenwinkel der Schwingung angibt. Für einen komplexen Wert x zum Zeitpunkt t ist damit der reale Wert der Feldstärke / Polarisation $\Re(xe^{-i\omega t})$.
P_j ist die induzierte elektrische Polarisation des Dipols j.
$E_{inc,j}$ ist ein dreidimensionaler Vektor, der das durch den einfallenden Lichtstrahl erzeugte elektrische Feld am Dipol j angibt.
Für $l \neq j$ ist $E_{j,l}$ ist das elektrische Feld, das der Dipol l am Dipol j erzeugt. Dieses Feld hängt linear von P_l ab. $-A_{j,l}$ gibt diese Abhängigkeit an, so dass $\forall j \neq l : E_{j,l} = -A_{j,l}P_l$.
E_j ist das elektrische Feld, das am Dipol j anliegt, und besteht aus der Überlagerung von des einfallenden Feldes $E_{inc,j}$ und der Felder der anderen Dipole $E_{j,l}$, d.h.:

$$E_j = E_{inc,j} + \sum_{l \neq j} E_{j,l} \tag{1}$$

α_j ist eine 3x3-Matrix, die die Polarisierbarkeit des Dipols j angibt (= Polarisierbarkeits-Tensor). Damit ist $P_j = \alpha_j E_j$.
Wenn die Dipole für eine längere Zeit dem einfallenden Feld ausgesetzt sind, stellt sich eine konsistente Lösung für P_j ein. Für diese gilt:

$$\begin{aligned}
P_j &= \alpha_j E_j && (2)\\
\alpha_j^{-1} P_j &= E_j && (3)\\
\alpha_j^{-1} P_j &= E_{inc,j} + \sum_{l \neq j} E_{j,l} && (4)\\
\alpha_j^{-1} P_j - \sum_{l \neq j} E_{j,l} &= E_{inc,j} && (5)\\
\alpha_j^{-1} P_j + \sum_{l \neq j} A_{j,l} P_l &= E_{inc,j} && (6)
\end{aligned}$$

Setze nun $A_{j,j} = \alpha_j^{-1}$.

$$\begin{aligned}
A_{j,j} P_j + \sum_{l \neq j} A_{j,l} P_l &= E_{inc,j} && (7)\\
\sum_{l} A_{j,l} P_l &= E_{inc,j} && (8)\\
AP &= E_{inc} && (9)
\end{aligned}$$

$$\begin{pmatrix} A_{1,1} = \alpha_1^{-1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} = \alpha_2^{-1} & \cdots & A_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N,1} & A_{N,2} & \cdots & A_{N,N} = \alpha_N^{-1} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_N \end{pmatrix} = \begin{pmatrix} E_{inc,1} \\ E_{inc,2} \\ \vdots \\ E_{inc,N} \end{pmatrix} \tag{10}$$

A ist eine Matrix mit $3N$x$3N$ Elementen und P und E_{inc} sind Vektoren mit $3N$ Elementen.
Die Elemente $A_{j,l}$ mit $j \neq l$ hängen dabei nur vom Verbindungsvektor zwischen den Dipolen j und l sowie der Wellenlänge λ ab. Die Elemente $A_{j,j} = \alpha_j^{-1}$ hängen vom Brechungsindex des Dipols j ab.
A und E_{inc} (das einfallende Feld) sind bekannt und durch Lösung des LGS kann P (die erzeugte Polarisation) ermittelt werden. Aus P lassen sich dann andere Streucharakteristika (z.B. das interne Feld, das gestreute Fernfeld und Absorptions- und Auslöschungsquerschnitte berechnen.
Die Gleichungen 8, 9 und 10 stellen die fundamentale Gleichung der DDA in verschiedenen Formen dar.

2.3 Iterative Lösungsverfahren

Um das LGS $AP = E_{inc}$ zu lösen scheiden direkte Lösungsverfahren im Allgemeinen aus, da z.B. für das Gauss-Verfahren $O(N^3)$ Rechenzeit benötigt wird und N in der Praxis oft etwa 10^6 ist.
Statt dessen werden iterative Lösungsverfahren verwendet. ADDA bietet vier verschiedene Lösungsverfahren an (siehe [yur10] Kapitel 11.1, [bar94] Kapitel 2.3), auf die auch in dieser Arbeit zurückgegriffen wird:

- **CGNR**
  „Conjugate Gradient on the normal equations“ ist ein modifiziertes „Conjugate Gradient (CG)“-Verfahren, das auch bei nicht positiv-definiten Matrizen funktioniert.

  Das Verfahren ist im Vergleich zu den anderen relativ langsam, hat dafür aber (bei exakter Arithmetik) garantierte Konvergenz nach $3N$ Schritten (für ein LGS mit $3N$ Gleichungen) und es ist weniger anfällig für numerische Instabilität als Bi-CG oder QMR (siehe unten), was vor allem bei der Nutzung von einfacher Genauigkeit vorteilhaft ist.

  Bei jeder Iteration sind 2 Matrix-Vektor-Multiplikationen nötig.

  Pseudo-Code für CGNR ist in Listing 1 zu sehen.

- **Bi-CG CS**
  „BiConjugate Gradient“ ist ein anderes auf CG aufbauendes Verfahren, bei dem zwei Folgen gegenseitig orthogonaler Rest-Vektoren verwendet werden.

  Bei diesem Verfahren wird ausgenutzt, dass die Matrix komplex-symmetrisch ist, was die Zahl der Matrix-Vektor-Multiplikationen pro Iteration von 2 auf 1 senkt.

  Bi-CG ist wesentlich schneller als CGNR, hat jedoch bei einfacher Genauigkeit Probleme mit Rundungsfehlern und konvergiert deswegen für Werte von N ab ungefähr $100\,000$ nicht mehr.

  Bi-CG und QMR sind im Allgemeinen die schnellsten der vier Verfahren.

```
1  // Lösung des LGS Ax = E_inc
2  x_0 = Startwert;
3  r_0 = E_inc - Ax_0; // Matrix-Vektor-Multiplikation
4  for (int i = 1; |r_{i-1}|^2 zu groß; i++) {
5     q_i = A^* r_{i-1}; // Matrix-Vektor-Multiplikation, A^* = zu A adjungierte Matrix
6     ρ_i = |q_i|^2;
7     if (i == 1) {
8        p_i = q_i;
9     } else {
10       p_i = (ρ_i/ρ_{i-1}) p_{i-1} + q_i;
11    }
12    s_i = Ap_i; // Matrix-Vektor-Multiplikation
13    α_i = ρ_i/|s_i|^2;
14    x_i = x_{i-1} + α_i p_i;
15    r_i = r_{i-1} - α_i s_i;
16 }
```

Listing 1. Pseudo-Code CGNR

- **Bi-CGSTAB**

  „BiConjugate Gradient Stabilized“ ist ein auf Bi-CG aufbauendes Verfahren, das sich durch ein weicheres Konvergenzverhalten auszeichnet.

  Bi-CGSTAB benötigt im allgemeinen weniger Iterationen als Bi-CG; da jedoch nicht ausgenutzt wird, dass die Matrix komplex-symmetrisch ist, werden 2 Matrix-Vektor-Multiplikationen pro Iteration benötigt und das Verfahren ist im Allgemeinen langsamer als Bi-CG.

  Numerisch ist das Verfahren stabiler als Bi-CG und kann deswegen auch mit einfacher Genauigkeit angewendet werden.

- **QMR CS**

  „Quasi-Minimal Residual“ ist ein weiteres iteratives Verfahren.

  QMR kann wie Bi-CG ausnutzen, dass die Matrix komplex-symmetrisch ist und benötigt damit nur 1 Matrix-Vektor-Multiplikation pro Iteration.

  Bei einfacher Genauigkeit hat QMR numerische Probleme und konvergiert für Werte von N ab ungefähr $50\,000$ nicht mehr.

  Im Allgemeinen ist QMR zusammen mit Bi-CG das schnellste der vier Verfahren.

2.4 Matrix-Vektor-Produkt

Die Berechnung des Matrix-Vektor-Produkts ist bei der DDA die mit Abstand rechenaufwendigste Komponente, aus diesem Grund wird hier näher auf sie eingegangen. Das Matrix-Vektor-Produkt

wird hierbei über den Zwischenschritt einer FFT berechnet [goo91, bar01].
Damit das Matrix-Vektor-Produkts wie hier beschrieben implementiert werden kann, müssen folgende Bedingungen erfüllt sein:

- Alle Dipole müssen auf einem quaderförmigen Gitter angeordnet sein.

  Dies ist notwendig, damit später D' eingeführt werden kann.

- Keiner der Dipole darf einen Brechungsindex von 1 haben. Ein Brechungsindex von 1 führt zu einer 0 in $\mathrm{diag}(S)$, was Probleme verursacht, weil S^-1 verwendet wird.

  Da nicht alle Punkte auf dem Gitter auf besetzt sein müssen können Dipole, die einen Brechungsindex von 1 haben, weggelassen werden.

Die Matrix D gibt die Abhängigkeiten zwischen den Dipolen ohne die Polarisierbarkeits-Tensoren an, d.h.

$$D_{j,l} = \begin{cases} 0 & \text{für } j = l \\ A_{j,l} & \text{für } j \neq l \end{cases} \tag{11}$$

Damit gilt: $A = \alpha^{-1} + D$.
Es wird angenommen, dass α_j eine Diagonalmatrix ist (siehe Kapitel 2.1). S_j wird nun so gewählt, dass $S_j^2 = \alpha_j$ ist (d.h. S_j ist ebenfalls eine Diagonalmatrix, deren Einträge jeweils die Wurzel des Eintrags in α_j sind).
Zur Verbesserung des Konvergenzverhaltens kann eine Präkonditionierung verwendet werden. Bei der von ADDA und dieser Implementierung verwendeten Präkonditionierung wird statt des LGS

$$AP = E_{inc} \tag{12}$$

das äquivalente Gleichungssystem

$$(SAS)(S^{-1}P) = SE_{inc} \tag{13}$$

gelöst.

$$\begin{aligned} (SAS)(S^{-1}P) &= SE_{inc} && (14)\\ (S(\alpha^{-1} + D)S)(S^{-1}P) &= SE_{inc} && (15)\\ (SS^{-2}S + SDS)(S^{-1}P) &= SE_{inc} && (16)\\ (I + SDS)(S^{-1}P) &= SE_{inc} && (17) \end{aligned}$$

Die Multiplikation von S mit einem beliebigen Vektor lässt sich in $O(N)$ durchführen, da S eine Diagonalmatrix ist. Damit lässt sich SE_{inc} errechnen und am Ende P aus $S^{-1}P$.

Die Matrix-Vektor-Multiplikation, die für das iterative Lösungsverfahren benötigt wird, muss also $y = (I+SDS)x = x+SDSx$ berechnen (wobei x das Argument und y das Ergebnis der Matrix-Vektor-Multiplikation ist und a, b und c Zwischenergebnisse sind) und besteht aus den folgenden Schritten:

1. Berechne $a = Sx$
2. Berechne $b = Da$
3. Berechne $c = Sb$
4. Berechne $y = x + c$

Manche Lösungsverfahren benötigen auch ein Matrix-Vektor-Produkt mit der Adjungierten der Matrix, also $y = (I + SDS)^* x$. Es gilt hierbei:

$$\begin{aligned} y &= (I + SDS)^* x \\ &= I^* x + (SDS)^* x \\ &= Ix + \overline{(SDS)^T} x \\ &= x + \overline{S^T D^T S^T \overline{x}} \\ &= x + \overline{SDS\overline{x}} \text{ (da } S \text{ und } D \text{ symmetrisch sind)} \end{aligned}$$

Zur Berechnung von y werden die folgenden Schritte durchgeführt:

1. Berechne $a = S\overline{x}$
2. Berechne $b = Da$
3. Berechne $c = \overline{Sb}$
4. Berechne $y = x + c$

Die Schritte 1, 3 und 4 lassen sich in $O(N)$ durchführen, da dies nur Operationen sind, bei denen jedes Element des Vektors unabhängig von den anderen verarbeitet werden kann.
Schritt 2 benötigt mit einer einfachen Implementierung als Matrix-Vektor-Multiplikation $O(N^2)$, der Aufwand lässt sich aber durch das in [fla90] entwickelte Verfahren, das im Folgenden beschrieben wird, auf $O(N \log N)$ reduzieren.
Der Wert für $D_{j,l}$ hängt nur vom Verbindungsvektor zwischen dem Dipol j und dem Dipol l ab. Wenn die Dipole auf einem gleichmäßigen Gitter angeordnet sind und Dipol j an den Koordinaten $j = (j_x, j_y, j_z)$ ist, dann lässt sich D' einführen:

$$D'_{j-l} = D_{j,l} \tag{18}$$

Analog zu j_x, j_y und j_z wird auch N_x, N_y und N_z als die Zahl der Dipole in X-, Y- und Z-Richtung eingeführt.
Nun wird das die Größe des Gitters in jeder Richtung verdoppelt und für die neu eingeführten Dipole wird $a_j = 0$ gesetzt. Wenn es Gitterpunkte gibt, die nicht mit Dipolen besetzt sind (bei einem nicht-quaderförmigen Partikel), dann werden diese ebenfalls mit $a_j = 0$ eingeführt.
Es gilt für $j = (j_x, j_y, j_z)$ mit $1 \leq j_x \leq N_x$, $1 < j_y \leq N_y$, $1 < j_z \leq N_z$:

$$\begin{aligned} b_j &= \sum_l D_{j,l} a_l \\ &= \sum_l D'_{j-l} a_l \\ &= \sum_{l_x=0}^{2N_x} \sum_{l_y=0}^{2N_y} \sum_{l_z=0}^{2N_z} D'_{j-(l_x,l_y,l_z)} a_{(l_x,l_y,l_z)} \end{aligned}$$

Dies ist eine 3D-Faltung, die sich über den Umweg einer FFT berechnen lässt. Über die FFT wird (im Gegensatz zu obiger Formel) eine zyklische Faltung berechnet, aber durch das Auffüllen mit Dipolen mit $a_j = 0$ für $j_{x/z/y} > N_{x/y/z}$ ergeben sich für b_j mit $1 \leq j_{x/y/z} \leq N_{x/y/z}$ trotzdem richtige Werte und die Werte b_j mit $j_{x/z/y} > N_{x/y/z}$ werden nicht benötigt.
Damit gilt ($*$ ist die elementweise Multiplikation, also $x * y = \mathrm{diag}(xy^T)$):

$$\begin{aligned} \mathrm{FFT}(b) &= \mathrm{FFT}(D') * \mathrm{FFT}(a) \\ b &= \mathrm{IFFT}(\mathrm{FFT}(D') * \mathrm{FFT}(a)) \end{aligned}$$

$\mathrm{FFT}(D')$ lässt sich in $O(N \log N)$ berechnen und muss bei jeder Berechnung nur einmal (also nicht für jede Iteration neu) berechnet werden.
Die elementweise Multiplikation lässt sich in $O(N)$ berechnen.
$\mathrm{FFT}(a)$ und die IFFT lassen sich in $O(N \log N)$ berechnen.
Die für die gesamte Matrix-Vektor-Multiplikation benötigte Zeit ist damit in $O(N \log N)$, der benötigte Speicher in $O(N)$.

2.5 Auswertung der Dipolpolarisation

Nachdem die Dipolpolarisationen P_j berechnet wurden, lassen sich aus diesen Streucharakteristika ableiten.
Das gestreute elektrische Fernfeld lässt sich durch Gleichung 19 berechnen (siehe [dra94] 4.A. Gleichung 10).

$$E_{sca,x} = \frac{1}{|x|} e^{ik|x|} k^2 (x_u x_u^T - I_3) \sum_{j=1}^{N} e^{-ik\langle x_u, r_j \rangle} P_j \tag{19}$$

Hierbei ist x der Vektor vom Mittelpunkt des Partikels zum gesuchten Punkt im Fernfeld, $|x|$ ist die Entfernung des Punktes vom Mittelpunkt des Partikels, x_u ist die Richtung des Punktes vom Partikel aus (der normierte Vektor x), k ist die Kreiswellenzahl des einfallenden Lichts ($k = \frac{2\pi}{\lambda}$), und r_j ist der Vektor vom Mittelpunkt des Partikels zum Dipol j. $E_{sca,x}$ gibt dann das elektrische Feld am Punkt x an.
Der Auslöschungsquerschnitt C_{ext} lässt sich durch Gleichung 20 berechnen (siehe [dra88] Gleichung 3.02, [dra94] 4.A. Gleichung 8 und [zha08] 3. Gleichung 12).

$$C_{ext} = \frac{4\pi k}{|E_{inc}|^2} \sum_{j=1}^{N} \Im \left(\langle P_j, E_{inc,j} \rangle \right) \tag{20}$$

Der Absorptionsquerschnitt C_{abs} lässt sich durch Gleichung 21 berechnen (siehe [dra88] Gleichung 3.06 und [dra94] 4.A. Gleichung 9).

$$C_{abs} = \frac{4\pi k}{|E_{inc}|^2} \sum_{j=1}^{N} \left(\Im \left(\left\langle P_j, \alpha_j^{-1} P_j \right\rangle \right) - \frac{2}{3} k^3 \left| P_j \right|^2 \right) \tag{21}$$

Der Streuungsquerschnitt C_{sca} lässt sich als $C_{sca} = C_{ext} - C_{abs}$ berechnen.

