Bachelor Thesis

vorgelegt an der Hochschule
für angewandte Wissenschaften Fachhochschule Würzburg-Schweinfurt
in der Fakultät Informatik und Wirtschaftsinformatik
zum Abschluss eines Studiums im Studiengang Informatik

Studienschwerpunkt: Medieninformatik

Schnelle 2-D/3-D-Registrierung medizinischer Datensätze auf Grafikprozessoren

-

-

-

Abgabetermin:

Eingereicht von: Christian Ofenberg aus -

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Würzburg, den

(Unterschrift)

Übersicht

Das Ziel dieser Arbeit besteht darin, eine schnelle 2-D/3-D-Registrierung medizinischer Datensätze auf Grafikprozessoren zu realisieren. Um dies zu erreichen, werden bekannte theoretische Ansätze der 2-D/3-D-Registrierung anhand des neuen GPGPU-Standards OpenCL 1.0 umgesetzt. Bei Grafikprozessoren handelt es sich um Hardware mit einer hochgradig parallelen Architektur, daher besteht die Notwendigkeit, parallele Algorithmen zu nutzen.

Einige Algorithmen, wie beispielsweise das Volume-Raytracing, sind von Natur aus einfach parallelisierbar. Bei anderen Algorithmen besteht hingegen die Notwendigkeit, diese von einer sequentiellen Form in eine parallele Form zu übertragen. Eine der Aufgaben der 2-D/3-D-Registrierung besteht darin, die Ähnlichkeit zwischen zwei Bildern festzustellen. Die Summenbildung spielt bei der Ähnlichkeitsbestimmung eine Schlüsselrolle, dazu wird auf dem Grafikprozessor eine parallele Reduktion eingesetzt.

In Experimenten wurde ermittelt, welche Aufgabenstellungen der 2-D/3-D-Registrierung sich besonders gut auf Grafikprozessoren realisieren lassen. Die Ergebnisse wurden aufgeschlüsselt und diskutiert. Abschließend wurde geprüft, ob OpenCL zu OpenGL vergleichbare Leistungsergebnisse liefert.

Abstract

The goal of this work is to implement a fast 2D/3D registration of medical images on a GPU. To achieve this, well known theoretical 2D/3D registration approaches are adopted by using the new GPGPU standard OpenCL 1.0. The architecture of GPUs is highly parallel, as a result it's necessary to use parallel algorithms.

Some algorithms, such as volume raytracing, are parallel by nature. Other algorithms however need to be transformed from a sequential form into a parallel form. One of the tasks during the 2D/3D registration is to measure the similarity between two images. The totaling is one of the key elements of the similarity determination, on the GPU this is performed by using parallel reduction.

In order to figure out which tasks of the 2D/3D registration are well suited for GPUs, several experiments were performed. The results are presented and discussed. Finally a test was performed in order to determine whether or not OpenCL achieves a performance similar to that of OpenGL.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Heutige Grafikprozessoren[1] haben eine lange Evolution hinter sich. Ursprünglich dienten GPUs lediglich zur Beschleunigung von 2-D/3-D-Grafiken, daher auch der Name für diese Hardwarekomponente. Die Leistung von Hauptprozessoren[2] wächst immer langsamer. Seit einigen Jahren werden daher den CPUs immer mehr Kerne hinzugefügt, um die Gesamtleistung weiter zu steigern. Grafikprozessoren hingegen nutzten schon immer einen extremen parallelen Ansatz mit vielen Rechenkernen. Zwar besitzt ein Kern einer klassischen CPU deutlich mehr Rechenleistung als einer der Kerne einer GPU, jedoch kommt selbst eine moderne 4-Kern-CPU bei weitem nicht an die schiere Masse der verfügbaren GPU-Kerne heran. Durch die parallele Architektur der Grafikprozessoren wächst die Leistung bereits seit Jahren praktisch ungebremst immer weiter und ein Ende ist nicht abzusehen. Aus Sicht der erreichbaren Rechenleistung haben GPUs die CPUs seit langem hinter sich gelassen. Die enorme zur Verfügung stehende Rechenleistung im System bleibt jedoch oft ungenutzt.

Bereits seit etlichen Jahren wird versucht, die rasant weiterwachsende Rechenleistung der Grafikprozessoren für allgemeine Berechnungen[3] zu nutzen. Diese Berechnungen haben oft wenig, oder gar nichts mit Grafik zu tun. Aufgrund des parallelen Ansatzes lassen sich nicht alle Problemstellungen mit Hilfe der GPUs beschleunigen, dort wo dies jedoch möglich ist, sind enorme Leistungssteigerungen gegenüber einer sequentiellen CPU-Lösung zu verzeichnen. In den Anfangsjahren dieses neuen Trends mussten die Berechnungen derart umformuliert werden,

[1] engl. Graphics Processing Unit (GPU)
[2] engl. Central Processing Unit (CPU)
[3] engl. General-Purpose Computing on Graphics Processing Units (GPGPU)

dass sich diese über klassische Grafik-Programmierschnittstellen[4] realisieren ließen - ein sehr mühsamer und wenig zufriedenstellender Ansatz.

Aus diesem Grund entstanden in den letzten Jahren zahlreiche spezielle APIs für GPGPU. Viele dieser APIs konnten sich jedoch nicht durchsetzen und verschwanden schnell wieder von der Bildfläche. Seit die NVIDIA Corporation (NVIDIA) ihre Compute Unified Device Architecture (CUDA)-Technologie [NVI09d] veröffentlicht hat, ist ein regelrechter GPGPU Goldrausch ausgebrochen. Dieser Boom führte beispielsweise zu Grafikkarten ohne Bildschirmausgabe. Diese neuartige Hardware wurde ausschließlich für GPGPU ausgelegt und hat im Grunde mehr mit einem Vektorrechner gemeinsam als mit einer Grafikkarte. Die Nutzung von Grafikprozessoren zum Beschleunigen von Berechnungen kommt allmählich aus den Kinderschuhen. Mit dem neuen offenen Standard Open Computing Language (OpenCL) erreicht der GPGPU-Trend einen neuen Höhepunkt. Während CUDA noch rein für GPUs von NVIDIA ausgelegt wurde, handelt es sich bei OpenCL um eine hersteller- und hardwareunabhängige Technologie, die GPGPU auf die nächste Stufe hebt. Eine einmal geschriebene OpenCL-Anwendung lässt sich nicht nur auf GPUs ausführen, sondern beispielsweise auch auf CPUs oder anderer für Berechnungen geeigneter Hardware. Durch die Nutzung von OpenCL sind Entwickler ebenfalls in der Lage, Multi-Kern-CPUs für Berechnungen vernünftig auszunutzen. Während Multithreading von Hand schnell sehr mühsam werden kann, steht dem Entwickler nun eine einheitliche API für parallele Programmierung zur Verfügung.

Die aktuellen Entwicklungen legen nahe, parallelisierbare Berechnungen zukünftig nicht mehr in klassischen sequentiellen Programmiersprachen oder mit Hilfe von proprietären Schnittstellen zu entwickeln. Es scheint heute sinnvoller zu sein, direkt mit OpenCL zu arbeiten, um die gesamte zur Verfügung stehende Rechenleistung voll nutzen zu können, ohne sich dabei von einem Hardwaretyp oder Hersteller abhängig zu machen.

Das Problem der 2-D/3-D-Registrierung medizinischer Datensätze, um die es in dieser Arbeit geht, ist sehr rechenaufwändig, sequentielle CPU-Lösungen sind daher für Echtzeitanwendungen nicht geeignet. Diese Aufgabenstellung wurde bereits in der Vergangenheit mit Grafikprozessoren angegangen. Unter Nutzung eingeschränkter *Fixed-Function-Pipeline* Grafik-APIs wurde meist lediglich das Volume-Raytracing über GPUs beschleunigt. Mit dem Aufkommen von Shadern wurden Grafik-APIs freier programmierbar. Das Problem der 2-D/3-D-Registrierung wurde erneut aufgegriffen und die resultierenden Lösungen konnten beachtliche Leistungssteigerungen verbuchen. Als mit CUDA eine weitere Möglichkeit der Programmierung von Grafikprozessoren bereitstand, folgte die nächste Welle von Lösungen der 2-D/3-D-Registrierung mit Hilfe von

[4] engl. Application Programming Interface (API)

GPUs. Dank OpenCL steht nun eine weitere Technologie zur Verfügung, mit deren Hilfe Berechnungen auf der GPU ausgeführt werden können. Es liegt daher nahe, die Problemstellung der 2-D/3-D-Registrierung anhand dieser neuen Schnittstelle erneut zu behandeln.

1.2 Zielsetzung

Das rechenintensive Problem der 2-D/3-D-Registrierung medizinischer Datensätze soll mit Hilfe von Grafikprozessoren beschleunigt werden. Es gilt, die Teilprobleme ausfindig zu machen und geeignete Algorithmen zu finden, mit deren Hilfe sich diese Teilprobleme parallelisieren lassen. Für die Implementation soll OpenCL verwendet werden, um eine möglichst flexible Lösung zu schaffen, die auf verschiedensten Grafikkarten unterschiedlicher Hersteller lauffähig ist. Die parallelisierten Teilprobleme sollen derart implementiert werden, dass diese auf der GPU schnell ausführbar sind. In Experimenten soll ermittelt werden, welche Teilprobleme sich dabei besonders gut auf der GPU umsetzen lassen und welche Beschleunigung in den einzelnen Teilen erzielt wurde. Es soll ermittelt werden, welche Faktoren die Ausführgeschwindigkeit auf der GPU wesentlich beeinflussen. In einem abschließenden Experiment soll festgestellt werden, ob die mit OpenCL erzielte Leistung vergleichbar mit der einer Open Graphics Library (OpenGL)-Realisierung ist.

1.3 Gliederung der Arbeit

Die Arbeit ist in 7 Kapitel unterteilt. Nachdem im Kapitel 1 die Motivation und Zielsetzung dieser Arbeit dargelegt wurde, folgt in Kapitel 2 eine Beschreibung der Grundlagen der 2-D/3-D-Registrierung. Dazu wird die Problemstellung in einzelne Teilaufgaben zerlegt. Nachdem über die realitätsnahe Erzeugung der künstlichen Rückprojektion[5] das 2-D/3-D-Problem auf ein 2-D/2-D-Problem reduziert wurde, folgt eine Übersicht der im Rahmen dieser Arbeit verwendeten Gütemaße. Die Gütemaße dienen dazu festzustellen wie ähnlich sich zwei Bilder sind. Das Optimierungsverfahren wird nicht auf Grafikprozessoren beschleunigt. Dieses Thema ist jedoch grundlegend für die 2-D/3-D-Registrierung, daher wird es im Rahmen dieses Kapitels ebenfalls kurz angeschnitten.

Kapitel 3 beschreibt die geschichtliche Entwicklung von GPUs anhand einiger Meilensteine. Es folgt ein Überblick über die Entwicklungsgeschichte von GPGPU und dessen neusten Ver-

[5] engl. Digitally Reconstructed Radiograph (DRR)

treter OpenCL. Da diese Arbeit OpenCL für die Programmierung der GPU nutzt, werden die wichtigsten Konzepte von OpenCL präsentiert.

Parallel zu dieser schriftlichen Ausarbeitung wurde die 2-D/3-D-Registrierung medizinischer Datensätze auf Grafikprozessoren ebenfalls mit Hilfe der Programmiersprache C++ und OpenCL implementiert, Details dazu werden in Kapitel 4 dargelegt. Das Kapitel beginnt mit einer Analyse der bei der 2-D/3-D-Registrierung vorhandenen generellen Problemstellungen, die sich parallelisieren und somit auf der GPU implementieren lassen.

Die Leistung der erstellten Implementation wird in Kapitel 5 präsentiert und diskutiert. Ebenfalls wurden einige Experimente durchgeführt um herauszufinden, welche Punkte für eine schnelle 2-D/3-D-Registrierung auf Grafikprozessoren besonders wichtig sind. Abschließend soll in diesem Kapitel geklärt werden, ob das noch recht neue OpenCL eine Leistung bringt, die besser oder gleichwertig zu OpenGL ist.

Die Arbeit wird in Kapitel 6 noch einmal zusammengefasst. Abschließend folgt im Kapitel 7 ein Ausblick über mögliche Erweiterungen der im Rahmen dieser Arbeit erstellten Implementation. Ebenfalls werden für GPGPU interessante kommende GPU-Entwicklungen genannt.

Kapitel 2

Grundlagen der 2-D/3-D-Registrierung

Dieses Kapitel gibt einen Überblick über die Grundlagen der 2-D/3-D-Registrierung, die zum Verständnis der restlichen Arbeit von Nöten sind. Aufgrund der umfangreichen Materie der 2-D/3-D-Registrierung bleibt dieses Kapitel oberflächlich und verweist an den entsprechenden Stellen auf vertiefende Literatur.

Im einleitenden Abschnitt 2.1 wird das allgemeine Prinzip der 2-D/3-D-Registrierung dargelegt. Ebenfalls wird hier kurz das Optimierungsverfahren angesprochen, welches die Ergebnisse der Gütemaße verwendet, um die optimale Transformation ausfindig zu machen.

Die Grundlagen der realitätsnahen Erzeugung der künstlichen Rückprojektion werden in Abschnitt 2.2 skizziert.

Der Abschnitt 2.3 geht auf die Gütemaße ein. Es werden verschiedene Klassen von Gütemaßen angesprochen und deren mathematische Grundlagen dargelegt.

2.1 Prinzip der 2-D/3-D-Registrierung

Das Ziel der 2-D/3-D-Registrierung medizinischer Datensätze besteht darin, eine geeignete Transformation zu finden, die ein *MRT*[1]- oder *CT*[2]-Volumen mit einem Durchleuchtungsbild[3] in Übereinstimmung bringt. Neben diagnostischen Anwendungen bietet dieses Verfahren vielfältige weitere Einsatzmöglichkeiten bspw. in der operativen Medizin. So können während einer Operation die Bewegungen eines Patienten durch die 2-D/3-D-Registrierung ausgeglichen werden. Wünschenswert ist es daher, diese Überlagerung verschiedener Datensätze in Echtzeit zu

[1] engl. Magnet-Resonanz-Tomographie (MRT)
[2] engl. Computer-Tomographie (CT)
[3] engl. Fluoroscopic image (FLL)

ermitteln. Dies wurde, zumindest unter Zuhilfenahme teurer Spezialhardware, auch bereits realisiert. Nach [Kre08] sind die medizinischen Registrierungsverfahren ein wachsendes Gebiet von steigendem Interesse, mit sehr unterschiedlichen Anforderungen, Bedingungen und Lösungen. Zahlreiche mathematische Grundlagen, wie beispielsweise die Entropie von Shannon [Sha48], sind hingegen seit längerem bekannt. Eine effiziente, adaptive 2-D/3-D-Registrierung von Röntgenbildern und 3-D-Volumina wird in [Kub08] beschrieben, die vorliegende Arbeit greift viele der dort beschriebenen theoretischen Ansätze auf.

Bild 2.1 visualisiert den im Folgenden beschriebenen Ablauf der 2-D/3-D-Registrierung. Die

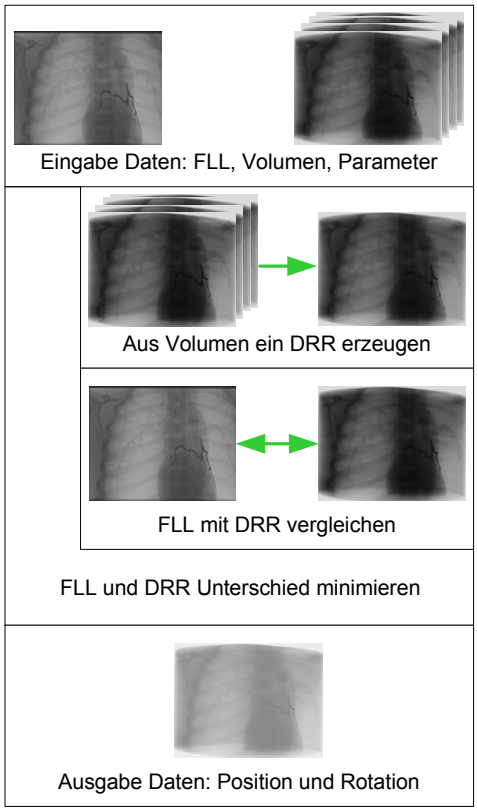

Bild 2.1: Ablauf der 2-D/3-D-Registrierung als Nassi-Shneiderman-Diagramm (Struktogramm)

2-D/3-D-Registrierung setzt sich aus drei Komponenten zusammen. Aufgabe der ersten Komponente ist es, anhand gegebener Einstellungen eine mit dem Röntgenbild vergleichbare 2-D-Abbildung des Volumens anzufertigen, das DRR. Eine zweite Komponente muss anschließend die Ähnlichkeit zwischen dem erzeugten künstlichen Röntgenbild und dem gegebenen konstanten Röntgenbild ermitteln. Dies geschieht unter Zuhilfenahme eines Gütemaßes oder einer Kombination mehrerer Gütemaße. Das Ergebnis dieser Komponente ist ein numerischer Wert, der die

Ähnlichkeit der zwei aktuellen Bilder ausdrückt. Die dritte Komponente besteht aus einem oder mehreren Optimierungsverfahren.

Optimierungsverfahren Die Aufgabe eines Optimierungsverfahren besteht ganz allgemein darin, eine Kostenfunktion zu minimieren oder maximieren. Bei der 2-D/3-D-Registrierung stellt die Ähnlichkeit der zwei Bilder diese Kostenfunktion dar. Das Optimierungsverfahren kann daher als Steuereinheit angesehen werden. Das Resultat eines Optimierungsverfahrens ist eine Transformation. Es existieren verschiedene Registrierungs-Transformationen, wie beispielsweise *rigide*, *affine*, *projektive* oder *elastische* Transformationen. Laut [Kre08] wird eine Transformation als *rigide* bezeichnet, wenn der Abstand zweier Punkte des ersten Bildes bei der Abbildung auf das zweite Bild bewahrt wird. In dieser Arbeit wird eine *rigide* Transformation $(t_x, t_y, t_z, r_x, r_y, r_z)$ verwendet, die jeweils die Verschiebung und Rotation des Volumens entlang der X-, Y- und Z-Achse darstellt. Die Aufgabe des Optimierungsverfahren besteht folglich darin, den kontinuierlichen, sechsdimensionalen Parameterraum zu durchsuchen und den Parametervektor ausfindig zu machen, der die Kostenfunktion maximiert. Da die Suche in einer endlichen Zeit abgeschlossen werden muss, sind spezielle Suchstrategien wie die in [Kub08] beschriebenen Verfahren *Adaptive Random Search* oder *Best-Neighbour* nötig.

Zwar sind Optimierungsverfahren wichtig für die 2-D/3-D-Registrierung als solche, aber in Hinblick auf GPU-Beschleunigung nicht relevant, da dieser Schritt auf der CPU abläuft. Daher sei an dieser Stelle auf [Kub08] und [Kre08] verwiesen, dort findet sich jeweils ein Überblick über dieses Thema.

2.2 Realitätsnahe DRR-Erzeugung

Laut [Kub08] ist es einfacher, ein DRR und ein Röntgenbild korrekt zu registrieren, wenn sich diese Bilder so ähnlich wie möglich sind. Daher wird in [Kub08] ausführlich dargelegt, wie sich die Röntgenbildentstehung in der C-Bogen-Anlage so nachbilden lässt, dass es nicht nötig ist, die Bilder in aufwändigen Nachbearbeitungsschritten anzugleichen. Die realitätsnahe DRR-Erzeugung lässt sich dabei in 3 Komponenten unterteilen: Die erste Komponente simuliert den Weg der Röntgenstrahlen durch die Materie, die Dichte der Materie ist dabei im Volumen unter Verwendung der *Hounsfield-Skala* beschrieben. Dies wird durch den Einsatz eines geeigneten Volume-Rendering-Algorithmus erreicht. In Kapitel 4 wird das im Rahmen dieser Arbeit verwendete Verfahren erörtert. Eine zweite Komponente ermittelt die Intensität im inneren Drittel des Bildes, dies ist in Bild 2.2 abgebildet. Hierbei handelt es sich um eine einfache Summenbil-

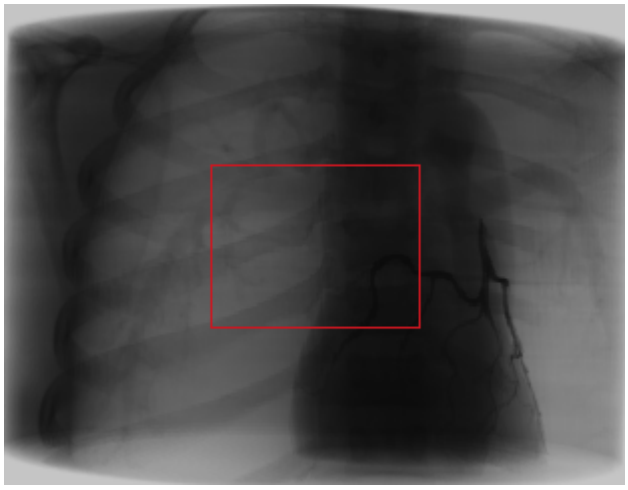

Bild 2.2: Inneres Drittel (rot) eines künstlichen Röntgenbildes

dung. Die dritte Komponente überführt anschließend die simulierten Röntgenstrahlen mit Hilfe eines in [Kub08] beschriebenen Verfahrens in eine dem Röntgenbild vergleichbare Form.

