FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Numerische Simulation von Dünnschicht-Silizium-Solarzellen auf General Purpose GPUs mit OpenCL

Nicolas Apelt

Bachelor Thesis

Numerische Simulation von Dünnschicht-Silizium-Solarzellen auf General Purpose GPUs mit OpenCL

Nicolas Apelt

Bachelor Thesis

Aufgabensteller:	Prof. Dr. Ch. Pflaum
Betreuer:	Dipl. Math. Kai Hertel
Bearbeitungszeitraum:	10.12.2010 - 10.05.2011

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 11. Mai 2011 .....

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Es soll das Verhalten einer Dünnschicht-Solarzelle bei Einfall von (Sonnen-)Licht simuliert werden. Die Simulation wird für eine spezifische Wellenlänge im Frequenzbereich durchgeführt.

Das Verfahren konvergiert, nach einer hinreichenden Anzahl an Iterationsschritten, gegen die Lösung - insofern für die gegebenen Eingabeparameter eine solche existiert. I. d. R. reichen einige tausend bis einige zehntausend Iterationsschritte, um ausreichend gut die Lösung zu approximieren. Die Simulation wird abgebrochen, wenn die betragsmäßig größte Differenz eines Update-Schrittes für alle Zellen kleiner ist als eine gegebene (obere) Schranke ϵ. Alternativ auch nach einer gegebenen Anzahl an Iterationsschritten.

Für den Fall, dass keine Lösung existiert, divergiert das Verfahren, d. h. es wird der Float-Wertebereich überschritten. Dies zeigt sich, in dem die Zellen nach und nach den Spezialwert *infinite* annehmen und dann als `FP_INFINITE` klassifiziert werden.

2 Grundlagen

2.1 Maxwell-Gleichungen

[TH05, S. 52] Die sog. Maxwell-Gleichungen sind vier partielle Differentialgleichungen welche die Erzeugung von elektrischen und magnetischen Feldern durch Ladungen und Ströme beschreiben. Mit ihnen können alle Phänomene der klassischen Elektrodynamik erklärt werden.

Faraday'sches Gesetz (Induktionsgesetz):

$$\frac{\partial B}{\partial t} = -\nabla \times \vec{E} - \vec{M} \tag{1}$$

Maxwell-Ampere-Gesetz (erweitertes Ampere'sches Gesetz):

$$\frac{\partial \vec{D}}{\partial t} = \nabla \times H - \vec{J} \tag{2}$$

Gauß'sches Gesetz:

$$\nabla \cdot \vec{D} = 0 \tag{3}$$

Gauß'sches Gesetz für Magnetfelder:

$$\nabla \cdot \vec{B} = 0 \tag{4}$$

Die verwendeten Symbole stehen für:

$\vec{E}$:	elektrische Feldstärke ($\frac{V}{m}$)
$\vec{D}$:	elektrische Flussdichte ($\frac{C}{m^2}$)
$\vec{H}$:	magnetische Feldstärke ($\frac{A}{m}$)
$\vec{B}$:	magnetische Flussdichte (T)
$\vec{J}$:	elektrische Stromdichte ($\frac{A}{m^2}$)
$\vec{M}$:	äquivalente magnetische Stromdichte ($\frac{V}{m^2}$)

In der Optik wird i. A. angenommen, dass das verwendete Material 1) linear, 2) isotrop und 3) nicht-dispersiv ist. Für diesen (Spezial-)Fall vereinfacht sich der Zusammenhang zwischen dem $\vec{D}$- und $\vec{E}$-Feld bzw. $\vec{B}$- und $\vec{H}$-Feld zu

$$\vec{D} = \epsilon \vec{E} = \epsilon_r \epsilon_0 \vec{E} \tag{5}$$

und

$$\vec{B} = \mu \vec{H} = \mu_r \mu_0 \vec{H} \tag{6}$$

Mit den Konstanten

ϵ	dielektrische Permittivität ($\frac{F}{m}$)
ϵ_r	relative Permittivität
ϵ_0	elektrische Feldkonstante ($8{,}854 \cdot 10^{-12} \frac{F}{m}$)
μ	magnetische Permeabilität ($\frac{H}{m}$)
μ_r	Permeabilitätszahl
μ_0	magnetische Feldkonstante ($4\pi \cdot 10^{-7} \frac{H}{m}$)

Die Vektorfelder $\vec{J}$ und $\vec{M}$ stellen die sog. Quellterme dar und finden sich in der Implementierung als H_{bnd} bzw. E_{bnd} wieder. Durch sie wird hier der Einfall von Licht modelliert.

2.1.1 Finite Differenzen

Finite Differenzen spielen eine wichtige Rolle zur numerischen Lösung von Differentialgleichungen, hier im speziellen den Maxwell-Gleichungen.

Der zentrale Differenzenquotient berechnet sich zu:

$$\frac{\partial_h f}{\partial x} = \frac{f(x + \frac{1}{2}\tau) - f(x - \frac{1}{2}\tau)}{\tau}$$

Der Approximierungsfehler bei Vorwärts- bzw. Rückwärtsdifferenzen ist linear abhängig von der Schrittweite τ ($O(\tau)$). Zentrale Differenzen bieten die Möglichkeit eine genauere Annäherung zu liefern, deren Fehler quadratisch mit der Schrittweite τ sinkt ($O(\tau^2)$), d. h. man gewinnt eine Ordnung beim Approximierungsfehler.

Weiterhin ist hier der Fehler durch Auslöschung, wegen der begrenzten Genauigkeit bei der Darstellung und Subtraktion von Gleitkommazahlen, geringer. Dies rührt daher, dass die beiden Werte i. d. R. durch die größere Schrittweite auch eine größere Differenz haben.

Ein Problem bei zentralen Differenzen ist, dass oszillierende Funktionen mit $f(nt + 1) = f(nt - 1)\ \ \forall n \in \mathbb{N}, t \in \mathbb{R}$ hiermit abgeleitet Null ergeben (z. B. $cos(\pi x)$). Um dies zu umgehen, sollte beim Gitter (siehe unten) eine Auflösung gewählt werden, die über doppelt so groß ist, als die kleinste Periodenlänge aller zu betrachtenden (oszilliernden) Funktion (vgl. Nyquist-Frequenz, Nyquist-Shannon-Abtasttheorem). In der Praxis sollte hier mindestens das 4- bis 10-fache gewählt werden.

2.2 Yee-Algorithmus

Der nach ihm benannte Algorithmus geht auf Kane S. Yee im Jahre 1966 zurück und stellt eine Diskretisierung der Maxwell-Gleichungen dar.