3 GPU-basierte parallele Implementierung der DDA

Dieses Kapitel beschreibt die Implementierungsdetails der im Rahmen dieser Arbeit entwickelten DDA-Parallelisierung auf Grafik-Hardware.
Besonders wird dabei auf Optimierungen, die speziell bei Grafikkarten nötig sind, eingegangen.

3.1 Architektur

Die Implementierung wird mit OpenCL[3] durchgeführt. OpenCL ist eine architektur- und herstellerunabhängige Schnittstelle, die dazu verwendet werden kann, rechenintensive Probleme zu lösen [khr10]. Besonders ausgelegt ist OpenCL für hochparallele Architekturen.
Die Implementierung ist optimiert für Grafikkarten, die auf der NVIDIA-GPU-Architektur „Compute Unified Device Architecture (CUDA)“ basieren. Eine Anpassung auf andere Grafikkarten-Architekturen, die OpenCL unterstützen, wie z.B. AMD-Grafikkarten, ist möglich. Ohne eine derartige Anpassung ist der Code zwar trotzdem lauffähig, vermutlich aber etwas langsamer.
Bei der Entwicklung für Grafikkarten wird zwischen der Host-Umgebung, die die CPU und das auf der Hauptplatine befindliche RAM bezeichnet, und der Grafikkarten, die aus GPU und dem Grafikkarten-RAM besteht, unterschieden. Um mit der GPU Daten zu bearbeiten müssen diese im Grafikkarten-RAM sein. Da das Kopieren der Daten vom Host-RAM in das Grafikkarten-RAM Zeit benötigt muss beim Entwickeln der Software für die Grafikkarte darüber nachgedacht werden, welche Daten im Host-RAM und welche im Grafikkarten-RAM gespeichert werden und wann diese in das jeweils andere RAM kopiert werden sollen.
Die GPU ist ein Prozessor, der eine Vielzahl von Threads gleichzeitig ausführen kann, jedoch im Gegensatz zur CPU jeden einzelnen Thread nur relativ langsam ausführt.
Im Folgenden wird die NVIDIA-GPU-Architektur beschrieben, für nähere Informationen siehe [nvi10].
Eine Routine, die auf der Grafikkarte ausgeführt wird und von der CPU angestoßen wird, wird als „Kernel“ bezeichnet.
Beim Start eines Kernels kann eine Anzahl von „Work-Groups“ (bei CUDA „Blocks“ genannt) angegeben werden, die jeweils eine Anzahl von „Work-Items“ (bei CUDA „Threads“) enthalten.
Die GPU ist in Einheiten eingeteilt, die bei OpenCL „Compute Units“ und bei CUDA „Streaming Multiprocessors“ genannt werden.
Eine Compute Unit führt dabei jeweils eine Work-Group aus; ist sie damit fertig, so beginnt sie mit der Ausführung der nächsten Work-Group. Die in einer Work-Group enthaltenen Work-Items werden in „Warps“ zu je 32 Work-Items zusammengefasst (der letzte Warp hat evtl. weniger als 32 Work-Items). Die Work-Items innerhalb eines Warps werden parallel ausgeführt. Wenn ein

[3] `http://www.khronos.org/opencl/`

```
1  __kernel void coalesced (__global float* f) {
2    int i = get_global_id (0); // Nummer des aktuellen Work-Items
3
4    f[i] = 4.3f; // Schreibe in den globalen Speicher
5    // Work-Items 0 schreibt hierbei an Speicherstelle f + 0
6    // Work-Items 1 schreibt an Speicherstelle f + 4
7    // Work-Items 2 schreibt an Speicherstelle f + 8
8    // ...
9    // Damit der Speicherzugriff verschmolzen wird, muss f noch auf eine durch
10   // 4 * 32 = 128 bytes teilbare Adresse ausgerichtet sein. Diese Ausrichtung
11   // wird jedoch bei der Allozierung von Speicher durch OpenCL sichergestellt.
12 }
```

Listing 2. Codebeispiel verschmolzener Speicherzugriff

Warp nicht ausgeführt werden kann (z.B. weil auf die Ergebnisse eines globalen Speicherzugriffs gewartet werden muss, siehe unten) so wird mit der Ausführung eines anderen Warps fortgefahren. Alle Work-Items, die gleichzeitig ausgeführt werden, müssen den selben Code ausführen. Verzweigt der Programmfluss bei einigen der Work-Items eines Warps („branch divergence“), dann führt die Compute Unit zuerst den Pfad der einen Work-Items und dann den Pfad der anderen Work-Items aus. Da dies die Performance beeinträchtigt, ist es sinnvoll, darauf zu achten, dass die in einem Warp ausgeführten Work-Items immer das selbe Programmstück ausführen.
Bei Speicherzugriffen ist bei OpenCL zwischen vier verschiedenen Speicherarten zu Unterscheiden:

- **Globaler Speicher**
  Der globale Speicher (bei CUDA auch „global memory“) entspricht dem Grafikkarten-RAM. Dieser Speicher ist verglichen mit den anderen Speicherarten sehr groß (256 MB bis 1,5 GB), hat aber beim Zugriff auch eine relativ hohe Latenzzeit.

  Bei Speicherzugriffen auf den globalen Speicher werden die Zugriffe aller Work-Items eines Warp „verschmolzen“ (englisch „coalesced“), falls die Zugriffe auf direkt hintereinanderliegende Adressen gehen und entsprechend ausgerichtet sind. Speicherzugriffe, die nicht verschmolzen werden können, werden hintereinander ausgeführt, was zu einer wesentlichen Beeinträchtigung der Performance führt. Ein Beispiel für einen verschmolzenen Speicherzugriff ist in Listing 2 zu sehen.

- **Konstanten-Speicher**
  Der Konstanten-Speicher (bei CUDA auch „constant memory“) kann von der GPU aus nicht beschrieben werden. Er hat eine Größe von ungefähr 64 kB hat sehr geringe Latenzzeiten (vergleichbar mit Registerzugriffen).

- **Lokaler Speicher**
  Der lokale Speicher (bei CUDA „shared memory“ genannt) wird von allen Work-Items einer Work-Group geteilt. Er hat eine Größe von etwa 16 kB, die Latenzzeiten sind ebenfalls mit denen von Registerzugriffen vergleichbar.

- **Privater Speicher**
  Der private Speicher (bei CUDA „local memory“ genannt, sollte nicht mit dem bei OpenCL als „local memory“ bezeichneten Speicher verwechselt werden) ist nur für ein einzelnes Work-Item sichtbar. Falls es dem Compiler gelingt, eine Variable im lokalen Speicher auf ein Register abzubilden, ist dieser sehr schnell, ansonsten ist die Geschwindigkeit mit der es globalen Speichers vergleichbar.

3.2 Aufteilung CPU/GPU

Die Lösung des LGS $AP = E_{inc}$ (siehe Gleichung 9) bzw. des äquivalenten LGS $(I+SDS)(S^{-1}P) = SE_{inc}$ (siehe Gleichung 17) benötigt die meiste Zeit bei der Durchführung der DDA.
Aus diesem Grund wird dieser Schritt auf die GPU ausgelagert.
Dabei werden normalerweise alle für die Berechnung notwendigen Vektoren im Speicher der Grafikkarte gehalten, wodurch eine hohe Geschwindigkeit erreicht werden kann. Falls dieser Speicher nicht ausreicht, kann die Matrix D, die einen Großteil des Speicherbedarfs verursacht, auch im RAM gespeichert und dann bei Bedarf stückweise in den Grafikkarten-Speicher kopiert werden. Hierdurch senkt sich die Performance etwas, aber der Algorithmus ist in der Lage, größere Partikel zu verarbeiten.

3.3 Iterative Lösungsverfahren

Bei der Durchführung des iterativen Lösungsverfahrens ist es (außer bei der Berechnung des Matrix-Vektor-Produkts, siehe Kapitel 2.4) ausreichend, nur die Dipole zu betrachten, die einen Brechungsindex ungleich 1 haben („non-void dipoles“). Um die einzelnen Dipole ihrer Position zuordnen zu können wird hierbei noch ein weiterer Vektor mit den Positionen der einzelnen Dipole gespeichert.
Dies führt sowohl zu einer Beschleunigung des iterativen Verfahrens (da weniger Daten bearbeitet werden müssen) als auch zu einer Reduzierung des Speicherbedarfs (da weniger Daten gespeichert werden müssen). Der zusätzliche Bedarf für den Positionsvektor verhältnismäßig gering.
Als Startwert für das iterative Lösungsverfahren werden die Werte 0 und E_{inc} probiert und es wird derjenige genommen, der einen kleineren Restvektor erzeugt. Der Startwert 0 ist dabei eine Näherung für ein großes Partikel, bei dem das angelegte elektrische Feld im Inneren keine Auswirkungen hat, der Wert E_{inc} ist eine Näherung für ein kleines Partikel, das komplett vom angelegten Feld durchdrungen wird.

Das Verfahren wird beendet, sobald die quadrierte Norm des Restvektors kleiner ist als $\epsilon \left|SE_{inc}\right|^2$, wobei normalerweise mit $\epsilon = 10^{-5}$ gearbeitet wird.
Die iterativen Lösungsverfahren benötigen außer der Matrix-Vektor-Multiplikation, die in Kapitel 3.4 erläutert wird, noch folgende Operationen:

- **Linearkombination** Eine häufig benötigte Operation sind Linearkombinationen (einschließlich Skalarmultiplikationen, Vektoradditionen und Vektorsubtraktionen).

  Diese Operationen lassen sich einfach parallelisieren, da die Ergebnisse der einzelnen Einträge in den Vektoren nicht voneinander abhängen.

- **Skalarprodukt und euklidische Norm** Eine andere Operation, die häufig benötigt wird, ist das Skalarprodukt. Die euklidische Norm lässt sich über das Skalarprodukt als $|x| = \sqrt{\langle x, x \rangle}$ berechnen.

  Das Skalarprodukt und die euklidische Norm werden auf der GPU in zwei Kerneln durchgeführt.

  Beim ersten Kernel wird der gesamte Vektor auf alle Compute Units aufgeteilt. Jede Compute Unit verwendet dann alle ihre Work-Items, um ihren Teil des Vektors aufzusummieren. Danach summiert ein Work-Item in jeder Compute Unit die Ergebnisse der einzelnen Work-Items und schreibt das Ergebnis in den globalen Speicher. Pseudo-Code für diesen Kernel ist in Listing 3 zu sehen.

  Der zweite Kernel summiert dann die Ergebnisse der einzelnen Compute Units aus dem ersten Kernel auf. Pseudo-Code für diesen Kernel ist in Listing 4 zu sehen.

- **Skalare Operationen** Operationen auf Skalaren werden auf der CPU ausgeführt, da der Overhead bei der Ausführung auf der GPU zu groß wäre, um einen Performancegewinn zu erbringen.

3.4 Matrix-Vektor-Produkt

Die Berechnung der Matrix-Vektor-Produkte nimmt die meiste Zeit der iterativen Lösungsverfahren in Anspruch.
Während es beim übrigen Lösungsverfahren ausreichend ist, nur die Dipole zu betrachten, die einen Brechungsindex $\neq 1$ haben (besetzte Dipole), müssen beim Matrix-Vektor-Produkt alle Dipole betrachtet werden und das Dipol-Array muss in jeder Dimensionen bis zur Größe $N_{x/y/z}$ mit Nullen aufgefüllt werden (siehe Kapitel 2.4).
Dieses Auffüllen sowie der erste Schritt $a = Sx$ bzw. $a = S\overline{x}$ (für den Fall, dass das Matrix-Vektor-Produkt mit der adjungierten Matrix gesucht ist) wird vom ersten Kernel durchgeführt. Dieser Schritt wird in Kapitel 3.4.1 genauer erklärt.

```
1  // Gibt die quadrierte Norm einer komplexen Zahl zurueck
2  float squared_norm (complex float f) {
3    return real (f) * real (f) + imag (f) * imag (f);
4  }
5
6  // Dieser Kernel wird mit mehreren Work-Groups ausgefuehrt
7  __kernel void norm_squared1(uint count, // Anzahl der Elemente in input
8                              __global float* tmp, // Temporaerer Speicherbereich
9                              __global const complex float* input // Eingabedaten
10                             ) {
11   // Jedes Work-Item liest Werte aus dem globalen Speicher und summiert sie auf
12   float sum = 0;
13   for (uint i = get_global_id (0); i < count; i += get_global_size (0))
14     sum += squared_norm (input[i]);
15
16   // Wert im lokalen Speicher, in dem die Werte der einzelnen Work-Items
17   // aufsummiert werden
18   __local float local_value;
19
20   // Das erste Work-Item setzt den Wert "local_value" auf 0
21   if (get_local_id (0) == 0)
22     local_value = 0;
23   // Barriere, um sicherzustellen, das alle Work-Items an diesem Punkt
       angelangt
24   // sind, bevor mit der Ausfuehrung fortgefahren wird
25   barrier (CLK_LOCAL_MEM_FENCE);
26   // Gehe durch alle Work-Items in dieser Work-Group
27   for (size_t i = 0; i < get_local_size (0); i++) {
28     // Das aktuelle Work-Item erhoeht local_value um seinen Wert
29     if (get_local_id (0) == i)
30       local_value += value;
31     // Warte auf die anderen Work-Items, damit nicht zwei Work-Items
32     // gleichzeitig auf local_value zugreifen
33     barrier (CLK_LOCAL_MEM_FENCE);
34   }
35
36   // Der erste Thread schreibt das Ergebnis der Work-Group in einen temporaeren
37   // Speicherbereich im globalen Speicher
38   if (get_local_id (0) == 0)
39     tmp[get_group_id (0)] = local_value;
40 }
```

Listing 3. Pseudo-Code euklidische Norm, Kernel 1

```
1  // Dieser Kernel wird mit nur einer Work-Group ausgefuehrt
2  __kernel void norm_squared2(uint count, // Anzahl der Elemente in tmp
3                              __global const float* tmp, // Temporaerer
4                                                         // Speicherbereich
5                              __global float* out // Zeiger auf Ausgabewert
6                              ) {
7    // Jedes Work-Item liest Werte aus dem temporaeren Speicherbereich globalen
8    // Speicher und summiert sie auf
9    float sum = 0;
10   for (uint i = get_global_id (0); i < count; i += get_global_size (0))
11     sum += tmp[i];
12
13   // Analog zu norm_squared1 werden die Werte aller Work-Items aufsummiert
14   __local float local_value;
15   if (get_local_id (0) == 0)
16     local_value = 0;
17   barrier (CLK_LOCAL_MEM_FENCE);
18   for (size_t i = 0; i < get_local_size (0); i++) {
19     if (get_local_id (0) == i)
20       local_value += value;
21     barrier (CLK_LOCAL_MEM_FENCE);
22   }
23
24   // Der erste Thread schreibt das Ergebnis in den globalen Speicher
25   if (get_local_id (0) == 0)
26     *out = local_value;
27 }
```

Listing 4. Pseudo-Code euklidische Norm, Kernel 2

Der nächste Schritt berechnet $b = Da = \mathrm{IFFT}(\mathrm{FFT}(D') * \mathrm{FFT}(a))$. Dieser Schritt wird in Kapitel 3.4.2 genauer erklärt.
Als letztes müssen die besetzten Dipole extrahiert werden und es muss $y = x+Sb$ bzw. $y = x+\overline{Sb}$ berechnet werden. Dieser Schritt wird in Kapitel 3.4.3 genauer erklärt.
Für die folgenden Schritte wird eine als „X-Matrix“ bezeichnete Speicherstruktur verwendet. Diese enthält anfangs den in X-Richtung bis $2N_x$ aufgefüllten Vektor a. Beim Durchführen der Faltung werden in der X-Matrix die Werte von a durch die Werte von b ersetzt. Am Ende wird dann der Wert für y aus der X-Matrix berechnet.
Die meisten Werte, die beim Berechnen des Matrix-Vektor-Produkts verwendet werden, sind dreidimensionale Vektoren (mit je einer Komponente für X-, Y- und Z-Richtung). In der Implementierung werden die einzelnen Komponenten dieser Vektoren getrennt voneinander gespeichert, um verschmolzene Speicherzugriffe (siehe Kapitel 3.1) zu erreichen. Um die folgenden Pseudo-Codes zu vereinfachen werden die Vektoren dort jedoch einfach mit den Typen `float3` etc. dargestellt.

