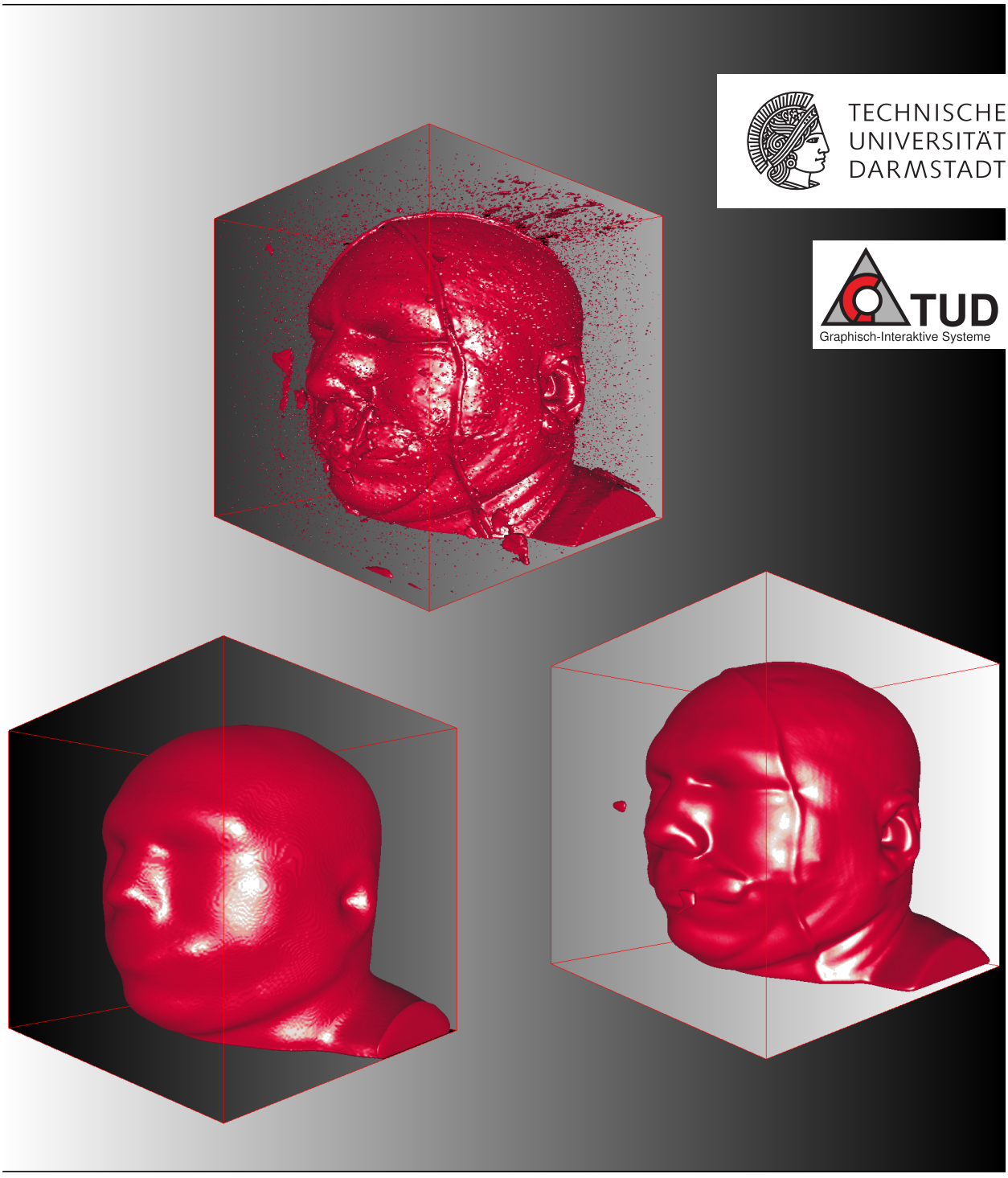
GPU basierte volumetrische anisotrope Diffusion

Diplomarbeit im Fachgebiet Graphisch-Interaktive Systeme



Diplomarbeit

GPU basierte volumetrische anisotrope Diffusion

Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker

vorgelegt von
Andreas Schwarzkopf
Geboren in Hanau
Matrikel Nummer: 1201387

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Graphisch-Interaktive Systeme
Capturing Reality Group
Prof. Dr.-Ing. Michael Goesele

12. Mai 2010

Betreuer:
Dipl.-Inf. Thomas Kalbe

Das Titelbild zeigt ein Isooberflächenrendering des bekannten »Head Male« CT Datensatzes aus dem Visible Human Project der »National Library of Medicine, National Institutes of Health, USA«. Der Datensatz wurde zunächst mit 5% Voxelfehlern (Additives gaußsches Rauschen, 10% Standardabweichung) überlagert (oben). Die Glättung mit einem linearen homogenen Diffusionsfilter (unten links) entfernt zwar nach wenigen Iterationen das Rauschen, zerstört aber auch schnell die Oberflächenstruktur. Die Glättung mit einem nichtlinearen anisotropen Diffusionsfilter (hier: edge enhancing diffusion) mit bilateralem Prefiltering erhält die Oberflächenstruktur trotz guter Glättungseigenschaften.

Danksagung

Eine erfolgreiche Diplomarbeit entsteht immer im Austausch mit erfahrenen Wissenschaftlern, Betreuern und Komilitonen und mit der Unterstützung von Freunden und Familie.

Ich möchte daher Prof. Dr.-Ing. Michael Goesele danken, in dessen Forschungsgruppe diese Diplomarbeit entstanden ist. Mein besonderer Dank gilt meinem Betreuer Dipl. Ing. Thomas Kalbe, der immer ein offenes Ohr für formelle und inhaltliche Fragen hatte und mir trotz seiner laufenden Dissertationsarbeit immer Zeit für E-Mails, Telefonate und spontane Besuche in seinem Büro einräumen konnte.

Ich danke Matthias Warkentin für die fruchtbaren Diskussionen zur Lösung der partiellen Differentialgleichungssysteme und meinen Komilitonen und langjährigen Freunden Steffen Roch und Jan Trautmann für Korrektur und Lektorat dieser Diplomarbeit.

Besonderer Dank gilt zuletzt meiner Familie und insbesondere meiner Ehefrau Margareta, die in den letzten Monaten geduldig viele Stunden auf die Fertigstellung der Programmierarbeiten und der vorliegenden Ausarbeitung gewartet hat. Ich liebe dich!

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 12. Mai 2010

_____

Andreas Schwarzkopf

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der effizienten Umsetzung volumetrischer Diffusionsfilter auf modernen Grafikkarten. Im Vordergrund stehen dabei die zu erwartenden Geschwindigkeitsvorteile, die sich durch eine geschickte GPGPU Implementierung und die damit einhergehende massiv paralleliesierte Ausführung erzielen lassen. Es zeigt sich, daß volumetrische Diffusionsfilter auf Grafikkarten von 40 bis 80-fachen Geschwindigkeitssteigerungen verglichen mit reinen CPU Lösungen profitieren können – und das auf handelsüblicher Hardware, die in vielen Heimcomputern zu finden ist.
Mit dieser Arbeit liegt außerdem ein fundierter Wegweiser vom physikalisch motivierten Hintergrund, über die verschiedenen mathematischen Formulierung bis hin zur Implementierung vor, der es dem interessierten Leser ermöglicht, volumetrische anisotrope Diffusionsfilter effizient und unabhängig von der verwendeten Hardwarearchitektur zu entwickeln.

Abstract

The presented work deals with the efficient implementation of volumetric diffusion filters on modern graphic cards. The focal point is on the speed advantages one can expect from a suitable GPGPU implementation and the resulting massively parallelized execution. It shows that volumetric diffusion filters can achieve from 40 up to 80 times the speed compared to pure CPU solutions – on mid-end commercial hardware, found in many home computers today.
This work is moreover a guideline, dealing with the physically motivated background and the different mathematical models up to the implementation, which enables the interested readers to develop their own volumetric anisotropic diffusion filters efficiently, regardless of the underlying hardware architecture.

Inhaltsverzeichnis

Inhaltsverzeichnis VII

Abbildungsverzeichnis X

Tabellenverzeichnis XII

1 Einleitung 1
- 1.1 Problemstellung und Eigenleistung . . . 2
- 1.2 Aufbau der Arbeit . . . 3

2 GPU Programmierung 4
- 2.1 Parallele Programmierung auf der GPU . . . 6
- 2.2 Nvidia CUDA . . . 7
- 2.3 Programmiermodell . . . 9
- 2.4 C für CUDA . . . 13
- 2.5 CUDA Filterbeispiel . . . 14

3 Mathematische Grundlagen 18
- 3.1 Diffusion . . . 18
- 3.2 Das 1. Fick'sche Gesetz . . . 20
- 3.3 Das 2. Fick'sche Gesetz . . . 22
- 3.4 Die n-dimensionale Diffusionsgleichung . . . 23

4 Diffusionsfilter in der Bildverarbeitung 24
- 4.1 Klassifikation der Diffusionsfilter . . . 24
- 4.2 Diffusion und Skalenraum Repräsentation . . . 25
- 4.3 Lineare homogene Diffusion . . . 27
- 4.4 Eigenschaften des linearen homogenen Diffusionsfilters . . . 28
- 4.5 Lineare inhomogene Diffusion . . . 30
- 4.6 Eigenschaften des linearen inhomogenen Diffusionsfilters . . . 30
- 4.7 Nichtlineare inhomogene Diffusion . . . 31
- 4.8 Diffusivität und Flussfunktion . . . 33
- 4.9 Nichtlineare anisotrope Diffusion . . . 34
- 4.10 Diffusions- und Strukturtensor . . . 36
- 4.11 Hesse-Matrix und Tangentialebene der Isooberfläche . . . 39
- 4.12 Algorithmus zur Diffusionstensorgewinnung . . . 43

4.13 Der »Custom made« - Diffusionstensor 44
4.14 Zusammenfassung 45

5 Prefilter **48**
5.1 Id Filter 49
5.2 Boxfilter 50
5.3 Gaußfilter 51
5.4 Medianfilter 53
5.5 Bilateraler Filter 54

6 Affine Transformationen **58**
6.1 Homogene Koordinaten 58
6.2 Fallbeispiel 59
6.3 Geschlossene Formel für affine Transformation 60
6.4 Implementierung 61

7 Implementierung **64**
7.1 Diskretisierung 64
7.1.1 Lineares homogenes Modell 65
7.1.2 Nachbarschaft und Randbedingungen 67
7.1.3 Inhomogenes isotropes Modell 69
7.1.4 Nichtlineares anisotropes Modell 71
7.2 Occupancy und Kernel–Voxel–Mapping 75
7.3 Texture memory 77
7.4 Separierbarkeit und texture memory 78
7.5 Numerische Stabilität 80

8 Ergebnisse **81**
8.1 Prefilter 82
8.2 Diffusions Filter 85
8.3 Affine Transformationen 87

9 Diskussion **89**
9.1 Diskussion der Messergebnisse 89
9.1.1 ID Filter 89
9.1.2 Prefilter 89
9.1.3 Diffusionsfilter 90
9.1.4 Affine Transformationen 91

9.2 Kritische Methodenreflexion 92
9.3 Zusammenfassung der wichtigsten Ergebnisse 93

10 Zusammenfassung und Ausblick **95**

Literaturverzeichnis **98**

Glossar **XIII**

Abkürzungsverzeichnis **XVI**

Symbolverzeichnis **XVIII**

A Pseudocode EED **XX**

B Herleitung der affinen Transformationsmatrix **XXVII**

C Messergebnisse **XXVIII**
C.1 ID Filter XXVIII
C.2 Prefilter XXVIII
C.3 Diffusions Filter XXXII
C.4 Affine Transformationen XXXIV

D Volumendatensätze **XXXVI**

Abbildungsverzeichnis

1 Nutzung der Transistoren auf CPUs und GPUs im Vergleich 4
2 Gleitkommaoperationen von CPUs und GPUs im Vergleich 5
3 Speicherbandbreite von CPUs und GPUs im Vergleich 6
4 Verteilung von Threadblöcken auf vorhandene GPU Kerne 9
5 Aufteilung eines Grids in Threadblöcke 11
6 Diffusion durch eine Grenzfläche G 19
7 Skalenraumrepräsentation eines Bildes 26
8 Lineare homogene Diffusion 28
9 Skalenraumrepräsentation einer MRT Slice durch Diffusion 29
10 Lineare homogene Diffusion 29
11 Nichtlineare inhomogene Diffusion 32
12 Skalenraumrepräsentation homogener und inhomogener Diffusion 32
13 Nichtlineare inhomogene Diffusion 33
14 Diffusivität und Flussfunktion 34
15 Nichtlineare anisotrope Diffusion (2D) 35
16 Nichtlineare anisotrope Diffusion (3D) 35
17 Basis auf der Tangentialebene der Isooberfläche 38
18 Projektion auf das Komplement des Spanns der Oberflächennormale 41
19 Diffusionsfilter im Vergleich (2D) 46
20 Diffusionsfilter im Vergleich (3D) 47
21 Id Filter 49
22 Boxfilter 50
23 Gaußfilter 51
24 Medianfilter 53
25 Aufbau des bilateralen Filterkerns 54
26 Visualisierung des bilateralen Filters 55
27 Bilateraler Filter 56
28 Edge enhancing diffusion mit bilateralem Prefiltering 57
29 Affine Transformation des Lobster Datensatzes 62
30 Darstellung des regulären Gitters und induzierter Fluss der 6-Nachbarschaft 67
31 Fluss der 8-Nachbarschaft eines 2D Grids 68
32 Fluss der 26-Nachbarschaft eines 3D Grids 68
33 Berechnung der diskreten 2. Ableitung auf einem regulären Gitter 73
34 Grid-Block-Voxel Mappings für ein $8 \times 8 \times 8$ Volumen 76
35 Geschwindigkeit der Prefilter im Vergleich 84

36 Geschwindigkeit der Diffusionsfilter im Vergleich 86
37 Geschwindigkeit der affinen Transformationen im Vergleich 88

Tabellenverzeichnis

1 Überblick über die verschiedenen Speicherarten in CUDA 12
2 Messergebnisse »ID Filter« 82
3 Messergebnisse »Boxfilter 1« (3 × 3 × 3 Kernel) 82
4 Messergebnisse »Boxfilter 2« (5 × 5 × 5 Kernel) 82
5 Messergebnisse »Boxfilter 1« (separiert, 5 × 5 × 5 Kernel) 83
6 Messergebnisse »Boxfilter 2 (separiert, 5 × 5 × 5 Kernel)« 83
7 Messergebnisse »Gaußfilter 1« (3 × 3 × 3 Kernel) 83
8 Messergebnisse »Gaußfilter 1« (separiert, 3 × 3 × 3 Kernel) 83
9 Messergebnisse »Gaußfilter 2« (separiert, 5 × 5 × 5 Kernel) 83
10 Messergebnisse »Medianfilter« (3 × 3 × 3 Nachbarschaft) 83
11 Messergebnisse »Bilateraler Filter« (3 × 3 × 3 Kernel) 83
12 Messergebnisse »Lineare homogene Diffusion« 85
13 Messergebnisse »Nichtlineare inhomogene Diffusion« 85
14 Messergebnisse »Nichtlineare Anisotrope Diffusion (EED)« 85
15 Messergebnisse »Affine Transformation (Nearest Neighbor)« 87
16 Messergebnisse »Affine Transformation (Trilinear)« 87
17 Messergebnisse »Affine Transformation (Cubic B-Splines)« 87
20 Messergebnisse »ID Filter« XXVIII
21 Messergebnisse »Boxfilter 1« XXIX
22 Messergebnisse »Boxfilter 2« XXIX
23 Messergebnisse »Boxilter 1 (separiert)« XXIX
24 Messergebnisse »Boxilter 2 (separiert)« XXX
25 Messergebnisse »Gaußfilter 1« XXX
26 Messergebnisse »Gaußfilter 1 (separiert)« XXX
27 Messergebnisse »Gaußfilter 2 (separiert)« XXXI
28 Messergebnisse »Medianfilter« XXXI
29 Messergebnisse »Bilateraler Filter« XXXI
30 Messergebnisse »Lineare homogene Diffusion« XXXII
31 Messergebnisse »Nichtlineare inhomogene Diffusion« XXXII
32 Messergebnisse »Nichtlineare Anisotrope Diffusion (EED)« XXXIII
33 Messergebnisse »Affine Transformation (Nearest Neighbor)« XXXIV
34 Messergebnisse »Affine Transformation (Trilinear)« XXXIV
35 Messergebnisse »Affine Transformation (Cubic B-Splines)« XXXV

1 Einleitung

Die vorliegende Arbeit beschäftigt sich mit der effizienten Umsetzung volumetrischer Diffusionsfilter auf modernen Grafikkarten. Die Klasse der Diffusionsfilter umfasst mehrere Filterausprägungen, die alle phyikalisch motiviert und mathematisch über eine partielle Differentialgleichung zweiter Ordnung beschreibbar sind. Im Rahmen dieser Arbeit werden alle wichtigen Aspekte der Implementierung auf der GPU erörtert und schließlich stellvertretend der »edge enhancing diffusion« Filter (EED) erarbeitet, welcher ohne Beschränkung der Allgemeinheit bereits alle wesentlichen Komponenten eines anisotropen »State of the Art« - Diffusionsfilters in sich vereint.
Im Vordergrund stehen dabei die zu erwartenden Geschwindigkeitsvorteile, die sich durch eine geschickte GPGPU[2] Implementierung und die damit einhergehende massiv parallelisierte Ausführung erzielen lassen.

Während der Vorbereitungen zu dieser Arbeit wurden zahlreiche Quellen unterschiedlicher Autoren gesichtet, die sich im Rahmen der Bildverarbeitung mit den verschiedenen Ausprägungen der – oftmals zweidimensional erläuterten – Diffusionsfilter beschäftigen und dabei die unterschiedlichsten Aspekte betrachten. Keine Quelle lieferte jedoch eine umfassende Einführung in den Themenkomplex, oder vermittelte das gesamte benötigte Hintergrundwissen für eine erfolgreiche und effiziente dreidimensionale Implementierung. Probleme bei der Verallgemeinerung auf drei Raumdimensionen bleiben in der Regel unausgesprochen.
Mit dieser Arbeit liegt nun ein fundierter Wegweiser vom physikalisch motivierten Hintergrund, über die verschiedenen mathematischen Formulierungen bis hin zur Implementierung vor, der einen roten Faden durch das gesamte Thema spannt und es dem interessierten Leser ermöglicht alle Schritte nachzuvollziehen, die benötigt werden, um volumetrische anisotrope Diffusionsfilter effizient – und zunächst unabhängig von der verwendeten Hardwarearchitektur – zu implementieren. Obwohl sich die Diskussions- und Messergebnisse auf die GPU beziehen, sind Theorie, Diskretisierung und die wichtigsten Implementierungsschritte auch für die Umsetzung auf der CPU geeignet, was Lesern ohne GPGPU Hintergrund ebenfalls einen fundierten Einstieg in die Thematik verschafft.

[2] »General-purpose Computation on Graphics Processing Units«, kurz GPGPU, beschreibt eine technische Entwicklung der letzten Dekade, bei der die besondere Hardwarearchitektur moderner Grafikkarten für Berechnungen ausserhalb der Computergrafik genutzt wird. Moderne Grafikkarten besitzen bereits mehrere hundert Streamprozessoren und der stetig wachsende Grafikkartenspeicher erlaubt es, immer größere Datensätze direkt auf der Grafikkarte zu verarbeiten. Die GPU wird dabei sozusagen als leistungsfähiger Co-Prozessor zweckentfremdet.

Im Rahmen dieser Arbeit wurde ein Prototyp erstellt, welcher alle in den kommenden Kapiteln besprochenen Filter und Transformationen in einer CPU und GPU Variante enthält und mit den erhaltenen Messergebnissen die Grundlage der Diskussion liefert. Es zeigt sich, daß volumetrische anisotrope Diffusionsfilter auf Grafikkarten je nach Anwendungsfall von 40 bis 80-fachen Geschwindigkeitssteigerungen verglichen mit reinen CPU Lösungen profitieren können – und das auf handelsüblicher Hardware, die in vielen Heimcomputern zu finden ist.

1.1 Problemstellung und Eigenleistung

Die Problemstellung dieser Arbeit lässt sich kompakt in den folgenden Fragen zusammenfassen:

1. Wie lässt sich die Klasse der volumetrischen Diffusionsfilter mathematisch beschreiben und diskretisieren?
2. Wie lassen sich volumetrische Diffusionsfilter effizient auf der GPU implementieren?
3. Wie hoch ist der theoretische und praktische Geschwindigkeitsgewinn einer solchen Implementierung?
4. Wie können mögliche Vorverarbeitungsschritte (Transformationen und Prefilter), im Wesentlichen der sogenannte »bilaterale Filter«, ebenfalls auf der GPU implementiert werden und welche Geschwindigkeitsvorteile sind hier zu erreichen?

Die Beantwortung dieser Fragen ergibt sich einerseits durch die Bereitstellung einer vollständigen Herleitung der mathematischen Formulierung verschiedener Ausprägungen volumetrischer Diffusionsfilter, andererseits durch die Diskussion der Messungen des implementierten Prototyps. Die Erstellung des Prototypen impliziert, dass die kontinuierlich formulierten mathematischen Modelle angemessen diskretisiert werden müssen und auch konkrete GPGPU Techniken erörtert werden, die eine effiziente Implementierung überhaupt erst ermöglichen.

Die Herleitung und Diskretisierung geschieht im Gegensatz zu gängiger Literatur von Anfang an auf dreidimensionalen Strukturen und überlässt es nicht dem Leser eine Verallgemeinerung auf die dritte Dimension selbst vorzunehmen.
Im Rahmen der nichtlinearen anisotropen Diffusion nimmt dabei der sogenannte Diffusionstensor eine zentrale Rolle ein, zu dessen Berechnung eine Eigenwertzerlegung erforderlich ist. Diese Eigenwertzerlegung ist im Zweidimensionalen, ganz im Gegensatz zur volumetrischen Problemstellung, einfach über geschlossene Formeln berechenbar. Die vorliegende Arbeit zeigt dem Leser detailliert einen Weg auf, die benötigten Eigenwerte durch

eine geschickte Projektion vom dreidimensionalen Objektraum in den zweidimensionalen Tangentialraum der lokalen Isooberfläche effizient zu berechnen.

1.2 Aufbau der Arbeit

Zunächst wird im Kapitel »GPU Programmierung« ein kurzer Abriss über die parallele Programmierung auf Grafikkarten gegeben und die »CUDA« (Compute Unified Device Architecture) Technologie des Grafikchipherstellers Nvidia vorgestellt. Das Kapitel stellt in groben Zügen die wichtigsten Eigenschaften des CUDA Programmiermodells vor und endet in einem kleinen Beispiel, das einen vollständigen naiv implementierten Box-Filter in C für CUDA angibt.
Danach wird im Kapitel »Mathematische Gundlagen« die physikalisch motivierte Basis in Form einer mathematischen Herleitung der allen Diffusionsfiltern zugrunde liegenden dreidimensionalen Wärmeleitungsgleichung besprochen.
Das Kapitel »Diffusionsfilter in der Bildverarbeitung« greift diese allgemeine dreidimensionale partielle Differentialgleichung wieder auf und arbeitet darauf die Formulierungen der verschiedenen in der Bildverarbeitung eingesetzten Diffusionsfilter weiter aus. Es folgt das Kapitel »Prefiltering«, das nach einer kurzen Motivation und Einordnung kompakt einige der möglichen Vorfilter vorstellt und schließlich den sogenannten »bilateralen Filter« erläutert.
Das Kapitel »Affine Transformationen« ist ein persönlich motivierter Exkurs, der untersucht, welche Potentiale die GPU Implementierung linearer Abbildungen birgt.
Im Kapitel »Implementierung« wird im Wesentlichen die Diskretisierung der kontinuierlich formulierten Diffusionsfilter besprochen, sowie auf einige programmiertechnische Fragestellungen eingegangen.
Schließlich werden in den Kapiteln »Ergebnisse« und »Diskussion« die Messergebnisse des Prototyps vorgestellt und besprochen.
Die wichtigsten Schlussfolgerungen und der Inhalt der Arbeit werden im Kapitel »Zusammenfassung und Ausblick« noch einmal abschließend dargestellt und weitere mögliche Fragestellungen für künftige Entwicklungen aufgegriffen.

2 GPU Programmierung

Seit den Anfängen der programmierbaren Grafikpipeline um die Jahrtausendwende hat sich die Flexibilität moderner Grafikkarten enorm gesteigert. Die Hersteller Nvidia (24,3% Marktanteil Q4 2009) und AMD (vormals ATI, 19,9% Marktanteil Q4 2009) teilen, lässt man die Firma Intel (55,2% Marktanteil) mit einem breiten Spektrum an integrierten GPUs und low-end Consumer Produkten außen vor, den Markt für high-performance Grafikkarten defakto vollständig unter sich auf und beeinflussen seit vielen Jahren maßgeblich die technologische Entwicklung (s. [Ped10]). Eine Schlüsselrolle auf dem Weg zur GPU Programmierung nahmen die programmierbaren Vertex- und Fragmentshader ein, die erstmals das Ausführen beliebiger Programme auf der Grafikhardware erlaubten. Nvidia veröffentlichte im März 2001 mit der NV20-GPU (eingesetzt in Grafikkarten der GeForce 3 Serie) den ersten Grafikchip mit programmierbaren Shadereinheiten, ATI antwortete im Oktober 2002 mit der Einführung des R300 Chips auf der ATI Radeon 9700, der frei programmierbare Pixel- und Vertexshader unterstützte, die bereits Fließkommaarithmetik und Schleifen erlaubten.
Die Weiterentwicklung von assemblerartigen Shadersprachen zu Hochsprachen setzte ein und die Flexibilität der auf GPUs ausführbaren Berechnungen wurde immer weiter gesteigert. Im Jahr 2004 erlangte das Thema »General-Purpose Computation on Graphics Processing Units«, kurz GPGPU, insbesondere durch einen auf der SIGGRAPH 2004 angebotenen Kurs breitere Bekanntheit. Mit der Veröffentlichung von mehreren Beiträgen zum Thema GPGPU in [Pha05] und später in [LHG+06] stand ab März 2005 auch ein umfassendes Rahmenwerk zur GPU Programmierung zur Verfügung.

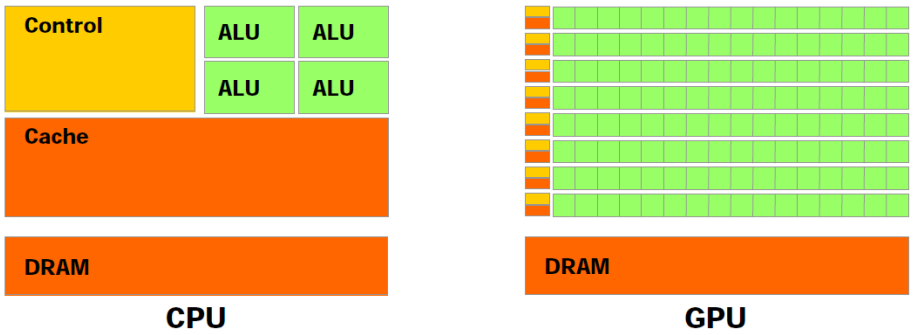


Abbildung 1: Nutzung der Transistoren auf CPUs und GPUs im Vergleich. Aus [Nvi10, S. 3]

Heute, zehn Jahre nachdem diese Entwicklung in Gang gesetzt wurde, kann die moderne GPU unstrittig als Streamprozessorarray aufgefasst werden, eine Art Co-Prozessor, der Datenströme hochgradig parallel verarbeiten kann. »GPUs are now viewed as inexpensive coprocessors that are ideally suited for many applications beyond computer graphics«

(Steele und Cochran in [SC07]).
Man spricht in diesem Zusammenhang von SIMD-Architekturen (single instruction mutliple data), da massive Datenparallelität ausgenutzt wird. Klassisch ist diese Entwicklungsstufe aus der unabhängigen parallelen Verarbeitung von Dreiecksnetzen und Bildpunkten hervorgegangen.
Während viele Probleme sequentiell auf der CPU berechnet werden müssen, da die Instruktionen und Ausführungspfade von den Zwischenergebnissen abhängen, bietet sich bei datenparallelen Problemen die GPU als SIMD Prozessor an. Sofern man CPUs und GPUs überhaupt vergleichen kann – Taktzyklen und Design sind sehr unterschiedlich – lässt sich dennoch festhalten, dass ein Großteil der Transistoren einer CPU dazu verwendet wird Ausführungspfade und Programmsteuerung bzw. das Datencaching zu optimieren. Nur ein relativ kleiner Anteil der Chipfläche führt tatsächlich Berechnungen aus. Demgegenüber steht das Design moderner Grafikchips, welche arithmetische Operationen auf vielen Daten gleichzeitig ausführen und eine viel höhere Nutzung der zur Verfügung stehenden Transistoren für Berechnungen aufweisen (s. Abb. 1).
Die reine Rechenleistung von GPUs, ausgedrückt in Gleitkommaoperationen pro Sekunde (Floating point operations per second, FLOPS) durchbricht im Consumer Markt seit 2009 längst die TerraFLOPS Grenze [Gmb09], aber auch die Bandbreite des Speichers übersteigt die Leistung moderner CPUs mittlerweile bei weitem (s. Abb. 2 und Abb. 3).

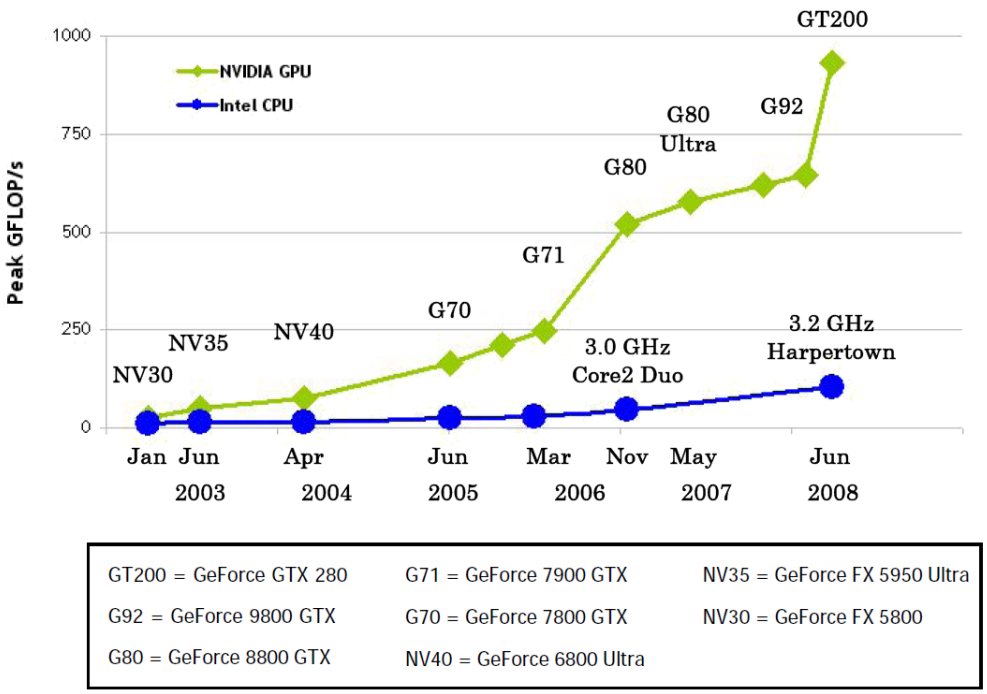


Abbildung 2: Gleitkommaoperationen von CPUs und GPUs im Vergleich. Aus [Nvi10, S. 2]

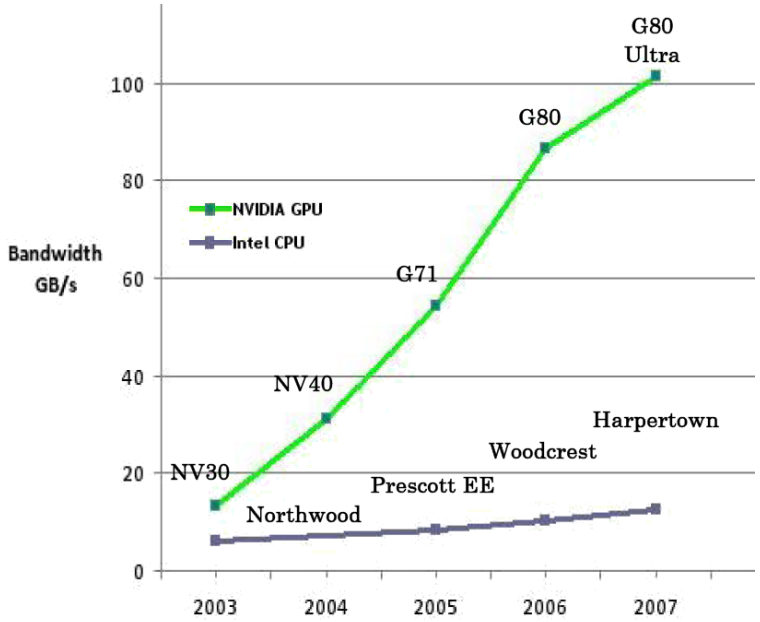


Abbildung 3: Speicherbandbreite von CPUs und GPUs im Vergleich. Aus [Nvi10, S. 2]

Die Programmierung von GPUs ist jedoch keine Alternative zur klassischen Programmierung auf CPUs, vielmehr sind die Problemklassen unterschiedlich und nur gewisse Problemstellungen können überhaupt effizient auf der Grafikhardware bearbeitet werden.
Im Folgenden sollen einige allgemeine Aspekte zur parallelen Programmierung auf Grafikkarten näher beleuchtet und das Programmiermodell der von Nvidia entwickelten CUDA Technologie vorgestellt werden.

2.1 Parallele Programmierung auf der GPU

Das Schreiben paralleler Programme auf GPUs stellt den Entwickler vor besondere Herausforderungen. Neben den technischen Details ist insbesondere die Frage zu klären, welche Berechnungen gut auf die Architektur der GPU abgebildet werden können.
In [Har05] und [Har07] arbeitet Harris heraus, dass zwei Aspekte essentiell für die erfolgreiche Umsetzung von Problemen auf der GPU sind: Einerseits sollten die gleichen oder ähnliche Operationen auf dem Datenstrom ausgeführt werden, andererseits sollten die Berechnungen selbst wenige oder keine Abhängigkeiten zu anderen Daten aufweisen. Beides fasst er in einem Konzept, der »arithmetischen Intensität«, zusammen: Arithmetische Intensität »...is the ratio of computation to bandwidth, or more formally: Arithmetic intensity = operations / words transferred.«. Im Falle der GPU ist der arithmetische Durchsatz optimiert, während viel weniger Transistoren für die Verwaltung einer Speicherhierarchie und Caches verwendet werden. Harris konkludiert: »GPUs demand high arithmetic intensity for peak performance« [Har05].

Für den Programmierer bedeutet das schlicht und einfach die Speicherzugriffe minimal zu halten, die vorhandenen Register maximal zu nutzen und die eigentlichen Berechnungen zu maximieren, ganz praktisch zum Beispiel durch die Neuberechnung von Werten statt deren Zwischenlagerung im Hauptspeicher. Techniken, die klassische CPU Programme enorm beschleunigen, wie z.B. die Ablage vorberechneter Werte in sogenannten Lookup-Tables, führen bei der GPU Programmierung zu massiven Geschwindigkeitseinbrüchen. Während einfache Integer- und Gleitkommaoperationen in wenigen Taktzyklen parallel abgearbeitet werden, blockiert ein Zugriff auf den Grafikkartenspeicher mehrere hundert Taktzyklen lang die weitere Ausführung.
Durch den Aufbau der GPU sind Operationen, die Daten blockweise verarbeiten, in der Regel sehr viel effizienter als einzelne Speicherzugriffe, da die internen Komponenten der Grafikkarte auf solche Verarbeitungsschritte hin optimiert sind. Probleme die sich beispielsweise auf einem Grid darstellen lassen, können meist sehr gut auf der GPU abgebildet werden: Die Texturspeicher bieten hier eine natürliche Datenstruktur für die Ablage von Werten.
Dennoch ist zu beachten, dass keine noch so gute Abstraktion die Tatsache verdeckt, dass eine GPU intern anders arbeitet als ein klassischer Prozessor. Technologien wie »ATI Stream« oder das im Folgenden vorgestellte »CUDA«, die dem Programmierer die Funktionen der unterliegenden Grafikhardware leichter zugänglich machen, legen dem Programmierer teilweise exotisch anmutende Restriktionen auf. Trotz der stetig andauernden Weiterentwicklung spürt man auch eine Dekade nach Entwicklungsbeginn der programmierbaren GPU noch historisch begründete Altlasten. Beispielsweise kündigte Nvidia erst Ende 2009 Grafikkarten an, die Fließkommaarithmetik in double precision nach dem IEEE 754-2008 Floating-Point Standard ermöglichen (vgl. [Nvi09c]).
Für viele Anwendungen aus Medizin, Naturwissenschaft und Wirtschaft ermöglichen Technologien wie »ATI Stream« und »CUDA« jedoch den schnellen Einstieg in die Materie und ermöglichen die effiziente Implementierung von entsprechenden Algorithmen - sofern sich diese wirklich als parallelisierbar im Sinne der GPU Programmierung erweisen.

2.2 Nvidia CUDA

Die Firma Nvidia erkannte schon vor einigen Jahren das Potential, das in den immer flexibler programmierbaren Grafikpipelines lag und begann den Entwicklungsfokus von reiner Geschwindigkeit für 3D Applikationen und Computerspiele hin zu allgemeineren Berechnungen zu verlagern. Einige Designentscheidungen sind bewusst zu Gunsten der Flexibilität statt kurzfristiger Geschwindigkeitssteigerungen gefällt worden. Nvidia hat natürlich ein wirtschaftliches Interesse daran sich langfristig auf dem relativ jungen »GPGPU Markt« zu behaupten, und so entwickelte man die im November 2006 vorgestellte CUDA -

Technologie (Compute Unified Device Architecture) [Nvi10, S. 3]:
Die Streamprozessoren sind dabei seit der GeForce 8 Grafikkarten Serie über eine Schnittstelle programmierbar und Nvidia veröffentlichte zu den Treibern ein entsprechendes API (Application Programming Interface), sowie den CUDA Compiler Driver »NVCC«, der C für CUDA ermöglicht. C für CUDA ist eine Erweiterung der Programmiersprache C um Sprachelemente, die es erlauben Code für die Streamprozessoren CUDA fähiger Grafikkarten in C ähnlicher Syntax zu schreiben. Der NVCC Compiler zerlegt diesen gemischten Quellcode in sogenannten »Host-Code«, der klassich an den normalen C Compiler weitergereicht wird und den sogenannten »Device Function Code«, der durch die Compiler von Nvidia in Device Code umgesetzt wird. Im offiziellen CUDA Compiler Driver Handbuch heisst es hier:

»Hence, source files for CUDA applications consist of a mixture of conventional C++ 'host' code, plus GPU 'device' (i.e. GPU-) functions. The CUDA compilation trajectory separates the device functions from the host code, compiles the device functions using proprietory NVIDIA compilers/assemblers, compiles the host code using any general purpose C/C++ compiler that is available on the host platform, and afterwards embeds the compiled GPU functions as load images in the host object file. In the linking stage, specific CUDA runtime libraries are added for supporting remote SIMD procedure calling and for providing explicit GPU manipulation such as allocation of GPU memory buffers and host-GPU data transfer.« [Nvi07, S. 1]

Der Erfolg dieser Architektur liegt sicherlich zum einen in der relativ einfachen Spracherweiterung von C, sowie dem durchaus angemessenen Arbeitsumfang, der benötigt wird, um erste Programme für die GPU zu schreiben. Hat man ein wenig Erfahrung im Umgang mit C ist der Einstieg in CUDA in wenigen Stunden möglich. Die große, weltweite Entwicklergemeinde, die strukturierten und gut besuchten Nvidia Entwicklerforen und die zahlreichen im Internet verfügbaren Beispielprogramme tragen Ihren Teil dazu bei. Auch wenn die CUDA Technologie noch recht jung ist, scheint sie sich, gerade im wissenschaftlichen Bereich, als Standard durchzusetzen. Viele Vorlesungen an Universitäten rund um den Globus erörtern allgemeine Themen zu parallelen Problemen und Fragestellungen aus dem Bereich der GPGPU konkret an Nvidia CUDA. Es ist zu erwarten, dass dieser Trend in den kommenden Jahren anhalten wird, zumal für das einzige derzeitige Konkurrenzprodukt, ATI Stream [AMD10], schlichtweg die Anwendungen fehlen (s. z.B. [BK09]). Die vorliegende Arbeit implementiert die vorgestellten Algorithmen auf Basis der Nvidia CUDA Technologie. Im Folgenden werden das CUDA Programmiermodell und die Spracherweiterungen von C für CUDA vorgestellt.

2.3 Programmiermodell

Das skalierbare Programmiermodell von CUDA basiert im wesentlichen auf drei Abstraktionen: Einer Hierarchie und Gliederungsstruktur von Threads, verschiedene Speicherarten, sowie Synchronisationsmechanismen für die parallel arbeitenden GPU Threads.
Die einzelnen GPU-Threads werden »Kernels« genannt. Ein Kernel ist damit die kleinste Einheit der Threadhierarchie, in deren Mittelpunkt die Überlegung steht, die jeweiligen Teilaufgaben von Threadgruppen, sogenannten »Blocks«, lösen zu lassen. Die Blocks können vom Laufzeitsystem auf die zur Verfügung stehenden Multiprozessor-Kerne der GPU, im Folgenden MPs genannt, verteilt werden. Jeder dieser MPs besteht in aktuellen Grafikkarten aus 8 SPs (Streamingprozessoren, klassisch als »Shading Units« bekannt). Das Laufzeitsystem hat so die Möglichkeit vorhandene Ressourcen voll auszunutzen und die Blöcke unabhängig voneinander auf der konkreten Hardware ablaufen zu lassen.

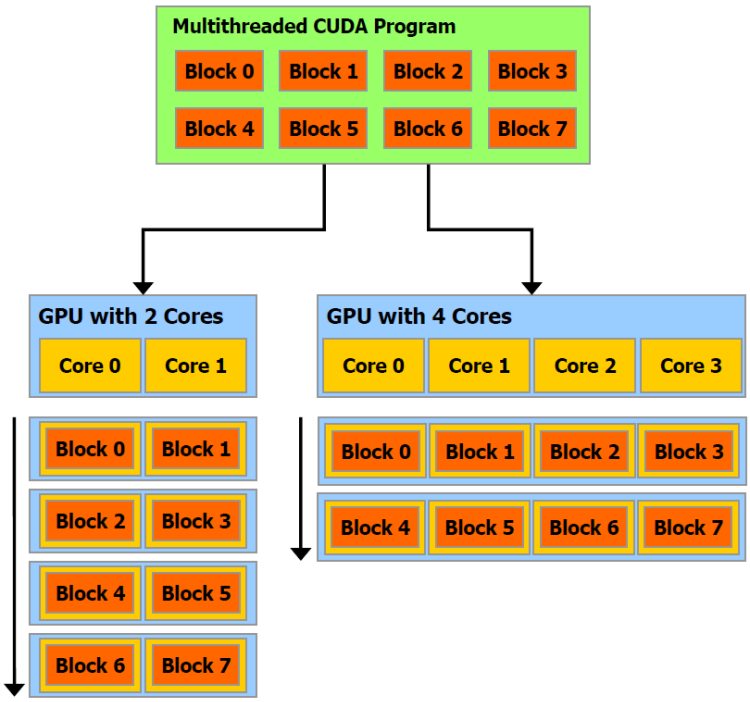


Abbildung 4: Verteilung von Threadblöcken auf vorhandene GPU Kerne. Aus [Nvi10, S. 5]

Auf diese Weise profitieren Systeme mit CUDA fähigen High-End Grafikkarten von der Vielzahl der zur Verfügung stehenden MPs. Der Programmierer ist auf der anderen Seite vom Design der Grafikkarte insofern losgelöst, als dass er bestenfalls um die Anzahl der SPs wissen muss (eine Information die zur Laufzeit abgefragt werden kann), nicht aber wieviele MPs im Einzelnen vorhanden sind: Um die Leistung der Grafikkarte optimal auszunutzen, sollten möglichst immer mindestens soviele Blöcke wie die GPU MPs besitzt gleichzeitig

ablaufen.

Die CUDA Spezifikation sieht vor, dass alle Threads eines Blocks auf genau einem MP ablaufen [AC08], was sich dann auch in der Speicherhierarchie widerspiegelt: Alle Threads eines Blocks haben den gleichen shared memory. Der Block bildet damit innerhalb der CUDA Abstraktion das Pendant zum einzelnen MP. Die Abarbeitung der Blöcke hingegen wird vollkommen autonom vom Laufzeitsystem bestimmt, hier können keine Annahmen über die zeitliche Reihenfolge getroffen werden. Im CUDA Programming Guide heißt es: »Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores ... enabling programmers to write code that scales with the number of cores.« [Nvi10, S. 10]

Die Blöcke selbst werden wiederum in einer größeren Struktur, dem sogenannten Grid, zusammengefasst. Die Nummerierung bzw. Indizierung der Kernels kann ein-, zwei- oder dreidimensional erfolgen, da viele Probleme aus Wissenschaft und Wirtschaft so leichter umzusetzen sind. Adressiert man beispielsweise die Kernels mit drei Indizes, kann man sich vorstellen, dass der Block dreidimensionale Struktur hat. Blöcke können zur Zeit nur über maximal zwei Indizes adressiert werden, das Grid ist also maximal als zweidimensionale Struktur vorstellbar. Diese Restriktion erfordert für dreidimensionale Daten, wie in der vorliegenden Arbeit, eine spezielle Behandlung des Block - Daten - Mappings, was im Abschnitt »CUDA Filterbeispiel« (Ab Seite 14) noch einmal angesprochen wird.