Die Maxwell-Gleichungen (s. o.) sind vier lineare partielle Differentialgleichungen. Zur numerischen Lösung werden diese nun durch zentrale Differenzen approximiert. Heraus kommt dabei ein System finiter Differenzengleichungen, durch die man eine Lösung gut numerisch bestimmen kann.

Der *Yee-Algorithmus* löst hierbei die Maxwell-Gleichungen sowohl für das E- als auch das H-Feld. *"Durch die Verwendung beider Informationen, E und H, ist die Lösung robuster, als wenn man eine allein benutzt. (...) Sowohl elektrische als auch magnetische Material-Eigenschaften können so in direkter Weise modelliert werden."*[TH05, S. 59]

2.3 Iterationsschritt

In einem kompletten Schritt wird der Datenvektor $\vec{x}$ zum Zeitschritt n (entspricht dem Zeitpunkt t) mit der Iterationsmatrix M multipliziert und ergibt so die Werte zum nächsten Zeitschritt $(n + 1)$ (entspricht nun dem Zeitpunkt $t' = t + \tau$):

$$\vec{x}^{(n+1)} = M\vec{x}^{(n)}$$

Der Datenvektor $\vec{x}$ setzt sich aus den Komponenten des E- und H-Feldes zusammen:

$$E := \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \quad \text{und} \quad H := \begin{pmatrix} h_x \\ h_y \\ h_z \end{pmatrix}$$

$$\vec{x} := \begin{pmatrix} E \\ H \end{pmatrix} = \begin{pmatrix} e_x \\ e_y \\ e_z \\ h_x \\ h_y \\ h_z \end{pmatrix}$$

Die Iterationsmatrix M beschreibt die jeweiligen Operationen für die einzelnen Feld-Komponenten:

$$M = \left(\begin{array}{ccc|ccc} t_{E,x}(\vec{p}) & 0 & 0 & 0 & s_{E,xy} & s_{E,xz} \\ 0 & t_{E,y} & 0 & s_{E,yx} & 0 & s_{E,yz} \\ 0 & 0 & t_{E,z} & s_{E,zx} & s_{E,zy} & 0 \\ \hline 0 & s_{H,xy} & s_{H,xz} & t_{H,x} & 0 & 0 \\ s_{H,yx} & 0 & s_{H,xz} & 0 & t_{H,y} & 0 \\ s_{H,zx} & s_{H,zy} & 0 & 0 & 0 & t_{H,z} \end{array}\right) =: \begin{pmatrix} A & B \\ C & D \end{pmatrix} \tag{7}$$

Durch das angewandte *Leapfrog-Schema* werden in einem Iterationsschritt zwei halbe Zeitschritte berechnet. Hierzu muss die Matrix-Vektor-Multiplikation (zeilenweise) in zwei Teilschritte zerlegt werden: die Neuberechnung des H-Feldes und die des E-Feldes.

Wie man hieran sieht, ist die Neuberechnung eines Punktes vollkommen unabhängig von allen anderen Punkten des gleichen Feldes. Es fließen nur die Differenzen des anderen Feldes und der momentane Wert des Punktes mit ein. **So können die Updates in einer beliebigen - auch nicht-determinierten - Reihenfolge ablaufen.** Der Prozess ist somit völlig datenparallel und Flaschenhälse durch Synchronisierung werden vermieden. Ein Synchronisierungspunkt (bzw. eine Barriere) ist erst nach einem halben Zeitschritt erforderlich, damit alle Updates eines Feldes abgeschlossen sind, bevor mit dem anderen Feld fortgefahren wird.

Zur Verdeutlichung sind hier die beiden Feld-Updates getrennt dargestellt:

H-Feld Update

$$H^{(n+1)} = \begin{pmatrix} C & D \end{pmatrix} \begin{pmatrix} E^{(n)} \\ H^{(n)} \end{pmatrix} \tag{8}$$

$$H^{(n+1)} = \begin{pmatrix} h_x \\ h_y \\ h_z \end{pmatrix} = \begin{pmatrix} s_{H,xy}e_y + s_{H,xz}e_z + t_{H,x}h_x \\ s_{H,yx}e_x + s_{H,xz}e_z + t_{H,y}h_y \\ s_{H,zx}e_x + s_{H,zy}e_y + t_{H,z}h_z \end{pmatrix}$$

E-Feld Update

$$E^{(n+1)} = \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} E^{(n)} \\ H^{(n)} \end{pmatrix} \tag{9}$$

$$\vec{E}^{(n+1)} = \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} = \begin{pmatrix} t_{E,x}e_x + s_{E,xy}h_y + s_{E,xz}h_z \\ t_{E,y}e_y + s_{E,yx}h_x + s_{E,yz}h_z \\ t_{E,z}e_z + s_{E,zx}h_x + s_{E,zy}h_y \end{pmatrix}$$

2.3.1 Leapfrog Schema

Die Berechnung erfolgt indem, abwechselnd aus den E-Feld-Komponenten und dem alten Wert vom Zeitschritt n, das H-Feld für den nächsten Zeitschritt $(n+1)$ berechnet wird - und vice versa für das H-Feld.

Dieses Verfahren der Integration wird auch *Leapfrog*-Verfahren oder -Schema genannt, was zu deutsch *"Bockspringen"* oder *"überspringen"* bedeutet, da eines der Felder immer einen Zeitschritt überspringt.

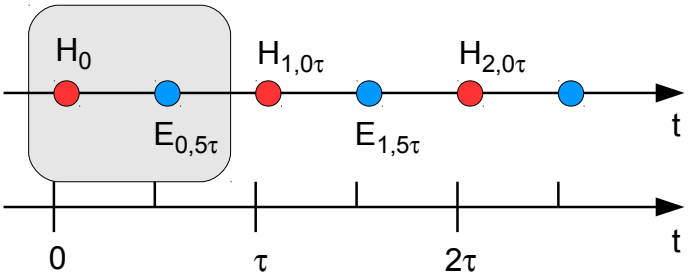

Abbildung 1: Leapfrog-Schema am Beispiel des Yee-Algorithmus: Es wird abwechselnd das elektrische Feld aus dem magnetischen berechnet und umgekehrt.