2.3 Gütemaße

Durch die DRR-Erzeugung wurde das Problem der 2-D/3-D-Registrierung auf ein 2-D/2-D-Problem reduziert. Es besteht folglich nur noch die Notwendigkeit festzustellen, wie groß die Ähnlichkeit zwischen dem gegebenen konstanten Röntgenbild I_{FLL} und dem während der Registrierung erzeugten DRR I_{DRR} ist. Dies ist die Aufgabe eines Gütemaßes

$$S(I_{FLL}, I_{DRR}). \tag{2.1}$$

Die berechnete Ähnlichkeit wird durch einen numerische Wert ausgedrückt der jeweils Werte aus einem bekannten Bereich annimmt. Die Ähnlichkeit wird, wie in Bild 2.3 abgebildet, nur in der Region Of Interest (ROI) bestimmt. Gerade an den Rändern der Bilder besteht die Gefahr, dass die zu vergleichenden Bilder sehr stark voneinander abweichen. Dies lässt sich aber durch eine gut gewählte ROI kompensieren.

In der Literatur existieren zahlreiche Gütemaße und Variationen dieser Gütemaße. [Kub08] gibt einen sehr guten Überblick über etliche Gütemaße und deren Zusammenhänge. Ebenfalls wird beschrieben, welche Wertebereiche die einzelnen Gütemaße annehmen können und wie sich die Gütemaße normalisieren lassen, damit diese untereinander vergleichbar werden. Im Folgenden soll lediglich auf die in dieser Arbeit verwendeten Gütemaße eingegangen werden. Dazu

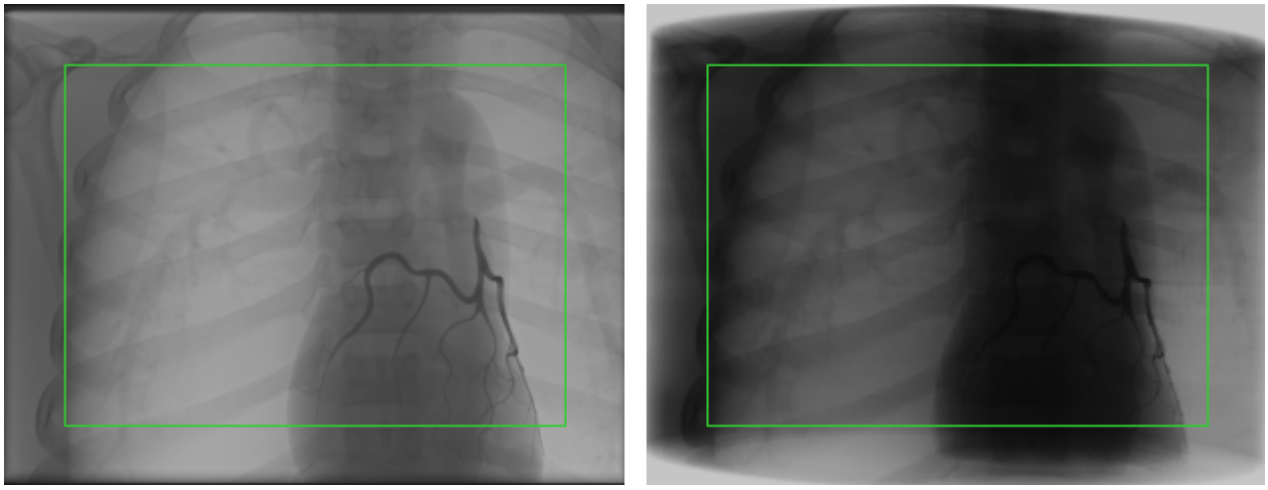

Bild 2.3: ROI (grün) jeweils für das Röntgenbild (links) und das künstliche Röntgenbild (rechts)

werden die für die Gütemaße nötigen Berechnungsschritte und Formeln dargelegt. Die Gütemaße werden dazu nach [Kub08] geordnet. Um den Umfang dieser Arbeit nicht zu sprengen, soll jedoch nicht allzu sehr ins Detail gegangen werden, stattdessen wird an den entsprechenden Stellen auf weiterführende Literatur verwiesen.

Rein intensitätsbasierte Gütemaße Das Gütemaß *Summe der quadratischen Abstände*[4]

$$S_{SSD}(I_{FLL}, I_{DRR}) = \frac{1}{|T|} \sum_{(i,j)\in T} (I_{FLL}(i,j) - I_{DRR}(i,j))^2 \tag{2.2}$$

ist eines der einfachsten Gütemaße. Bei T handelt es sich um den überlappenden Bereich der beiden Bilder, die ROI. $|T|$ hingegen stellt die Anzahl an Bildpunkten innerhalb dieses Bereiches dar. In diesem Verfahren werden die Bildintensität $I_{FLL}(i,j)$ eines jeden Bildpunktes des Röntgenbildes und die Bildintensität $I_{DRR}(i,j)$ jedes Bildpunktes des erzeugten künstlichen Röntgenbildes paarweise vergleichen. Laut [Kre08] kann gezeigt werden, dass SSD das optimale Gütemaß ist, wenn sich die Bilder nur durch Gaußsches Rauschen unterscheiden. Im Rahmen der 2-D/3-D-Registrierung unterscheiden sich die Bilder jedoch deutlicher voneinander, so sieht das dynamisch erzeugte DRR sichtbar etwas anders aus als das gegebene Röntgenbild. Sollte es eine kleine Anzahl von Bildpunkten mit sehr großen Intensitätsunterschieden geben, so wird das Ergebnis von SSD dadurch stark beeinflusst.

[4]engl. Sum of Squared Differences (SSD)

Das Gütemaß *Summe der absoluten Abstände*[5]

$$S_{SAD}(I_{FLL}, I_{DRR}) = \frac{1}{|T|} \sum_{(i,j)\in T} |I_{FLL}(i,j) - I_{DRR}(i,j)| \tag{2.3}$$

soll hier Abhilfe schaffen [Kre08] [Kub08]. Dieses Gütemaß ist fast identisch zu SSD, lediglich die Quadratsumme wurde durch den Absolutwert ersetzt.

Mit der *Summe der positiven Differenzen*[6]

$$S_{SPD}(I_{FLL}, I_{DRR}) = \frac{1}{|T|} \sum_{(i,j)\in T} \max(0, I_{FLL}(i,j) - I_{DRR}(i,j)) \tag{2.4}$$

existiert eine weitere Variante von SAD [Kub08]. Negative Ergebnisse fließen hier nicht in die Summenbildung ein.

Die *Summe der absoluten Abstände oberhalb eines Schwellenwertes*[7]

$$S_{SDT}(I_{FLL}, I_{DRR}) = \frac{1}{|T|} \sum_{(i,j)\in T} \max(b, |I_{FLL}(i,j) - I_{DRR}(i,j)|) \tag{2.5}$$

ist ein weiteres sehr einfaches Gütemaß [Kub08]. Auch hier werden ausschließlich paarweise Intensitätsunterschiede zweier Bildpunkte verglichen, daher stellt dieses Gütemaß im Grunde eine weitere Variation von SAD dar. Statt negative Ergebnisse zu ignorieren, werden hier alle Ergebnisse unterhalb eines gegebenen Schwellenwertes b durch den Schwellenwert selbst ersetzt.

Räumlich intensitätsbasierte Gütemaße Als Vertreter für räumlich intensitätsbasierte Gütemaße wurde in dieser Arbeit die als

$$S_{GC}(I_{FLL}, I_{DRR}) = \frac{S_{NCC}(\frac{\delta I_{FLL}}{\delta i}, \frac{\delta I_{DRR}}{\delta i}) + S_{NCC}(\frac{\delta I_{FLL}}{\delta j}, \frac{\delta I_{DRR}}{\delta j})}{2} \tag{2.6}$$

definierte *Gradientenkorrelation*[8] gewählt [Kub08]. Dieses Gütemaß ist unempfindlich gegenüber Helligkeitsveränderungen der beiden Eingabebilder.

[5] engl. Sum of Absolute Differences (SAD)
[6] engl. Sum of Positive Differences (SPD)
[7] engl. Sum of the absolute values of Differences above a Threshold (SDT)
[8] engl. Gradient Correlation (GC)

In einem ersten Schritt werden aus dem Röntgenbild I_{FLL} und dem simulierten Röntgenbild I_{DRR} mit Hilfe einer horizontalen

$$S_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \tag{2.7}$$

und vertikalen

$$S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{2.8}$$

3×3 Sobel-Maske vier Gradientenbilder erzeugt. Die zwei horizontalen Gradientenbilder werden als $\frac{\delta I_{FLL}}{\delta i}$ und $\frac{\delta I_{DRR}}{\delta i}$ bezeichnet, die zwei vertikalen Gradientenbilder hingegen als $\frac{\delta I_{FLL}}{\delta j}$ und $\frac{\delta I_{DRR}}{\delta j}$. In Bild 2.4 ist ein aus dem Röntgenbild erzeugtes horizontales und vertikales Gradientendbild zu sehen. Diese Gradientendbilder sind das Ergebnis einer Faltung unter Zuhilfenahme

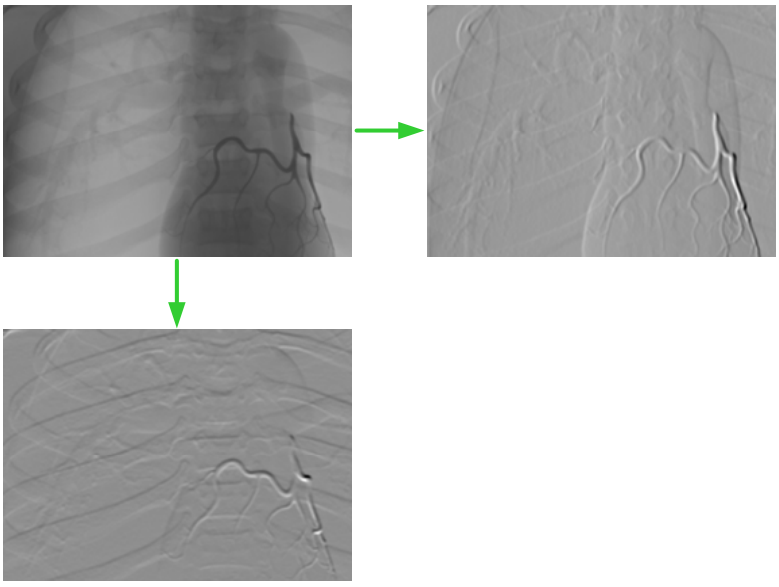

Bild 2.4: Aus dem Röntgenbild erzeugtes horizontales (rechts) und vertikales (unten) Gradientenbild

der gezeigten Sobel-Masken. Es wurden dazu alle Bildpunkte durchlaufen und dabei jeweils die Nachbarn des aktuellen Bildpunktes gemäß der durch die Sobel-Maske beschriebenen Gewichtung aufsummiert.

Die erzeugten Gradientenbilder dienen als Eingabe für die normalisierte Kreuzkorrelation[9]

$$S_{NCC}(I_{FLL}, I_{DRR}) = \frac{\sum_{(x,y)\in T} |(I_{FLL}(x,y) - \overline{I_{FLL}}) \cdot (I_{DRR}(x,y) - \overline{I_{DRR}})|}{\sqrt{\sum_{(x,y)\in T}(I_{FLL}(x,y) - \overline{I_{FLL}})^2} \cdot \sqrt{\sum_{(x,y)\in T}(I_{DRR}(x,y) - \overline{I_{DRR}})^2}} \tag{2.9}$$

, wobei $\overline{I_{FLL}}$ und $\overline{I_{DRR}}$ jeweils die Mittelwerte der Eingabebilder I_{FLL} und I_{DRR} sind [Kre08]. Der Mittelwert eines Bildes wird über

$$\overline{I_x} = \frac{1}{|T|} \sum_{(x,y)\in T} I(x,y) \tag{2.10}$$

berechnet. Alle Graustufen-Werte des Bildes werden aufsummiert und das Ergebnis anschließend durch die Anzahl der Bildpunkte $|T|$ geteilt. Laut [Kub08] handelt es sich bei NCC um ein gängiges eigenständiges Gütemaß, das bei vielen Registrierungsproblemen eingesetzt wird.

Histogramm-basierte Gütemaße Histogramm-basierte Gütemaße gehören zu den aufwändigeren Gütemaßen, liefern dafür jedoch im Allgemeinen auch gute Ergebnisse. Die Ähnlichkeit zweier Bilder wird hier nicht direkt anhand der Werte der Eingabebilder, sondern über eine Wahrscheinlichkeitsverteilung berechnet. Dazu werden Histogramme verwendet, anhand derer sich ablesen lässt, wie oft ein bestimmter Graustufen-Wert jeweils in den vorliegenden Bilddaten vorkommt. Bild 2.5 zeigt jeweils ein FLL- und DRR-Histogramm, sowie auch das Verbundhistogramm, ein 2-D-Histogramm beider Eingabebilder. Die Werte der Bildpunkte im Röntgenbild stellen hier einen Wert auf der X-Achse dar, die Werte der Bildpunkte im künstlichen Röntgenbild hingegen einen Wert auf der Y-Achse. Jedes Bildelement i der beiden Eingabebilder stellt demnach in Kombination eine 2-D-Koordinate $(I_{FLL}(i), I_{DRR}(i))$ dar. Anhand dieser Koordinate beschreibt ein Eintrag im Verbundhistogramm, wie häufig die spezifische Wertekombination in den Eingabebildern anzutreffen war. Die Wahrscheinlichkeit $p(x)$ lässt sich aus einem Histogramm ableiten, die Verbundwahrscheinlichkeit $p(x,y)$ hingegen aus einem Verbundhistogramm. Nachdem ein Histogramm aufgestellt wurde, werden dazu die einzelnen Einträge durch die Anzahl der Bildpunkte geteilt, um eine Wahrscheinlichkeitsverteilung mit Werten zwischen 0 und 1 zu erhalten.

Als Vertreter für Histogramm-basierte Gütemaße wurde in dieser Arbeit die *Normalisierte Transinformation*[10] gewählt. Wie in [Kub08] beschrieben, existieren zur Berechnung von *NMI*

[9] engl. Normalized Cross Correlation (NCC)
[10] engl. Normalized Mutual Information (NMI)

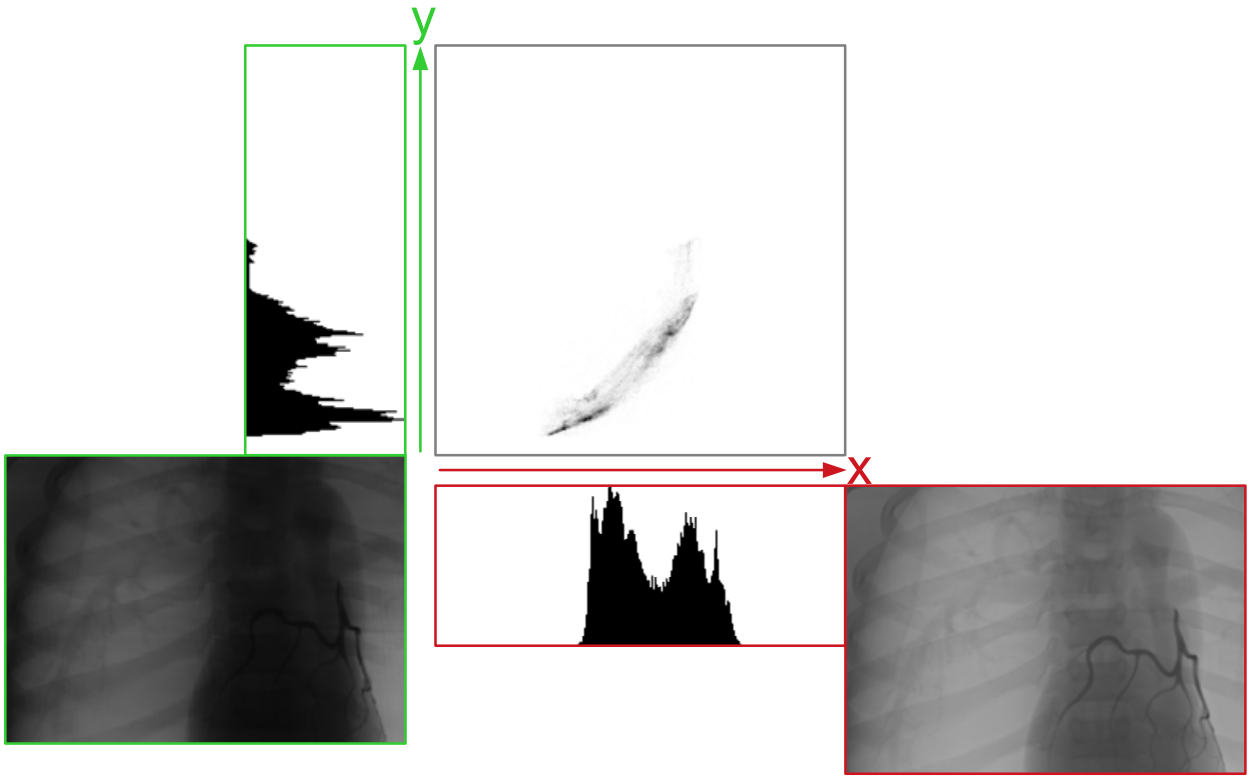

Bild 2.5: In der Mitte ist das FLL/DRR-Verbundhistogramm abgebildet, unten befindet sich das FLL-Histogramm (rot), links das DRR-Histogramm (grün)

verschiedene Ansätze. In dieser Arbeit wurde der als

$$S_{ECC}(I_{FLL}, I_{DRR}) = \sqrt{2 \cdot (1 - \frac{E(I_{FLL}, I_{DRR})}{E(I_{FLL}) + E(I_{DRR})})} \tag{2.11}$$

definierte *Entropie Korrelationskoeffizient*[11] mit Entropie $E(I_{FLL})$, $E(I_{DRR})$ und Verbund-Entropie[12] $E(I_{FLL}, I_{DRR})$ aus [Ast82] gewählt. ECC lässt sich auf *gegenseitige Information*[13]

$$S_{MI}(I_{FLL}, I_{DRR}) = E(I_{FLL}) + E(I_{DRR}) - E(I_{FLL}, I_{DRR}) \tag{2.12}$$

zurückführen. Die Entropie

$$E(I) = -\sum_{x} p(x) \cdot \log p(x) \tag{2.13}$$

[11] engl. Entropy Correlation Coefficient (ECC)
[12] engl. Joint Entropy (JE)
[13] engl. Mutual Information (MI)

von Shannon [Sha48] stammt ursprünglich aus der Informationstechnik. Die Verbund-Entropie hingegen ist als

$$E(I_{FLL}, I_{DRR}) = -\sum_{x}\sum_{y} p(x,y) \cdot \log p(x,y) \tag{2.14}$$

definiert. Laut [Kub08] kann die Verbund-Entropie ebenfalls als eigenständiges Gütemaß verwendet werden.

Nach Aussage von [Mae97] ist MI ein grundlegendes Konzept aus der Informationstheorie, welches die statistische Abhängigkeit zweier Zufallsvariablen oder den Informationsgehalt misst, den eine Variable über eine andere besitzt. Des Weiteren sei dieses Gütemaß sehr allgemein und mächtig, da keine Annahmen über die Natur der Abhängigkeit der Bildintensitäten getroffen wird.

Kombination mehrerer Gütemaße Um die Leistung der erstellten Implementation mit der anderen Arbeiten vergleichen zu können, wurden alle in diesem Kapitel beschriebenen Gütemaße implementiert. Da der Optimierungsschritt jedoch nur anhand eines einfachen numerischen Wertes arbeitet, besteht die Notwendigkeit, die verschiedenen berechneten Gütemaße in einem einzigen Gütemaß zu kombinieren. [Kub08] beschreibt, wie die verschiedenen Gütemaße normalisiert und kombiniert werden können. In der erstellten Implementation wurden die Gütemaße normalisiert und anschließend über fest eingestellte Gewichtungen aufsummiert. Da sich diese Arbeit auf die Realisierung einer schnellen 2-D/3-D-Registrierung auf der GPU konzentriert, wäre eine Vertiefung in dieses Thema nicht angemessen.

Kapitel 3

Grundlagen der GPU-Programmierung

In diesem Kapitel wird in Abschnitt 3.1 die geschichtliche Entwicklung von GPUs anhand einiger Meilensteine beschrieben.

Im Abschnitt 3.2 folgt ein Überblick über die GPGPU-Entwicklungsgeschichte, von den Anfängen bis hin zu den neuen und modernen Vertretern dieser Programmierschnittstellen.