3.4.1 Initialisieren der X-Matrix

Als erstes wird die die X-Matrix mit Nullen gefüllt. Dies geschieht, damit später in der X-Matrix für unbesetzte Dipole und Dipole mit $N_x < j_x \leq 2N_x$ Nullen stehen.
Danach wird ein Kernel aufgerufen (Pseudo-Code in Listing 5), der alle besetzen Dipole durchgeht und für jeden Dipol Sa berechnet und in die X-Matrix schreibt. Sa kann für jeden Dipol unabhängig berechnet werden, weil S eine Diagonalmatrix ist.
Der Index, in den der Eintrag in der X-Matrix geschrieben wird, hängt dabei von der Position des Dipols auf dem Gitter ab.
Der Wert von S für jeden Dipol hängt vom Material des Dipols bzw. von dessen Polarisierbarkeits-Tensor ab. `ccSqrt` enthält dabei für jedes Material die Wurzel des Polarisierbarkeits-Tensors α_j.
Dieser Kernel bildet auch den konjugiert-komplexen Wert von a, falls dies notwendig ist (bei der Berechnung von A^*x).
Der Kernel kann jeden einzelnen Dipol unabhängig von den anderen bearbeiten und lässt sich damit einfach parallelisieren. Die Zugriffe auf `positions`, `materials` und `arg` sind verschmolzen (siehe Kapitel 3.1) und damit schnell. `ccSqrt` ist im Konstanten-Speicher, der Zugriff darauf ist deswegen auch schnell. Der Zugriff auf `xMatrix` ist nicht verschmolzen, da die Werte von `position` nicht notwendigerweise die Positionen von aufeinanderfolgenden Dipolen angeben. Da ausgenutzt werden soll, dass in den Vektoren für das iterativen Lösungsverfahren nur die besetzen Dipole gespeichert werden müssen, lässt sich dies nicht ohne großen Aufwand verhindern, aber die Performanceeinbußen durch dies sind verglichen mit dem Zeitverbrauch durch die Faltung nicht signifikant.

```
1  __kernel void initXMatrix (
2      __global complex float3 xMatrix[][][], // Ausgabe-Matrix (Sx, inklusive
3                                             // Padding)
4      __global const uchar materials[],      // Material der Dipole
5      __global const uint3 positions[],      // Position der Dipole
6      __constant complex float3 ccSqrt[],    // Wurzel des Polarisierbarkeits-
7                                             // Tensors der einzelnen Materialien
8      __global const complex float3 a[],     // Argument x
9      bool adj,                              // true, wenn S\bar{x} berechnet werden
10                                            // soll
11     uint nvCount                           // Zahl der Elemente in materials
12                                            // und positions
13   ) {
14   for (uint i = get_global_id (0); i < nvCount; i += get_global_size (0)) {
15     uint3 position = positions[i];
16     uchar material = materials[i];
17     complex float3 argument = adj ? conj (arg[i]) : arg[i];
18     complex float3 mat = ccSqrt[material];
19     complex float3 res = argument * mat;
20     xMatrix[position.x, position.y, position.z] = res;
21   }
22 }
```

Listing 5. Pseudo-Code zum Initialisieren der X-Matrix

3.4.2 Faltung

Der nächste Schritt ist das Berechnen von $b = Da$. Dies ist eine Faltung (siehe Kapitel 2.4). Die Faltung wird über $b = \mathrm{IFFT}(\mathrm{FFT}(D') * \mathrm{FFT}(a))$ implementiert, FFT und IFFT sind hierbei 3D-FFTs.
Der Wert für $\mathrm{FFT}(D')$ ist unabhängig von x und muss deswegen nicht bei jeder Iteration neu berechnet werden. Sei $D'' = \mathrm{FFT}(D')$.

$$\begin{aligned} b &= \mathrm{unpad}^{XYZ}\mathrm{IFFT}^{XYZ}(D'' * \mathrm{FFT}^{XYZ}\mathrm{pad}^{XYZ}a) && (22)\\ &= \mathrm{unpad}^{Z}\mathrm{unpad}^{Y}\mathrm{unpad}^{X}\mathrm{IFFT}^{Z}\mathrm{IFFT}^{Y}\mathrm{IFFT}^{X} \\ &\quad (D'' * \mathrm{FFT}^{Z}\mathrm{FFT}^{Y}\mathrm{FFT}^{X}\mathrm{pad}^{Z}\mathrm{pad}^{Y}\mathrm{pad}^{X}a) && (23)\end{aligned}$$

Da am Ende nur ein Teil des Ergebnisses der IFFT gebraucht wird, kann, wenn die Berechnung der 3D-IFFT in 3 1D-IFFTs aufgespalten wird, der Berechnungsaufwand verringert werden. Gleiches gilt für die FFT: Bei den drei Dimensionen der FFT ist die obere Hälfte der Daten jeweils mit Nullen gefüllt.

$$\begin{aligned} b &= \mathrm{unpad}^{Z}\mathrm{IFFT}^{Z}\mathrm{unpad}^{Y}\mathrm{IFFT}^{Y}\mathrm{unpad}^{X}\mathrm{IFFT}^{X} \\ &\quad (D'' * \mathrm{FFT}^{Z}\mathrm{pad}^{Z}\mathrm{FFT}^{Y}\mathrm{pad}^{Y}\mathrm{FFT}^{X}\mathrm{pad}^{X}a) && (24)\end{aligned}$$

Da die verwendete FFT-Bibliothek (siehe Abschnitt 3.4.2.1) davon ausgeht, dass die Dimension, über die die FFT durchgeführt wird, eine Schrittweite von 1 hat, muss zwischen den FFTs jeweils noch eine Matrix-Transposition durchgeführt werden.
Pseudo-Code für diese Funktion ist in Listing 6 zu sehen.
Dieses Zerlegen hat nun zum einen den Vorteil, dass die X-Matrix nur $2N$ Elemente hat anstatt $8N$ Elementen, und zum anderen den Vorteil, dass in X-Richtung nur N_yN_z statt $4N_yN_z$ FFTs und IFFTs und in Y-Richtung nur $2N_xN_z$ statt $4N_xN_z$ FFTs und IFFTs durchgeführt werden müssen. Um den Speicherbedarf weiter zu reduzieren kann die Berechnung in Y- und Z-Richtung außerdem in einzelnen Scheiben durchgeführt werden. Entsprechender Pseudo-Code ist in Listing 7 zu sehen. Der Speicherbedarf für die Matrix-Vektor-Multiplikation ist hierbei (abgesehen von dem Speicherbedarf für D'') nur noch $2N+6N_yN_z$`sliceCount`. Ein höherer Wert für `sliceCount` erhöht dabei die Parallelität und senkt den Verwaltungsoverhead, führt jedoch auch zu einem erhöhten Speicherverbrauch.

3.4.2.1 FFT

Als FFT-Implementierung wird CUFFT[4] von NVIDIA verwendet.

[4] `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUFFT_Library.pdf`

```
1  void matrixVectorProduct (const complex float3 input[],                  // Argument x
2                            complex float3 output[],                       // Ausgabe y
3                            const complex float3x3 dmatrixFft[][][]        // FFTD'
4                            ) {
5    complex float3 xMatrix[2N_x][N_y][N_z];    // 2N Elemente
6    complex float3 yMatrix[2N_y][N_z][2N_x];   // 4N Elemente
7    complex float3 zMatrix[2N_z][2N_y][2N_x];  // 8N Elemente
8
9    setToNull (xMatrix);              // Setze auf 0, damit aufgefüllte Dipole den Wert 0 haben
10   initXMatrix (input, xMatrix);     // Berechne a = Sx, siehe Kapitel 3.4.1
11
12   fftX (xMatrix);                   // N_yN_z FFTs mit einer Größe von 2N_x
13   setToNull (yMatrix);              // Für aufgefüllte Dipole in y-Richtung
14   transpose (xMatrix, yMatrix);
15   fftY (yMatrix);                   // 2N_xN_z FFTs mit einer Größe von 2N_y
16   setToNull (zMatrix);              // Für aufgefüllte Dipole in z-Richtung
17   transpose (yMatrix, zMatrix);
18   fftZ (zMatrix);                   // 2N_x2N_z FFTs mit einer Größe von 2N_z
19
20   // Für alle j_{x,y,x}: zMatrix_{x,y,z} = dmatrixFft_{x,y,z} zMatrix_{x,y,z}
21   multiplication (dmatrixFft, zMatrix);
22
23   ifftZ (zMatrix);                  // 2N_x2N_z IFFTs mit einer Größe von 2N_z
24   transpose (zMatrix, yMatrix);
25   ifftY (yMatrix);                  // 2N_xN_z IFFTs mit einer Größe von 2N_y
26   transpose (yMatrix, xMatrix);
27   ifftX (xMatrix);                  // N_yN_z IFFTs mit einer Größe von 2N_x
28
29   createResVec (input, xMatrix, output); // Berechne Ergebnis y = x + Sb, siehe Kapitel 3.4.3
30 }
```

Listing 6. Pseudo-Code Matrix-Vektor-Produkt

```
1  const int sliceCount = 8;
2  void matrixVectorProduct (const complex float3 input[],              // Argument x
3                            complex float3 output[],                   // Ausgabe y
4                            const complex float3x3 dmatrixFft[][][]    // FFTD'
5                            ) {
6    complex float3 xMatrix[2N_x][N_y][N_z];          // 2N Elemente
7    complex float3 yMatrix[2N_y][N_z][sliceCount];   // 2N_yN_z sliceCount Elemente
8    complex float3 zMatrix[2N_z][2N_y][sliceCount];  // 4N_yN_z sliceCount Elemente
9
10   setToNull (xMatrix);             // Setze auf 0, damit aufgefüllte Dipole den Wert 0 haben
11   initXMatrix (input, xMatrix);    // Berechne a = Sx, siehe Kapitel 3.4.1
12
13   fftX (xMatrix);                  // N_yN_z FFTs mit einer Größe von 2N_x
14   for (int x = 0; x < 2N_x; x += sliceCount) {
15     // Führe die folgenden Operationen für min(sliceCount, 2N_x - x) Elemente in X-Richtung aus
16     setToNull (yMatrix);           // Für aufgefüllte Dipole in y-Richtung
17     transpose (xMatrix, yMatrix);
18     fftY (yMatrix);                // 2N_xN_z FFTs mit einer Größe von 2N_y
19     setToNull (zMatrix);           // Für aufgefüllte Dipole in z-Richtung
20     transpose (yMatrix, zMatrix);
21     fftZ (zMatrix);                // 2N_x2N_z FFTs mit einer Größe von 2N_z
22
23     // Für alle j_{x,y,x}: zMatrix_{x,y,z} = dmatrixFft_{x,y,z} zMatrix_{x,y,z}
24     multiplication (dmatrixFft, zMatrix);
25
26     ifftZ (zMatrix);               // 2N_x2N_z IFFTs mit einer Größe von 2N_z
27     transpose (zMatrix, yMatrix);
28     ifftY (yMatrix);               // 2N_xN_z IFFTs mit einer Größe von 2N_y
29     transpose (yMatrix, xMatrix);
30   }
31   ifftX (xMatrix);                 // N_yN_z IFFTs mit einer Größe von 2N_x
32
33   createResVec (input, xMatrix, output); // Berechne Ergebnis y = x + Sb, siehe Kapitel 3.4.3
34 }
```

Listing 7. Pseudo-Code Matrix-Vektor-Produkt, scheibenweise

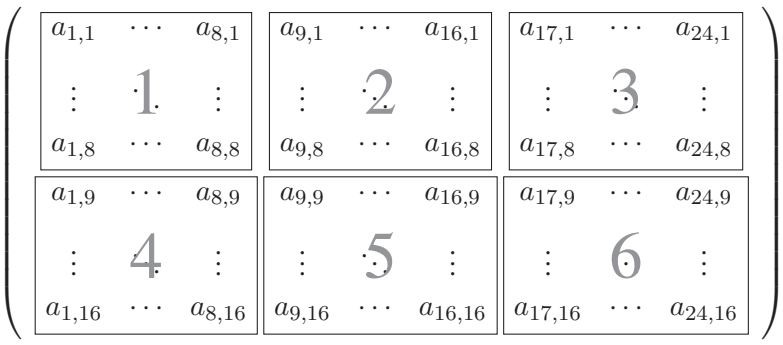

Abbildung 1. Eingabematrix für Transposition

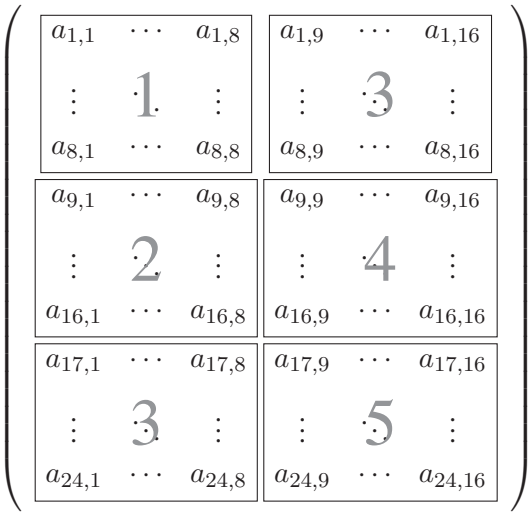

Abbildung 2. Ausgabematrix der Transposition

Diese Implementierung ist besonders schnell, wenn die Größe der FFT nur die Primfaktoren 2, 3, 5 und 7 enthält. Um dies zu erreichen wird a nicht nur bis zur Größe $2N_x$ aufgefüllt, sondern bis zur kleinsten Größe, die mindestens $2N_x$ groß ist und nur die Primfaktoren 2, 3, 5 und 7 enthält. (Gleiches gilt für N_y und N_z.) Hierbei muss auch D' aufgefüllt werden, damit es die gleiche Größe wie a hat.

3.4.2.2 Transposition Zwischen den FFT- bzw. IFFT-Schritten muss eine Matrix-Transposition durchgeführt werden.
Um diese auf der GPU effizient zu implementieren muss darauf geachtet werden, dass die Speicherzugriffe auf den globalen Speicher verschmolzen sind.
Als Beispiel wird hier die Transposition einer 24x16-Matrix in Abbildung 1 zu einer 16x24-Matrix in Abbildung 2 betrachtet. Die Matrizen enthalten komplexe Gleitkommazahlen mit einfacher Ge-

```
__kernel void multiplication (__global complex float3 zMatrix[][][],
                              __global const complex float3x3 dMatrixFft[][][])
                                  {
  uint z = get_global_id (0);
  uint y = get_global_id (1);
  uint x = get_global_id (2);

  complex float3 val = zMatrix[z, y, x];
  complex float3x3 d = dMatrixFft[x, y, z];

  complex float3 result;
  result.x = d.xx * val.x + d.xy * val.y + d.xz * val.z;
  result.y = d.xy * val.x + d.yy * val.y + d.yz * val.z;
  result.z = d.xz * val.x + d.yz * val.y + d.zz * val.z;

  zMatrix[z, y, x] = result;
}
```

Listing 8. Pseudo-Code zum Berechnen des Produkts aus D-Matrix und Z-Matrix

nauigkeit, d.h. jeder Eintrag benötigt $4 + 4 = 8$ Bytes. Beide Matrizen sind zeilenweise angeordnet, d.h. bei der Eingabematrix ist die Anordnung im Speicher $\{A_{1,1}, A_{2,1}, A_{3,1}, \dots\}$, bei der Ausgabematrix ist die Anordnung im Speicher $\{A_{1,1}, A_{1,2}, A_{1,3}, \dots\}$.
Bei der Transposition wird immer ein 8x8-Block von einem Half-Warp, d.h. von 16 Work-Items, bearbeitet. (Wenn die Höhe oder Breite der Matrix kein Vielfaches von 8 ist sind einige der Blöcke unvollständig.) Dieser 8x8-Block wird mit 16 Work-Items aus dem globalen Speicher gelesen und in den lokalen Speicher geschrieben. Jedes Work-Item liest bzw. schreibt dabei 4 Bytes auf einmal. Auf diese Weise kann sichergestellt werden, dass sowohl die Lesezugriffe als auch die Schreibzugriffe verschmolzen sind.