Da sich alle Kernel (Kernel = GPU-Thread) eines Blocks die zur Verfügung stehenden Ressourcen, insbesondere die Register, teilen müssen, ist es essentiell wichtig zu entscheiden, wieviele Kernel benötigt werden. Nvidia gibt hierzu im »CUDA Programming Best Practices Guide« [Nvi09a] ausführliche Hilfestellungen bei der Wahl der sogenannten »Execution Configuration«, die festlegt wieviele Kernel per Block und wieviele Blöcke per Grid angelegt werden sollen. Es sollte unbedingt beachtet werden, dass ein Kernel nicht mit einem klassischen CPU Thread vergleichbar ist:
Moderne Grafikkarten bieten zwischen 128 SPs (G80 Serie, 16 MPs × 8 SPs, 680 Mio. Transistoren) und 240 SPs (GT200, 30 MPs × 8 SPs, 1400 Mio. Transistoren). Die nächste angekündigte Chip-Generation, zur Zeit unter dem Codenamen »Fermi« bekannt, soll 512 SPs (16 SMs × 32 SPs) mit Level 1 und Level 2 Cache, 64 Bit Address Bus und Double Precision Floating Point bei rund 3000 Mio. Transistoren mit sich bringen. [Cor09, S. 11]
In [Nvi09a] heisst es: »All NVIDIA GPUs can support 768 active threads per multiprocessor, and some GPUs support 1,024 active threads per multiprocessor. On devices that have 30 multiprocessors (such as the NVIDIA® GeForce® GTX 280), this leads to more than

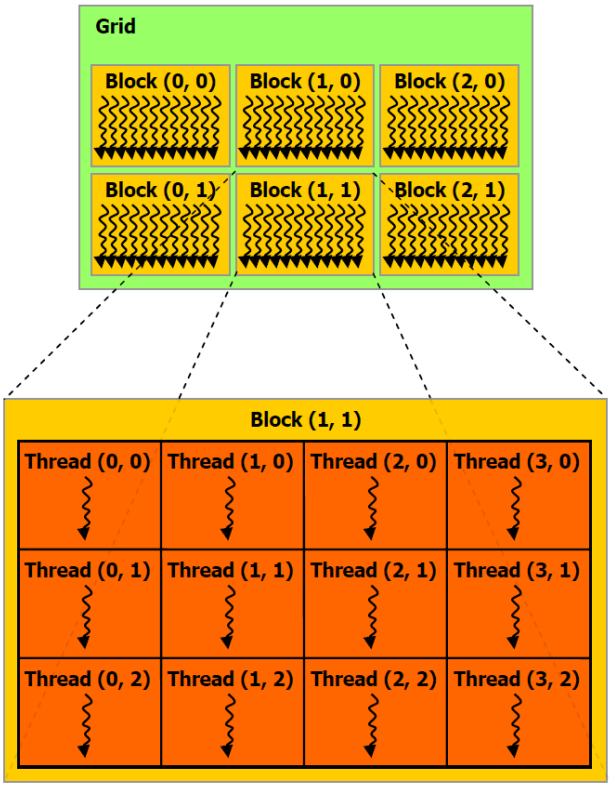


Abbildung 5: Aufteilung eines Grids in Threadblöcke. Aus [Nvi10, S. 9]

30,000 active threads. In addition, devices can hold literally billions of threads scheduled to run on these GPUs.«

Im Gegensatz zu CPU Threads, bei denen eine Quadcore CPU maximal 4 oder bei Hyperthreading 8 Threads gleichzeitig ausführen kann, handelt es sich also bei GPU Kernels um sehr leichtgewichtige Einheiten. Der Hardware Scheduler teilt die Blöcke intern in sogenannte »Warps« (Einheiten von 32 Threads) auf und weist diese den SPs optimal zu. Durch schnelle Wechsel der Threads können auftretende Latenzen (zum Beispiel das Warten auf Speicherinhalte) relativ gut kaschiert werden. Auch wenn das Wissen um die internen Vorgänge die Optimierung des Codes erleichtert, nimmt die CUDA Architektur und der Hardware Scheduler dem Programmierer einen Großteil der Arbeit ab.

So wie die Organisation der Kernels in Blöcke und Grids erfolgt, um der unterliegenden Architektur zu entsprechen und dennoch den Anforderung an ein abstrahierendes Modell Rechnung zu tragen, ist auch das Speichermodell hierarchisch aufgebaut:

Jeder Kernel besitzt zunächst seine Register. Diese sind jedoch nur auf wenige Bytes beschränkt, da in aktuellen Chipgenerationen nur ca. 8 - 16 KB Speicher pro SP zur Verfügung stehen und bis zu 768 Kernel gleichzeitig aktiv sein können, was im Endeffekt nur ca. 10 - 20 Register pro Kernel bedeutet. Die nächste Ebene der Speicherhierarchie bildet der soge-

nannte »shared memory«. Dieser Speicher ist ebenfalls im SP (also »on-chip«) lokalisiert, auf 16 KB begrenzt und dient, wie der Name vermuten lässt, allen Kernels eines Blocks als gemeinsamer Speicherbereich. Die Register und der shared memory sind die beiden schnellsten zur Verfügung stehenden Speicher.

Der off-chip Speicher lässt sich auf verschiedene Weisen nutzen. Allgemein kann jeder Thread auf den vollen off-chip Speicher (bei Consumer Produkten zur Zeit zwischen 512 MB bis 1 GB) lesend und schreibend zugreifen. Diese Nutzung wird als »global memory« bezeichnet. Theoretisch kann auch hier jeder einzelne Thread bis zu 16 KB für eigene Berechnungen reservieren, diese Nutzung wird als »local memory« bezeichnet, ist aber entsprechend langsam.

Allgemein ist der Zugriff auf den off-chip Speicher mit 400 bis 600 Taktzyklen Wartezeit extrem langsam, kann jedoch durch sogenannten »coalesced access« beschleunigt werden: Werden von einem Half-Warp (16 Threads) konsekutive Speicherzellen abgefragt, können diese parallel an den SP übertragen werden, was die Geschwindigkeit bei der Kommunikation mit dem global memory signifikant erhöht.

Da aber nicht immer die Voraussetzungen für den coalesced access geschaffen werden können – beispielsweise ist es in der vorliegenden Arbeit notwendig gewesen räumlich benachbarte Werte eines 3D Volumens zu verarbeiten, die nicht linear im Speicher arrangiert werden können – gibt es noch die Möglichkeit Speicherbereiche des global memory als »constant memory« oder »texture memory« auszuweisen und so die internen Caching-Mechanismen der Grafikkarte zu aktivieren.

Eine Zusammenfassung der verschiedenen Speicherarten findet sich in [Nvi09a, S. 20], siehe Tabelle 1.

Memory	Location	Cached	Access	Scope	Lifetime
Register	on chip	n/a	R/W	1 thread	Thread
Local	off chip	No	R/W	1 thread	Thread
Shared	on chip	n/a	R/W	All threads in block	Block
Global	off chip	no	R/W	All threads + host	Host allocation
Constant	off chip	yes	R	All threads + host	Host allocation
Texture	off chip	yes	R	All threads + host	Host allocation

Tabelle 1: Überblick über die verschiedenen Speicherarten in CUDA

2.4 C für CUDA

C für CUDA führt zunächst sogenannte »function type qualifier« ein: *__device__*, *__global__* und *__host__*. Der »host-qualifier« ist dabei der Standardtyp und spezifiziert Host[3]-Funktionen, die nur für die CPU kompiliert werden. Um eine C Funktion als Kernel zu deklarieren, die als GPU Code compiliert wird, deren Einsprung jedoch vom Host aus initiiert werden kann, verwendet man den »global-qualifier«. Der »device-qualifier« deklariert schließlich Funktionen, die im GPU Code vorliegen und die entweder von global oder anderen device Funktionen aufgerufen werden können.

Viele Techniken, die im Host-Code problemlos möglich sind, können auf der GPU nicht umgesetzt werden und führen zu Restriktionen bei der Umsetzung eines konkreten Algorithmus. Hierzu zählen insbesondere: Rekursion, Ellipsen (variable Anzahl an Parametern) und statische Variablendeklarationen. Auch das Anlegen von device function pointers oder die Möglichkeit der Rückgabe von Funktionswerten von global Funktionen ist nicht möglich.

Analog zu Funktionen, können auch Variablen näher spezifiziert werden. Hierzu stehen *__device__*, *__constant__*, *__shared__* und volatile zur Verfügung. Genauere Informationen zur deren Funktionsweise findet man in [Nvi10, S. 99]

Der Aufruf eines Kernels erfolgt über die etwas eigenwillig anmutenden Syntax:

```
1 KernelName<<<blocksPerGrid , threadsPerBlock>>>(Parameter...);
```

Innerhalb des Device Codes stehen die »built-in Variablen« *gridDim* und *blockDim* bzw. *blockIdx* und *threadIdx* zur Verfügung, welche Informationen über die Grid und Block Dimensionen bzw. die aktuellen Block und Thread Indizes zur Verfügung stellen.

Die wichtigsten Funktionen für das Speichermanagement sind *cudaMalloc*, um vom Host aus global memory auf dem Device zu allokieren, sowie *cudaMemcpy*, um Daten zwischen Speicherbereichen des Hosts und des Devices oder innerhalb des Devices zu transferieren. Die Aufrufprototypen dieser Funktionen lauten:

```
1 cudaMalloc(void** ptr, size_t size)
2 cudaMemcpy(void* dst, const void* src, size_t size, enum direction)
```

[3] Der »Host« bezeichnet in CUDA den eigentlichen PC mit CPU und Hauptspeicher. Demgegenüber steht die Grafikkarte, das »Device«. Man spricht von Host-Code bzw. Device- oder GPU-Code, wenn der erzeugte Maschinencode auf der CPU bzw. der GPU ausgeführt wird.

Die letzte wichtige Neuerung betrifft die Funktionen zur Threadsynchronisation: Auf dem Host wird dazu cudaThreadSynchronize() und im Devicecode __syncthreads() verwendet. Obwohl eine Unmenge weiterer Funktionen und Sprachelemente zum Timing, Scheduling und der Code- und Speichersynchronisation existieren, sind im Rahmen dieser Vorstellung alle wesentlichen Elemente benannt. Abschließend soll in diesem Kapitel ein kleines Beispielprogramm skizziert werden, das einen volumetrischen $3 \times 3 \times 3$ Boxfilter in C für CUDA realisiert.

2.5 CUDA Filterbeispiel

Wir kennen nun bereits alle Mechanismen, die für die Angabe eines kleinen parallelen Programmes notwendig sind. Im diesem Abschnitt soll ein CUDA Grundgerüst skizziert werden, dem vom Prinzip her alle in dieser Arbeit evaluierten Filter in ihrer Implementierung folgen.

Der Host-Code eines Boxfilters könnte wie folgt aussehen:

```
1  void processBoxFilterGPU(
2    float *pcfIn, float *pcfOut,
3    int iDimX, int iDimY, int iDimZ
4  ) {
5    cudafloat *pcfDevIn  = NULL;
6    cudafloat *pcfDevOut = NULL;
7    size_t memSize;
8
9    int iBlocksNeededSlice = (iDimX / 8) * (iDimY / 8);
10   int iBlocksNeededDepth = (iDimZ / 8);
11
12   dim3 dimBlock(8, 8, 8);
13   dim3 dimGrid(iBlocksNeededSlice, iBlocksNeededDepth);
14
15   memSize = iDimX * iDimY * iDimZ * sizeof(float);
16   cudaMalloc((void **) &pcfDevIn, memSize);
17   cudaMalloc((void **) &pcfDevOut, memSize);
18   cudaMemcpy(pcfDevIn, pcfIn, memSize, cudaMemcpyHostToDevice);
19   processBox<<<dimGrid, dimBlock>>>(pcfDevIn, pcfDevOut,
20                                     iDimX, iDimY, iDimZ);
21   cudaThreadSynchronize()
22   cudaMemcpy(pcfOut, pcfDevOut, memSize, cudaMemcpyDeviceToHost);
23
24   cudaFree(pcfDevIn);
25   cudaFree(pcfDevOut);
26 }
```

Dieses kurze Beispiel zeigt bereits den typischen grundlegenden Ablauf eines jeden CUDA Programmes:

1. Die Allokation eines Speicherbereiches auf dem Device (der Grafikkarte) durch den Host (die CPU) in den Codezeilen 16 und 17.
2. Den Transfer der Daten vom Hauptspeicher der CPU in den off-chip Speicher der Grafikkarte in Codezeile 18.
3. Der Aufruf des Device Codes (der Kernel) mit einer passenden »execution configuration«, welche die Anzahl der Kernel und Blöcke festlegt in den Codezeilen 19 und 20. Dieser Aufruf ist asynchron, das heisst, die Kontrolle kehrt direkt zum Host zurück, der nun andere Aufgaben berechnen kann oder wie hier per barrier synchronisation die Terminierung aller Kernel abwartet (Codezeile 21).
4. Danach können etwaige Fehlercodes abgefragt werden und die Ergebnisse vom off-chip Speicher zurück in den Hauptspeicher der CPU übertragen (Codezeile 22).
5. Schließlich erfolgt die Freigabe des allokierten Device Speichers durch den Host in den Codezeilen 24 und 25.

Da dieses Beispiel bereits für einen dreidimensionalen Datensatz gelten soll, muss die Ausführungskonfiguration entsprechend angepasst werden: Es sollen pro Block $8 \times 8 \times 8 = 512$ Kernel erzeugt werden. Dabei wird hier angenommen, dass die Daten in einem über den Pointer pcfIn referenzierten Host-Speicherbereich liegen und die Dimensionen des Datensatzes alle Vielfache von 8 sind. Für einen Datensatz mit 256^3 Elementen benötigen wir also ein Grid mit $32 \times 32 \times 32$ Blöcken. Da ein Grid in CUDA zur Zeit nur zweidimensional angelegt werden kann müssen wir für ein entsprechende Mappingfunktion sorgen und legen es mit den Dimensionen $(32 \times 32, 32)$an.

Der zugehörige Kernel könnte nun wie folgt aussehen:

```
1  __global__ static void processBox(
2    cudafloat *pcfIn, cudafloat *pcfOut,
3    int iDimX, int iDimY, int iDimZ
4  ) {
5    int iBlockX = blockIdx.x % (iDimX / blockDim.x);
6    int iBlockY = blockIdx.x / (iDimX / blockDim.x);
7    int iBlockZ = blockIdx.y;
8
9    int iVoxelX = threadIdx.x + iBlockX * blockDim.x;
10   int iVoxelY = threadIdx.y + iBlockY * blockDim.y;
11   int iVoxelZ = threadIdx.z + iBlockZ * blockDim.z;
```

```
12
13   int iX, iY, iZ;
14   cudafloat cfSum = 0.0;
15   int iIndex      = iVoxelX +
16                     iVoxelY * iDimX +
17                     iVoxelZ * iDimX * iDimY;
18
19   if ((iVoxelX < 1) || (iVoxelX > iDimX - 2) ||
20       (iVoxelY < 1) || (iVoxelY > iDimY - 2) ||
21       (iVoxelZ < 1) || (iVoxelZ > iDimZ - 2)) {
22     pcfOut[iIndex] = pcfIn[iIndex];
23     return;
24   }
25
26   for (iZ = -1; iZ <= 1; iZ++)
27     for (iY = -1; iY <= 1; iY++)
28       for (iX = -1; iX <= 1; iX++)
29         cfSum += pcfIn[(iVoxelX + iX) +
30                        (iVoxelY + iY) * iDimX) +
31                        (iVoxelZ + iZ) * iDimX * iDimY];
32
33   pcfOut[iIndex] = cfSum / 27.0;
34 }
```

Zunächst bestimmt der laufende Kernel über die built-in Variablen des zweidimensional angelegten Grids, zu welchen »virtuellen« dreidimensionalen Block er gehört. Die z Komponente war direkt in den y Index des Grids codiert (Codezeile 7), die x und y Komponenten werden aus der x Indizierung des Grids extrahiert. Die built-in Variablen blockDim.x, blockDim.y und blockDim.z liefern bei unserer Ausführungskonfiguration konstant 8 für alle Dimensionen und so ist es ein Leichtes über die Threadindizierug in den Codezeilen 9 bis 11 die genaue Position des Volumenelementes zu bestimmen, für dessen Berechnung der aktuelle Kernel zuständig ist.

Da die Berechnung des Boxfilters die unmittelbare Nachbarschaft eines Voxels einbezieht, werden die Codezeilen 19 bis 24 benötigt, um die nachbarschaftslosen Randvoxel vom Eingabespeicher direkt in den Ausgabespeicher zu kopieren und den Kernel zu terminieren. In den Zeilen 26 bis 31 werden die 27 benachbarten Voxel aus dem Eingabebereich des global memorys geladen und akkumuliert. In Zeile 33 wird schließlich das Ergebnis zurück in den global memory geschrieben.

Dieses einfache GPU Programm erzielt auf einem Intel Core 2 Duo mit 3,16 Ghz Taktung und einer GeForce 9800GT mit 112 SPs bei einem Volumendatensatz mit 256^3 Elementen bereits eine Verfünffachung der Geschwindigkeit (1.37 sec CPU vs. 0.28 sec GPU), obwohl die arithmetische Intensität im Mittel bei $28/28 = 1$ liegt (27 Schreibzugriffe und 1 Lesezugriff in den global memory bei 27 Additionen und 1 Division).

Durch Nutzung des cachingfähigen texture memorys erzielt ein ähnliches Programmgerüst bereits massive Geschwindigkeitsteigerungen und erreicht eine Ausführungszeit von 0.13 sec, was einen 10-fachen Performancegewinn bedeutet. Bei einem derart trivialen Problem fällt der Datentransfer vom Host zum Device und vice versa von rund zweimal 25 Millisekunden über Gebühr ins Gewicht. Abzüglich dieses Datentransfers benötigt die GPU 0.08 sec für die Berechnungen, erzielt also eine 17-fache Geschwindigkeitssteigerung. Da je nach Problemstellung die Daten auf der GPU weiterverarbeitet werden können oder auch nicht, sind im Ergebnisteil dieser Arbeit immer die Brutto-Geschwindigkeitssteigerungen (ohne Host-Device Memory Transfer) aber auch die vollständigen Netto-Geschwindigkeitssteigerungen angegeben.

Wir haben nun einen kurzen Einstieg in die GPU Programmierung mit C für CUDA[4] erhalten. Die nächsten zwei Kapitel erörtern die mathematischen Grundlagen der Diffusionsfilter und deren Anwendung in der Bildverarbeitung.

[4] Weitere Informationen über C für CUDA finden sich in den von Nvidia veröffentlichten Handbüchern und Sprachreferenzen in [Nvi10], [Nvi09a] und [Nvi09b]

3 Mathematische Grundlagen

Im Folgenden sollen die mathematischen Grundlagen dieser Arbeit behandelt werden. Da Diffusionsfilter bereits seit vielen Jahren Verwendung in der Bildverarbeitung finden, existiert eine breite Palette unterschiedlicher Literatur, welche die Motivation, grundlegende Formulierungen und auch die mathematischen Eigenschaften der Diffusionsfilter diskutiert. Der interessierte Leser findet im Internet ebenfalls unzählige Skripte und Vortragsfolien, welche die grundlegenden Ideen diskutieren, für ein tiefergehendes Verständnis jedoch einfach zu ungenau sind.
Zur Vorbereitung dieses Abschnitts war daher erstaunlicherweise sehr viel Recherche zu Diffusionvorgängen im Hinblick auf die Bildverarbeitung notwendig und so wurde es zum Ziel dieses Kapitels eine fundierte, in sich geschlossene und nachvollziehbare Erklärung der physikalischen Motiviation des gesamten Themenkomplexes zu liefern. Im Anschluss an dieses Kapitel werden die besprochenen Grundlagen schließlich dazu verwendet, die eigentlichen Diffusionsfilter in ihren verschiedenen Ausprägungen für die Bildverarbeitung zu formulieren.

Da der gesamte Filterungsprozess ursprünglich physikalisch motiviert ist, wird im kommenden Abschnitt zunächst ein Überblick über die »molekulare Diffusion« gegeben. Darauf aufbauend werden dann im Laufe der Arbeit zum einen die Diffusionsgleichungen und zum anderen die resultierenden Differentialgleichungssysteme hergeleitet. Die Bedeutung des Diffusionstensors wird kurz anhand der historischen Entwicklung erläutert und schließlich werden verschiedene Aspekte der Diskretisierung diskutiert, um das mathematische Problem im Rahmen dieser Arbeit effizient auf die Hardware der Grafikkarte abzubilden.

Vom physikalischen Prozess über die mathematische Formulierung bis hin zur anwendbaren Diskretisierung soll hier ein Beitrag geleistet werden, die Theorie der Diffusionsfilter in der Bildverarbeitung adäquat zu vermitteln.

3.1 Diffusion

In der Regel haben die meisten Menschen ein intuitives Verständnis von Diffusion, ohne dass der zugrunde liegende physikalische Prozess als solcher verstanden oder der Vorgang gar mathematisch formuliert wird. Mit Diffusion ist im allgemeinen die *molekulare Diffusion* gemeint und in der Wissenschaft zunächst aus der Physik und Chemie oder auch von informationstheoretischen Überlegungen zur Entropieerhöhung her bekannt. Laut [Cra80, S. 1] ist Diffusion »...the process by which matter is transported from one part of a system

to another as a result of molecular motions«. Wesentlich für das Verständnis des Diffusionsprozesses ist, dass keine externen Kräfte auf die Teilchen wirken, sondern aufgrund statistischer Prozesse ein netto Massetransport verzeichnet werden kann. Ein Gedankenexperiment erklärt den Prozess recht anschaulich (s. Abb. 6):

Zum Startzeitpunkt t_0 befinden sich auf der linken Seite einer Grenzfläche G, die das betrachtete Gesamtvolumen halbiert, 1000 und auf der rechten Seite 100 Teilchen eines Stoffes. Die Teilchen bewegen sich aufgrund der brown'schen Molekularbewegung ungerichtet und zufällig in alle Raumrichtungen. Dabei kommt es nun vor, dass sich Teilchen durch die Grenzfläche G bewegen und damit auf die andere Seite des geteilten Raumes gelangen. Wir betrachten nun den ersten Zeitpunkt, an dem ein Teilchen zufällig die Grenzfläche in den anderen Teilraum passiert. Die Wahrscheinlichkeit, dass es sich dabei um ein Teilchen handelt, welches von links nach rechts wandert, ist mit $1000/1100 = \frac{10}{11}$ exakt 10 mal größer, als der umgekehrte Fall mit $100/1100 = \frac{1}{11}$.
Über einen längeren Zeitraum ist also die Wahrscheinlichkeit, dass Teilchen von links nach rechts wandern höher, als die Wahrscheinlichkeit, dass Teilchen von rechts nach links wandern. Damit ist ein netto Fluss an Teilchen von links nach rechts verbunden, der schließlich dazu führen wird, dass der Konzentrationsunterschied ausgeglichen wird.

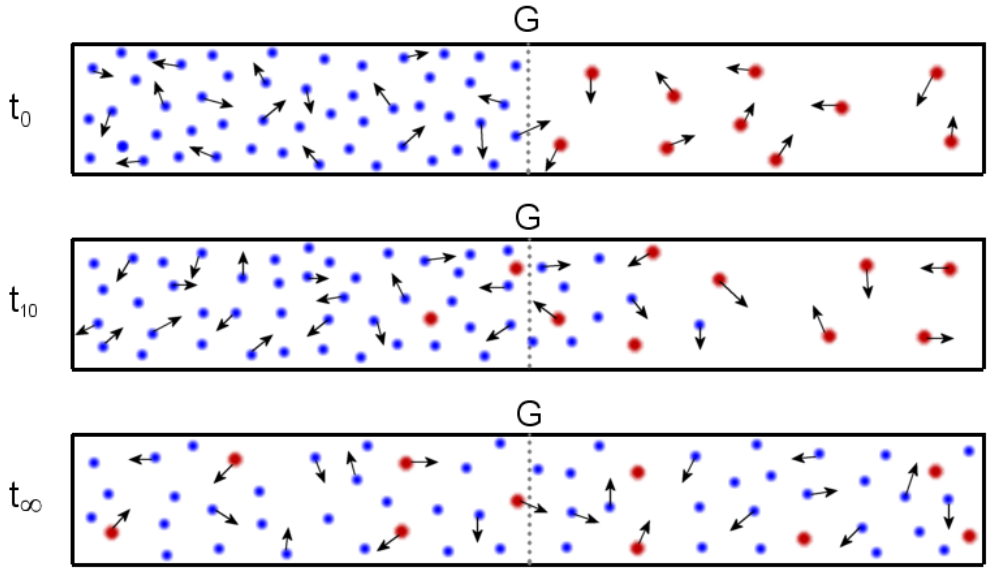


Abbildung 6: Diffusion durch eine Grenzfläche G

Sind die Konzentrationen der Teilräume ausgeglichen, befinden sich also auf jeder Seite der Grenzfläche rund 550 Teilchen des Stoffes, ist kein netto Teilchenfluss mehr zu erwarten: Zwar bewegen sich nach wie vor Teilchen durch die Grenzfläche G, jedoch ist die Wahrscheinlichkeit in beiden Richtungen gleich groß und damit kein netto Teilchenfluss

mehr feststellbar.

Dieses Gedankenexperiment zeigt bereits die wichtigsten Konzepte des Diffusionsprozesses auf:

- der Prozess ist zeitlich kontinuierlich fortschreitend
- die treibende Kraft ist systemimmanent, nämlich der Konzentrationsgradient
- der Konzentrationsgradient induziert einen dem Gradienten entgegengesetzt gerichteten, ausgleichenden Fluss
- nach hinreichend langer Zeit ($t \to \infty$) gerät das System in einen stabilen Ausgleichszustand
- aus genau diesem Grund ist die Lösung des inversen Problems offenbar nicht eindeutig

Neben der Masse transportierenden Diffusion, wie zum Beispiel bei einem Tropfen Tinte in Wasser, ist auch der Temperaturausgleich in einem geschlossenen System ein Diffusionsprozess. Hier wird keine Masse, sondern thermische Energie transportiert. Verallgemeinert ist Diffusion also ein zeitlich fortschreitender Prozess, der Konzentrationsunterschiede ausgleicht und schließlich zu einer homogenen Verteilung bzw. einem stabilen Ausgleichszustand führt.

3.2 Das 1. Fick'sche Gesetz

Adolf Eugen Fick hat den Zusammenhang zwischen Konzentrationsgradienten und induziertem Fluss im nach ihm benannten ersten Fick'schen Gesetz formuliert (vgl. [Cus09, S. 16ff] und [Meh05, S. 4]). Im eindimensionalen Fall lautet es:

$$j(x) = -D\frac{\partial \Phi(x)}{\partial x} \tag{3.1}$$

In dieser Formulierung ist die materialabhängige Proportionalitätskonstante D – der sogenannte Diffusionskoeffizient – ein Skalar, der den induzierten Diffusionsfluss j ins Verhältnis zum Konzentrationsgradienten setzt. Da der Fluss dem Gradienten entgegengerichtet ist, wird der Diffusionskoeffizient mit einem negativen Vorzeichen versehen notiert.
Dabei ist j der Diffusionsfluss und gibt an, welche Menge eines Stoffes durch einen infinitesimal kleinen Bereich in einem ebenfalls infinitesimal kleinen Zeitintervall fließt. Der

Fluss j wird physikalisch gewöhnlich in $\left[\frac{mol}{m^2 \cdot s}\right]$ angegeben.

Die Verallgemeinerung auf drei Raumdimensionen lässt sich mit Hilfe des Gradienten-Operators ∇ definiert als $\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots \frac{\partial f}{\partial x_n})^T$ wie folgt formulieren:

$$\vec{j}_D(x,y,z) \quad = \quad -D\nabla\Phi(x,y,z) = -D \begin{pmatrix} \frac{\partial\Phi(x,y,z)}{\partial x} \\ \frac{\partial\Phi(x,y,z)}{\partial y} \\ \frac{\partial\Phi(x,y,z)}{\partial z} \end{pmatrix} \tag{3.2}$$

In mehreren Dimensionen ist der Diffusionsfluss also eine vektorielle Größe und vom Diffusionskoeffizienten abhängig. Da es sich um ein Vektorfeld handelt, das jedem Ort $\vec{x} \in \mathbb{R}^n$ einen Vektor zuordnet, schreiben wir künftig nur noch $\vec{j}_D$. Der Diffusionskoeffizient D ist in dieser Formulierung zwar noch ein Skalar, er kann damit aber bereits als *Tensor 0-ter Stufe* [5] aufgefasst werden. Wir sprechen in Zukunft vom Diffusionstensor D.

Mit diesen Konventionen vereinfacht sich die Schreibweise zu:

$$\vec{j}_D = -D\nabla\Phi \tag{3.3}$$

Formel (3.3) ist damit die auf mehrere Raumdimensionen verallgemeinerte Formulierung des ersten Fick'schen Gesetzes. Bereits jetzt sollten wir uns im Klaren darüber sein, dass die Funktion Φ, offensichtlich ein Skalarfeld, das jedem Punkt des Raumes eine Konzentration zuordnet, zeitlich nicht konstant ist. In der Formulierung (3.2) wird deutlich, wie aufgrund des örtlichen Konzentrationsgradienten ein Diffusionsfluss induziert wird - der aber zu einer Änderung der Bildfunktion Φ führen wird. Es wird deutlich, dass sowohl der Diffusionsfluss $\vec{j}_D$, als auch die Bildfunktion Φ sowohl Funktionen des Ortes, als auch der Zeit sein müssen: Zu jedem Zeitpunkt des Diffusionsprozesses liegt an einem Ort des Raumes ein spezieller Konzentrationsgradient vor, der einen Diffusionsfluss induziert und damit das Konzentrationsfeld bzw. die Bildfunktion Φ ändert. Diese Überlegungen lassen bereits erahnen, dass wir auf eine partielle Differentialgleichung hinarbeiten.

[5] Anmerkung: Für das hier benötigte Verständnis soll folgende Erklärung ausreichen: Tensoren sind indizierbare Größen. Ein Skalar benötigt keinen Index, kann also als Tensor 0-ter Stufe bezeichnet werden. Werden die Elemente eines Tensors mit einem Index fortlaufend abgezählt, erhält man einen Tensor 1-ter Stufe. Ein Tensor 1-ter Stufe kann durch einen Vektor repräsentiert werden. Ein Tensor 2-ter Stufe wird entsprechend über zwei Indizes definiert und könnte durch eine Matrix dargestellt werden. In der Regel werden Tensoren als Koeffizienten eingesetzt und Tensoren höherer Stufen können einfach als Fortführung des Konzeptes eines skalaren Koeffizienten verstanden werden.

3.3 Das 2. Fick'sche Gesetz

Das zweite Fick'sche Gesetz, die sogenannte Diffusionsgleichung, kann aus dem ersten Fick'schen Gesetz und der Kontinuitätsgleichung hergeleitet werden. Die Kontinuitätsgleichung ist eine Erhaltungsgleichung und sagt aus, dass die Änderung der Menge eines Stoffes in einem abgegrenzten räumlichen Gebiet (solange sich keine Quelle oder Senke im betrachteten Gebiet befindet) ausschließlich auf den zeitlichen Zu- und Abfluss durch die Begrenzungflächen des Gebietes zurückzuführen ist. Die Herleitung in differentieller und integraler Form kann in Grundlagenbüchern (z.B. [HES07, S. 31ff]) nachgelesen werden. Die für unsere Betrachtungen relevante Form (ohne zusätzliche Quelle und Senke) lautet:

$$\begin{aligned} \frac{\partial \Phi}{\partial t} &= -\nabla \vec{j} \quad \text{bzw.} \\ \frac{\partial \Phi}{\partial t} + \nabla \vec{j} &= 0 \end{aligned} \tag{3.4}$$

Mit anderen Worten: Die zeitliche Änderung ist exakt die Summe der partiellen Zu- und Abflüsse in das bzw. aus dem betrachteten Gebiet. Hier ist zu beachten, dass ∇ die *Divergenz*[6], einen Differentialoperator für Vektorfelder darstellt.[7]

---

[6] Die Divergenz ist ein Funktional eines Vektorfeldes. Sie bildet ein Vektorfeld auf ein skalares Feld ab. $div = \nabla = (\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n})^T$ ist dabei der formale Nabla Operator. Somit kann die Divergenz als Skalarprodukt geschrieben werden:

$$\nabla \vec{F} = \nabla \begin{pmatrix} F_1 \\ \vdots \\ F_n \end{pmatrix} = \sum_{t=1}^{n} \frac{\partial F_t}{\partial x_t}$$

Im $\mathbb{R}^3$ gilt also:

$$\text{Sei } \vec{F}(x,y,z) = \begin{pmatrix} F_x(x,y,z) \\ F_y(x,y,z) \\ F_z(x,y,z) \end{pmatrix} \quad \text{so gilt: } \nabla \vec{F} = \begin{pmatrix} \frac{\partial}{\partial x} \frac{\partial}{\partial y} \frac{\partial}{\partial z} \end{pmatrix} \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}$$

Anschaulich: Im $\mathbb{R}^3$ besteht die Divergenz einfach aus der Summe der drei partiellen Richtungsableitungen.

[7] *Achtung:* Es bleibt dem Leser überlassen kontextabhängig zu entscheiden, ob der Differentialoperator ∇ im konkreten Fall den Gradienten oder die Divergenz beschreibt. Angewand auf ein Skalarfeld beschreibt ∇ den Gradienten und liefert ein Vektorfeld (die Vektoren zeigen entlang des steilsten Anstiegs), angewandt auf ein Vektorfeld beschreibt ∇ die Divergenz und liefert ein Skalarfeld (der Wert des Skalars ist die Summe der partialen Zu- und Abflüsse).

Die Kontinuitätsgleichung (3.4) beschreibt also die zeitliche Änderung des Bildes Φ mit Hilfe des Flusses $\vec{j}$ und das erste Fick'sche Gesetz gibt den aufgrund der Konzentrationsunterschiede induzierten Diffusionsfluss $\vec{j}$ an.
Durch Einsetzen von (3.1) in (3.4) erhalten wir nun die eindimensionale Diffusionsgleichung, eine partielle Differentialgleichung zweiter Ordnung:

$$\begin{aligned} \frac{\partial \Phi}{\partial t} &= -\nabla\left(-D\frac{\partial \Phi}{\partial x}\right) \\ \frac{\partial \Phi}{\partial t} &= D\nabla\left(\frac{\partial \Phi}{\partial x}\right) \\ \frac{\partial \Phi}{\partial t} &= D\frac{\partial^2 \Phi}{\partial x^2} \end{aligned} \tag{3.5}$$

Umfassende Erläuterungen zur eindimensionalen Diffusionsgleichung finden sich in [Can84]. Sie sagt im Endeffekt nichts anderes aus, als dass die zeitliche Änderung proportional zur zweiten Ableitung der Bildfunktion ist. Es folgt die Verallgemeinerung der Diffusionsgleichung auf mehrere Dimensionen.

3.4 Die n-dimensionale Diffusionsgleichung

Um die Diffusionsgleichung auf mehrere Dimensionen zu erweitern, setzen wir (3.3) in (3.4) ein und erhalten ganz allgemein:

$$\frac{\partial \Phi}{\partial t} = \nabla\left(D\ \nabla\Phi\right) \quad \text{bzw.} \tag{3.6}$$

$$\frac{\partial \Phi}{\partial t} = \operatorname{div}\left(D\ \operatorname{grad}\left(\Phi\right)\right) \tag{3.7}$$

Dabei ist (3.7) ist die allgemeingültige Formulierung. Unter Nutzung des Laplace Operators $\Delta = \nabla^2$ und der *vorläufigen Annahme*, dass der Diffusionstensor ein Skalar ist, kann man (3.6) weiter vereinfachen. In diesem Fall gilt:

$$\frac{\partial \Phi}{\partial t} = \nabla\left(D\ \nabla\Phi\right) = D\nabla\nabla\Phi = D\nabla^2\Phi = D\Delta\Phi \tag{3.8}$$

Für drei Raumdimensionen erhalten wir:

$$\frac{\partial \Phi}{\partial t} - D\left(\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial^2 \Phi}{\partial z^2}\right) = 0 \tag{3.9}$$

Die Formulierung (3.9) ist nun die in drei Raumdimensionen ausgeprägte Variante der Wärmeleitungsgleichung (»heat equation«) $\frac{\partial \Phi}{\partial t} = D\Delta\Phi$ aus (3.8).

4 Diffusionsfilter in der Bildverarbeitung

Seit einigen Jahren wird das physikalisch motivierte Konzept der Diffusion auch in der Bildverarbeitung eingesetzt. Die Anwendung reicht dabei je nach Zweck von der Glättung und Entrauschung des Bildes über Kantenverstärkung, von kohärenzverstärkender Diffusion bis hin zur direkten Vorverarbeitung für Segmentierungsalgorithmen durch die inhärente Skalenraumrepräsentation über die Zeit.

Insbesondere die Arbeiten von Weickert, zusammengefasst in seinem Buch »Anisotropic Diffusion in Image Processing« [Wei98], stellen die mathematischen Grundlagen der Diffusionsfilter kompakt zusammen. Weickert arbeitet sowohl für die kontinuierliche, die semidiskrete als auch die diskrete Formulierung des Diffusionsfilters die grundlegenden Eigenschaften heraus und beschreibt und bewertet verschiedene Ansätze im Rahmen der allgemeinen Formulierung des Diffusionsfilters.

In der vorliegenden Arbeit stützen wir uns auf die im Folgenden kompakt zusammengefassten Begriffsdefinitionen nach Weickert und werden dann die einzelnen Diffusionsfilter entsprechend charakterisieren. Dabei wird in diesem Kapitel zunächst ausschließlich eine kontinuierliche Beschreibung vorgenommen. Aspekte, die mit der räumlichen und zeitlichen Diskretisierung einhergehen, werden im Kapitel »Implementierung« besprochen.

4.1 Klassifikation der Diffusionsfilter

Diffusionsfilter können hinsichtlich ihres Verhaltens in verschiedene Typen eingeteilt oder benannt werden. Die treibende Kraft hinter dem Ausgleichsprozess ist immer der lokale Temperatur- oder Konzentrationsunterschied, der einen Fluss $\vec{j}_D$ bewirkt. Die folgende Schreibweise von (3.7) kann man zur Klassifikation des Diffusionsfilters nutzen:

$$\frac{\partial \Phi}{\partial t} = \operatorname{div}\left(\vec{j}_D\right)$$

Anhand des Flusses $\vec{j}_d = D\nabla\Phi$ kann man nun in lineare und nichtlineare, isotrope und anisotrope sowie homogene und inhomogene Diffusion unterteilen. Die entscheidende Rolle spielt hierbei der Diffusionstensor D, der nicht unbedingt eine konstante Größe sein muss. Allgemein ist der Diffusionstensor eine Funktion des Ortes und der Zeit und für die in der Bildverarbeitung insbesondere interessanten anisotropen Diffusionsvorgänge handelt es sich nicht einmal um eine skalare Größe.

Lineare und nichtlineare Diffusion: Man spricht von *linearer Diffusion*, wenn der Fluss linear vom Gradienten abhängt (vgl. [MRC97, S. 474]). Weickert weist in diesem

Zusammenhang in [Wei97] auf folgenden Sachverhalt hin: In der Formulierung $\vec{j}_d = g(\Phi_0)\nabla\Phi$, könnte der Diffusionstensor zum Beispiel eine skalare Größe sein, die von der Struktur des Urbildes Φ_0 abhängt. Dennoch bleibt die Diffusionsgleichung linear, es handelt sich um inhomogene (ortsabhängige), lineare Diffusion. Im Gegensatz dazu steht die Formulierung $\vec{j}_d = g(\Phi)\nabla\Phi$. Der Diffusionstensor ist nun nicht mehr eine Funktion des Urbildes, sondern des aktuellen, geglätteten Skalarfeldes. Es findet also eine Rückkopplung statt, dieser Prozess ist eine nichtlineare inhomogene Diffusion.

isotrope und anisotrope Diffusion: Man spricht von *isotroper Diffusion*, wenn der durch den Gradienten induzierte Fluss parallel zum Gradienten verläuft, also in die genau entgegengesetzte Richtung des Gradienten fließt. Für skalare Diffusionstensoren ist dies immer der Fall, unabhängig davon, ob es sich um homogene oder inhomogene Diffusion handelt. Diffusionsvorgänge, bei denen der Fluss zum Beispiel entlang einer Kante ausgerichtet wird, sind anisotrop.

homogene und inhomogene Diffusion: Man spricht von *homogener Diffusion*, wenn der Diffusionstensor ortsunabhängig ist. Weickert definiert in [Wei98, S. 2] *homogene* und *inhomogene* Diffusion wie folgt: »If the diffusion tensor is constant over the whole image domain, one speaks of *homogeneous* diffusion, and a space-dependent filtering is called *inhomogeneous*.« Unsere allgemeine Formulierung (3.6) lässt beispielsweise offen, ob mit D ein konstanter oder ortsabhängiger Diffusionstensor, also $D(\vec{x})$ gemeint ist.

Diese Definitionen halten sich an die Begriffe von Weickert in [Wei98] und können in anderer Literatur ggf. erheblich abweichen. [8]

4.2 Diffusion und Skalenraum Repräsentation

Seit den Anfängen der Skalenraumtheorie für Signale (»scale-space representation«, vgl. [Wit83]), hat sich dieses Konzept auch in der Bildverarbeitung immer wieder als sinnvoll erwiesen. Insbesondere im Bereich des »künstlichen Sehens«, ist die Bedeutung eines auf verschiedenen Skalen vorliegenden Signales wieder von zentraler Bedeutung, um Berei-

[8] *Anmerkung:* Seit Perona und Malik in [PM90] ihre Idee eines lokal adaptiven Diffusionskoeffizienten vorgestellt haben, wird in der Literatur oftmals eine andere Definition zugrunde gelegt: Meist ist mit »isotroper Filterung« ein homogener, räumlich invarianter, Diffusionstensor gemeint, während sich »anisotropische Filterung« auf einen lokal adaptiven, also nach Weickerts Definition inhomogenen Diffusionstensor bezieht. Literatur, die von anisotropem Verhalten spricht, verwendet nicht notwendigerweise einen nicht skalaren Diffusionstensor, siehe z.B. [ALM92]

che des Bildes sinnvoll zu segmentieren und klassifizieren zu können (siehe z.B. [Lin98]). Nach Lindeberg ist »scale-space representation ... a special type of multi-scale representation that comprises a *continuous* scale parameter and preserves the same spatial sampling at all«. [Lin94]

Der Schwerpunkt liegt hier auf der Existenz eines kontinuierlichen Skalenraum-Parameters. Im Regelfall werden in der Bildverarbeitung ausgehend von einem hochaufgelösten Originalbild durch sukzessive Anwendung einer kontinuierlich parametrisierbaren Operation immer einfachere Repräsentationen des ursprünglichen Bildes erzeugt. Wird beispielsweise ein zweidimensionales Bild mit einem Gaußfilter geglättet, lässt sich der Skalenraum-Parameter leicht ablesen: Es findet eine Faltung des Bildes mit einer zweidimensionalen Normalverteilungsfunktion statt. Die Varianz σ bestimmt dabei wie stark die Glättung ist. Sei $\Phi_0 : \mathbb{R}^2 \mapsto \mathbb{R}$ das ungeglättete Bild, so erhält man eine Skalenraumrepräsentation $\Phi_\sigma : \mathbb{R}^2 \mapsto \mathbb{R}$ wie folgt:

$$\Phi_\sigma(x, y) = (\Phi_0 * g_\sigma)(x, y) \tag{4.1}$$

mit

$$g_\sigma(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{4.2}$$

$\Phi_\sigma(x, y)$ ist also eine Multi-Skalen-Repräsentation des Bildes, der kontinuierliche Parameter σ erlaubt es immer weiter vereinfachte Versionen des Bildes zu betrachten.

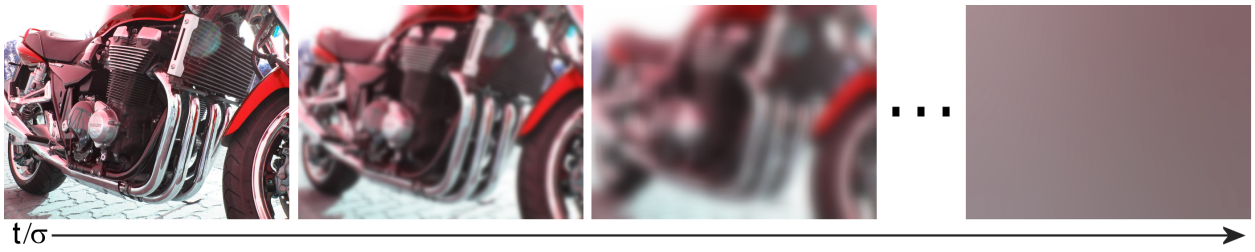


Abbildung 7: Skalenraumrepräsentation eines Bildes

Im Falle der Diffusion ist die fortschreitende Zeit ein natürlicher Parameter der vom hochaufgelösten Urbild zu immer weiter geglätteten und damit vereinfachten Versionen des Bildes führt. »Time represents the scale parameter which leads from noisy fine scale to smoothed and enhanced coarse scale representation of the data.« [BWX, S. 2]

Für $t \to \infty$ verhält sich der Diffusionsfilter analog zu $\sigma \to \infty$ bei der Glättung mit einem Gaußkern und führt dazu, dass das Bild schließlich in eine homogene Fläche überführt wird.

4.3 Lineare homogene Diffusion

Wir wollen nun die einfachste Formulierung, den linearen homogenen Diffusionsfilter, im $\mathbb{R}^3$ beschreiben.

Sei $\Phi_0(\vec{x}) : \Omega \mapsto T$ eine Funktion auf einem Gebiet $\Omega \subset \mathbb{R}^3$, die jedem Punkt $\vec{x} \in \Omega$ einen Wert aus $T \subset \mathbb{R}$ zuordnet. In der medizinischen Bildverarbeitung ist T in der Regel ein Grauwert (z.B. $T = [0 \ldots 255]$) oder eine Dichteangabe (z.B. CT Bild, Hounsfield Skala $T = [-1024 \ldots 3071]$). Den zugeordneten Wert aus T kann man sich im Folgenden immer als Temperatur am Ort $\vec{x}$ vorstellen. Das Gebiet Ω ist in der Regel der betrachtete Volumendatensatz, also meist ein Quader im $\mathbb{R}^3$.

Die Wärmeleitungsgleichung (3.8) ist eine parabolische partielle Differentialgleichung zweiter Ordnung. Um eine Lösung zu finden ist daher die Angabe von Anfangs- und Randwerten notwendig. Wir können den linearen Diffusionsfilter nun ausgehend von (3.8) wie folgt definieren:

$$\frac{\partial}{\partial t}\Phi(\vec{x}, t) = D\Delta\Phi(\vec{x}, t) \quad \text{für} \quad \vec{x} \in \Omega, t > 0 \tag{4.3}$$

$$\Phi(\vec{x}, 0) = \Phi_0(\vec{x}) \quad \text{für} \quad \vec{x} \in \Omega \tag{4.4}$$

$$\frac{\partial}{\partial \vec{n}}\Phi(\vec{x}, t) = 0 \quad \text{für} \quad \vec{x} \in \partial\Omega \tag{4.5}$$

Dabei ist $\Phi(\vec{x}, t)$ die gefilterte Bildfunktion zum Zeitpunkt t. Für $t = 0$ liefert (4.4) die Anfangsbedingung des Gleichungssystems: Φ_0 ist die ungeglättete Version des Bildes und damit der Startwert des Diffusionsprozesses.

Die Formulierung (4.5) gibt die räumlichen Randwerte vor: Die partielle Ableitung der Bildfunktion Φ in Richtung $\vec{n}$ soll für alle Punkte $\vec{x}$ auf dem Rand $\partial\Omega$ des Gebietes Ω gleich null sein, wobei n die jeweilige Normalenrichtung des Randes ist. Diese Randwertbedingung ist eine sogenannte Neumannbedingung, da sie die Randwerte über die Ableitungen definiert. In der Bildverarbeitung wird dies in der Regel erreicht, indem über

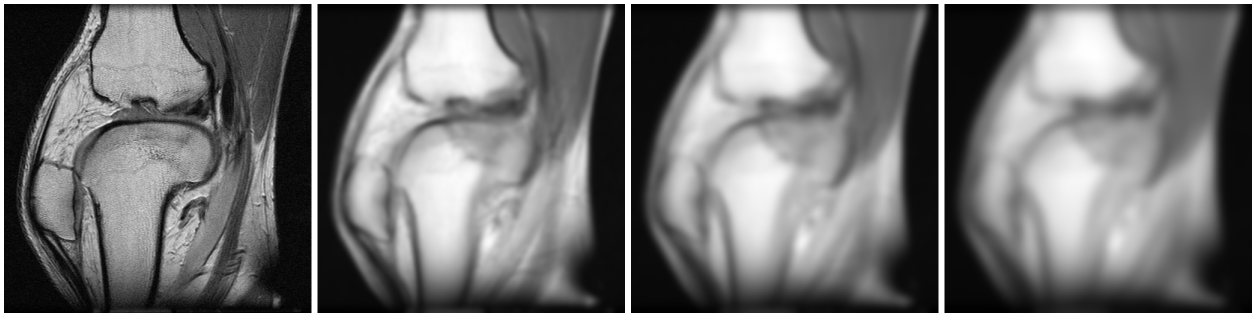


Abbildung 8: Lineare homogene Diffusion.
Datensatz: Knee, Größe: $512 \times 512 \times 88$, Slice 49
Diffusion mit $\Delta t = 0.05$
Von links: t = 0, t = 1.0, t = 2.5, t = 5.0
(vgl. auch Abb. 11, S. 32)

die Ränder das Volumendatensatzes hinaus einfach der letzte gültige Voxelwert angenommen wird oder Zugriffe auf Werte auserhalb des Volumens auf die Ränder des Datensatzes zurückgesetzt werden (»clamp values«).
Eine andere Variante wären sogenannte Dirichlet-Randbedingungen, bei denen die Randwerte fest vorgegeben sind - in der Bildverarbeitung wird das oft einfach durch Setzen der Werte ausserhalb des Volumens auf Null realisiert. Da der eigentliche Diffusionsfluss vom Gradienten induziert wird, ist die hier gewählte Neumannbedingung (4.5) einer Dirichlet-Randbedingung vorzuziehen, da auf diese Weise kein Fluss durch den Rand des Gebietes Ω stattfindet. Der mittlere Grauwert im Gebiet Ω bleibt daher erhalten.
Der Diffusionstensor ist eine echte skalare Konstante. Das führt uns im Folgenden auch zur Erklärung, warum dieser Filter als »linearer homogener Diffusionsfilter« bezeichnet wird.