Diese Arbeit verwendet OpenCL für die Programmierung der GPU. Für das bessere Verständnis der nachfolgenden Kapitel präsentiert der Abschnitt 3.3 die wichtigsten Konzepte von OpenCL.

3.1 GPU-Entwicklungsgeschichte

Seit den 1980er Jahren existieren Grafikkarten, die Text- und einfache Grafikausgaben übernehmen. Mitte der 1990er Jahre kamen die ersten 3-D-Beschleuniger hinzu und läuteten einen Wandel bei Grafikkarten ein. Die im Jahre 1999 erschienene *NVIDIA GeForce* brachte neben Geschwindigkeitsverbesserungen auch einen Umbruch bei Grafikprozessoren. Die Aufgabe von GPUs bestand zukünftig nicht mehr nur in der Rasterisierung von Dreiecken, es wurden ebenfalls Vertex-Berechnungen über eine neue Technik[1] von der CPU auf die GPU verlagert.

Während Entwickler die GPUs anfangs ausschließlich über eine sogenannte *Fixed-Function-Pipeline* ansprechen konnten, wurde die Grafik-Hardware im Laufe der Jahre schrittweise konfigurierbar und programmierbar. So wurde es in OpenGL mit der *GL_ARB_texture_env_combine* Erweiterung möglich, das Verhalten von *Multitexturing* genauer zu definieren. Damit ließen sich für damalige Verhältnisse überwältigende Effekte wie beispielsweise *Dot3 Bump Mapping* rea-

[1] engl. Transform and Lighting (T&L)

lisieren. Über die Zeit hinweg erschienen weitere Erweiterungen, so dass die Entwickler immer mehr Handlungsfreiraum bekamen.

Mit der *NVIDIA GeForce3*[2] war im Jahre 2001 der erste Grafikprozessor mit Unterstützung für Fragment- und Vertex-Shader nach DirectX 8 und OpenGL 1.3 verfügbar. Die Einführung von Shadern stellte einen Meilenstein der GPU-Programmierbarkeit dar. Der Grafikkartenhersteller ATI Technologies Inc. (ATI) zog kurz danach noch im gleichen Jahr mit seiner *ATI Radeon 8500*[3] nach, die erste über Shader programmierbare Hardware dieses Herstellers. Die von ATI als *SMARTSHADER* [ATI01] umworbene Shader-Technologie war in der Lage, Fragment-Shader mit bis zu 22 Instruktionen zu verarbeiten. Dies stellte eine deutliche Steigerung zu den in DirectX 8 spezifizierten 12 Instruktionen pro Fragment-Shader dar. Es vergingen zahlreiche GPU-Generationen, bis sich die ursprünglich sehr eingeschränkten und kleinen Assembler-Shader zu in Hochsprachen programmierbaren Shadern entwickelten. Mit dem Erscheinen des *Shader Model 4.0* fielen zahlreiche Einschränkungen weg, so dass mit Shadern heutzutage ein mächtiges Werkzeug zur Verfügung steht.

3.2 GPGPU-Entwicklungsgeschichte

Die Geburtsstunde von GPGPU lässt sich nicht eindeutig feststellen. Bereits vor dem Aufkommen von Shadern wurden vereinzelt Berechnungen, die wenig oder gar nichts mit Grafik gemeinsam hatten, auf der GPU realisiert. Jedoch wurden erst über programmierbare Shader mehr Einsatzmöglichkeiten für diesen Ansatz eröffnet.

In [Fer04] wird erläutert, wie über Fragment-Shader und mehrere Renderschritte parallele Reduktion auf der GPU realisiert werden kann. [Pha05] präsentiert zahlreiche Ansätze für die Umsetzung verschiedener Problemstellungen auf der GPU. Unter der Zuhilfenahme von Vertex-Shadern werden in [Sch07] Histogramme auf der GPU berechnet, [Ngu07] hingegen verwendet dafür Geometry-Shader.

Über viele Jahre hinweg konnten GPUs lediglich über Grafik-APIs wie DirectX oder OpenGL angesprochen werden. Berechnungen, die auf der GPU ausgeführt werden sollten, mussten daher derart umformuliert werden, dass diese sich über die vorgegebene Grafik-Pipeline realisieren ließen. Erschwerend kam hinzu, dass die für die Datenspeicherung verfügbaren Datentypen anfangs eingeschränkt waren, so war bspw. die Speicherung von Fließkommazahlen in Texturen lange Zeit keine Selbstverständlichkeit. Die Grafik-APIs und die Grafikkartentreiber

[2] Codename NV20
[3] Codename R200

sind auf die Darstellung von Grafik ausgelegt und das menschliche Auge toleriert zahlreiche Berechnungsfehler. Grafikkartentreiber bieten daher Einstellmöglichkeiten, um dem Benutzer die Möglichkeit zu geben, zwischen optischer Qualität und Ausführungsgeschwindigkeit zu wählen. GPGPU über klassische Grafik-APIs unterliegen folglich vielen Einflussfaktoren, bereits eine Aktualisierung des Grafikkartentreibers kann bewirken, dass bei den nächsten Berechnungen andere Ergebnisse herauskommen als zuvor. Der reale Nutzen von GPGPU war folglich noch eingeschränkt.

GPGPU Revolution Laut [AMD09b] veröffentlichte ATI im November 2006 die Low-Level-Programmierschnittstelle Close To Metal (CTM) und löste damit die GPGPU-Revolution aus. CTM stellte ein Gegenstück zu den klassischen Grafik-APIs DirectX und OpenGL dar und ermöglichte es Entwicklern, in einer Assembler-Sprache GPU Funktionalitäten direkt anzusprechen. Zeitgleich mit CTM stellte ATI eine neue GPU-Klasse vor, die speziell auf GPGPU ausgelegt wurde und unter anderem ohne Bildschirmausgabe daherkam. Dieser neue Grafikprozessor wurde *AMD FireStream* getauft, der neue Hardwaretyp im allgemeinen mit GPGPUs bezeichnet. Der große unmittelbare Erfolg blieb allerdings aus. Ein Jahr später veröffentlichte ATI das *ATI Stream SDK* [AMD09a], mit dem Entwicklern GPGPU einfacher gemacht werden sollte. Die in Compute Abstraction Layer (CAL) [AMD09a] umbenannte CTM-API wurde als Zwischensprache weiterentwickelt. Als High-Level-Programmierschnittstelle wurde die C-ähnliche Sprache *Brook+* hinzugefügt. Im gleichen Jahr wurde die zweite Generation von *AMD FireStream* vorgestellt, die nun ebenfalls Fließkommazahlen doppelter Genauigkeit in Hardware unterstützte [AMD07].

CUDA von NVIDIA wurde am 8. November 2006 zusammen mit dem neuen Grafikprozessor *G80* angekündigt [NVI06], zeitgleich mit der Veröffentlichung von ATIs CTM. Die erste öffentlich zugängliche Beta-Version des CUDA-Software Development Kit (SDK) erschien kurz danach am 16. Februar 2007 [NVI07a]. Laut einer Pressemitteilung von NVIDIA [NVI07b], wurde *CUDA SDK Version 1.0* am 16. Juni 2007 zeitgleich mit *Tesla* veröffentlicht, einem GPGPU-Grafikprozessor von NVIDIA. Die im Juni 2008 präsentierte *GeForce GTX 280*[4] war die erste NVIDIA-GPU mit Hardwareunterstützung für Fließkommazahlen doppelter Genauigkeit, die zweite Generation der Computing-Plattform Tesla wurde zeitgleich präsentiert [NVI08]. Am 22. August 2008 gab NVIDIA die Veröffentlichung von CUDA 2.0 bekannt. Eine der Neuerungen bestand in der hardwareseitigen Unterstützung für Fließkommazahlen mit doppelter Ge-

[4] Codename GT200, eine Weiterentwicklung des G80 Grafikprozessors

nauigkeit. Laut [Hal09] bezeichnete NVIDIA anfangs mit dem Namen CUDA die GPGPU-API, mittlerweile würde der Name allerdings für die grundlegende GPU-Architektur verwendet.

3.3 OpenCL

OpenCL ist eine moderne und noch recht junge Programmierschnittstelle für heterogene Parallelcomputer, die von Hardware, Betriebssystem und Hersteller unabhängig ist. Der neue Standard erhält von der Industrie eine breite Unterstützung. Bedeutende Firmen wie Apple, NVIDIA, ATI und Intel sind an dieser Technologie sehr interessiert und investieren entsprechend viele Ressourcen. Microsoft hingegen scheint eher daran interessiert zu sein, ihre neue *DirectCompute*-Schnittstelle für GPGPU durchzusetzen.

NVIDIA hatte bereits im Jahre 2007 die eigene proprietäre GPGPU-Schnittstelle CUDA veröffentlicht. Diese Technologie wird auch heute noch intensiv beworben und stetig weiterentwickelt, daher ist die Tatsache, dass gerade NVIDIA eine so aktive Rolle in der Entwicklung von OpenCL einnimmt, sehr interessant. Wie in [NVI09b] gezeigt, sind sich die *CUDA Driver API* und OpenCL sehr ähnlich. Daher ist es laut [AMD09d] recht einfach, bestehende Anwendungen, welche die CUDA-API verwenden, nach OpenCL zu portieren. Fast könnte der Eindruck entstehen, NVIDIA würde sich selbst Konkurrenz machen oder gar CUDA fallen lassen. Dem ist allerdings nicht so. Der NVIDIA-Konkurrent ATI besitzt mit *Brook+* ebenfalls seit etlichen Jahren eine eigene GPGPU-Schnittstelle und Hochsprache. Entwickler können diese Schnittstelle über das *ATI Stream SDK* nutzen. Der große Erfolg bliebt bisher jedoch aus.

Die Spezifikationsversion 1.0 des offenen OpenCL Standards [Gro09] wurde am 8. Dezember 2008 vom Industriegremium *Khronos Group* veröffentlicht. Öffentlich zugängliche OpenCL-Implementationen ließen allerdings einige Monate auf sich warten. Das Betriebssystem Mac OS X 10.6 (Snow Leopard) von Apple, das am 28. August 2009 veröffentlicht wurde, unterstützt OpenCL bereits vom Betriebssystem aus. Für Microsoft Windows und Linux muss OpenCL über einen Grafikkartentreiber installiert werden. Für viele Monate standen Entwicklern lediglich stellenweise langsame und fehlerbehaftete Beta-Treiber zur Verfügung. Ebenfalls boten NVIDIA und ATI zu den Beta-Treibern passende Beta-SDKs an. Mit diesen Beta-Versionen war es Entwicklern möglich, die ersten Schritte mit OpenCL zu machen, ohne auf fertige öffentliche Versionen warten zu müssen. Erst gegen Ende der Beta-Phase beider Hersteller wurden die OpenCL-SDKs des einen Herstellers jeweils mit den Treibern des anderen Herstellers kompatibel. Dadurch wurde es möglich, eine OpenCL-Anwendung ohne Neuübersetzung auf der Hardware von NVIDIA und ATI laufen zu lassen. Dies macht das Leben von Entwicklern deutlich einfacher,

da nicht für jede Hardware eine neue Implementation geschrieben muss, und stellt daher eines der zahlreichen Argumente dar, die für die Nutzung von OpenCL sprechen.

Der erste offizielle Grafikkartentreiber mit Unterstützung für OpenCL 1.0 wurde von NVIDIA veröffentlicht. Der am 26. November 2009 erschienene *195.62 WHQL* Treiber bot bereits zahlreiche OpenCL-Erweiterungen und eine gute Leistung. ATI zog am 15. Dezember 2009 mit dem Treiber *Catalyst_9.12 Hotfix* nach. Das erste offizielle OpenCL-SDK für Microsoft Windows und Linux wurde am 21. Dezember 2009 von ATI veröffentlicht. *ATI Stream SDK v2.0* bietet neben einer GPU-Implementation ebenfalls eine CPU-Implementation, es stehen folglich zwei unterschiedliche *Computing Device* Typen zur Verfügung. Über die CPU-Implementation ist es Entwicklern möglich, auch ohne ATI-Grafikkarte OpenCL-Anwendungen zu entwickeln. Dadurch sind - im Gegensatz zur CUDA-API - keine Emulatoren nötig. Das OpenCL-SDK von NVIDIA befand sich zum Zeitpunkt dieser Ausarbeitung noch immer in der Beta-Phase und bietet lediglich eine GPU-Implementation. Die SDKs der beiden Hersteller kommen mit zahlreichen Beispielen daher. Für die Entwicklung hochperformanter OpenCL-Anwendungen sind Hilfsmittel zur Leistungsanalyse hilfreich, daher werden im Anhang B einige OpenCL-Werkzeuge aufgelistet.

Zusammenhänge zwischen OpenCL, NVIDIA CUDA und ATI Stream Die Zusammenhänge zwischen OpenCL, NVIDIA CUDA und ATI Stream werden in Bild 3.1 verdeutlicht. Aus diesem Bild wird ersichtlich, dass NVIDIA weiterhin CUDA, ATI hingegen weiterhin *ATI Stream* nutzt.

OpenCL dient als universelle Schnittstelle für die verschiedenen Technologien, nicht als deren Ersatz. Programme werden in OpenCL C geschrieben, eine mit Erweiterungen für die Parallelisierung versehene Untermenge von ISO-C99. Der OpenCL C-Quellcode wird, je nachdem ob NVIDIA- oder ATI-Technologie verwendet wird, in die assemblerartige Zwischensprache Parallel Thread eXecution assembly language (PTX) [NVI09a] oder Intermediate Language (IL) [AMD09c] übersetzt.

Das in der Zwischensprache vorliegende Programm wird anschließend an den Treiber übergeben. Der Treiber übersetzt unter Zuhilfenahme eines Just In Time (JIT) Compilers das Programm in die Befehlssatzarchitektur[5] der jeweils verwendeten Hardware. Aus der *CL_DRIVER_VERSION*-Spalte der Tabellen im Anhang A ist zu entnehmen, dass jeder *Computing Device*-Typ einen eigenen Treiber besitzt, der die Zwischensprache in hardwarespezifischen Code übersetzt. Während bei NVIDIA schlichtweg der Grafikkartentreiber die Zwischenspra-

[5] engl. Instruction Set Architecture (ISA)

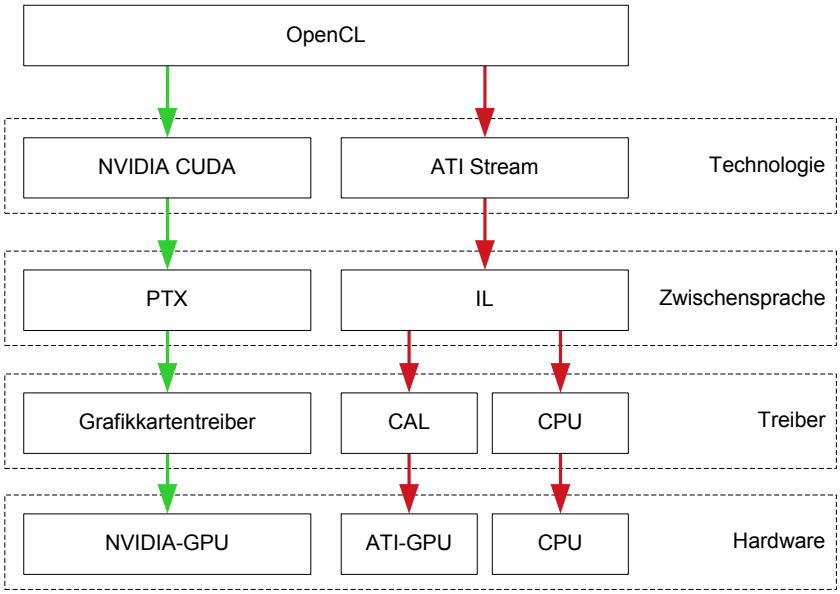

Bild 3.1: Zusammenhang zwischen OpenCL, NVIDIA CUDA und ATI Stream

che für die Hardware übersetzt [NVI09c], übernimmt bei ATI die Zwischenebene CAL diese Aufgabe.

Die Kernkonzepte von OpenCL werden in der Spezifikation [Gro09] anhand von vier Modellen beschrieben.

Plattformmodell In Bild 3.2 ist das OpenCL Plattformmodell visualisiert. Über dieses Modell werden die Zusammenhänge von Hardwarekomponenten losgelöst von konkreter Hardware beschrieben. Die Anwendungen selbst laufen auf dem *Host*, ein Überbegriff für unter anderem CPU und Arbeitsspeicher. Über OpenCL ausgelagerte parallele Arbeitsschritte der Anwendung werden auf einem oder mehreren OpenCL-fähigen Geräten ausgeführt. Die Spezifikation [Gro09] nennt als geeignete Geräte unter anderem CPUs, GPUs und andere Prozessoren, wie z. B. digitale Signalprozessoren[6] und den *Cell/B.E.* Prozessor.

Ein Gerät besitzt eine oder mehrere Recheneinheiten[7], Arbeiten werden automatisch auf diese verteilt. Auf der CUDA-Architektur entspricht eine Recheneinheit einem Streaming-Multiprozessor[8].

[6] engl. Digital Signal Processing (DSP)
[7] engl. Compute Unit (CU)
[8] engl. Streaming Multiprocessor (SM)

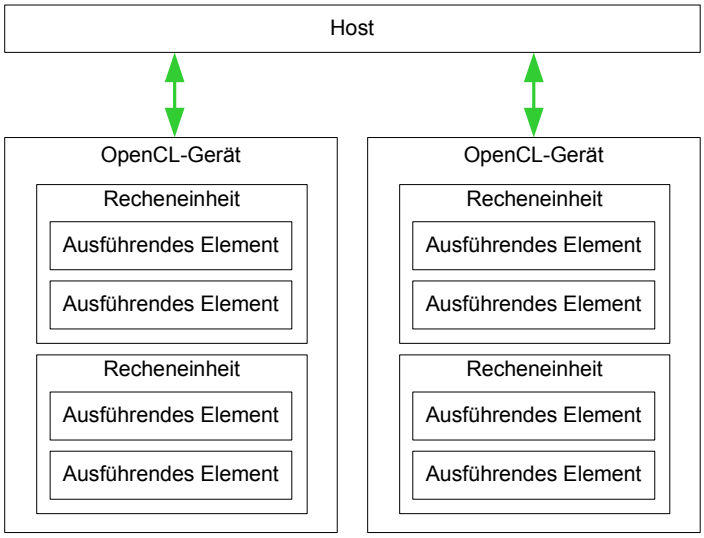

Bild 3.2: OpenCL Plattformmodell

Jede Recheneinheit besteht aus einem oder mehreren ausführenden Elementen[9], auf denen die Berechnungen durchgeführt werden. Auf NVIDIA-GPUs entspricht ein ausführendes Element einem *CUDA-Kern*.

Ausführungsmodell Die Ausführung von OpenCL-Programmen lässt sich in zwei Bereiche einteilen. Der Host ist im ersten Bereich angesiedelt, dieser übernimmt die Steuerung des Programmablaufes und versorgt die OpenCL-fähigen Geräte mit abzuarbeitenden Aufgaben. Das Gegenstück dazu bildet das auf dem OpenCL-Gerät ablaufende Programm, das als Kernel bezeichnet wird. Ein Kernel wird jeweils von einem Thread abgearbeitet, der in OpenCL als *Work-Item* bezeichnet wird. Auf GPUs sind Threads und deren Verwaltung über Hardware realisiert, daher läuft das umschalten der Threads vernachlässigbar schnell ab. Über die Programmiersprache OpenCL C wird vom Entwickler beschrieben, was ein *Work-Item* zu berechnen hat. Das geschriebene Programm wird automatisch über viele *Work-Item*s parallel abgearbeitet. Kernel und *Work-Item* sind wichtige Konzepte, die grundlegend für das Verständnis der weiteren Kapitel sind, daher soll im folgenden etwas näher darauf eingegangen werden.

Das, was innerhalb eines *Work-Item*s verarbeitet wird, bleibt dem Entwickler überlassen. Beispielsweise wäre es möglich, ein einziges *Work-Item* das gesamte zu bearbeitende Problem abarbeiten zu lassen. Dadurch würde das Problem wie auf der CPU sequentiell abgearbeitet werden. Massiv parallele Hardware derart zu unterfordern, kann allerdings nicht im Interesse eines Entwicklers liegen.

[9] engl. Processing Element (PE)

Ein anschauliches Beispiel für eine sinnvolle parallele Verarbeitung ist die Verarbeitung einzelner Bildpunkte eines Bildes. Soll beispielsweise die Farbe eines jeden Bildpunktes des Bildes aufgehellt werden, so bietet es sich an, dass ein *Work-Item* jeweils einen Bildpunkt verarbeitet. Die Größe des Bildes stellt die Problemgröße da, dies wird in OpenCL als *globale Größe* bezeichnet. Ein Bild stellt ein 2-D-Problem da, ebenfalls werden 1-D- und 3-D-Probleme von OpenCL direkt unterstützt. Die verschiedenen vorhandenen Dimensionen erleichtern die Entwicklung, Problemstellungen müssen nicht umständlich umformuliert werden.