Dadurch ändert sich der Update-Schritt des $\vec{E}$-Feldes:

$$E^{(n+1)} = \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} E^{(n)} \\ H^{(n+1)} \end{pmatrix} \tag{10}$$

Einsetzen von (8) liefert

$$\begin{aligned} E^{(n+1)} &= \begin{pmatrix} A & B \end{pmatrix} \begin{pmatrix} \vec{E}^{(n)} \\ C \cdot \vec{E}^{(n)} + D \cdot \vec{H}^{(n)} \end{pmatrix} \\ &= \left(\vec{E}^{(n)} \cdot [A + B \cdot C] + \vec{H}^{(n)} \cdot [B \cdot D] \right) \\ &= \begin{pmatrix} A + BC & BD \end{pmatrix} \begin{pmatrix} \vec{E}^{(n)} \\ \vec{H}^{(n)} \end{pmatrix} \end{aligned}$$

Und somit ergibt sich eine eine neue Iterationsmatrix $\tilde{M}$:

$$\tilde{M} = \begin{pmatrix} A + BC & BD \\ C & D \end{pmatrix} \tag{11}$$

Ein kompletter Iterationsschritt für zwei halbe Zeitschritte mit Hilfe des Leapfrog-Schemas lautet dann

$$\begin{pmatrix} \vec{E}^{(n+1)} \\ \vec{H}^{(n+1)} \end{pmatrix} = \begin{pmatrix} A + BC & BD \\ C & D \end{pmatrix} \cdot \begin{pmatrix} \vec{E}^{(n)} \\ \vec{H}^{(n)} \end{pmatrix} \tag{12}$$

2.3.2 Staggered Grid und Yee-Zelle

Das sog. *Staggered Grid* (zu deutsch: versetztes Gitter) ist hier ein räumliches Gitter aus Wertekuben (oder Datenvoxeln). In jeder Dimension werden abwechselnd die Werte für beide Felder beschrieben: an den Zellenrändern das E-Feld und in den Zellenmitten das H-Feld.

Ein einzelner solcher Datenvoxel wird auch *Yee-Zelle* genannt und besteht aus den jeweils drei (räumlichen) Komponenten des E- und H-Feldes. Sie sind so angeordnet, dass jede E-Feld-Komponente von sechs H-Feld-Komponenten umgeben ist - und umgekehrt für das H-Feld.

2.3.3 Diskretisierter Iterations-Schritt

In den diskreten Iterations-Schritten wird jeweils für einen Punkt (oder Voxel) im Gitter aus den umliegenden Punkten ein neuer Wert berechnet. D. h. beim E-Feld werden neue Werte aus den umliegenden Punkten des H-Feldes und dem alten E-Feld-Wert dieses Punktes berechnet - und umgekehrt für die Werte des H-Feldes.

E-Feld

$$E^{(n+1)} = t_E E^{(n)} + s_E \left(\nabla_H \times H^{(n)} \right)$$

Dieser setzt sich zusammen aus den drei Einzelschritten:

$$E_x^{(n+1)} \mid_M = t_{E,x} E_x^{(n)} \mid_M + s_{E,x} \left(\frac{H_z^{(n)} \mid_N - H_z^{(n\tau)} \mid_S}{h_y} - \frac{H_y^{(n)} \mid_T - H_y^{(n)} \mid_B}{h_z} \right)$$

$$E_y^{(n+1)} \mid_M = t_{E,y} E_y^{(n)} \mid_M + s_{E,y} \left(\frac{H_x^{(n)} \mid_T - H_x^{(n)} \mid_B}{h_z} - \frac{H_z^{(n)} \mid_E - H_z^{(n)} \mid_W}{h_x} \right)$$

$$E_z^{(n+1)} \mid_M = t_{E,z} E_z^{(n)} \mid_M + s_{E,z} \left(\frac{H_x^{(n)} \mid_N - H_x^{(n)} \mid_S}{h_y} - \frac{H_y^{(n)} \mid_W - H_y^{(n)} \mid_E}{h_x} \right)$$

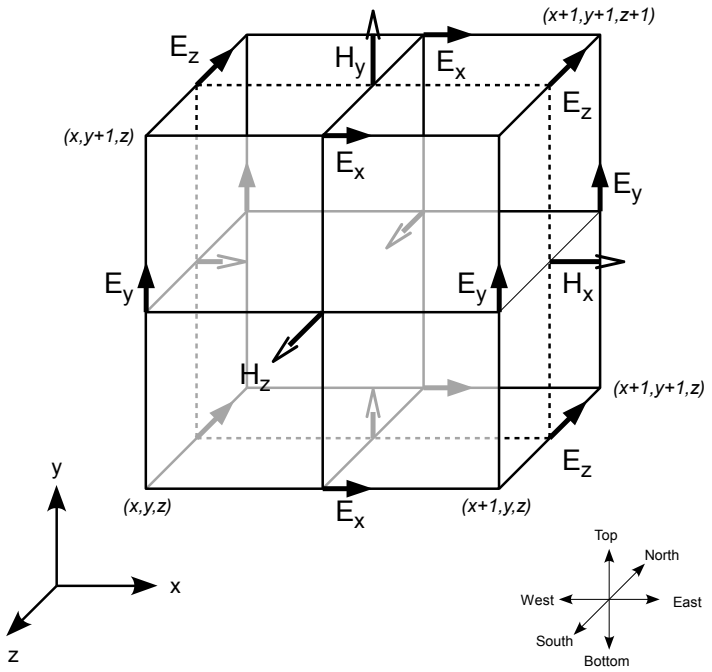

Abbildung 2: Schematische Darstellung einer 3-dreidimensionalen Yee-Zelle. Die E-Feld-Komponenten sitzen auf den Kantenmitten, die des H-Feldes auf den Seitenmitten [TH05, S. 59].

Für die Konstanten s und t gilt:

$$s = \frac{1}{1 + \frac{\sigma\rho\tau}{\epsilon}} \Big(1 - \frac{\tau}{\epsilon}(\sigma(1-\rho))\Big)$$

$$t = \frac{1}{1 + \frac{\sigma\rho\tau}{\epsilon}} \frac{\tau}{\epsilon}$$

H-Feld

$$H^{(n+1)} = t_H H^{(n)} + s_H \Big(\nabla_E \times E^{(n)}\Big)$$

$$H_x^{(n+1)} \mid_M = t_{H,x} H_x^{(n)} \mid_M + s_{H,x} \left(\frac{E_y^{(n)} \mid_B - E_y^{(n)} \mid_T}{e_z} - \frac{E_z^{(n)} \mid_N - E_z^{(n)} \mid_S}{e_x} \right)$$

$$H_y^{(n+1)} \mid_M = t_{H,y} H_y^{(n)} \mid_M + s_{H,y} \left(\frac{E_x^{(n)} \mid_T - E_x^{(n)} \mid_B}{e_z} - \frac{E_z^{(n)} \mid_W - E_z^{(n)} \mid_E}{e_x} \right)$$

$$H_z^{(n+1)} \mid_M = t_{H,z} H_z^{(n)} \mid_M + s_{H,z} \left(\frac{E_x^{(n)} \mid_S - E_x^{(n)} \mid_N}{e_y} - \frac{E_y^{(n)} \mid_E - E_y^{(n)} \mid_W}{e_x} \right)$$

3 Implementierung

Implementiert wird der Algorithmus in OpenCL, da er so plattform-unabhängig ist und auf jeder Grafikkarte bzw. jedem OpenCL-fähigen Gerät ausgeführt werden kann.