4.4 Eigenschaften des linearen homogenen Diffusionsfilters

Die lineare Diffusionsgleichung besitzt laut Literatur (vgl. [BWX], [Wei97]) die folgende Lösung

$$\Phi(\vec{x}, t) = \begin{cases} \Phi_0(\vec{x}) & (t = 0) \\ (G_{\sqrt{2t}} * f)(\vec{x}) & (t > 0) \end{cases}$$

Wobei $G_{\sqrt{2t}}$ ein 3 dimensionaler Gaußkern mit der Varianz $\sigma = \sqrt{2t}$ ist. Weickert gibt in [Wei98, S. 3f] einen Überblick, wie die Glättung durch Faltung mit einem Gaußkern mithilfe der Diffusionsgleichung verstanden werden kann. Anhand obiger Lösung der linearen Diffusionsgleichung können wir ablesen, dass die Faltung des Bildes mit einem Gaußkern der Varianz σ genau der linearen Diffusion entspricht, die nach $t = \frac{1}{2}\sigma^2$ gestoppt werden muss und alle Bildstrukturen glättet, deren räumliche Ausbreitung kleiner

als σ ist.

Der lineare Diffusionsfilter entspricht also der Faltung mit einem Gaußkern und glättet über den gesamten Bildbereich Ω gleichmäßig stark. Er hat damit den Nachteil, dass er außer dem Rauschsignal auch die Kanten glättet (s. Abb. 10). Um das zu umgehen, müssen lokal adaptive Filter eingesetzt werden.

Abbildung 9: Skalenraumrepräsentation einer MRT Slice durch Diffusion.
Datensatz: MRI Head, Größe $256 \times 256 \times 256$, Slice 130
200 Diffusionsschritte mit $\Delta t = 0.05$
Vorne: Zeitpunkt t = 0, Herausgehoben: Zeitpunkt t = 5.0

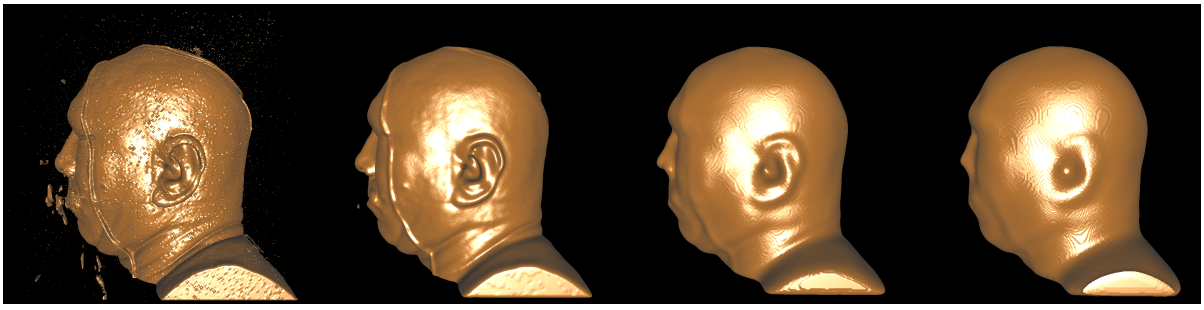


Abbildung 10: Lineare homogene Diffusion, $\Delta t = 0.05$. Isooberflächenwert: 50 von 255
Von links: Verrauschter Datensatz (ca. 5% betroffene Voxel, additives gaußsches Rauschen, Standardabweichung 10%), Lineare homogene Diffusion mit 5, 25, 50 Iterationen. Es ist ein deutlicher Verlust der Kanteninformationen erkennbar.

4.5 Lineare inhomogene Diffusion

Die grundlegende Idee hinter lokal adaptiven Filtern ist es, den eigentlichen Filterungsprozess abhängig von der lokalen Bildstruktur zu gestalten. Auf diese Weise ist es adaptiven Filtern möglich, wichtige Strukturen im Bild zu erhalten, indem an entsprechenden Stellen weniger geglättet wird.

Als Richtlinie gilt: Wichtige Strukturen des Bildes sind in der Regel die enthaltenen Kanten. Kantendetektoren basieren zumeist auf den Maxima der ersten Ableitung oder den Nulldurchgängen der zweiten Ableitung. Lindeberg schreibt in [Lin93, S. 18]: » A natural way to define edges from a continuous grey-level image $L : \mathbb{R}^2 \mapsto \mathbb{R}$ is as the set of points for which the gradient magnitude assumes a maximum in the gradient direction.« und Weickert stellt in [Wei97] den Nutzen heraus: »We can use $|\nabla f|$ [Anm.: f ist hier die Bildfunktion] as a fuzzy edge detector: locations with large $|\nabla f|$ have a higher likelihood to be an edge.«

Der Gradientenvektor ist also als Kantendetektor nutzbar, seine Länge $|\nabla\Phi|$ ist eine Maßzahl für die Kantenstärke. Der konstante Diffusionstensor wird nun durch eine monoton fallende Funktion g ersetzt, deren Parameter die Länge des Gradienten am Ort $\vec{x}$ ist. Im Folgenden verkürzen wir nun auch die Schreibweise, um uns auf die wesentlichen Elemente der Formulierung zu konzentrieren. Dabei ist $\Phi = \Phi(\vec{x}, t)$ immer eine Funktion des Ortes und der Zeit, $\Phi_0 = \Phi(\vec{x}, 0)$ das Urbild.

Wir ändern 4.3 entsprechend ab und erhalten so die Formulierung für den linearen inhomogenen Diffusionsfilter:

$$\partial_t \Phi = \operatorname{div}(\, g(|\nabla\Phi_0|)\, \nabla\Phi) \quad \text{auf} \quad \Omega \times [0,\infty] \tag{4.6}$$

$$\Phi(\vec{x}, 0) = \Phi_0(\vec{x}) \quad \text{auf} \quad \Omega \tag{4.7}$$

$$\partial_{\vec{n}} \Phi = 0 \quad \text{auf} \quad \partial\Omega \times [0,\infty] \tag{4.8}$$

Mit den Eigenschaften der Funktion g und dem resultierenden Diffusionsfluss $\vec{j}$ werden wir uns im Abschnitt »4.8 Diffusivität und Flussfunktion« (S. 33) beschäftigen.

4.6 Eigenschaften des linearen inhomogenen Diffusionsfilters

Der durch den Gradienten induzierte Fluss wird nun durch einen veränderlichen Diffusionstensor geregelt. Die monoton fallende Funktion $g : \mathbb{R}_0^+ \mapsto \mathbb{R}_0^+$ wird dabei so ge-

wählt, dass $g(0) = 1$ und $g(x \to \infty) = 0$ gilt. In Bildbereichen mit niedrigem Gradienten ähnelt der Diffusionsprozess der linearen homogenen Diffusion, während in der Nähe von Kanten der Diffusionsprozess annähernd unterdrückt wird, es findet fast kein Temperaturaustausch statt.

Die Funktionsweise entspricht einer Glättung des Bildes mit Gaußkernen veränderlicher Größe, wobei in der Nähe von Kanten nur kleine Gaußkerne verwendet werden. Folgendes ist zu beachten:

Kanten werden zwar weniger stark ausgewaschen, allerdings wird in der Nähe der Kanten auch das Rauschsignal nicht geglättet.

Der absolute Kontrast der Kanten nimmt im geglätteten Bild ebenfalls monoton ab. Nach hinreichend langer Zeit wird bei jedem Diffusionsfilter ein homogenes Grauwertebild erzeugt.

4.7 Nichtlineare inhomogene Diffusion

In Form einer PDE[9] wurde die nichtlineare inhomogene Diffusion erstmals von Pietro Perona und Jitendra Malik in [PM90] genutzt. Ihr nichtlinearer Diffusionsfilter, bekannt als »Perona-Malik model« oder »Perona-Malik method« (s. z.B. [Wei98, S. 15] oder [Ese01, S. 3]), ist wie folgt formuliert:

$$\partial_t \Phi = \operatorname{div}(\, g(|\nabla \Phi|)\, \nabla \Phi) \quad \text{auf} \quad \Omega \times [0, \infty] \tag{4.9}$$

$$\Phi(\vec{x}, 0) = \Phi_0(\vec{x}) \quad \text{auf} \quad \Omega \tag{4.10}$$

$$\partial_{\vec{n}} \Phi = 0 \quad \text{auf} \quad \partial\Omega \times [0, \infty] \tag{4.11}$$

Perona und Malik nannten diesen Diffusionsfilter anisotrop, da der Fluss der lokalen Bildstruktur angepasst ist. Nach Weickerts Definitionen handelt es sich dennoch um einen isotropen Filter, da ein Skalar als Diffusionskoeffizient genutzt wird. Die Tatsache, dass der Diffusionskoeffizient lokal evaluiert wird, macht dieses Modell in Weickerts Nomenklatur zu einem »inhomogenen« Diffusionsfilter.
Die einzige Änderung an dieser Formulierung zu 4.6 ist, dass die Funktion g auf die geglätteten Versionen des Urbildes zurückgreift. Diese Rückkopplung macht das Gleichungssystem nichtlinear.

[9] In [BC92, S. 122 ff.] wird neben den Grundlagen zu PDEs auch auf die Wärmeleitungsgleichung eingegangen. [GR06] gibt einen guten Einstieg in die Thematik und geht ab Seite 33. auch explizit auf Differenzverfahren ein.

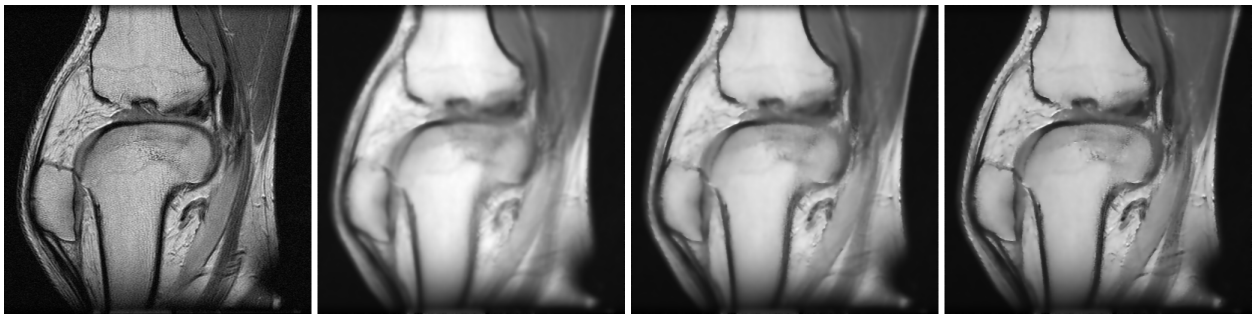


Abbildung 11: Nichtlineare inhomogene Diffusion.
Datensatz: Knee, Größe: $256 \times 512 \times 512 \times 88$, Slice 49
Links: t = 0.0 (Originalbild), danach dreimal zum Zeitpunkt t = 5.0
mit $\lambda = 0.05$, $\lambda = 0.03$, $\lambda = 0.02$ (100 Iterationen, $\Delta t = 0.05$)
Deutlich zu erkennen: Rauschen innerhalb homogener Flächen wird gut geglättet und Kanten bleiben dennoch lange erhalten. Das Rauschen in der Nähe von Kanten wird jedoch kaum noch geglättet (vgl. auch Abb. 8, S. 28 und Abb. 15, S. 35).

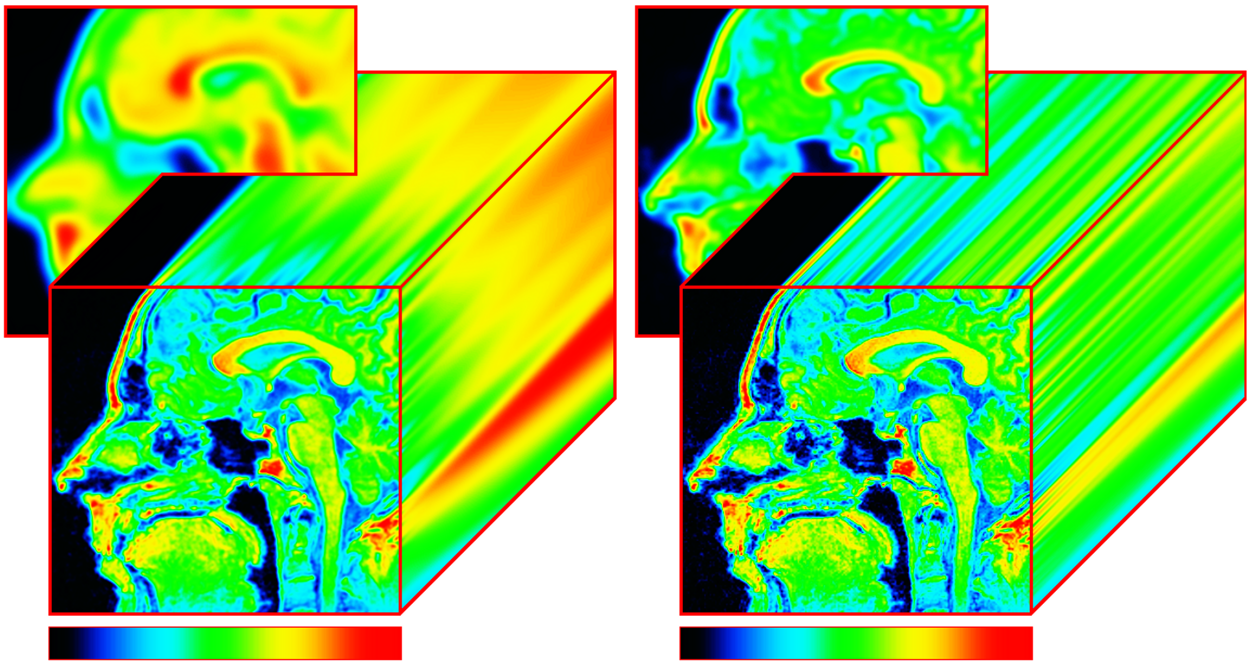


Abbildung 12: Vergleich der Skalenraumrepräsentation homogener (links) und inhomogener (rechts) Diffusion. Datensatz: MRI Head, Größe $256 \times 256 \times 256$, Slice 130
200 Diffusionsschritte mit $\Delta t = 0.05$
Vorne: Zeitpunkt t = 0, Herausgehoben: Zeitpunkt t = 5.0

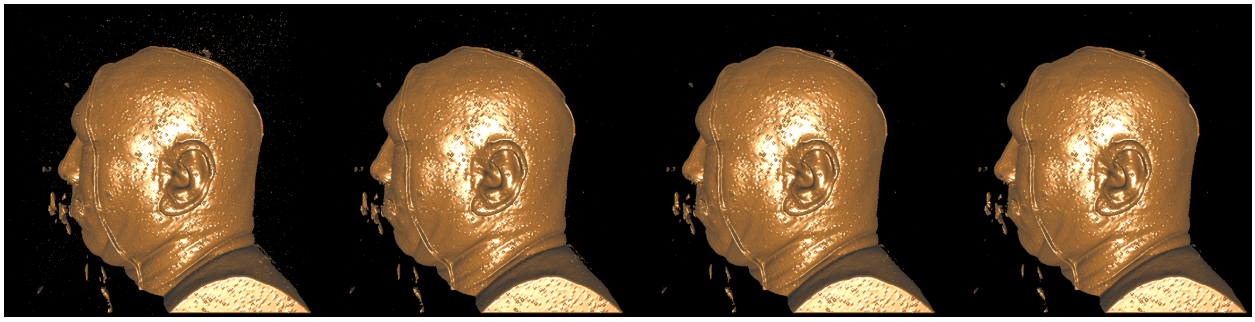


Abbildung 13: Nichtlineare inhomogene Diffusion, $\Delta t = 0.05$.
Isooberflächenwert: 50 von 255
Von links: Verrauschter Datensatz (ca. 5% betroffene Voxel, additives gaußsches Rauschen, Standardabweichung 10%), Nichtlineare inhomogene Diffusion mit 5, 25, 50 Iterationen. Die Kanteninformation bleibt lange erhalten, das Rauschsignal in der Nähe der Kanten wird jedoch fast nicht geglättet.

4.8 Diffusivität und Flussfunktion

Der Diffusionsfluss ist die Summe der partiellen Richtungsflüsse induziert durch den Gradienten und gesteuert durch die Diffusivität g. Perona und Malik schlagen in [PM90] die folgenden Funktionen für die Diffusivität vor:

$$g(\nabla\Phi) = e^{-(|\nabla\Phi|/\lambda)^2} \tag{4.12}$$

oder

$$g(\nabla\Phi) = \frac{1}{1+\left(\frac{|\nabla\Phi|}{\lambda}\right)^2} \tag{4.13}$$

Im eindimensionalen Fall können wir den Zusammenhang zwischen Diffusivität und Fluss besonders leicht untersuchen und Gleichung (4.9) unter Nutzung von Gleichung (4.13) wie folgt notieren:

$$\partial_t \Phi = \frac{\nabla\Phi}{1+\left(\frac{|\nabla\Phi|}{\lambda}\right)^2} \tag{4.14}$$

Bei niedrigem Gradienten $\nabla\Phi$ wird ein geringer Diffusionsfluss induziert, da der Zähler $|\nabla\Phi|$ relativ klein ist. Für sehr hohe Gradienten $\nabla\Phi \gg \lambda$ dominiert der gesamte Nenner den Term und der Wert des vom Gradienten induzierten Flusses wird abgeschwächt – Kanten blockieren den Diffusionsvorgang. Der größte Fluss findet für $\nabla\Phi \approx \lambda$ statt.
Die freie Variable λ dient als intuitiver, leicht zu bedienender Paramter der Diffusion: Features, deren Gradienten in der Nähe des Schwellwerts λ liegen, induzieren einen Fluss

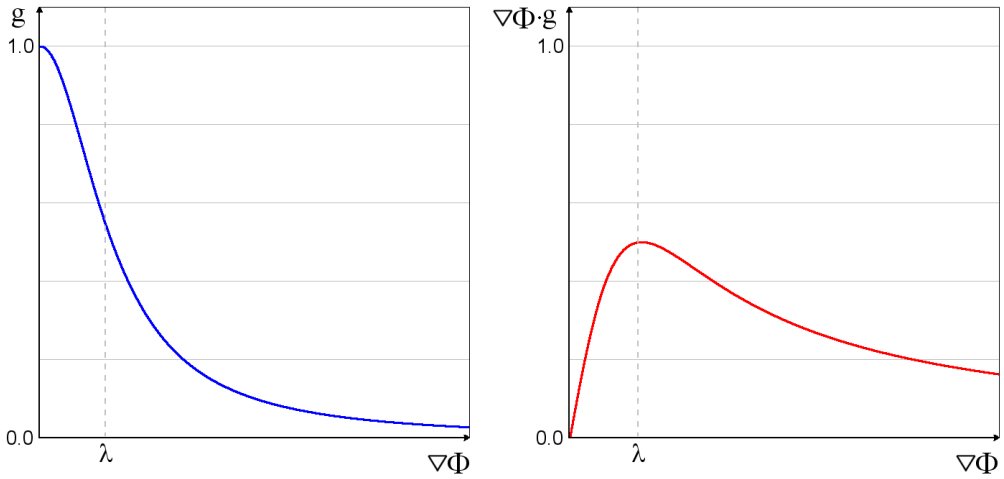


Abbildung 14: (a) Links: Diffusivität $g(\nabla\Phi) = \frac{1}{1+\left(\frac{|\nabla\Phi|}{\lambda}\right)^2}$ (b) Rechts: Fluss $\nabla\Phi \cdot g(\nabla\Phi)$

und werden schnell ausgeglichen. Idealerweise sollte λ also entsprechend der Eigenschaften des Rauschsignals gewählt werden.

Die Tatsache, dass Rauschen in der Nähe von Kanten nicht unterdrückt wird, führt uns nun zu der entscheidenden Verallgemeinerung der skalaren Diffusivität zu einem echten Tensor.

4.9 Nichtlineare anisotrope Diffusion

Während bei den vorangegangenen isotropen Modellen der induzierte Diffusionsfluss parallel zur Gradientenrichtung verläuft, wollen wir nun einen anisotropen Diffusionsfilter formulieren. Die Idee dahinter ist es, den gesamten induzierten Diffusionsfluss lokal jeweils in den Fluss entlang der Kante und den Fluss orthogonal zur Kante zu zerlegen. Der parallel und senkrecht zur Kante induzierte Fluss kann dann mithilfe beliebiger Funktionen separat gesteuert werden, insbesondere kann der Fluss orthogonal zur Kante unterdrückt werden.

Auf diese Weise sollen die im Urbild enthaltenen Kanten lange Zeit erhalten bleiben, während das Rauschen entlang der Kanten relativ schnell geglättet wird.

$$\partial_t \Phi = \operatorname{div}(\, D(S)\, \nabla\Phi) \quad \text{auf} \quad \Omega \times [0,\infty] \tag{4.15}$$

$$\Phi(\vec{x},0) = \Phi_0(\vec{x}) \quad \text{auf} \quad \Omega \tag{4.16}$$

$$\partial_{\vec{n}} \Phi = 0 \quad \text{auf} \quad \partial\Omega \times [0,\infty] \tag{4.17}$$

Der Diffusionstensor D ist dabei nicht länger vom Gradienten $\nabla\Phi$ abhängig, sondern vom Strukturtensor S, der im folgenden Abschnitt ausführlich erklärt wird.

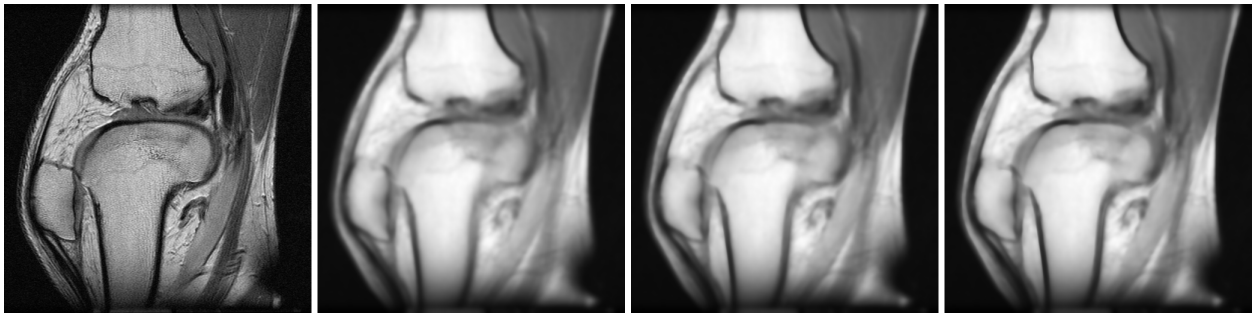


Abbildung 15: Nichtlineare anisotrope Diffusion (2D) (hier: EED).
Datensatz: Knee, Größe: $512 \times 512 \times 88$, Slice 49
Links: t = 0.0 (Originalbild), danach dreimal zum Zeitpunkt t = 5.0
mit $\lambda = 0.05$, $\lambda = 0.03$, $\lambda = 0.02$. (100 Iterationen, $\Delta t = 0.05$)
Das Rauschen innerhalb und entlang von Kanten wird gut geglättet, dennoch bleiben die Kanten lange erhalten (vgl. auch Abb. 11, S. 32).

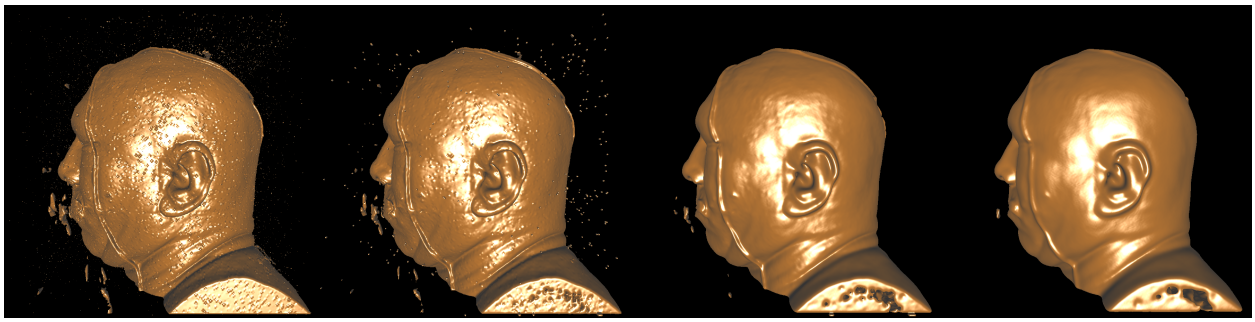


Abbildung 16: Nichtlineare anisotrope Diffusion (3D) (hier: EED), $\Delta t = 0.05$.
Isooberflächenwert: 50 von 255
Von links: Verrauschter Datensatz (ca. 5% betroffene Voxel, additives gaußsches Rauschen, Standardabweichung 10%), Nichtlineare inhomogene Diffusion mit 5, 25, 50 Iterationen. Die Kanteninformation bleibt trotz guter Glättung lange erhalten.

4.10 Diffusions- und Strukturtensor

Es ist wichtig zu verstehen, dass ein Tensor zweiter Stufe den Fluss beliebig ausrichten kann. Dazu zunächst ein kleines Beispiel: Sei der Gradient ortsabhängig $\nabla\Phi = (1x, 2y, 3z)^T$, und der Diffusionstensor die Einheitsmatrix $D = E$. Wir erhalten den isotropen Fall und der Fluss wird entsprechend der einzelnen Gradientenkomponenten induziert:

$$\partial_t\Phi = div(E\nabla\Phi) = div\left(\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1x\\2y\\3z\end{pmatrix}\right) = div\begin{pmatrix}1x\\2y\\3z\end{pmatrix} = 1+2+3 = 6$$

Wählen wir statt der Einheitsmatrix $D = \begin{pmatrix}0&1&0\\2&0&0\\0&0&0,5\end{pmatrix}$ ergibt sich:

$$\partial_t\Phi = div(D\nabla\Phi) = div\left(\begin{pmatrix}0&1&0\\2&0&0\\0&0&0,5\end{pmatrix}\begin{pmatrix}1x\\2y\\3z\end{pmatrix}\right) = div\begin{pmatrix}2y\\2x\\1,5z\end{pmatrix} = 0+0+1,5 = 1,5$$

Offensichtlich handelt es sich hier um einen echten anisotropen Fluss, da sich die Divergenz nun aus verschiedenen Raumrichtungen unterschiedlich stark zusammensetzt. Wir sehen also, dass der Diffusionstensor die absolute Größe des Flusses sowie dessen Flussrichtung bzw. dessen Zusammensetzung aus den verschiedenen Raumrichtungen bestimmt.

Wir müssen nun den Fluss über den Diffusionstensor so ausrichten, dass er das Rauschen des Bildes bevorzugt entlang der Kanten glättet. In der Literatur finden sich mit dem Strukturtensor und der Hesse-Matrix zwei verschiedene Vorgehensweisen, die beide darauf abzielen, den Diffusionstensor mithilfe einer Rotation an der Oberflächenstruktur auszurichten. Die Hauptrichtungen des transformiert dargestellten Diffusionstensors sollen dabei die Partialflüsse einzeln und entlang der Struktur bzw. normal dazu steuern. Beide Vorgehensweisen sollen in diesem bzw. dem folgenden Abschnitt erläutert werden.

Weickert verwendet in [Wei98] den sogenannten Strukturtensor. Der Strukturtensor kann leicht aus dem Gradienten bestimmt werden und hat die Aufgabe die vorherrschende Geometrie durch lokale Auswertung des Gradienten zu beschreiben.

Der Gradient ist eine vektorielle Größe deren Komponenten die Ableitungen in den einzelnen Dimensionen sind: $\nabla\Phi(\vec{x}) = \left(\frac{\partial\Phi(\vec{x})}{\partial x_1}, \ldots, \frac{\partial\Phi(\vec{x})}{\partial x_n}\right)^T = (\Phi_{x_1}, \ldots, \Phi_{x_n})^T$.

Der Strukturtensor setzt nun diese Richtungsableitungen in Bezug zueinander, indem der Gradient über das Tensorprodukt (äußeres Produkt) mit sich selbst multipliziert wird. Um eine Mittelung über die lokale Umgebung zu erreichen wird dieses dyadische Produkt zusätzlich meist noch über einen Filter, zum Beispiel die Faltung mit einem Gaußkern G_σ, ermittelt.

$$S(\nabla\Phi) \quad = G_\sigma * (\nabla\Phi \otimes \nabla\Phi) = G_\sigma * \begin{pmatrix} \Phi_x^2 & \Phi_x\Phi_y & \Phi_x\Phi_z \\ \Phi_y\Phi_x & \Phi_y^2 & \Phi_y\Phi_z \\ \Phi_z\Phi_x & \Phi_z\Phi_y & \Phi_z^2 \end{pmatrix} \tag{4.18}$$

Sowohl der einfache Strukturtensor als auch dessen Faltung ergibt eine symmetrische Matrix, da die Summation symmetrischer Matrizen ebenfalls symmetrische Matrizen ergibt. Für eine symmetrische Matrix $S \in \mathbb{R}^{n\times n}$ gelten insbesondere zwei Sätze, die in vielen Lehrbüchern und Skripten (vgl. [Wel07]) nachgelesen werden können:

- S hat ausschließlich reelle Eigenwerte
- die Eigenvektoren zu verschiedenen Eigenwerten sind orthogonal

Durch Eigenwertzerlegung des Strukturtensors erhalten wir drei reelle Eigenwerte und dazugehörige Eigenvektoren. Ohne Beschränkung der Allgemeinheit seien dies $\lambda_1 < \lambda_2 < \lambda_3$ mit den normierten Eigenvektoren $\vec{u}, \vec{v}, \vec{n}$. (Dass tatsächlich die Normale $\vec{n}$ ein Eigenvektor ist, liegt daran, dass der lineare Operator $\vec{n}\vec{n}^T$ auf den Spann der Normale abbildet. Mehr dazu im nächsten Abschnitt).
Die Eigenvektoren stehen senkrecht aufeinander und bilden normiert eine Orthonormalbasis V. Die Inverse einer orthogonalen Matrix ist ihre Transponierte. Durch eine Basentransformation mit $V^T = \{\vec{u}, \vec{v}, \vec{n}\}^T$, die den Raum in den ersten zwei Dimensionen auf die Hauptkrümmungsrichtungen (principal curvature directions, vgl. [PR02]) und in der dritten Dimension auf die Normale (parallel zum Gradienten) zur unterliegenden Struktur ausrichtet, erhalten wir eine diagonalisierte Darstellung Λ der ursprünglichen Matrix – die Diagonalelemente sind gerade die zu den Eigenwerten gehörenden Eigenwerte.

$$S = V^T\Lambda V = V^T \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & |\nabla\Phi| \end{pmatrix} V$$

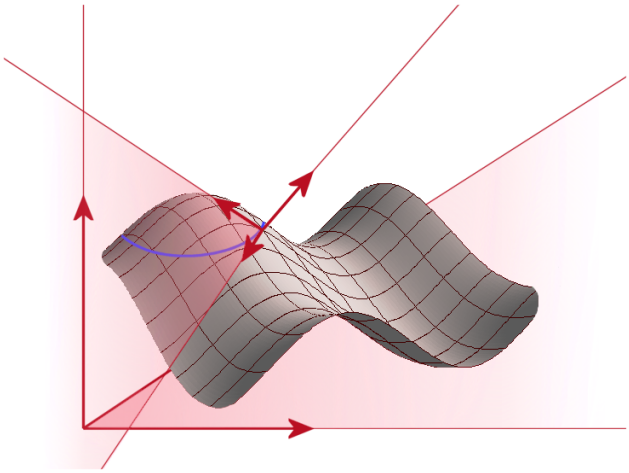


Abbildung 17: Schematische Darstellung einer Basis auf der Tangentialebene der Isooberfläche

Der auf diese Weise dargestellte Strukturtensor kann nun zur Definition des Diffusionstensors herangezogen werden, indem in der Regel die Diagonalelemente mehr oder weniger willkürlich dem Diffusionsziel angepasst werden. In [Wei96, S. 17] schlägt Weickert (sinngemäß auf 3 Raumdimensionen erweitert) folgenden Diffusionstensor für die »edge enhancing diffusion« (EED) vor:

$$D = V\Lambda^* V^T = V \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & g(|\nabla\Phi|) \end{pmatrix} V^T$$

Dabei ist g wieder die monoton fallende Diffusivitätsfunktion (4.13). Im Inneren eines annähernd homogenen Bereiches, wir nehmen hier $|\nabla\Phi| \ll \lambda$ an, ergibt sich $\Lambda \approx E$ und es findet prinzipiell isotrope Diffusion statt. Angewandt auf einen Bildbereich nahe einer Kante erhalten wir jedoch aufgrund des dritten Diagonalelements – also für $g(|\nabla\Phi \gg \lambda|) \approx 0$ – annähernd keinen Fluss entlang der Gradientenrichtung. Parallel zur Isooberfläche findet bei diesem Modell immer maximale Diffusion statt.

Zusammenfassung der bisherigen Überlegungen: Aus der Bildfunktion Φ können wir den Gradienten $\nabla\Phi$ bestimmen. $\nabla\Phi$ führt über das Tensorprodukt mit sich selbst zum Strukturtensor S. Dieser kann derart in $S = V\Lambda V^T$ zerlegt werden, dass eine orthonormale Basentransformation V und ihre Inverse $V^{-1} = V^T$ bekannt werden. Mithilfe von V kann nun ein im Prinzip willkürlicher Diffusionstensor definiert werden, der die Partialflüsse entlang der Hauptkrümmungsrichtungen und orthogonal zur Isooberfläche ausrichtet. Diese Überlegungen finden sich soweit auch in einer Vielzahl aktueller Publikationen, z.B. [ST06], die sich mit volumetrischer Diffusion befassen. Genaugenommen sind wir aber nicht an der kompletten Diagonalzerlegung $S = V^T\Lambda V = V^T$ interessiert. Lediglich die Transformation

V ist für unsere Defintion des Diffusionstensors interessant. Der folgende Abschnitt untersucht nun, inwiefern man eine Basis V finden kann, die den Raum an der Tangentialebene der Isooberfläche ausrichtet, ohne unnötige Berechnungen durchzuführen.

4.11 Hesse-Matrix und Tangentialebene der Isooberfläche

Die Eigenwerte einer 2×2 Matrix können einfach über geschlossene Formeln berechnet werden. Das charakteristische Polynom einer 3×3 Matrix hat jedoch Polynomgrad 3 und ist entsprechend aufwändig zu lösen.
Die folgenden Überlegungen zielen darauf ab mit relativ wenig Aufwand eine Basentransformation $V : \mathbb{R}^3 \mapsto \mathbb{R}^3$ zu finden, die ein Koordinatensystem beschreibt, das senkrecht auf der Tangentialebene der Isooberfläche am betrachteten Punkt steht. Außerdem soll das Koordinatensystem dabei so ausgerichtet sein, dass die zwei in der Tangentialebene liegenden Basisvektoren entlang der Hauptkrümmungsrichtungen[10] der Isooberfläche zeigen.

In [HSS+05, »4.1 Differential Surface Properties«, S. 6f,] und [KWTM03, »3. Curvature Measurement«, S. 2f] wird beschrieben, wie man basierend auf der Gradienteninformation die Krümmung der Oberfläche charakterisieren kann und zusätzlich zur Isooberflächennormale die zwei Hauptrichtungskrümmungen erhält. Wir benötigen zunächst einfache Definitionen.

Definition (Oberflächennormale):
Unter der Annahme, dass im Inneren des Volumens höhere Dichtewerte vorliegen, definieren wir die *Oberflächennormale* über den Gradienten $\vec{g} = \nabla\Phi = \left[\frac{\partial\Phi}{\partial x}\frac{\partial\Phi}{\partial y}\frac{\partial\Phi}{\partial z}\right]^T$ als

$$\vec{n} = -\frac{\vec{g}}{|\vec{g}|}$$

Dass die Oberflächennormale tatsächlich im Spann des Gradienten liegt, lässt sich leicht wie folgt veranschaulichen: Der Gradient zeigt in Richtung des stärksten Dichteanstiegs innerhalb des Volumens. Damit zeigt er ebenfalls exakt in die Richtung, in die sich die Isooberfläche verschiebt, wenn man einen höheren Isowert wählt.

[10] Anmerkung: Die Hauptkrümmungsrichtungen der Isooberfläche stehen immer senkrecht aufeinander. Siehe z.B. [MDSB02], [HZ00] oder Grundlagenbücher zur Differentialgeometrie, wie [Kre91, S. 118ff] und [Kre68]

Analog zum Gradienten, handelt es sich auch bei $\vec{n}$ eigentlich um ein Vektorfeld $\vec{n}(\vec{x})$, das an jedem Ort $\vec{x}$ die jeweilige Oberflächennormale angibt. Die Krümmung einer Oberfläche lässt sich über das Änderungsverhältnis von Position und Oberflächennormale, also den Gradienten von $\vec{n}$ beschreiben: Bewegen wir uns in infinitesimaler Umgebung eines Punktes $\vec{x}$, ändert sich die Oberflächennormale entsprechend. Ist beispielsweise die betrachtete Isooberfläche eine Ebene und stehen alle Normalen parallel, gilt $\nabla\vec{n}^T = 0$ und es ist keine Krümmung vorhanden.
Wir sehen also, dass $\nabla\vec{n}^T$, die Ableitung des Normalenfeldes am Ort $\vec{x}$, die Krümmungsinformation der Oberfläche beinhaltet. Gleichzeitig ist $\nabla\vec{n}^T$ eine 3×3 Matrix und kann als linearer Operator aufgefasst werden. Es gilt:

$$\nabla\vec{n}^T = -\frac{1}{|\vec{g}|}(E - \vec{n}\vec{n}^T)H \tag{4.19}$$

In [KWTM03] zeigen Kindlmann et al. die komplette Herleitung dieser Umformung. Für unser Verständnis entscheidend ist aber deren Schlussfolgerung: Diese Darstellung gibt sehr intuitiv Aufschluss über den Operator $\nabla\vec{n}^T$. E bezeichnet hierbei die Einheitsmatrix, H die Hesse-Matrix, welche alle Kombinationen der zweiten partiellen Ableitungen der Bildfunktion Φ trägt:

$$H = \nabla\vec{g}^T = \begin{bmatrix} \frac{\partial^2\Phi}{\partial x^2} & \frac{\partial^2\Phi}{\partial x\partial y} & \frac{\partial^2\Phi}{\partial x\partial z} \\ \frac{\partial^2\Phi}{\partial x\partial y} & \frac{\partial^2\Phi}{\partial y^2} & \frac{\partial^2\Phi}{\partial y\partial z} \\ \frac{\partial^2\Phi}{\partial x\partial z} & \frac{\partial^2\Phi}{\partial y\partial z} & \frac{\partial^2\Phi}{\partial z^2} \end{bmatrix} \tag{4.20}$$

So wie der Gradient das Änderungsverhalten der Dichtefunktion Φ beschreibt, so beschreibt die Hesse-Matrix das Änderungsverhalten des Gradienten, bzw. der Oberflächennormale in der infinitesimalen Nähe des betrachteten Ortes. Die Änderungen des Gradienten können nun in zwei Komponenten zerlegt werden, nämlich entlang des Gradienten und in der Tangentialebene. Kindlmann et. al. schreiben hierzu in [KWTM03]: »The changes in g have a component along g (the gradient can change length), and a component within the tangent plane (the gradient can change direction). For the purposes of describing curvature, only the latter component matters.«.

Vernachlässigen wir den Skalierungsfaktor $-\frac{1}{|\vec{g}|}$ in (4.19) und untersuchen wir den restlichen Term. Es gilt $(\vec{n}\vec{n}^T)\vec{x} = (\vec{n}\vec{x})\vec{n}$, also projiziert der Operator $(\vec{n}\vec{n}^T)$ einen beliebigen Punkt des Raumes auf den Spann der Normalen. Somit können wir nun über Subtrakti-

on von der Einheitsmatrix eine Projektion angeben, die einen beliebigen Punkt $\vec{x}$ in das Komplement des Spanns projiziert (s. Abb. 18).

Definition (Projektion in die Tangentialebene):
Wir definieren die Projektion P in die Tangentialebene der Isooberfläche als

$$P = (E - \vec{n}\vec{n}^T) = (E - \frac{\vec{g}\vec{g}^T}{|\vec{g}|^2})$$

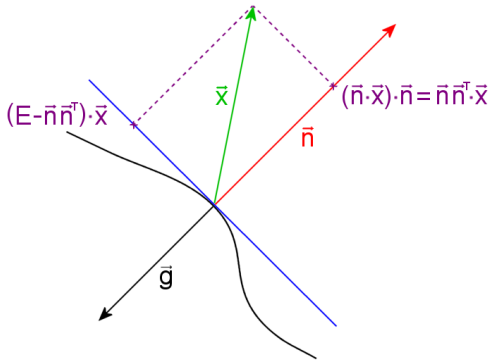


Abbildung 18: Projektion auf den Spann der Normale und dessen Komplement

Die Projektion P extrahiert aus der Hesse-Matrix H den Anteil der Richtungsänderung des Gradienten innerhalb der Tangentialebene.
Über P definieren wir nun den sogenannten Shape Operator S wie folgt:

$$S = P^T \frac{H}{|g|} P \qquad\qquad P = E - \frac{g g^T}{|g|^2}$$

Der Shape Operator S besitzt einen größeren und einen kleineren Eigenwert mit Eigenvektoren in der Tangentialebene, welche die Hauptkrümmungsrichtungen angeben. In [KWTM03] wird auch gezeigt, dass S symmetrisch ist und somit das charakteristische Polynom wieder 3 reelle Lösungen besitzt und das Eigenwertproblem 3 orthogonale Eigenvektoren liefert. Dennoch ist der Rechenaufwand für das Eigenwertproblem einer 3×3 Matrix relativ hoch, zumal bereits ein Eigenvektor, die Normale, bekannt ist.

Genau das machen sich Hadwiger et al. in [HSS+05]) zu Nutze und ermitteln die erste und zweite Hauptkrümmung (λ_1, λ_2) der Isooberfläche direkt in der Tangentialebene: »The principal curvature magnitudes amount to two eigenvalues of the shape operator S, defined as the tangent space projection of the normalized Hessian«.

Wir können die Eigenwertanalyse im zweidimensionalen Tangentialraum durchführen, ohne dass wir S direkt berechnen. Die Transformation des Shape Operators in eine beliebige orthogonale Basis $(\vec{u}, \vec{v})$ des Raumes der Tangentialebene lautet:

$$S' = \begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix} = (\vec{u}, \vec{v})^T \frac{H}{|g|} (\vec{u}, \vec{v})$$

Um ein beliebiges $\vec{u}$ und $\vec{v}$ aus der Tangentialebene zu berechnen können wir beispielsweise wie folgt vorgehen: Wir wählen den ersten kanonischen Einheitsvektor des euklidischen Raumes, $\vec{e}_1 = (1,0,0)^T$, in der Annahme, dass $\vec{e}_1 \nparallel \vec{n}$ gilt, die Vektoren also linear unabhängig sind. Wir bilden das Kreuzprodukt $\vec{u} = \vec{e}_1 \times \vec{n}$. Ist $\vec{u} = \vec{0}$ war die Annahme falsch und die Vektoren sind linear abhängig. In diesem Fall wiederholen wir den Vorgang mit dem zweiten kanonischen Einheitsvektor, $\vec{e}_2 = (0,1,0)^T$.
$\vec{u}$ steht nun senkrecht auf $\vec{n}$ und liegt damit in der Tangentialebene. Wir ergänzen die neue Basis mit $\vec{v} = \vec{u} \times \vec{n}$.