3.4.2.3 Innere Multiplikation Zur Berechnung der Faltung muss nach der FFT eine Multiplikation zwischen $\mathrm{FFT}D'$ und $\mathrm{FFT}a$ durchgeführt werden. Die Elemente von $\mathrm{FFT}D'$ sind hierbei symmetrische 3x3-Matrizen, die Elemente von $\mathrm{FFT}a$ 3D-Vektoren. Die 3x3-Matrizen werden durch 6 Werte dargestellt (die redundanten Einträge werden ausgelassen).
Pseudo-Code für diese Operation ist in Listing 8 zu sehen.

3.4.3 Berechnen des Ergebnisvektors

Der Ergebnisvektor der Matrix-Vektor-Multiplikation wird berechnet, indem nach Rücktransformation der in Abschnitt 3.4.2.3 berechneten Werte aus der X-Matrix die Werte für die besetzten Dipole extrahiert werden und dann $y = x + Sb$ bzw. $y = x + \overline{sb}$ berechnet wird.

```
__kernel void createResVec (
    __global const complex float3 xMatrix[][][], // Eingabe-Matrix
    __global const uchar materials[],            // Material der Dipole
    __global const uint3 positions[],            // Position der Dipole
    __constant complex float3 ccSqrt[],          // Wurzel des
                                                 // Polarisierbarkeits-Tensors
                                                 // der einzelnen Materialien
    __global const complex float3 arg[],         // Argument x
    __global complex float3 res[],               // Ergebnisvektor y
    bool adj,                                    // true, wenn x+sb berechnet
                                                 // werden soll
    uint nvCount                                 // Zahl der Elemente in
                                                 // materials und positions
  ) {
  for (uint i = get_global_id (0); i < nvCount; i += get_global_size (0)) {
    uint3 position = positions[i];
    uint material = materials[i];
    complex float3 xval = xMatrix[position.x, position.y, position.z];
    complex float3 mat = ccSqrt[material];
    complex float3 argument = adj ? conj (arg[i]) : arg[i];
    complex float3 res = xval * mat + argument;
    res[i] = adj ? conj (res) : res;
  }
}
```

Listing 9. Pseudo-Code zum Berechnen des Ergebnisvektors

```
complex float3 calcEsca (const complex float3 pvec[], // Dipolpolarisationen
                         const float3 positions[],     // Positionsvektoren der Dipole
                         complex float3 scatDir,       // Richtung, für die die Streuung
                                                       // berechnet wird
                         float dist                    // Entfernung, für die die Streuung
                                                       // berechnet wird
                         ) {
  complex float3 sum = 0;
  for (int i = 0; i < N; i++) {
    sum += exp (-i * k * (scatDir * positions[i])) * pvec[i];
  }
  return 1 / dist * exp (i * k * dist) * (scatDir * (scatDir * sum) - sum);
}
```

Listing 10. Pseudo-Code Berechnung $E_{sca,x}$

Der Kernel in Listing 9 führt dies durch. Dieser Kernel ist im wesentlichen analog zu dem in Kapitel 3.4.1 vorgestellten: Er extrahiert die Werte aus der X-Matrix und wendet S auf sie an. Auch dieser Kernel kann für den Fall, dass A^*x berechnet werden soll, bei den entsprechenden Vektoren den konjugiert-komplexen Vektor bilden.
Wie beim Kernel zur Initialisierung der X-Matrix sind auch bei diesem Kernel alle Speicherzugriffe außer dem Zugriff auf die X-Matrix entweder verschmolzen oder Zugriffe auf den Konstanten-Speicher.

3.5 Auswertung der Dipolpolarisation

Aus den Dipolpolarisationen P_j lassen sich verschiedene Streucharakteristika berechnen, siehe Kapitel 2.5.
Pseudo-Code für die Berechnung von $E_{sca,x}$ ist in Listing 10 zu sehen.
Die Werte für C_{ext} und C_{abs} können analog berechnet werden.

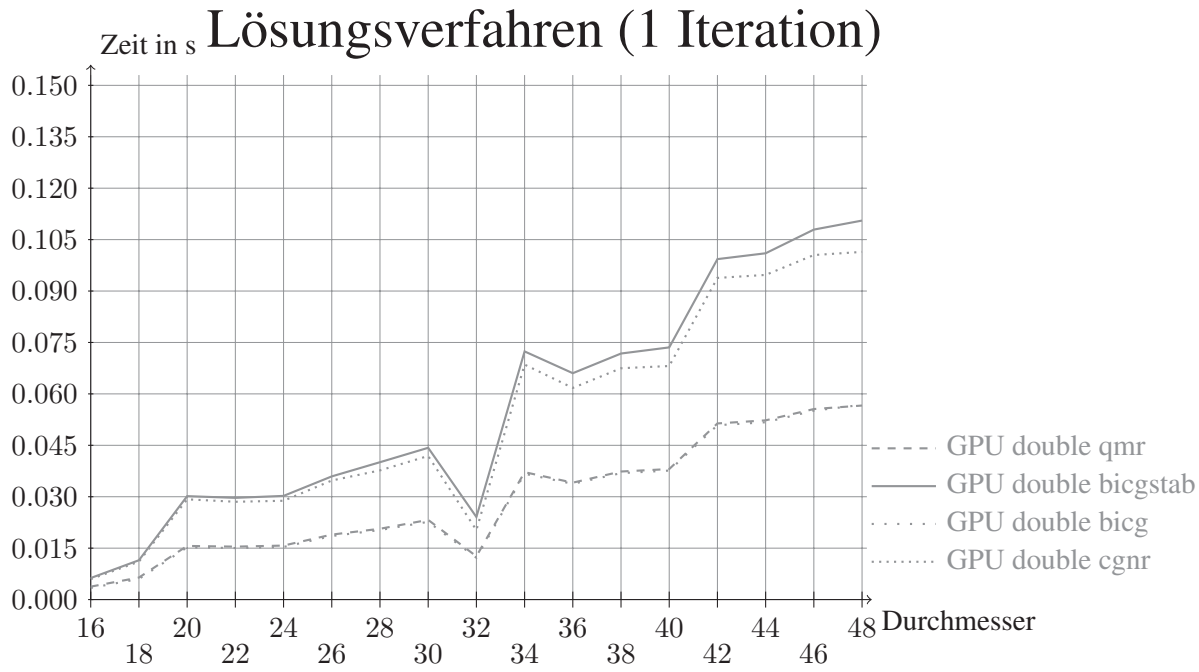

Abbildung 3. Performance Lösungsverfahren, Zeit pro Iteration

4 Performance-Evaluierung

In diesem Kapitel werden verschiedene Performance-Ergebnisse präsentiert.
Die Performance-Messungen werden anhand von Kugeln mit einem Durchmesser von 16 bis 128 Dipolen durchgeführt. Ein Dipol entspricht dabei $0.4189\,\mu\mathrm{m}$, die Wellenlänge des einfallenden Lichts ist $6.2832\,\mu\mathrm{m}$ und der Brechungsindex der Kugel ist 1.5.
Als Grafik-Hardware wurde eine GeForce GTX 480 mit 1,5 GB RAM verwendet. Als CPU wurde eine Intel Core i7 CPU 960 mit 3.20 GHz verwendet, das System hatte 12 GB RAM.
Bei den Kugelgrößen 64 bis 128 wurde die D-Matrix im Host-RAM gespeichert, ansonsten auf der Grafikkarte.

4.1 Verschiedene iterative Lösungsverfahren

Die verschiedenen iterativen Lösungsverfahren unterscheiden sich in ihrer Performance. In Abbildung 3 ist für die vier Lösungsverfahren die für eine Iteration benötigte Zeit dargestellt. Alle Messungen wurden auf der GPU mit doppelter Genauigkeit durchgeführt. Als Partikel wurden Kugeln mit einem Durchmesser von 16 bis 48 Dipolen genommen. Die Algorithmen Bi-CG CS und QMR CS benötigen immer etwa die gleiche Zeit für eine Iteration, die Algorithmen Bi-CGSTAB und CGNR benötigen etwa die doppelte Zeit für eine Iteration. Der Grund hierfür ist, dass Bi-CG CS

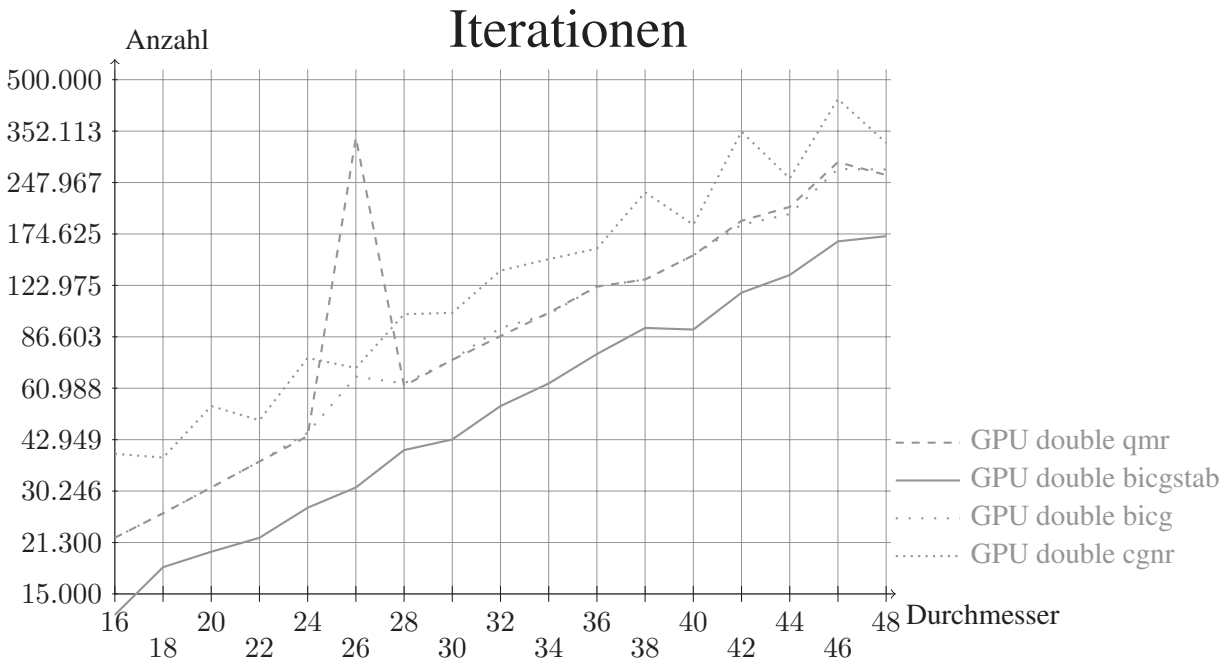

Abbildung 4. Performance Lösungsverfahren, Zahl der Iterationen

und QMR CS nur eine Matrix-Vektor-Multiplikation pro Iteration ausführen während Bi-CGSTAB und CGNR zwei Matrix-Vektor-Multiplikationen pro Iteration ausführen. QMR CS benötigt etwas mehr Zeit als CGNR für eine Iteration, da die zusätzlichen Vektor-Operationen aufwendiger sind, allerdings ist dies verglichen mit dem Zeitaufwand für die Matrix-Vektor-Multiplikation vergleichsweise gering.

In Abbildung 4 ist die Zahl der Iterationen, die zur Lösung des LGS benötigt wird, dargestellt. Bi-CGSTAB benötigt hierbei immer die kleinste Anzahl an Operationen, Bi-CG CS und QMR CS benötigen mehr, haben dafür aber nur eine Matrix-Vektor-Multiplikation pro Iteration. QMR CS benötigt außerdem bei einem Durchmesser von 26 erheblich mehr Zeit, was auf numerische Probleme zurückzuführen ist. CGNR benötigt (außer bei einem Durchmesser von 26) die meisten Iterationen und ist damit am langsamsten, hat aber den Vorteil, dass es stabiler ist als die anderen Verfahren.

In Abbildung 5 ist die benötigte Gesamtzeit (bestehend aus der Zeit pro Iteration mal der Zahl der Iterationen plus Overhead beim Start) zu sehen. Bi-CG CS und QMR CS sind hierbei im Allgemeinen die schnellsten Algorithmen.

Bei größeren Durchmessern wird der Vorsprung von Bi-CG CS und QMR CS größer: In Abbildung 6 und Abbildung 7 ist die Zahl der benötigten Iterationen bzw. der benötigten Gesamtzeit für Durchmesser von 64 bis 128 zu sehen.