Die Implementierung erfolgt für den Host-Code in C bzw. C++ und für den OpenCL-Code in eben diesem C-Dialekt.

3.1 OpenCL

OpenCL ist eine plattform-unabhängige Programmiersprache zur GPGPU-Programmierung auch auf heterogenen Plattformen. Sie wurde ursprünglich von Apple entwickelt und dann zur Standardisierung an die Khronos Group übergeben.

3.1.1 Vergleich zu CUDA

CUDA ist eine von Nvidia ins Leben gerufene Technik zur Programmierung ihrer Grafikkarten und ist somit nur auf Nvidia-Geräten lauffähig.

Da OpenCL als plattform-unabhängiger Standard entworfen wurde, ist es nicht notwendig, sich intensiv mit spezifischen Hardware-Eigenschaften auseinanderzusetzen, da im vornherein nicht feststehen muss auf welcher Architektur das Programm läuft. Das erleichtert die Implementierung, da nicht auf die genauen Hardware-Belange geachtet werden muss. Diese Arbeit wird später vom Compiler übernommen. Da der OpenCL-Compiler i. d. R. vom Hersteller selbst geliefert wird, können architektur-abhängige Optimierungen hier vorgenommen werden.

3.2 Framework

Entwickelt wurde das Programm für die OpenCL-Spezifikation 1.0 [Khr09] mit dem *NVidia GPU Computing SDK Version 3.0* [Nvi] unter Linux (Ubuntu 10.04).

Weiterhin wurde, um den Programmieraufwand weiter zu reduzieren, eine vom Lehrstuhl erstellte Wrapper-Klasse um die *C++ Bindings für OpenCL 1.0 (rev 45)* verwendet. Außerdem soll dies die spätere Einbindung in ein bestehendes C++-Framework erleichtern.

3.3 Eingabedaten

Die Daten liegen in Form gleichmäßiger, rechtwinkliger dreidimensionaler Gitter (rectilinear grids) vor.

Das E- und H-Feld besteht jeweils zwei Konstanten zur Skalierung (s und t) und je einem Randwert für die x- und y-Richtung beider Felder.

Jeder Voxel (Volumenpixel) enthält sechs komplexe Werte. Für jede Ausbreitungsrichtung gibt es zwei Werte, die die Feldstärke senkrecht dazu ausdrücken:

- in x-Richtung: y, z
- in y-Richtung: x, z
- in z-Richtung: x, y

Die Randwerte E_{bnd} und H_{bnd} sind nur für die x- und y-Achse beschrieben und enthalten jeweils einen komplexen Wert je Datenpunkt.

Die einzelnen Komponenten sind jeweils in einer Datei im VTK-Format als einfach genaue Fließkommazahlen abgelegt [Vtka] [Vtkb].

Ein Voxel enthält sechs mal sechs und zwei mal zwei, somit insgesamt 40 komplexe Werte:

- $E, H, s_E, s_H, t_E, t_H : \big\{\{(x,y),(x,z)\},\{(y,x),(y,z)\},\{(z,x),(z,y)\}\big\}$
- $E_{bnd}, H_{bnd} : \big\{x, y\big\}$

Für die Anzahl der Datenpunkte in jeder Dimension bedeutet dies:

- Alle E-Feld-Komponenten (E, s_E, t_H, E_{bnd}) haben in ihrer jeweiligen Richtung immer einen Datenpunkt mehr.
- Alle H-Feld-Komponenten (H, s_H, t_H, H_{bnd}) haben außer in ihrer jeweiligen Richtung einen Datenpunkt mehr.

Beispiel Die Komponenten eines dreidimensionalen Staggered Grids der Größe 1x1x1 haben folgende Ausmaße:

E_x, $s_{E,x}$, $t_{E,x}$, $E_{bnd,x}$	: 1 x 2 x 2
E_y, $s_{E,y}$, $t_{E,y}$, $E_{bnd,y}$	: 2 x 1 x 2
E_z, $s_{E,z}$, $t_{E,z}$	: 2 x 2 x 1
H_x, $s_{H,x}$, $t_{H,x}$, $H_{bnd,x}$	: 2 x 1 x 1
H_y, $s_{H,y}$, $t_{H,y}$, $H_{bnd,y}$	: 1 x 2 x 1
H_z, $s_{H,z}$, $t_{H,z}$	: 1 x 1 x 2

3.4 Datenstruktur

Zwei häufig benutzte Typen von Datenstrukturen sind:

Structure of Arrays: in einer Datenstruktur liegen mehrere Arrays.

Array of Structures: ein Array mit mehreren Datenstrukturen als Elementen.

[Nvi10, S. 38]

3.4.1 Structure of Arrays

Bei einer *Structure of Arrays* (kurz: SoA) gibt es übergeordnet eine Datenstruktur, die für jede Komponente ein Array mit Datenelementen enthält. Die Elemente werden so komponentenweise zusammengefasst.

Listing 1: Code-Beispiel einer Struct-of-Arrays in C

```
struct {
  complex float Exy[N], Exz[N], Eyx[N], Eyz[N], Ezx[N], Ezy[N];
  complex float Hxy[N], Hxz[N], Hyx[N], Hyz[N], Hzx[N], Hzy[N];
  complex float sExy[N], sExz[N], sEyx[N], sEyz[N], sEzx[N], sEzy[N];
  complex float sHxy[N], sHxz[N], sHyx[N], sHyz[N], sHzx[N], sHzy[N];
  complex float tExy[N], tExz[N], tEyx[N], tEyz[N], tEzx[N], tEzy[N];
  complex float tHxy[N], tHxz[N], tHyx[N], tHyz[N], tHzx[N], tHzy[N];
  complex float Exbnd[N], Eybnd[N];
  complex float Hxbnd[N], Hybnd[N];
} SoA;
```

3.4.2 Array of Structs

Bei dem *Array-of-Structs* (kurz: AoS) liegen die Werte für einen Datenpunkt in einer Struktur und mehrere Datenpunkte werden zu einem Array zusammengefasst. D. h., hier werden die Werte nach ihren Koordinaten gruppiert.