Es gilt $\vec{u}, \vec{v} \in \mathbb{R}^3$ wir erhalten also mit S' eine 2×2 Matrix, deren Eigenwerte $\lambda_{1,2}$ wir mit Hilfe einer geschlossenen Formel, die sich aus der Lösung des charakteristischen Polynoms von S' ergibt, direkt berechnen können:

$$\begin{aligned} \det(S' - \lambda E) = \begin{vmatrix} s_{11} - \lambda & s_{12} \\ s_{21} & s_{22} - \lambda \end{vmatrix} &= 0 \\ (s_{11} - \lambda)(s_{22} - \lambda) - s_{12}s_{21} &= 0 \\ \lambda^2 - (s_{11} + s_{22})\lambda + (s_{11}s_{22} - s_{12}s_{21}) &= 0 \\ \lambda^2 - \operatorname{trace}(S')\lambda + \det(S') &= 0 \end{aligned}$$

$$\Rightarrow \lambda_{1,2} = \frac{\operatorname{trace}(S')}{2} \pm \sqrt{\frac{\operatorname{trace}(S')^2}{4} - \det(S')} \tag{4.21}$$

Mithilfe der Eigenwerte $\lambda_{1,2}$ können wir nun die Eigenvektoren bestimmen. Die zugehörige Formel in [HSS+05] ist fehlerhaft wiedergegeben und kann in korrekter Fassung bei [Sig06, S. 96] gefunden werden. Da die vollständige Herleitung dort nicht zu finden ist, wird sie im Folgenden erläutert (vgl. hierzu auch [Kni04]). Die Eigenvektoren $w^*_{1,2}$ werden zunächst bezüglich der Basis (u, v) bestimmt und dann zurück in den 3D Raum transformiert:

$$w_1^* = \begin{pmatrix} w_{1u}^* \\ w_{1v}^* \end{pmatrix} = \begin{cases} \begin{pmatrix} \lambda_1 - s_{22} \\ s_{12} \end{pmatrix} & \text{für } s_{12} = s_{21} \neq 0 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \text{für } s_{12} = s_{21} = 0 \end{cases} \tag{4.22}$$

$$w_2^* = \begin{pmatrix} w_{2u}^* \\ w_{2v}^* \end{pmatrix} = \begin{cases} \begin{pmatrix} \lambda_2 - s_{22} \\ s_{12} \end{pmatrix} & \text{für } s_{12} = s_{21} \neq 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \text{für } s_{12} = s_{21} = 0 \end{cases} \tag{4.23}$$

Die Transformation der zweidimensionalen Eigenvektoren in den ursprünglichen Objektraum erfolgt über die Erweiterung der (u, v)-Tangentialebenenbasis um eine dritte Dimension und die Rücktransformation dieses Raumes in die Ursprungsorientierung mittels $V = \{\vec{u}, \vec{v}, \vec{n}\}$:

$$w_i = \begin{pmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{pmatrix} \begin{pmatrix} w_{iu}^* \\ w_{iv}^* \\ 0 \end{pmatrix} = \begin{pmatrix} u_x w_{iu}^* + v_x w_{iv}^* \\ u_y w_{iu}^* + v_y w_{iv}^* \\ u_z w_{iu}^* + v_z w_{iv}^* \end{pmatrix} \tag{4.24}$$

4.12 Algorithmus zur Diffusionstensorgewinnung

Wir können nun aufbauend auf den vorangegangenen Überlegungen einen vollständigen, leicht zu implementierenden Algorithmus angeben, der uns die Definition eines anisotropen Diffusionstensors erlaubt. Kern des Algorithmus ist eine mittels geschlossener Formeln durchführbare 2×2 Eigenwertanalyse der in den Tangentialraum der Isooberfläche projizierten Hesse-Matrix. Der vollständige Ablauf kann wie folgt zusammengefasst werden:

1. Wir bilden aus Φ den Gradienten $\vec{g} = \nabla\Phi$
2. Wir bilden die Isooberflächennormale $\vec{n} = -\frac{\vec{g}}{|\vec{g}|}$
3. Wir bilden die Hesse-Matrix $H = \nabla g$
4. Wir ergänzen $\vec{n}$ mit $\vec{u}$ und $\vec{v}$ zu einer orthonormalen Basis, deren u-v Ebene die Tangentialebene der Isooberfläche ist
5. Wir projizieren H auf die Tangentialebene und erhalten die 2×2 Matrix S'
6. Mit Hilfe der geschlossenen Formel (4.21) erhalten wir die Eigenwerte $\lambda_{1,2}$

7. Mit Hilfe der geschlossenen Formel (4.22) erhalten wir die zugehörigen Eigenvektoren $w^*_{1,2}$ bezüglich der (u, v) Basis

8. Wir sortieren o.b.d.A. die Eigenwerte und -vektoren so, dass $\lambda_1 < \lambda_2$ gilt – falls das notwendig für die Definition des Diffusionstensors ist

9. Mit Hilfe der geschlossenen Formel (4.24) transformieren wir die 2D Eigenvektoren zurück in den Objektraum und erhalten die 3D Eigenvektoren w_1, w_2

10. Wir definieren $V = (w_1, w_2, n)$ und $V^{-1} = V^T$

11. Wir definieren $D = V \cdot \text{diag}(1, 1, g(\nabla\Phi)) \cdot V^{-1}$

Schritt 11 definiert einen Diffusionstensor, der normal zur Kantenrichtung die Glättung unterdrückt. Dieser Diffusionsfilter ist allgemein als »Edge Enhancing Diffusion« bekannt. Entsprechend können abgewandelte Diffusionstensoren für anisotrope Diffusionsfilter mit anderen Zielsetzungen definiert werden.

4.13 Der »Custom made« - Diffusionstensor

Prinzipiell kann jeder durch einen Diffusionstensor ausgerichtete Fluss anisotrop sein. Der spezielle Fall den Fluss genau entlang der Kanten auszurichten und orthogonal dazu zu unterdrücken, wird allgemein als »Edge Enhancing Diffusion« (EED) bezeichnet und ist in der Regel auch das, was man unter *der* »nichtlinearen anisotropen Diffusion« versteht.
Meist ist die Eigenwertzerlegung der Hesse-Matrix buchstäblich eine sinnvolle Basis zur Definition des Diffusionstensors: Die Inverse der sich ergebenden Rotationsmatrix kann leicht durch Transponation angegeben werden und beide Matrizen zusammen ermöglichen eine Darstellungsmatrix, welche entlang der Hauptkrümmungsrichtungen Diagonalgestalt hat. Es sind aber durchaus auch andere Definitionen denkbar und sinnvoll.

Ein weiterer prominenter Vertreter der nichtlinearen anisotropen Diffusionsfilter ist die sogenannte »Coherence Enhancing Diffusion« (CED), bei der mithilfe des Diffusionstensors der induzierte Fluss entlang von eindimensionalen Strukturen ausgerichtet wird. »CED is able to connect interrupted lines and improve flow-like structures« ([FLL07, S. 63]). Die »Kohärenz« ist dabei ein Maß für die strukturelle Einheit der Umgebung und kann über die Verhältnisse der zu den Eigenvektoren gehörigen Eigenwerte definiert werden.

Im dreidimensionalen Raum erhalten wir drei Eigenwerte λ_i und sind so in der Lage zwischen unstrukturierten bzw. isotropen Gebieten ($\lambda_1 \approx \lambda_2 \approx \lambda_3$), Ebenen ($\lambda_1 \approx \lambda_2 \ll \lambda_3$) und linienartigen Strukturen ($\lambda_1 \ll \lambda_2 \approx \lambda_3$) zu unterscheiden ($\lambda_3$ sei hier der Eigenwert

zum Eigenvektor w_3, also der Isooberflächennormale).
Weickert erläutert die gesamte CED sehr ausführlich und gibt in [Wei99, S. 15, Formel (14)] die genaue Definition der Diagonalelemente der CED an.

Manche Anwendungsfälle erfordern die Kombination verschiedener Diffusionsfilter, so zum Beispiel der hybride EED – CED Diffusionsfilter, der je nach lokaler Bildstruktur anhand der Eigenwerte entscheidet, ob EED oder CED zum Einsatz kommt [FH99, S. 388f].

Im Rahmen dieser Arbeit wurde der Edge Enhancing Diffusion Filter als stellvertretender nichtlinearer anisotroper Diffusionsfilter implementiert. Da weitere Berechnungen auf bereits vorhandenen Daten (z.B. die Kohärenz basierend auf den Eigenwerten) die »arithmetische Intensität« steigern, ist davon auszugehen, dass GPU Implementierungen bei komplizierten Diffusionstensoren weitere Geschwindigkeitsvorteile gegenüber reinen CPU Implementierungen bieten.

4.14 Zusammenfassung

In diesem Kapitel wurden die einzelnen Diffusionfilter mathematisch formuliert und deren Eigenschaften besprochen. Die Erweiterung des linearen homogenen Diffusionsfilters zum nichtlinearen inhomogenen Diffusionsfilter erhält die Strukturen des Volumendatensatzes sehr viel länger, glättet jedoch in der Nähe von Kanten das Rauschen nur ungenügend.

Um dieses Problem zu umgehen ist die anisotrope Ausrichtung des Diffusionsflusses notwendig. Die Definition des dazu benötigten Diffusionstensors erfordert eine Basis, die senkrecht zur Isooberfläche ausgerichtet ist und entlang der Kanten glättet. Um diese Basis zu erhalten, ist eine Eigenwertzerlegung der lokalen Hesse-Matrix erforderlich. Da die Eigenwertzerlegung im dreidimensionalen Objektraum sehr aufwändig ist, haben wir eine Projektion entlang der Oberflächennormalen auf die Tangentialebene der Isooberfläche vorgenommen und konnten mit geschlossenen Formeln den gesamten Algorithmus zur Diffusionstensorgewinnung angeben.

Bevor wir uns mit der Diskretisierung der Diffusionsfilter beschäftigen, wollen wir uns im nächsten Kapitel mit möglichen Prefiltern beschäftigen, die eine Vorglättung des Datensatzes ermöglichen und die Qualität der ersten und zweiten Ableitungen erhöhen sollen.

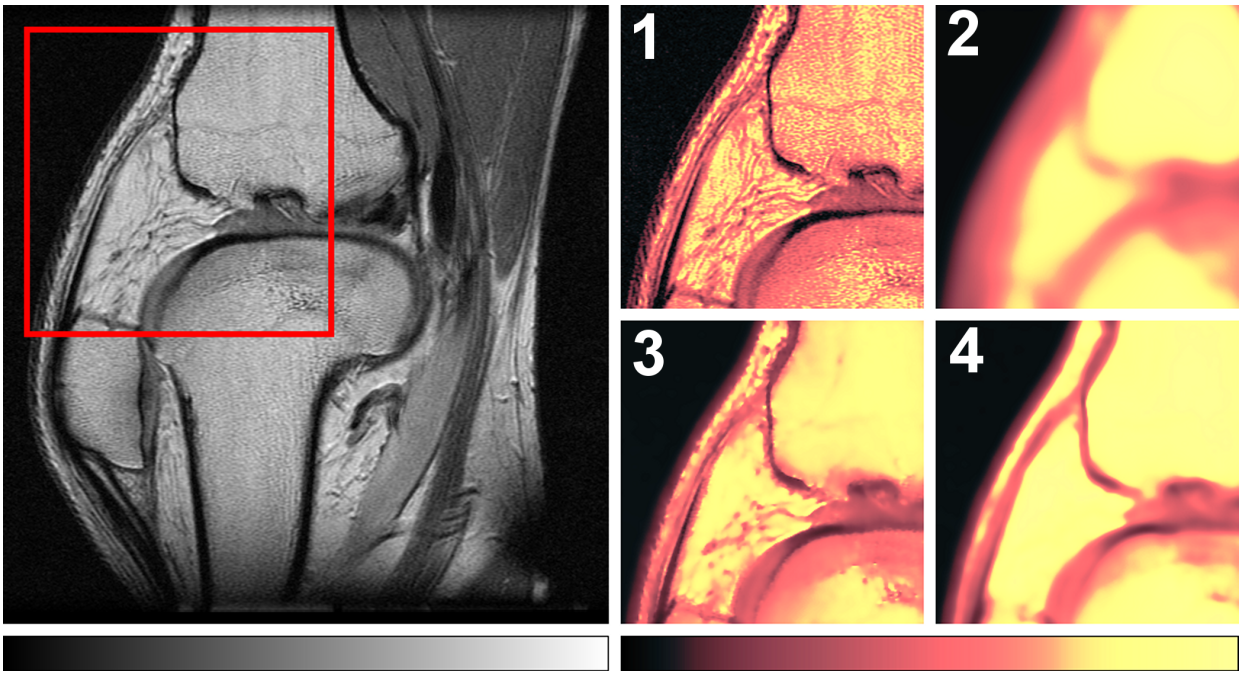


Abbildung 19: Die Diffusionsfilter im Vergleich (2D).
Datensatz: Knee, Größe: $512 \times 512 \times 88$, Slice 49. *Links*: Originalbild (t = 0.0), *Rechts*: Ausschnitte mit Transferfunktion: 1. Original, 2. Lineare homogene Diffusion, 3. Nichtlineare inhomogene Diffusion, 4. Nichtlineare anisotrope Diffusion (EED). 2-4. zum Zeitpunkt t = 5.0 ($\Delta t = 0.05$, jeweils 100 Iterationen.)
Man kann deutlich erkennen, wie das Rauschen (»rot-gelbe Maserung des Knochens«) geglättet wird. Die gute Glättung der linearen homogenen Diffusion (2) zerstört die Kanten, die nichtlineare inhomogene Diffusion (3) erhält die Kanten, erreicht aber insbesondere in der Nähe von Kanten bei weitem nicht die gleichen Glättungseigenschaften. Die EED (4) kann schließlich stark glätten und dennoch die Kanten lange Zeit erhalten.

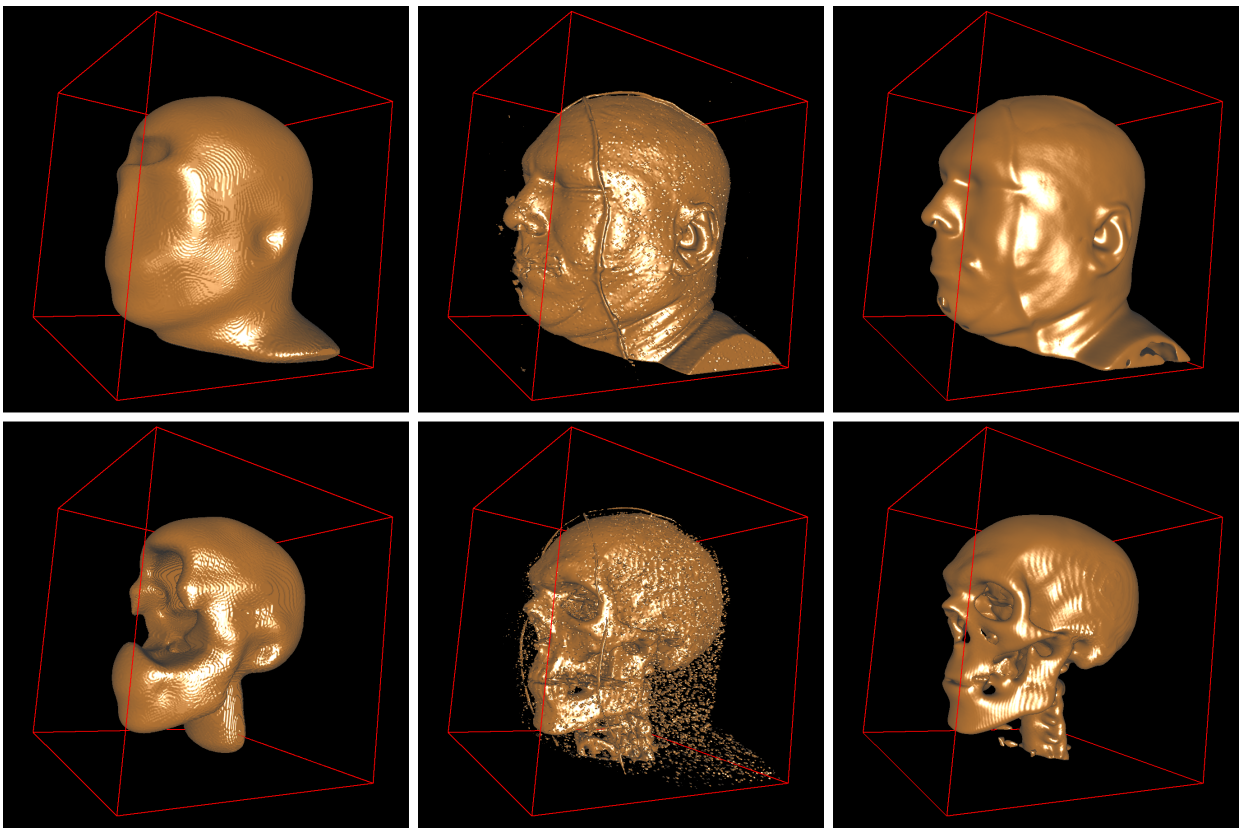


Abbildung 20: Die Diffusionsfilter im Vergleich (3D), jeweils nach 100 Iterationen, $\Delta t = 0.05$.
Oben: Isooberflächenwert: 50 von 255. *Unten*: Isooberflächenwert: 82 von 255.
Von links: Lineare homogene Diffusion, Nichtlineare inhomogene Diffusion, Edge enhancing Diffusion
Die lineare homogene Diffusion glättet die Kanten (gut zu sehen: das Lumen der Stirnhöhle zerstört die Oberflächenstruktur der Stirn). Der nichtlineare inhomogene Diffusionsfilter glättet schnell Fehler in homogenen Regionen, erfasst das Rauschen in der Nähe der Kanten jedoch fast nicht. Der edge enhancing diffusion Filter erhält auch nach 100 Iterationen noch viele Strukturdetails.

5 Prefilter

Die allgemeine n-dimensionale Wärmeleitungsgleichung führt im Kontext der Bildverarbeitung (vgl. (3.6)) zu einer Differentialgleichung, die ganz allgemein gesprochen das Bild Φ auf zwei Arten nutzt: Einerseits induziert der Gradient den eigentlichen Diffusionsfluss, andererseits wird dieser Fluss zeitlich und räumlich über den Diffusionstensor gesteuert.
Der Diffusionstensor D ist entweder direkt eine Funktion des Gradienten oder allgemeiner eine Funktion eines Interest-Operators bzw. des Strukturtensors. Die Interest-Operatoren bzw. der Strukturtensor sind jedoch im Endeffekt wieder Funktionen von $\Phi(\vec{x}, t)$. Man kann also formulieren:

$$\partial_t \Phi = \mathrm{div}(D(S(\Phi))\nabla\Phi) \tag{5.1}$$

Hier sieht man deutlich, wie die Bildfunktion »qualitativ« ($D(S(\Phi))$) und »quantitativ« ($\nabla\Phi$) in die Berechnung einfließt.
Im Endeffekt ist also die Berechnung hochwertiger erster (für die Gradienten) bzw. zweiter Ableitungen (für die Hesse-Matrix bei der anisotropen Diffusion) entscheidend. Gerade zu Beginn des Diffusionsprozesses stören kleine Rauschanteile die diskrete Berechnung der Ableitungen enorm. Strukturtensoren, die Aussagen über die Beschaffenheit der Oberfläche liefern, werden daher in der Bildverarbeitung üblicherweise gewonnen, indem sie über eine Nachbarschaft gebildet und gaußgewichtet aufsummiert, also mit einem Gaußkern gefaltet werden.
Richten wir unser Augenmerk also auf die Anfangsbedingung des Diffusionsprozesses. In der ursprünglichen Formulierung der Diffusionsfilter lautete sie in den vorangegangenen Kapiteln stets:

$$\Phi(\vec{x}, 0) = \Phi_0(\vec{x}) \tag{5.2}$$

Um den Diffusionsprozess optimal einleiten zu können führt man nun in der Regel eine Vorfilterung F_{pre} (prefiltering) des Urbildes durch. Die Anfangsbedingung lautet in einem solchen Konzept also:

$$\Phi(\vec{x}, 0) = F_{pre}(\Phi_0(\vec{x})) \tag{5.3}$$

Der Vorfilter ist für uns mathematisch gesehen eine Abbildung $F_{pre} : (f : \mathbb{R}^n \mapsto \mathbb{R}) \mapsto (f^* : \mathbb{R}^n \mapsto \mathbb{R})$, die eine Funktion auf eine andere Funktion abbildet. Definieren wir den Vorfilter als Identität ($F_{pre} = Id, F_{pre}(\Phi_0) = \Phi_0$), erhalten wir die ursprüngliche Formulierung der

Anfangsbedingung. Wir sehen also, dass das Konzept der Vorfilterung lediglich eine Generalisierung der bisher behandelten Diffusionsprozesse darstellt bzw. der Diffusionsprozess ohne Vorfilterung in diesem Kontext als ein Spezialfall aufgefasst werden kann.
Um im Rahmen dieser Arbeit einen vollständigen Überblick geben zu können, werden im Folgenden kurz die gängigen Vorfilter vorgestellt, sowie einige Details im Hinblick auf deren Implementierung in CUDA erörtert.

5.1 Id Filter

Der Id Filter (Identitätsfilter) ist ein trivialer Filter, der jeden Wert der Bildfunktion auf sich selbst abbildet:

$$Id[\Phi]_x = \Phi_x \tag{5.4}$$

Im Rahmen der vorgestellten Arbeit ist dieser mathematisch uninteressante Filter jedoch zweifach erwähnenswert: Die Messergebnisse spiegeln den »minimalen Overhead« eines jeden denkbaren Filters wieder, da es keine einfachere Filterung als die Identität gibt. Die gemessenen Zeiten im Ergebnisteil enthalten im wesentlichen die Aufrufzeiten der Funktionen, sowie den unvermeidlichen, einmalig lesenden und schreibenden Speicherzugriff. Desweiteren liefert das Programmgerüst des Id Filters die grundlegende Codestruktur, in der auch alle anderen Filter implementiert sind und ist damit ein guter Startpunkt zur weiteren Exploration neuer Hardwarearchitekturen und des Quellcodes des Prototyps.

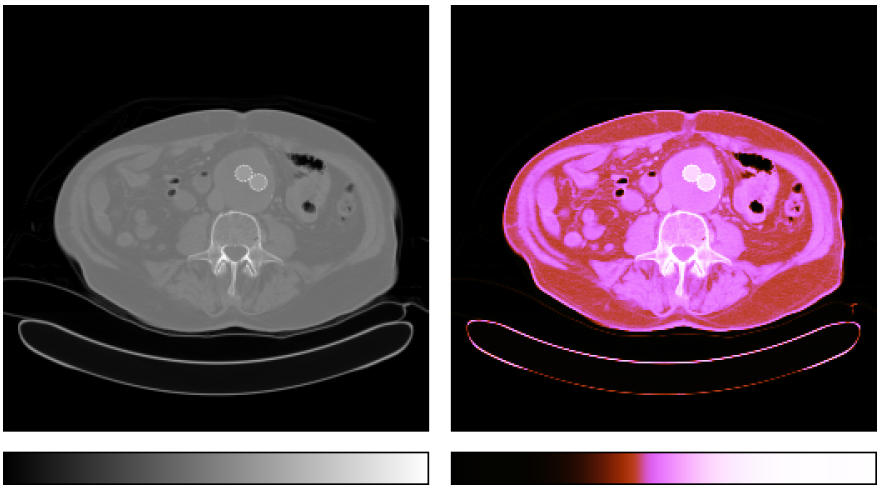


Abbildung 21: Id Filter, Datensatz: Stent, Größe: $352 \times 352 \times 352$
Der Datentransfer zur GPU überkompensiert alle Geschwindigkeitsvorteile:
0.09 Sekunden (CPU) vs. 0.17 Sekunden (GPU)

5.2 Boxfilter

Der Boxfilter, auch Mittelwertfilter genannt, ist ein separierbarer, nicht isotroper, lokaler Filter. Zur Berechnung des gefilterten Bildpunktes wird das arithmetische Mittel der Nachbarschaft ermittelt. Im einfachsten 3 dimensionalen Fall besteht der Filter aus einem mit $\frac{1}{27}$ gewichteten $3 \times 3 \times 3$ Kernel.

Die diskretisierte Formulierung lautet allgemein:

$$BF[\Phi]_y = \sum_{x \in \Omega} B_\sigma(y - x)\Phi_x \tag{5.5}$$

Wobei BF den Boxfilter, y das Ausgabevoxel und x ein Eingangsvoxel aus dem Bildbereich Ω bezeichnet. B_σ ist die normalisierte 3D-Boxfunktion mit Ausdehnung σ. Sie gibt 0 zurück, falls der Differenzvektor $(y - x)$ außerhalb des Quaders mit Ausdehnung σ liegt und andernfalls einen konstanten Wert $1/w_\sigma$. Normalisiert bedeutet hier, dass die Summe aller Rückgabewerte der Boxfunktion den Wert 1 ergibt. Der Filter ist nicht isotrop, da der Kernel aufgrund der rechteckigen Boxfunktion Voxel aus größerer Distanz in die Faltung einfließen lässt, als dies in reiner Hauptachsenrichtung der Fall ist.

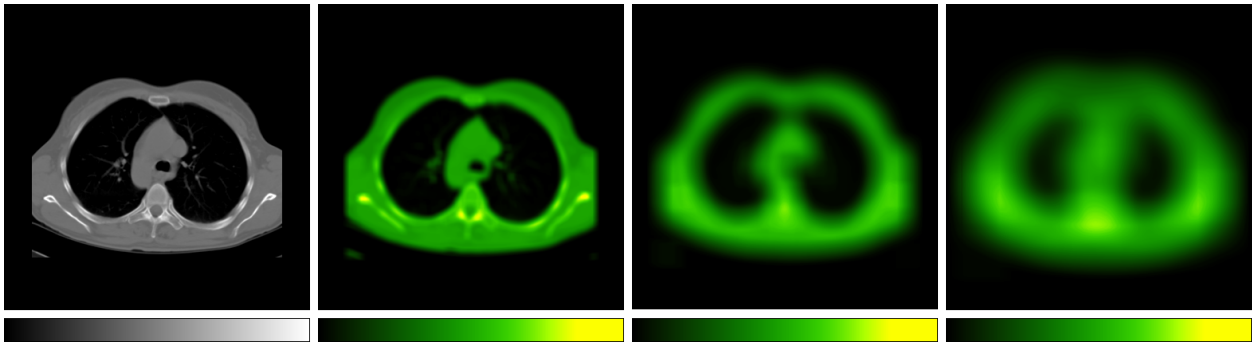


Abbildung 22: Boxfilter, Datensatz: Female Chest, Größe: $384 \times 384 \times 240$, Slice 155
Von links nach rechts: Original, Filterung mit Boxfunktion der Größe 9^3, 33^3, 65^3

Die Faltung wird im Dreidimensionalen entweder naiv implementiert, indem über drei Schleifen die Werte der Nachbarvoxel akkumuliert werden und der so erhaltene Wert anschließend durch die Anzahl der betrachteten Voxel geteilt wird, oder man nutzt die Separierbarkeit des Filters aus und filtert mit jeweils einer Schleife zuerst in x, dann in y und schließlich in z Richtung.

Für eine $m \times n \times l$ Nachbarschaft sind bei naiver Implementierung $m \cdot n \cdot l$ Additionen und 1 Division zur Normalisierung durchzuführen. Im separierten Fall sind es nur noch $m + n + l$

Additionen und 1 Divison. Die Komplexitätsklasse liegt also bei konstantem Volumendatensatz und Kernelgröße $n \times n \times n$ bei naiver Implementierung in $\mathcal{O}(n^3)$ und bei Nutzung der Separierbarkeit in $\mathcal{O}(n)$.

5.3 Gaußfilter

Der Gaußfilter ist ein separierbarer, isotroper, lokaler Filter. Die Filterung mit einem Gaußkern erfolgt ähnlich zu der eines Boxfilters. Die Faltung lässt sich im Dreidimensionalen mithilfe des Produktes dreier eindimensionaler Normalverteilungen wie folgt formulieren:

$$\begin{aligned}
\Phi_* &= \Phi_0 * \left(\varphi_{\sigma_x} \cdot \varphi_{\sigma_y} \cdot \varphi_{\sigma_z}\right) \\
&= \Phi_0 * \left(\frac{1}{\sigma_x\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{x}{\sigma_x}\right)^2} \cdot \frac{1}{\sigma_y\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{y}{\sigma_y}\right)^2} \cdot \frac{1}{\sigma_z\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{z}{\sigma_z}\right)^2}\right) \\
&= \Phi_0 * \left(\frac{1}{\sqrt{2\pi}^3\sigma_x\sigma_y\sigma_z} \cdot e^{-\frac{1}{2}\left(\left(\frac{x}{\sigma_x}\right)^2+\left(\frac{y}{\sigma_y}\right)^2+\left(\frac{z}{\sigma_z}\right)^2\right)}\right) \\
&= \Phi_0 * \varphi_{\sigma_x,\sigma_y,\sigma_z}
\end{aligned} \tag{5.6}$$

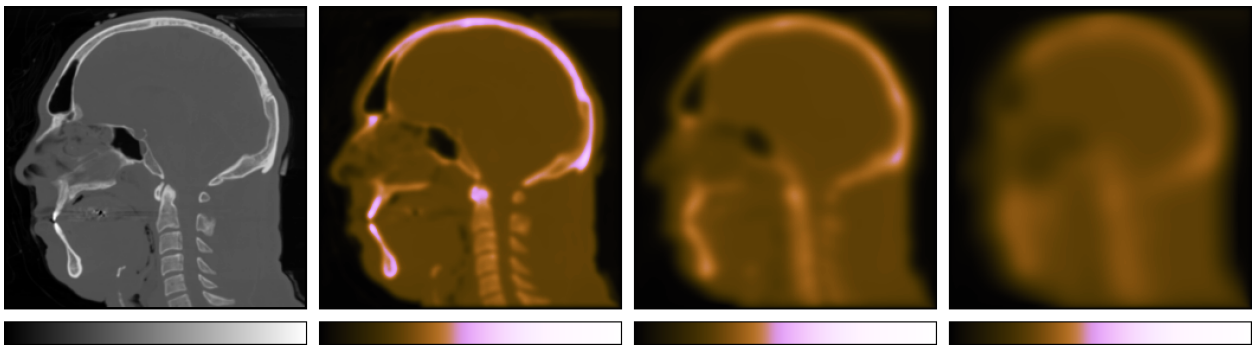


Abbildung 23: Gaußfilter, Datensatz: Head Male, Größe: $384 \times 384 \times 240$, Slice 155
Von links nach rechts: Original, Filterung mit isotropen Gaußkernen der Größe $\sigma_1 = 2.0, \sigma_2 = 5.0, \sigma_3 = 20.0$

Die Glättung mit einem solchen Gaußkern hat dabei also drei freie Parameter σ_x, σ_y und σ_z, welche die Form des Filterkerns in den drei Raumdimensionen bestimmen. Um die Berechnung der Faltung zu beschleunigen, macht man es sich zunutze, dass die Gaußfunktion relativ schnell Richtung 0 abfällt.

Dabei gilt bekanntlich für eine Normalverteilungsfunktion $\varphi_{0,\sigma}$ mit Varianz σ, dass rund 68%, 95% bzw. 99% des uneigentlichen Integrals $\int_{-\infty}^{+\infty} \varphi_\sigma$ im Intervall $[-\sigma, \sigma]$, $[-2\sigma, 2\sigma]$

bzw. $[-3\sigma, 3\sigma]$ anfallen. Man erhält also bereits sehr gute Ergebnisse, wenn man den dreidimensionalen Filterkern auf die Größe $[-t\sigma_x, t\sigma_x] \times [-t\sigma_y, t\sigma_y] \times [-t\sigma_z, t\sigma_z]$ beschränkt und $t \geq 2$ wählt.

Außerdem kann man sich, analog zum Boxfilter, die Separierbarkeit zu Nutze machen, um die Komplexitätsklasse des Gaußfilters von $\mathcal{O}(\sigma^3)$ auf $\mathcal{O}(\sigma)$ herabzusetzen.

Die diskretisierte Formulierung (vgl. (5.5)) lautet allgemein:

$$G[\Phi]_y = \sum_{x \in \Omega} \varphi_{\sigma_x, \sigma_y, \sigma_z}(||y - x||)\Phi_x \tag{5.7}$$

Hier bezeichnet $\varphi_{\sigma_x, \sigma_y, \sigma_z}$ die normalisierte Gaußfunktion.

In Abbildung 23 ist deutlich zu erkennen, wie größere Kernel einerseits das Bild glätten, andererseits die Kanten auswaschen.

5.4 Medianfilter

Der Medianfilter ist ein nichtlinearer, nicht separierbarer Filter und kann nicht durch eine Faltung dargestellt werden. Ein Voxelwert des Ausgangsbildes wird ermittelt, indem die Voxelwerte der Nachbarschaft aus dem Eingangsbild aufsteigend sortiert werden und der Median bestimmt wird.

Der Medianfilter eignet sich besonders für eine Vorfilterung von Volumendatensätzen, bei denen ein schwarz-weiß (»salt and pepper«) Rauschmodell angenommen werden kann. Die Komplexitätsklasse ist stark abhängig vom verwendeten Sortierverfahren und der Art der Daten (bzw. einer möglicherweise vorhandenen Vorsortierung). Im allgemeinen verwendet man Sortierverfahren mit logarithmischem Komplexitätsverhalten (z.B. Quicksort) und nimmt eine Komplexitätsklasse von $\mathcal{O}(n\ log\ n)$ an.

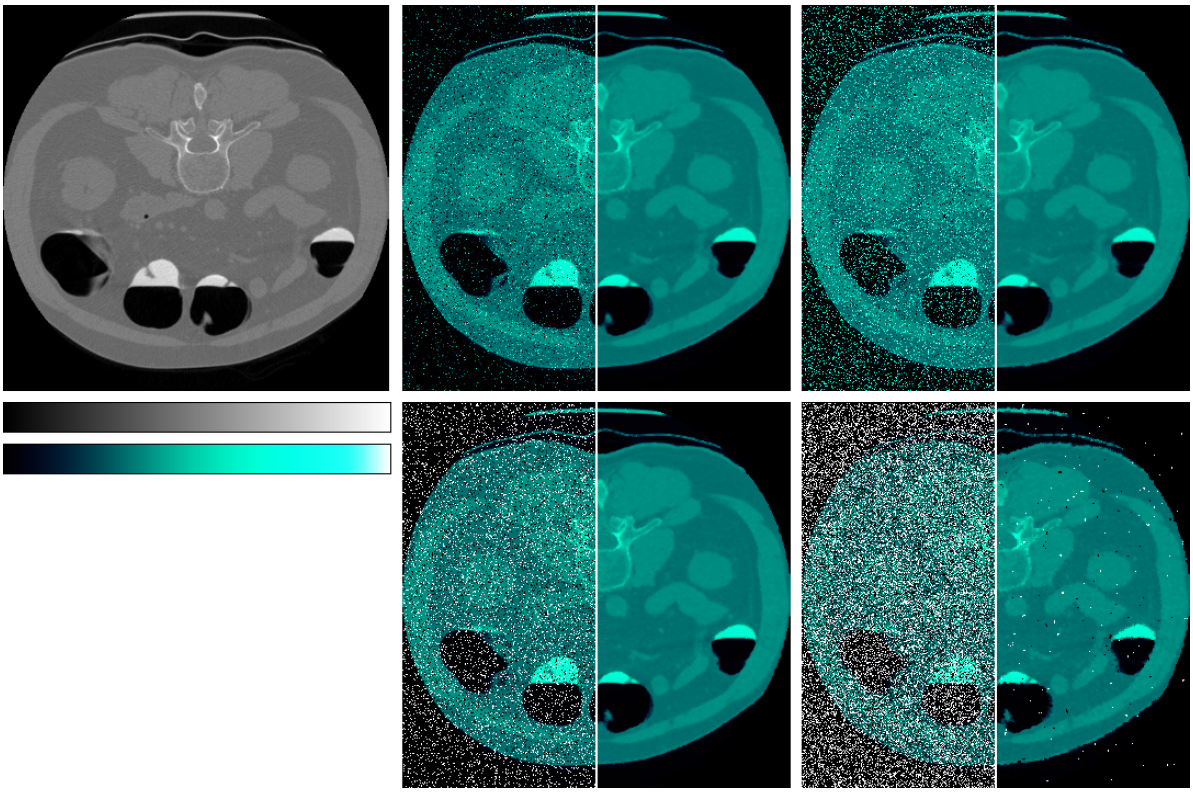


Abbildung 24: Medianfilter, Datensatz: Colon Supine, Größe: 352 × 352 × 352 (ca. 44 Mio. Voxel), Slice 204. *Oben links*: Originalbild ohne Transferfunktion, danach jeweils verrauschte und mediangefilterte Bildhälfte
Oben mitte: Additives gaußsches Rauschen (Standardabweichung 0,25), 10 Mio. Voxeldefekte, *oben rechts*: Weisses Rauschen, 10 Mio. Voxeldefekte, unten mitte: »Salt and pepper« Rauschen, 10 Mio. Voxeldefekte, unten rechts: »Salt and pepper« Rauschen, 30 Mio. Voxeldefekte

5.5 Bilateraler Filter

Der bilaterale Filter kann als eine Erweiterung des Gaußfilters mit einem zusätzlichen Kantenterm verstanden werden und wurde 1998 von Tomasi und Manduchi in [TM98] vorgestellt. Bei der Glättung mit einem Gaußfilter fließen die Werte der Umgebung des neu zu berechnenden Voxels mit einem von der Entfernung abhängigen Gewicht ein. Als Wichtungsfunktion dient dabei eine Normalverteilung mit Varianz σ. Liegt ein zu glättender Punkt in der Nähe einer Kante, fließen die räumlich relativ nah liegenden Voxelwerte auf der anderen Seite der Kante entsprechend in die Berechnung ein und die Kante verläuft (blurring).
Der bilaterale Filter ergänzt die Wichtungsfunktion um einen weiteren Term, der nicht nur die räumliche Distanz, sondern auch die Intensitätsdifferenz einfließen lässt. Der Ausdruck »bilateral« steht also für eine Ausdifferenzierung des Entfernungsmaßes in eine räumliche (»spatial domain«) und intensitätsmäßige (»intensity domain«) Komponente.

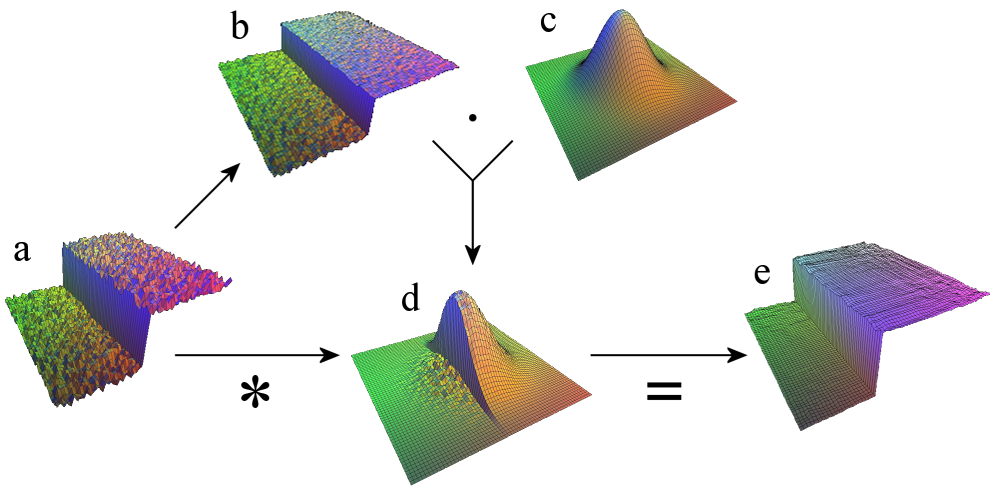


Abbildung 25: Aufbau des bilateralen Filterkerns. Nach [SPD07] und [DD02].
Das Originalbild (a) wird mit dem bilateralen Filterkern (d) zu (e) geglättet. Der bilaterale Filterkern besteht aus zwei Anteilen: Der Entfernung in der Intensitätsdomäne (b) und der räumlichen Entfernung (c).

Die grafische Interpretation zeigt, wie die kombinierte Gewichtung in der Raum- und Intensitätsdomäne einen adaptiven Filterkern ergibt, der trotz seiner kantenerhaltenden Eigenschaften das Rauschen gut glättet.

Die diskretisierte Formulierung (vgl. (5.5) und (5.7)) lautet:

$$BI[\Phi]_y = \frac{1}{W_y} \sum_{x \in \Omega} \varphi_{\sigma_x,\sigma_y,\sigma_z}(||y-x||)\varphi_{\sigma_i}(|\Phi_y - \Phi_x|)\Phi_x \tag{5.8}$$

Hier bezeichnet $\varphi_{\sigma_x,\sigma_y,\sigma_z}$ eine dreidimensionale Gaußfunktion mit den Varianzen σ_x, σ_y, σ_z und φ_{σ_i} eine eindimensionale Gaußfunktion mit Varianz σ_i. Der Faktor $\frac{1}{W_y}$ dient als Gewicht, das die Summe wieder normalisiert und kann über die Akkumulation der ausgewerteten Gaußfunktionen erhalten werden.

Abbildung 26 zeigt an konkreten Daten visuell die Zusammensetzung des bilateralen Filterkerns aus räumlichem und intensitätsmäßigem Filterkern auf. Aus dieser Darstellung

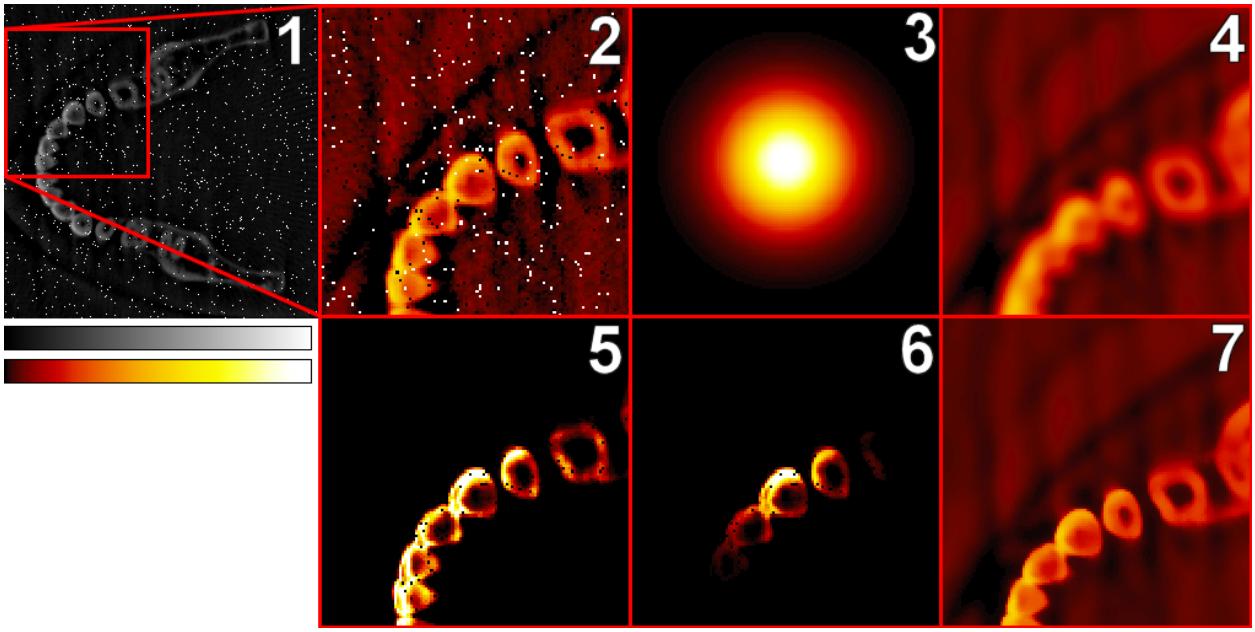


Abbildung 26: Visualisierung des bilateralen Filters, Datensatz: Skull, Größe: $256 \times 256 \times 256$

1. Verrauschtes Originalbild
2. Vergrößerter und mit Transferfunktion dargestellter Ausschnitt aus (1)
3. Visualisierter räumlicher Filterkern (spatial domain)
4. Faltung des Originalbildes (2) mit räumlichem Filterkern (3)
5. Visualisierter intensitätsmäßiger Filterkern (intensity domain)
6. Visualisierter bilateralter Filterkern: Produkt aus (3) und (5)
7. Faltung des Originalbildes (2) mit bilateralem Filterkern (6)

wird auch deutlich, dass der bilaterale Filter nicht separierbar ist. Der Filterkern muss immer aufgrund der veränderten Intensität im betrachteten Voxel neu berechnet werden. Auf der anderen Seite können bei sorgfältiger Wahl des zusätzlichen Parameters bereits kleinere Filterkerne zu hervorragenden Ergebnissen führen, da Rauschen effektiv ausgeblendet werden kann. Der räumliche Filterkern muss stets größer als die zu glättenden Strukturen gewählt werden, der intensitätsmäßige Filterkern sollte Werteschwankungen im Bereich des Rauschens effektiv unterdrücken.

Bei zu kleinem räumlichen Filter findet keine Glättung statt. Wird der räumliche Bereich vergrößert und der Intensitätsbereich klein gewählt, werden nur sehr ähnliche Voxel zur Glättung herangezogen - ein dünn besetzter Filterkern mit sehr hoher Kantenerhaltung aber auch geringer Glättungseigenschaften ist die Folge. Wird sowohl der räumliche als auch der intensitätsmäßige Bereich groß gewählt, ähnelt der bilaterale Filter dem Gaußfilter. Für $\sigma_i \to \infty$ erhalten wir $\varphi_{\sigma_i} \approx 1$, der Gaußfilter kann also als Spezialfall des bilateralen Filters aufgefasst werden. Abbildung 27 verdeutlicht diesen Zusammenhang.