In Bild 3.3 ist das Ausführungsmodell von OpenCL visualisiert. Die Abbildung der glo-

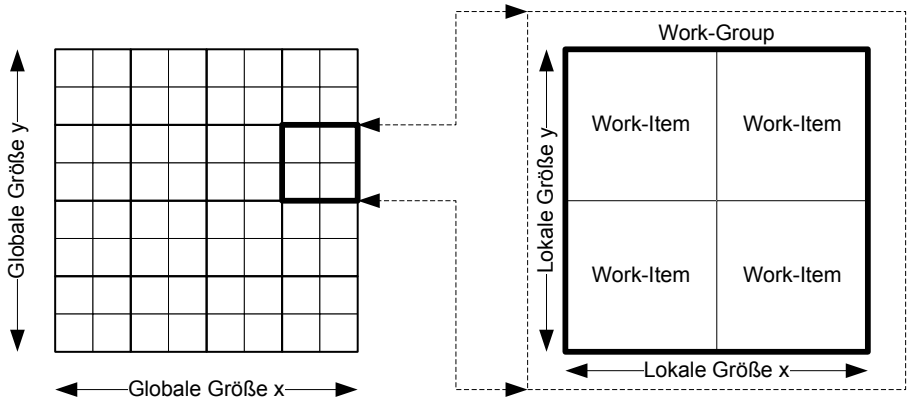

Bild 3.3: OpenCL Ausführungsmodell nach [Gro09]

balen Größe auf der linken Seite wurde bereits beschrieben. Neu ist der rechte Teil der Abbildung: Aus hardwaretechnischen Gründen werden *Work-Item*s zu *Work-Group*s zusammengefasst, deren Größe *lokale Größe* genannt wird. Ein Entwickler muss sich nicht direkt mit *Work-Group*s befassen, diese Zusammenfassung von *Work-Item*s kann allerdings bei bestimmten Problemstellungen von Vorteil sein. Aus diesem Grund bietet OpenCL für die Kernel-Entwicklung nicht nur Funktionen an, welche die Position des aktuellen *Work-Item*s innerhalb der globalen Größe zurückgeben, sondern es existieren ebenfalls Funktionen, welche die Position des aktuellen *Work-Item*s innerhalb der lokalen Größe als Ergebnis liefern.

Programmiermodell OpenCL unterstützt *Daten-parallele* und *Aufgaben-parallele* Verarbeitung. Die gleichzeitige Verarbeitung von Daten geschieht in OpenCL über die bereits im vorherigen Abschnitt *Ausführungsmodell* erläuterten *Work-Item*s.

Aufgaben werden in OpenCL über eine Befehlswarteschlange[10] an die OpenCL-Geräte vergeben. Standardmäßig erfolgt die Abarbeitung dieser Aufgaben sequentiell in der übergebenen

[10]engl. Command Queue

Reihenfolge. Falls von der verwendeten Hardware unterstützt, ist es ebenfalls möglich, Aufgaben parallel verarbeiten zu lassen. Dies ermöglicht eine noch bessere Auslastung der Hardware, stellt jedoch weitere Herausforderungen an den Entwickler. Bei der Entwicklung muss nun berücksichtigt werden, ob Ergebnisse von vorherigen Aufgaben für darauf folgenden Aufgaben benötigt werden. Gegebenenfalls muss eine Synchronisation von Aufgaben stattfinden.

Speichermodell In Bild 3.4 ist das Speichermodell von OpenCL visualisiert. Nicht abgebildet sind die Register, die unter den *Work-Item*s aufgeteilt werden sowie Texturen, die in OpenCL als *Bild*[11] bezeichnet werden. Unter dem globalen Speicher ist der umfangreiche Grafikkarten-

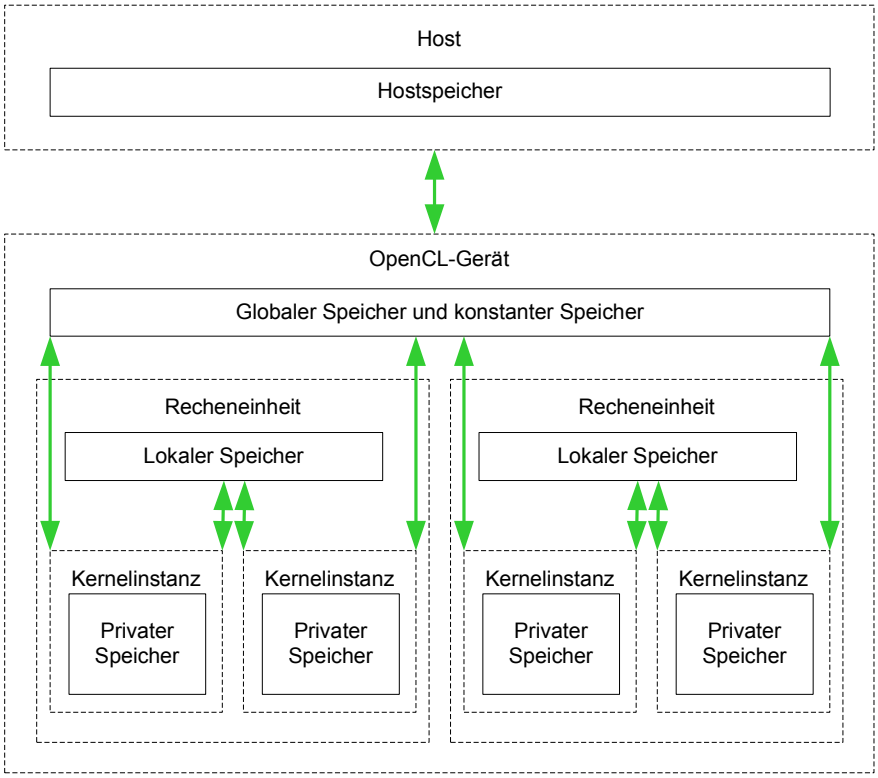

Bild 3.4: OpenCL Speichermodell

speicher zu verstehen, dieser kann mit dem Heap der Programmiersprache C verglichen werden. Beim lokalen Speicher handelt es sich, in der Regel zumindest, um schnellen Speicher, der von allen *Work-Item*s innerhalb einer *Work-Group* verwendet werden kann. Die Größe des lokalen Speichers beträgt auf aktuellen GPUs nur wenige Kilobyte, findet dafür allerdings auch nicht bei jeder Problemstellung Einsatz. Der lokale Speicher muss zwar vom Entwickler auf der Host-Seite eingerichtet werden, dessen Inhalt kann von der Host-Seite aus allerdings nicht gelesen oder

[11] engl. Image

geschrieben werden. Die Größe des schnellen konstanten Speichers ist ebenfalls eingeschränkt, kann aber von allen *Work-Item*s unabhängig von *Work-Group*s lesend genutzt werden. Der Inhalt des konstanten Speichers wird vom Entwickler von der Host-Seite aus angelegt und mit Werten gefüllt. Register und privater Speicher sind weder vom Host, noch über die Hochsprache *OpenCL C* vom Entwickler direkt ansprechbar. Der private Speicher wird laut [NVI09c] zum Auslagern von Registern[12], für Funktionsaufrufe und für die automatische Allokation von Arrays verwendet und kann daher im Grunde mit dem Laufzeitstack aus der Programmiersprache C verglichen werden. GPUs der aktuellen Generation besitzen keine CPU ähnlichen Caches, daher sollten die in [NVI09d] beschriebenen Hinweise und Regeln beachtet werden, damit Speicherzugriffe nicht zu schnell zum Flaschenhals werden.

[Wan09] nennt beispielhafte Geschwindigkeiten für die Datenübertragung zwischen Host und globalem Speicher, ebenfalls werden Latenzzeiten der anderen Speicherzugriffe angegeben. Die Geschwindigkeit der Datenübertragung zwischen Host und globalem GPU-Speicher wird mit 8 GB/s (PCI-e, x16 Gen2) angegeben, während die Zugriffsgeschwindigkeit auf den globalen Speicher mit 141 GB/s (GTX 280) angegeben wird. Es sei zudem besser, Datenübertragungen zu bündeln, anstatt viele kleinere Datenübertragungen auszuführen. Die Latenz des globalen Speichers wird mit 400-800 Taktzyklen angegeben, während die Latenz des lokalen Speichers etwa 100 mal kleiner sei als die des globalen Speichers. Die Latenz des privaten Speichers wird mit 24 Taktzyklen angegeben.

Die Datenübertragungsraten zwischen den einzelnen Speicherarten verdeutlicht, wieso die Nutzung einer Speicherart einer anderen vorzuziehen ist. Der Datentransfer zwischen Host und Grafikkarte sollte auf ein Minimum reduziert werden. Wenn möglich sollten innerhalb eines Kernels Daten vom globalen Speicher in den lokalen Speicher geladen werden, dies lohnt sich allerdings nur, wenn innerhalb des Kernels mehrmals auf diese Daten zugegriffen wird.

[12] engl. Register Spills

Kapitel 4

Realisierung der 2-D/3-D-Registrierung auf der GPU

Dieses Kapitel beschäftigt sich mit der Realisierung der 2-D/3-D-Registrierung auf der GPU. Da es zu umfangreich wäre, die gesamte im Rahmen dieser Arbeit erstellte Implementation bis ins Detail zu erörtern, und eine solche Detaillierung nur wenig an Mehrwert bieten würde, wurde ein anderer Ansatz gewählt. Anhand einer abstrakten Sichtweise sollen die verschiedene Prinzipien vermittelt werden, anhand derer die Umsetzung des Problems auf der GPU erfolgte. Für Themen, bei denen es angebracht scheint, werden hingegen auch Details der realisierten 2-D/3-D-Registrierung dargelegt.

Viele in der 2-D/3-D-Registrierung nötigen Teilaufgaben lassen sich auf ein gemeinsames Grundproblem reduzieren. Bei der GPU handelt es sich um Hardware mit einer parallelen Architektur. Die Hauptaufgabe bei der Programmierung der GPU besteht folglich darin, geeignete parallelisierbare Algorithmen zu finden. Daher sollen im Abschnitt 4.1 grundlegende Problemstellungen unabhängig von der 2-D/3-D-Registrierung beschrieben werden.

Die für die realitätsnahe DRR-Erzeugung nötigen Arbeitsschritte und gewählten Algorithmen werden in Abschnitt 4.2 dargelegt.

Auf die Realisierung der verschiedenen Gütemaße soll in Abschnitt 4.3 eingegangen werden. Die Implementation der Gütemaße ist lediglich eine Kombination bereits vorgestellter Algorithmen. Die Berechnung von Verbundhistogrammen auf der GPU hingegen stellt eine besondere Herausforderung da, daher soll darauf näher eingegangen werden.

4.1 Parallelisierung der 2-D/3-D-Registrierung

Die in der 2-D/3-D-Registrierung verwendeten Problemstellungen lassen sich in allgemeinere Problemklassen abstrahieren, die eigentliche Herausforderung besteht in der effizienten Parallelisierung dieser allgemeinen Problemklassen. In der erstellten Implementation werden einige grundsätzliche Strategien verwendet, auf die im Folgenden genauer eingegangen werden soll.

Abstrakte Sicht Laut [Pha05] ist es sinnvoll, bei den GPU-Kommunikationstypen zwischen *Gather* und *Scatter* zu unterscheiden. GPUs kommen mit Problemen, die darauf basieren, Daten zu akkumulieren, sehr gut zurecht, da hier eine Parallelisierung ohne größere Probleme möglich ist. Diese Probleme werden *Gather*-Probleme genannt. Bei den *Scatter*-Problemen dagegen werden Daten verteilt, mehrere Elemente können folglich beispielsweise in eine gleiche Speicheradresse schreiben. Eine sequentielle Implementierung solcher Probleme ist zwar einfach, eine Parallelisierung kann hier allerdings schnell komplex und ineffizient werden. Ein Ansatz besteht darin, Methoden der Synchronisation zur Vermeidung von Schreibkonflikten zu nutzen. Dies kann allerdings dazu führen, dass im schlimmsten Fall, wenn alle Work-Items in eine gleiche Speicheradresse schreiben, die Ausführung dadurch wieder sequentiell abläuft und somit kein Gewinn durch die Parallelisierung zu erreichen ist. Dieser Ansatz führt dazu, dass die erzielbare Leistung abhängig von den eingegebenen Daten ist. Bei wenig bis keinen Zugriffskonflikten wird die Implementation leistungsfähiger sein als im erwähnten schlimmsten Fall. Andere Ansätze zerlegen das zu parallelisierende Problem in Teilprobleme, die in einem abschließenden Schritt zusammengeführt werden. Dadurch wird die Notwendigkeit von Synchronisation vermieden und die Leistung der Implementation wird unabhängig von den eingegeben Daten. Dieser Ansatz kann allerdings in der Implementation recht komplex werden sowie Einschränkungen mit sich bringen, was den bearbeiten Datenumfang betrifft. *Scatter*-Probleme auf GPUs zu Implementieren ist daher generell nicht einfach zu lösen. Bild 4.1 zeigt den Unterschied zwischen *Gather*- und *Scatter*-Problemen, dies ist vergleichbar zum Lesen von beliebigen Stellen und dem Schreiben in beliebige Stellen.

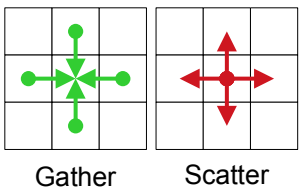

Bild 4.1: Unterschied von *Gather*- und *Scatter*-Problemen

Reduktion In den zur Ähnlichkeitsbestimmung verwendeten Gütemaßen nimmt die Summenbildung eine zentrale Stellung ein, keines der verwendeten Gütemaße kommt ohne die Berechnung einer Summe aus. Beim Bilden einer Summe kann verallgemeinert von Reduktion gesprochen werden. Die Reduktion gehört in die *Scatter*-Problemklasse. Das Problem lässt sich allerdings umformulieren, so dass möglich ist, Reduktion als *Gather*-Problem zu behandeln. Durch diese Umformulierung wird eine effiziente Parallelisierung auf der GPU möglich. Eine effiziente parallelisierte Reduktion wird in [Har07] vorgestellt und schrittweise für die GPU optimiert. Die im Rahmen dieser Arbeit erstellte Implementation nutzt die dort vorgestellte Lösung des Problems. Die Eingabedaten werden dabei in Blöcke unterteilt, die anschließend jeweils in einem baumbasierenden Verfahren weiterverarbeitet werden. Im Folgenden wird der Ablauf in-

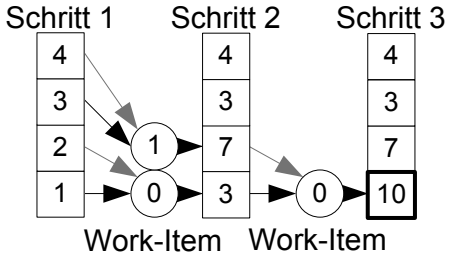

Bild 4.2: Parallele Reduktion

nerhalb eines Blockes der Größe n erläutert. Mehrere Blöcke werden parallel auf die gleiche Art und Weise verarbeitet, jeder Block wird von jeweils n Work-Items parallel bearbeitet. In einem ersten Schritt lädt jedes Work-Item einen Wert in den lokalen Speicher, der n Elemente beinhaltet, ein Element pro Work-Item. Diese Werte können entweder direkt aus einem Speicher gelesen, oder durch Berechnungen erzeugt werden. Für den nächsten Arbeitsschritt ist es von entscheidender Bedeutung, dass der lokale Speicher vollständig gefüllt wurde. Dies wird über die OpenCL-Anweisung *barrier(CLK_LOCAL_MEM_FENCE)* sichergestellt. Hierbei handelt es sich um eine Barriere, durch die jedes Work-Item solange blockiert wird, bis alle anderen Work-Items der zugehörigen Work-Group ebenfalls an der Barriere angelangt sind. Es handelt sich folglich um eine Synchronisation. Nun folgt der eigentliche, in Bild 4.2 abgebildete, Reduktionsschritt. Jeweils ein Work-Item addiert zwei Werte aus dem lokalen Speicher auf und speichert das Ergebnis wiederum im lokalen Speicher. Anschließend ist wiederum eine Synchronisation nötig, da die folgenden Schritte auf dem veränderten Inhalt des lokalen Speichers arbeiten. Dieser Schritt wird mehrmals wiederholt, die Anzahl der beschäftigten Work-Items halbiert sich in jedem Schritt. Am Ende ist lediglich nur noch ein Work-Item beschäftigt, welches am Ende das Ergebnis der gesamten Block-Reduktion zurück in den globalen Speicher schreibt. Nach der parallelen Reduktion liegen alle Block-Ergebnisse vor und können nun entweder durch weitere

parallele Reduktionen solange verarbeitet werden, bis das Endergebnis vorliegt, oder aber das Endergebnis wird auf der CPU berechnet. Der Einfachheit halber wurde das Endergebnis in der erstellten Implementation auf der CPU berechnet, da die GPU mit kleinen Datenmengen ohnehin nicht sonderlich effektiv zurecht kommt.

Der verwendete OpenCL C-Quellcode 4.1 für die parallele Reduktion wird in dieser Form auch an anderen Stellen in der erstellten Implementation verwendet. Der Hauptunterschied liegt dabei jeweils in der *calculate*-Funktion, die pro Work-Item eine Berechnung durchführt und den Wert anschließend im lokalen Speicher ablegt. Anhand dieses OpenCL C-Quellcodes ist zu

```
1  __kernel void reduction(__global uint4 *result, __local uint4 *
       localMemory, uint numOfElements, __global uchar *dataFLL,
       __global uchar *dataDRR, int thresholdSDT)
2  {
3    // Daten des aktuellen Work-Item abfragen
4    uint tid = get_local_id(0);
5    uint gid = get_global_id(0);
6    uint localSize = get_local_size(0);
7    // Lokalen Speicher füllen
8    if (gid < numOfElements) {
9      localMemory[tid] = calculate(gid, dataFLL, dataDRR,
         thresholdSDT);
10   } else {
11     // Null beeinflusst die Summe nicht
12     localMemory[tid] = 0;
13   }
14   // Reduktion im lokalen Speicher (>>1 = /2)
15   barrier(CLK_LOCAL_MEM_FENCE);
16   for (uint s=localSize>>1; s>0; s>>=1) {
17     if (tid < s)
18       localMemory[tid] += localMemory[tid + s];
19     barrier(CLK_LOCAL_MEM_FENCE);
20   }
21   // Ergebnis des Blocks in globalen Speicher schreiben
22   if (tid == 0)
23     result[get_group_id(0)] = localMemory[0];
24 }
```

Quellcode 4.1: Parallele Reduktion in OpenCL

erkennen, dass die GPU-Programmierung in modernen GPGPU-Programmiersprachen anders abläuft, als bei der Verwerndung von Grafik-APIs wie beispielsweise OpenGL. Anstatt Grafik-

Funktionalitäten wie z. B. Mipmaps von Texturen für die Reduktion zweckzuentfremden, wie dies in früheren Ansätzen nötig war, lassen sich in OpenCL Problemstellungen oftmals durch den Einsatz von in der Informatik bekannten Verfahren realisieren.

Vorbedingungen Exakte Vorbedingungen an eine Aufgabe zu stellen, kann für eine Parallelisierung hilfreich sein. Gemäß dem *Blackbox*-Prinzip bekommt der Anwender nichts von der internen Implementierung mit. Dabei ist es wichtig, Eingaben und Ausgaben genau zu spezifizieren, um Optimierungsmöglichkeiten zu identifizieren. Der Registrierungs-Schritt bekommt fest definierte Eingaben und liefert als Ergebnis eine Transformation zurück. Dadurch besteht die Möglichkeit, beispielsweise ein künstliches Röntgenbild während der Registrierung in einer anderen Form zu erzeugen, als das künstliche Röntgenbild, welches ein Benutzer am Ende zu sehen bekommt. Im eigentlichen Registrierungs-Schritt wird nur die Transformation verändert, alle anderen Daten wie beispielsweise das Röntgenbild oder das Volumen ändern sich hingegen nicht. Hierdurch besteht die Möglichkeit, diverse während der Registrierung mehrmals benötigte Daten einmalig in eine optimale Form zu bringen oder gar nur einmalig zu berechnen.

Vorberechnungen Weniger umfangreiche Aufgaben sind im Allgemeinen einfacher zu parallelisieren. Wo immer es möglich ist, sollten Daten daher vor der eigentlichen Registrierung einmalig vorberechnet werden. Moderne GPUs besitzen zwar eine große Rechenleistung, aber Berechnungen, die nur einmalig ausgeführt werden, sind weiterhin deutlich schneller. Aus diesem Grunde lohnt ein kritischer Blick auf die auszuführenden Berechnungen im Hinblick darauf, wo Berechnungen eingespart werden können.