Listing 2: Code-Beispiel einer Array-of-Structs in C

```
struct {
  complex float Exy, Exz, Eyx, Eyz, Ezx, Ezy;
  complex float Hxy, Hxz, Hyx, Hyz, Hzx, Hzy;
  complex float sExy, sExz, sEyx, sEyz, sEzx, sEzy;
  complex float sHxy, sHxz, sHyx, sHyz, sHzx, sHzy;
  complex float tExy, tExz, tEyx, tEyz, tEzx, tEzy;
  complex float tHxy, tHxz, tHyx, tHyz, tHzx, tHzy;
  complex float Exbnd, Eybnd;
  complex float Hxbnd, Hybnd;
} AoS[N];
```

3.4.3 Stencil-Operation

Die sog. Stencil-Operation bildet die zentralen Differenzen der jeweils sechs umliegenden Datenpunkte. Drei dieser Datenpunkte liegen in dem aktuellen Voxel, die drei anderen sind um $+1$ für das E-Feld bzw. -1 für das H-Feld in der jeweiligen Richtung verschoben.

Indizes Die linearisierten Indizes berechnen sich wie folgt:

$$\mathcal{I} : \mathbb{N}^3 \to \mathbb{N}, (x, y, z) \mapsto ((((z \cdot ny) + y) \cdot nx) + x) \cdot 80)$$

E-Feld

N(orth)	= $\mathcal{I}$(x,	y+1,	z)
S(outh)	= $\mathcal{I}$(x,	y,	z)
W(est)	= $\mathcal{I}$(x,	y,	z)
E(ast)	= $\mathcal{I}$(x+1,	y,	z)
T(op)	= $\mathcal{I}$(x,	y,	z+1)
B(ottom)	= $\mathcal{I}$(x,	y,	z)

H-Feld

N(orth)	= $\mathcal{I}$(x,	y,	z)
S(outh)	= $\mathcal{I}$(x,	y-1,	z)
W(est)	= $\mathcal{I}$(x-1,	y,	z)
E(ast)	= $\mathcal{I}$(x,	y,	z)
T(op)	= $\mathcal{I}$(x,	y,	z)
B(ottom)	= $\mathcal{I}$(x,	y,	z-1)

3.4.4 Lese- und Schreibzugriffe je Schritt

Jeder Datenpunkt enthält – unabhängig von der gewählten Datenstruktur – $6 \times 6 + 2 \times 2$ komplexe Werte, sprich 80 einfach genaue Fließkommawerte und hat somit eine Größe von $80 \times 4 = 320$ Bytes.

Jeder einzelne Iterationsschritt liest die sechs umliegenden Datenpunkte (s. Stencil-Operation) und schreibt das Ergebnis in den Aktuellen.

Lesezugriffe aus dem aktuellen Datenpunkt:

Stencil	$3 \times 6 \times 2$:	36	
Konstanten	$2 \times 2 \times 2$:	8	
aktueller Wert	$1 \times 6 \times 2$:	12	
Boundary Condition	$1 \times 2 \times 2$:	4	
Gesamt	=	**60**	Float-Werte ($\widehat{=}$ 240 Bytes)

und aus benachbarten Punkten: $6 \times 2 \times 2$ (Stencil) = 24 Float-Werte (6×16 Bytes).

Abschließend noch ein schreibender Zugriff auf 6×2 Float-Werte (48 Bytes).

3.4.5 Wahl der Datenstruktur

Bei Betrachtung eines Iterationsschrittes stellt man fest, dass die neuen Werte jeder Feld-Komponente aus den verschiedenen Komponenten des jeweils anderen Feldes berechnet werden. Es werden also zuerst alle Komponenten eines Punktes vollständig berechnet – und nicht alle Punkte einer Komponente.

Die Speicherzugriffe in einem Schritt sind lokaler, wenn die Daten nach ihren Koordinaten und nicht nach ihren Komponenten geordnet sind.

Aus dieser theoretischen Überlegung heraus fiel die Wahl für die zu verwendende Datenstruktur auf das *Array of Structs*, da es die Daten punktweise zusammenfasst.

Es sollte allerdings auch bedacht werden, dass die Daten linearisiert im Speicher liegen, d. h. ein in y-Richtung "benachbarter"Datenpunkt liegt n_x Punkte entfernt, in z-Richtung sind es sogar schon $n_x \cdot n_y$ Punkte. Dennoch bietet hier die AoS den Vorteil, dass ein Großteil der benötigten Werte im aktuellen Datenpunkt und somit einer einzelnen Stelle im Speicher stehen.

3.5 Ausgabedaten

Als Ergebnis wird für jeden Gitterpunkt die euklidische Norm (2-Norm) des E-Feldes berechnet:

$$\begin{aligned}\|E\|_2 &= \sqrt{\left|\vec{e_x}\right|^2 + \left|\vec{e_y}\right|^2 + \left|\vec{e_z}\right|^2 + \left|\vec{h_x}\right|^2 + \left|\vec{h_y}\right|^2 + \left|\vec{h_z}\right|^2} \\ &= \sqrt{(e_{x,y} \cdot e^*_{x,y}) + (e_{x,z} \cdot e^*_{x,z}) + (e_{y,x} \cdot e^*_{x,y}) + \dots}\end{aligned}$$

und anschließend im VTK-Format in die Datei `results.vtk` geschrieben.

3.6 OpenCL-Kernel

Der Algorithmus besteht aus insgesamt vier Schritten, wobei zwei jeweils das E- bzw. H-Feld neu berechnen und die anderen beiden die sog. Rand-Updates ausführen. Jeder dieser vier Schritte ist in einem separaten CL-Kernel implementiert:

- Rand-Update des E-Feldes: *updateE(...)*
- Neuberechnung des H-Feldes: *iterateH(...)*
- Rand-Update des H-Feldes: *updateH(...)*
- Neuberechnung des E-Feldes: *iterateE(...)*

3.6.1 Iterations-Schritt

Diskrete Implementierung X-Achse:

$$E_{xy} = t_{E,xy} \cdot E_{xy} + s_{E,xy} \cdot \big((H_{yx}\,|_B + H_{yz}\,|_B) - (H_{yx}\,|_T + H_{yz}\,|_T)\big) + E_{x,bnd}$$

$$E_{xz} = t_{E,xz} \cdot E_{xz} + s_{E,xz} \cdot \big((H_{zx}\,|_B + H_{zy}\,|_B) - (H_{zx}\,|_T + H_{zy}\,|_T)\big)$$