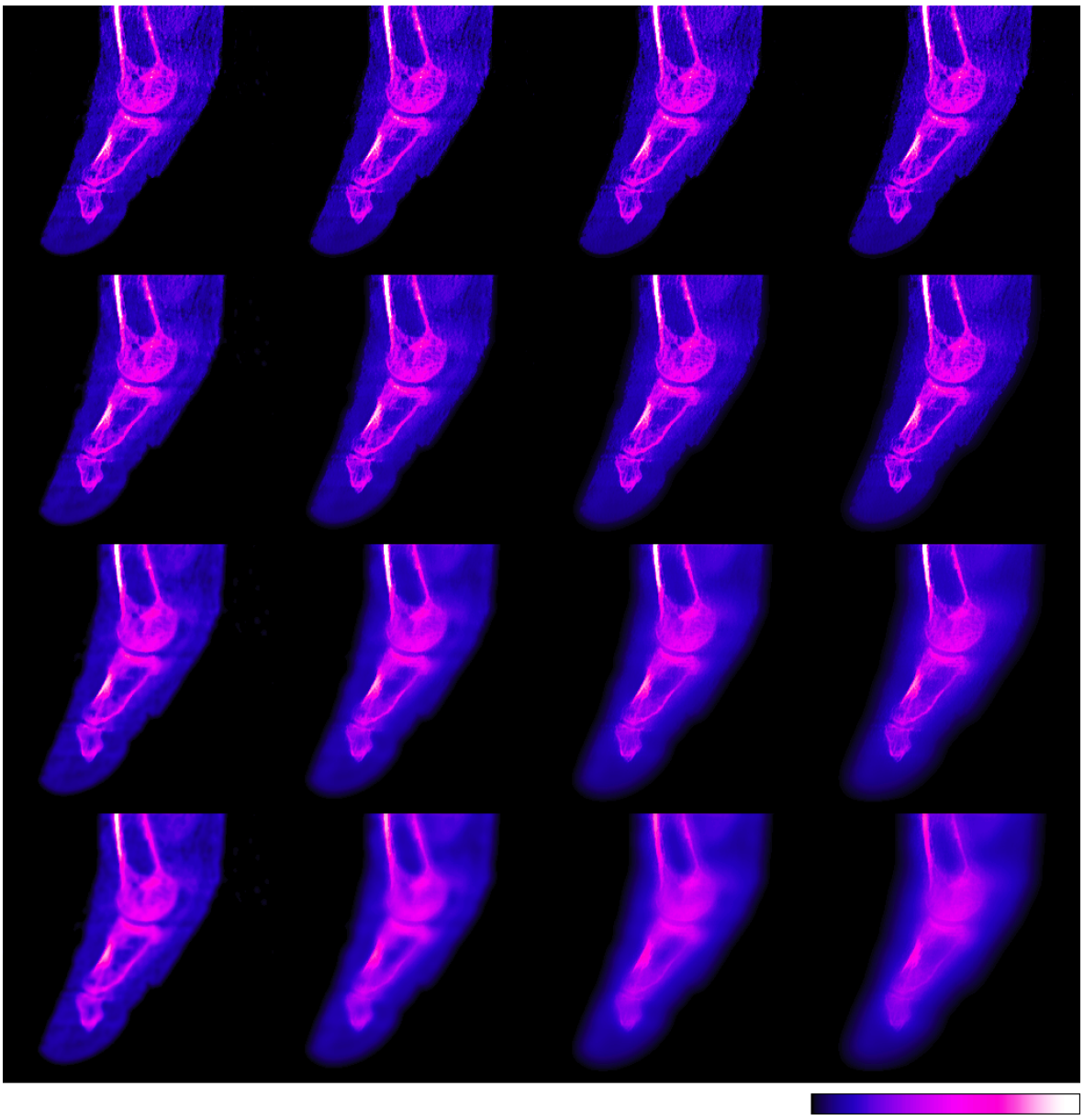


Abbildung 27: Bilateraler Filter, Datensatz: Foot, Größe: $256 \times 256 \times 256$, Slice 46
Wertebereich: 0 bis 255
Zeile (von links): $\sigma_s = 2, \sigma_r = 4, \sigma_r = 6, \sigma_r = 8$ (Spatial domain)
Spalte (von oben): $\sigma_i = 10, \sigma_i = 20, \sigma_i = 50, \sigma_i = 100$ (Intensity domain)

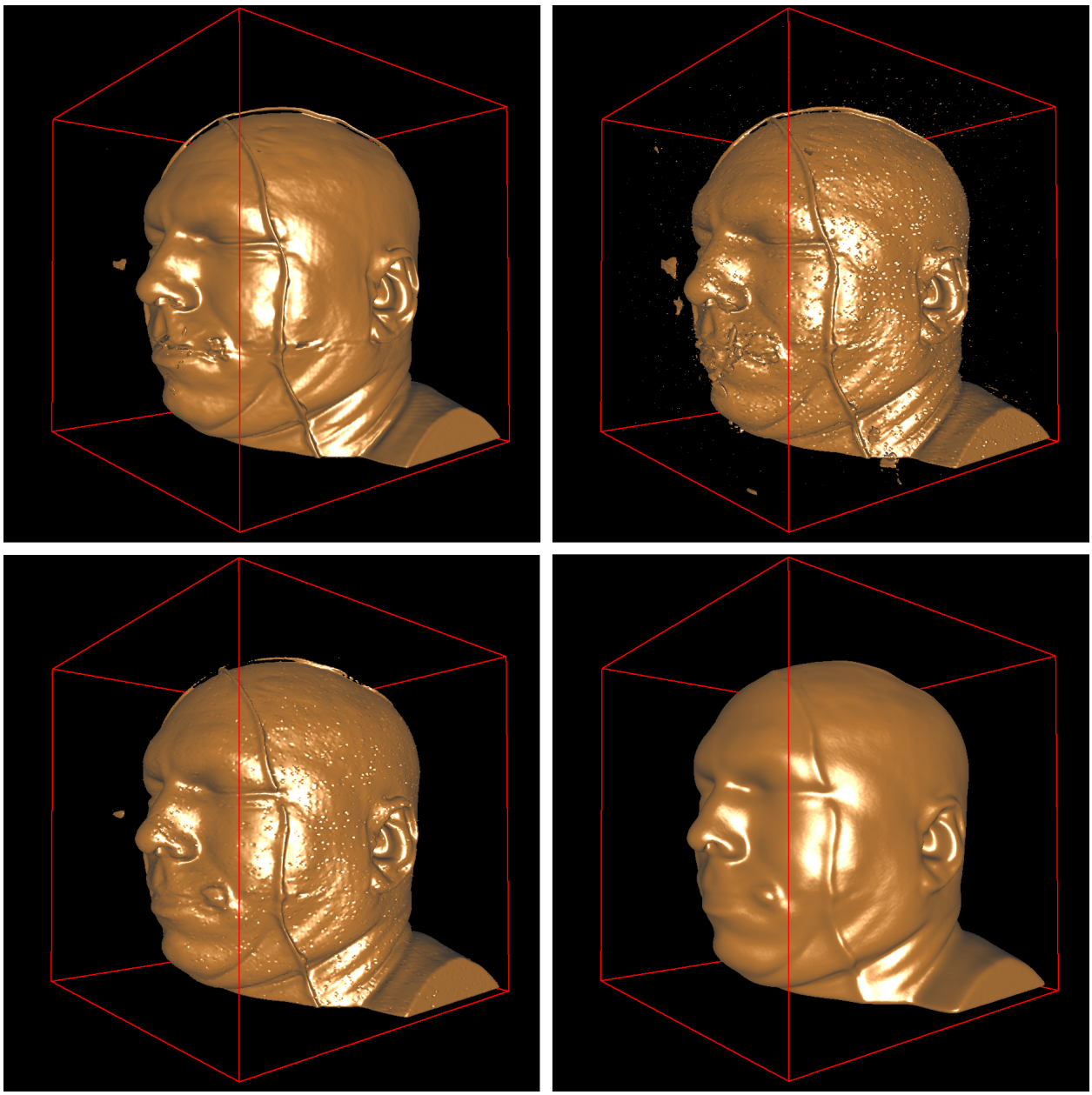


Abbildung 28: Edge enhancing diffusion mit bilateralem Prefiltering.
Oben links: Original Datensatz
Oben rechts: Verrauschtes Originalbild (Rauschen: Ca. 5% betroffene Voxel, Additives Gaußsches Rauschen, Standardabweichung 10%)
Unten links: Ergebnis des bilateralen Prefilterings
Unten rechts: Nach 50 Iterationen des EED-Filter ($\Delta t = 0.05$)

6 Affine Transformationen

Die in der Bildverarbeitung genutzten Filter sind Operatoren, die mit Hilfe mathematischer Abbildungen ein Quellbild in ein Zielbild überführen. Während die Ergebnisse von Filtern im Allgemeinen nicht umkehrbar sind, spricht man von Transformationen, wenn die mathematische Abbildung strukturerhaltend ist. Im engeren Sinne sind im Kontext der Bildverarbeitung mit »Transformationen« meist »affine[11] Transformationen«, insbesondere die Translation, Skalierung und Rotation gemeint.
Am Beispiel einer tatsächlich eingesetzten, einfachen affinen Volumentransformation, soll die Leistungsfähigkeit eines straight-forward implementierten GPU Kernels im Vergleich zu einer CPU Lösung ermittelt werden. Ziel dieses Exkurses ist es zu klären, ob CUDA Implementierungen im normalen Entwicklungsalltag Geschwindigkeitsvorteile bringen. Mit Hilfe des entwickelten Prototyps können die Ergebnisse schnell miteinander vergleichbar gemacht werden.

Es folgt eine kurze Einführung in homogene Koordinaten[12], dann wird eine einfache zusammengesetzte affine Transformation vorgestellt, wie sie im Rahmen einer Bestrahlungsplanungssoftware des GSI Helmholtzzentrums für Schwerionenforschung tatsächlich zum Einsatz kommt, und die Performanceergebnisse diskutiert. Dieses Kapitel wurde direkt durch die Mitarbeit des Diplomanden an genannter Bestrahlungsplanungssoftware motiviert.

6.1 Homogene Koordinaten

Mit Hilfe homogener Koordinaten lassen sich alle affinen Transformationen, insbesondere Translationen und Rotationen, einheitlich in Matrizenschreibweise darstellen und handhaben. Homogene Koordinaten haben sich mittlwerweile als Standardwerkzeug der 3D Computergraphik etabliert, DirectX und OpenGL arbeiten ebenso mit homogenen Koordinaten, wie unzählige Softwarepakete aus Industrie und Medizin. Eine Einführung in das Thema »Homogene Koordinaten« findet man in vielen Lehrbüchern, die sich mit den mathematischen Grundlagen der Computergraphik beschäftigen (z.B. [Len01, S. 62 ff.] oder [Mor99, S. 69 ff.]). Im Folgenden ist es vollkommen ausreichend zu wissen, dass alle affinen Transformationen mit Hilfe von 4×4 Matrizen ausgedrückt werden können

[11] Affine Abbildungen erweitern das Konzept der linearen Abbildung um eine Parallelverschiebung bzw. »Translation« $\vec{d} \neq 0$. In Koordinatendarstellung lautet die affine Abbildung allgemein: $f(\vec{x}) = A \cdot \vec{x} + \vec{d}$.

[12] In [BM01] werden homogene Koordinaten an 2D Beispielen eingeführt, [BB05] liefert sehr gute Erklärungen im 3 dimensionalen Raum. Auch das bekannte »OpenGL Redbook« [NDW97] liefert dem interessierten Leser im Appendix F gute Ausführungen zu dieser Thematik.

und ein Punkt $P = (x, y, z)^T$ in homogener Darstellung als $P = (x, y, z, 1)^T$ notiert wird. Die Translation und anschließende Rotation eines Punktes P hätte beispielsweise folgende Gestalt:

$$P^* = R \cdot T \cdot P$$

Wobei $P^*, P \in \mathbb{R}^4$ den transformierten Punkt bzw. den Ursprungspunkt und $R, T \in \mathbb{R}^{4\times4}$ die Rotations- bzw. die Translationsmatrix bezeichnet. Werden mehrere Punkte P_i auf gleiche Weise transformiert, macht man sich die Assoziativität der Matrizenmultiplikation zu nutze: Man fasst zunächst alle 4×4 Matrizen A_i zu einer Gesamttransformation T_Ω zusammen und multipliziert diese erst anschließend mit den zu transformierenden Vektoren:

$$\begin{aligned} P^* &= (A_n \dots (A_2 \cdot (A_1 \cdot P))) \\ &= \underbrace{(A_n \dots A_2 \cdot A_1)}_{T_\Omega} \cdot P \end{aligned} \tag{6.1}$$

Für n Punkte und m Transformationen sind bei dieser Vorgehensweise lediglich $n+(m-1)$ statt $n * m$ Matrizenmultiplikationen durchzuführen.

6.2 Fallbeispiel

Die einfachste zusammengesetzte affine Transformation, die in der Bestrahlungsplanungssoftware *TRiP* (s. [Krä09]) des GSI Helmholtzzentrums für Schwerionenforschung in Darmstadt zum Einsatz kommt, besteht aus einer initialen Translation T_{pre}, einer Rotation R_ω, sowie einer abschließenden Translation T_{post}. Die Rotation R_ω ist zusammengesetzt aus drei einfachen Rotationen um die X, Y und Z Achse. In Matrizenschreibweise lautet die Transformation $T_\Omega : \mathbb{R}^4 \mapsto \mathbb{R}^4, P \mapsto P^*$ für einen Punkt P:

$$P^* = T_\Omega \cdot P \tag{6.2}$$

$$T_\Omega = T_{post} \cdot \underbrace{R_Z \cdot R_Y \cdot R_X}_{R_\omega} \cdot T_{pre} \tag{6.3}$$

Die beiden Translationen erfordern jeweils drei Richtungsangaben, die einzelnen Rotationen benötigen jeweils einen Winkel. Die gesamte Transformation hat also neun freie Parameter, die in Form zweier Translationsvektoren und der Rotationswinkel gegeben werden:

$$\vec{d_1} = \begin{pmatrix} t_{X1} \\ t_{Y1} \\ t_{Z1} \end{pmatrix}, \quad \vec{d_2} = \begin{pmatrix} t_{X2} \\ t_{Y2} \\ t_{Z2} \end{pmatrix}, \quad \alpha, \quad \beta, \quad \gamma$$

Die vollständige, parametrisierte Darstellung der affinen Transformation gemäß (6.2) und 6.3 lautet damit:

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \underbrace{T_{post}(\vec{d_2}) \cdot R_Z(\gamma) \cdot R_Y(\beta) \cdot R_X(\alpha) \cdot T_{pre}(\vec{d_1})}_{T_\Omega} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{6.4}$$

6.3 Geschlossene Formel für affine Transformation

Wie man in Formel (6.4) sieht, kann die Transformation T_Ω komplett aus den 5 einzelnen homogenen Matrizen zusammengefasst und im Voraus berechnet werden. Im Anhang B (S. XXVII) ist die komplette Herleitung und Umformung der Transformationsmatrix T_Ω beschrieben. Wir haben damit T_Ω in eine geschlossene Form überführt, mit der die x, y und z Werte eines Eingangspunktes direkt in das neue Koordinatensystem überführt werden können.
Da die Darstellung der Transformation T_Ω in einer einzelnen Matrix ohne Abkürzungen relativ schreibaufwändig ist, verwenden wir im Folgenden die Definitionen des Anhangs (s. Anhang B, »Herleitung der affinen Transformationsmatrix«):

$$T_\Omega = \begin{pmatrix} r_1 & r_2 & r_3 & r_4 \\ r_5 & r_6 & r_7 & r_8 \\ r_9 & r_{10} & r_{11} & r_{12} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die einzelnen Matrixeinträge r_i sind dabei mehr oder weniger kompliziert zusammengesetzte Produktsummen, deren Faktoren überwiegend aus trigonometrischen Funktionen bestehen. Der Aufwand zur Berechnung der Matrix kann annähernd vernachlässigt werden, da die Vorberechnung der Einträge r_i nur einmalig für alle zu transformierenden Voxelkoordinaten eines Datensatzes durchzuführen ist. Für jeden Punkt (x, y, z) wird dann die Transformation für die einzelne Koordinate wie folgt ausgeführt:

$$x^* = r_1 \cdot x + r_2 \cdot y + r_3 \cdot z + r_4 \tag{6.5}$$
$$y^* = r_5 \cdot x + r_6 \cdot y + r_7 \cdot z + r_8 \tag{6.6}$$
$$z^* = r_9 \cdot x + r_{10} \cdot y + r_{11} \cdot z + r_{12} \tag{6.7}$$

6.4 Implementierung

Da man durch einfache Vorwärtstransformation der Voxelkoordinaten des Ursprungsvolumens nicht alle Voxel des Ausgabevolumens treffen würde (man denke zum Beispiel an die Skalierung mit dem Faktor 2, die nur jeden zweiten Ausgabevoxel treffen würde), geht man den umgekehrten Weg: Man definiert die Inverse Transformation T_{Ω}^{-1} indem man zur Definition einfach die negativen Winkel und die negativen Translationsvektoren heranzieht. Nun iteriert man durch alle Voxel des zu befüllenden Ausgabedatensatzes, transformiert die Voxelkoordinaten in das Ursprungskoordinatensystem und ermittelt so alle benötigten Werte.

In jedem Fall liegen die transformierten Voxelmittelpunktkoordinaten bzw. die Gitterpunkte nicht exakt auf dem ursprünglichen Gitter. Im diskreten Fall kann also der Wert am transformierten Punkt (x^*, y^*, z^*) nicht exakt ermittelt werden. Vier bekannte Verfahren sind der Zugriff auf den nächst kleineren diskret vorhandenen Gitterwert (»Floor Values«), auf den räumlich nächstgelegenen Gitterwert (»Nearest Neighbor«), die lineare Interpolation in alle Dimensionen (hier »Trilineares Filtering«), oder die Interpolation mit Splines (hier z.B. »trikubische B-Splines«).

Die Transformation eines einzelnen Punktes erfordert gemäß der Formeln (6.5), (6.6), (6.7) insgesamt 9 Multiplikationen und 9 Additionen. Hinzu kommen die Berechnungen für den indizierten Zugriff auf die diskret vorliegenden Voxelwerte, sowie der eigentliche Speichertransfer:

Im Falle des Nearest Neighbor Verfahrens werden lediglich drei Rundungen auf Ganzzahlen sowie ein Speicherzugriff benötigt. Im Falle trikubischer B-Splines[13], die pro Dimension vier Stützstellen benötigen, fallen neben der Berechnung der Basispolynome zusätzlich noch weitere $4^3 = 64$ Speicherzugriffe an. Je nachdem, ob die Auswertung der Basispolynome über sogenannte vorberechnete Lookup-Tables (LUTs) realisiert wird oder die Basispolynome exakt ausgewertet werden, werden hier 64 weitere kleinere Indexberechnungen oder rund 1600 arithmetische Operationen benötigt.

[13] Die Mathematik zur Berechnung trikubischer B-Splines ist nicht Bestandteil der Ausführungen dieser Arbeit. Der interessierte Leser sei auf [Uns99] als Einstieg in die Glättung mit Splines verwiesen, bzw. an [JLR03], [Pol93] und [RtHRS08] für die Glättung mit B-Splines im Speziellen. Diese Quellen liefern hinreichende Erläuterungen zum Tensorprodukt der volumetrischen B-Spline Interpolation und geben die Formeln für die interpolierenden Basissplines an. In [Har03, S. 53] findet sich eine straffe Erklärung, die direkt zur Implementierung der entsprechenden Algorithmen des Prototyps herangezogen wurde.

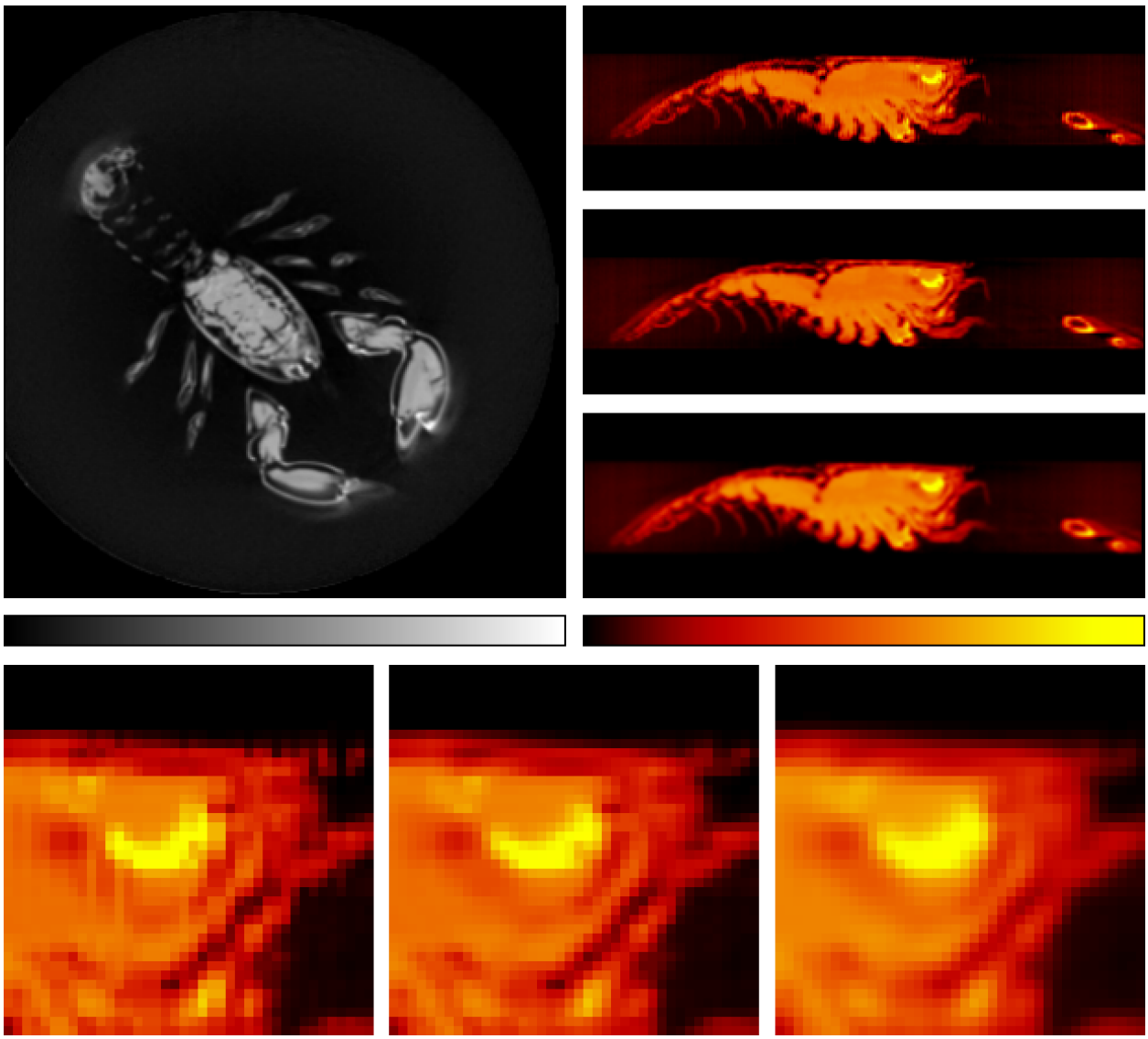


Abbildung 29: Affine Transformation des Lobster Datensatzes: 45° Rotation um z-Achse, Translation in x-Richtung um 10 Voxel, −90° Grad Rotation um neue x-Achse
Oben links: Original Datensatz (Slice 42), *Oben rechts*: Transformierter Datensatz (Slice 42). *Von oben*: Nearest Neighbor, trilineare Filterung, trikubische B-Splines. Unten: Ausschnitt bei 5 facher Vergrößerung. *Von links*: Nearest Neighbor, trilineare Filterung, trikubischer BSpline.

Bei der affinen Transformation eines Volumendatensatz mit 256^3 Voxeln unter Zuhilfenahme trikubischer B-Splines sind das bereits rund 30 Milliarden arithmetische Operationen und rund 1 Milliarde Speicherzugriffe bzw. unter Nutzung dreier LUTs für x, y, z Richtung und Wertezugriff fallen rund 10 Milliarden arithmetische Operationen und 4,3 Milliarden Speicherzugriffe ($256^3 \cdot 64 \cdot 4$) an.
Diese Überlegungen führten letztlich dazu, die affine Transformationen T_Ω in allen drei Varianten (Nearest Neighbor, trilineare Interpolation und trikubische B-Splines) sowohl als CPU Variante, als auch als GPU Variante zu implementieren und die Ergebnisse zu vergleichen.

Die Messergebnisse werden im Kapitel »Ergebnisse« aufgeführt. Die CPU Variante wurde durch den Einsatz von LUTs optimiert. Die GPU Variante arbeitet hingegen mit der expliziten Berechnung der Basissplines effizienter, was auf die relativ großen Latenzen der Speicherzugriffe zurückzuführen ist.

7 Implementierung

Im Folgenden Kapitel werden alle relevanten Aspekte der konkreten Umsetzung erörtert. Die bisherigen mathematischen Grundlagen und die Überlegungen zu den einzelnen Filtern sind als kontinuierliche Gleichungssysteme formuliert und müssen zunächst dikretisiert werden. Für die Erstellung eines Prototyps, der die besprochenen Filter auf der Nvidia CUDA Architektur umsetzen soll, sind dabei verschiedene Überlegungen notwendig. Es handelt sich dabei sowohl um allgemeine Problemstellungen, als auch um CUDA spezifische Gegebenheiten.
Die aktuelle Entwicklung lässt vermuten, dass sich CUDA als feste GPGPU Architektur längst etabliert hat, und so ist es in naher Zukunft möglich, die wesentlichen Codefragmente und Ergebnisse in kommende CUDA Programme einfließen zu lassen. Auch unter der Prämisse, dass Nvidias CUDA Architektur vielleicht in einiger Zeit vom Markt überholt wird, sind die wesentlichen Überlegungen dieses Kapitels allgemein auf parallele Implementierungen in anderen Architekturen übertragbar.

7.1 Diskretisierung

Die kommenden Abschnitte befassen sich mit der Diskretisierung der partiellen Differentialgleichungssysteme. Die gesamte Theorie der vorangegangenen Kapitel, insbesondere die Gleichungssysteme der Diffusionsfilter, liegen bisher lediglich in einer kontinuierlichen Formulierung vor. Um eine numerische Umsetzung zu ermöglichen, müssen die kontinuierlichen Modelle nun räumlich und zeitlich diskretisiert werden.

Die räumliche Diskretisierung nutzt als natürliche Zerlegung die Gitternetzstruktur (das Grid) des unterliegenden Datensatzes. In der Praxis trifft man auf sehr unterschiedliche Datensatzgrößen. Gängige medizinische Volumendatensätze bestehen heutzutage in der Regel aus 64 bis 256 Bildscheiben mit Auflösungen von 64×64 bis 512×512 Pixeln. Die räumlichen Abtastraten sind zumeist in x und y Richtung (innerhalb einer Bildscheibe) gleichgroß, der Abstand zwischen den einzelnen Schichten ist jedoch oft in einer anderen Größe bemessen oder gar nicht äquidistant gewählt. Der Wertebereich der Messung lässt sich bei CT Scans (Hounsfield Skala) mit 12 Bit abbilden, wird dann in der Regel aber aufgrund der gängigen Computer-Integergrößen mit 16 bzw. 32 Bits gespeichert. Allgemein kann angenommen werden, dass ein typischer Datensatz aus rund $256 \times 256 \times 256 \approx 16,7$ Millionen Voxeln besteht und damit 32 bzw. 64 MB Speicher belegt.

7.1.1 Lineares homogenes Modell

Im Folgenden sei mit $\widehat{\Phi}$ die diskretisiert vorliegende Form der Bildfunktion $\Phi : \Omega \subset \mathbb{R}^3 \mapsto \mathbb{R}$ bezeichnet. Die Diskretisierung des linearen homogenen Diffusionsfilters nutzt die Gitternetzstruktur des Datensatzes als natürliche räumliche Zerlegung aus. Jeder Gitterpunkt ist assoziiert mit einem Wert $\widehat{\Phi}_{x,y,z}$ und steht für eine diskrete Abtastung der eigentlichen Bildfunktion Φ für den Mittelpunkt des Voxels mit Index x, y, z. Genaugenommen bildet bei medizinischen Datensätzen der Wert $\widehat{\Phi}_{x,y,z}$ die mittlere Dichte des jeweiligen Voxels ab.
Wir einigen uns nun auf eine verkürzte Schreibweise: $\widehat{\Phi}$ bezeichnet, soweit keine Parameter explizit angegeben sind, immer den Wert des Voxels $\widehat{\Phi}_{x,y,z}$. Für das Indextripel gilt dabei zunächst:

$$(x, y, z) \in [0 \ldots \mathrm{DimX} - 1] \times [0 \ldots \mathrm{DimY} - 1] \times [0 \ldots \mathrm{DimZ} - 1] \subset \mathbb{N}^3$$

DimX, DimY und DimZ bezeichnen die Größe des Volumendatensatzes. Die Voxel sind also beginnend bei 0 in alle Raumrichtungen indiziert. Weiterhin bezeichnen wir die zu $\widehat{\Phi} = \widehat{\Phi}_{x,y,z}$ in x Richtung benachbarten Voxel mit $\widehat{\Phi}_{x_-} = \widehat{\Phi}_{x-1,y,z}$ und $\widehat{\Phi}_{x_+} = \widehat{\Phi}_{x+1,y,z}$ bzw. sinngemäß die in y und z Richtung benachbarten Voxel mit $\widehat{\Phi}_{y_-}, \widehat{\Phi}_{y_+}, \widehat{\Phi}_{z_-}$ und $\widehat{\Phi}_{z_+}$

Wir betrachten nun die bereits bekannte Differentialgleichung der linearen homogenen Diffusion in ihrer kontinuierlichen Formulierung (4.3). Es gilt $D \in \mathbb{R}$ und wir formen die Gleichung wie folgt um:

$$\begin{aligned} \frac{\partial}{\partial t}\Phi(\vec{x}, t) &= D\Delta\Phi(\vec{x}, t) = D\nabla(\nabla(\Phi(\vec{x}, t))) = D \operatorname{div}\left(\frac{\partial\Phi}{\partial x}, \frac{\partial\Phi}{\partial y}, \frac{\partial\Phi}{\partial z}\right)^T \\ &= D\left(\frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} + \frac{\partial^2\Phi}{\partial z^2}\right) \end{aligned} \tag{7.1}$$

Zunächst diskretisieren wir die Lösung zeitlich durch eine lineare Approximation des Gesamtflusses über das Zeitinterval $\Delta t = \tau$. Grundlage ist also die Zeitdiskretisierung über Vorwärtsdifferenzen:

$$\frac{\partial}{\partial t}\Phi(\vec{x}, t) \approx \frac{\Phi(\vec{x}, t + \tau) - \Phi(\vec{x}, t)}{\tau} \tag{7.2}$$

Wir haben nun zwei Beschreibungen für die zeitliche Entwicklung der Bildfunktion Φ und erhalten durch Umformen von (7.1) nach $\Phi(\vec{x}, t + \tau)$ und Einsetzen von (7.2):

$$\begin{aligned}\Phi(\vec{x}, t+\tau) &\approx \Phi(\vec{x}, t) + \tau \cdot \frac{\partial}{\partial t}\Phi(\vec{x}, t) \\ &= \Phi(\vec{x}, t) + \tau \cdot D\left(\frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} + \frac{\partial^2\Phi}{\partial z^2}\right) \end{aligned} \tag{7.3}$$

Man sieht sofort, dass der physikalisch motivierte Diffusionskoeffizient lediglich linear in die Zeitspanne eingeht und wir können o. B. d. A im Folgenden $D = 1$ annehmen. Als nächstes erkennt man, dass die zeitliche Zu- oder Abnahme schlussendlich von der Summe der zweiten Ableitung in alle Raumrichtungen abhängt. Das ist - wenn wir den Sachverhalt wieder im physikalischen Kontext betrachten - einleuchtend: Ein Gradient alleine induziert zwar einen Fluss, ruft aber noch keine Änderung der Dichte hervor: Ist beispielsweise die zweite Ableitung Null, ändern sich die Gradienten nicht und der Zu- und Abfluss heben sich auf. Ein diskretisiertes eindimensionales Beispiel wären 3 benachbarte Elemente eines dünnen Drahtstückes, die 8°, 10° und 12° Temperatur haben. Das mittlere Element wird sich nicht abkühlen, obwohl ein linksseitiger Gradient von -2° besteht, denn der rechtsseitige Gradient von $+2°$ gleicht den einseitigen Temperaturverlust wieder aus.

Formel (7.3) ist nun die zeitlich diskretisierte Wärmeleitungsgleichung. Sie kann mit Hilfe der eingeführten Schreibkonventionen nun räumlich diskretisiert werden. Die vollständig zeitlich und räumlich diskretisierte Formulierung lautet dann:

$$\begin{aligned}\widehat{\Phi}_{x,y,z}(t+\tau) &\approx \widehat{\Phi}_{x,y,z}(t)+\tau\cdot\left(\left(\frac{\frac{\widehat{\Phi}_{x_+}-\widehat{\Phi}}{\Delta x}-\frac{\widehat{\Phi}-\widehat{\Phi}_{x_-}}{\Delta x}}{\Delta x}\right)+\left(\frac{\frac{\widehat{\Phi}_{y_+}-\widehat{\Phi}}{\Delta y}-\frac{\widehat{\Phi}-\widehat{\Phi}_{y_-}}{\Delta y}}{\Delta y}\right)+\left(\frac{\frac{\widehat{\Phi}_{z_+}-\widehat{\Phi}}{\Delta z}-\frac{\widehat{\Phi}-\widehat{\Phi}_{z_-}}{\Delta z}}{\Delta z}\right)\right) \\ &= \widehat{\Phi}_{x,y,z}(t)+\tau\cdot\left(\frac{\widehat{\Phi}_{x_+}-\widehat{\Phi}}{(\Delta x)^2}+\frac{\widehat{\Phi}_{x_-}-\widehat{\Phi}}{(\Delta x)^2}+\frac{\widehat{\Phi}_{y_+}-\widehat{\Phi}}{(\Delta y)^2}+\frac{\widehat{\Phi}_{y_-}-\widehat{\Phi}}{(\Delta y)^2}+\frac{\widehat{\Phi}_{z_+}-\widehat{\Phi}}{(\Delta z)^2}+\frac{\widehat{\Phi}_{z_-}-\widehat{\Phi}}{(\Delta z)^2}\right)\end{aligned} \tag{7.4}$$

Hierbei bezeichnen Δx, Δy und Δz die räumliche Breite der Voxel in x, y und z Richtung. Setzen wir nun voraus, dass die Abstände nicht nur äquidistant sind, sondern auch $\Delta x = \Delta y = \Delta z = 1$ gilt, so können wir für die lineare homogene Diffusion aus (4.3) ein einfaches, vollständig diskretisiertes Iterationsschema über die 6 - Nachbarschaft des Voxels x, y, z angeben:

$$\begin{aligned}\widehat{\Phi}_{x,y,z}(t+\tau) &\approx \widehat{\Phi}_{x,y,z}(t)+\tau\cdot\left(\widehat{\Phi}_{x_+}-\widehat{\Phi}+\widehat{\Phi}_{x_-}-\widehat{\Phi}+\widehat{\Phi}_{y_+}-\widehat{\Phi}+\widehat{\Phi}_{y_-}-\widehat{\Phi}+\widehat{\Phi}_{z_+}-\widehat{\Phi}+\widehat{\Phi}_{z_-}-\widehat{\Phi}\right) \\ &= \widehat{\Phi}_{x,y,z}(t)+\tau\cdot\left(\widehat{\Phi}_{x_+}+\widehat{\Phi}_{x_-}+\widehat{\Phi}_{y_+}+\widehat{\Phi}_{y_-}+\widehat{\Phi}_{z_+}+\widehat{\Phi}_{z_-}-6\widehat{\Phi}\right)\end{aligned} \tag{7.5}$$

7.1.2 Nachbarschaft und Randbedingungen

Wir haben durch die Diskretisierung nun ein einfaches iteratives Verfahren erhalten, um das neue Skalarfeld $\widehat{\Phi}(t+\tau)$ aus dem alten Skalarfeld $\widehat{\Phi}(t)$ zu berechnen. Die geometrische Interpretation ist ebenfalls relativ einfach:
Bei jedem Zeitschritt wird der Wert eines jeden Voxels gemäß der ermittelten partiellen Flüsse aller Raumrichtungen angepasst - zentrales Element ist also der Beitrag $j_N = \tau \frac{\widehat{\Phi_N} - \Phi}{\Delta x}$ (»neighbor flow contribution«) den jeder benachbarte Voxel N einbringt. Grafisch sieht der Zusammenhang wie folgt aus:

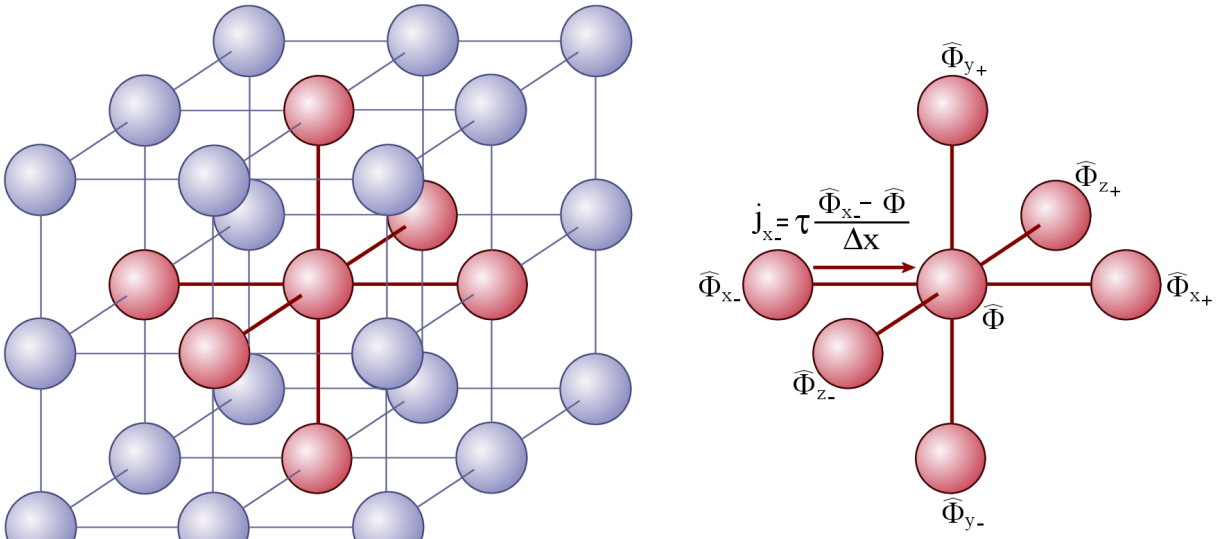


Abbildung 30: Darstellung des regulären Gitters und induzierter Fluss der 6-Nachbarschaft eines Gridvoxels

Man kann im 2 dimensionalen Fall anstatt der 4-Nachbarschaft auch die 8-Nachbarschaft bzw. im 3 dimensionalen Fall anstatt der 6-Nachbarschaft auch analog die 26-Nachbarschaft zur Berechnung der neuen Werte heranziehen (vgl. [MUF95, S. 30]). Die Beiträge der jeweiligen Voxel müssen dabei entsprechend der relativen Distanz gewichtet werden. Für die 8 Nachbarschaft im 2D Fall müsste der Fluss für die Eckpunkte $\widehat{\Phi}_{x_-y_-}, \widehat{\Phi}_{x_-y_+}, \widehat{\Phi}_{x_+y_-}$ und $\widehat{\Phi}_{x_+y_+}$ mit $1/\sqrt{\Delta X^2 + \Delta Y^2}$ bzw. im allgemeinen Fall eines anisotrop gewählten Gitters entsprechend der euklidischen Abstandsnorm gewichtet werden.

Im 3 Dimensionalen bildet die 26 Nachbarschaft einen Quader, dessen Kantenvoxel einen mit $1/\sqrt{\Delta X^2 + \Delta Y^2}$, $1/\sqrt{\Delta X^2 + \Delta Z^2}$ bzw. $1/\sqrt{\Delta Y^2 + \Delta Z^2}$ gewichteten Beitrag leisten und deren Eckvoxel mit $\sqrt{\Delta X^2 + \Delta Y^2 + \Delta Z^2}$ eingerechnet werden.

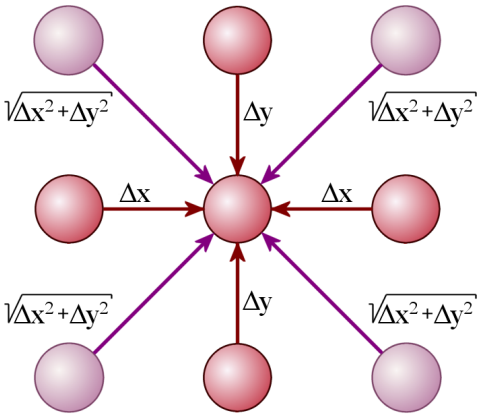


Abbildung 31: Fluss der 8-Nachbarschaft eines 2D Grids

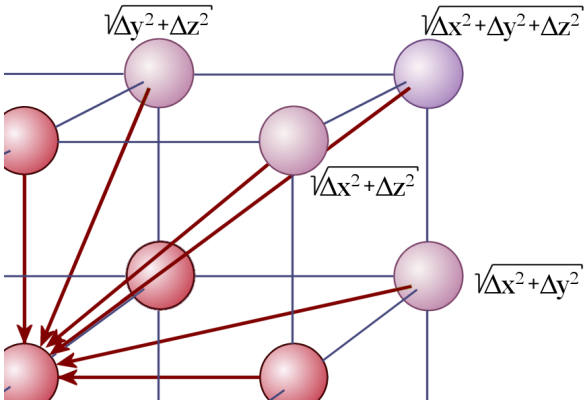


Abbildung 32: Fluss der 26-Nachbarschaft eines 3D Grids

Legen wir ein isotropes Gitter mit $\Delta x = \Delta y = \Delta z = 1$ zugrunde, ergibt sich für die lineare homogene Diffusion im 3D Fall unter Einbeziehung der 2-Nachbarschaft in Anlehnung an (7.5) die diskretisierte Formulierung wie folgt:

$$\begin{aligned}
\widehat{\Phi}_{x,y,z}(t+\tau) \approx\ & \widehat{\Phi}_{x,y,z}(t) + \tau \cdot \left(\widehat{\Phi}_{x_+} + \widehat{\Phi}_{x_-} + \widehat{\Phi}_{y_+} + \widehat{\Phi}_{y_-} + \widehat{\Phi}_{z_+} + \widehat{\Phi}_{z_-} - 6 \cdot \widehat{\Phi}\right) \\
+\ & \frac{\tau}{\sqrt{2}} \cdot \left(\widehat{\Phi}_{x_+y_+} + \widehat{\Phi}_{x_+y_-} + \widehat{\Phi}_{x_+z_+} + \widehat{\Phi}_{x_+z_-} + \widehat{\Phi}_{y_+z_+} + \widehat{\Phi}_{y_+z_-}\right. \\
+\ & \left. \widehat{\Phi}_{y_-z_+} + \widehat{\Phi}_{y_-z_-} + \widehat{\Phi}_{x_-y_+} + \widehat{\Phi}_{x_-y_-} + \widehat{\Phi}_{x_-z_+} + \widehat{\Phi}_{x_-z_-} - 12 \cdot \widehat{\Phi}\right) \\
+\ & \frac{\tau}{\sqrt{3}} \cdot \left(\widehat{\Phi}_{x-+y_+z_+} + \widehat{\Phi}_{x_+y_+z_-} + \widehat{\Phi}_{x_+y_-z_+} + \widehat{\Phi}_{x_+y_-z_-}\right. \\
+\ & \left. \widehat{\Phi}_{x_-y_+z_+} + \widehat{\Phi}_{x_-y_+z_-} + \widehat{\Phi}_{x_-y_-z_+} + \widehat{\Phi}_{x_-y_-z_-} - 8 \cdot \widehat{\Phi}\right) \qquad (7.6)
\end{aligned}$$

Die Implementierung eines solchen iterativen Verfahrens erfordert noch die Umsetzung der Anfangs- und Randbedingungen, vgl. (4.4) und (4.5). Die Umsetzung der Anfangsbedingung besteht lediglich in der Initialisierung des Datensatzes mit dem zu glättenden Urbild. Die Randbedingung $\frac{\partial}{\partial n}\Phi(\vec{x},0) = 0$ (4.5) besagt, dass kein Partialfluss durch die Ränder des Diffusionsvolumens stattfindet und liegt physikalisch in der Energieerhaltung des betrachteten Volumens begründet. Im Zuge der Bildverarbeitung bedeutet dies, dass keine globale Zu- oder Abnahme der Dichtewerte stattfindet, der mittlere Grauwert des Volumens also erhalten bleibt.

Diese Bedingung lässt sich technisch auf zwei Arten umsetzen: Im Randbereich des Volumens wird analog der obigen diskretisierten Formeln nicht auf die gesamte Nachbarschaft des betrachteten Voxels zurückgegriffen und nur die Partialflüsse der im Inneren des Volumens vorhandenen Nachbarvoxel werden aufsummiert. Andererseits kann man – die

treibende Kraft des Diffusionsflusses ist ja der vorhandene Gradient – Zugriffe auf Voxel ausserhalb des definierten Gebietes auf den nächst möglichen Voxel in Richtung des betrachteten Volumens zurücksetzen (clamp values). Auf diese Weise kann in Normalenrichtung des betrachteten Gebiets kein Gradient wirken und es wird kein Ausgleichsfluss erzeugt.

7.1.3 Inhomogenes isotropes Modell

Der inhomogene isotrope Diffusionsfilter koppelt den induzierten Fluss noch an einen Koeffizienten, der je nach Struktur des Bildes lokal den Ausgleich hemmt oder zulässt, so dass Kanten im Bild nicht so schnell verwaschen. Zum Einsatz kommt dabei eine monoton fallende, reelle Funktion g: die Diffusivitätsfunktion. Als einzigen Parameter erhält sie ein Wahrscheinlichkeitsmaß für eine lokal vorliegende Kante und liefert dann einen Skalar zurück, der als »Fluss–hemmender Faktor« verstanden werden kann. Im Normalfall dient als Eingabe einfach die Länge des lokalen Gradienten (»gradient as fuzzy Edgedetektor«, vgl. [Wei97]).
Basiert die Kanteninformation auf dem Urbild Φ_0 oder einer im voraus geglätteten Version $F_{pre}(\Phi_0)$, so handelt es sich um einen linearen Diffusionsvorgang. Findet eine Rückkopplung statt, indem der Diffusionsprozess im jeweiligen Iterationsschritt die Diffusivität aufgrund des aktuellen Bildes Φ ermittelt, spricht man von nichtlinearer inhomogener isotroper Diffusion.

Ausgehend von (4.9) soll im Folgenden eine vollständig diskretisierte Formulierung des nichtlinearen Diffusionsfilters angegeben werden.

$$\partial_t \Phi = \operatorname{div}(\, g_\lambda(|\nabla\Phi|)\, \nabla\Phi)$$

Die Diffusivitätsfunktion kann zum Beispiel nach [PM90] wie folgt definiert sein:

$$g_\lambda(|\nabla\Phi|) = \frac{1}{1+\left(\frac{|\nabla\Phi|}{\lambda}\right)^2}$$

Wir beginnen mit der ursprünglichen Formulierung gemäß (4.9):

$$\begin{aligned}
\partial_t\Phi &= \operatorname{div}\left(g_\lambda(|\nabla\Phi|)\,\nabla\Phi\right) \\
&= \operatorname{div}\left(g_\lambda(|\nabla\Phi|)\cdot\frac{\partial\Phi}{\partial x}, g_\lambda(|\nabla\Phi|)\cdot\frac{\partial\Phi}{\partial y}, g_\lambda(|\nabla\Phi|)\cdot\frac{\partial\Phi}{\partial z}\right)^T \\
&= \frac{\partial\left(g_\lambda(|\nabla\Phi|)\cdot\frac{\partial\Phi}{\partial x}\right)}{\partial x} + \frac{\partial\left(g_\lambda(|\nabla\Phi|)\cdot\frac{\partial\Phi}{\partial y}\right)}{\partial y} + \frac{\partial\left(g_\lambda(|\nabla\Phi|)\cdot\frac{\partial\Phi}{\partial z}\right)}{\partial z}
\end{aligned} \tag{7.7}$$

Mit zeitlicher Diskretisierung über Vorwärtsdifferenzen und der Annahme eines isotropen Gitters mit $\Delta x = \Delta y = \Delta z = 1$ erhalten wir:

$$\begin{aligned}\widehat{\Phi}(t+\tau) \;\approx\; \widehat{\Phi}(t) + \tau \cdot \Bigg(& \frac{g_\lambda(|\nabla \Phi_{x_+}|) \cdot \frac{\partial \Phi_{x_+}}{\partial x} - g_\lambda(|\nabla \Phi_{x\_}|) \cdot \frac{\partial \Phi_{x_-}}{\partial x}}{2} + \\ & \frac{g_\lambda(|\nabla \Phi_{y_+}|) \cdot \frac{\partial \Phi_{y_+}}{\partial y} - g_\lambda(|\nabla \Phi_{y\_}|) \cdot \frac{\partial \Phi_{y_-}}{\partial y}}{2} + \\ & \frac{g_\lambda(|\nabla \Phi_{z_+}|) \cdot \frac{\partial \Phi_{z_+}}{\partial z} - g_\lambda(|\nabla \Phi_{z\_}|) \cdot \frac{\partial \Phi_{z_-}}{\partial z}}{2} \Bigg)\end{aligned} \tag{7.8}$$

Als Paramter erhält die Diffusivitätsfunktion g_λ einen skalaren Wert, im Falle des ersten Termes zum Beispiel $|\nabla \Phi_{x_+}|$. Um diese partielle Ableitung an der diskreten Stelle $x+1$ zu errechnen, wird wieder eine zentrale Differenz benötigt. Wir erhalten damit $|\nabla \Phi_{x_+}| = |\frac{1}{2} \cdot (\Phi(x+2,y,z) - \Phi(x,y,z))|$ bzw, analoge Formeln für die Gradienten in y und z Richtung. Die diskretisierte Formel benötigt also in Summe die 2-Nachbarschaft des betrachteten Voxels um einerseits die diskreten Gradienten und andererseits darüberhinaus die partiellen Ableitungen zu bestimmen.