Während des Volume-Raytracings werden Berechnungen auf den aus dem Volumen ausgelesenen Voxel-Werten ausgeführt. Diese Volumendaten werden ausschließlich für das Erzeugen des künstlichen Röntgenbildes benötigt. Während der 2-D/3-D-Registrierung ändert sich das Volumen nicht, aufgrund dessen ist es sinnvoll, die Volumendaten vor der eigentlichen Registrierung einmalig in den benötigten Wertebereich umzurechnen, anstatt dies am Anfang jedes Berechnungsschrittes auszuführen. Dabei handelt es sich um eine Transformation der Volumenwerte von der Hounsfield-Skala in den Graustufenbereich anhand einer festgelegten Übertragungsfunktion. Während der zahlreichen Schritte beim Volume-Raytracing, die im Rahmen einer 2-D/3-D-Registrierung benötigt werden, beschränkt sich die Arbeit pro Volume-Raytracing daher nur noch auf das Auslesen und Addieren von Voxel-Daten. Sollten sich die Werte dieser einmaligen Berechnung im Laufe des Programmes doch einmal ändern, so wäre es denkbar, einen OpenCL-Kernel hinzuzufügen, der ausschließlich die Volumendaten aktualisiert. Diese Vorberechnung macht es allerdings nötig, die Voxel-Daten nun als Fließkommazahlen zu speichern,

anstatt wie beim Wertebereich der *Hounsfield-Skala* in Werten von 16 bit pro Voxel. Aktuelle GPUs arbeiten intern mit 32 bit, daher sind Fließkommazahlen mit einfacher Genauigkeit ohnehin besser geeignet als ein 16 bit-Datentyp. Zwar wird dadurch für das ohnehin schon speicherintensive Volumen das Doppelte an Speicher benötigt, allerdings sind Grafikkartenspeicher von 1024 GiB oder mehr heutzutage bereits nicht mehr unüblich.

Das gegebene Röntgenbild ist während der Registrierung statisch. Aus diesem Grunde wird anhand einer gegebenen ROI ein neues 2-D-Teilbild erzeugt, so dass die restliche Implementation keine ROI für dieses Röntgenbild berücksichtigen muss. Die für das Gütemaß GC benötigten Gradienten-Daten des statischen Röntgenbildes lassen sich vor der 2-D/3-D-Registrierung einmalig vorberechnen. Dadurch besteht während der 2-D/3-D-Registrierung nur noch die Notwendigkeit, die Gradienten-Daten des dynamischen künstlichen Röntgenbildes zu berechnen. Bei NMI, einem Histogramm-basierten Gütemaß, lässt sich die Entropie des Röntgenbildes vor der 2-D/3-D-Registrierung einmalig vorberechnen. Während der 2-D/3-D-Registrierung muss nur noch das Verbundhistrogramm erstellt und daraus anschließend die Entropie des künstlichen Röntgenbildes und die Verbund-Entropie ermittelt werden.

Datentypen Gerade beim Erstellen einer GPU-basierten Implementation ist es entscheidend, die technischen Rahmenbedingungen zu kennen. Im Folgenden wird daher auf die zur Verfügung stehenden Datentypen eingegangen, da die Kenntnisse darüber im Rahmen einer GPU-Implementation wichtig sind.

Aktuell erhältliche GPUs besitzen deutlich mehr Verarbeitungseinheiten für Fließkommazahlen mit einfacher Genauigkeit als für Fließkommazahlen mit doppelter Genauigkeit. Laut [Hal09] sind selbst auf der kommenden Fermi-Architektur Fließkommazahlen mit doppelter Genauigkeit nur halb so schnell wie Fließkommazahlen mit einfacher Genauigkeit. Daher wurden für die erstellte Implementation Fließkommazahlen mit einfacher Genauigkeit genutzt, um die GPU-Leistung voll ausschöpfen zu können.

Laut [NVI09d] können pro Takt acht Fließkommazahlen addiert werden, dies treffe ebenfalls auf Integerzahlen zu. Bei der Multiplikation sei es möglich, pro Takt acht Fließkommazahlen zu verarbeiten, jedoch nur zwei 32 bit-Integerzahlen. Um pro Takt acht Integerzahlen multiplizieren zu können, müsse die *mul24*-Anweisung genutzt werden. Diese Funktion multipliziert 24 bit-Integerzahlen, was meistens ausreichend ist. Es wird allerdings davor gewarnt, dass *mul24* auf zukünftigen Architekturen langsamer sein wird als eine normale 32 bit-Integer-Multiplikation. Laut [NVI09c] und [NVI10] besitzt bereits die kommende neue GPU-Generation von NVIDIA neben der Einheit für Fließkommazahlen eine vollwertige 32 bit-Integer-Einheit. Dadurch gibt

es zukünftig keinen technisch bedingten Grund mehr, Integerzahlen künstlich über Fließkommazahlen zu verarbeiten oder umständlich spezielle Befehle zu nutzen, um die Ausführgeschwindigkeit zu erhöhen. Um zukunftssicher zu sein, nutzt die erstellte Implementation daher dort, wo Integerzahlen angebracht sind, diesen Datentyp. Dies macht die Implementation nicht unnötig umständlicher, und Rechenungenauigkeiten wie bei den Fließkommazahlen existieren bei Integerzahlen ebenfalls nicht.

4.2 DRR

Die realitätsnahe künstliche Röntgenbild-Erzeugung, kurz DRR, lässt sich in drei wesentliche Arbeitsschritte unterteilen. Das Volume-Raytracing erzeugt anhand gegebener Parameter eine 2-D-Abbildung der Volumendaten. Die Daten dieses Schrittes liegen jedoch noch in Fließkommazahlen vor, weshalb sie so noch nicht direkt mit dem Röntgenbild verglichen werden können. Ein zweiter Arbeitsschritt ermittelt daher den Helligkeitswert der erzeugten Daten, mit dessen Hilfe in einem dritten Schritt die Fließkommazahlen in eine Form überführt werden, die mit einem Röntgenbild verglichen werden können. Im Folgenden soll näher auf die Realisierung der einzelnen Schritte auf der GPU eingegangen werden.

Benötigter DRR-Bereich Die Erzeugung des DRR ist sehr rechenaufwändig, daher sollte nur der Bereich des DRR berechnet werden, der wirklich benötigt wird. In Bild 4.3 sind die verschiedenen relevanten Bildbereiche zu sehen. Das innere Drittel des DRR wird dazu benötigt, die Helligkeit des Bildes zu ermitteln. Die ROI ist der Bereich, in dem die Gütemaße berechnet werden. Es besteht daher die Notwendigkeit, einen DRR-Bereich zu berechnen, der sowohl das innere Drittel als auch die ROI umschließt. Die Berechnungen innerhalb dieses Rechtecks könnten, je nachdem wie die einzelnen Bereiche liegen, weiter optimiert werden. Das Berechnen von DRR-Bildpunkten, die zwar im Rechteck liegen, aber weder für das innere Drittel, noch für die ROI benötigt werden, könnte durch Abfragen übersprungen werden.

Volume-Raytracing Beim Volume-Raytracing (siehe Bild 4.4) handelt es sich um ein direktes Volume-Rendering. Während bei indirekten Methoden des Volume-Renderings die Volumendaten in ein polygonales Modell überführt werden, das anschließend gerendert wird, wird beim direkten Volume-Rendering das Volumen ohne Umwege direkt anhand der Volumendaten gerendert. Nach [Krü03] können die Volumendaten dazu in einer 3-D-Textur gespeichert werden, anschliessend könne das Volumen mittels einer Grafik-API bspw. mit Hilfe von *3D-Texture-*

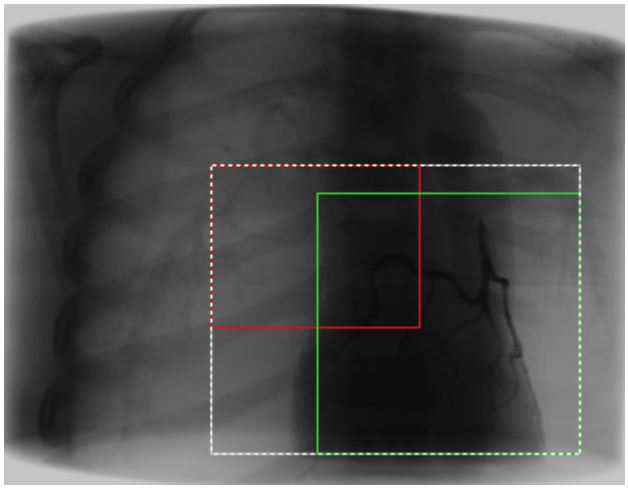

Bild 4.3: Relevante Bildbereiche: Inneres Drittel (rot), ROI (grün), benötigter DRR-Bereich (weiß)

Slicing gerendert werden. [Krü03] führt weiter aus, wie ein Volumen durch Volume-Raytracing gerendert werden kann. Dabei wird für jeden Bildpunkt ein Strahl durch das Volumen geschickt, wobei die Voxel-Daten aufsummiert werden. Das Volume-Raycasting verfolgt den gleichen Ansatz, jedoch wird ein Strahl nicht durch das gesamte Volumen geführt, sondern von Voxel-Daten über einem bestimmten Schwellenwert geblockt.

In der erstellten OpenCL-Implementation wird Volume-Raytracing verwendet. Des Weiteren werden die Volumendaten, sofern von der Hardware unterstützt, in 3-D-Texturen gespeichert. Dies ermöglicht der GPU einen effektiven Zugriff auf die Volumendaten, falls gewünscht kann die GPU auf diese Weise ebenfalls gefilterte Voxel-Werte zurückliefern. Beim Volume-Raytracing handelt es sich um ein *Gather*-Problem. Jeder Bildpunkt wird getrennt von den restlichen Bildpunkten berechnet, aufgrund dessen wird im Folgenden lediglich ein einziger Bildpunkt betrachtet. Die GPU berechnet anschließend viele Bildpunkte parallel.

Schnelle Schnittpunktberechnung beim Volume-Raytracing Der erste Schritt beim Volume-Raytracing besteht darin zu ermitteln, an welcher Stelle der Strahl des aktuellen Bildpunktes das Volumen betritt und wo er wieder austritt. Eine schnelle Schnittpunktberechnung für das Volume-Raytracing unter Zuhilfenahme einer traditionellen Grafik-API wird in [Krü03] beschrieben. Das Verfahren arbeitet mit mehreren Renderschritten unter Zuhilfenahme des Renderns in Texturen. Über Fragment-Shader werden dann die Daten der dynamisch erzeugten Texturen ausgelesen und daraus die Start- und Endpositionen einzelner Strahlen berechnet.

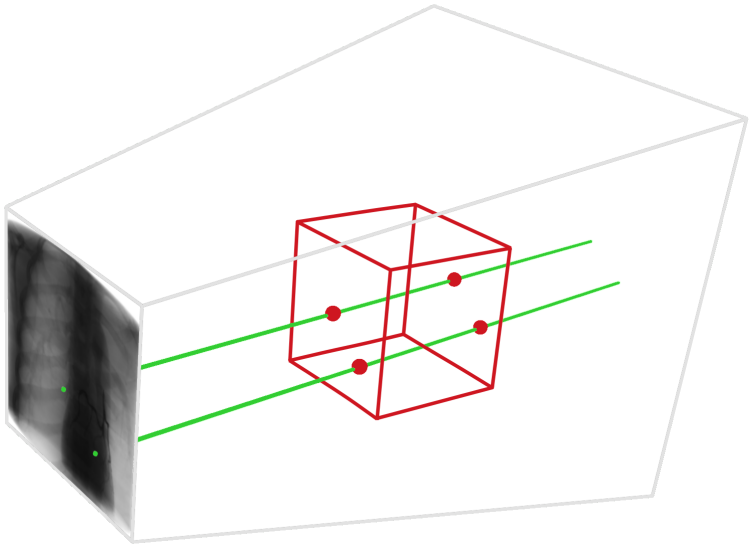

Bild 4.4: Volume-Raytracing

Für die OpenCL-Implementation wurde ein anderer Ansatz gewählt. So wird eine Strahl/Würfel-Schnittpunktberechnung mit Hilfe der *Slab*-Methode von Kay und Kayjia verwendet [Kay86]. Der Begriff *Slab* steht dabei für den Raum zwischen zwei parallelen Ebenen, ein Würfel ist daher durch eine Schnittmenge von drei *Slabs* definiert. Dies entspricht einer Ebenenmenge mit sechs Elementen. Die Berechnung des Intervalls $[tfar, tnear]$ eines *Slabs* benötigt zwei Strahl/Ebene-Schnittpunktberechnungen. In der erstellten Implementation reduzieren sich diese Berechnungen auf einfache größer/kleiner-Vergleiche, was dadurch ermöglicht wird, dass der Strahl jedes Bildpunktes des zu erzeugenden DRR in das lokale Koordinatensystem des Volumens transformiert wird. Insgesamt werden so zwar mehr Berechnungen im Volume-Raytracing-Schritt benötigt, als in der in [Krü03] vorgestellten Methode, dafür sind allerdings keine Lese/Schreib-Zugriffe auf den globalen Speicher nötig. Berechnungen auf der GPU sind sehr schnell, während der Zugriff auf den globalen Speicher langsam ist. Die Rechenleistung jeder neuen GPU-Generation wächst zudem schneller als die Datenübertragungsraten. Ebenfalls wäre es mühselig, Verfahren, die auf die Grafik-Pipeline ausgelegt sind, in OpenCL zu implementieren.

Weg durch das Volumen Nach der Schnittpunktberechnung ist bekannt, an welcher Koordinate der aktuelle Strahl das Volumen betritt und an welcher Koordinate es wieder verlassen wird. Innerhalb einer Schleife wird nun der Weg des Strahles durch das Volumen verfolgt. Die Schrittweite der Strahlverfolgung wurde dazu anhand der Volumengröße festgelegt, die Anzahl

der Schritte durch das Volumen muss hingegen dynamisch berechnet werden. Die einzelnen abgetasten Werte werden dann addiert und das Ergebnis als Fließkommazahl an der Koordinate des gerade berechneten Bildpunktes abgelegt. Frühere GPUs konnten in Shadern einer Grafik-API lediglich eine geringe Anzahl an Instruktionen ausführen, echte Schleifen existierten in den Anfängen ebenfalls nicht. Um das Instruktionslimit nicht zu überschreiten, musste die Strahlverfolgung daher in mehrere einzelne Renderschritte aufgeteilt werden. Dies verursachte wiederum mehr API-Aufrufe und in Folge dessen eine erhöhte kostspielige Kommunikation zwischen CPU und GPU. Dieses Problem existiert heute glücklicherweise nicht mehr, weshalb sich, wie anhand des OpenCL C-Codefragmentes 4.2 ersichtlich wird, die Strahlverfolgung durch das Volumen recht effektiv umsetzen lässt:

```
1 for (int i=numOfSteps; i>0; i--) {
2   sum += read_imagef(volume, volumeSampler, position).x;
3   position += volumeStep;
4 }
```

Quellcode 4.2: Strahlverfolgung durch das Volumen

Inneres Drittel Nachdem das Volume-Raytracing durchgeführt wurde, besteht die nächste Aufgabe darin, alle errechneten Werte im inneren Drittel des gerenderten Bildes aufzusummieren. Für diese Aufgabe wird die parallele Reduktion eingesetzt, die am Anfang dieses Kapitels beschrieben wurde. Dies ist ein Beispiel dafür, dass sich, wenn erst einmal grundlegende, wiederkehrende Aufgabenstellungen erfolgreich parallelisiert und auf der GPU implementiert wurden, darauf aufbauende Methoden oftmals schnell realisieren lassen.

Umwandlung Anhand des in [Kub08] beschriebenen Verfahrens werden die gerenderten Fließkommazahlen mit Hilfe der ermittelten Summe der Werte im inneren Drittel in ein Bild umgewandelt, das sich mit einem Röntgenbild vergleichen lässt. Bei dieser Problemstellung handelt es sich um ein einfaches *Gather*-Problem, weshalb alle Bildpunkte komplett voneinander getrennt berechnet werden können. Aus diesem Grunde muss für die Parallelisierung kein weiterer Aufwand betrieben werden, eine Realisierung mit OpenCL ist schnell umgesetzt.

4.3 Gütemaße

Für die rein intensitätsbasierten Gütemaße muss pro Bildpunkt eine einfache Berechnung gemäß gewählter mathematischer Formeln ausgeführt werden. Anschließend werden die Werte der Bildpunkte mit Hilfe der am Anfang des Kapitels beschrieben parallelen Reduktion aufsummiert. Diese Klasse der Gütemaße stellt keine große Herausforderung bei der Implementierung dar. Gleiches gilt für räumlich intensitätsbasierte Gütemaße, die lediglich etwas rechenintensivere Berechnungen nutzen. Die in Kapitel 2 vorgestellten mathematischen Formeln für intensitätsbasierte Gütemaße lassen sich daher, wie sich anhand des OpenCL C-Quellcodes 4.3 erkennen lässt, recht einfach auf die GPU übertragen. Dieser Quellcode ist das Gegenstück zum bereits präsentierten OpenCL C-Quellcode 4.1 der parallelen Reduktion. Die Gütemaße SSD, SPD, SAD und

```
1  uint4 calculate(uint gid, __global uchar *dataFLL, __global
       uchar *dataDRR, int thresholdSDT)
2  {
3    // Ergebnisvektor
4    uint4 result;
5    // SSD
6    int value = dataFLL[gid] - dataDRR[gid];
7    result.x = value * value;
8    // SPD
9    result.y = max(0, value);
10   // SAD
11   value = abs(value);
12   result.z = value;
13   // SDT
14   result.w = max(thresholdSDT, value);
15   // Ergebnisvektor zurückgeben
16   return result;
17 }
```

Quellcode 4.3: OpenCL-Berechnungsfunktion für die Gütemaße SSD, SPD, SAD und SDT

SDT lassen sich gemeinsam berechnen und in einem 4-D-Vektor speichern. Die anschließende parallele Reduktion arbeitet folglich mit 4-D-Vektoren. Dies bringt zwar auf einer Hardware mit Skalar-Architektur wie z. B. NVIDIA-GPUs keine Leistungsvorteile, auf Vektor-Architekturen wie z. B. ATI-GPUs hingegen verbessert es die Leistung. Bei einer Vektor-Architektur lassen sich in einem Takt mehrere Daten gleichzeitig verarbeiten. Für GPGPU erweist sich allerdings oftmals eine Skalar-Architektur als von Vorteil, da sich nicht alle Problemstellungen effektiv mit

Vektorendaten darstellen lassen. Die Effektivität von 1-, 2- oder 4-Element Vektoren ist demnach von der Hardwarearchitektur abhängig. Über das Codefragment 4.4 lässt sich in OpenCL ermitteln, welche Vektorbreiten die aktuelle Hardware bevorzugt. Die Grafikkarte *ATI Mobility*

```
1 cl_uint nVectorWidth = 0;
2 clGetDeviceInfo(pDeviceID,
    CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT, sizeof(cl_uint), &
    nVectorWidth, NULL);
```

Quellcode 4.4: Vektorbreite der Hardware über OpenCL ermitteln

Radeon HD 4850 arbeitet beispielsweise effektiv mit Vektorbreiten von 4, die *NVIDIA GeForce 285 GTX* hingegen kommt mit einfachen Fließkommazahlen gut zurecht. Wenn möglich ist es daher sinnvoll zu versuchen, Vektoren für Berechnungen zu nutzen, um auf einer möglichst breiten Hardwarepalette eine gute Leistung zu erzielen.

Histogramm-basierte Gütemaße nutzen zwar ebenfalls parallele Reduktion, diese findet jedoch auf Histogrammen statt, die es vorher zu berechnen gilt. Da die Berechnung von Histogrammen eine besondere Aufgabenstellung ist, soll im Folgenden darauf eingegangen werden.

Verbundhistogramm Die in der erstellten Implementation verwendeten Histogramm-basierten Gütemaße JE und NMI benötigen ein Verbundhistogramm. Die Erstellung von Verbundhistogrammen, oder verallgemeinert Histogrammen, gehören jedoch zur *Scatter*-Problemklasse, weshalb eine effiziente Parallelisierung nicht trivial ist.