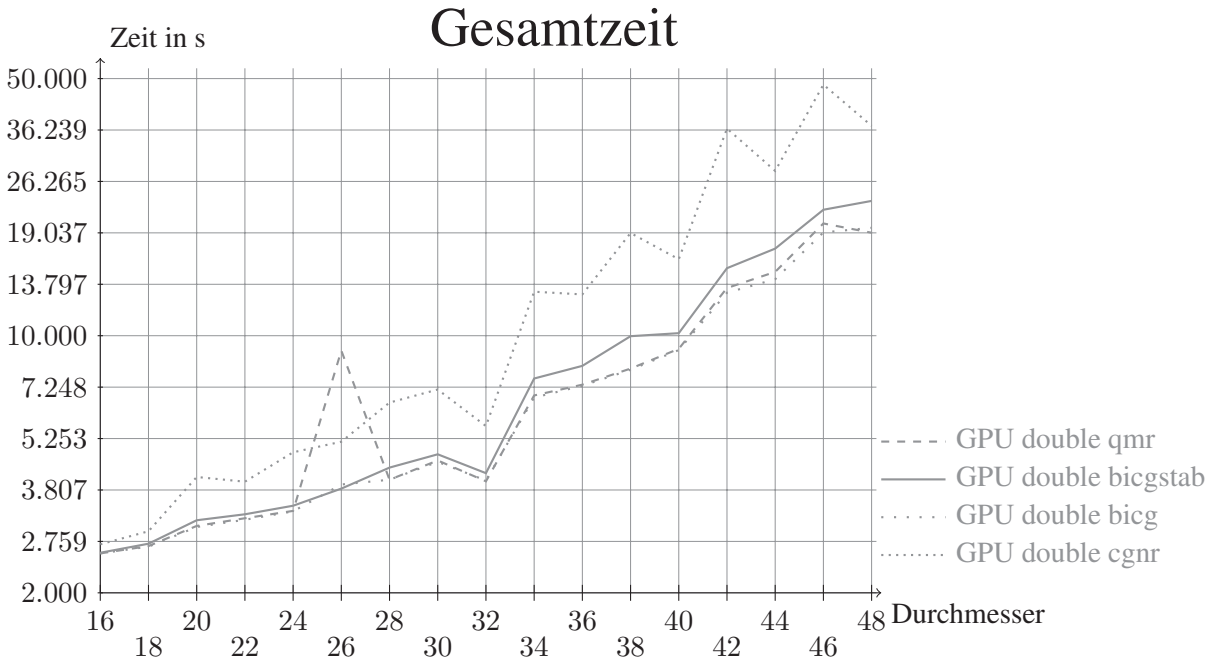

Abbildung 5. Performance Lösungsverfahren, Gesamtzeit

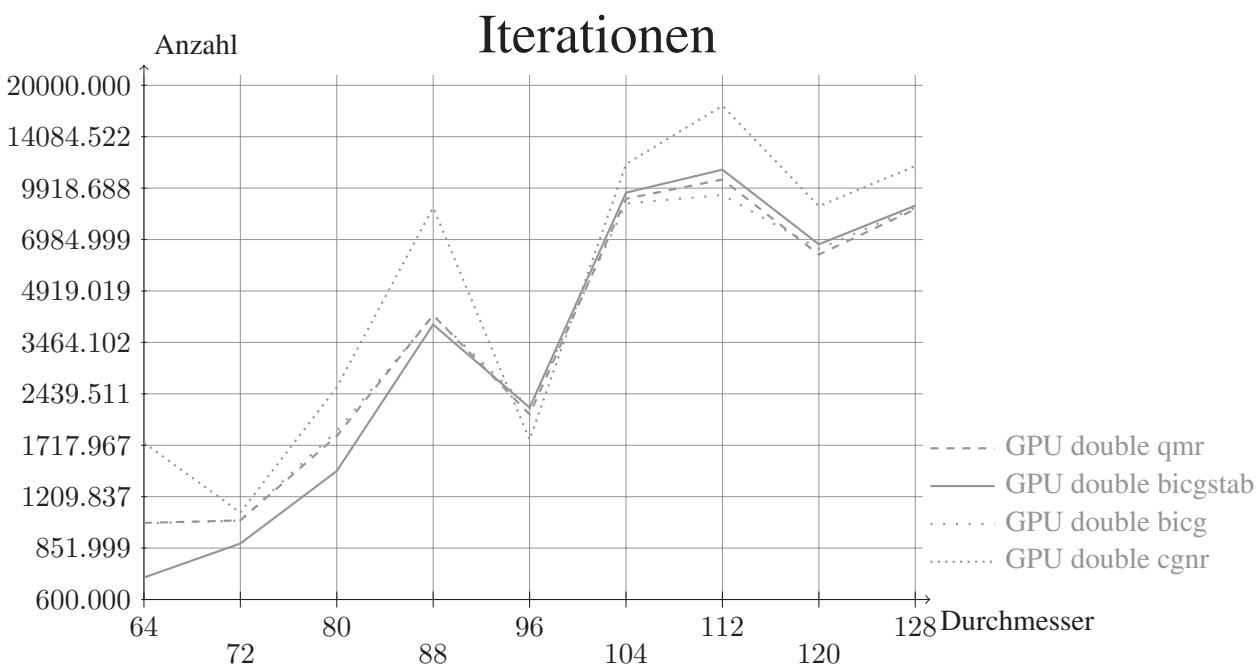

Abbildung 6. Performance Lösungsverfahren, Zahl der Iterationen

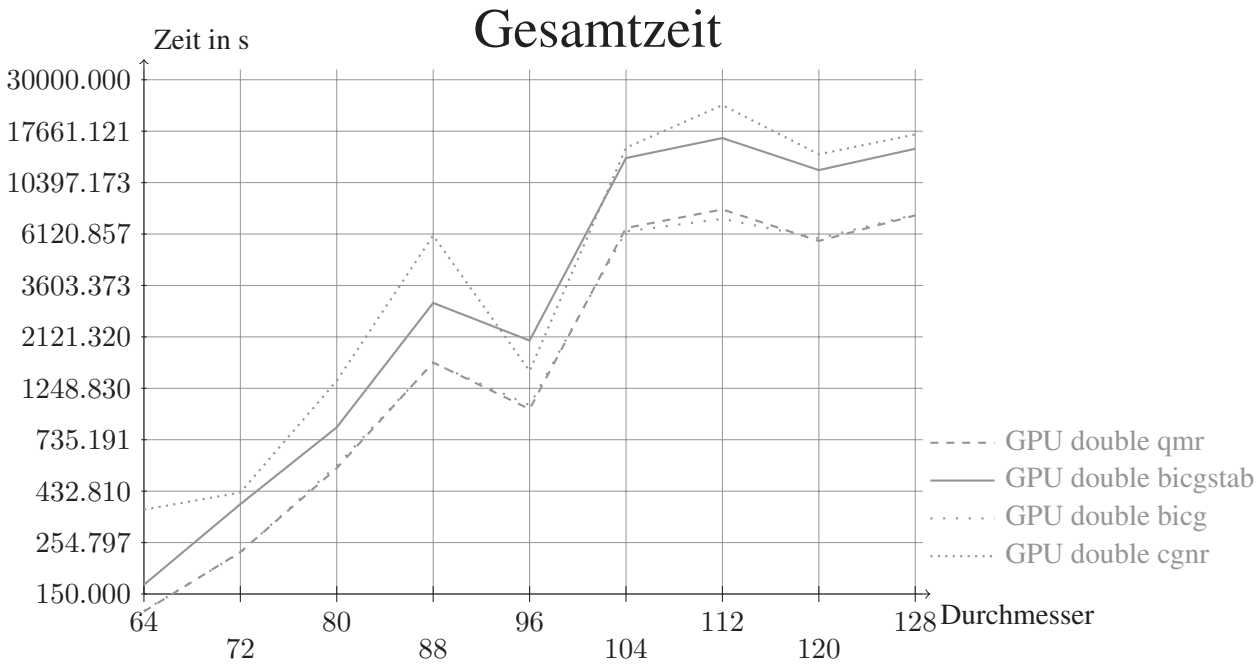

Abbildung 7. Performance Lösungsverfahren, Gesamtzeit

4.2 Genauigkeit der Gleitkommazahlen

Bei der Berechnung der DDA kann man Gleitkommazahlen mit unterschiedlicher Genauigkeit verwenden. Eine geringere Genauigkeit hat dabei den Vorteil, dass die einzelnen Berechnungen schneller sind, während eine größere Genauigkeit im Allgemeinen mit weniger Iterationen auskommt und auch numerisch instabilere Verfahren zulässt.
Auf der CPU kann hierbei eine Genauigkeit von 32 bit (float), 64 bit (double) oder 80 bit (long double) verwendet werden. Auf der GPU stehen 32 und 64 bit zur Verfügung.
In Abbildung 8 ist die benötigte Zeit pro Iteration auf der CPU zu sehen. Mit 32-bit-Genauigkeit ist der Code etwas schneller als mit 64-bit-Genauigkeit. 80-bit-Genauigkeit ist mit Abstand am langsamsten. Der Grund hierfür liegt darin, dass 32-bit-Genauigkeit und 64-bit-Genauigkeit auf der SSE-Einheit ausgeführt werden, während 80-bit-Genauigkeit auf der x87-Einheit ausgeführt werden muss. Die SSE-Einheit kann mehrere Werte gleichzeitig verarbeiten (4 bei 32-bit-Genauigkeit und 2 bei 64-bit-Genauigkeit) und ist auf modernere Prozessoren ausgelegt, während die x87-Einheit älter ist und immer nur einen Wert verarbeiten kann.
In Abbildung 9 ist die Zahl der benötigten Iterationen zu sehen. Diese ist bei größerer Genauigkeit kleiner.
In Abbildung 10 ist die benötigte Gesamtzeit zu sehen. Zu Erkennen ist, dass sich 80-bit-Genauigkeit

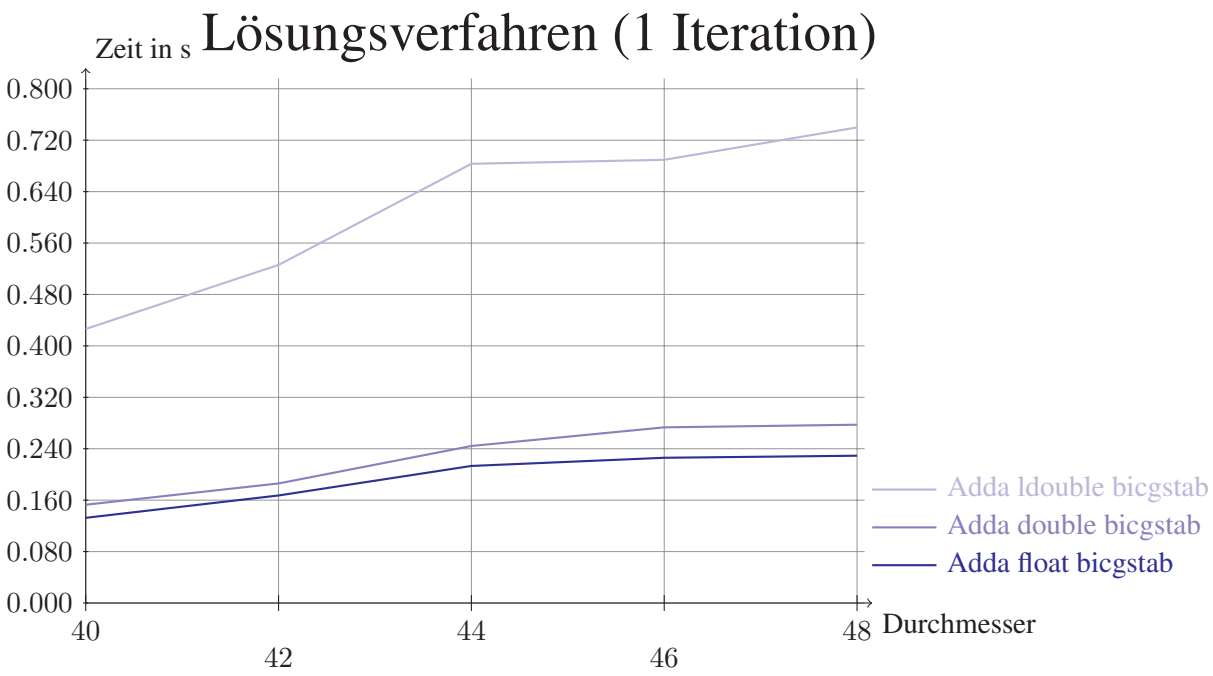

Abbildung 8. Performance Genauigkeit, Zeit pro Iteration

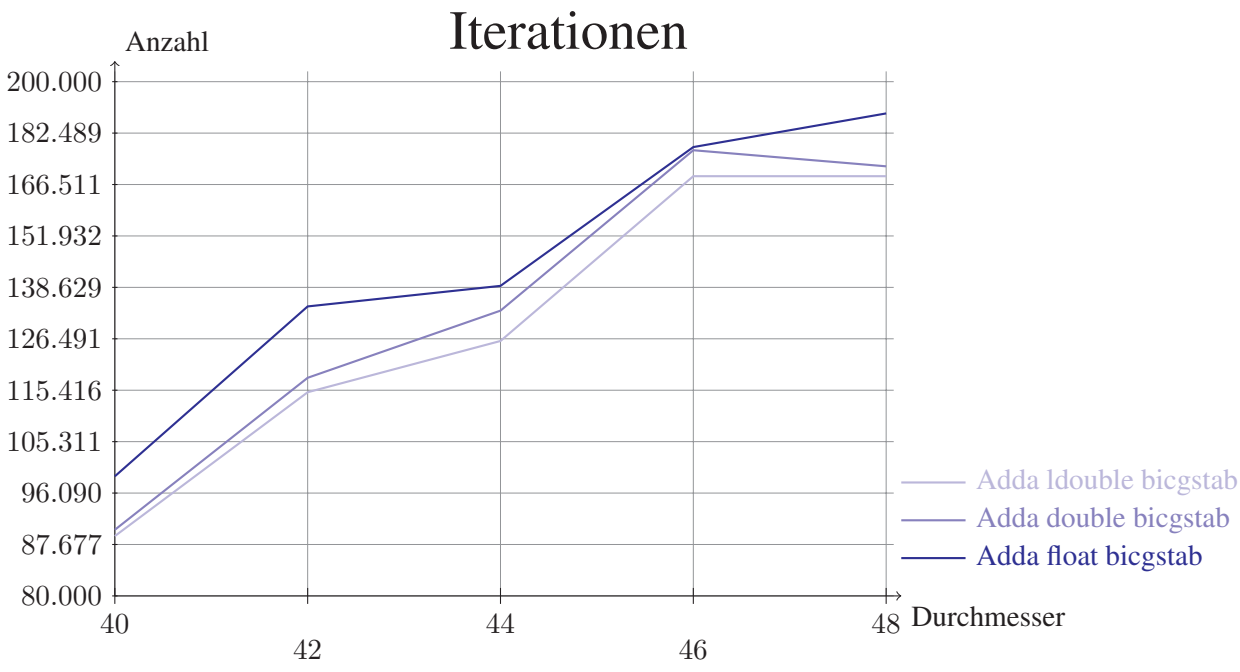

Abbildung 9. Performance Genauigkeit, Zahl der Iterationen

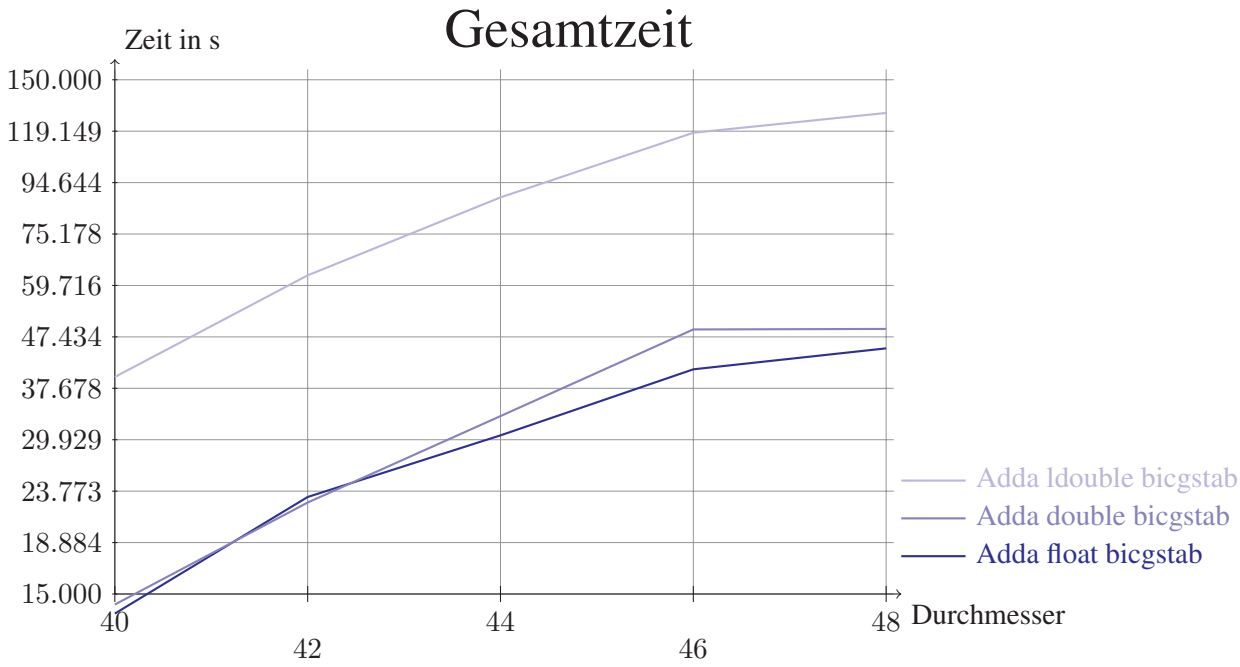

Abbildung 10. Performance Genauigkeit, Gesamtzeit

unabhängig von der Größe des Partikels nicht lohnt. 32-bit-Genauigkeit und 64-bit-Genauigkeit haben vergleichbare Performance, für große Partikel (ab einen Durchmesser von etwa 100 Dipolen) ist 64-bit-Genauigkeit schneller.
Die Algorithmen Bi-CG CS und QMR CS ergeben für Kugelgrößen ab etwa 64 Dipolen keine Ergebnisse mehr, da das iterative Lösungsverfahren nicht mehr konvergiert.