7.1.4 Nichtlineares anisotropes Modell

Das inhomogene nichtlineare anisotrope Modell kann nicht ganz so einfach wie das isotrope Modell diskretisiert werden. Da der Diffusionstensor kein Skalar mehr ist, kann er nicht aus der Divergenz in Formel 4.15 ausgeklammert werden. Mit den bisher verwendeten Schreibkonventionen erhalten wir, wieder unter Zuhilfenahme von Vorwärtsdifferenzen für den Zeitschritt, das Folgende diskretisierte Modell:

$$
\begin{aligned}
\widehat{\Phi}(t+\tau) &\approx \widehat{\Phi}(t) + \tau \cdot \operatorname{div}(D(\nabla\Phi)\nabla\Phi) \\
&= \widehat{\Phi}(t) + \tau \cdot \operatorname{div}\left(\begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial\widehat{\Phi}}{\partial x} \\ \frac{\partial\widehat{\Phi}}{\partial y} \\ \frac{\partial\widehat{\Phi}}{\partial z} \end{pmatrix}\right) \\
&= \widehat{\Phi}(t) + \tau \cdot \operatorname{div}\begin{pmatrix} d_{11}\frac{\partial\widehat{\Phi}}{\partial x} + d_{12}\frac{\partial\widehat{\Phi}}{\partial y} + d_{13}\frac{\partial\widehat{\Phi}}{\partial z} \\ d_{21}\frac{\partial\widehat{\Phi}}{\partial x} + d_{22}\frac{\partial\widehat{\Phi}}{\partial y} + d_{23}\frac{\partial\widehat{\Phi}}{\partial z} \\ d_{31}\frac{\partial\widehat{\Phi}}{\partial x} + d_{32}\frac{\partial\widehat{\Phi}}{\partial y} + d_{33}\frac{\partial\widehat{\Phi}}{\partial z} \end{pmatrix} \\
&= \widehat{\Phi}(t) + \tau \cdot \left(\frac{\partial}{\partial x}\left(d_{11}\frac{\partial\widehat{\Phi}}{\partial x} + d_{12}\frac{\partial\widehat{\Phi}}{\partial y} + d_{13}\frac{\partial\widehat{\Phi}}{\partial z}\right) + \right. \\
&\qquad \frac{\partial}{\partial y}\left(d_{21}\frac{\partial\widehat{\Phi}}{\partial x} + d_{22}\frac{\partial\widehat{\Phi}}{\partial y} + d_{23}\frac{\partial\widehat{\Phi}}{\partial z}\right) + \\
&\qquad \left.\frac{\partial}{\partial z}\left(d_{31}\frac{\partial\widehat{\Phi}}{\partial x} + d_{32}\frac{\partial\widehat{\Phi}}{\partial y} + d_{33}\frac{\partial\widehat{\Phi}}{\partial z}\right)\right)
\end{aligned}
$$

Die diskrete Richtungsableitung der einzelnen Summen kann wieder über zentrale Differenzen gelöst werden, also zum Beispiel:

$$
\begin{aligned}
&\frac{\partial}{\partial x}\left(d_{11}\frac{\partial\widehat{\Phi}}{\partial x} + d_{12}\frac{\partial\widehat{\Phi}}{\partial y} + d_{13}\frac{\partial\widehat{\Phi}}{\partial z}\right) \\
\approx\ &\frac{\left(d_{11}\frac{\partial\widehat{\Phi}_{x+}}{\partial x} + d_{12}\frac{\partial\widehat{\Phi}_{x+}}{\partial y} + d_{13}\frac{\partial\widehat{\Phi}_{x+}}{\partial z}\right) - \left(d_{11}\frac{\partial\widehat{\Phi}_{x-}}{\partial x} + d_{12}\frac{\partial\widehat{\Phi}_{x-}}{\partial y} + d_{13}\frac{\partial\widehat{\Phi}_{x-}}{\partial z}\right)}{2}
\end{aligned}
$$

Hierbei bezeichnen die Einträge d_{ij} die Komponenten des Diffusionstensors D. Zu deren Berechnung benötigen wir schließlich die Hesse-Matrix H, bzw. deren Einträge h_{ij}, deren Diskretisierung wir im Folgenden betrachten wollen.

Diskretisierung der Hesse-Matrix

Nach dem Satz von Schwarz gilt $\frac{\partial}{\partial x}\left(\frac{\partial}{\partial y}f(x,y)\right) = \frac{\partial}{\partial y}\left(\frac{\partial}{\partial x}f(x,y)\right)$, was bedeutet, dass die Reihenfolge der partiellen Ableitungen bei mehrfach differenzierbaren Funktionen nicht entscheidend ist. Die Hesse-Matrix kann also im 3-dimensionalen für jeden Ort x, y, z mit 6 Einträgen beschrieben werden und ist symmetrisch.

$$H = \begin{pmatrix} h_{xx} & h_{xy} & h_{xz} \\ h_{yx} & h_{yy} & h_{yz} \\ h_{zx} & h_{zy} & h_{zz} \end{pmatrix} = \begin{pmatrix} h_{xx} & h_{xy} & h_{xz} \\ h_{xy} & h_{yy} & h_{yz} \\ h_{xz} & h_{yz} & h_{zz} \end{pmatrix} \tag{7.9}$$

Die Diagonalelemente bestehen also aus zweifachen Ableitung in derselben Raumrichtung. Sie können über einem isotropen Gitter mit Abstand $\Delta x = \Delta y = \Delta z = 1$ mit Hilfe zentraler Differenzen wie folgt diskretisiert werden:

$$h_{xx} = \frac{(\frac{\widehat{\Phi}_{x_+} - \widehat{\Phi}_x}{\Delta x}) - (\frac{\widehat{\Phi}_x - \widehat{\Phi}_{x_-}}{\Delta x})}{\Delta x} = \widehat{\Phi}_{x_+} - 2\widehat{\Phi}_x + \widehat{\Phi}_{x_-} \tag{7.10}$$

$$h_{yy} = \frac{(\frac{\widehat{\Phi}_{y_+} - \widehat{\Phi}_y}{\Delta y}) - (\frac{\widehat{\Phi}_y - \widehat{\Phi}_{y_-}}{\Delta y})}{\Delta y} = \widehat{\Phi}_{y_+} - 2\widehat{\Phi}_y + \widehat{\Phi}_{y_-} \tag{7.11}$$

$$h_{zz} = \frac{(\frac{\widehat{\Phi}_{z_+} - \widehat{\Phi}_z}{\Delta z}) - (\frac{\widehat{\Phi}_z - \widehat{\Phi}_{z_-}}{\Delta z})}{\Delta z} = \widehat{\Phi}_{z_+} - 2\widehat{\Phi}_z + \widehat{\Phi}_{z_-} \tag{7.12}$$

Die zweiten partiellen Ableitungen in unterschiedliche Raumrichtungen erhält man am einfachsten, indem man zunächst über vier benachbarte Gitterpunkte einen Mittelwert bildet und diesen als interpolierten Wert in der Mitte einer Gitterzelle annimmt. Nun kann mit Hilfe zentraler Differenzen die erste Richtungsableitung zum Mittelwert der nächsten Gitterzelle gebildet werden, der Abstand beträgt dabei wieder 1. Der so gefundene Wert der Ableitung liegt in der ersten Ableitungsrichtung direkt auf dem Gitter, in der zweiten Ableitungsrichtung aber auf halber Gitterbreite und kann nun verwendet werden, um wieder mit Abstand 1 über eine zweite zentrale Differenz die zweite Ableitung zu approximieren. Abbildung (33) verdeutlicht dieses Vorgehen.
Wir erhalten so für die drei übrigen gemischten Ableitungen der Hesse-Matrix:

$$h_{xy} = \left(\frac{\widehat{\Phi}_{x_+y_+z} + \widehat{\Phi}_{x_+yz} + \widehat{\Phi}_{xy_+z} + \widehat{\Phi}_{xyz}}{4} - \frac{\widehat{\Phi}_{x_-y_+z} + \widehat{\Phi}_{x_-yz} + \widehat{\Phi}_{xy_+z} + \widehat{\Phi}_{xyz}}{4} \right) - \left(\frac{\widehat{\Phi}_{x_+y_-z} + \widehat{\Phi}_{x_+yz} + \widehat{\Phi}_{xy_-z} + \widehat{\Phi}_{xyz}}{4} - \frac{\widehat{\Phi}_{x_-y_-z} + \widehat{\Phi}_{x_-yz} + \widehat{\Phi}_{xy_-z} + \widehat{\Phi}_{xyz}}{4} \right) \quad (7.13)$$

$$h_{xz} = \left(\frac{\widehat{\Phi}_{x_+yz_+} + \widehat{\Phi}_{x_+yz} + \widehat{\Phi}_{xyz_+} + \widehat{\Phi}_{xyz}}{4} - \frac{\widehat{\Phi}_{x_-yz_+} + \widehat{\Phi}_{x_-yz} + \widehat{\Phi}_{xyz_+} + \widehat{\Phi}_{xyz}}{4} \right) - \left(\frac{\widehat{\Phi}_{x_+yz_-} + \widehat{\Phi}_{x_+yz} + \widehat{\Phi}_{xyz_-} + \widehat{\Phi}_{xyz}}{4} - \frac{\widehat{\Phi}_{x_-yz_-} + \widehat{\Phi}_{x_-yz} + \widehat{\Phi}_{xyz_-} + \widehat{\Phi}_{xyz}}{4} \right) \quad (7.14)$$

$$h_{yz} = \left(\frac{\widehat{\Phi}_{xy_+z_+} + \widehat{\Phi}_{xy_+z} + \widehat{\Phi}_{xyz_+} + \widehat{\Phi}_{xyz}}{4} - \frac{\widehat{\Phi}_{xy_-z_+} + \widehat{\Phi}_{xy_-z} + \widehat{\Phi}_{xyz_+} + \widehat{\Phi}_{xyz}}{4} \right) - \left(\frac{\widehat{\Phi}_{xy_+z_-} + \widehat{\Phi}_{xy_+z} + \widehat{\Phi}_{xyz_-} + \widehat{\Phi}_{xyz}}{4} - \frac{\widehat{\Phi}_{xy_-z_-} + \widehat{\Phi}_{xy_-z} + \widehat{\Phi}_{xyz_-} + \widehat{\Phi}_{xyz}}{4} \right) \quad (7.15)$$

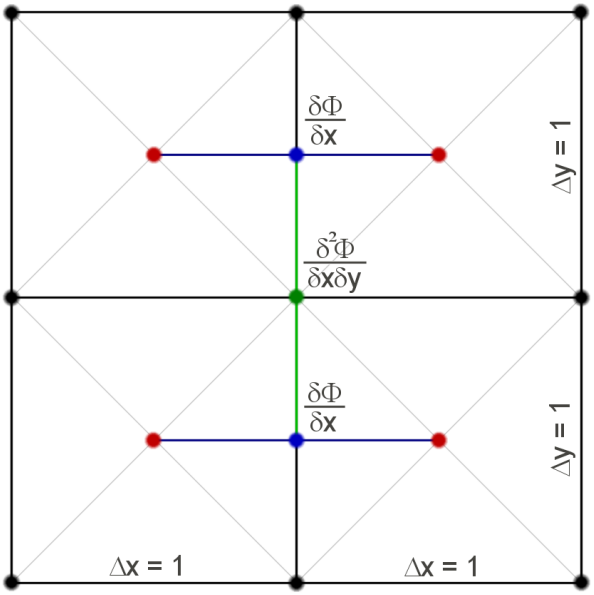


Abbildung 33: Berechnung der diskreten 2. Ableitung auf einem regulären Gitter
Zunächst wird der Mittelwert benachbarter Gittermittelpunkte gebildet (rot). Nun kann in x - Richtung über zentrale Differenzen die erste Richtungsableitung mit Abstand 1 gebildet werden (blau). Abschließend kann über eine weitere zentrale Differenz die Ableitung in der y - Richtung approximiert werden und wir erhalten diesen Wert anschaulich für den Mittelpunkt des Gitters (grün). Die Berechnung erfolgt analog für 3D Gitter – siehe Formel (7.13), (7.14) und (7.15)

Der EED Diffusionstensor

Der Diffusionstensor der EED ist definiert als $D = V \cdot Diag(1, 1, g(\nabla\Phi)) \cdot V^{-1}$. Die orthonormale Rotationsmatrix V besteht aus den Eigenvektoren der auf die Tangentialebene der Isooberfläche projizierten Hesse-Matrix $e1$ und $e2$ und der Isooberflächennormale n:

$$
\begin{aligned}
D &= \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{pmatrix} \\
&= \begin{pmatrix} e1_x & e2_x & n_x \\ e1_y & e2_y & n_y \\ e1_z & e2_z & n_z \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & g(\nabla\Phi) \end{pmatrix} \begin{pmatrix} e1_x & e1_y & e1_z \\ e2_x & e2_y & e2_z \\ n_x & n_y & n_z \end{pmatrix}
\end{aligned} \tag{7.16}
$$

mit

$$
\begin{aligned}
d_{11} &= e1_x e1_x + e2_x e2_x + n_x n_x g(\nabla\Phi) \\
d_{12} &= e1_x e1_y + e2_x e2_y + n_x n_y g(\nabla\Phi) \\
d_{13} &= e1_x e1_z + e2_x e2_z + n_x n_z g(\nabla\Phi) \\
d_{21} &= e1_y e1_x + e2_y e2_x + n_y n_x g(\nabla\Phi) \\
d_{22} &= e1_y e1_y + e2_y e2_y + n_y n_y g(\nabla\Phi) \\
d_{23} &= e1_y e1_z + e2_y e2_z + n_y n_z g(\nabla\Phi) \\
d_{31} &= e1_z e1_x + e2_z e2_x + n_z n_x g(\nabla\Phi) \\
d_{32} &= e1_z e1_y + e2_z e2_y + n_z n_y g(\nabla\Phi) \\
d_{33} &= e1_z e1_z + e2_z e2_z + n_z n_z g(\nabla\Phi)
\end{aligned}
$$

Die Eigenwerte und Eigenvektoren können über die geschlossenen Formeln (4.21) und (4.22) direkt aus der Projektion der Hesse-Matrix auf die Tangentialebene der Isooberfläche berechnet werden. Die diskretisierte Version der Hesse-Matrix wurde im vorangegangenen Abschnitt besprochen. Damit sind nun alle Berechnungsschritte behandelt worden, welche für die nichtlineare anisotrope Diffusion benötigt werden. Ein kompakter C artiger Pseudocode, der die besprochene Diskretisierung nutzt und sich im Aufbau an den einzelnen Schritten aus Kapitel 4.12 (S. 43) orientiert, findet sich im Anhang A (S. XX).

7.2 Occupancy und Kernel–Voxel–Mapping

Bereits im Kapitel »GPU Programmierung« wurde auf das Thema Kernel–Voxel–Mapping eingegangen. Die Prämisse lautet, dass ein Thread genau einen Voxel bearbeiten soll. Besteht ein Block aus $8 \times 8 \times 8$ Threads, kann damit ein Subvolumen des Datensatzes mit 512 Voxeln bearbeitet werden. Die feste Codierung der Blockgröße ist jedoch, wie wir im Folgenden noch sehen werden, kritisch zu bewerten und so bietet sich die Entwicklung eines flexibleren Indizierungsschemas an.

Gehen wir davon aus, dass ein Block eine variable Größe von $tx \times ty \times tz$ Threads einnimmt und der zu bearbeitende Volumendatensatz eine Größe von $vx \times vy \times vz$ Voxeln hat. Die Größen seien alle Potenzen von 2 und der Volumendatensatz sei größer als die Ausdehnung des Blocks, so dass die Divisionen ohne Rest aufgehen. Wir benötigen zur volumetrischen Bearbeitung des gesamten Volumens damit mehrere Blöcke:

$$\text{Anzahl der Blöcke } = \frac{vx}{tx} \times \frac{vy}{ty} \times \frac{vz}{tz}$$

Da das Grid jedoch nur zweidimensional angelegt werden kann, muss man in der Blockindizierung eine Komponente doppelt nutzen, um volumetrisch arbeiten zu können. Nutzen wir die x Komponente des Grids, um die x und y Komponente des Volumens bzw. die y Komponente des Grids um die z Komponente des Volumens abzubilden ergibt sich folgende Gridgröße:

$$\text{Größe des Grids } = \frac{vx}{tx} \cdot \frac{vy}{ty} \times \frac{vz}{tz}$$

Dieser lineare Zusammenhang erlaubt es uns, im x und y Index des aktuellen Blocks eine virtuelle dritte Blockkoordinate zu codieren. Abbildung 34 verdeutlicht diesen Zusammenhang für einen Volumendatensatz mit $8 \times 8 \times 8$ Voxeln und einer $4 \times 4 \times 4$ Blockgröße.

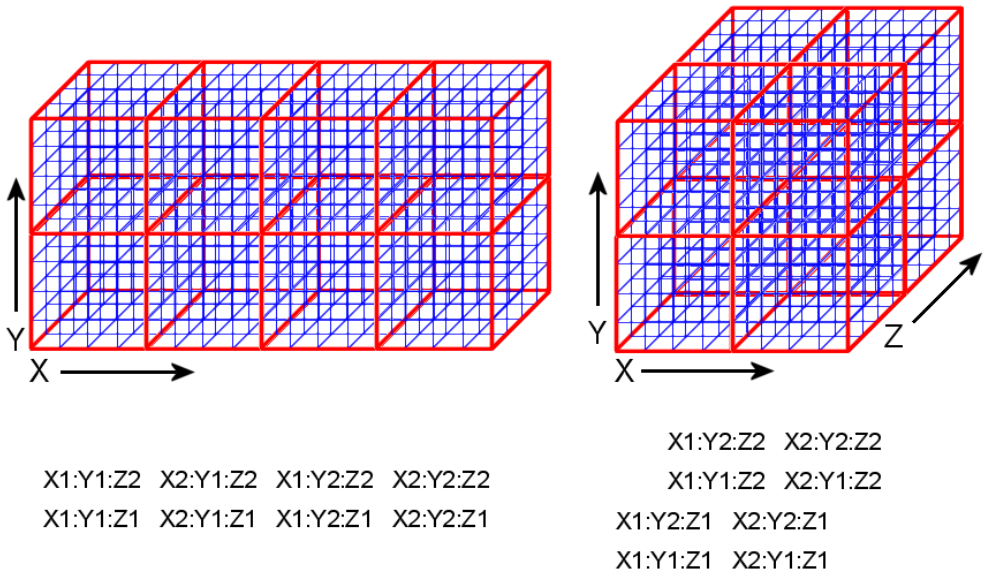


Abbildung 34: Grid-Block-Voxel Mappings für ein $8 \times 8 \times 8$ Volumen.
Es wird ein dreidimensionales Grid mit $2 \times 2 \times 2$ Blöcken benötigt. Diese 8 Blöcke werden jedoch als zweidimensionales Grid mit der Ausdehnung $2 \cdot 2 \times 2$ angelegt (links). Dieses Schema ermöglicht es dem einzelnen Thread aus seiner 2D Blockkoordinate eine virtuelle 3D Blockkoordinate zu ermitteln (rechts) und durch Multiplikation mit der Blockgröße den Offset zum aktuellen Subvolumen zu bestimmen. Der Threadindex wird auf diesen Offset addiert und so die exakte Position des Volumenelements bestimmt.

Die Wahl einer geeigneten Blockgröße kann die Performance des CUDA Programmes erheblich beeinflussen. Je nach verwendetem Device ergeben sich Restriktionen für die maximale Anzahl an Threads pro Block, die Dimensionierung des Blocks und die Dimensionierung des Grids. Diese Informationen können zur Laufzeit vom Device abgefragt werden.

Zusätzlich darf die Summe aller Threads die Zahl der zur Verfügung stehenden Register nicht überschreiten und die minimale Blockgröße sollte unbedingt ein Vielfaches der Warpgröße sein, um die vorhandene Hardware zu nutzen und »coalesced memory access« zu ermöglichen.

Selbst wenn die Zahl der zur Verfügung stehenden Register ausreicht um die maximal unterstützte Blockgröße auszunutzen, kann die optimale Performance bei weitaus weniger Threads liegen. Der technische Hintergrund wird in [Nvi09a, S. 37] beschrieben und liegt unter anderem darin begründet, dass bei kleinerer Blockgröße mehr Blocks auf einem Multiprozessor untergebracht werden können. Dies kann vom CUDA Laufzeitsystem genutzt werden, um Latenzen beim Speicherzugriff zu kaschieren. Nvidia selbst gibt die folgenden

drei Faustregeln an:

»Threads per block should be a multiple of warp size to avoid wasting computation on underpopulated warps and to facilitate coalescing. A minimum of 64 threads per block should be used, but only if there are multiple concurrent blocks per multiprocessor. Between 128 and 256 threads per block is a better choice and a good initial range for experimentation with different block sizes.« [Nvi09a, S. 41]

Die Ausführungskonfiguration des Prototyps kann mit Hilfe des vorgestellten, flexiblen Indizierungsschemas manuell bestimmt werden oder wird – falls keine Angaben vom Benutzer gemacht werden – mittels folgender Heuristik bestimmt:

1. Annahme der kleinstmöglichen Blockkonfiguration: $tx = 1 \times ty = 1 \times tz = 1$
2. Sukzessive Vergrößerung jeder Dimension nach folgender Regel: Solange die gesamte Blockkonfiguration die Hälfte der maximal zulässigen Threads pro Block und die aktuelle Dimension die Warpgröße nicht überschreitet, wird diese Dimension verdoppelt
3. Ansonsten wird mit der nächsten Dimension (ty bzw. tz) fortgefahren.

Für eine maximale Anzahl von 512 Threads pro Block und einer Warpgröße von 32 ergibt sich damit automatisch eine Blockkonfiguration der Größe $32 \times 8 \times 1$. Das Grid wird dann entsprechend gewählt, um den gesamten Datensatz zu verarbeiten. Einige manuelle Versuche haben ergeben, dass dieser heuristisch gefundene Wert immer sehr nahe am jeweiligen Optimum lag.

7.3 Texture memory

CUDA ermöglicht es Teile des global memorys als sogenannten Texturspeicher (texture memory) auszuweisen, indem eine Texturreferenz erzeugt, parametrisiert und an den allokierten global memory gebunden wird. Die Programmierung dieser Texturreferenzen leidet ein wenig an technischen Kinderkrankheiten der aktuellen C für CUDA Version. So können Texturreferenzen zur Zeit nur im sogenannten file scope angelegt werden und die Kapselung in Klassen (C++) ist nicht möglich. Es ist davon auszugehen, dass diese Details der Programmierung in naher Zukunft behoben werden.

Da moderne Grafikkarten nativ ein-, zwei- und dreidimensionale Texturen unterstützen und C für CUDA den Zugriff auf 3D Texturen seit der Entwicklungsversion 2.0 [Nvi08, S. 29] ermöglicht, bieten diese eine natürliche Möglichkeit volumetrische Datensätze auf der

Grafikkarte zu handhaben. Durch Nutzung der Texturreferenzen profitieren die räumlich nahegelegenen Zugriffe auf umgebende Voxel von den 3D Cachingmechanismen der unterliegenden Speicherarchitektur. Der programmiertechnische Aufwand ist trotz leichter Komplikationen vernachlässigbar und die Nutzung des texture memory bietet auf Grund der internen Cachingstrategien der GPU bemerkenswerte Geschwindigkeitsvorteile. Alle Algorithmen des Prototyps wurden in einer den texture memory nutzenden Variante implementiert, um sie im Kapitel »Ergebnisse« mit den Messergebnissen der einfachen »global memory« nutzenden GPU Variante vergleichen zu können.

7.4 Separierbarkeit und texture memory

Die Ausnutzung der Separierbarkeit von Filtern verringert deren Komplexitätsklasse und ist daher für große Datensätze wichtiger als der reine Geschwindigkeitszuwachs durch die Grafikhardware. Es ist daher von Bedeutung sich mit den Aspekten der Separierbarkeit bei der GPU Programmierung zu beschäftigen.

Separierbare Filter arbeiten die Daten in jeder Dimension einzeln ab. Die Zwischenergebnisse müssen entweder in einem temporären Speicherblock gelagert werden oder durch physisches bzw. logisches Kopieren (pointer swapping) auf den benutzten Eingangs- und Ausgangsspeicherblöcken entsprechend arrangiert werden. Einer CPU Implementierung steht in der Regel genug Speicher zur Verfügung, um einen temporären Bereich für die Zwischenspeicherung zu allokieren, falls die Eingangsdaten nicht zerstört werden dürfen.

Da bei der Implementierung auf der Grafikkarte in CUDA ohnehin zwei Speicherbereiche $DATA_{in}$ und $DATA_{out}$ angelegt werden müssen, von denen der erste die initiale Kopie der Hostdaten empfängt und der zweite das Ergebnis nach Abschluss der Filterung enthalten soll, darf man in der Regel davon ausgehen, dass die Originaldaten in $DATA_{in}$ zerstört werden können. Sie werden in der Regel nicht mehr vom Device zurück zum Host transferiert, sondern freigegeben.
Führt man zuerst die Filterung in x Richtung von $DATA_{in}$ nach $DATA_{out}$ aus, dann die Filterung in y Richtung zurück nach $DATA_{in}$ und schließlich die letzte Filterung in z Richtung wieder nach $DATA_{out}$, so kann man ohne weitere Allokation von Speicher die gesamte Filterung effizient auf der Grafikkarte ausführen. Bei geraden Dimensionszahlen bietet sich logisches Kopieren als Lösung für die letzte Vertauschung der Speicherbereiche an, bzw. der Host erwartet die Daten direkt wieder im Eingabebereich $DATA_{in}$.

Diese Vorgehensweise setzt allerdings voraus, dass $DATA_{in}$ von der GPU beschreibbar ist, also global memory genutzt wird. Werden die Eingangsdaten vom Host an die GPU über

read only memory, zum Beispiel den »constant memory« oder »texture memory« geliefert, wäre eine Separierung des Filters zunächst nur über weitere Allokation von Speicher möglich.

Texture memory ist nicht veränderbar (read only), dafür sind die Zugriffe über die internen Caching-Mechanismen stark beschleunigt. Unter der Prämisse, dass medizinische Datensätze im Vergleich zum verfügbaren Grafikkartenspeicher relativ groß ausfallen, müssen wir davon ausgehen, dass kein weiterer Speicherbereich allokiert werden kann. Andererseits soll die Separierbarkeit des Filters bei der Implementierung auf der Grafikkarte auf jeden Fall umgesetzt werden. Ohne das Caching des texture memorys einzubüßen, konnte dies im Prototyp zu dieser Arbeit wie folgt realisiert werden:

Nach Initiierung des GPU Codes wird zum Host zurückgewechselt und die CUDA Threads werden nach Filterung einer Dimension direkt wieder synchronisiert. Danach wird die GPU angewiesen die Daten in $DATA_{out}$ nicht zurück zum Host, sondern wieder in den $DATA_{in}$ Bereich des Devices umzukopieren (»cudaMemcpy3D« mit Option »cudaMemcpyDeviceToDevice«). Damit stehen die Ergebnisse der ersten Filterung wieder in einem Speicherbereich zur Verfügung, an den die texture memory Referenz gebunden ist und die Filterung in der zweiten Richtung über das Device kann vom Host aus gestartet werden. Das Umkopieren innerhalb der Grafikkarte nimmt vernachlässigbar[14] geringe Zeiten in Anspruch, verglichen mit dem Nachteil des Verzichts auf den texture memory Cache bzw. gar einer nicht separierten Implementierung.

[14] Vernachlässigbar heisst hier, dass die tatsächlich benötigten Zeiten stärker fluktuieren, als die Messgenauigkeit es erlaubt. Teilweise waren die im Millisekundenbereich gemessenen Speichertransferzeiten separierbarer Filter, die intern 2 zusätzliche Speichertransfers benötigten, geringer als die gemessenen Zeiten der nicht separierbaren Varianten, was auch durch den mehrfachen Zugriff auf cachefähigen Speicher erklärt werden kann und sinnvolle Messungen erheblich erschwert. Die Messergebnisse zeigen keine signifikanten Mehrzeiten für den deviceinternen Transfer.

7.5 Numerische Stabilität

Wie bei allen numerischen Lösungen spielt die Stabilität des entwickelten diskreten Diffusionsfilters eine wichtige Rolle. Der vom Gradienten induzierte Diffusionsfluss wird über einen Zeitschritt der Länge Δt interpoliert und die Bildfunktion entsprechend für den nächsten Zeitschritt ausgewertet. Wird der Zeitschritt zu klein gewählt sind sehr viele Iterationen notwendig, um die Lösung für einen bestimmten Zeitpunkt zu erhalten. Wird der Zeitschritt zu groß gewählt, führen numerische Instabilitäten dazu, dass die Lösung nicht konvergiert:

Seien beispielsweise drei benachbarte Volumenelemente mit 0°, 10° und 30° temperiert. Für einen Diffusionstensor von 1, einem Zeitschritt von $\Delta t = 0,1$ Sekunden ergibt sich (Neumannbedingung, kein Fluss über den Rand) bei Nutzung zentraler Differenzen für die Bildung der örtlichen Ableitungen:

$$\begin{aligned} 0° + 0,1 \cdot (10°) &= 1° \\ 10° + 0,1 \cdot ((30° - 10°) - (10° - 0°)) &= 11° \\ 30° + 0,1 \cdot (-20°) &= 28° \end{aligned}$$

Nach $\Delta t = 0,1$ Sekunden sind die Volumenelemente also bei 1°, 11° und 28° Temperatur. Nach hinreichend vielen Iteration werden die Volumenelemente alle 13,3° erreichen. Bei der Wahl eines Zeitschritts von $\Delta t = 2$ Sekunden würde jedoch eine numerische Instabilität die Simulation in einem Schritt zerstören:

$$\begin{aligned} 0° + 2 \cdot (10°) &= 20° \\ 10° + 2 \cdot (10°) &= 30° \\ 30° + 2 \cdot (-20°) &= -10° \end{aligned}$$

In [MUF95] finden sich einfache Übelegungen zur Stabilität des Diffusionsfilters. Die Summe der in die Iteration einfließenden Partialflüsse aller Nachbarvoxel muss kleiner sein, als der Wert des zentral gelegenen Voxels, damit »Oszillationen«, also numerische Instabilitäten, vermieden werden. Mackiewich zeigt, dass die kritischen Zeitschritte von der Dimension und der einbezogenen Nachbarschaft abhängen und gibt für drei Dimensionen und die 26-Nachbarschaft eine Grenze von $\Delta t_{max} = \frac{3}{44} \approx 0.0682$ an.

8 Ergebnisse

Nachfolgend werden die Messergebnisse des Prototyps verkürzt aufgeführt. Es werden dabei lediglich die Ergebnisse von Datensätzen der Größe 256^3 dargestellt. Die ausführlichen Ergebnistabellen mit verschiedenen Datensatzgrößen finden sich im Anhang C ab Seite XXVIII.

Die CPU Messungen wurden auf einem Testsystem mit einer Intel E8500 CPU (»Intel(R) Core(TM)2 Duo CPU, 3.16GHz«) und einer Nvidia G92 GPU (GeForce 9800 GT) durchgeführt. Die Bewertung dieser Systemkonfiguration und der gemessenen Zeiten wird dann im folgenden Kapitel besprochen.

Der Datensatz jeder einzelnen Messung enthält die folgenden Informatonen:

Dim: Die Tests wurden jeweils mit Volumendatensätzen der Größe $64^3, 128^3, 192^3, 256^3$ und 320^3 in »Single Precision« (Fließkommaarithmetik, einfache Genauigkeit, 4 Bytes pro Voxelwert) durchgeführt, die Volumina hatten also Größen von (1 MB, 8 MB, 27 MB, 64 MB und 187,5 MB). Die ausführlichen Ergebnisse befinden sich im Anhang. Die folgende Kurzübersicht verwendet lediglich Volumina der Größe 256^3.

Code: Es wurden jeweils die Geschwindigkeit des CPU Codes (»cpu«), des GPU Codes (»gpu«) sowie des GPU Codes mit Caching über Texturememory (»tex«) für eine komplette Iteration des Filters ermittelt. Im Falle der Edge Enhancing Diffusion (EED) wurden zwei spezielle Codeversionen der GPU und GPU Texture Variante (»gpuopt« und »texopt«) entwickelt, welche durch reine Codeoptimierungen Geschwindigkeitsvorteile erreichen sollen.

MemCpy: Für GPU basierten Code fällt ein Upload (Host-Device) und ein Download (Device-Host) des Volumendatensatzes an. Up- und Downloadzeiten sind gleichgroß und sind daher aus Gründen der Übersichtlichkeit in den Ergebnistabellen nicht noch einmal getrennt dargestellt. Die Messergebnisse liegen in Millisekunden vor.

Filter: Die Laufzeit des eigentlichen Filtercodes. Die Messergebnisse liegen in Millisekunden vor.

Total: Die Gesamtzeit aus eventuell anfallenden MemCopy Operationen und der eigentlichen Laufzeit des Filters. Die Messergebnisse liegen in Millisekunden vor.

Brutto: Der Brutto-Wert ist ein Geschwindigkeitsfaktor und setzt die benötigte Zeit des korrespondierenden (gleiche Datensatzgröße) CPU Filters in Bezug auf die benötigte Zeit des betrachteten Filters (exklusive etwaiger MemCpy Zeiten).

Netto: Der Netto-Wert ist ein Geschwindigkeitsfaktor und setzt die benötigte Zeit des korrespondierenden (gleiche Datensatzgröße) CPU Filters in Bezug auf die benötigte Zeit des betrachteten Filters (inklusive etwaiger MemCpy Zeiten).

Benötigt die CPU beispielsweise 80 ms und die GPU 20 ms für die eigentliche Filteroperation, so ist der Brutto Speedup $\frac{80ms}{20ms} = 4$. Benötigt die GPU jedoch zusätzlich 20 ms für den Up- und Download des Datensatzes erhalten wir einen Netto Speedup von $\frac{80ms}{20ms+20ms} = 2$.

8.1 Prefilter

Es folgen die Messungen für die verschiedenen Prefilter:

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	34,57	34,57	1,0	1,0
gpu	50,84	11,84	62,68	2,9	0,6
tex	51,41	12,17	63,59	2,8	0,5

Tabelle 2: Messergebnisse »ID Filter«

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	1.372,31	1.372,31	1,0	1,0
gpu	51,81	226,40	278,20	6,1	4,9
tex	51,31	77,64	128,95	17,7	10,6

Tabelle 3: Messergebnisse »Boxfilter 1« ($3 \times 3 \times 3$ Kernel)

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	6.222,91	6.222,91	1,0	1,0
gpu	52,60	1.189,98	1.242,58	5,2	5,0
tex	51,42	271,73	323,16	22,9	19,3

Tabelle 4: Messergebnisse »Boxfilter 2« ($5 \times 5 \times 5$ Kernel)

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	684,74	684,74	1,0	1,0
gpu	50,85	66,80	117,65	10,3	5,8
tex	51,35	74,04	125,39	9,2	5,5

Tabelle 5: Messergebnisse »Boxfilter 1« (separiert, $5 \times 5 \times 5$ Kernel)

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	1.115,91	1.115,91	1,0	1,0
gpu	50,82	96,16	146,97	11,6	7,6
tex	51,57	78,90	130,47	14,1	8,6

Tabelle 6: Messergebnisse »Boxfilter 2 (separiert, $5 \times 5 \times 5$ Kernel)«

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	91.396,26	91.396,26	1,0	1,0
gpu	50,87	1.272,99	1.323,86	71,8	69,0
tex	51,33	710,52	761,85	128,6	120,0

Tabelle 7: Messergebnisse »Gaußfilter 1« ($3 \times 3 \times 3$ Kernel)

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	10.678,07	10.678,07	1,0	1,0
gpu	50,81	121,95	172,76	87,6	61,8
tex	51,45	114,58	166,03	93,2	64,3

Tabelle 8: Messergebnisse »Gaußfilter 1« (separiert, $3 \times 3 \times 3$ Kernel)

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	19.148,52	19.148,52	1,0	1,0
gpu	50,82	195,49	246,31	98,0	77,7
tex	51,43	147,80	199,23	129,6	96,1

Tabelle 9: Messergebnisse »Gaußfilter 2« (separiert, $5 \times 5 \times 5$ Kernel)

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	14.754,21	14.754,21	1,0	1,0
gpu	50,82	1.832,01	1.882,83	8,1	7,8
tex	52,36	1.623,84	1.676,20	9,1	8,8

Tabelle 10: Messergebnisse »Medianfilter« ($3 \times 3 \times 3$ Nachbarschaft)

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	301.215,50	301.215,50	1,0	1,0
gpu	50,93	1.452,63	1.503,56	207,4	200,3
tex	52,69	1.371,56	1.424,25	219,6	211,5

Tabelle 11: Messergebnisse »Bilateraler Filter« ($3 \times 3 \times 3$ Kernel)

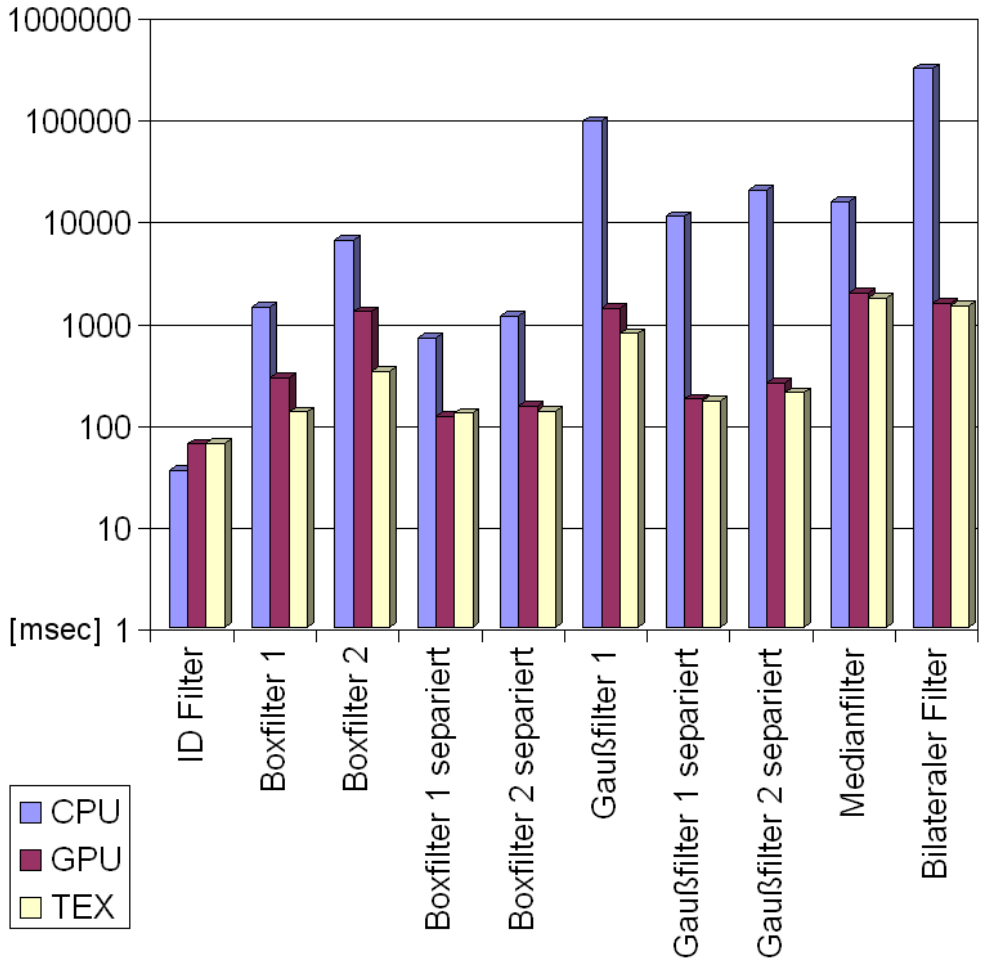


Abbildung 35: Geschwindigkeit der Prefilter im Vergleich. Die Laufzeiten der verschiedenen Filter sind in Millisekunden bemessen und für Volumendatensätze der Größe 256^3 ermittelt worden. Logarithmische Darstellung.

8.2 Diffusions Filter

Die Messung der Diffusions Filter:

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	397,91	397,91	1,0	1,0
gpu	72,09	40,61	112,70	9,8	3,5
tex	72,80	26,10	98,91	15,2	4,0

Tabelle 12: Messergebnisse »Lineare homogene Diffusion«

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	10.108,54	10.108,54	1,0	1,0
gpu	72,20	273,38	345,58	37,0	29,3
tex	72,81	128,31	201,12	78,8	50,3

Tabelle 13: Messergebnisse »Nichtlineare inhomogene Diffusion«

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	37.312,49	37.312,49	1,0	1,0
gpu	72,06	961,71	1.033,77	38,8	36,1
gpuopt	71,85	934,08	1.005,92	39,9	37,1
tex	74,68	1.929,65	2.004,33	19,3	18,6
texopt	74,35	1.071,46	1.145,81	34,8	32,6

Tabelle 14: Messergebnisse »Nichtlineare Anisotrope Diffusion (EED)«

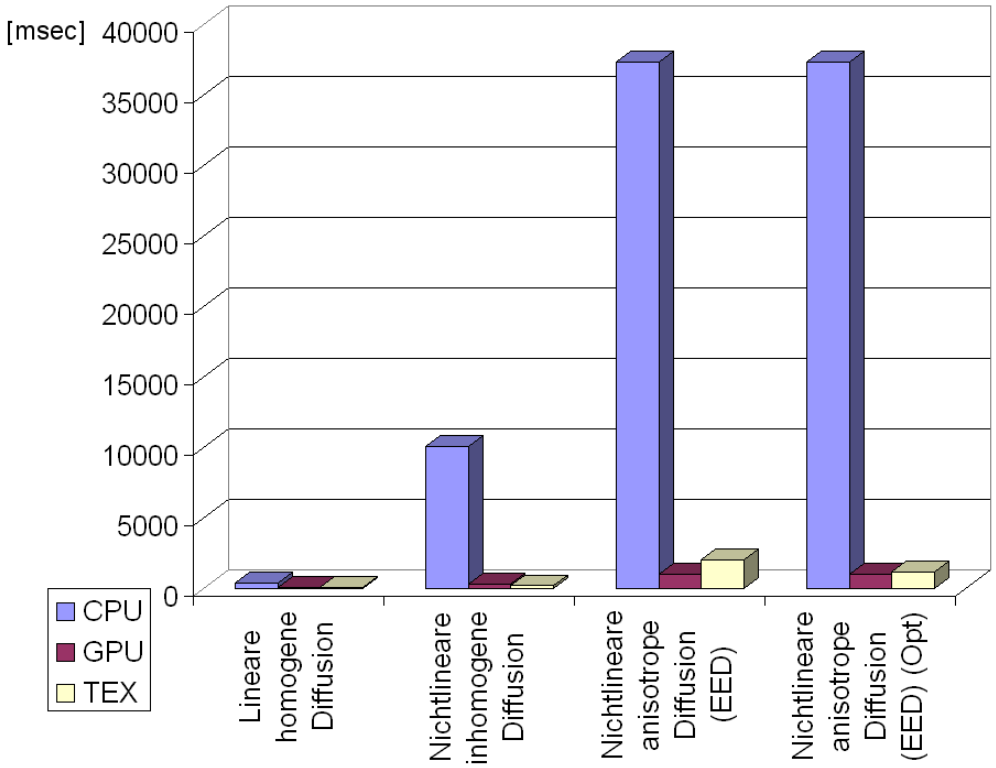


Abbildung 36: Geschwindigkeit der Diffusionsfilter im Vergleich. Die Laufzeiten der verschiedenen Filter sind in Millisekunden bemessen und für Volumendatensätze der Größe 256^3 ermittelt worden.

8.3 Affine Transformationen

Die Messungen der affinen Transformationen:

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	1.173,75	1.173,75	1,0	1,0
gpu	50,75	33,53	84,28	35,0	13,9
tex	51,34	17,59	68,93	66,7	17,0

Tabelle 15: Messergebnisse »Affine Transformation (Nearest Neighbor)«

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	1.664,23	1.664,23	1,0	1,0
gpu	50,77	246,15	296,93	6,8	5,6
tex	51,86	31,46	83,31	52,9	20,0

Tabelle 16: Messergebnisse »Affine Transformation (Trilinear)«

Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
cpu	0,00	4.151,91	4.151,91	1,0	1,0
gpu	51,75	1.846,91	1.898,66	2,2	2,2
tex	51,39	341,39	392,79	12,2	10,6

Tabelle 17: Messergebnisse »Affine Transformation (Cubic B-Splines)«

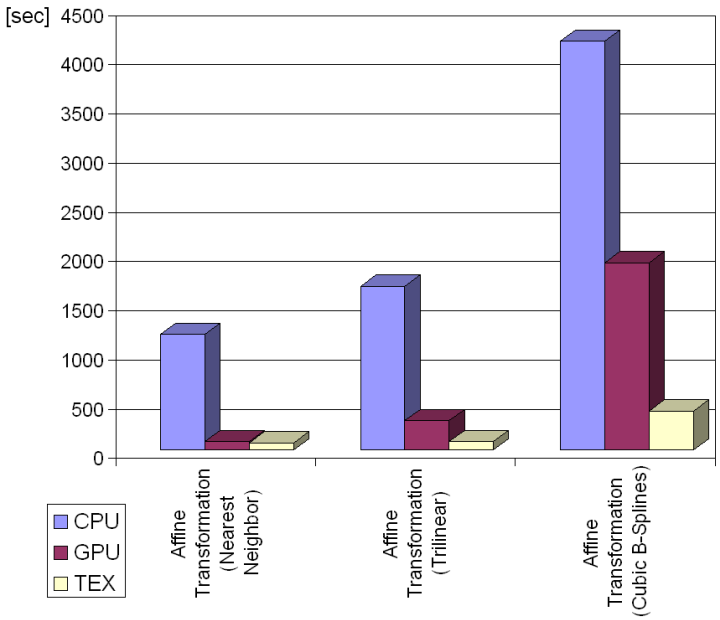


Abbildung 37: Geschwindigkeit der affinen Transformationen im Vergleich. Die Laufzeiten der verschiedenen Transformationen sind in Millisekunden bemessen und für Volumendatensätze der Größe 256^3 ermittelt worden.

9 Diskussion

Es folgt die Diskussion der Messergebnisse vor dem Hintergrund der Aufgabenstellung, eine Reflexion über die Vergleichbarkeit der Daten und eine Zusammenfassung der wichtigsten Erkenntnisse.

9.1 Diskussion der Messergebnisse

In den folgenden Abschnitten werden zunächst die Messergebnisse (basierend auf den Daten der Tabellen im Kapitel »Ergebnisse«) näher erläutert und bewertet. Die Gliederung erfolgt analog zur Vorstellung der Messergebnisse.

9.1.1 ID Filter

Der ID Filter dient einerseits als Minimalbeispiel eines Filters, andererseits zeigt er deutlich die Folgen einer ungünstigen Aufgabenstellung für die GPU Programmierung. Die »niedrige« arithmetische Intensität (0 arithmetische Operationen pro 2 Lese-/Schreibzugriffe im global memory) führt zu schlechter Performance:
Für kleine Datensätze bringt die GPU Implementierung kaum oder gar negative Performancegewinne verglichen mit der CPU Implementierung. Für größere Datensätze, werden zwar rein rechnerisch bis zu 300% an Geschwindigkeitssteigerung verzeichnet, der Memory Transfer zum und vom Device überkompensieren dies jedoch bei weitem, so dass netto nur die halbe Geschwindigkeit der Gesamtausführungszeit der CPU Variante erreicht wird (siehe Tabelle (20), S. XXVIII).

9.1.2 Prefilter

Kennzeichnend für die Ergbnisse des Box Filters sind die zunächst relativ geringen Geschwindigkeitsvorteile von rund 500% der GPU gegenüber den CPU Filtern. Dies ist auf die relativ geringe arithmetische Intensität einerseits und die nicht synchronisierten (non-coalesced) Speicherzugriffe andererseits zurückzuführen. Erwartungsgemäß verringert der Einsatz des texture memorys auf Grund der engen Lokalität der Speicherzugriffe die Latenzzeiten und wir erreichen eine 17- bis 18-fache Ausführungsgeschwindigkeit. Für größere Kernel steigert der texture memory die Ausführungsgeschwindigkeit auf das bis zu 21-fache. Der separierte Boxfilter steigert die absolute Ausführungsgeschwindigkeit beider Implementierungen natürlich erheblich. Der Geschwindigkeitsvorteil der GPU Variante wird etwas geschmälert, da die arithmetische Intensität rapide sinkt. Da in ernstzunehmenden Anwendungen separierbare Filter benötigt werden, können wir nur einen

»netto Speedup« von 5- bis 8-fach festhalten.