Eine sequentielle CPU-Implementation zur Erzeugung eines Verbundhistrogramms ist, wie in Codefragment 4.5 zu sehen ist, eine einfache Aufgabe. Es werden alle Bildpunkte der zwei Bilder paarweise durchlaufen. Die Werte der Bildpunkte bilden eine 2-D-Koordinate, mit der jeweils ein Eintrag im Verbundhistogramm identifiziert und um eins erhöht wird. Auf der GPU

```
1 unsigned char *pDataFLL = ...
2 unsigned char *pDataEndFLL = pDataFLL + nNumOfPixels;
3 unsigned char *pDataDRR = ...
4 for (; pDataFLL<pDataEndFLL; pDataFLL++, pDataDRR++) {
5   pnJointHistogram[*pDataFLL + (*pDataDRR)*NumOfBins]++;
6 }
```

Quellcode 4.5: Verbundhistogramm auf der CPU

hingegen würde der in Zeile 5 abgebildete Code parallel ablaufen. Es besteht folglich die Gefahr, dass mehrere Work-Items zeitgleich in ein und die selbe Speicheradresse schreiben, was für

die fehlerfreie Ausführung des Algorithmus unbedingt vermieden werden muss. Die einfachste denkbare Lösung ist im OpenCL C-Quellcode 4.6 abgebildet. Diese Lösung verwendet ato-

```
1 __kernel void joint_histogram(__global uint *result, uint
    numOfBins, __global uchar *dataFLL, __global uchar *dataDRR)
2 {
3   uint gid = get_global_id(0);
4   atom_inc(&result[dataFLL[gid] + dataDRR[gid]*numOfBins]);
5 }
```

Quellcode 4.6: Verbundhistogramm auf der GPU über atomare Schreiboperationen

mare Schreiboperationen, die über die OpenCL-Erweiterung *cl_khr_global_int32_base_atomics* genutzt werden können, sofern die GPU diese Erweiterung unterstützt. Der einzige Inhalt des abgebildeten Kernels ist im Grunde die Funktion *atom_inc*, die den Inhalt einer Speicherzelle um eins erhöht. Sobald mehrere Work-Items in die gleiche Speicheradresse schreiben wollen, kann jeweils nur ein Work-Item mit der Arbeit fortfahren, alle anderen müssen warten. Da diese ungünstige Situation jedoch häufig vorkommt und außer der atomaren Funktion im Kernel sonst keine Arbeit anfällt, wird die GPU von dieser Lösung nicht sonderlich gut ausgelastet. Diese ungünstige Situation könnte etwas entschärft werden, indem verschiedene Kernel zu einem zusammengefasst werden. Dadurch würde die Wahrscheinlichkeit steigen, dass Work-Items Berechnungen ausführen, während nur wenige Work-Items auf den atomaren Schreibzugriff warten. Erschwerend kommt jedoch hinzu, dass aktuelle GPUs atomare Schreiboperationen nicht sonderlich schnell ausführen.

Es ist jedoch bereits abzusehen, dass sich dieser Umstand bei kommenden GPU-Generationen ändern wird. Atomare Schreiboperationen sind für verschiedenste Problemstellungen hilfreich und daher für GPGPU eine wichtige Funktionalität. Aufgrund dessen wird von Seiten der Entwickler ein entsprechender Druck auf die Grafikkarten-Hersteller ausgeübt, diese Funktionalität effektiver zu realisieren [Hal09]. Wenn diese Funktionalität nicht gegeben ist oder auf der jeweiligen Hardware nur zu langsam ausgeführt werden kann, so werden die Implementierungen entsprechender Aufgaben auf der GPU schlagartig deutlich aufwändiger. Verfahren zur Erzeugung von Verbundhistogrammen, bei denen keine Synchronisation nötig ist, werden in [Sha09] vorgestellt. Aufgrund der Komplexität der dort vorgestellten Verfahren soll an dieser Stelle jedoch nicht weiter darauf eingegangen werden.

Kapitel 5

Experimente und Ergebnisse

In diesem Kapitel wird die erzielte Leistung der erstellten Implementation detailliert aufgeschlüsselt. Anhand der gesammelten Leistungsdaten wird diskutiert, welche Beschleunigungen erzielt wurden. Des Weiteren wird erörtert, weshalb sich unter bestimmten Voraussetzungen auf der GPU keine guten Leistungen erzielen lassen, oder Berechnungen auf der GPU gar länger benötigen als auf der CPU.

In Abschnitt 5.1 werden die Versuchsumgebung und das verwendete medizinische Datenmaterial beschrieben. Die genutzte Hard- und Software wird detailliert aufgelistet, während vom Datenmaterial mehrere Abbildungen gezeigt werden.

Eine ausgiebige Präsentation und Diskussion der erzielten Leistung findet in Abschnitt 5.2 statt. Die Ergebnisse werden schrittweise verfeinert, dadurch werden sowohl die Berechnungszeit des Gesamtsystems als auch die der einzelnen Bestandteile ersichtlich.

Im letzten Abschnitt 5.3 werden wesentliche Einflüsse auf die GPU-Leistung erörtert.

5.1 Versuchsumgebung und Datenmaterial

Hard- und Software Während der Implementation wurde regelmäßig auf NVIDIA- und ATI-Hardware getestet. Zwar ist OpenCL detailliert spezifiziert, im Laufe der Erstellung der Implementation wurden jedoch geringfügige Abweichungen bei den OpenCL Compiler-Implementationen der verschiedenen Hersteller festgestellt. Dies führt dazu, dass es vorkommen kann, dass OpenCL C-Quellcode in einer OpenCL-Implementation übersetzbar ist, in einer anderen allerdings einen Compilerfehler erzeugt. Aus diesem Grunde war es wichtig, die erstellte Implementation nicht nur auf einem einzigen System eines Herstellers zu testen. OpenCL ist noch recht neu, die Grafikkartentreiber von ATI und NVIDIA sind daher zum Zeitpunkt dieser

Ausarbeitung erst seit einigen Monaten öffentlich verfügbar. Aufgrund dessen ist es möglich, dass die beschriebenen Abweichungen der OpenCL-Spezifikation im Laufe der Zeit verschwinden. Sollte dies nicht geschehen, so würde einer der größten Vorteile von OpenCL - die Plattformunabhängigkeit - dadurch eingeschränkt werden.

In Tabelle 5.1 ist die für die Leistungstests verwendete Hard- und Software aufgelistet. Im Anhang A befinden sich Tabellen mit einer ausführlichen Auflistung der OpenCL-Eigenschaften beider Testsysteme.

	System 1	System 2
Typ	Desktop-PC	Notebook[1]
GPU	NVIDIA GeForce 285 GTX (2048 MiB)	ATI Mobility Radeon HD 4850 (512 MiB)
CPU	Intel Core 2 Quad Q9550 2,83 GHz	Intel Core 2 Quad Q9000 2,00 GHz
RAM	8 GiB	4 GiB
OS	Windows Vista Ultimate (64 bit)	Windows 7 Professional (64 bit)
GPU-Treiber	196.21 WHQL	Catalyst 10.3[2]

[1] Typbezeichnung *MSI GT725Q-9047VHP*
[2] *Catalyst 10.3 Preview*, ab dieser Version offiziell auf den meisten Notebooks installierbar

Tabelle 5.1: Die für die Leistungstests verwendete Hard- und Software

Bevor OpenCL-Leistungsergebnisse der ATI-Hardware präsentiert werden, sind einleitende Worte angebracht. Die in dieser Ausarbeitung präsentierte OpenCL-Implementation wurde auf ATI-Hardware entwickelt, die NVIDIA-Hardware wurde nur für zusätzliche Tests genutzt. Folglich wurde die OpenCL basierende Implementation nicht gezielt auf NVIDIA-Hardware optimiert. Zwar besitzt die verwendete ATI-Hardware 512 MiB Grafikkartenspeicher, die OpenCL-Implementation kann davon jedoch lediglich 128 MiB nutzen. Aufgrund des daraus resultierenden Speichermangels war es nicht möglich, das Volumen der Größe $512 \times 512 \times 378$ zu verwenden. Daher konnten auf dieser Grafikkarte nicht alle Tests durchgeführt werden. Weitere Mängel, auf die an den entsprechenden Stellen eingegangen wird, machen es nicht ohne weiteres möglich, die gemessene Laufzeit des ATI-Testsystems mit der des NVIDIA-Testsystems zu vergleichen.

Verwendeter Datensatz Der für die Tests verwendete medizinische Datensatz ist in Bild 5.1 abgebildet. Im Röntgenbild wird jeder Bildpunkt als 1 B gespeichert. Bei einer Größe von 2480×1920 bedeutet dies, dass ca. 4,5 MiB Speicher belegt werden. Das verwendete Volumen besteht aus $512 \times 512 \times 378$ Voxel. Jeder Voxel benötigt 2 B, dies führt zu einem Speicherverbrauch von 189 MiB. Damit während des Volume-Raytracings die Voxel-Werte nur noch ausgelesen werden müssen, wurde das Volumen vor der Registrierung vor verarbeitet. Jeder vor verarbeitete Voxel

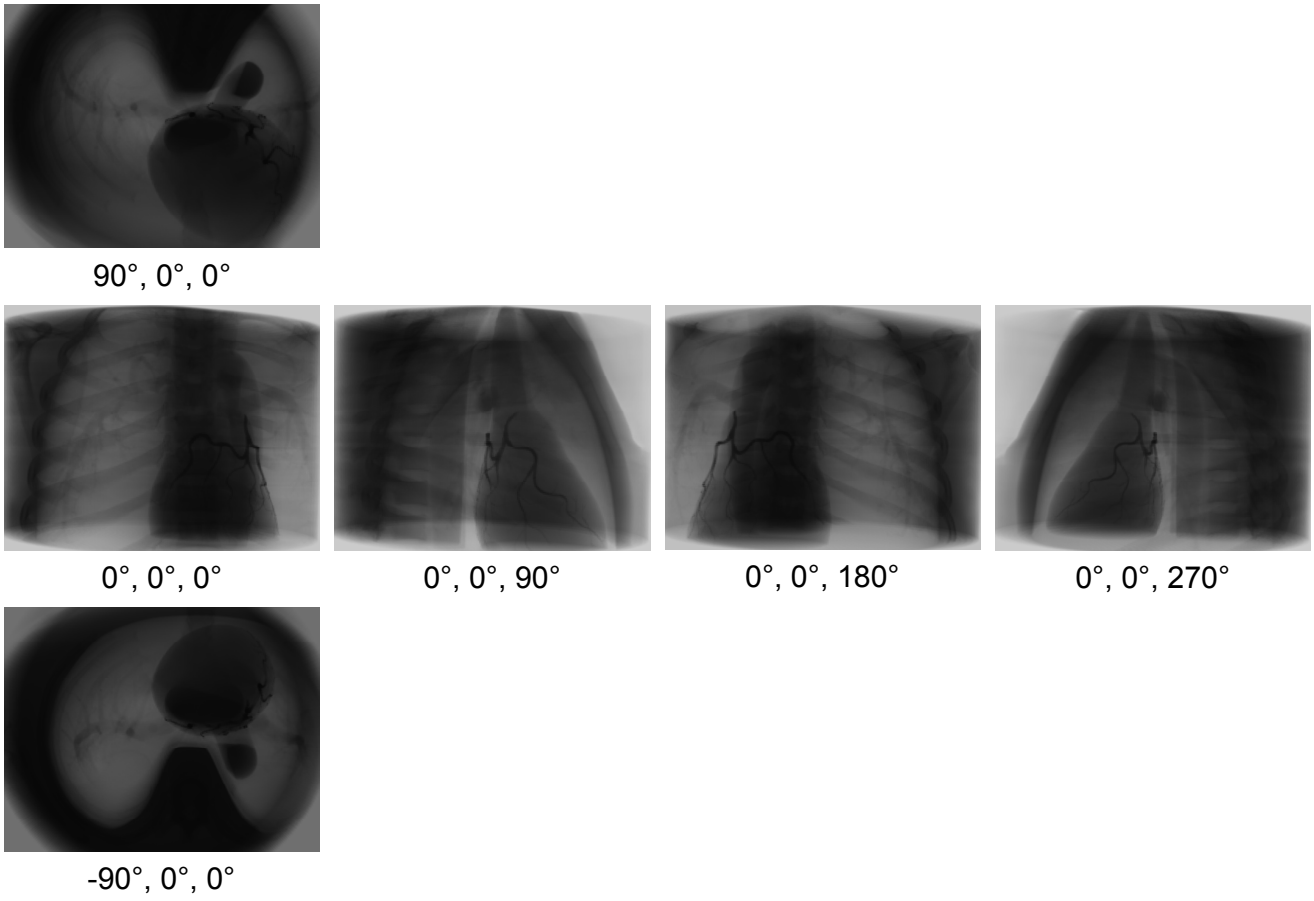

Bild 5.1: Rundumsicht des verwendeten medizinischen Datensatzes

benötigt zur Laufzeit 4 B, das während den Berechnungen verwendete Volumen belegt daher 378 MiB an Speicher.

Rahmenbedingungen der Ergebnisse Um ermitteln zu können wie die verschiedenen Implementationen jeweils mit unterschiedlich großen Datensätzen skalieren, wurde der vorgestellte Datensatz schrittweise verkleinert. Die verwendete ROI beginnt bei $(240, 240)$ (einschließlich), und endet bei $(2232, 1672)$ (einschließlich). Diese ROI wurde jeweils gemeinsam mit dem Röntgenbild verkleinert. Sofern nicht anders angegeben, beziehen sich alle präsentierten Werte auf die benötigte mittlere Zeit pro Registrierungs-Schritt einer 64 bit-Implementierung. Sofern auf den Testsystemen vorhanden, wurden OpenCL-Erweiterungen verwendet um eine bessere Leistung zu erzielen. Alle OpenCL-Ergebnisse beziehen sich jeweils auf eine GPU-Implementation von OpenCL. Die CPU-Implementation ist rein sequentiell und nutzt daher nur einen CPU-Kern.

5.2 Erzielte Leistung

32 bit/64 bit Zu Beginn wurde untersucht, ob sich Geschwindigkeitsunterschiede einstellen, wenn die Implementation einmal für 32 bit und ein anderes mal für 64 bit übersetzt wird. In Tabelle 5.2 wurde die mittlere benötigte Zeit pro Registrierungsschritt festgehalten. Zum Erheben der Daten wurde eine Röntgenbild-Größe von 620×480 und eine Volumen-Größe von $256 \times 256 \times 189$ verwendet. Wie zu sehen ist, haben die 64 bit-Versionen auf der CPU sichtbar

Implementation	Bit	System 1	System 2
CPU	32	281,54 ms	360,75 ms
CPU	64	260,78 ms	337,83 ms
OpenCL	32	6,33 ms	25,18 ms
OpenCL	64	6,35 ms	26,01 ms

Tabelle 5.2: Geschwindigkeitsunterschiede zwischen 32 bit und 64 bit

bessere Laufzeiten als die identische Implementation mit 32 bit. Bei OpenCL stellte sich das umgekehrte Ergebnis ein. Auf beiden Testsystemen lief die 64 bit-Version minimal langsamer als die 32 bit-Version. Aus der *ADDRESS_BITS*-Spalte der Tabellen im Anhang A ist zu entnehmen, dass die GPUs beider Testsysteme intern mit 32 bit arbeiten. Dass die Laufzeiten unter 64 bit nicht identisch sind sondern etwas schlechter, könnte eventuell darauf schließen lassen, dass die 64 bit-Treiber noch nicht so gut optimiert sind wie die 32 bit-Treiber.

Laufzeiten des Gesamtsystems Wie die erstellten Implementationen mit der Datengröße skalieren, lässt sich aus der Tabelle 5.3 ablesen. Da der auf Testsystem 2 zur Verfügung stehende

Röntgenbild-Größe	Volumen-Größe	1: CPU	1: OpenCL	2: CPU	2: OpenCL
155×120	$128 \times 128 \times 94$	9,98 ms	1,67 ms	12,84 ms	5,68 ms
310×240	$256 \times 256 \times 189$	80,06 ms	3,09 ms	98,69 ms	11,80 ms
620×480	$256 \times 256 \times 189$	260,78 ms	6,35 ms	337,83 ms	26,01 ms
1240×960	$512 \times 512 \times 378$	2658,86 ms	34,73 ms	3252,80 ms	-
2480×1920	$512 \times 512 \times 378$	9554,68 ms	130,30 ms	11 641,96 ms	-

Tabelle 5.3: Laufzeiten des Gesamtsystems auf beiden Testsystemen für jeweils einen Registrierungsschritt

globale Speicher nicht ausreicht, um das Volumen in einer Auflösung von $512 \times 512 \times 378$ zu speichern, war es nicht möglich, hier Ergebnisse zu ermitteln. Generell ist zu erkennen, dass die auf Testsystem 2 durch OpenCL erzielte Beschleunigung verglichen mit Testsystem 1 eher

bescheiden ausfällt. Auf Testsystem 1 hingegen konnte auf der GPU eine beachtliche Beschleunigung erzielt werden. Die gesamte 2-D/3-D-Registrierung des voll auflösenden medizinischen Datensatzes benötigte auf der CPU dieses Testsystems 2914,18 s, umgerechnet 48,57 min. Die GPU-Lösung benötigte auf dem gleichen Testsystem hingegen nur 41,31 s.

In Bild 5.2 wird die erreichte Beschleunigung auf beiden Testsystemen als Balkendiagramm veranschaulicht. Es ist zu erkennen, dass der über die GPU erreichbare Beschleunigungsfaktor

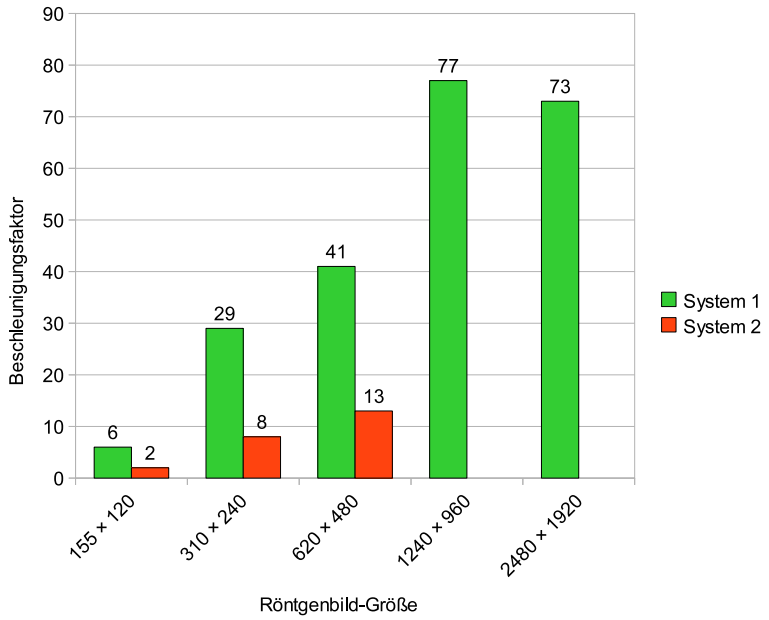

Bild 5.2: Durch die GPU erzielte Beschleunigung auf beiden Testsystemen, Volume-Größen wie in Tabelle 5.3

prinzipiell besser wird, je umfangreicher die zu verarbeitende Datenmenge ist. Dies liegt darin begründet, dass bei der gleichen Anzahl von API-Aufrufen auf der GPU mehr Daten verarbeitet werden können. Das Leistungspotential der GPU lässt sich damit besser ausschöpfen. Der zur CPU verglichene erreichbare Beschleunigungsfaktor ist jedoch nach oben hin nicht offen. Wie zu erkennen ist, befindet sich die beste erreichbare Beschleunigung bei Testsystem 1 bei einer Röntgenbild-Größe von 1240×960 und einer Volumen-Größe von $512 \times 512 \times 378$. Bei einer Röntgenbild-Größe von 2480×1920 und Volumen-Größe von $512 \times 512 \times 378$ wird die erzielbare Beschleunigung bereits durch die Leistungsfähigkeit der verwendeten GPU begrenzt.

Im folgenden werden die Daten aufgeschlüsselt. Dadurch wird ersichtlich welcher Anteil der benötigten Rechenzeit auf die Erzeugung des DRR fällt, und wie viel Zeit für die Berechnung der Gütemaße benötigt wird.

Tabelle 5.4 listet die für das Erzeugen des DRR benötigte Zeit auf. Anhand dieser Daten ist

Röntgenbild-Größe	Volumen-Größe	1: CPU	1: OpenCL	2: CPU	2: OpenCL
155×120	$128 \times 128 \times 94$	9,45 ms	0,55 ms	12,18 ms	2,08 ms
310×240	$256 \times 256 \times 189$	78,74 ms	1,82 ms	97,10 ms	7,30 ms
620×480	$256 \times 256 \times 189$	256,50 ms	4,42 ms	332,51 ms	18,27 ms
1240×960	$512 \times 512 \times 378$	2642,14 ms	30,45 ms	3232,66 ms	-
2480×1920	$512 \times 512 \times 378$	9491,39 ms	117,52 ms	11 562,84 ms	-

Tabelle 5.4: DRR-Anteil von Tabelle 5.3

direkt zu erkennen, dass für die realitätsnahe DRR-Erzeugung die meiste Rechenzeit benötigt wird. Der Vollständigkeit halber wird die für die Gütemaße benötigte Zeit in Tabelle 5.5 aufgelistet. In dieser Tabelle befinden sich auf 2 Stellen nach dem Komma gerundete Zahlen, daher kann beim Addieren der Werte von Tabelle 5.4 und Tabelle 5.5 ein minimal anderer Wert herauskommen als in Tabelle 5.3 aufgeführt wird. Dass gerade große Datenmengen auf der GPU

Röntgenbild-Größe	Volumen-Größe	1: CPU	1: OpenCL	2: CPU	2: OpenCL
155×120	$128 \times 128 \times 94$	0,53 ms	1,11 ms	0,66 ms	3,61 ms
310×240	$256 \times 256 \times 189$	1,32 ms	1,27 ms	1,59 ms	4,50 ms
620×480	$256 \times 256 \times 189$	4,28 ms	1,93 ms	5,32 ms	7,74 ms
1240×960	$512 \times 512 \times 378$	16,72 ms	4,28 ms	20,13 ms	-
2480×1920	$512 \times 512 \times 378$	63,29 ms	12,78 ms	79,12 ms	-

Tabelle 5.5: Gütemaße-Anteil von Tabelle 5.3

schneller als auf der CPU verarbeitet werden können, wird in Tabelle 5.5 anhand der Spalte für die Röntgenbild-Größe 155×120 ersichtlich. Hier ist auf Testsystem 1 die CPU fast doppelt so schnell wie die GPU. Dies liegt darin begründet, dass die Kommunikation mit der GPU mit einem gewissen Verwaltungsaufwand einhergeht. Des Weiteren ist die GPU bei kleinen Datenmengen nur schlecht ausgelastet, viel Rechenleistung bleibt ungenutzt.