Y-Achse:

$$E_{yx} = t_{E,yx} \cdot E_{yx} + s_{E,yx} \cdot \big((H_{xy}\,|_T + H_{xz}\,|_T) - (H_{xy}\,|_B + H_{xz}\,|_B)\big) + E_{y,bnd}$$

$$E_{yz} = t_{E,yz} \cdot E_{yz} + s_{E,yz} \cdot \big((H_{zx}\,|_W + H_{zy}\,|_W) - (H_{zx}\,|_E + H_{zy}\,|_E)\big)$$

Z-Achse:

$$E_{zx} = t_{E,zx} \cdot E_{zx} + s_{E,zx} \cdot \big((H_{xy}\,|_S + H_{xz}\,|_S) - (H_{xy}\,|_N + H_{xz}\,|_N)\big)$$

$$E_{zy} = t_{E,zy} \cdot E_{yz} + s_{E,zy} \cdot \big((H_{yx}\,|_E + H_{yz}\,|_E) - (H_{yx}\,|_W + H_{yz}\,|_W)\big)$$

3.6.2 Update-Schritt

Durch den Update-Schritt oder Rand-Update wird das Gebiet periodisch fortgesetzt. Beim E-Feld werden die Werte vom oberen Rand an den unteren Rand kopiert und umgekehrt werden beim H-Feld alle Werte vom unteren an den oberen Rand kopiert.

Oberer Rand bedeutet hier die folgenden Ebenen:

- in X-Richtung die YZ-Ebene bei $y = n_y - 1, z = n_z - 1$
- in Y-Richtung die XZ-Ebene bei $x = n_x - 1, z = n_z - 1$

Umgekehrt sind die unteren Ränder diese Ebenen:

- in X-Richtung die YZ-Ebene bei $y = 0, z = 0$
- in Y-Richtung die XZ-Ebene bei $x = 0, z = 0$

E-Feld x-Achse:

$$E_{xy}[n_x - 2, y, z] = E_{xy}[1, y, z]$$

$$E_{xz}[n_x - 2, y, z] = E_{xy}[1, y, z]$$

y-Achse:

$$E_{yx}[x, n_y - 1, z] = E_{yx}[x, 1, z]$$

$$E_{yz}[x, n_y - 1, z] = E_{yz}[x, 1, z]$$

z-Achse:

$$E_{zx}[x, y, n_z - 1] = E_{zx}[x, y, 1]$$

$$E_{zy}[x, y, n_z - 1] = E_{zy}[x, y, 1]$$

für alle

$$x \in \{0, \ldots, n_x - 1\}, y \in \{0, \ldots, n_y - 1\}, z \in \{0, \ldots, n_z - 1\}$$

Update-Schritt H-Feld x-Achse:

$$H_{xy}[n_x - 2, y, z] = H_{xy}[1, y, z]$$

$$H_{xz}[n_x - 2, y, z] = H_{xy}[1, y, z]$$

y-Achse:

$$H_{yx}[x, n_y - 1, z] = H_{yx}[x, 1, z]$$

$$H_{yz}[x, n_y - 1, z] = H_{yz}[x, 1, z]$$

z-Achse:

$$H_{zx}[x, y, n_z - 1] = H_{zx}[x, y, 1]$$

$$H_{zy}[x, y, n_z - 1] = H_{zy}[x, y, 1]$$

für alle

$$x \in \{0, \ldots, n_x - 1\}, y \in \{0, \ldots, n_y - 1\}, z \in \{0, \ldots, n_z - 1\}$$

3.6.3 Hilfs-Routinen

Als einzige Hilfs-Routine wird die komplexe Multiplikation zweier Float-Werte benötigt (`complexMul(a,b)`). Sie tritt bei der Berechnung jeder Komponente auf und ist (noch) nicht im derzeitigen OpenCL-Standard 1.0 enthalten.

3.7 Fehlersuche

Um falsche Zugriffe bei den Iterations-Schritten aufzufinden, werden Datenpunkte, auf die nicht zugegriffen werden sollte mit *NaN* (Not a number) gefüllt.

Diese Datenpunkte entstehen, an den Rändern des Gebietes, da hier Punkte aufgefüllt werden müssen, um einheitliche Dimensionen bzw. Array-Größen zu gewährleisten.

Gemäß der OpenCL-Spezifikation ([Khr09, S. 203]) ist die Unterstützung denormalisierter, einfach genauer Fließkommazahlen optional. Zumindest NVidia-Geräte ([Nvi10, S. 53]) liefern bei Operationen mit *NaN* als Eingabe wieder ein (stilles) *NaN* als Ausgabe. Das kann helfen falsche Indizierungen frühzeitig zu erkennen.

3.8 Kommandozeilen-Parameter

Kommandozeile:

```
pmaxwell [directory] [iterations] ([snapshot interval])
```

directory Verzeichnis mit den Eingabedaten.

iterations Anzahl der zu durchlaufenden Iterationen.

snapshot interval *(optional)* Hier kann ein Intervall angegeben werden in dem ein Zwischenergebnis ausgegeben werden soll.

Die Zwischen- und das Endergebnis werden in einer VTK-Datei mit dem Namen `results.N.vtk` gespeichert, wobei `N` eine laufende Nummer darstellt, beginnend bei 0.

3.9 Kernel-Code

Der Code-Auszug unten stellt einen Teil des OpenCL-Kernels zur Neuberechnung des H-Feldes dar (hier: nur y-Richtung der X-Komponente).

Listing 3: Code-Auszug aus dem OpenCL-Kernel

```
__kernel void iterateAoS_H(__global float * AoS, const ushort4 dim)
{
  // current coordinates
  const unsigned int x = get_global_id(0);
  const unsigned int y = get_global_id(1);
  const unsigned int z = get_global_id(2);

  // dimension sizes
  const unsigned int nx = dim.x;
  const unsigned int ny = dim.y;
  const unsigned int nz = dim.z;

  // bound check
  if ((x+1 >= nx) || (y+1 >= ny) || (z+1 >= nz))
    return;

  // indices
  const unsigned int idx = INDEX(x, y, z, nx, ny, nz);

  // Stencil Indices (for E fields)
  /* ... */
  const unsigned int idxT = INDEX(x  , y  , z+1, nx, ny, nz);
  const unsigned int idxB = INDEX(x  , y  , z  , nx, ny, nz);

  float2 H_xy = (float2) (AoS[idx + H_XY_RE], AoS[idx + H_XY_IM]);
  /* ... */

  {
    const float2 T_E_yx = (float2) (AoS[idxT + E_YX_RE], AoS[idxT +
        E_YX_IM]);
    const float2 T_E_yz = (float2) (AoS[idxT + E_YZ_RE], AoS[idxT +
        E_YZ_IM]);
    const float2 B_E_yx = (float2) (AoS[idxB + E_YX_RE], AoS[idxB +
        E_YX_IM]);
    const float2 B_E_yz = (float2) (AoS[idxB + E_YZ_RE], AoS[idxB +
        E_YZ_IM]);
    const float2 s_H_xy = (float2) (AoS[idx + sH_XY_RE], AoS[idx +
        sH_XY_IM]);
    const float2 t_H_xy = (float2) (AoS[idx + tH_XY_RE], AoS[idx +
        tH_XY_IM]);
    const float2 H_x_bnd = (float2) (AoS[idx + H_X_BND_RE], AoS[idx +
        H_X_BND_IM]);

    H_xy = complexMul(t_H_xy, H_xy) + complexMul(s_H_xy, (B_E_yx +
        B_E_yz) - (T_E_yx + T_E_yz) ) + H_x_bnd;
  }

  /* ... */
}
```