4.3 GPU vs. ADDA

Das Ziel dieser Arbeit war es, die DDA durch Umsetzung auf Grafik-Hardware zu beschleunigen. In diesem Kapitel wird nun die erstellte Implementierung mit ADDA verglichen.
In einigen der folgenden Graphen fehlen teilweise Werte für bestimmte Algorithmen und Kugelgrößen. Dies liegt daran, dass der betreffende Algorithmus entweder nicht mehr konvergiert (z.B. bei Bi-CG CS und QMR CS mit einfacher Genauigkeit) oder dass der entsprechende Lauf zu lange gedauert hat (mehr als 10 Stunden bei den Kugelgrößen 64 bis 128).
In Abbildung 11 ist ein Vergleich der Performance zwischen der GPU-Implementierung und ADDA zu sehen. Abbildung 12 zeigt die benötigten Zeiten für einige größere Kugeln.
Hierbei ist zu sehen, dass die GPU-Implementierung immer deutlich schneller ist als ADDA. Die Schwankungen in Abhängigkeit der Kugelgröße sind größtenteils unabhängig vom Algorithmus,

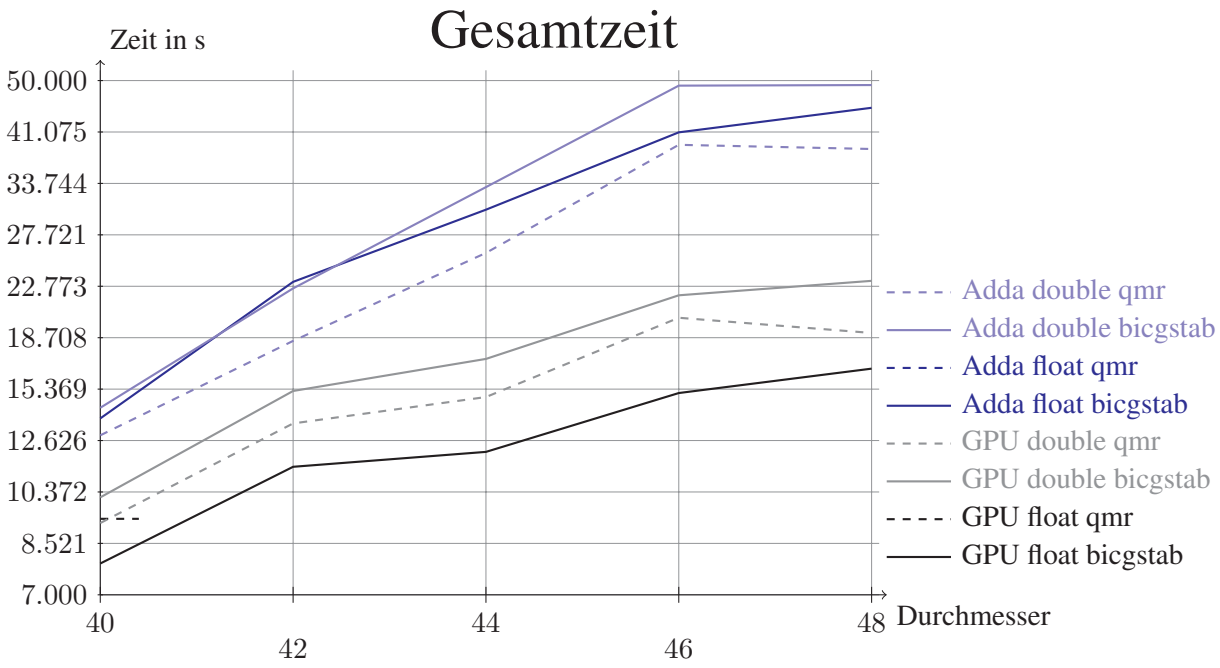

Abbildung 11. Performance GPU vs. CPU klein, Gesamtzeit

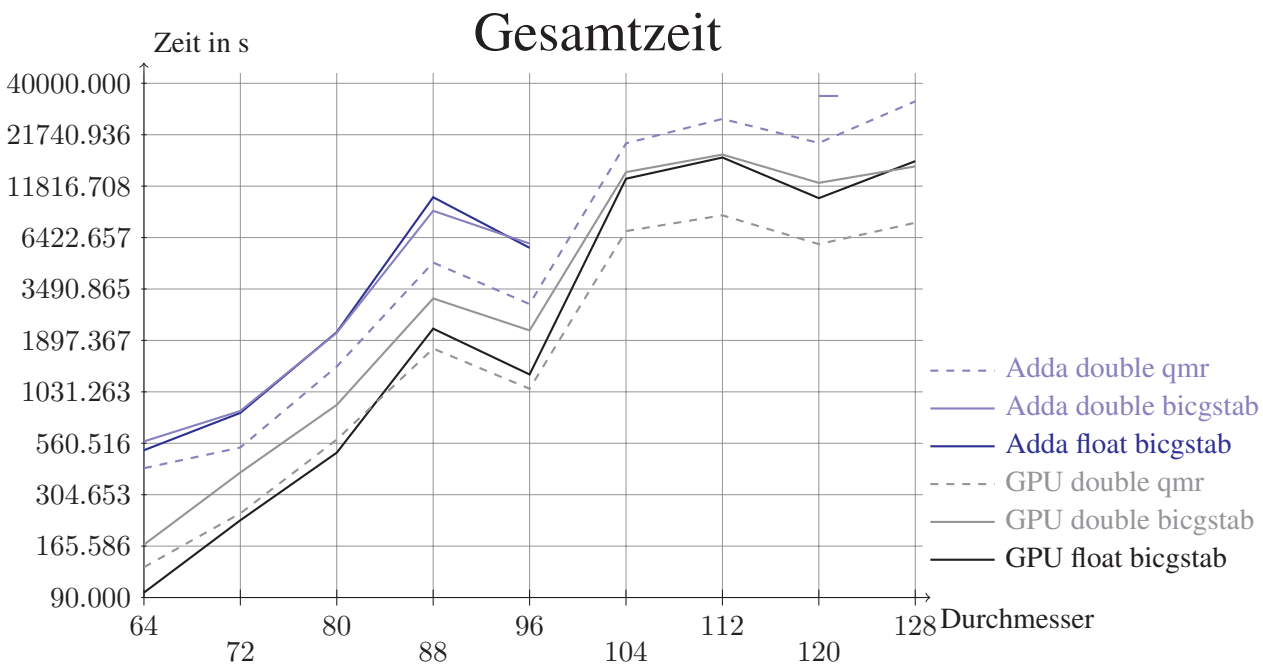

Abbildung 12. Performance GPU vs. CPU groß, Gesamtzeit

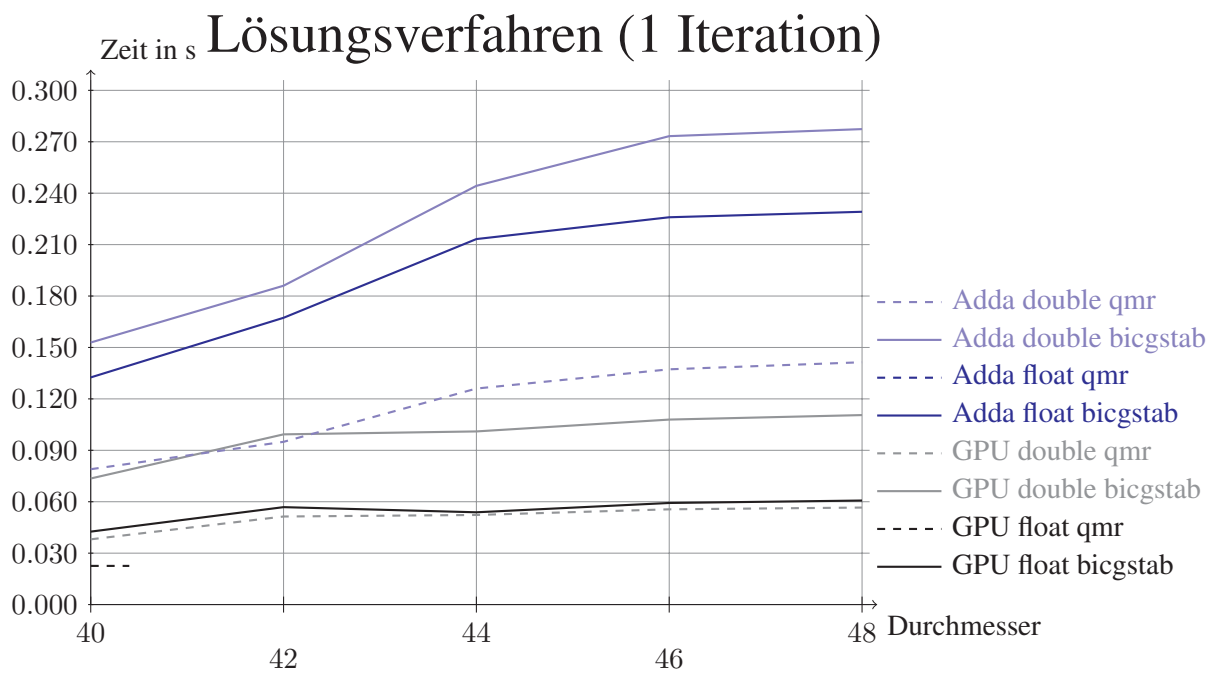

Abbildung 13. Performance GPU vs. CPU klein, Zeit pro Iteration

der Genauigkeit und davon, ob die CPU-Implementierung verwendet wird. Für ADDA fehlen ab der Größe 96 die Werte für Bi-CG STAB (außer bei doppelter Genauigkeit bei Größe 120), da die benötigte Gesamtzeit 10 Stunden überschritt. Für Größen bis etwa 80 ist bei den GPU-Algorithmen Bi-CG STAB mit einfacher Genauigkeit der schnellste Algorithmus, ab dann ist aufgrund größerer numerischer Probleme QMR CS mit doppelter Genauigkeit der beste Algorithmus.
In Abbildung 13 und Abbildung 14 ist die benötigte Zeit pro Iteration zu sehen, in Abbildung 15 und Abbildung 16 die Anzahl der benötigten Iterationen.
Erkennbar ist hier, dass die benötigte Zeit bei einer Iteration bei der GPU-Variante bei Zweierpotenzen teilweise sogar geringer ist als bei kleineren Größen, die keine Zweierpotenzen darstellen. (Beispielsweise ist die Zeit pro Iteration bei Größe 128 kleiner als bei 120 Dipolen.) Dies liegt daran, dass die verwendete FFT-Implementierung mit Zweierpotenzen besonders gute Performance zeigt.
In Abbildung 17 und Abbildung 18 ist die benötigte Zeit der schnellsten ADDA-Variante geteilt durch die benötigte Zeit der schnellsten GPU-Variante dargestellt.
Hier sieht man auch wieder, dass die GPU-Implementierung bei Zweierpotenzen (64 und 128) einen größeren Performance-Vorsprung hat als bei anderen Größen.
Insgesamt ist die GPU-Implementierung für große Partikel etwa dreimal bis viermal so schnell wie ADDA.