Anhand von Tabelle 5.4 und Tabelle 5.5 wird ersichtlich, dass sich die realitätsnahe DRR-Erzeugung auf der GPU insgesamt betrachtet wesentlich besser beschleunigen lässt als die Berechnung der Gütemaße. So ließ sich die DRR-Erzeugung auf Testsystem 1 für eine Röntgenbild-Größe von 1240×960 mit einer Volumen-Größe von $512 \times 512 \times 378$ um den Faktor 87 Beschleunigen, die Berechnung der Gütemaße hingegen lediglich um einen Faktor von 4.

Diese zwei Tabellen lassen ebenfalls erkennen, weshalb die Beschleunigung auf Testsystem 2 eher enttäuschend ausfällt. Während sich bei der DRR-Erzeugung eine Beschleunigung einstellt, verliert dieses Testsystem bei der Berechnung der Gütemaße deutlich gegen die CPU.

Diese Beobachtung lässt sich darin begründen, dass für die Berechnung der Gütemaße die über lokalen Speicher beschleunigte parallele Reduktion intensiv genutzt wird. Auf Testsystem 2 wird der lokale Speicher allerdings über den globalen Speicher emuliert. Daraus resultiert, dass sich durch den Einsatz von lokalem Speicher statt einer Beschleunigung sogar eine Verschlechterung einstellt. Dies macht die verwendete ATI-Hardware für bestimmte OpenCL-Implementationen in Hinsicht auf die Leistung praktisch nutzlos. Ob ein OpenCL-Gerät lokalen Speicher emuliert, lässt sich über Quellcode 5.1 ermitteln.

```
1 cl_device_local_mem_type nMemoryType = 0;
2 clGetDeviceInfo(pDeviceID, CL_DEVICE_LOCAL_MEM_TYPE, sizeof(
    cl_device_local_mem_type), &nMemoryType, NULL);
3 bool bLocalMemoryAvailable = (nMemoryType == CL_LOCAL);
```

Quellcode 5.1: Abfrage des Typs des lokalen Speichers eines OpenCL-Gerätes

Im Folgenden werden die ermittelten Ergebnisse weiter aufgeschlüsselt. Da Testsystem 2 aufgrund der festgestellten Einschränkungen weniger für GPGPU geeignet ist und eine Detaillierung der Ergebnisse für beide Testsysteme der Übersichtlichkeit nicht dienlich wäre, findet die Aufschlüsselung der Ergebnisse lediglich für Testsystem 1 statt.

DRR Geschwindigkeit Die DRR-Erzeugung wird über 3 Kernel realisiert. Der erste Kernel führt das Volume-Raytracing durch, die gemessenen Zeiten sind in Tabelle 5.6 zusammengefasst. Anhand dieser Tabelle wird ersichtlich, dass für den Volume-Raytracing Kernel die meiste Zeit

Röntgenbild-Größe	Volumen-Größe	CPU	OpenCL
155×120	$128 \times 128 \times 94$	9,35 ms	0,31 ms
310×240	$256 \times 256 \times 189$	78,33 ms	1,57 ms
620×480	$256 \times 256 \times 189$	254,93 ms	4,11 ms
1240×960	$512 \times 512 \times 378$	2635,94 ms	29,89 ms
2480×1920	$512 \times 512 \times 378$	9466,61 ms	116,23 ms

Tabelle 5.6: Geschwindigkeit des DRR-Kernels für Volume-Raytracing auf Testsystem 1

benötigt wird. Glücklicherweise ließ sich das Volume-Raytracing sehr gut mit Hilfe einer GPU beschleunigen. So stellte sich beim Volume-Raytracing auf Testsystem 1 für eine Röntgenbild-Größe von 1240×960 und einer Volumen-Größe von $512 \times 512 \times 378$ eine Beschleunigung um Faktor 88 ein. Bild 5.3 stellt diesen Sachverhalt nochmals über ein Kurvendiagramm dar und verdeutlicht, dass Volume-Raytracing von der massiven parallelen Rechenleistung des Grafikprozessors profitiert.

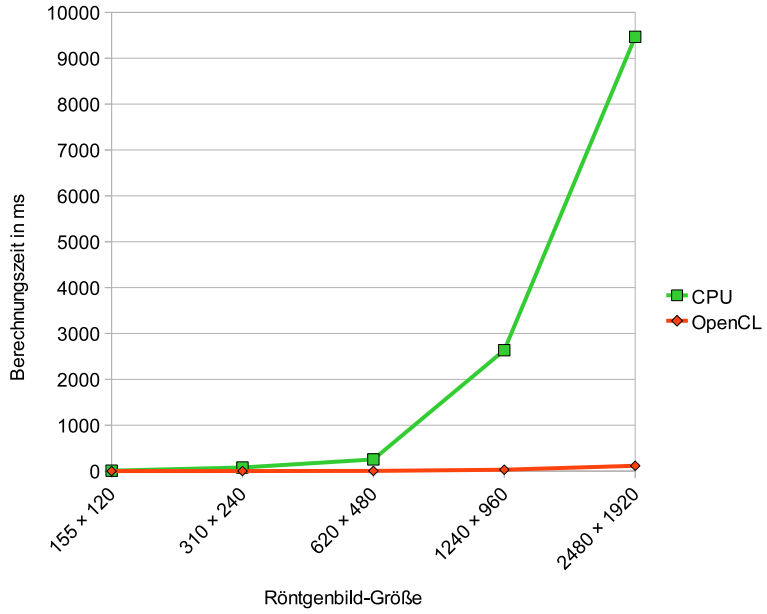

Bild 5.3: Geschwindigkeit des DRR Volume-Raytracing auf Testsystem 1

Für die Berechnungsdauer eines einzelnen 2-D/3-D-Registrierungs-Schrittes ist die Volumen-Größe nur für den Volume-Raytracing Kernel von Interesse. Daher wird die Volumen-Größe für eine bessere Übersichtlichkeit in den folgenden Tabellen nicht mehr explizit aufgeführt. Wenn nicht anders angegeben, bleiben diese Größen identisch zu denen in den bisherigen Tabellen.

Die nach dem Volume-Raytracing Kernel folgenden zwei Kernel wandeln die berechneten Fließkommazahlen in ein 8 bit Graustufenbild um, welches bei den Gütemaßen anschließend mit einem gegebenen 8 bit Röntgenbild verglichen wird. Im ersten dieser zwei Kernel wird über parallele Reduktion die Summe der Werte im inneren Drittel gebildet. Im zweiten Kernel werden die Fließkommazahlen in ein 8 bit Graustufenbild überführt. Tabelle 5.7 listet die für diese zwei Schritte benötigte Zeit auf. Wie zu erkennen ist, überholt die GPU bei der Berechnung der

Röntgenbild-Größe	1: CPU	1: OpenCL	2: CPU	2: OpenCL
155 × 120	0,005 ms	0,141 ms	0,096 ms	0,099 ms
310 × 240	0,025 ms	0,151 ms	0,382 ms	0,102 ms
620 × 480	0,074 ms	0,185 ms	1,502 ms	0,133 ms
1240 × 960	0,217 ms	0,308 ms	5,982 ms	0,249 ms
2480 × 1920	0,757 ms	0,606 ms	24,024 ms	0,678 ms

Tabelle 5.7: Geschwindigkeit der DRR-Kernel auf Testsystem 1. Kernel 1 berechnet die Summe des inneren Drittels, Kernel 2 überführt die Daten in ein 8 bit Graustufenbild.

Summe der Werte im inneren Drittel die CPU erst bei einer Größe von 2480 × 1920 Bildpunkten. Dies bestätigt die bereits gemachte Beobachtung, dass die GPU mit großen Datenmengen besonders gut zurechtkommt. Des Weiteren verdeutlicht dies ebenfalls, dass nicht jede Problemstellung ohne weiteres auf der GPU eine enorme Beschleunigung erfährt, gerade wenn weniger Berechnungen als Speicherzugriffe erfolgen. Bild 5.4 stellt diesen Sachverhalt nochmals über ein Kurvendiagramm dar. Die Umwandlung der Daten in ein 8 bit Graustufenbild ließ sich hingegen

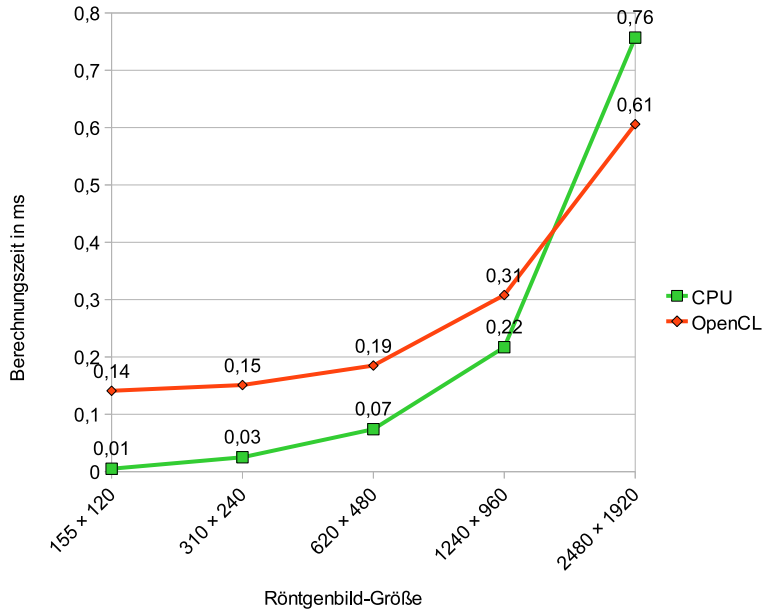

Bild 5.4: Die Berechnungsgeschwindigkeit der Summe des inneren Drittels auf Testsystem 1

gut beschleunigen.

Rein intensitätsbasierte Gütemaße Die Berechnung aller Gütemaße erfolgt ausschließlich innerhalb einer gegebenen ROI. Die Gütemaße SSD, SPD, SAD und SDT werden in einem gemeinsamen Kernel berechnet und nutzen genauso wie die Summenberechnung im inneren Drittel parallele Reduktion. Aufgrund der gemeinsamen Berechnung dieser Gütemaße lässt sich die Berechnungszeit nicht für die einzelnen Gütemaße getrennt ermitteln. Tabelle 5.8 listet die ermittelten Berechnungszeiten auf. Der Unterschied dieses Kernels zum Kernel für die Berechnung der Summe der Werte im inneren Drittel liegt im Grunde nur darin, dass im SSD, SPD, SAD und SDT Kernel mehr Berechnungen stattfinden. Dieser kleine Unterschied ist allerdings dafür verantwortlich, dass über die Nutzung der GPU eine Beschleunigung erzielt werden kann. Dieser Sachverhalt wird in Bild 5.5 visualisiert. Diese Beobachtung zeigt recht deutlich, wie wichtig es

Röntgenbild-Größe	CPU	OpenCL
155 × 120	0,044 ms	0,149 ms
310 × 240	0,174 ms	0,172 ms
620 × 480	0,683 ms	0,264 ms
1240 × 960	2,724 ms	0,612 ms
2480 × 1920	10,850 ms	1,975 ms

Tabelle 5.8: Gütemaße SSD, SPD, SAD und SDT Anteil von Tabelle 5.5

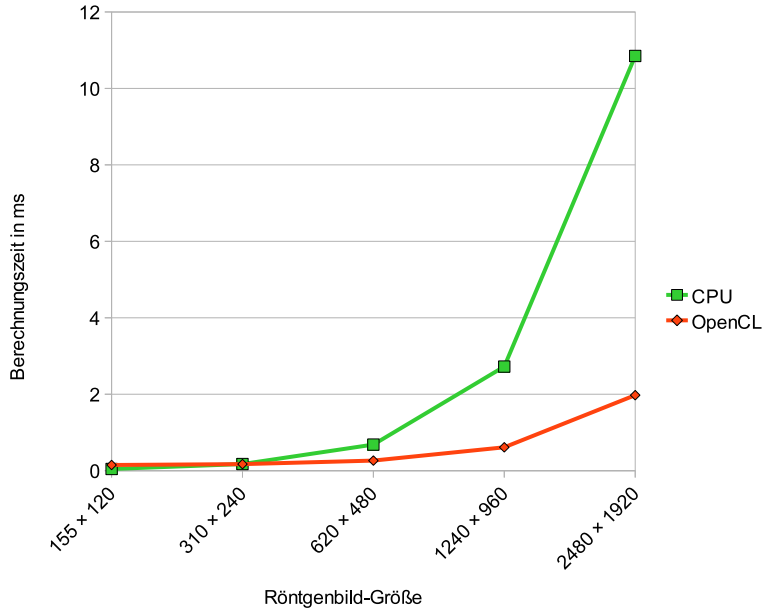

Bild 5.5: Die Berechnungsgeschwindigkeit der SSD, SPD, SAD und SDT Gütemaße auf Testsystem 1

ist, so viele Arbeitsschritte wie möglich in einem Kernel zu kombinieren, um Speicherlatenzen durch Berechnungen zu verbergen.

Räumlich intensitätsbasierte Gütemaße Das Gütemaß GC wurde über zwei Kernel berechnet. Im ersten Kernel wird über den Sobel-Operator das horizontale und vertikale Gradientendbild des aktuellen DRR errechnet. Im gleichen Kernel findet eine parallele Reduktion statt, dort werden die gerade errechneten Gradienten aufsummiert. Im zweiten Kernel wird der horizontale und vertikale NCC der Gradientendbilder errechnet. Bei dieser Berechnung handelt es sich abermals um eine Summenbildung der im gleichen Kernel errechneten Werte. Die Summenbildung findet auch hier durch parallele Reduktion statt. In Tabelle 5.9 wurde die Berechnungszeit beider Kernel festgehalten. Bei GC sind keine großen Überraschungen zu beobachten. GPUs sind

Röntgenbild-Größe	1: CPU	1: OpenCL	2: CPU	2: OpenCL
155×120	0,145 ms	0,139 ms	0,036 ms	0,139 ms
310×240	0,583 ms	0,158 ms	0,148 ms	0,158 ms
620×480	2,280 ms	0,235 ms	0,626 ms	0,253 ms
1240×960	9,018 ms	0,498 ms	3,244 ms	0,561 ms
2480×1920	36,196 ms	1,537 ms	10,991 ms	1,773 ms

Tabelle 5.9: Berechnungsgeschwindigkeit des GC-Gütemaßes auf Testsystem 1. Kernel 1 berechnet Gradientendbilder, Kernel 2 berechnet NCC.

darauf optimiert, grafische Ausgaben zu erzeugen und können daher beim Sobel-Operator ihr Leistungspotential entfalten. Dank Texturen und Texture-Caches ist hierbei eine gute Beschleunigung zu erzielen. Die Berechnungszeit für NCC bewegt sich ebenfalls in erwarteten Regionen und skaliert aufgrund der ähnlichen Berechnungsabläufe vergleichbar zu den rein intensitätsbasierten Gütemaßen aus Tabelle 5.8.

Histogramm-basierte Gütemaße Die Gütemaße JE und NMI werden über mehrere Kernel berechnet. Das Hauptproblem bei diesen Gütemaßen liegt in der Erstellung des Verbundhistogrammes. Beim Verbundhistogramm handelt es sich um ein *Scatter*-Problem, eine Umsetzung auf der GPU ist daher nicht trivial, da diese Art der Problemstellung nicht mit der parallelen Architektur von Grafikprozessoren harmoniert. Die einfachste Möglichkeit der Berechnung besteht darin, das 8 bit DRR Graustufenbild über die OpenCL-API von der Grafikkarte herunterzuladen, um anschließend das Verbundhistogramm auf der CPU zu berechnen. Nach diesem Schritt steht der Entwickler vor der Wahl, den Rest des gewünschten Gütemaßes ebenfalls auf der CPU zu berechnen, oder das aufgestellte Verbundhistogramm zur weiteren Verarbeitung auf die Grafikkarte hochzuladen. Letzteres wurde in der erstellten Implementation realisiert um ermitteln zu können, wie gut die GPU mit den restlichen Schritten zurechtkommt. In Tabelle 5.10 sind drei Implementationen zur Berechnung des Verbundhistogrammes aufgelistet. Die Laufzeiten aus der ersten Spalte stammen aus einer vollständigen CPU-Implementation der gesamten 2-D/3-D-Registrierung. In der mittleren Spalte stehen Laufzeiten aus der OpenCL-Implementation, das Verbundhistogramm wurde hier allerdings auf der CPU berechnet. In der letzten Spalte der Tabelle stehen Laufzeiten aus einer OpenCL-Implementation mit einem auf der GPU erzeugten Verbundhistogramm, dazu wurden atomare Schreiboperationen verwendet. Der Tabelle ist zu entnehmen, dass die Berechnung des Verbundhistogrammes für die CPU keine große Hürde darstellt. Die CPU-Lösung läuft rein sequentiell ab, daher ist keine aufwändige Synchronisation nötig. Wird die zweite Spalte der Tabelle von der dritten Spalte der gleichen Tabelle abgezogen,

Röntgenbild-Größe	CPU	CPU + OpenCL	OpenCL
155 × 120	0,087 ms	0,291 ms	0,214 ms
310 × 240	0,150 ms	0,391 ms	0,538 ms
620 × 480	0,386 ms	0,779 ms	2,206 ms
1240 × 960	1,396 ms	2,201 ms	10,642 ms
2480 × 1920	4,877 ms	7,063 ms	56,446 ms

Tabelle 5.10: Die Berechnungsgeschwindigkeit für das Verbundhistogramm auf Testsystem 1, vollständige CPU-Lösung, nur Verbundhistogramm auf CPU und Verbundhistogramm mit OpenCL über atomare Schreiboperationen

so lässt sich ermitteln, wie viel Zeit für den Datentransfer zwischen Host und GPU benötigt wird. Anhand dieser Daten wird ersichtlich, dass dies der Flaschenhals der Gütemaße JE und NMI ist. In der letzten Zeile dieser Tabelle wurde das Verbundhistogramm mit Hilfe atomarer Schreiboperationen realisiert, daher besteht keine Notwendigkeit eines zeitintensiven Datentransfers. Da der Kernel für die Erstellung dieses Verbundhistogrammes sehr simpel ist, wirkt sich die für die atomaren Schreiboperationen benötigte Zeit spürbar auf die für die Berechnung benötigte Zeit aus. Atomare Schreiboperationen sind auf aktuellen GPUs noch nicht so schnell wie dies wünschenswert wäre, diese Situation könnte sich allerdings laut [NVI09c] und [NVI10] bereits mit der nächsten GPU-Generation von NVIDIA ändern. Die realisierte reine OpenCL-Lösung ist auf aktuellen GPUs letztendlich deutlich langsamer als die Lösung, in der das Verbundhistogramm auf der CPU erstellt wird. Zwar existieren weitere Ansätze, wie ein Verbundhistogramm auf der GPU erzeugt werden kann, aufgrund der Komplexität der meisten Ansätze konnten diese jedoch im Rahmen dieser Arbeit nicht implementiert und getestet werden.

Alle weiteren Berechnungen der JE und NMI Gütemaße finden nur noch auf dem erzeugten Verbundhistogramm statt. Daraus folgt, dass die Berechnungszeit der weiteren Schritte theoretisch unabhängig von der Röntgenbild-Größe ist und nur noch eine Abhängigkeit von der Anzahl der im Verbundhistogramm genutzten Bins besteht. Die Anzahl der Bins ist in der erstellten Implementation konstant. Da 8 bit Graustufenbilder verwendet werden, beträgt die Anzahl der Bins in der erstellten Implementation 256. Des Weiteren werden Berechnungen für Einträge mit dem Wert 0 im Verbundhistogramm übersprungen. Dadurch stellte sich für kleinere Röntgenbild-Größen eine bessere Laufzeit ein, da weniger Einträge im Verbundhistogramm eingefügt wurden und folglich die Wahrscheinlichkeit, dass ein Eintrag 0 ist, steigt.