4 Laufzeitmessungen

4.1 Wahl der Datenstruktur

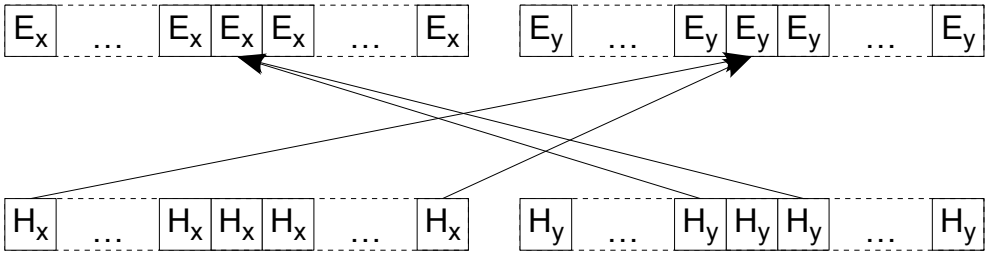

Abbildung 3: Structure-of-Arrays: Die Daten sind komponentenweise abgelegt und Zugriffe beim Update erfolgen auf verschiedenen Stellen des Speichers (hier: 2-dimensionaler Fall).

SoA (Structure of Arrays) Die Datenstruktur SoA erwies sich als zu inperformant, da immer wieder Zugriffe auf einzelne Werte ueber den gesamten Datenbereich erfolgten. Dies liegt daran, dass hier Werte nicht auf Grund ihrer Koordinaten sondern nach ihrer Propagations-Richtung gruppiert sind.

Der implementierte bzw. zu implementierende Algorithmus arbeitet jedoch alle drei Richtungen je Datenpunkt auf einmal ab.

Außerdem treten hier mehr Schreibzugriffe auf, da mehrere Richtungen in die Neuberechnung eines Datenpunktes einfließen und immer nur eine Richtung auf einmal berechnet wurde.

Grundsätzlich ließe sich, der Algorithmus umformulieren, so dass er die verschiedenen Richtungen auf einmal berechnet. Dies führt jedoch zu dem AoS-Verhalten und so würde der Algorithmus nicht mehr zur Datenstruktur passen.

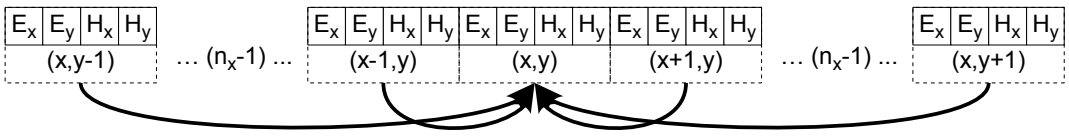

Abbildung 4: Array-of-Structures: Die Daten liegen koordinatenweise zusammen, beim Update wird auf Bereiche dieser sechs (sieben) Blöcke zugegriffen (hier: 2-dimensionaler Fall).

AoS (Array of Structures) Das Array-of-Structs erwies sich als die praktikablere der beiden Datenstrukturen, zumal der Algorithmus auch immer eine Koordinate nach der anderen abarbeitet.

Hier wird in jedem Berechnungsschritt des CL-Kernels jeweils vollständig das E- bzw. H-Feld eines Datenpunktes berechnet.

4.2 Work-Group-Größen (NDRanges)

Die nachfolgenden Tabellen zeigen verschiedene Laufzeitmessungen für 1-, 2- und 3-dimensionale Work-Groups [1]. Der Wert in der ersten Spalte gibt die Gesamtgröße der Work-Group an, d. h. die Anzahl der Datenpunkte die gemeinsam in einer auf einer *Compute Unit* ausgeführt werden [Khr09, S. 17]. In der zweite Spalte folgt die Kantenlänge der Work-Group in den jeweiligen Dimensionen.

[1] Gemessen wurde auf einer NVidia GT 330M (6 Compute Units, 48 Cores, 1 GB Global Memory, Clock: 1265 MHz).

Die die höchste zulässige Zahl der Datenpunkte entspricht der maximalen Anzahl an Work-Items je Work-Group. Dieser Wert ist geräteabhängig und kann über die Funktion *clGetDeviceInfo()* unter `CL_DEVICE_MAX_WORK_GROUP_SIZE` abgefragt werden [Khr09, S. 33]. In der beschriebenen Messung lag die maximale Work-Group-Größe bei 512 Elementen.

Die anderen Spalten enthalten die durchschnittlichen Laufzeiten für jeweils einen (1) Iterationsschritt.

4.2.1 Laufzeit je Iterationsschritt bei 1-dimensionaler Work-Group

Work-Group Größe	n	$(n, 1, 1)$	$(1, n, 1)$	$(1, 1, n)$
1	1	39.8 ms	39.9 ms	39.8 ms
2	2	22.4 ms	24.3 ms	21.3 ms
4	4	14.9 ms	16.6 ms	13.4 ms
8	8	13.9 ms	14.2 ms	11.4 ms
16	16	13.5 ms	13.9 ms	11.9 ms
32	32	13.0 ms	13.1 ms	14.8 ms
64	64	13.0 ms	13.0 ms	16.4 ms

4.2.2 Laufzeit je Iterationsschritt bei 2-dimensionaler Work-Group

Work-Group Größe	n	$(n, n, 1)$	$(1, n, n)$	$(n, 1, n)$
1	1	39.8 ms	39.8 ms	39.8 ms
4	2	16.5 ms	14.3 ms	13.7 ms
16	4	14.5 ms	11.5 ms	11.2 ms
64	8	12.8 ms	12.6 ms	11.4 ms
256	16	14.7 ms	13.8 ms	10.8 ms

4.2.3 Laufzeit je Iterationsschritt bei 3-dimensionaler Work-Group

Work-Group Größe	n	(n,n,n)
1	1	39.8 ms
8	2	12.6 ms
64	4	11.4 ms

5 Schluss

5.1 Interpretation der Ergebnisse

Bei den hier präsentierten Messergebnissen treten keinerlei Effekte durch unterschiedliche Dimensionierung der Work-Groups auf. Das bedeutet, dass keine der Techniken wie Prefetching, Aligning oder benachbarte Zugriffe, Wirkung zeigt.