Ganz im Gegensatz dazu steht die Filterung mit einem Gaußkern. Die Berechnung der Gewichte durch die exponentiale Normalverteilungsfunktion erhöht die arithmetische Intensität des Kernel beträchtlich, wir erhalten brutto für die naive Implementierung Spitzenwerte von annähernd 13000% - die 130-fache Geschwindigkeit der CPU. Diese Ergebnisse werden jedoch wieder relativiert durch die Speichertransferzeiten und die Anwendung der Separierbarkeit. Dennoch lassen sich als großartiges Ergebnis die typischen Zeiten für ein 256^3 Volumen und einen 5^3 Filterkern nennen: Fast 100-fache Geschwindigkeitsteigerung bei Nutzung der GPU texture memory Variante gegenüber der CPU Implementierung.

Der Medianfilter kann nur weitaus geringer von der GPU Implementierung profitieren. Der netto Speedup von 8- bis 8,8-fach ist darauf zurückzuführen, dass die Sortierung der Voxelwerte die interne Codeausführung der Streamprozessoren oft verzweigen (branchen) lässt. Parallelisierung findet nur statt, wenn alle Daten gleich behandelt werden können, was hier offensichtlich in der Regel nicht der Fall ist. Branching code führt auf der GPU oft zu sequentieller Ausführung und starken Geschwindigkeitseinbrüchen trotz hoher Speicherzugriffslokalität.

Der bilaterale Filter hat die besten Erwartungen mehr als übertroffen. Ein netto Geschwindigkeitsgewinn von teilweise über 211-facher Ausführungsgeschwindigkeit wurde gegenüber der CPU Variante erzielt. Die hervorragenden Eigenschaften des bilateralen Filters werden durch dessen Einsatz recht teuer erkauft, da dieser Filter nicht separierbar ist. Da analog zum Gaußfilter eine hohe arithmetische Intensität vorliegt – es müssen sogar vier Gewichte pro Volumenwert berechnet werden – profitiert dieser Filter enorm von der parallelen Ausführung.

9.1.3 Diffusionsfilter

Die Ergebnisse der Diffusionsfilter können und müssen unter einem speziellen Gesichtspunkt bewertet werden. Da die zugrunde liegende mathematische Formulierung über viele Iterationen mit kleinen Zeitschritten gelöst wird, ist der einzelne Iterationsschritt nicht aussagekräftig. Die Daten müssen bei hundert Iterationen nur einmal zum bzw. vom Device kopiert werden. Die Transferzeiten sind mit anderen Worten nach wenigen Iterationen verschwindend gering, so dass man in der Praxis im allgemeinen annehmen kann, dass sie nicht ins Gewicht fallen. Folglich können wir hier tatsächlich die brutto Geschwindigkeitssteigerungen zum Vergleich mit der CPU Variante heranziehen.

Wir erhalten rund 15-fache Ausführungsgeschwindigkeiten für die lineare homogene Diffusion, fast 80-fache Ausführungsgeschwindigkeiten für die nichtlineare inhomogene Diffusion, sowie fast 40-fache Ausführungsgeschwindigkeiten für die edge enhancing diffusion. Die vergleichsweise niedrigen Geschwindigkeitssteigerungen der linearen homogenen Diffusion sind auf die relativ geringe arithmetische Intensität zurückzuführen.

Da die EED als Beispiel einer typisch nichtlinearen anisotropen Diffusion einerseits viele Berechnungen für die Eigenwertzerlegung benötigt, andererseits auch viele Speicherzugriffe für die Berechnung der ersten und zweiten Ableitungen aufweist, kann man die Ausführungsgeschwindigkeit steigern, indem man den Programmcode so arrangiert, dass Speicherzugriffe bereits initiiert werden, bevor der jeweilige Wert tatsächlich benötigt wird und danach weiter Berechnungen ausgeführt werden. Der Programmcode entbehrt zwar aus Sicht des Programmierers jeglicher vernünftigen Struktur, erreicht aber eine bessere Maskierung der Zugriffslatenzen.
Die auf diese Weise optimierten Filter konnten bis zu 35-fache Geschwindigkeitssteigerungen erreichen.
Für aufwändigere Diffusionstensoren (beispielsweise die »Coherence-Enhancing-Diffusion«, die weitere Nebenrechnungen auf Grundlage der Eigenwerte durchführt) sind vermutlich noch bessere Ergebnisse zu erwarten.

9.1.4 Affine Transformationen

Die netto Geschwindigkeit der affinen Transformationen mit »nearest neighbor« – Zugriff übersteigt die CPU Variante um das rund 15-20 fache. Die trilineare Filterung ist als »Code-Einzeiler« in der texture memory Variante quasi trivial – unterstützen Grafikkarten doch inhärent den trilinear gefilterten Zugriff zwischen den Werten. Auf Grund der verhältnismäßig geringen bis mittleren arithmetischen Intensität – viele Werte der affinen Transformationsmatrix können einmal für alle Voxel vorberechnet werden – erreichen diese Varianten nur rund 20-fache netto Geschwindigkeitssteigerungen - allerdings ist eine affine Transformation oft der Beginn zu einer weiteren Bearbeitung und so könnten abhängig vom konkreten Anwendungsfall (der eventuell ohnehin auf der GPU stattfinden wird) unter Umständen auch die brutto Zeiten mit 50- bis 75-fachen Ausführungsgeschwindigkeiten angenommen werden.

Die aufwändige affine Transformation mit Filterung über B-Splines ist zunächst ein vielversprechender Kandidat für die GPU Implementierung. Dennoch verhindern zwei Faktoren den absoluten Durchbruch: Einerseits müssen 64 Stützstellen aus dem Speicher gelesen werden, andererseits kann die CPU Variante ihre Polynomberechnungen massiv durch den

Einsatz von Lookup-Tables beschleunigen. Unter diesen Voraussetzungen erzielt die GPU Variante zunächst nur eine Steigerung von 220% bis 240%. Wird der texture memory genutzt, kann zumindest der Zugriff auf die 64 Stützstellen massiv beschleunigt werden und wir erreichen immerhin eine über 10-fach schnellere Ausführungsgeschwindigkeit.

9.2 Kritische Methodenreflexion

Der Vergleich von CPU und GPU Code muss immer sehr sorgfältig erfolgen und eine valide Bewertung muss ganzheitlich auf Grundlage vieler verschiedener Aspekte erfolgen. In der vorliegenden Arbeit wurde bewusst Wert darauf gelegt, nicht über dem reinen Geschwindigkeitsvorteil andere Aspekte zu vernachlässigen. Salopp gesprochen verleitet der GPGPU Hype schnell dazu, das Versprechen nach hundertfachen Geschwindigkeitssteigerungen als Fakt anzusehen. Der Blick auf die reinen Zahlen führt zu keinen brauchbaren Ergebnissen und so sollen hier qualitative Aussagen über die wirklich erzielbaren Geschwindigkeitvorteile getroffen werden. Um die Schlussfolgerungen im folgenden Abschnitt weiter zu untermauern, sollen noch einmal drei wesentliche Punkte klar angesprochen werden:

1. Die Geschwindigkeitsmessung erfolgte auf im Consumer-Markt üblicher Hardware. Zum Zeitpunkt der Evaluierung der Messergebnisse war eine GeForce 9800GT für ca. 80 Euro im Handel erhältlich, ein Intel Core 2 Duo E8500 mit 3.16 GHz kostete rund 200 Euro. Die eingesetzte CPU ist verglichen zur GPU relativ leistungsstark aber durchaus üblich. Für eine Umrechnung »Geschwindigkeit pro Hardwarekosten« müssten also Korrekturfaktoren angelegt werden, welche die Messergebnisse der GPU Implementierung vermutlich noch besser abschneiden lassen, als dies in der vorliegenden Arbeit der Fall ist.

2. Bei der Evaluierung der Messergebnisse wurde stets berücksichtigt, dass der Geschwindigkeitszuwachs nur dann als zuverlässig angesehen werden kann, wenn alle Aspekte des Filterdesigns mit berücksichtigt werden. Daher wurden zum Beispiel in der Regel die schlechteren netto Zeitfaktoren (mit Speichertransfer) verwendet, nach Möglichkeit separierbare Filter mit schlechterer arithmetischer Intensität betrachtet oder CPU Optimierungen (Lookup-Table) aber auch GPU Optimierungen (Texture memory oder Code design) mit in Betracht gezogen. Die Vergleichsergebnisse sollen die unterschiedlichen Architekturen mit ihren Potentialen voll berücksichtigen.

3. Ein Vorteil, der in echten Umgebungen einen weiteren Geschwindigkeitsgewinn für die GPU Implementierungen verspricht, ist die Nebenläufigkeit des GPU Kernels. Während die GPU die Filterung durchführt, kann der Host Thread bereits weitere sinnvolle Berech-

nungen durchführen.

Diese drei Punkte verdeutlichen einerseits, wie schwer es ist CPU und GPU allgemein zu vergleichen, und dass es daher sinnvoll ist, nur ungefähre Geschwindigkeitssteigerungen anzugeben. Andererseits sollte sichergestellt sein, dass die folgenden zusammenfassenden Aussagen eher zu niedrig als zu hoch gegriffen sind und ein konkreter volumetrischer Diffusionsfilter unter Einbeziehung aller Randbedingungen eher noch bessere Ergebnisse liefert als die hier vorgestellten Messungen des Prototyps.

9.3 Zusammenfassung der wichtigsten Ergebnisse

Auf Grund der Messungen am Prototyp kann man für maßgeschneiderte Implementierungen eines GPU basierten volumetrischen Filters mindestens folgende Geschwindigkeitssteigerungen gegenüber der CPU Implementierung erwarten:

1. Volumetrischer Boxfilter: 5 bis 8-fache Geschwindigkeit
2. Volumetrischer Gaußfilter: ca. 100-fache Geschwindigkeit
3. Volumetrischer Medianfilter: 8 bis 9-fache Geschwindigkeit
4. Volumetrischer bilateraler Filter: 200- bis 210-fache Geschwindigkeit
5. Volumetrischer linearer homogener Diffusionsfilter: ca. 15-fache Geschwindigkeit
6. Volumetrischer nichtlinearer inhomogener Diffusionsfilter: ca. 80-fache Geschwindigkeit
7. Volumetrischer nichtlinearer anisotroper Diffusionsfilter: ca. 40-fache Geschwindigkeit
8. Volumetrische affine Transformation (Nearest Neighbor): 15 bis 20-fache Geschwindigkeit (mit Weiterverarbeitung auf der GPU 60 bis 75-fach)
9. Volumetrische affine Transformation (Trilinear): ca 20-fache Geschwindigkeit (mit Weiterverarbeitung auf der GPU 35 bis 55-fach)
10. Volumetrische affine Transformation (kubische B-Splines): ca. 10-fache Geschwindigkeit

Diese Geschwindigkeitssteigerungen wurden mit dem Prototyp mindestens erzielt, und – wo immer möglich – zu niedrig angesetzt.

Ein typisches Beispiel für eine vollständige Anwendungskette könnte die affine Transformation mit trilinearer Filterung eines Volumendatensatzes mit 256^3 Voxeln sein, der anschließend mit bilateralem Filter vorgeglättet und dann hundert Iterationen der Edge Enhancing Diffusion durchläuft. Auf Grund der Messergebnisse des Testsystems kann man für die jeweilige Implementierung rund

$$\begin{aligned} t_{\text{cpu}} &= t_{\text{afftri\_cpu}} + t_{\text{bilat\_cpu}} + 100 \cdot t_{\text{eed\_cpu}} \qquad (9.1) \\ &= 1664 \text{ ms} + 301216 \text{ ms} + 100 \cdot 37312 \text{ ms} \\ &= 4034080 \text{ ms} \\ t_{\text{gpu}} &= t_{\text{memcpy\_gpu}} + t_{\text{afftri\_gpu}} + t_{\text{bilat\_gpu}} + 100 \cdot t_{\text{eed\_gpu}} \qquad (9.2) \\ &= 51 \text{ ms} + 31 \text{ ms} + 1372 \text{ ms} + 100 \cdot 1071 \text{ ms} \\ &= 108554 \text{ ms} \end{aligned}$$

annehmen und damit eine $\frac{t_{cpu}}{t_{gpu}} \approx 40$-fache Geschwindigkeitssteigerung erreichen. Das Ergebnis einer solchen Berechnung läge nach 1 Minute und 49 Sekunden statt rund 67 Minuten vor, ohne dass weitere Potentiale, wie schnellere Grafikhardware oder die Nebenläufigkeit der CPU Threads, genutzt wurden.

Ein letzter Aspekt der ebenfalls zur Sprache gebracht werden sollte, ist der relativ einfache Einstieg in die Programmierung mit CUDA. Die gute Dokumentation und die unzähligen Beispiele im Internet verhelfen einem C Programmierer innerhalb weniger Stunden zu passablen Ergebnissen. Die Programmierung selbst gestaltet sich als relativ nahtlos in C integriert. Leichte Probleme mit C++ (Templates und Vererbung) sollten zum Abgabetermin dieser Arbeit bereits behoben sein, da Nvidia bereits CUDA 3.0 veröffentlicht hat. Der letzte wissenschaftliche Makel, die Beschränkung auf float Typen (Fließkommaarithmetik mit einfacher Genauigkeit), ist mit Ankündigung der neuen Produktreihen auch im Consumer Markt in absehbarer Zeit aus dem Weg geräumt. Der implementierte Prototyp nutzt eine »typdef cudafloat float« Definition. Die Änderung dieser Zeile in »typedef cudafloat double« sollte bereits genügen, um den Prototypen mit doppelter Genauigkeit auf neuen Grafikkarten laufen zu lassen.

10 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde untersucht, wie sich volumetrische Diffusionsfilter mathematisch beschreiben und diskretisieren lassen und wie deren Implementierung auf der GPU konkret realisiert werden kann. Anhand eines Prototyps in C für CUDA sollte geprüft werden, welche Geschwindigkeitsvorteile eine solche Implementierung im Vergleich zur CPU Lösung erzielt und ob Vorfilterungsschritte, insbesondere der sogenannte bilaterale Filter, ebenfalls effizient auf der GPU umgesetzt werden können.

Dazu geben die Kapitel »Mathematische Grundlagen« und »Diffusionsfilter in der Bildverarbeitung« einen fundierten Überblick über die Beschreibung volumetrischer Diffusionsfilter und deren Formulierung über partielle Differentialgleichungssysteme. Alle Überlegungen finden dabei direkt im $\mathbb{R}^3$ statt. Es werden Schritt für Schritt Verbesserungen am Modell des linearen homogenen Diffusionsfilter vorgenommen, bis wir die Formulierung des nichtlinearen anisotropen Diffusionsfilter erhalten.
Der nichtlineare anisotrope Diffusionsfilter steuert den Diffusionsprozess durch die gezielte Ausrichtung der partiellen Ausgleichsflüsse an der lokalen Bildstruktur mit Hilfe des sogenannten Diffusionstensors. Der bekannteste anisotrope Diffusionsfilter, der »edge enhancing diffusion filter« (EED) richtet den Fluss dabei entlang der Kanten aus und unterdrückt den Fluss senkrecht zur Kante. Zur Definition des Diffusionstensors wird die Ableitung der Hesse-Matrix untersucht, um die Raumrichtungen mit den größten und kleinsten Gradientenänderungen zu bestimmen. Dies geschieht über eine Eigenwertzerlegung, die im volumetrischen Fall die Lösung eines charakteristischen Polynoms dritten Grades nach sich zieht. Über die Projektion der Hesse-Matrix auf die Tangentialebene der Isooberfläche wird ein Weg aufgezeigt das Problem deterministisch, mit wenigen Fallunterscheidungen, direkt über geschlossene Formeln im $\mathbb{R}^2$ zu lösen. Bisher ist keine Arbeit bekannt, die sich diese Vorgehensweise zunutze macht und so die Berechnungsgeschwindigkeit der volumetrischen ansiotropen Diffusion erhöht.

Es wird ein vollständiger Algorithmus zur Gewinnung des Diffusionstensors angegeben und im Kapitel »Implementierung« werden schließlich insbesondere alle Aspekte der räumlichen und zeitlichen Diskretisierung besprochen, sowie GPU spezifische Aspekte der Umsetzung näher beleuchtet: Bereits die Überlegungen im Abschnitt »GPU Programmierung« hatten deutlich gemacht, dass der sogenannte coalesced memory access für dreidimensionale Daten nicht effizient umgesetzt werden kann, was zu Geschwindigkeitseinbußen führt. Der vielversprechende Ansatz 3D Texturen (ein »read-only« memory) zu verwenden, um über die Texture-Caching Mechanismen der GPU die Geschwindigkeitsverluste

beim Speicherzugriff auszugleichen, führt im Fall von separierbaren Filtern zunächst zu Problemen. Diese können jedoch durch das sehr schnelle Umkopieren von Daten und neues Binden an Texturspeicher innerhalb des Devices umgangen werden. Gleichzeitig ist mit diesem Ansatz auch die wiederholte Anwendung von Filtern bzw. die mehrfache Anwendung einzelner Iterationen des Diffusionsfilters ohne erneuten Host-Device Speichertransfer möglich, was den zusätzlichen Zeitaufwand für die Nutzung der GPU erheblich relativiert.

Es zeigt sich, dass bereis bei geringer arithmetischer Intensität gute Geschwindigkeitssteigerungen erzielt werden können: Selbst affine Transformationen mit kubischen B-Splines, die relativ wenige Berechnungen und viele Speicherzugriffe benötigen, erreichen wenigstens die zehnfache Ausführungsgeschwindigkeit verglichen mit einer CPU Implementierung. Die untersuchten Vorfilter erreichten je nach Funktion unterschiedliche Geschwindigkeitssteigerungen, ungefähr 8 bis 9-fach für die volumetrische Medianfilterung, 100-fach für volumetrische Gaußfilterung und bis zu 210-fache Geschwindigkeitssteigerungen für die bilaterale Vorfilterung.
Die im Prototyp umgesetzten Diffusionsfilter erreichten 55-fache (lineare homogene Diffusion), 80-fache (nichtlineare inhomogene Diffusion) bzw. 40-fache (nichtlineare anistrope Diffusion, EED) Geschwindigkeitssteigerungen. Es ist davon auszugehen, dass eine zusammengesetzte Filterung im Rahmen eines echten Softwareprojekts mit kombinierter affiner Transformation, Vorfilterung und mehreren Iterationen eines anisotropen Diffusionsfilters rund 40-fache Ausführungsgeschwindigkeiten gegenüber einer CPU Implementierung erreichen kann.

Im Hinblick auf die relativ geringe Einarbeitungszeit und die guten Ergebnisse, kann man die Umsetzung von Diffusionsfiltern auf Grafikkarten uneingeschränkt empfehlen. Die rasante Weiterentwicklung der GPGPU Technologie verspricht ständig wachsende Leistungen, die Kosten für die benötigte Hardware sind verschwindend gering. Der technologische Fortschritt im Bereich der GPU Entwicklung ist sehr viel höher als im klassischen CPU Markt. Neue Chipgenerationen führen leistungsfähige Cachinghierarchien und die im wissenschaftlichen Kontext lang ersehnten Fließkommazahlen in double precision nach dem IEEE 754-2008 Floating-Point Standard ein.

Das vorgestellte Thema lässt sich in verschiedene Richtungen weiterentwickeln. Weitere Fragestellungen der Diffusionsfilter betreffen beispielsweise die automatische Parameterbestimmung, wie die optimale Größe des Zeitschritts einer Iteration, die Anzahl der durchzuführenden Iterationen und vor allem die Wahl der Diffusivitätsfunktion bzw. deren Para-

meterbestimmung. Vermutlich sind hier die Erstellung und Auswertung vollständiger oder statistisch repräsentativer Bild- und Gradientenhistogramme notwendig, deren effiziente Umsetzung auf der GPU untersucht werden kann. Ebenfalls könnte man untersuchen, wie sich »coherence enhancing diffusion« (CED) oder hybride Diffusionsfilter geschickt auf der GPU umsetzen lassen, da hier zwar einerseits die arithmetische Intensität höher ist, andererseits eventuell Fallunterscheidungen zu behandeln sind, die zum sogenannten »Branching« – also der divergenten Programmausführungen und damit sequentiellen Berechnungen der GPU Multiprozessoren – führen.

Aufbauend auf erfolgreicher Vorfiltertung und dem durchgeführten Diffusionsprozess könnte man aus medizinischer Sicht die Segmentierung in Angriff nehmen. Fragestellungen der GPU-gestützten, parallelen Segmentierung dürften Ansatzpunkte für eine Fülle von weiterreichenden Arbeiten liefern. Im Falle volumetrischer Daten ändert Diffusion ausserdem das von einer Isooberfläche umschlossene Volumen. Ein weiterer medizinisch interessanter Ansatz könnte hier die Entwicklung eines »volumenerhaltenden Diffusionsfilters« sein, ähnlich der volumenerhaltenden Interpolation, z.B. in [Ras07].

Die im Prototyp enthaltenen »skalaren« Diffusionsfilter könnten außerdem weiterentwickelt werden, um vektorielle Datensätze zu verarbeiten, wie sie beispielsweise bei der Erstellung von DW-MRI (diffusion weighted magnetic resonance imaging) Bildern entstehen. Ansatzpunkte hierfür könnten 3D-RGBA-Texturen sein, die dreidimensional angeordnete 4D Vektoren repräsentieren könnten. Im Zuge dessen müssten dann, gerade bei double precision Fließkommaarithmetik, eventuell weitaus größere Datensätze verarbeitet werden, so dass eine einzelne Iteration nicht mehr vollständig im Speicher der Grafikkarte durchgeführt werden kann. Dann müssten Mechanismen entwickelt werden, um Teile des Datensatzes getrennt bearbeiten zu können und die Teilergebnisse in Form von Subvolumen wieder zusammen zu führen. Eine solche »Splitting«-Technik könnte auch genutzt werden, um große Datensätze auf ein Array mit mehreren Grafikkarten zu verteilen und so noch höhere Geschwindigkeiten zu erzielen.

Literaturverzeichnis

[AC08] Fedy Abi-Chahla. Nvidia's cuda: The end of the cpu, hardware point of view. Online / Press, June 2008. http://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954-9.html (Stand: 12.03.2010 05:07).

[ALM92] Luis Alvarez, Pierre-Louis Lions, and Jean-Michel Morel. Image selective smoothing and edge detection by nonlinear diffusion ii. In *SIAM J. Numer. Anal.*, volume 29, pages 845–866. Society for Industrial and Applied Mathematics, Jun 1992.

[AMD10] AMD. Ati stream software development kit (sdk). Online, April 2010. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx (Stand: 10.04.2010 10:30).

[BB05] M. Bender and M. Brill. *Computergrafik: Ein anwendungsorientiertes Lehrbuch*. Hanser Verlag, 2005.

[BC92] D. Bleecker and G. Csordas. *Basic partial differential equations*. Chapman & Hall/-CRC, 1992.

[BK09] William van Winkle Benjamin Kraft. Stream und anwendungen – ein rückblick. Online / Press, July 2009. http://www.tomshardware.de/ATi-Stream-AMD-gpgpu,testberichte-240350-2.html (Stand: 18.01.2010 11:00).

[BM01] B. Brüderlin and A. Meier. *Computergrafik und geometrisches Modellieren*. Vieweg+ Teubner Verlag, 2001.

[BWX] Chandrajit L. Bajaj, Qiu Wu, and Guoliang Xu. Level set based volumetric anisotropic diffusion for image filtering.

[Can84] J.R. Cannon. *The one-dimensional heat equation*. Cambridge Univ Pr, 1984.

[Cor09] Nvidia Corporation. Whitepaper nvidia's next generation cuda compute architecture: Fermi. Online, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (Stand: 12.03.2010 17:00 Uhr).

[Cra80] John Crank. *The Mathematics of Diffusion*. Oxford University Press, USA, 1980.

[Cus09] E. L. Cussler. *Diffusion: Mass Transfer in Fluid Systems*. Cambridge Series in Chemical Engineering. Cambridge University Press, 2009.

[DD02] Fredo Durand and Julie Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Trans. Graph.*, 21(3):257–266, 2002.

[Ese01] S. Esedoglu. An analysis of the perona-malik scheme. In *Communications on Pure and Applied Mathematics.*, volume 54, pages 1442–1487. John Wiley & Sons, Inc., 2001.

[FH99] Achilleas S. Frangakis and Reiner Hegerl. Nonlinear anisotropic diffusion in three-dimensional electron microscopy. In *Scale-Space Theories in Computer Vision*, volume 1682/1999 of *Lecture Notes in Computer Science*, pages 386–397. Springer Berlin / Heidelberg, 1999.

[FLL07] J. J. Fernandez, S. Li, and V. Lucic. Three-dimensional anisotropic noise reduction with automated parameter tuning: Application to electron cryotomography. In Spanish Association for Artificial Intelligence, editor, *Current Topics in Artificial Intelligence: 12th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2007, Salamanca, Spain, November 12-16, 2007. Selected Papers*, pages 60–69. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN: 978-3-540-75270-7.

[Gmb09] Klaß & Ihlenfeld Verlag GmbH. Nvidia gtx-285 erreicht teraflop-grenze mit 55nm-technik. Press, January 2009. http://www.golem.de/0901/64523.html (Stand: 11.03.2010 15:59).

[GR06] C. Großmann and H.G. Roos. *Numerik partieller Differentialgleichungen*. Vieweg+ Teubner Verlag, 2006.

[Har03] T. Hartkens. *Measuring, analysing, and visualising brain deformation using non-rigid registration*. PhD thesis, PhD thesis, King's College London, 2003. Computational Imaging Sciences Group, Division of Imaging Sciences.

[Har05] Mark Harris. Mapping computational concepts to gpus. *ACM SIGGRAPH 2005 Courses (Los Angeles, California, July 31 – August 4, 2005)*, chapter 31:493–508, 2005.

[Har07] M. Harris. Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007.

[HES07] P. U. Thamsen H. E. Siekmann. *Strömungslehre: Grundlagen*. Auflage: 2., aktualisierte Auflage. Springer, Berlin, Oktober 2007. ISBN: 354073726X.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus H. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Comput. Graph. Forum*, 24(3):303–312, 2005.

[HZ00] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526. Citeseer, 2000.

[JLR03] Jun Jiang, Wayne Luk, and Daniel Rueckert. Fpga-based computation of free-form deformations in medical image registration. In Peter Y. K. Cheung, George A. Constantinides, and Jose T. de Sousa, editors, *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, volume 2778 of *Lecture Notes in Computer Science*, pages 1057–1061, 2003. ISBN: 3-540-40822-3.

[Kni04] Oliver Knill. Eigenvalues and eigenvectors of 2x2 matrices. http://www.math.harvard.edu/archive/21b_fall_04/exhibits/2dmatrices/index.html (03.03.2010 - 07:30 Uhr), 2004.

[Krä09] Krämer, M. Swift ions in radiotherapy-Treatment planning with TRiP98. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 267(6):989–992, 2009.

[Kre68] E. Kreyszig. *Introduction to differential geometry and Riemannian geometry*. University of Toronto Press Toronto, 1968.

[Kre91] Erwin Kreyszig. *Differential Geometry*. Dover Pubn Inc; Auflage: New edition, Juni 1991. ISBN-10: 0486667219 / ISBN-13: 978-0486667218.

[KWTM03] G Kindlmann, R Whitaker, T Tasdizen, and T Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of IEEE Visualization 2003*, pages 513–520, October 2003.

[Len01] Eric Lengyel. *Mathematics for 3D Game Programming & Computer Graphics (Game Development Series (Charles River Media).)*. Charles River Media, 2001. ISBN: 1584500379, homogeneous coordinates pp. 62..

[LHG+06] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. GPGPU: general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208. ACM, 2006.

[Lin93] Tony Lindeberg. Discrete derivative approximations with scale-space properties: A basis for low-level feature extraction. *J. of Mathematical Imaging and Vision*, 3:pp. 349 – 376, 1993.

[Lin94] Tony Lindeberg. Scale-space theory: A basic tool for analysing structures at different scales. *Journal of Applied Statistics - Supplement on Advances in Applied Statistics: Statistics and Images*, 21(2):225–270, 1994.

[Lin98] Tony Lindeberg. Feature detection with automatic scale selection. In *International Journal of Computer Vision*, volume 30, pages pp 77 – 116. Department of Numerical Analysis and Computing Science, Royal Institute of Technology, 1998.

[MDSB02] M. Meyer, M. Desbrun, P. Schröder, and A.H. Barr. Discrete differential-geometry operators for triangulated 2-manifolds. *Visualization and mathematics*, 3:35–57, 2002.

[Meh05] Helmut Mehrer. *Diffusion in Condensed Matter - Methods, Materials, Models*. Springer, 2005.

[Mor99] Michael Mortenson. *Mathematics for Computer Graphics Applications*. Industrial Press, Inc., 1999. ISBN: 083113111X, homogeneous coordinates pp. 69..

[MRC97] Mark Maasland, Ronald Rösch, and Bernhard Claus. Diffusionsfilter in der bildverarbeitung. In *at – Automatisierungstechnik*, volume 45, pages 473–479. Oldenbourg Verlag, Oktober 1997. ISSN 0178-2312.

[MUF95] Blair Mackiewich, Simon Fraser University, and Dr. B. Funt. Intracranial boundary detection and radio frequency correction in magnetic resonance images, 1995.

[NDW97] J. Neider, T. Davis, and M. Woo. *OpenGL. Programming guide*. Addison-Wesley Reading, MA, 1997.

[Nvi07] Nvidia. *The CUDA Compiler Driver NVCC*. Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, v. 1.1 edition, 05 2007.

[Nvi08] Nvidia. *NVIDIA CUDA Programming Guide 2.0*. Nvidia Corporation, version 2.0 edition, 07 2008.

[Nvi09a] Nvidia. *NVIDIA CUDA C Programming Best Practices Guide*. Nvidia Corporation, July 2009.

[Nvi09b] Nvidia. *NVIDIA CUDA Reference Manual*. Nvidia Corporation, version 2.3 edition, July 2009.

[Nvi09c] Nvidia. Presseartikel. NVIDIA kündigt neue CUDA GPU Architektur mit Codenamen »Fermi« an. Press, Oktober 2009. http://www.nvidia.de/object/io_1254418018894.html (Stand: 10.03.2010 21:10).

[Nvi10] Nvidia. *NVIDIA CUDA Programming Guide 3.0*. Nvidia Corporation, version 3.0 edition, 09 2010.

[Ped10] Jon Peddie. Astounding year-to-year growth in pc graphics; quarter-to-quarter also beats expectations. Technical report, Jon Peddie Research, 2010. (10.03.2010, 17:13).

[Pha05] Matt Pharr, editor. *GPU Gems 2*. Addison-Wesley Longman, Amsterdam; Auflage: Har/Cdr (17.03.2005), 2005. ISBN: 0321335597.

[PM90] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:629–639, 1990.

[Pol93] D.S.G. Pollock. Smoothing with cubic splines, 1993. http://webspace.qmul.ac.uk/dsgpollock/public_html/PAPERS/SPLINES.PDF.

[PR02] T. Preußer and M. Rumpf. A level set method for anisotropic geometric diffusion in 3d image processing. *SIAM Journal on Applied Mathematics*, 62(5):1772–1793, 2002.

[Ras07] Wolf-Dieter Rase. Volumenerhaltende Interpolation aus polygonbezogenen Daten in einem unregelmäßigen Dreiecksnetz (TIN). Technical report, Bundesamt für Bauwesen und Raumordnung, Bonn, 2007.

[RtHRS08] Daniel Ruijters, Bart M. ter Haar Romeny, and Paul Suetens. Efficient gpu-based texture interpolation using uniform b-splines. *journal of graphics, gpu, and game tools*, 13(4):61–69, 2008.

[SC07] J. Steele and R. Cochran. Introduction to GPGPU programming. In *Proceedings of the 45th annual southeast regional conference*, page 508. ACM, 2007.

[Sig06] Christian Sigg. *Representation and Rendering of Implicit Surfaces*. PhD thesis, ETH Zurich, 2006.

[SPD07] Jack Tumblin Sylvain Paris, Pierre Kornprobst and Frédo Durand. A gentle introduction to bilateral filtering and its applications. A course at ACM SIGGRAPH 2007, August 6th 2007. Course Materials available: http://people.csail.mit.edu/sparis/bf_course/ (18. Feb. 2010).

[ST06] I. Garcia J.J. Fernandez S. Tabik, E.M. Garzon. Implementation of anisotropic nonlinear diffusion for filtering 3d images in structural biology on smp clusters. In F.J. Peters O. Plata P. Tirado E. Zapata G.R. Joubert, W.E. Nagel, editor, *Parallel Computing: Current & Future Issues of High-End Computing,*, volume 33 of *Proceedings of the International Conference ParCo 2005*, pages 727–734. John von Neumann Institute for Computing, Jülich, 2006.

[TM98] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of IEEE International Conference on Computer Vision '98*, pages 839–846, January 1998.

[Uns99] M. Unser. Splines: A perfect fit for signal and image processing. *IEEE Signal Processing Magazine*, 16(6):22–38, November 1999. IEEE Signal Processing Society's 2000 magazine award.

[Wei96] Joachim Weickert. *Anisotropic Diffusion in Image Processing*. Dissertation zur Erlangung des akademischen Grades Doktor der Naturwissenschaften, Universität Kaiserslautern, 01 1996.

[Wei97] Joachim Weickert. A review of nonlinear diffusion filtering. In *Scale-Space Theory in Computer Vision*, volume 1252/1997 of *Lecture Notes in Computer Science*, pages 1–28. Springer Berlin / Heidelberg, 1997. ISBN: 978-3-540-63167-5; ISSN 0302-9743 (Print) 1611-3349 (Online).

[Wei98] Joachim Weickert. *Anisotropic Diffusion in Image Processing*. B.G. Teubner Stuttgart, 1998.

[Wei99] Joachim Weickert. Coherence enhancing diffusion filtering. *International Journal of Computer Vision*, 31:111–127, 1999.

[Wel07] Martin Welk. Mathematik für Informatiker II. (Skriptum zur Vorlesung), 2007.

[Wit83] Andrew P. Witkin. Scale-space filtering. In *Proc. 8th Int. Joint Conf. Art. Intell.*, pages 1019–1022, Karlsruhe, West Germany, 1983.

Glossar

Anisotropie
: Anisotropie ist die Eigenschaft eines Vorganges richtungsabhängig zu sein.

Arithmetische Intensität
: Die arithmetische Intensität ist das Verhältnis von Berechnungen (meist analog: Fließkommaoperationen) zu Speicherzugriffen.

Bildverarbeitung
: Die Bildverarbeitung umfasst alle Schritte und Techniken die benötigt werden, um mit Hilfe mathematischer Methoden und mit Erkenntnissen der Signalverarbeitung Informationen aus Bildern zu gewinnen.

Block
: Ein Block ist eine Thread-Hierarchieebene im CUDA Programmiermodell. Ein Block fasst mehrere Kernel zusammen, die alle auf einem Multiprozessor der GPU ablaufen und gemeinsamen Speicher nutzen können.

C für CUDA
: C für CUDA ist eine Erweiterung der Sprache C um Sprachelemente der CUDA Technologie. Zur Kompilierung wird der NVCC, ein spezieller Compiler der Firma Nvidia benötigt, der den Quellcode in CPU und GPU Code auftrennt und separat kompiliert.

CUDA
: CUDA, die Compute Unified Device Architecture, ist eine Technologie der Firma Nvidia, um auf die Multiprozessoren moderner Grafikkarten zugreifen zu können.

Device
: Als Device wird in der GPU Programmierung die Grafikkarte mit ihren Multiprozessoren und dem Grafikkartenspeicher verstanden.

Diffusion
: Diffusion ist ein Prozess, der Konzentrationsunterschiede mit fortschreitender Zeit ausgleicht.

Diskretisierung
: Unter der Diskretisierung versteht man im Rahmen der numerischen Lösung von Differentialgleichungen eine räumliche und zeitliche Zerlegung des Problems in endlich viele Teile, welche die kontinuierliche Lösung mit Differenzen approximieren.

Divergenz

Die Divergenz ist ein Skalar, der die Summe der Richtungsableitungen eines Vektors darstellt. Mathematisch ist die Divergenz ein Operator, der aus einem Vektorfeld ein Skalarfeld erzeugt.

Eigenwertzerlegung

Die Eigenwertzerlegung liefert die Eigenwerte einer Abbildung. Eigenwerte sind Vektoren, die durch die Abbildung nicht in ihrer Richtung, sondern lediglich in ihrer Länge verändert werden.

Filter

Ein Filter ist in der Bildverarbeitung eine in der Regel mathematisch formulierte Abbildungsforschrift, die ein Quellbild in ein Zielbild überführt und in der Regel nicht umkehrbar ist.

global memory

Global memory ist im CUDA Speichermodell der gesamte Grafikkartenspeicher.

Gradient

Der Gradient ist ein Vektor, der die Richtung des steilsten Anstiegs eines Skalarfeldes anzeigt. Mathematisch ist der Gradient ein Operator, der aus einem Skalarfeld ein Vektorfeld erzeugt.

Grid

Ein Grid ist eine Thread-Hierarchieebene im CUDA Programmiermodell. Ein Grid fasst mehrere Blöcke zusammen, die benötigt werden, um eine Aufgabe zu bewältigen.

Hesse-Matrix

Die Hesse-Matrix enthält als Komponenten alle Kombinationen der Richtungsableitungen einer mehrdimensionalen Funktion.

Host

Als Host wird in der GPU Programmierung die Wirtsarchitektur, im engeren Sinne die CPU und der Hauptspeicher, verstanden.

Isooberfläche

Die Isooberfläche ist die Oberfläche eines Volumendatensatzes, die sich bildet, wenn man einen konstanten Volumenwert betrachtet. Diskrete, vom Rauschen beeinflusste Abtastungen eines Datensatzes liefern in der Regel nicht zusammenhängende Isooberflächen mit Löchern.

Kernel
: Ein Kernel ist der einzelne Thread, der auf den Multiprozessoren der GPU abläuft.

Konzentration
: Die Konzentration ist eine Angabe über die Menge eines Stoffes oder Stoffgemisches in einem Gesamtvolumen.

Shader
: Shader bezeichnen kleine Programme und Hardwarebausteine, welche Geometrie- und Lichtberechnungen in der Computergrafik ausführen. Moderne Hardwareshader sind mittlerweile frei programmierbare Multiprozessoren.

shared memory
: Shared memory ist im CUDA Speichermodell Speicher, der allen Kernels eines Blocks zur gemeinsamen Nutzung zur Verfügung steht.

Tangentialraum
: Die Tangentialebene ist das dreidimensionale Pendant zur Tangente einer Funktion im zweidimensionalen Raum. Der durch die Tangentialebene aufgespannte Raum wird als Tangentialraum bezeichnet. Zwei beliebige nicht linearabhängige Vektoren der Ebene bilden eine Basis des Tangentialraumes.

Tensor
: Ein Tensor ist eine indizierbare Größe. Ein Skalar kann als Tensor 0-ter Stufe aufgefasst werden. Tensoren 2-ter Stufe haben zwei Indizes und können als Matrix dargestellt werden. Tensoren können oft als Verallgemeinerung eines Koeffizienten auf höhere Dimensionen angesehen werden.

texture memory
: Texture memory ist im CUDA Speichermodell Grafikkartenspeicher, der als Texturspeicher ausgewiesen wird und für den damit Hardwarecaching aktiviert worden ist. Der texture memory ist ein read-only Speicher.

Transformation
: Eine Transformation ist in der Bildverarbeitung eine in der Regel mathematisch formulierte mathematische Abbildungsforschrift, die ein Quellbild in ein Zielbild überführt. Im Gegensatz zum Filter, ist die Transformation meist geometrisch motiviert und insbesondere umkehrbar.

Abkürzungsverzeichnis

CED	Coherence enhancing diffusion, ein anisotroper Diffusionsfilter, der linienartige Strukturen verstärkt
CPU	Central Processing Unit, der Hauptprozessor eines Computers
CUDA	Compute Unified Device Architecture
EED	Edge enhancing diffusion, ein anisotroper Diffusionsfilter, der kantenerhaltend arbeitet
FLOPS	Floatingpoint operations per second, eine Maßzahl für die Geschwindigkeit eines Prozessors
GPGPU	General-purpose Computation on Graphics Processing Units
GPU	Graphics Processing Unit, der Prozessor einer Grafikkarte
IEEE	Institute of Electrical and Electronics Engineers, einer der größten technischen Berufsverbände der Welt. Die Publikationen haben einen hohen fachlichen Stellenwert und die verabschiedeten Standards werden in der Regel zur Industrienorm.
LUT	Lookup Tables, eine Technik zur Beschleunigung von Berechnungen: Vorberechnete Werte (zum Beispiel Sinuswerte) werden für eine Vielzahl von möglichen Parametern (zum Beispiel 3600 Zehntelgradschritte) vorberechnet und auf Abruf in einer Tabelle bereitgehalten.
MP	Multiprozessor, moderne GPUs bestehen aus mehreren hundert Multiprozessoren

NVCC Der NVCC ist ein Compilerdriver, der Host- und Devicecode aus C für CUDA Quelltexten extrahiert und an die entsprechenden Compiler delegiert.

PDE Partial Differential Equation, deutsch: Partielle Differentialgleichung. Ein Differentialgleichung, deren Formulierung partielle Ableitungen enthält. Differentialgleichungen sind Gleichungen, deren Lösung eine gesuchte Funktion ist, die über Abhängigkeiten zu Variablen und Ableitungen der gesuchten Funktion definiert ist. Die einfache Differentialgleichung $f = -f''$ beschreibt beispielsweise eine Funktion f, deren zweite Ableitung die negierte Funktion selbst ist. Eine Lösung ist beispielsweise die Sinus-Funktion. Da eine zweite Ableitung zur Definition genutzt wurde, handelt es sich um eine Differentialgleichung zweiter Ordnung. Partielle Ableitungen sind Ableitungen von Funktionen mit mehreren Funktionsargumenten nach genau einem Argument.

SIMD Single Instruction Multiple Data, ein Prozessordesign, das parallele Verarbeitung von Datenströmen erlaubt. Eine Operation wird dabei auf mehreren Daten gleichzeitig ausgeführt.

SP Streamingprozessor. Jeder MP einer Nvidia GPU besteht aus 8 Streamingprozessoren. Klassisch sind diese als Shading Units bekannt. Heutzutage sind sie frei programmierbare SIMD Prozessoren.

VAD Volumetrische anisotrope Diffusion

Symbolverzeichnis

D	Der Diffusionskoeffizient bzw. der Diffusionstensor. $D(x,t)$ ist ein Tensorfeld, welches an jedem Ort x zu jedem Zeitpunkt t den lokalen Diffusionstensor D angibt.
Δ	Δ markiert in der Regel einen diskreten Raum- oder Zeitschritt (Δt für Zeitschritte bzw. Δx, Δy, Δz für Raumschritte.)
g_λ	Die Diffusivitätsfunktion g_λ ist eine monoton fallende reelle Funktion. Sie erhält in der Regel ein Argument aus $\mathbb{R}^+$ und bildet meist auch auf $\mathbb{R}^+$ ab.
$\vec{j}_D$	Der Diffusionsfluss. $\vec{j}_D(x,t)$ ist ein Vektorfeld, welches, gesteuert vom Diffusionstensor D, an jedem Ort x zu jedem Zeitpunkt t die Richtung und die Größe des Diffusionsflusses angibt.
λ	Freier Parameter der Diffusivitätsfunktion. Steuert in der Regel das Ansprechverhalten auf Rauschen.
∇	Bezeichnet den formalen Nablaoperator und somit in dieser Arbeit kontextabhängig einen der beiden Differentialoperatoren »Gradient« oder »Divergenz«. Es gilt $\nabla\Phi = grad(\Phi)$ und $\nabla\nabla\Phi = \Delta\Phi = div(grad(\Phi))$.
ϕ	Die Bildfunktion. $\phi(x,t) : \mathbb{R}^n \mapsto \mathbb{R}$ ist ein Skalarfeld, das jedem Ort x zu jedem Zeitpunkt t einen reellen Wert zuordnet.
$\widehat{\Phi}_{x_+}$	Die Kurzform des in x-Richtung nächsten diskret vorliegenden Bildwertes: $\widehat{\Phi}_{x_+} = \widehat{\Phi}_{x+1,y,z}$. Analoge Bezeichnungen gelten in y und z Richtung.
ϕ_0	Das ursprüngliche Bild zum Startzeitpunkt. $\phi_0(x) = \phi(x,0)$.
$\widehat{\Phi}$	Die diskretisiert vorliegende Form der Bildfunktion $\Phi : \Omega \subset \mathbb{R}^3 \mapsto \mathbb{R}$.

T_Ω	Bezeichnet eine affine Transformation mit Hilfe homogener Koordinaten, ist also eine 4×4 Matrix.
τ	Der griechische Buchstabe τ bezeichnet den Zeitschritt, der während einer Iteration des diskretisierten Diffusionsfilters berechnet wird.
x oder $\vec{x}$	Der Ort bzw. ein Ortsvektor mit (in der vorliegenden Arbeit) drei Komponenten.