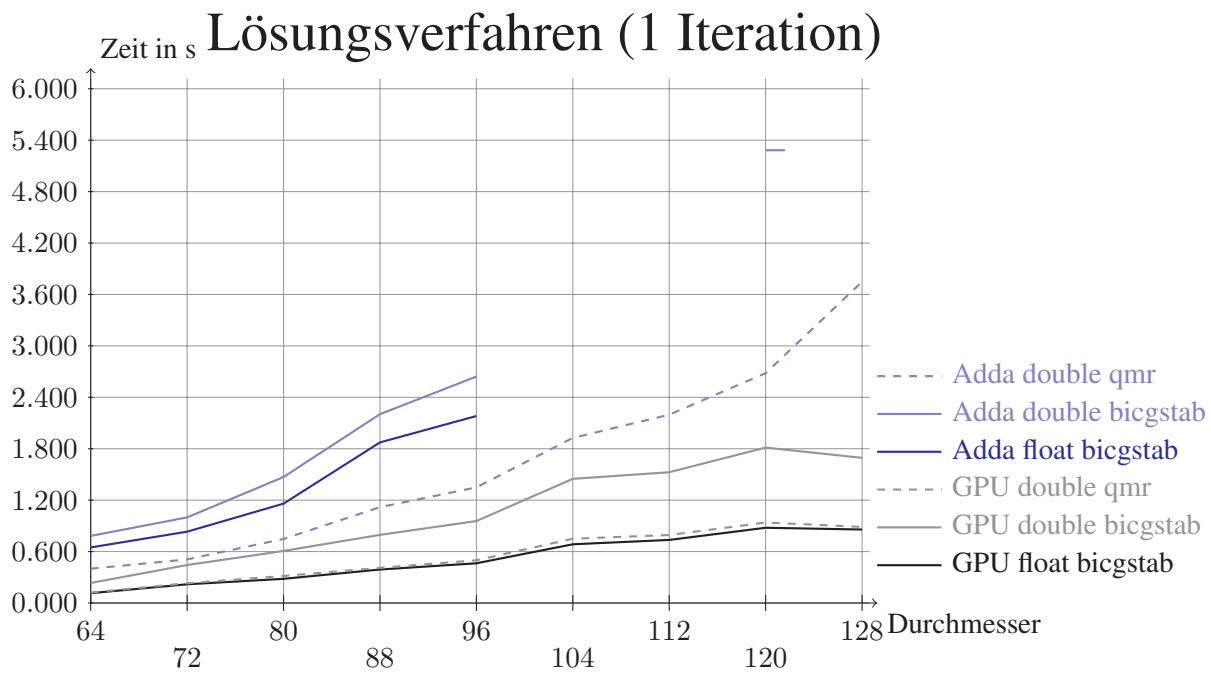

Abbildung 14. Performance GPU vs. CPU groß, Zeit pro Iteration

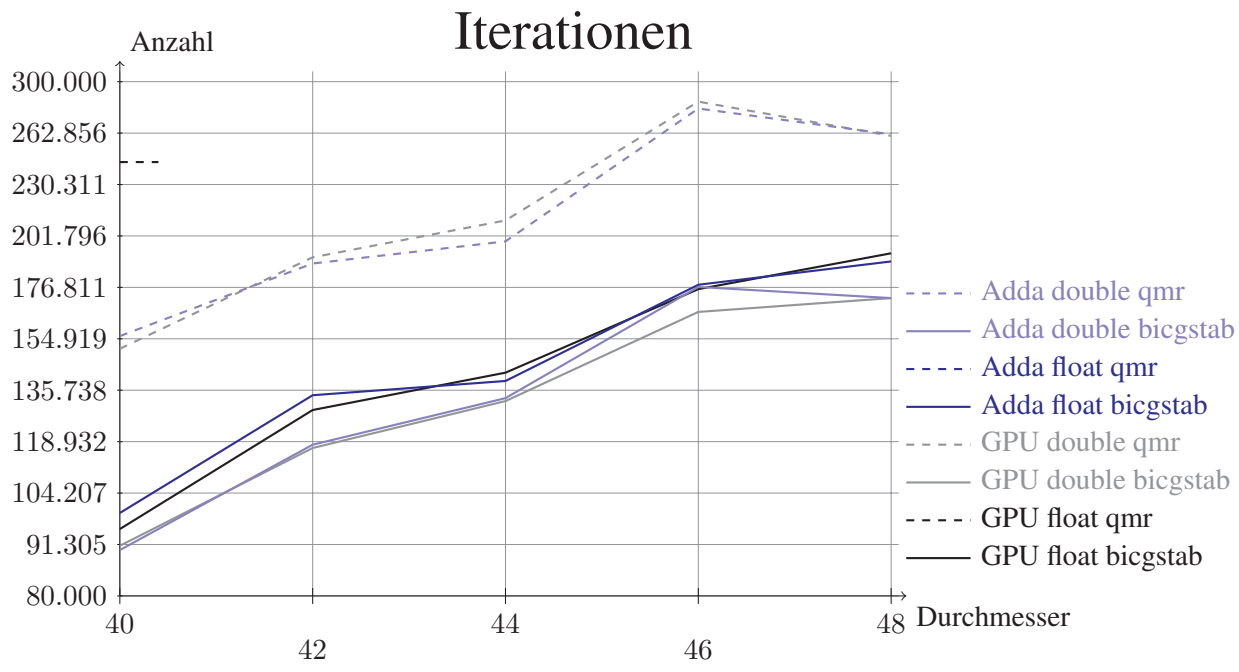

Abbildung 15. Performance GPU vs. CPU klein, Zahl der Iterationen

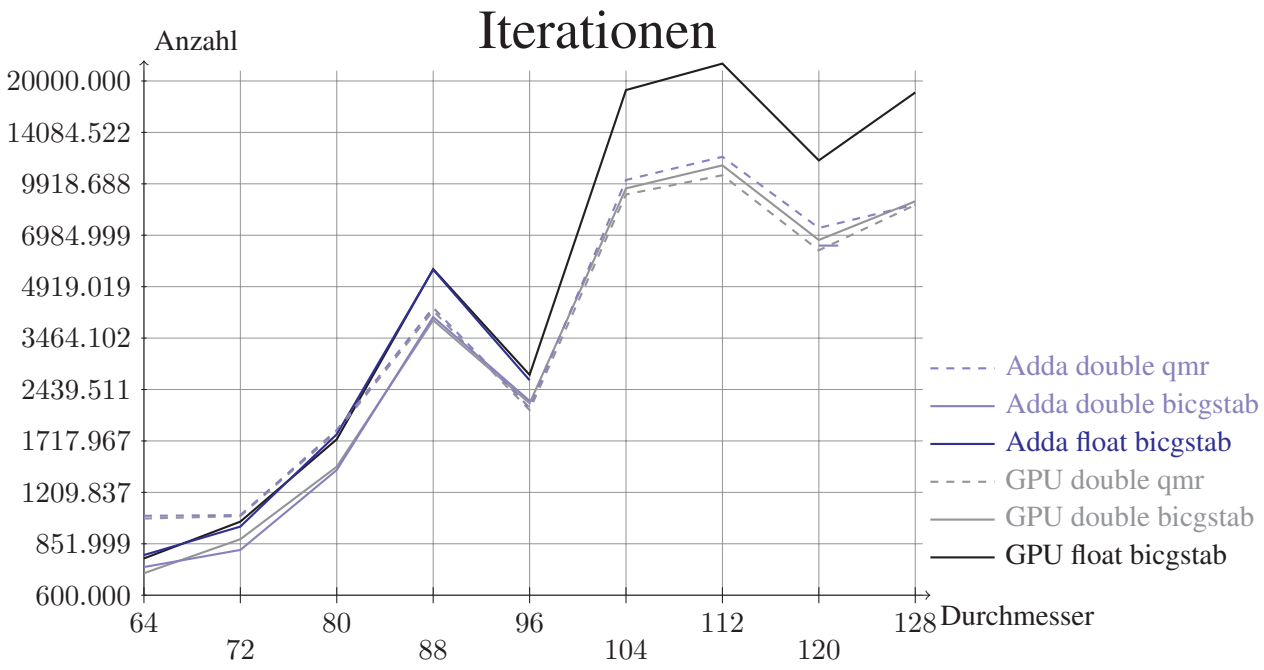

Abbildung 16. Performance GPU vs. CPU groß, Zahl der Iterationen

Abbildung 17. Performance GPU vs. CPU klein, Beschleunigungsfaktor

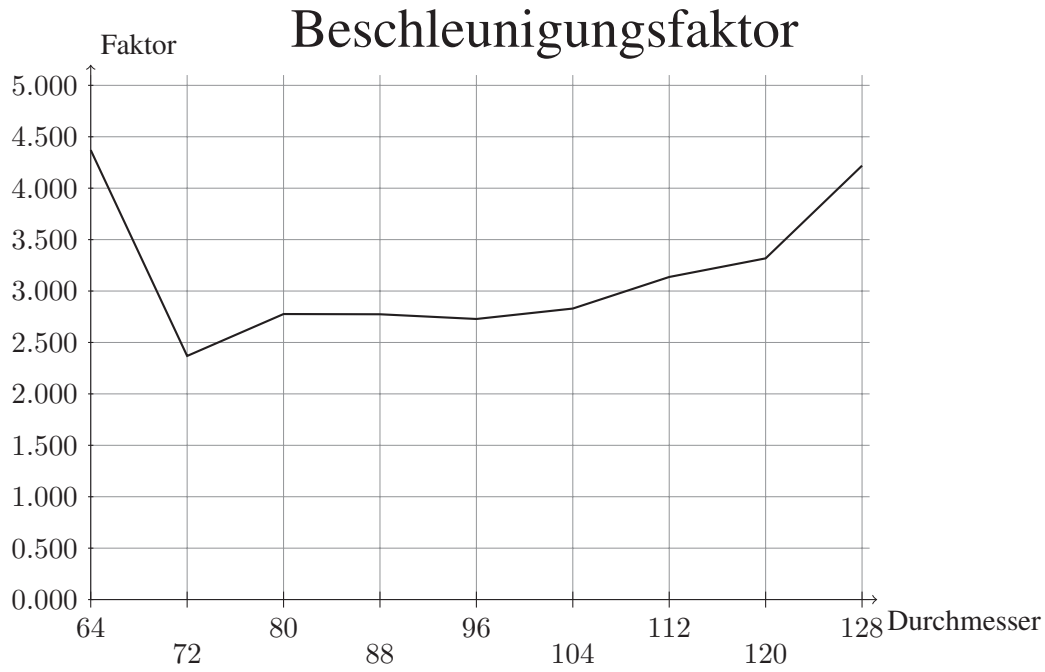

Abbildung 18. Performance GPU vs. CPU groß, Beschleunigungsfaktor

4.4 Iterationen GPU vs. ADDA

Im Allgemeinen wird auf der GPU und auf der CPU eine ähnliche Anzahl an Iterationen benötigt, wenn die verwendete Genauigkeit und das verwendete Lösungsverfahren dasselbe ist.
Für große Partikel ist allerdings teilweise die Zahl der benötigten Iterationen auf der GPU etwas geringer als mit ADDA auf der CPU. Ein Vergleich der Werte kann in Abbildung 19 gesehen werden.
Der Grund hierfür liegt darin, dass das Skalarprodukt und die euklidische Norm auf der GPU parallel berechnet werden. Bei diesen Operationen muss eine eine große Anzahl von komplexen Zahlen addiert werden. ADDA macht dies mit einer einfachen `for`-Schleife, das heißt es wird das folgende berechnet:

$$\sum_i a_i = (((((a_0 + a_1) + a_2) + a_3) + a_4) + a_5 + \dots) \tag{25}$$

Auf der GPU werden von verschiedenen Threads parallel Werte aufsummiert und am Ende werden die Ergebnisse der einzelnen Threads zusammengezählt:

$$\sum_i a_i = [[[(((a_0{+}a_{16}){+}a_{32}){+}\dots){+}(((a_1{+}a_{17}){+}a_{33}){+}\dots)]{+}(((a_2{+}a_{18}){+}a_{34}){+}\dots)]{+}\dots] \tag{26}$$

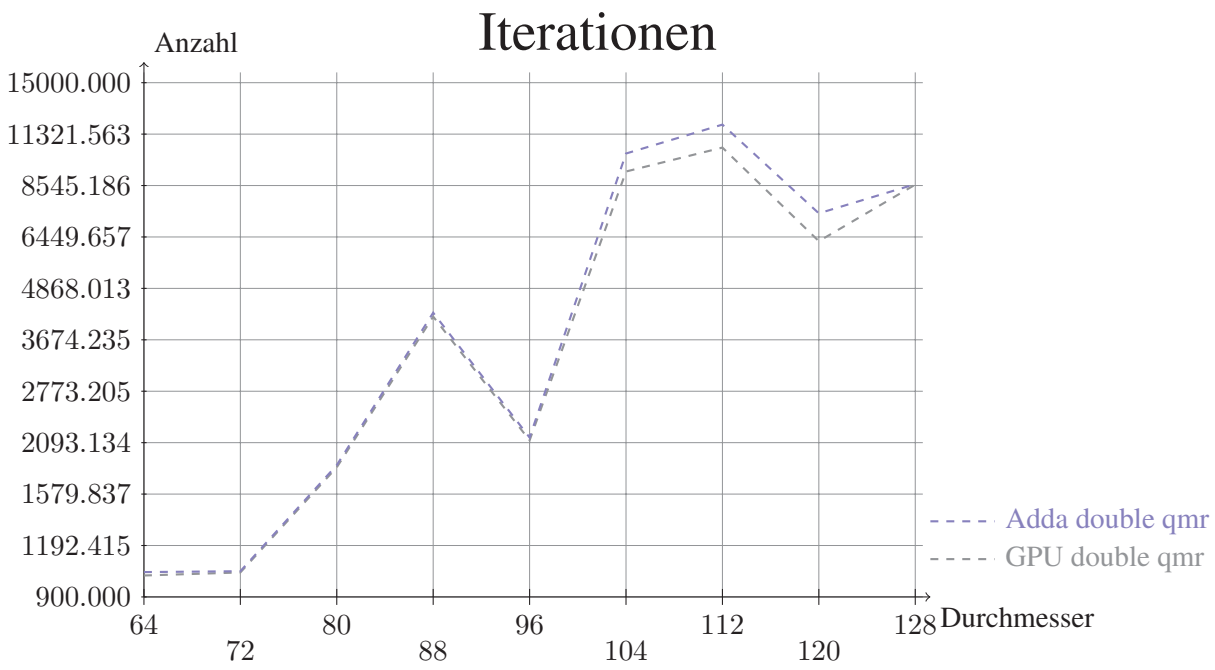

Abbildung 19. Vergleich Zahl der Iterationen GPU vs. ADDA

Da hierbei die Ergebnisse einer einzelnen Addition im Durchschnitt wesentlich geringer sind und die Rundungsfehler bei einer Addition von der Größe des Ergebnisses abhängen, ist die parallele Version numerisch günstiger und liefert genauere Ergebnisse. Diese genaueren Ergebnisse bewirken, dass das Lösungsverfahren nach weniger Iterationen die gewünschte Genauigkeit erreicht hat.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Ziel dieser Arbeit war es, durch Implementierung der DDA auf Grafik-Hardware einen Geschwindigkeitsgewinn zu erzielen.
Nach Einarbeitung in die theoretischen Grundlagen zur Lichtstreuung und die Programmierung mit OpenCL war der nächste Schritt, anhand von Entwurfsdokumenten und anhand des Quelltextes ein grundlegendes Verständnis der Funktionsweise von ADDA [add, yur07] zu entwickeln.
Aufbauend darauf wurde eine CPU-Implementierung in C++ geschrieben.
Anhand dieser Implementierung wurde der Algorithmus auf Stellen untersucht, die für die Performance wichtig sind, und nach Möglichkeiten zur Parallelisierung dieser gesucht, wobei insbesondere die Besonderheiten von Grafik-Hardware berücksichtigt wurden. Da die Berechnung des Matrix-Vektor-Produkts die meiste Zeit in Anspruch nimmt wurde dieses auf der GPU implementiert. Um unnötiges Kopieren von Daten zwischen dem Host-RAM und dem Grafikkarten-RAM zu vermeiden wurden außerdem die iterativen Lösungsverfahren auf der GPU implementiert.
Zur Kontrolle des Performancegewinns wurde die von der GPU-Implementierung benötigte Zeit mit der von ADDA benötigten Zeit verglichen.
Auf dem eingesetzten Testsystem war die GPU-Implementierung etwa dreimal bis viermal so schnell wie ADDA.

5.2 Weitere Entwicklung

Bei der GPU-Implementierung der DDA gibt es noch weitere Forschungs- und Entwicklungsmöglichkeiten.
Einer der Punkte, bei denen noch eine weitere Entwicklung möglich ist, ist die Erweiterung der Implementierung auf den Betrieb auf mehreren Grafikkarten gleichzeitig. Dies kann beispielsweise analog zur MPI-Implementierung von ADDA durchgeführt werden.
Eine weitere Möglichkeit wäre es, eine Option für einen verteilten Betrieb auf mehreren Rechnern zu implementieren. Hierbei ist es notwendig, zu untersuchen, ob sich dies trotz des erheblich gesteigerten Kommunikationsaufwands noch lohnt.
Außerdem kann die Implementierung noch um Funktionen erweitert werden, die bisher nur in ADDA vorhanden sind.
Weitere Optimierungspotentiale gibt es noch auf algorithmischer Ebene, beispielsweise kann die Möglichkeit der Ausnutzung von Periodizitäten des Partikels untersucht werden.

A Verwendete Notation

Symbol	Dimension	Bedeutung
e	Skalar	Eulersche Zahl, $e = \sum_{s=0}^{\infty} \frac{1}{s!}$
i	Skalar	Imaginäre Einheit, $i^2 = -1$
$\Re(x)$		Realteil von x
$\Im(x)$		Imaginärteil von x
$\overline{x}$		Zu x komplex konjugierte Zahl
I	Matrix	Einheitsmatrix
I_n	nxn-Matrix	nxn-Einheitsmatrix
M^*	Matrix	Zu M adjungierte Matrix, $M^*{}_{s,t} = \overline{M_{t,s}}$
M^T	Matrix	Zu M transponierte Matrix, $M^T{}_{s,t} = M_{t,s}$
$\langle a, b \rangle$	Skalar	Skalarprodukt von a und b, $\langle a, b \rangle = \sum_s a_s \overline{b_s}$
$a * b$	Vektor	Elementweises Produkt von a und b, also $\mathrm{diag}(ab^T)$
$\|a\|$	Skalar	Betrag des Vektors a, $\|a\| = \sqrt{\langle a, a \rangle}$
a_u	Vektor	Normalisierter Vektor a, $a_u = a/\|a\|$
N	Skalar	Anzahl der Dipole
j, l	Skalar	Indizes für Dipole
v	Skalar	Geschwindigkeit des Lichts im umgebenden Medium
f	Skalar	Frequenz des einfallenden Lichtstrahls
ω	Skalar	Kreisfrequenz des einfallenden Lichtstrahls ($\omega = 2\pi f$)
λ	Skalar	Wellenlänge des einfallenden Lichtstrahls im umgebenden Medium ($\lambda = v/f$)
k	Skalar	Kreiswellenzahl des einfallenden Lichtstrahls im umgebenden Medium ($k = 2\pi f/v = 2\pi/\lambda = \omega/v$)
P_j	3D-Vektor	Induzierte elektrische Polarisation des Dipols j
$E_{inc,j}$	3D-Vektor	Elektrisches Feld des einfallenden Lichtstrahls am Dipol j
$E_{j,l}$ für $l \neq j$	3D-Vektor	Elektrisches Feld, das der Dipol l am Dipol j erzeugt
E_j	3D-Vektor	Am Dipol j anliegendes elektrisches Feld
α_j	3x3-Matrix	Polarisierbarkeits-Tensor des Dipols j
$A_{j,l}$ für $l \neq j$	3x3-Matrix	Negierte Matrix, die die Abhängigkeit des vom Dipol l am Dipol j erzeugten elektrischen Feldes von der elektrischen Polarisation P_l des Dipols l angibt
$A_{j,j}$	3x3-Matrix	α_j^{-1}

Tabelle 1: Verwendete Notation

Symbol	**Dimension**	**Bedeutung**
$D_{j,l}$ für $l \neq j$	3x3-Matrix	Negierte Matrix, die die Abhängigkeit des vom Dipol l am Dipol j erzeugten elektrischen Feldes von der elektrischen Polarisation P_l des Dipols l angibt
$D_{j,j}$	3x3-Matrix	0
S_j	3x3-Matrix	$\sqrt{\alpha_j}$
$E_{sca,x}$	3D-Vektor	Gestreutes E-Feld am Punkt x
x_j	3D-Vektor	Argument der Matrix-Vektor-Multiplikation
y_j	3D-Vektor	Ergebnis der Matrix-Vektor-Multiplikation
a_j, b_j, c_j	3D-Vektor	Zwischenergebnisse der Matrix-Vektor-Multiplikation

Tabelle 1: Verwendete Notation

B Verwendete Begriffe

Deutsch	**Englisch**
Multilevel-Block-Toeplitz-Matrix	multilevel block-Toeplitz matrix
zeilenweise Ordnung	row-major order
iteratives Lösungsverfahren	iterative solver
Matrix-Transposition	matrix transpose
verschmolzener Speicherzugriff	coalesced memory access
symbolische Verknüpfung	symbolic link (or symlink)
(elektrische) Polarisation	polarization density, (electric) polarization
Restvektor	residual vector
Polarisationsvektor	polarization vector
Auslöschungsquerschnitt	extinction cross section
Absorptionsquerschnitt	absorption cross section
Streuungsquerschnitt	scattering cross section
globaler Speicher	global memory
Konstanten-Speicher	constant memory
lokaler Speicher	local memory
privater Speicher	private memory

Tabelle 2: Verwendete Begriffe

C Bedienungsanleitung

Dieses Kapitel erläutert die Verwendung der in dieser Arbeit vorgestellten Implementierung.
Die Bedienung der Implementierung geschieht über die Kommandozeile. Die Optionen für das Programm müssen im GNU-Stil übergeben werden.
Nach dem Parsen den Kommandozeilen-Optionen erzeugt das Programm ein Ausgabeverzeichnis (siehe Kapitel C.6) und schreibt dorthin eine Logdatei und die Ergebnisse der Berechnung.

C.1 Allgemeine Optionen

C.1.1 `--help`

Zeige die verfügbaren Optionen mit Erläuterung an.
Wird diese Option angegeben, dann wird das restliche Programm nicht ausgeführt.

C.2 Geometrie des Partikels

Die Geometrie des Partikels kann entweder aus einer Datei gelesen werden oder es kann eine Kugel erzeugt werden.
Beim Lesen aus einer Datei wird das gleiche Dateiformat wie bei ADDA verwendet (siehe Kapitel B.5 in [yur10]). Hierbei wird eine Liste von Dipolen mit Position und einem Materialindex angegeben. Zur Angabe des Brechungsindex für die Materialien siehe Kapitel C.3.1.
Bei einer Kugel muss der Durchmesser in Dipolen angegeben werden.
Wird keine Option für die Geometrie angegeben so nimmt das Programm eine Kugel mit einem Durchmesser von 16 Dipolen.

C.2.1 `--shape-file [dateiname]`

Lese die Geometrie aus `dateiname` aus.

C.2.2 `--shape-sphere-size [durchmesser]`

Nimm eine Kugel mit einem Durchmesser von `durchmesser` Dipolen.

C.2.3 `--save-shape-file [dateiname]`

Speichere die verwendete Geometrie in `dateiname`.
Diese Option ist beispielsweise bei der Verwendung von `--shape-sphere-size` nützlich.
Die Datei kann später mit `--shape-file` wieder geladen werden.