Die gemessenen Zeiten hängen also lediglich von der Geschwindigkeit der jeweiligen GPUs und dem Speicherdurchsatz ab.

Wählt man die Workgroup-Größen zu klein, werden die SIMDs einer *Compute Unit* nicht vollständige ausgelastet – man verschenkt praktisch Rechenzeit. ATI empfiehlt sogar für die *ATI Radeon HD 4000er*-Familie, dass die Work-Group-Size immer ein Vielfaches von 64 sein sollte, um das sog. *Latency Hiding*, das "Verstecken von Latenzen", in vollem Umfang nutzen zu können ([Hou09, S. 13]).

Workgroup-Größen größer als von der Compute-Unit auf einmal verarbeitbar ergaben praktisch keine negativen Effekte.

Durch die Größe einer halben Datenstruktur (da meist nur eines der beiden Felder benötigt wird) von 160 Bytes (entspricht 1280 Bits) wird der Datenbus in jedem Fall vollständig ausgelastet. Zumal diese auch linear in einem Block im Speicher vorliegt.

5.2 Ausblick

5.2.1 Komplexe Datentypen

Im hier verwendeten Standard OpenCL 1.0 wurden noch ausschließlich reelle Datentypen unterstützt. Die komplexen Datentypen waren hier bereits als reservierte Ausdrücke enthalten.

In der Version 1.1 des Standards, der im zweiten Halbjahr 2010 vorgestellt wurde, sind diese komplexen Datentypen bereits enthalten. Alle Fließkomma-Variablen können somit nun halb-, einfach-, doppelt- und vierfach genaue reelle, imaginäre oder komplexe Datentypen sein.

5.2.2 Optimierungen

Der Algorithmus bietet selbst keinerlei direkte Optimierungsmöglichkeiten, da nur wenig redundante Zugriffe vorliegen.

Der offensichtlich redundanteste Zugriff ist der bei der Berechnung zweier benachbarter zentraler Differenzen. Hier wird z. B. in X-Richtung zweimal nacheinander auf das selbe Element zugegriffen, z. B. einmal als "rechtes" und danach als "linkes" Element. Hier gibt es jedoch einige grundsätzliche Probleme:

Reihenfolge Es ist nicht garantiert, dass die Compute-Unit die Work-Items in einer bestimmten Reihenfolge abarbeitet.

Lokaler Speicher Der lokale Speicher einer Compute-Unit ist sehr klein und reicht kaum aus um mehrere solch großer Strukturen abzulegen.

Lokalität Die Optimierung ist auf die Elemente einer Work-Group beschränkt, da über die Grenzen einer Compute-Unit hinaus kein gemeinsamer Speicher existiert (außer dem langsameren globalen Speicher).

Auf Geräten der ATI Radeon 5000er-Reihe unterstützt der interne Speicherbus 128-bit breite Datentransfers. Hier besteht eine weitere Optimierungs-möglichkeit mit moderatem Potential durch die Verwendung von Datentypen, wie z. B. *float4*, die diesen vollständig ausnutzen. (Hierzu schreibt das NVidia OpenCL Programming Guide, dass Compiler und Thread Scheduler die Instruktionen so optimal wie möglich anordne und keine Notwendigkeit besteht, mehrere skalare Werte in einen Vektor zu packen; hier jedoch um Speicherbank-Konflikte zu vermeiden [Nvi09, S. 27, Kap. 3.2.6]).

Es ist jedoch nicht trivial, ob auf diesem Weg eine signifikate Verbesserung der Rechenzeit überhaupt möglich ist, da nicht einfach mehrere Punkte bzw. Berechnungen durch bloßes Verwenden eines vektoriellen Datentyps zusammengefasst werden können.

5.2.3 Integration in bestehendes Framework

Die hier vorgestellte Arbeit stellt die Implementation bzw. Portierung des bestehenden Algorithmus auf GPGPUs dar.

Die Datenbasis wird jedoch von einem bereits bestehenden Framework des Lehrstuhls bereitgestellt. Es ist angedacht, diese Arbeit später in dieses Framework zu integrieren.

Wie oben beschrieben, werden die initialen Daten aus einer Reihe VTK-Dateien ausgelesen. Nach Integration können so beliebige Simulationen ohne diesen Umweg direkt nach Generierung der Startwerte auf der Grafikkarte berechnet werden.

5.2.4 Just-in-Time Kompilierung mit Expression Templates

Durch die Verwendung von sog. *Expression Templates*, wie sie bereits im o. g. Framework Verwendung finden, kann der Code für den erforderlichen OpenCL-Kernel zur Laufzeit generiert und übersetzt werden. Das würde den Spielraum für Berechnungen erweitern, da so praktisch für beliebige Berechnungen ein OpenCL-Kernel generiert werden kann, insofern dafür ein Expression Template existiert.

Wären solche Templates einmal korrekt implementiert, wird es auch leichter kompliziertere Berechnungen auf die Grafikkarte auszulagern, da keine aufwändige manuelle Kodierung und Fehlersuche mehr nötig ist.

6 Literatur

[Her10] Kai Hertel. Simulation of high-frequency optical waves. August 2010.

[Hou09] Mike Houston. AMD and OpenCL. August 26, 2009. http://www.khronos.org/developers/library/2009_siggraph_bof_opencl/OpenC_BOF_-_AMD-Siggraph_Aug09.pdf.

[Khr09] Khronos Group. OpenCL Specification v1.0 Document Revision 48. 2009. http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf.

[Nvi] NVidia GPU SDK, NVidia Developer Zone. http://developer.nvidia.com/object/opencl.html.

[Nvi09] NVidia OpenCL Best Practices Guide, Version 2.3. August 31, 2009.

[Nvi10] NVidia OpenCL Programming Guide, Version 2.3, 18.02.2010. 2010.

[Pfl09] Christoph Pflaum. Computational Optics. 2009.

[TH05] Allen Taflove and Susan C. Hagness. Computational electrodynamics. 2005.

[Vtka] Homepage of VTK - The Visualization Toolkit. http://www.vtk.org.

[Vtkb] VTK File Formats. http://vtk.org/VTK/img/file-formats.pdf.