A Pseudocode EED

Listing 1: Pseudocode EED

```
// c like pseudo code. the gridvalues of the input and
// output arrays are stored linear in x, y, z order
// Input:  pfIn contains the input data
//         iDimX, iDimY, iDimZ: dimensions of pfIn/pfOut
//         fDeltaT: the timestep
//         fNoiseGrad: diffusivity function g parameter
// Output: pfOut (array with same dimensions as pfIn)

// the outer loop determines the voxel which we are
// processing at the moment.
for (iOZ = 0; iOZ < iDimZ; iOZ++) {
for (iOY = 0; iOY < iDimY; iOY++) {
for (iOX = 0; iOX < iDimX; iOX++) {

  // we have to evaluate 6 points for the 3 partial derivatives
  // approximated by central differences at the end:
  fXHigh = 0;
  fXLow  = 0;
  fYHigh = 0;
  fYLow  = 0;
  fZHigh = 0;
  fZLow  = 0;

  // this is why we do the same algorithm 6 times again with
  // slightly different iX, iY, iZ values:
  for (iLoop = 0; iLoop < 6; iLoop++) {

    switch (iLoop){
      case 5:
       iX = iOX; iY = iOY; iZ = iOZ+1;
      break;
      case 4:
       iX = iOX; iY = iOY; iZ = iOZ-1;
      break;
      case 3:
       iX = iOX; iY = iOY+1; iZ = iOZ;
      break;
      case 2:
       iX = iOX; iY = iOY-1; iZ = iOZ;
      break;
      case 1:
```

```
42        iX = iOX+1; iY = iOY; iZ = iOZ;
43      break;
44      default:
45        iX = iOX-1; iY = iOY; iZ = iOZ;
46      break;
47    }
48
49    iActIndex      = iX + iY * iDimX + iZ * iDimX * iDimY;
50    fActVoxelValue = pfIn[iActIndex];
51
52    // ensure, we are inside a region where we have neighbor
53    // voxels... otherwise skip this iteration
54    if ((iX == 0) || (iX == iDimX-1) ||
55        (iY == 0) || (iY == iDimY-1) ||
56        (iZ == 0) || (iZ == iDimZ-1)) {
57      continue;
58    }
59
60    // STEP 1 - Build gradient:
61    fGradX   = (pfIn[(iX+1) + iY*iDimX + iZ*iDimX*iDimY] -
62                pfIn[(iX-1) + iY*iDimX + iZ*iDimX*iDimY]) / 2.0;
63    fGradY   = (pfIn[iX + (iY+1)*iDimX + iZ*iDimX*iDimY] -
64                pfIn[iX + (iY-1)*iDimX + iZ*iDimX*iDimY]) / 2.0;
65    fGradZ   = (pfIn[iX + iY*iDimX + (iZ+1)*iDimX*iDimY] -
66                pfIn[iX + iY*iDimX + (iZ-1)*iDimX*iDimY]) / 2.0;
67
68    // STEP 2 - Build Isosurface normal:
69    fGradLength  = sqrt(fGradX^2 + fGradY^2 + fGradZ^2);
70    if (fGradLength < EPSILON) {
71      continue;
72    }
73
74    fNormX = - fGradX / fGradLength;
75    fNormY = - fGradY / fGradLength;
76    fNormZ = - fGradZ / fGradLength;
77
78    //STEP 3 - Build Hessian:
79    fHXX = (   pfIn[(iX+1) + iY * iDimX + iZ * iDimX*iDimY] -
80            2*pfIn[ iX     + iY * iDimX + iZ * iDimX*iDimY] +
81              pfIn[(iX-1) + iY * iDimX + iZ * iDimX*iDimY] );
82
83    fHYY = (   pfIn[iX + (iY+1) * iDimX + iZ * iDimX*iDimY] -
84            2*pfIn[iX +  iY     * iDimX + iZ * iDimX*iDimY] +
85              pfIn[iX + (iY-1) * iDimX + iZ * iDimX*iDimY] );
86
```

```
87      fHZZ = (   pfIn[iX + iY * iDimX + (iZ+1) * iDimX*iDimY] -
88               2*pfIn[iX + iY * iDimX +  iZ    * iDimX*iDimY] +
89                 pfIn[iX + iY * iDimX + (iZ-1) * iDimX*iDimY] );
90      fHXY =
91        (((pfIn[(iX+1) + (iY+1) * iDimX + iZ * iDimX*iDimY] +
92           pfIn[(iX+1) + (iY  ) * iDimX + iZ * iDimX*iDimY] +
93           pfIn[(iX  ) + (iY+1) * iDimX + iZ * iDimX*iDimY] +
94           pfIn[(iX  ) + (iY  ) * iDimX + iZ * iDimX*iDimY])/4.0)-
95         ((pfIn[(iX-1) + (iY+1) * iDimX + iZ * iDimX*iDimY] +
96           pfIn[(iX-1) + (iY  ) * iDimX + iZ * iDimX*iDimY] +
97           pfIn[(iX  ) + (iY+1) * iDimX + iZ * iDimX*iDimY] +
98           pfIn[(iX  ) + (iY  ) * iDimX + iZ * iDimX*iDimY])/4.0))-
99        (((pfIn[(iX+1) + (iY-1) * iDimX + iZ * iDimX*iDimY] +
100          pfIn[(iX+1) + (iY  ) * iDimX + iZ * iDimX*iDimY] +
101          pfIn[(iX  ) + (iY-1) * iDimX + iZ * iDimX*iDimY] +
102          pfIn[(iX  ) + (iY  ) * iDimX + iZ * iDimX*iDimY])/4.0)-
103        ((pfIn[(iX-1) + (iY-1) * iDimX + iZ * iDimX*iDimY] +
104          pfIn[(iX-1) + (iY  ) * iDimX + iZ * iDimX*iDimY] +
105          pfIn[(iX  ) + (iY-1) * iDimX + iZ * iDimX*iDimY] +
106          pfIn[(iX  ) + (iY  ) * iDimX + iZ * iDimX*iDimY])/4.0));
107
108     cfHXZ =
109        (((pfIn[(iX+1) + iY * iDimX + (iZ+1) * iDimX*iDimY] +
110           pfIn[(iX+1) + iY * iDimX + (iZ  ) * iDimX*iDimY] +
111           pfIn[(iX  ) + iY * iDimX + (iZ+1) * iDimX*iDimY] +
112           pfIn[(iX  ) + iY * iDimX + (iZ  ) * iDimX*iDimY])/4.0)-
113         ((pfIn[(iX-1) + iY * iDimX + (iZ+1) * iDimX*iDimY] +
114           pfIn[(iX-1) + iY * iDimX + (iZ  ) * iDimX*iDimY] +
115           pfIn[(iX  ) + iY * iDimX + (iZ+1) * iDimX*iDimY] +
116           pfIn[(iX  ) + iY * iDimX + (iZ  ) * iDimX*iDimY])/4.0))-
117        (((pfIn[(iX+1) + iY * iDimX + (iZ-1) * iDimX*iDimY] +
118           pfIn[(iX+1) + iY * iDimX + (iZ  ) * iDimX*iDimY] +
119           pfIn[(iX  ) + iY * iDimX + (iZ-1) * iDimX*iDimY] +
120           pfIn[(iX  ) + iY * iDimX + (iZ  ) * iDimX*iDimY])/4.0)-
121         ((pfIn[(iX-1) + iY * iDimX + (iZ-1) * iDimX*iDimY] +
122           pfIn[(iX-1) + iY * iDimX + (iZ  ) * iDimX*iDimY] +
123           pfIn[(iX  ) + iY * iDimX + (iZ-1) * iDimX*iDimY] +
124           pfIn[(iX  ) + iY * iDimX + (iZ  ) * iDimX*iDimY])/4.0));
125
126     cfHYZ =
127        (((pfIn[iX + (iY+1) * iDimX + (iZ+1) * iDimX*iDimY] +
128           pfIn[iX + (iY+1) * iDimX + (iZ  ) * iDimX*iDimY] +
129           pfIn[iX + (iY  ) * iDimX + (iZ+1) * iDimX*iDimY] +
130           pfIn[iX + (iY  ) * iDimX + (iZ  ) * iDimX*iDimY])/4.0)-
131         ((pfIn[iX + (iY-1) * iDimX + (iZ+1) * iDimX*iDimY] +
```

```
132           pfIn[iX + (iY-1) * iDimX + (iZ  ) * iDimX*iDimY] +
133           pfIn[iX + (iY  ) * iDimX + (iZ+1) * iDimX*iDimY] +
134           pfIn[iX + (iY  ) * iDimX + (iZ  ) * iDimX*iDimY])/4.0))-
135        (((pfIn[iX + (iY+1) * iDimX + (iZ-1) * iDimX*iDimY] +
136           pfIn[iX + (iY+1) * iDimX + (iZ  ) * iDimX*iDimY] +
137           pfIn[iX + (iY  ) * iDimX + (iZ-1) * iDimX*iDimY] +
138           pfIn[iX + (iY  ) * iDimX + (iZ  ) * iDimX*iDimY])/4.0)-
139         ((pfIn[iX + (iY-1) * iDimX + (iZ-1) * iDimX*iDimY] +
140           pfIn[iX + (iY-1) * iDimX + (iZ  ) * iDimX*iDimY] +
141           pfIn[iX + (iY  ) * iDimX + (iZ-1) * iDimX*iDimY] +
142           pfIn[iX + (iY  ) * iDimX + (iZ  ) * iDimX*iDimY])/4.0));
143
144     // STEP 4 - Find orthonormal basis:
145     fUX = 0;
146     fUY = fNormZ;
147     fUZ = -fNormY;
148     fTmp = sqrt(fUX^2 + fUY^2 + fUZ^2);
149     if (fTmp < EPSILON) {
150       fTmp = sqrt(fUX^2 + fUY^2 + fUZ^2);
151       fUX  = -fNormZ;
152       fUY  = 0;
153       fUZ  = fNormX;
154     }
155     fUX /= fTmp;
156     fUY /= fTmp;
157     fUZ /= fTmp;
158
159     fVX   =   fNormY*fUZ - fNormZ*fUY;
160     fVY   = - fNormX*fUZ + fNormZ*fUX;
161     fVZ   =   fNormX*fUY - fNormY*fUX;
162     fTmp  = sqrt(fVX^2 + fVY^2 + fVZ^2);
163     fVX  /= fTmp;
164     fVY  /= fTmp;
165     fVZ  /= fTmp;
166
167     // STEP 5 - Projection H[3x3] -> S[2x2]
168     fS11 = ( fUX * (fHXX * fUX + fHXY * fUY + fHXZ * fUZ) +
169               fUY * (fHXY * fUX + fHYY * fUY + fHYZ * fUZ) +
170               fUZ * (fHXZ * fUX + fHYZ * fUY + fHZZ * fUZ) ) /
171               fGradLength;
172     fS12 = ( fUX * (fHXX * fVX + fHXY * fVY + fHXZ * fVZ) +
173               fUY * (fHXY * fVX + fHYY * fVY + fHYZ * fVZ) +
174               fUZ * (fHXZ * fVX + fHYZ * fVY + fHZZ * fVZ) ) /
175               fGradLength;
176     fS21 = ( fVX * (fHXX * fUX + fHXY * fUY + fHXZ * fUZ) +
```

```
177             fVY * (fHXY * fUX + fHYY * fUY + fHYZ * fUZ) +
178             fVZ * (fHXZ * fUX + fHYZ * fUY + fHZZ * fUZ) ) /
179             fGradLength;
180     fS22 = ( fVX * (fHXX * fVX + fHXY * fVY + fHXZ * fVZ) +
181             fVY * (fHXY * fVX + fHYY * fVY + fHYZ * fVZ) +
182             fVZ * (fHXZ * fVX + fHYZ * fVY + fHZZ * fVZ) ) /
183             fGradLength;
184
185     // STEP 6 - Retrieve eigenvalues kappa1/2
186     fKapSqrt = ((fS11+fS22)^2 / 4.0) - (fS11*fS22 - fS12*fS21);
187     if (fKapSqrt > EPSILON)
188       fKapSqrt  = sqrt(fKapSqrt);
189     else
190       fKapSqrt = 0.0;
191     fKappa1  = (fS11 + fS22) / 2.0 - fKapSqrt;
192     fKappa2  = (fS11 + fS22) / 2.0 + fKapSqrt;
193
194     // STEP 7 - Retrieve eigenvectors ev1/2
195     if (fabs(fS12) < EPSILON) {
196       fEV1X = fUX; fEV1Y = fUY; fEV1Z = fUZ;
197       fEV2X = fVX; fEV2Y = fVY; fEV2Z = fVZ;
198     } else {
199       fEV1X = (fUX * (fKappa1 - fS22)) + (fVX * fS21);
200       fEV1Y = (fUY * (fKappa1 - fS22)) + (fVY * fS21);
201       fEV1Z = (fUZ * (fKappa1 - fS22)) + (fVZ * fS21);
202       fEV2X = (fUX * (fKappa2 - fS22)) + (fVX * fS21);
203       fEV2Y = (fUY * (fKappa2 - fS22)) + (fVY * fS21);
204       fEV2Z = (fUZ * (fKappa2 - fS22)) + (fVZ * fS21);
205     }
206
207     // STEP 8 - Ensure ew1 < ew2
208     if (fKappa2 < fKappa1) {
209       swap(fKappa1, fKappa2);
210       swap(fEV1X, fEV2X);
211       swap(fEV1Y, fEV2Y);
212       swap(fEV1Z, fEV2Z);
213     }
214
215     // STEP 9 - Define V with normalized eigenvectors
216     fTmp = sqrt(fEV1X^2 + fEV1Y^2 + fEV1Z^2);
217     if (fTmp > EPSILON)
218       { fEV1X /= fTmp; fEV1Y /= fTmp; fEV1Z /= fTmp; }
219
220     fTmp = sqrt(fEV2X^2 + fEV2Y^2 + fEV2Z^2);
221     if (fTmp > EPSILON)
```

```
222       { fEV2X /= fTmp; fEV2Y /= fTmp; fEV2Z /= fTmp; }
223
224     fV11 = fEV1X;      fV12 = fEV2X;      fV13 = fNormX;
225     fV21 = fEV1Y;      fV22 = fEV2Y;      fV23 = fNormY;
226     fV31 = fEV1Z;      fV32 = fEV2Z;      fV33 = fNormZ;
227
228     // STEP 10 - Define D
229     fInhibit = 1.0 / (1.0 + pow(fGradLength/fNoiseGrad, 2.0));
230
231     fD11 = fV11*fV11 + fV12*fV12 + fV13*fV13 * fInhibit;
232     fD12 = fV11*fV21 + fV12*fV22 + fV13*fV23 * fInhibit;
233     fD13 = fV11*fV31 + fV12*fV32 + fV13*fV33 * fInhibit;
234     fD21 = fV21*fV11 + fV22*fV12 + fV23*fV13 * fInhibit;
235     fD22 = fV21*fV21 + fV22*fV22 + fV23*fV23 * fInhibit;
236     fD23 = fV21*fV31 + fV22*fV32 + fV23*fV33 * fInhibit;
237     fD31 = fV31*fV11 + fV32*fV12 + fV33*fV13 * fInhibit;
238     fD32 = fV31*fV21 + fV32*fV22 + fV33*fV23 * fInhibit;
239     fD33 = fV31*fV31 + fV32*fV32 + fV33*fV33 * fInhibit;
240
241     switch (iLoop){
242       case 5:
243         fZHigh = (fD13*fGradX + fD23*fGradY + fD33*fGradZ);
244       break;
245       case 4:
246         fZLow  = (fD13*fGradX + fD23*fGradY + fD33*fGradZ);
247       break;
248       case 3:
249         fYHigh = (fD12*fGradX + fD22*fGradY + fD23*fGradZ);
250       break;
251       case 2:
252         fYLow  = (fD12*fGradX + fD22*fGradY + fD23*fGradZ);
253       break;
254       case 1:
255         fXHigh = (fD11*fGradX + fD12*fGradY + fD13*fGradZ);
256       break;
257       default:
258         fXLow  = (fD11*fGradX + fD12*fGradY + fD13*fGradZ);
259       break;
260     }
261   }
262     //STEP 11 - Evaluate flow:
263     fFlow  = ( fXHigh - fXLow ) / 2.0 +
264              ( fYHigh - fYLow ) / 2.0 +
265              ( fZHigh - fZLow ) / 2.0;
266
```

```
267     // STEP 12. Evaluate output...
268     pfOut[iActIndex] = fActVoxelValue + fDeltaT * fFlow;
269 }
270 }
271 }
```

B Herleitung der affinen Transformationsmatrix

$$\begin{pmatrix} x^* \\ y^* \\ z^* \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_{X2} \\ 0 & 1 & 0 & t_{Y2} \\ 0 & 0 & 1 & t_{Z2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \underbrace{\begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{T_{Rot}} \cdot \begin{pmatrix} 1 & 0 & 0 & t_{X1} \\ 0 & 1 & 0 & t_{Y1} \\ 0 & 0 & 1 & t_{Z1} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{B.1}$$

$$= \underbrace{\begin{pmatrix} 1 & 0 & 0 & t_{X2} \\ 0 & 1 & 0 & t_{Y2} \\ 0 & 0 & 1 & t_{Z2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & 0 \\ \cos\beta\sin\gamma & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & 0 \\ -\sin\beta & \sin\alpha\cos\beta & \cos\alpha\cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & t_{X1} \\ 0 & 1 & 0 & t_{Y1} \\ 0 & 0 & 1 & t_{Z1} \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{T_\Omega} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\beta\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \swarrow \\ & \rightarrow \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & (\cos\beta\cos\gamma)t_{X1} + (\sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma)t_{Y1} + (\cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma)t_{Z1} + t_{X2} \\ \cos\beta\sin\gamma & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \swarrow \\ & \rightarrow \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & (\cos\beta\sin\gamma)t_{X1} + (\sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma)t_{Y1} + (\cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma)t_{Z1} + t_{Y2} \\ -\sin\beta & \sin\alpha\cos\beta & \swarrow \\ & \rightarrow \cos\alpha\cos\beta & (-\sin\beta)t_{X1} + (\sin\alpha\cos\beta)t_{Y1} + (\cos\alpha\cos\beta)t_{Z1} + t_{Z2} \\ 0 & 0 & \swarrow \\ & \rightarrow 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Mit den folgenden Definitionen verkürzt sich die Schreibweise erheblich:

$$r_1 = \cos\beta\cos\gamma \qquad r_2 = \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \qquad r_3 = \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \qquad r_4 = (\cos\beta\cos\gamma)t_{X1} + (\sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma)t_{Y1} + (\cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma)t_{Z1} + t_{X2}$$

$$r_5 = \cos\beta\sin\gamma \qquad r_6 = \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma \qquad r_7 = \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma \qquad r_8 = (\cos\beta\sin\gamma)t_{X1} + (\sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma)t_{Y1} + (\cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma)t_{Z1} + t_{Y2}$$

$$r_9 = -\sin\beta \qquad r_{10} = \sin\alpha\cos\beta \qquad r_{11} = \cos\alpha\cos\beta \qquad r_{12} = (-\sin\beta)t_{X1} + (\sin\alpha\cos\beta)t_{Y1} + (\cos\alpha\cos\beta)t_{Z1} + t_{Z2}$$

Wir erhalten die folgenden geschlossenen Formeln für den transformierten Punkt $P^* = (x^*, y^*, z^*)^T$:

$$x^* = r_1 \cdot x + r_2 \cdot y + r_3 \cdot z + r_4 \tag{B.2}$$

$$y^* = r_5 \cdot x + r_6 \cdot y + r_7 \cdot z + r_8 \tag{B.3}$$

$$z^* = r_9 \cdot x + r_{10} \cdot y + r_{11} \cdot z + r_{12} \tag{B.4}$$

C Messergebnisse

C.1 ID Filter

Die Messung des ID Filters:

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	0,22	0,22	1,0	1,0
64	gpu	1,61	0,27	1,88	0,8	0,1
64	tex	7,01	0,28	7,29	0,8	0,0
128	cpu	0,00	4,97	4,97	1,0	1,0
128	gpu	7,88	1,55	9,43	3,2	0,5
128	tex	8,09	1,61	9,70	3,1	0,5
192	cpu	0,00	14,92	14,92	1,0	1,0
192	gpu	22,48	5,03	27,51	3,0	0,5
192	tex	22,77	5,24	28,00	2,8	0,5
256	cpu	0,00	34,57	34,57	1,0	1,0
256	gpu	50,84	11,84	62,68	2,9	0,6
256	tex	51,41	12,17	63,59	2,8	0,5
320	cpu	0,00	65,14	65,14	1,0	1,0
320	gpu	97,43	23,01	120,44	2,8	0,5
320	tex	98,85	23,59	122,44	2,8	0,5

Tabelle 20: Messergebnisse »ID Filter«

C.2 Prefilter

Es folgen die Messungen für die verschiedenen Prefilter:

- Boxfilter 1 ($3 \times 3 \times 3$ Kernel)
- Boxfilter 2 ($5 \times 5 \times 5$ Kernel)
- Boxfilter 1, separiert ($3 \times 3 \times 3$ Kernel)
- Boxfilter 2, separiert ($5 \times 5 \times 5$ Kernel)
- Gaußfilter 1 ($3 \times 3 \times 3$ Kernel)
- Gaußfilter 1, separiert ($3 \times 3 \times 3$ Kernel)
- Gaußfilter 2, separiert ($5 \times 5 \times 5$ Kernel)
- Medianfilter ($3 \times 3 \times 3$ Nachbarschaft)
- Bilateraler Filter ($3 \times 3 \times 3$ Kernel)

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	21,02	21,02	1,0	1,0
64	gpu	1,86	3,86	5,72	5,4	3,7
64	tex	1,98	1,25	3,22	16,9	6,5
128	cpu	0,00	170,72	170,72	1,0	1,0
128	gpu	7,86	31,28	39,14	5,5	4,4
128	tex	8,08	9,60	17,69	17,8	9,7
192	cpu	0,00	581,04	581,04	1,0	1,0
192	gpu	22,45	98,34	120,80	5,9	4,8
192	tex	26,22	32,37	58,59	17,9	9,9
256	cpu	0,00	1.372,31	1.372,31	1,0	1,0
256	gpu	51,81	226,40	278,20	6,1	4,9
256	tex	51,31	77,64	128,95	17,7	10,6
320	cpu	0,00	2.682,51	2.682,51	1,0	1,0
320	gpu	97,26	456,64	553,90	5,9	4,8
320	tex	98,05	151,96	250,01	17,7	10,7

Tabelle 21: Messergebnisse »Boxfilter 1«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	89,17	89,17	1,0	1,0
64	gpu	1,92	18,99	20,91	4,7	4,3
64	tex	2,02	4,32	6,34	20,6	14,1
128	cpu	0,00	733,09	733,09	1,0	1,0
128	gpu	7,88	160,28	168,16	4,6	4,4
128	tex	8,90	34,01	42,91	21,6	17,1
192	cpu	0,00	2.496,18	2.496,18	1,0	1,0
192	gpu	22,52	516,11	538,63	4,8	4,6
192	tex	26,56	115,21	141,77	21,7	17,6
256	cpu	0,00	6.222,91	6.222,91	1,0	1,0
256	gpu	52,60	1.189,98	1.242,58	5,2	5,0
256	tex	51,42	271,73	323,16	22,9	19,3
320	cpu	0,00	11.648,18	11.648,18	1,0	1,0
320	gpu	98,64	2.382,23	2.480,87	4,9	4,7
320	tex	98,02	531,08	629,10	21,9	18,5

Tabelle 22: Messergebnisse »Boxfilter 2«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	10,75	10,75	1,0	1,0
64	gpu	1,91	1,77	3,68	6,1	2,9
64	tex	2,04	2,03	4,07	5,3	2,6
128	cpu	0,00	85,40	85,40	1,0	1,0
128	gpu	7,92	9,16	17,08	9,3	5,0
128	tex	8,13	10,78	18,91	7,9	4,5
192	cpu	0,00	288,39	288,39	1,0	1,0
192	gpu	22,50	27,96	50,46	10,3	5,7
192	tex	22,87	32,28	55,15	8,9	5,2
256	cpu	0,00	684,74	684,74	1,0	1,0
256	gpu	50,85	66,80	117,65	10,3	5,8
256	tex	51,35	74,04	125,39	9,2	5,5
320	cpu	0,00	1.335,60	1.335,60	1,0	1,0
320	gpu	97,40	126,38	223,78	10,6	6,0
320	tex	99,14	143,23	242,37	9,3	5,5

Tabelle 23: Messergebnisse »Boxilter 1 (separiert)«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	15,71	15,71	1,0	1,0
64	gpu	1,91	2,19	4,10	7,2	3,8
64	tex	2,01	2,10	4,11	7,5	3,8
128	cpu	0,00	127,83	127,83	1,0	1,0
128	gpu	8,36	12,52	20,88	10,2	6,1
128	tex	8,91	11,42	20,33	11,2	6,3
192	cpu	0,00	433,44	433,44	1,0	1,0
192	gpu	22,57	39,47	62,04	11,0	7,0
192	tex	22,88	34,31	57,19	12,6	7,6
256	cpu	0,00	1.115,91	1.115,91	1,0	1,0
256	gpu	50,82	96,16	146,97	11,6	7,6
256	tex	51,57	78,90	130,47	14,1	8,6
320	cpu	0,00	2.008,65	2.008,65	1,0	1,0
320	gpu	97,41	179,52	276,93	11,2	7,3
320	tex	98,30	152,68	250,98	13,2	8,0

Tabelle 24: Messergebnisse »Boxilter 2 (separiert)«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	1.370,79	1.370,79	1,0	1,0
64	gpu	1,96	18,67	20,63	73,4	66,4
64	tex	1,97	10,82	12,80	126,7	107,1
128	cpu	0,00	11.269,22	11.269,22	1,0	1,0
128	gpu	7,87	152,68	160,55	73,8	70,2
128	tex	8,10	87,85	95,95	128,3	117,4
192	cpu	0,00	38.380,96	38.380,96	1,0	1,0
192	gpu	22,48	514,52	536,99	74,6	71,5
192	tex	22,82	299,00	321,82	128,4	119,3
256	cpu	0,00	91.396,26	91.396,26	1,0	1,0
256	gpu	50,87	1.272,99	1.323,86	71,8	69,0
256	tex	51,33	710,52	761,85	128,6	120,0
320	cpu	0,00	179.061,98	179.061,98	1,0	1,0
320	gpu	99,64	2.441,96	2.541,60	73,3	70,5
320	tex	98,89	1.392,39	1.491,28	128,6	120,1

Tabelle 25: Messergebnisse »Gaußfilter 1«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	162,68	162,68	1,0	1,0
64	gpu	1,90	2,59	4,49	62,8	36,2
64	tex	2,01	2,64	4,65	61,6	35,0
128	cpu	0,00	1.314,88	1.314,88	1,0	1,0
128	gpu	7,98	16,19	24,18	81,2	54,4
128	tex	8,59	15,97	24,56	82,4	53,5
192	cpu	0,00	4.451,71	4.451,71	1,0	1,0
192	gpu	22,47	50,91	73,39	87,4	60,7
192	tex	22,91	49,07	71,98	90,7	61,8
256	cpu	0,00	10.678,07	10.678,07	1,0	1,0
256	gpu	50,81	121,95	172,76	87,6	61,8
256	tex	51,45	114,58	166,03	93,2	64,3
320	cpu	0,00	20.685,47	20.685,47	1,0	1,0
320	gpu	98,42	232,72	331,14	88,9	62,5
320	tex	98,27	221,10	319,37	93,6	64,8

Tabelle 26: Messergebnisse »Gaußfilter 1 (separiert)«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	290,33	290,33	1,0	1,0
64	gpu	1,93	3,65	5,59	79,5	52,0
64	tex	2,03	3,16	5,19	91,9	55,9
128	cpu	0,00	2.363,96	2.363,96	1,0	1,0
128	gpu	8,67	25,83	34,50	91,5	68,5
128	tex	8,12	20,09	28,22	117,7	83,8
192	cpu	0,00	8.022,08	8.022,08	1,0	1,0
192	gpu	26,61	81,26	107,86	98,7	74,4
192	tex	22,87	63,59	86,46	126,2	92,8
256	cpu	0,00	19.148,52	19.148,52	1,0	1,0
256	gpu	50,82	195,49	246,31	98,0	77,7
256	tex	51,43	147,80	199,23	129,6	96,1
320	cpu	0,00	37.335,31	37.335,31	1,0	1,0
320	gpu	97,61	375,25	472,87	99,5	79,0
320	tex	98,14	287,92	386,05	129,7	96,7

Tabelle 27: Messergebnisse »Gaußfilter 2 (separiert)«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	224,75	224,75	1,0	1,0
64	gpu	1,87	26,59	28,47	8,5	7,9
64	tex	1,95	25,08	27,03	9,0	8,3
128	cpu	0,00	1.830,00	1.830,00	1,0	1,0
128	gpu	7,91	224,94	232,84	8,1	7,9
128	tex	8,17	204,78	212,95	8,9	8,6
192	cpu	0,00	6.207,23	6.207,23	1,0	1,0
192	gpu	22,56	751,45	774,00	8,3	8,0
192	tex	22,78	687,97	710,75	9,0	8,7
256	cpu	0,00	14.754,21	14.754,21	1,0	1,0
256	gpu	50,82	1.832,01	1.882,83	8,1	7,8
256	tex	52,36	1.623,84	1.676,20	9,1	8,8
320	cpu	0,00	28.858,33	28.858,33	1,0	1,0
320	gpu	98,53	3.509,20	3.607,72	8,2	8,0
320	tex	98,70	3.181,21	3.279,91	9,1	8,8

Tabelle 28: Messergebnisse »Medianfilter«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	4.534,66	4.534,66	1,0	1,0
64	gpu	1,96	22,37	24,33	202,7	186,4
64	tex	1,95	20,86	22,81	217,4	198,8
128	cpu	0,00	37.528,56	37.528,56	1,0	1,0
128	gpu	7,87	181,42	189,29	206,9	198,3
128	tex	8,10	169,58	177,69	221,3	211,2
192	cpu	0,00	126.431,78	126.431,78	1,0	1,0
192	gpu	22,51	622,85	645,36	203,0	195,9
192	tex	22,82	577,52	600,34	218,9	210,6
256	cpu	0,00	301.215,50	301.215,50	1,0	1,0
256	gpu	50,93	1.452,63	1.503,56	207,4	200,3
256	tex	52,69	1.371,56	1.424,25	219,6	211,5
320	cpu	0,00	581.928,81	581.928,81	1,0	1,0
320	gpu	98,48	2.880,99	2.979,47	202,0	195,3
320	tex	99,74	2.694,47	2.794,21	216,0	208,3

Tabelle 29: Messergebnisse »Bilateraler Filter«

C.3 Diffusions Filter

Die Messung der Diffusions Filter:

- Lineare homogene Diffusion
- Nichtlineare inhomogene Diffusion
- Nichtlineare anisotrope Diffusion (EED)

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	5,81	5,81	1,0	1,0
64	gpu	1,86	0,69	2,55	8,4	2,3
64	tex	2,00	0,51	2,51	11,4	2,3
128	cpu	0,00	48,80	48,80	1,0	1,0
128	gpu	7,86	5,30	13,16	9,2	3,7
128	tex	8,11	3,51	11,62	13,9	4,2
192	cpu	0,00	166,63	166,63	1,0	1,0
192	gpu	22,53	17,59	40,12	9,5	4,2
192	tex	22,80	11,71	34,51	14,2	4,8
256	cpu	0,00	397,91	397,91	1,0	1,0
256	gpu	72,09	40,61	112,70	9,8	3,5
256	tex	72,80	26,10	98,91	15,2	4,0
320	cpu	0,00	779,82	779,82	1,0	1,0
320	gpu	139,27	82,08	221,35	9,5	3,5
320	tex	140,08	54,21	194,29	14,4	4,0

Tabelle 30: Messergebnisse »Lineare homogene Diffusion«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	135,96	135,96	1,0	1,0
64	gpu	1,88	3,62	5,50	37,6	24,7
64	tex	2,00	2,09	4,09	65,1	33,3
128	cpu	0,00	1.200,06	1.200,06	1,0	1,0
128	gpu	7,86	32,84	40,70	36,5	29,5
128	tex	8,13	16,07	24,20	74,7	49,6
192	cpu	0,00	4.183,98	4.183,98	1,0	1,0
192	gpu	22,44	108,04	130,48	38,7	32,1
192	tex	22,88	54,11	77,00	77,3	54,3
256	cpu	0,00	10.108,54	10.108,54	1,0	1,0
256	gpu	72,20	273,38	345,58	37,0	29,3
256	tex	72,81	128,31	201,12	78,8	50,3
320	cpu	0,00	19.849,77	19.849,77	1,0	1,0
320	gpu	139,26	506,39	645,64	39,2	30,7
320	tex	140,07	251,18	391,24	79,0	50,7

Tabelle 31: Messergebnisse »Nichtlineare inhomogene Diffusion«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	541,97	541,97	1,0	1,0
64	gpu	1,90	14,36	16,26	37,7	33,3
64	gpuopt	1,87	12,73	14,60	42,6	37,1
64	tex	2,00	30,30	32,29	17,9	16,8
64	texopt	1,99	16,82	18,81	32,2	28,8
128	cpu	0,00	4.545,40	4.545,40	1,0	1,0
128	gpu	7,87	122,09	129,96	37,2	35,0
128	gpuopt	7,87	113,67	121,54	40,0	37,4
128	tex	8,13	240,50	248,63	18,9	18,3
128	texopt	8,10	133,56	141,67	34,0	32,1
192	cpu	0,00	15.562,77	15.562,77	1,0	1,0
192	gpu	22,47	400,13	422,60	38,9	36,8
192	gpuopt	26,37	377,01	403,38	41,3	38,6
192	tex	22,84	810,61	833,45	19,2	18,7
192	texopt	31,85	452,64	484,49	34,4	32,1
256	cpu	0,00	37.312,49	37.312,49	1,0	1,0
256	gpu	72,06	961,71	1.033,77	38,8	36,1
256	gpuopt	71,85	934,08	1.005,92	39,9	37,1
256	tex	74,68	1.929,65	2.004,33	19,3	18,6
256	texopt	74,35	1.071,46	1.145,81	34,8	32,6
320	cpu	0,00	72.912,65	72.912,65	1,0	1,0
320	gpu	141,14	1.859,84	2.000,98	39,2	36,4
320	gpuopt	141,24	1.772,21	1.913,44	41,1	38,1
320	tex	141,42	3.762,75	3.904,17	19,4	18,7
320	texopt	141,99	2.094,62	2.236,61	34,8	32,6

Tabelle 32: Messergebnisse »Nichtlineare Anisotrope Diffusion (EED)«

C.4 Affine Transformationen

Die Messung der affinen Transformationen:

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	16,04	16,04	1,0	1,0
64	gpu	1,83	0,41	2,25	39,0	7,1
64	tex	1,96	0,36	2,31	45,0	6,9
128	cpu	0,00	136,92	136,92	1,0	1,0
128	gpu	7,88	4,24	12,12	32,3	11,3
128	tex	8,64	2,31	10,95	59,2	12,5
192	cpu	0,00	471,32	471,32	1,0	1,0
192	gpu	22,48	11,98	34,46	39,3	13,7
192	tex	23,33	7,54	30,87	62,5	15,3
256	cpu	0,00	1.173,75	1.173,75	1,0	1,0
256	gpu	50,75	33,53	84,28	35,0	13,9
256	tex	51,34	17,59	68,93	66,7	17,0
320	cpu	0,00	2.579,43	2.579,43	1,0	1,0
320	gpu	97,56	59,96	157,52	43,0	16,4
320	tex	98,19	34,23	132,43	75,3	19,5

Tabelle 33: Messergebnisse »Affine Transformation (Nearest Neighbor)«

Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	17,32	17,32	1,0	1,0
64	gpu	1,82	1,51	3,33	11,5	5,2
64	tex	1,95	0,56	2,52	30,7	6,9
128	cpu	0,00	149,05	149,05	1,0	1,0
128	gpu	7,88	27,75	35,63	5,4	4,2
128	tex	8,94	4,09	13,02	36,5	11,4
192	cpu	0,00	516,43	516,43	1,0	1,0
192	gpu	22,52	79,91	102,43	6,5	5,0
192	tex	22,86	13,58	36,43	38,0	14,2
256	cpu	0,00	1.664,23	1.664,23	1,0	1,0
256	gpu	50,77	246,15	296,93	6,8	5,6
256	tex	51,86	31,46	83,31	52,9	20,0
320	cpu	0,00	3.326,11	3.326,11	1,0	1,0
320	gpu	97,30	410,47	507,76	8,1	6,6
320	tex	98,05	61,33	159,38	54,2	20,9

Tabelle 34: Messergebnisse »Affine Transformation (Trilinear)«

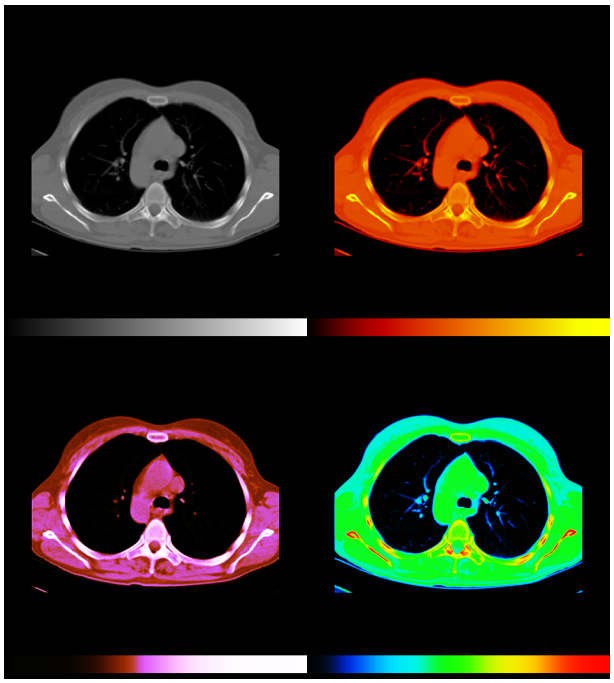
Dim	Code	MemCpy [ms]	Filter [ms]	Total [ms]	Brutto	Netto
64	cpu	0,00	49,84	49,84	1,0	1,0
64	gpu	1,83	10,43	12,26	4,8	4,1
64	tex	1,96	5,45	7,41	9,1	6,7
128	cpu	0,00	430,00	430,00	1,0	1,0
128	gpu	7,89	210,04	217,93	2,0	2,0
128	tex	8,10	42,96	51,06	10,0	8,4
192	cpu	0,00	1.485,10	1.485,10	1,0	1,0
192	gpu	22,48	638,07	660,55	2,3	2,2
192	tex	22,85	144,95	167,80	10,2	8,9
256	cpu	0,00	4.151,91	4.151,91	1,0	1,0
256	gpu	51,75	1.846,91	1.898,66	2,2	2,2
256	tex	51,39	341,39	392,79	12,2	10,6
320	cpu	0,00	7.868,51	7.868,51	1,0	1,0
320	gpu	98,55	3.224,56	3.323,11	2,4	2,4
320	tex	98,07	666,02	764,10	11,8	10,3

Tabelle 35: Messergebnisse »Affine Transformation (Cubic B-Splines)«

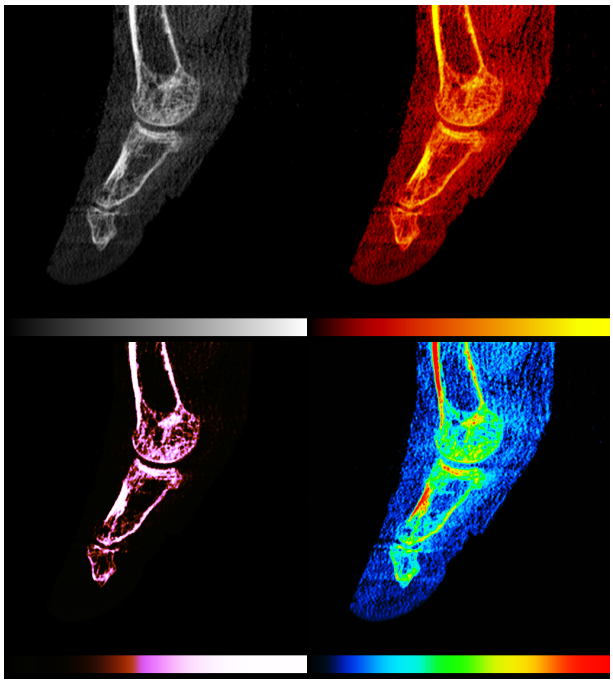
D Volumendatensätze

Die im Rahmen dieser Arbeit verwendeten medizinischen Volumendatensätze stammen aus *The Volume Library* (Female Chest, Foot, Knee, Lobster, MRI Head, Headmale) bzw. dem *VolVis Archiv* (Skull, Ventricles, Stent, Colon Supine).
Die Datensätze sind in vielen Artikeln und Publikationen aus den Bereichen grafische Datenverarbeitung, (Volumen-)Visualisierung und Medizin in den letzten Jahren verwendet worden und über die genannten Archiven online frei verfügbar. Die Datensätze wurden am 16. März 2010 um 19:00 Uhr aus *The Volume Library* unter der Adresse http://www9.informatik.uni-erlangen.de/External/vollib/ bzw. aus dem *VolVis Archiv* unter der Adresse http://www.volvis.org/ abgerufen.
Die Datensätze aus *The Volume Library* wurden zunächst mit dem PVM2RAW Tool aus dem V^3 Volume Rendering Package, abgerufen auf http://www.stereofx.org/volume.html (16. März 2010) ins RAW Format konvertiert. Da die erstellte GPU Implementierung aus technischen Gründen spezielle Datensatzdimensionen erfordert (Datensatzgröße in jeder Dimension ein Vielfaches von 8), wurden die original Volumendatensätze gegebenenfalls mithilfe trilinearer Interpolation entsprechend skaliert. Der implementierte Prototyp unterstützt neben der »trilinearen Filterung« zusätzlich die Resampling-Methoden »Nearest Neighbor« und »B-Spline« – Filtering.

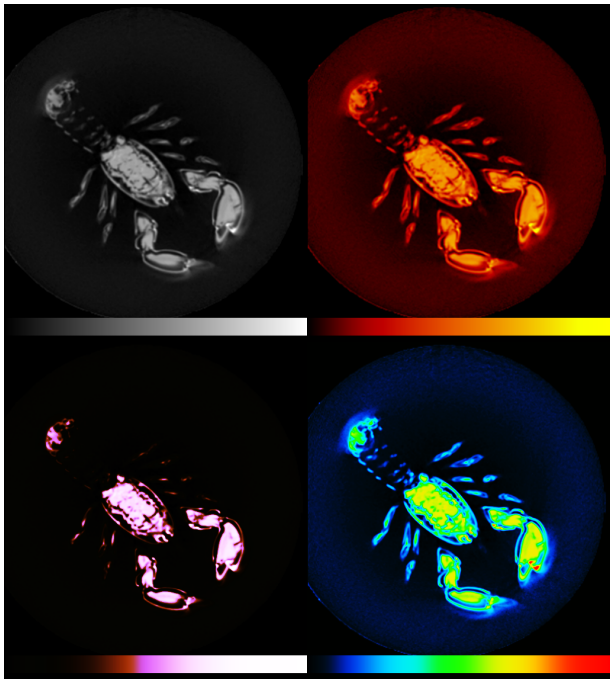
Female Chest: 384 × 384 × 240, 8 Bit



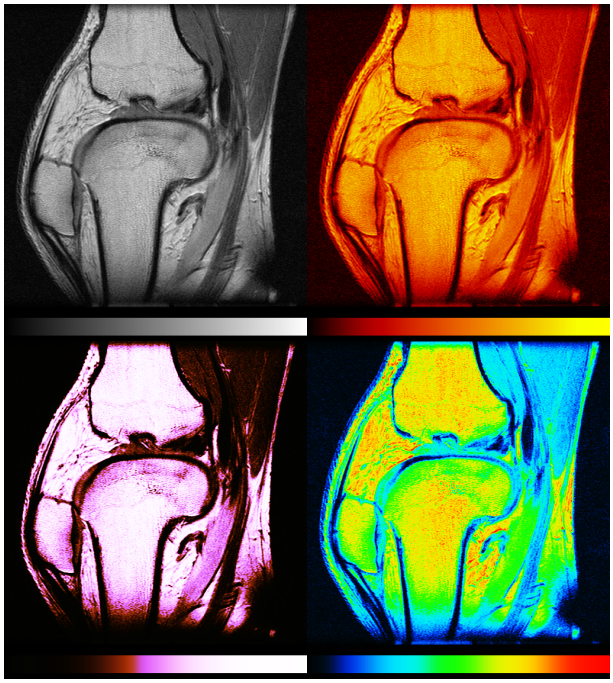
Foot: 256 × 256 × 256, 8 Bit



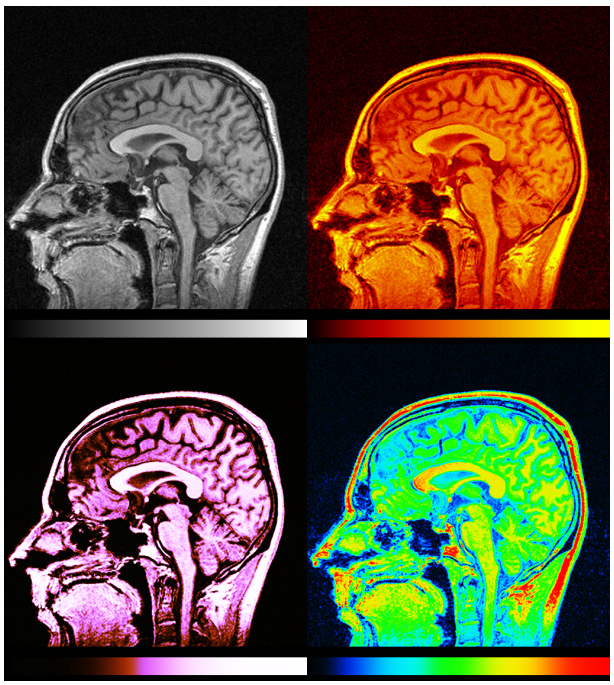
Lobster: 304 × 320 × 56, 8 Bit



Knee: 512 × 512 × 88, 16 Bit

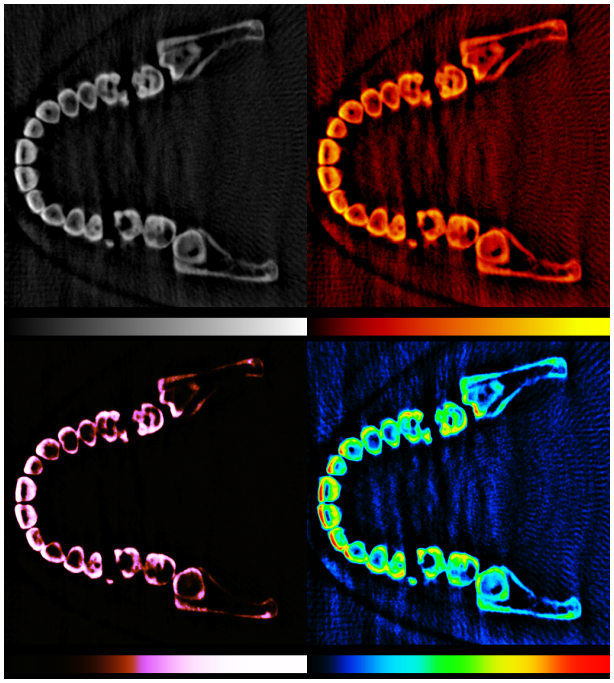


MRI Head: 256 × 256 × 256, 8 Bit

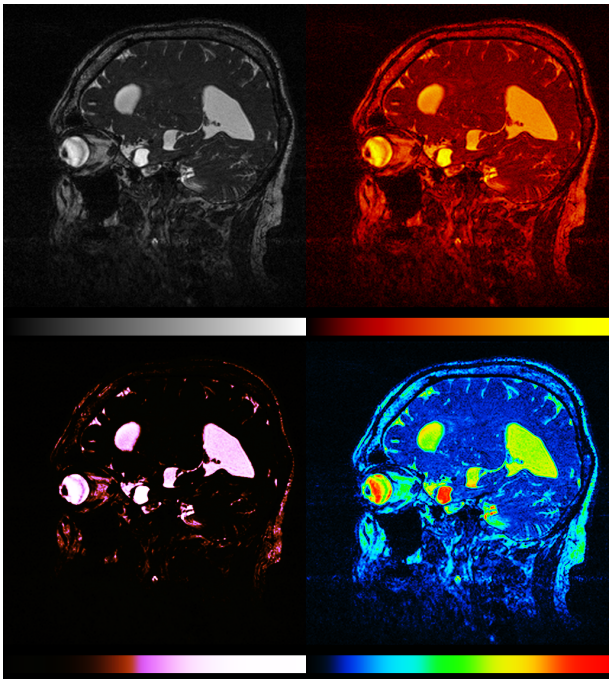


Head Male: 256 × 256 × 128, 8 Bit

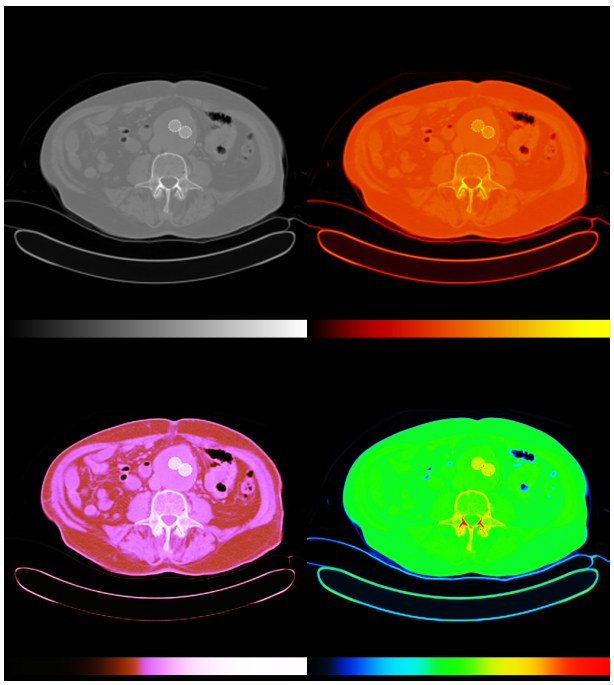
Skull: 256 × 256 × 256, 8 Bit



Ventricles: 256 × 256 × 128, 8 Bit



Stent: 256 × 256 × 88, 16 Bit



Colon Supine: 352 × 352 × 352, 16 Bit