C.3 Material und Größe des Partikels

Außer der Geometrie des Partikels muss noch für jeden verwendeten Materialindex ein Brechungsindex angegeben werden und es muss der Abstand zwischen zwei Dipolen angegeben werden.

C.3.1 `--m [brechungsindex]`

Gibt den Brechungsindex für ein Material an. Werden mehrere Materialien verwendet muss diese Option entsprechend oft angegeben werden.
`brechungsindex` hat dabei die Form $((\Re(m_x), \Im(m_x)), (\Re(m_y), \Im(m_y)), (\Re(m_z), \Im(m_z)))$ wobei $m_{x/y/z}$ jeweils den Brechungsindex in x/y/z-Richtung mit Realteil und Imaginärteil angibt. Beispielsweise hat `((1.5,0.1),(1.5,0.1),(1.6,0))` in x- und y-Richtung einen Brechungsindex mit einem Realteil von 1.5 und einem Imaginärteil von 0.1 und in z-Richtung einen Realteil von 1.6 und einem Imaginärteil von 0.
Ist keine `--m`-Option angegeben wird von einem Material mit einem Brechungsindex von 1.5 ausgegangen.

C.3.2 `--lambda [laenge]`

Gibt die Wellenlänge des einfallenden Lichts in μm an.
Der Standardwert ist 2π.

C.3.3 `--grid-unit [distanz]`

Gibt an, dass zwei Dipole eine Distanz von `distanz` μm haben.
Der Standardwert ist abhängig von `--lambda` und `--m`.

C.3.4 `--no-symmetry`

Ignoriere Symmetrien im Partikel.
Normalerweise wird die Rotationssymmetrie bei Kugeln ausgenutzt, um das Lösungsverfahren nur einmal anstatt zweimal für zwei Polarisationsrichtungen ausführen zu müssen. Mit dieser Option wird dieses Verhalten abgeschaltet und es werden immer beide Polarisationsrichtungen berechnet.

C.4 Lösungsverfahren und Genauigkeit

Mit diesen Optionen kann das verwendete Iterative Lösungsverfahren und die verwendete Genauigkeit angegeben werden. Außerdem kann ausgewählt werden, ob die Lösung auf der GPU oder auf der CPU berechnet werden soll.

C.4.1 `--iter [solver]`

Gibt das verwendete iterative Lösungsverfahren an (siehe Kapitel 2.3).
Mögliche Werte sind `qmr` für QMR CS, `cgnr` für CGNR, `bicg` für Bi-CG und `bicgstab` für Bi-CGSTAB.
Das standardmäßig verwendete Verfahren ist QMR CS.

C.4.2 `--ftype [genauigkeit]`

Gibt die verwendete Genauigkeit an.
Mögliche Werte sind `float` für einfache (32-bit) Genauigkeit, `double` für doppelte (64-bit) Genauigkeit und `ldouble` für 80-bit-Genauigkeit.
Auf der GPU kann 80-bit-Genauigkeit nicht verwendet werden. Doppelte Genauigkeit kann nur verwendet werden, wenn die Grafikkarte dies unterstützt (bei NVIDIA alle Grafikkarten mit einer Compute Capability ab 1.3, d.h. z.B. alle Fermi-Grafikkarten).
Standardmäßig wird doppelte Genauigkeit verwendet.

C.4.3 `--cpu`

Lässt die Berechnung auf der CPU ablaufen.
Diese Option ist nur zum Testen gedacht; sie liefert wesentlich schlechtere Performance als auf der GPU oder mit ADDA.

C.4.4 `--opencl`

Lässt die Berechnung mit OpenCL auf der GPU ablaufen.
Wenn weder `--cpu` noch `--opencl` angegeben sind wird `--opencl` angenommen.

C.4.5 `--grid [groesse]`

Gibt die Größe des erweiterten Gitters an (siehe Kapitel 3.4.2).
Dieses Gitter muss in jeder Richtung mindestens doppelt so groß sein wie das Partikel, allerdings lohnt sich unter Umständen, das Gitter noch größer zu machen, um eine für die FFT günstigere Größe zu erreichen.
Standardmäßig wird das Gitter in jeder Richtung so weit vergrößert, dass die Größe nur noch die Primfaktoren 2, 3, 5 und 7 enthält.

C.4.6 `--epsilon [wert]`

Gibt den Wert für das Abbruchkriterium ϵ des iterativen Lösungsverfahrens an (siehe Kapitel 3.3).
Je kleiner dieser Wert ist, desto genauer ist das Ergebnis und desto länger braucht das Verfahren.

Der Standardwert ist $\epsilon = 10^{-5}$.

C.5 OpenCL

Das Verhalten der OpenCL-Implementierung kann mit einigen Optionen genauer gesteuert werden.

C.5.1 `--device [device]`

Wählt das verwendete OpenCL-Device aus.
Wenn `list` als `device` angegeben wird wird eine Liste der verfügbaren OpenCL-Devices ausgegeben und die Anwendung beendet sich, ohne eine Berechnung durchzuführen.
Mit `cpu` wird automatisch das erste verfügbare CPU-Device ausgewählt, mit `gpu` das erste verfügbare GPU-Device. `auto` wählt automatisch die erste verfügbare Grafikkarte, oder, wenn dies keine vorhanden ist, das erste verfügbare CPU-Device aus.
Mit dieser Option kann ein CPU-Device für OpenCL ausgewählt werden (z.B. das ATI Stream SDK[5]), aber da die OpenCL-Implementierung für GPUs entwickelt wurde, ist die Performance hierbei sehr schlecht (einiges schlechter als mit der `--cpu`-Option).
Standardwert ist `auto`.

C.5.2 `--dmatrix-host`

Speichere die D-Matrix im Host-RAM (siehe Kapitel 3.2).
Mit dieser Option wird die D-Matrix im Host-RAM gespeichert und immer nur bei Bedarf auf die GPU kopiert. Dies führt zu geringen Performance-Einbußen, verringert aber den Speicherbedarf auf der GPU.

C.5.3 `--mem-info`

Gebe Informationen über die GPU-Speichernutzung auf der Standardausgabe aus.

C.5.4 `--sync`

Führe nach jeder GPU-Operation einen Aufruf zu `clFinish()` aus.
Mit dieser Option wartet die CPU nach jeder GPU-Operation, bis die GPU-Operation abgeschlossen ist. Dies verschlechtert die Performance, verbessert aber die Aussagekraft von Profiling-Informationen.

[5] `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`

C.5.5 `--blocking-sync`

Verwende für `clFinish()` einen blockierenden Systemaufruf.
Diese Option wiest die OpenCL-Implementierung an, beim Warten auf die GPU einen blockierenden Systemaufruf anstatt Busy Waiting zu verwenden.
Diese Option funktioniert nur mit der NVIDIA-Implementierung.

C.6 Ausgabe

Das Programm schreibt eine Logdatei und die Ergebnisse in das Ausgabeverzeichnis.
Nach der Ausführung befinden sich in diesem Verzeichnis folgende Dateien:

- **log**
  Eine Logdatei mit Informationen über die Eingabeparameter und den Ablauf der Berechnung
- **meminfo**
  Eine Datei mit Informationen über die GPU-Speichernutzung
- **prof**
  Eine Datei mit Profiling-Informationen
- **CrossSec**
  Eine Datei mit Absorptions- und Auslöschungsquerschnitten
- **DipPol**
  Die elektrische Polarisation P_j der einzelnen Dipole
- **IntField**
  Die an den einzelnen Dipolen anliegenden elektrischen Felder E_j
- **efield** (wenn `--efield` angegeben wurde)
  Das elektrischen Fernfeld
- **mueller** (wenn `--mueller-matrix` angegeben wurde)
  Die Müller-Matrix

Die Ausgabedateien sind zu den Ausgabedateien von ADDA kompatibel.

C.6.1 `--output-dir [verzeichnis]`

Gibt das zu verwendende Ausgabeverzeichnis an.
Wird kein Verzeichnis angegeben, so wird automatisch ein Verzeichnis mit einer fortlaufenden

Nummer im Unterverzeichnis `output.log` des Verzeichnisses, in dem sich das Programm befindet, erstellt. Auf Unix-Systemen wird außerdem die symbolische Verknüpfung `output.log/latest` aktualisiert, so dass sie auf dieses Verzeichnis zeigt.

C.6.2 `--efield`

Gib das elektrische Fernfeld in die Datei `efield` aus.
Da das Berechnen dieses Feldes einige Zeit benötigt, ist diese Option standardmäßig abgeschalten.

C.6.3 `--mueller-matrix`

Gib die Müller-Matrix in die Datei `mueller` aus.
Da das Berechnen dieser Matrix einige Zeit benötigt, ist diese Option standardmäßig abgeschalten.

Tabellenverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

[boh98] BOHREN, C.F. und D.R. HUFFMAN: *Absorption and Scattering of Light by Small Particles*, Band 16.
Cambridge University Press, New York, NY, USA, 1998, ISBN 978-0471057727.
Seite 6.

[dev64] DEVOE, HOWARD: *Optical Properties of Molecular Aggregates. I. Classical Model of Electronic Absorption and Refraction*.
The Journal of Chemical Physics, 41(2):393–400, 1964, doi:10.1063/1.1725879.
Seite 6.

[dra94] DRAINE, BRUCE T. und PIOTR J. FLATAU: *Discrete-dipole approximation for scattering calculations*.
J. Opt. Soc. Am. A, 11(4):1491–1499, 1994, doi:10.1364/JOSAA.11.001491.
Seiten 6, 7, 9, 15 und 16.

[pur73] PURCELL, E. M. und C. R. PENNYPACKER: *Scattering and Absorption of Light by Nonspherical Dielectric Grains*.
Astrophys. J., 186:705–714, Dezember 1973, doi:10.1086/152538.
Seite 6.

[yun78] YUNG, YUK L.: *Variational principle for scattering of light by dielectric particles*.
Appl. Opt., 17(23):3707–3709, 1978, doi:10.1364/AO.17.003707.
Seite 6.

[dra88] DRAINE, B. T.: *The discrete-dipole approximation and its application to interstellar graphite grains*.
Astrophys. J., 333:848–872, Oktober 1988, doi:10.1086/166795.
Seiten 6, 9 und 16.

[fla90] FLATAU, PIOTR J., GRAEME L. STEPHENS und BRUCE T. DRAINE: *Light scattering by rectangular solids in the discrete-dipole approximation: a new algorithm exploiting the Block-Toeplitz structure*.
J. Opt. Soc. Am. A, 7(4):593–600, 1990, doi:10.1364/JOSAA.7.000593.
Seiten 6 und 14.

[goo91] GOODMAN, J. J., B. T. DRAINE und P. J. FLATAU: *Application of fast-Fourier-transform techniques to the discrete-dipole approximation*.
Opt. Lett., 16(15):1198–1200, 1991, doi:10.1364/OL.16.001198.
Seiten 7 und 13.

[pen07] PENTTILÄ, ANTTI, EVGENIJ ZUBKO, KARI LUMME, KARRI MUINONEN, MAXIM A. YURKIN, BRUCE DRAINE, JUSSI RAHOLA, ALFONS G. HOEKSTRA und YURI SHKURATOV: *Comparison between discrete dipole implementations and exact techniques*. Journal of Quantitative Spectroscopy and Radiative Transfer, ISSN 0022-4073, 106(1-3):417 – 436, 2007, IX Conference on Electromagnetic and Light Scattering by Non-Spherical Particles, doi:10.1016/j.jqsrt.2007.01.026.
Seite 7.

[yur07] YURKIN, M.A. und A.G. HOEKSTRA: *The discrete dipole approximation: An overview and recent developments*.
Journal of Quantitative Spectroscopy and Radiative Transfer, ISSN 0022-4073, 106(1-3):558 – 589, 2007, IX Conference on Electromagnetic and Light Scattering by Non-Spherical Particles, doi:10.1016/j.jqsrt.2007.01.034.
Seite 7.

[dds] *DDSCAT*, `http://www.astro.princeton.edu/~draine/DDSCAT.html`.
Seite 7.

[dra10] DRAINE, B. T. und P. J. FLATAU: *User Guide for the Discrete Dipole Approximation Code DDSCAT 7.1*, 2010, `http://arxiv.org/abs/1002.1505v1`.
Seite 7.

[add] *ADDA code*, `http://www.science.uva.nl/research/scs/Software/adda/`.
Seiten 7 und 44.

[yur07] YURKIN, M.A., V.P. MALTSEV und A.G. HOEKSTRA: *The discrete dipole approximation for simulation of light scattering by particles much larger than the wavelength.* Journal of Quantitative Spectroscopy and Radiative Transfer, ISSN 0022-4073, 106(1-3):546 – 557, 2007, IX Conference on Electromagnetic and Light Scattering by Non-Spherical Particles, doi:10.1016/j.jqsrt.2007.01.033.
Seiten 7 und 44.

[ope] *OpenDDA*, `http://www.opendda.org/`.
Seite 8.

[mcd07] MC DONALD, JAMES: *OpenDDA: A Novel High-Performance Computational Framework for the Discrete Dipole Approximation*.
Doktorarbeit, National University of Ireland, Galway, Ireland, 2007, `http://www.opendda.org/assets/docs/thesis_JMcD_OpenDDA.pdf`.
Seite 8.

[mcd09] MC DONALD, JAMES, AARON GOLDEN und S. GERARD JENNINGS: *Opendda: a Novel High-Performance Computational Framework for the Discrete Dipole Approximation*.
International Journal of High Performance Computing Applications, 23(1):42–61, 2009, doi:10.1177/1094342008097914.
Seite 8.

[mis00] MISHCHENKO, M I, J W HOVENIER und L D TRAVIS: *Light Scattering by Nonspherical Particles: Theory, Measurements, and Applications*.
Academic Press, 2000, ISBN 0124986609.
Seite 9.

[mis02] MISHCHENKO, M.I., L.D. TRAVIS und A.A. LACIS: *Scattering, Absorption, and Emission of Light by Small Particles*.
Cambridge University Press, 3rd revised electronic Auflage, 2002, `http://pubs.giss.nasa.gov/books/2002_Mishchenko_etal/`.
Seite 9.

[lak92.a] LAKHTAKIA, A.: *General theory of the Purcell-Pennypacker scattering approach and its extension to bianisotropic scatterers*.
Astrophys. J., 394:494–499, August 1992, doi:10.1086/171600.
Seite 9.

[lak92.b] LAKHTAKIA, A.: *Strong and Weak Forms of the Method of Moments and the Coupled Dipole Method for Scattering of Time-Harmonic Electromagnetic Fields*.
International Journal of Modern Physics C, 3:583–603, 1992, doi:10.1142/S0129183192000385.
Seite 9.

[yur10] YURKIN, M.A. und A.G. HOEKSTRA: *User manual for the discrete dipole approximation code ADDA v.1.0*, 2010, `http://a-dda.googlecode.com/svn/tags/rel_1.0/doc/manual.pdf`.
Seiten 11 und 48.

[bar94] BARRETT, R., M. BERRY, T. F. CHAN, J. DEMMEL, J. DONATO, J. DONGARRA, V. EIJKHOUT, R. POZO, C. ROMINE und H. VAN DER VORST: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*.
SIAM, Philadelphia, PA, 2nd Edition Auflage, 1994, `http://www.netlib.org/linalg/html_templates/Templates.html`.
Seite 11.

[bar01] BARROWES, B.E., F.L. TEIXEIRA und J.A. KONG: *Fast algorithm for matrix-vector multiply of asymmetric multilevel block-toeplitz matrices*.
Band 4, Seiten 630 –633 vol.4, 2001, doi:10.1109/APS.2001.959544.
Seite 13.

[zha08] ZHAO, JING, ANATOLIY O. PINCHUK, JEFFREY M. MCMAHON, SHUZHOU LI, LOGAN K. AUSMAN, ARIEL L. ATKINSON und GEORGE C. SCHATZ: *Methods for Describing the Electromagnetic Properties of Silver and Gold Nanoparticles*.
Accounts of Chemical Research, 41(12):1710–1720, 2008, doi:10.1021/ar800028j.
Seite 16.

[khr10] KHRONOS OPENCL WORKING GROUP: *OpenCL Specification 1.1*, 2010, `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.
Seite 17.

[nvi10] NVIDIA: *NVIDIA CUDA Programming Guide Version 3.0*, 2010, `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf`.
Seite 17.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe.

Stuttgart, den 17.11.2010

Steffen Kieß