Das Gütemaß NMI lässt sich unter Zuhilfenahme des JE Gütemaßes errechnen, daher gilt es zunächst JE zu berechnen. In der Berechnung für JE wird die Verbund-Entropie ermittelt, diese Berechnung erfolgt über einen Kernel, der nahezu identisch zu den anderen Kerneln ist,

die parallele Reduktion nutzen. Tabelle 5.11 listet die ermittelten Laufzeiten auf. Während sich

Röntgenbild-Größe	CPU	OpenCL
155 × 120	0,186 ms	0,149 ms
310 × 240	0,234 ms	0,147 ms
620 × 480	0,268 ms	0,148 ms
1240 × 960	0,293 ms	0,148 ms
2480 × 1920	0,323 ms	0,157 ms

Tabelle 5.11: Der NMI-Gütemaß Anteil von Tabelle 5.5

bei der CPU-Lösung für größere Röntgenbilder längere Laufzeiten einstellen, bleibt die GPU-Lösung vergleichsweise konstant. Dies lässt sich dadurch begründen, dass die meiste Zeit im Kernel für parallele Reduktion benötigt wird und die Berechnung nur einen kleinen Teil der benötigten Zeit ausmacht.

Die DRR-Entropie ändert sich für jeden Registrierungs-Schritt, daher ist es für das NMI-Gütemaß nötig, diese Entropie für jedes aktuelle DRR zu berechnen. Die Entropie wird über das DRR-Histogramm berechnet, daher ist es nötig dieses Histogramm aufzustellen. Wie beim Verbundhistogramm zu erkennen war, kommt die GPU mit dieser Art der Problemstellung nicht besonders gut zurecht. Aus diesem Grund wurde das Histogramm aus dem bereits vorhandenen Verbundhistogramm erzeugt. Dies lässt sich durch ein einfaches Aufsummieren von Spalten erreichen. Da jede Zeile lediglich aus 256 Werten besteht, wurde für das Aufsummieren der einzelnen Zeile ein anderer Ansatz gewählt als bei den anderen Problemstellungen, die parallele Reduktion benötigen. Für jede Zeile wurde jeweils ein Thread gestartet, dadurch führen 256 Threads gleichzeitig die Aufsummierung jeder Zeile durch. Dies ist zwar keine effiziente Ausnutzung der GPU, aufgrund der geringen Datenmenge dieser Berechnung ist die GPU allerdings auch bei komplizierteren Ansätzen unterfordert. 5.12 listet die ermittelten Laufzeiten auf. Wie anhand der OpenCL-Werte zu erkennen ist, ist die GPU unterfordert und benötigt sogar

Röntgenbild-Größe	CPU	OpenCL
155 × 120	0,035 ms	0,247 ms
310 × 240	0,035 ms	0,241 ms
620 × 480	0,036 ms	0,247 ms
1240 × 960	0,043 ms	0,257 ms
2480 × 1920	0,055 ms	0,278 ms

Tabelle 5.12: Der DRR-Entropie Anteil von Tabelle 5.5

länger, als für die Berechnung der Verbund-Entropie aus Tabelle 5.11. Dies verdeutlicht abermals, dass die GPU eine Verarbeitung kleiner Datenmengen nicht gut beschleunigen kann und

resultiert darin, dass mit einer längeren Laufzeit zu rechnen ist als bei CPU-Lösungen. Die von der Röntgenbild-Größe abhängigen Ergebnisse erklären sich dadurch, dass nach dem Erstellen des DRR-Histogrammes vergleichbar zur Berechnung des Verbundhistogrammes die Einträge des Histogrammes nach einer kleinen Berechnung aufsummiert werden. Diese Berechnung wird übersprungen, falls ein Eintrag 0 ist.

OpenCL und OpenGL Leistungsvergleich Teile der Implementation wurden zum Vergleich auch mit OpenGL implementiert. Während sich *Gather*-Probleme mit OpenGL ähnlich einfach wie mit OpenCL realisieren lassen, sind *Scatter*-Probleme mit OpenGL deutlich umständlicher realisierbar als mit OpenCL. Für DRR Volume-Raytracing wird die meiste Zeit benötigt, daher wird in Tabelle 5.13 lediglich die Zeit für DRR Volume-Raytracing präsentiert. Der OpenGL Shading Language (GLSL)-Shader und das OpenCL-Programm sind, bis auf die leicht andere Syntax, identisch. Da gerade durch die Unterstützung für 3-D-Bilder das DRR Volume-

Röntgenbild-Größe	Volumen-Größe	1: OpenCL	1: OpenGL	2: OpenCL	2: OpenGL
155×120	$128 \times 128 \times 94$	0,31 ms	1,96 ms	1,42 ms	0,88 ms
310×240	$256 \times 256 \times 189$	1,57 ms	3,42 ms	6,49 ms	3,17 ms
620×480	$256 \times 256 \times 189$	4,11 ms	5,95 ms	17,13 ms	7,65 ms
1240×960	$512 \times 512 \times 378$	29,89 ms	35,99 ms	-	187,73 ms
2480×1920	$512 \times 512 \times 378$	116,23 ms	120,21 ms	-	283,59 ms

Tabelle 5.13: OpenCL und OpenGL DRR Volume-Raytracing Leistungsvergleich für beide Testsysteme

Raytracing deutlich beschleunigt werden kann, stellt die fehlende Unterstützung von Bildern generell auf der verwendeten ATI-Hardware einen gravierenden Nachteil für die Ausführungsgeschwindigkeit der OpenCL-Implementation dar. Da OpenGL unter anderem auf 3-D-Texturen zurückgreifen kann, lag die OpenGL-Leistung auf der ATI-Hardware deutlich über der von OpenCL. Diese Beobachtung gilt allerdings nur für die genutzte ATI-Hardware. Die präsentierten NVIDIA-Leistungsergebnisse deuten an, dass OpenCL und OpenGL vergleichbare Leistungsergebnisse erzielen können, in der hier erstellten Implementation OpenCL jedoch etwas bessere Leistungsergebnisse erreichte. Es ist daher das Fazit zu ziehen, dass die Zeiten, in denen GPGPU über klassische Grafik-Schnittstellen realisiert wird, vorbei sind und stattdessen besser auf GPGPU ausgelegte Schnittstellen wie OpenCL verwendet werden sollten.

Die Leistung der verwendeten ATI-Hardware unter OpenGL ist gerade für eine Mobile-GPU recht beeindruckend und auf der Höhe der Zeit, daher war die beobachtete sehr schlechte OpenCL-Leistung unerwartet. Die schlechte Leistung lässt sich darin begründen, dass lokaler

Speicher auf dieser Hardware über globalen Speicher emuliert wird, keine Unterstützung für Bilder existiert und generell keine OpenCL-Erweiterungen zur Verfügung stehen, die zur Leistungsverbesserung genutzt werden könnten. Zwar ist die im DRR Volume-Raytracing erreichte Leistung akzeptabel, mit Unterstützung für 3-D-Texturen könnte diese aber noch deutlich besser ausfallen. Für viele Problemstellungen ist die aktuell erreichbare Leistung jedoch inakzeptabel, und es stellt sich die Frage, ob ATI die Treiberunterstützung für OpenCL für die *4xxx*-Generation noch verbessern wird. Es stand keine ATI-Hardware der aktuellen Generation zur Verfügung, daher kann keine Aussage darüber getroffen werden, ob ATI-Hardware im Allgemeinen für GPGPU weniger geeignet ist. Da die vorhandene ATI-Hardware Anfang 2009 erschien, kann diese jedoch auch nicht als hoffnungslos veraltet gezählt werden, es lag daher nahe, die in dieser Arbeit erzielten OpenCL-Leistungen dieser Hardware der Vollständigkeit halber festzuhalten.

NVIDIA-Hardware der letzten Jahre unterstützt für GPGPU CUDA. OpenCL nutzt auf dieser Hardware CUDA, daher gab es bei der OpenCL-Leistung keine unerwarteten Ergebnisse. Es werden zahlreiche hilfreiche OpenCL-Erweiterungen unterstützt, die zur Leistungssteigerung genutzt werden können.

5.3 Auswirkung von globaler und lokaler Problemgröße

In OpenCL wird eine Berechnung mit *clEnqueueNDRangeKernel* gestartet, diese Funktion hat als Parameter unter anderem die globale und lokale Problemgröße. Laut OpenCL-Spezifikation [Gro09] kann bei *clEnqueueNDRangeKernel* für die lokale Größe ein *NULL*-Zeiger übergeben werden. In diesem Fall übernimmt die zugrunde liegende *OpenCL*-Implementation die Ermittlung geeigneter Einstellungen für die lokale Größe. Die globale Größe kann in dieser Situation beliebig gewählt werden und muss beispielsweise nicht durch 2 teilbar sein. Sobald die lokale Größe per Hand gesetzt wird, muss die übergebene globale Größe jedoch ein Vielfaches dieser lokalen Größe sein. In vielen Anwendungsfällen ist es wünschenswert, die globale Größe frei festlegen zu können, daher ist es nicht direkt ersichtlich, aus welchem Grund eine lokale Größe per Hand eingestellt werden sollte. Im Folgenden sollen die zwei erwähnten Parameter und deren Relevanz für die Leistung anhand des sehr rechenaufwändigen Kernels für das DRR Volume-Raytracing näher beleuchtet werden.

Die folgenden im Rahmen der Implementation gemachten Beobachtungen beziehen sich auf Testsystem 1 aus Tabelle 5.1.

In der NVIDIA CUDA-Architektur werden einzelne *Threads* zu sogenannten *Warps*[1] zusammengefasst. Alle *Threads* innerhalb eines *Warps* durchlaufen den *Kernel* gleichzeitig, so dass sich jeder *Thread* an der identischen Stelle im *Kernel* befindet. Auf der verwendeten Hardware besteht ein *Warp* aus 32 *Threads*. Laut [NVI09d] ist es für eine gute Leistung ratsam, die zu bearbeitenden Probleme an *Warp*-Größen auszurichten.

In den folgenden Experimenten wurde mit einem auf 1240×960 herunter skalierten Röntgenbild, dem nicht skalierten Volumen von $512 \times 512 \times 378$ und einer ROI von 997×717 Bildpunkten gearbeitet. Dadurch wird die GPU nicht überfordert, aber auch nicht unterfordert.

Im **ersten Experiment** wurde *clEnqueueNDRangeKernel* als lokale Größe ein *NULL*-Zeiger übergeben. Bei Nichtbeachtung der *Warps* wurde für das Volume-Raytracing eines DRR eine durchschnittliche Berechnungszeit von 672 ms gemessen.

In einem **zweiten Experiment** wurde die ROI von der Implementation automatisch auf einen durch 32 teilbaren Wert justiert. Alle Bildpunkte außerhalb der ursprünglichen ROI wurden dabei innerhalb des *Kernels* über Verzweigungen[2] ignoriert. Das von OpenCL zu bearbeitende Problem hatte nun folglich eine globale Problemgröße von 1024×736 statt 997×717. Als lokale Größe wurde *clEnqueueNDRangeKernel* weiterhin ein *NULL*-Zeiger übergeben. Mit der nun durch die *Warp*-Größe teilbaren globalen Problemgröße konnte eine durchschnittliche Berechnungszeit von 35 ms gemessen werden, was eine beachtliche Leistungssteigerung darstellt.

In einem **dritten Experiment** wurde die globalen Problemgröße auf einen durch 2 teilbaren Wert gesetzt, alle anderen Einstellungen blieben identisch. Dies resultierte in einer Berechnungszeit von 33 ms.

Für das **vierte Experiment** wurde die lokale Größe auf 16 gesetzt, dies entspricht bei einer *Warp*-Größe von 32 einem sogenannten *Halb-Warp*. Aufgrund der per Hand gesetzten lokalen Größe musste die globale Größe nun ein Vielfaches von 16 sein, daher wurde die globale Größe auf einen durch 32 teilbaren Wert gesetzt. Mit diesen Einstellungen betrug die Berechnungszeit des Volume-Raytracing für ein DRR 30 ms. Aufgrund der begrenzten Anzahl an Registern und dem aufwändigen DRR Volume-Raytracing Kernel, der viele Register pro Thread benötigt, waren auf dem verwendeten Testsystem lokale Größen von 32 oder mehr nicht möglich.

Um zu prüfen, ob sich OpenCL auf ATI-Hardware ähnlich verhält, wurden die Experimente auf Testsystem 2 aus Tabelle 5.1 wiederholt. Die Ergebnisse wurden in Tabelle 5.14 zusammengetragen. Da die Grafikkarte dieses Testsystems nicht genügend Grafikkartenspeicher besitzt, um das Volumen in seiner vollen Auflösung zu speichern, wurde ein auf $256 \times 256 \times 189$ herunter skaliertes Volumen verwendet. Die restlichen Einstellungen und der Ablauf blieben gleich.

[1] Auf ATI-Hardware Wavefront genannt

[2] engl. Branching

Experiment	System 1	System 2	Globale Größe	Lokale Größe
1	672 ms	2551 ms	Nicht verändert	*NULL*-Zeiger
2	35 ms	58 ms	Durch 32 teilbar	*NULL*-Zeiger
3	33 ms	1278 ms	Durch 2 teilbar	*NULL*-Zeiger
4	30 ms	49 ms	Durch 32 teilbar	16

Tabelle 5.14: Leistungsunterschiede globale und lokale Größe auf Testsystem 1 und 2

Es wurde gezeigt, dass die Einstellungen der globalen und lokalen Arbeitsgröße großen Einfluss auf die zu erwartende Leistung haben können. Dieses Ergebnis verdeutlicht, dass *Warp*-Größen für eine gute Leistung unbedingt einzuhalten sind und lokale Größen per Hand gesetzt werden sollten, auch wenn dies durch die OpenCL-Spezifikation nicht vorgeschrieben wird. Wie der Gegentest auf ATI-Hardware zeigte, verhalten sich die Einstellungen für die globale und lokale Größe nicht auf jeder Hardware identisch. Es ist allerdings zu beobachten, dass auf der verwendeten NVIDIA- und ATI-Hardware eine durch 32 teilbare globale Größe und eine auf 16 gesetzte lokale Größe für das DRR Volume-Raytracing die beste Leistung brachte.

Automatischer Neustart des Grafikkartentreibers Im Laufe der Experimente zum Feststellen der Auswirkungen von globaler und lokaler Problemgröße auf Testsystem 1 konnte noch eine weitere Beobachtung gemacht werden: Wurde das Röntgenbild in der ursprünglichen Auflösung von 2480×1920 verwendet, so wurde der Grafikkartentreiber bei schlechten *clEnqueueNDRangeKernel*-Einstellungen während der Berechnung vom Betriebssystem zurückgesetzt. Dies resultierte darin, dass anschließend keine weiteren Berechnungen mehr durchgeführt werden konnten, die Anwendung kam sehr schnell mit falschem Ergebnis zum Ende.

Dieses Phänomen lässt sich dadurch erklären, dass *Microsoft Windows Vista* mit einer Technik namens Windows Display Driver Model (WDDM) ausgestattet wurde. Diese Technik ermöglicht es dem Betriebssystem Hardwaretreiber zurückzusetzen, um einen kompletten Neustart des Systems zu vermeiden. Bei GPGPU-Anwendungen kann jedoch die Situation auftreten, dass Berechnungen sehr aufwändig sind und die Grafikkarte sich daher aus der Sicht des Betriebssystems nicht mehr normal verhält. Wenn *Windows Aero* aktiviert ist, so kann nach dem Beenden der Anwendung im harmlosen Fall eine Warnung vom Betriebssystem erscheinen, in der darauf hingewiesen wird, dass die Leistung beeinträchtigt ist. Als Problemlösung wird in diesem Hinweis empfohlen, *Windows Aero* zu deaktivieren. Bei deaktiviertem *Windows Aero* erschien diese Warnung nicht mehr. Sollte GPGPU die Grafikkarte außergewöhnlich stark belasten, funktioniert die Grafikkarte aus Sicht des Betriebssystems nicht mehr ordnungsgemäß, WDDM führt als Konsequenz automatisch einen Neustart des entsprechenden Treibers durch.

Diese Verhalten ist für GPGPU unangebracht, da aufwändige Berechnungen dadurch unmöglich sind.

Eine Lösung dieses Problems wird in [Mic09] beschrieben. Mit Hilfe der *Windows-Registry* lässt sich das WDDM-Verhalten konfigurieren. Durch das Setzen bestimmter Registry-Schlüssel kann die Zeitspanne bis zum Neustart des Grafikkartentreibers erhöht werden. Eine komplette Deaktivierung dieser Sicherungsmaßnahme ist nicht zu empfehlen, denn sollte der Grafikkartentreiber einmal wirklich nicht mehr reagieren, wäre ansonsten ein Neustart des gesamten Systems nötig.

Kapitel 6

Zusammenfassung

Die 2-D/3-D-Registrierung medizinischer Datensätze findet Einsatz in diagnostischen Anwendungen sowie bspw. in der operativen Medizin. Das Ziel dieser Aufgabenstellung besteht darin, eine geeignete Transformation zu finden, die ein MRT- oder CT-Volumen mit einem Röntgenbild in Übereinstimmung bringt. Dabei ist es wünschenswert, diese Überlagerung verschiedener Datensätze in Echtzeit zu ermitteln. Es wurde ein Überblick über die umfangreichen Grundlagen der 2-D/3-D-Registrierung gegeben. Wichtige Formeln wurden präsentiert und auf Literatur mit vertiefenden Informationen verwiesen.

Grafikprozessoren haben eine lange Evolution hinter sich. Nachdem diese spezielle Hardware anfangs lediglich Text- und einfache Grafikausgaben übernahm, wuchs im Laufe der Jahre der Funktionsumfang deutlich. Mit der stetig wachsenden Rechenleistung und mehr Freiheitsgraden in der Programmierung wurde ebenfalls das Aufgabengebiet ausgedehnt. Heutzutage sind auf Grafikprozessoren allgemeine Berechnungen möglich, die nichts mehr mit Grafik gemeinsam haben. Dieses Aufgabengebiet ist mittlerweile genauso wichtig wie die klassische Grafikerzeugung, die dieser Hardware ihren Namen verleiht. Für die Programmierung von allgemeinen Berechnungen auf GPUs entstanden zahlreiche eigene Programmierschnittstellen. Davon konnten sich wiederum nur wenige durchsetzen. Bisher hatte jeder Grafikkartenhersteller eigene GPGPU-Programmierschnittstellen, Entwickler mussten sich folglich für die Hardware eines Herstellers entscheiden. Wie gezeigt wurde, führt der neue OpenCL-Standard Welten zusammen. Im Gegensatz zur älteren CUDA-API ist es nun nicht nur möglich, GPGPU-Programme auf Grafikprozessoren unterschiedlicher Hersteller laufen zu lassen, sondern die Programme lassen sich auch auf beliebigen Plattformen wie z. B. einer einfachen CPU ausführen, da OpenCL nicht mehr nur für Grafikkarten spezifiziert wurde. Es wurde ein Überblick über den aktuellen Entwicklungsstand von OpenCL gegeben. Dabei wurde festgestellt, dass die Unterstützung von

OpenCL durch die Grafikkartentreiber mittlerweile einen Stand erreicht hat, der es ermöglicht, OpenCL in produktiven Systemen zu nutzen. Des Weiteren bekommen Entwickler von den Grafikkartenherstellern gute Unterstützung in Form von SDKs, Tutorials und Dokumentationen, sogar umfangreiche Video-Tutorials sind keine Seltenheit. Das Kapitel über die Grundlagen der GPU-Programmierung wurde mit einem Überblick über die wichtigsten Konzepte von OpenCL abgeschlossen.

Die schnelle 2-D/3-D-Registrierung medizinischer Datensätze auf Grafikprozessoren ist dank moderner GPGPU-Programmierschnittstellen vergleichsweise einfach umzusetzen. Diese API-Typen sind meist kompakt und die Problemstellungen lassen sich oftmals ohne aufwändige Umformulierungen der Problemstellung realisieren. Dies steht im deutlichen Gegensatz zu den traditionellen Grafik-Programmierschnittstellen. Besonders deutlich zeigte sich dies beim Volume-Raytracing Kernel zur realitätsnahen künstlichen Röntgenbild-Erzeugung. Die GPU-Implementation ist im Grunde identisch mit dem Schleifenrumpf einer sequentiellen CPU-Implementation. Andere Problemstellungen müssen jedoch gezielt parallelisiert werden, um eine effiziente Ausführung auf der GPU zu ermöglichen. Für viele Algorithmen existieren aber bereits gute parallelisierte Varianten, so lässt sich bspw. die parallele Reduktion, die ein wichtiges Kernproblem bei den Gütemaßen darstellt, mit vergleichsweise wenig Aufwand auf der GPU umsetzen. Für diese Umsetzung bietet die Literatur zahlreiche Varianten, die unterschiedlich komplex sind und unterschiedliche Laufzeiten besitzen. Es wurde eine CUDA-Implementation aus der Literatur aufgegriffen und nach OpenCL portiert.

Für das Volume-Rendering, das im Rahmen der Erzeugung des realitätsnahen künstlichen Röntgenbildes benötigt wird, existieren für Grafik-APIs zahlreiche Ansätze. Diese nutzen die Grafik-Pipeline und 3-D-Texturen, um eine gute Beschleunigung der Aufgabe auf der GPU zu erzielen. Diese Verfahren lassen sich jedoch nicht direkt auf moderne GPU-Programmierschnittstellen wie beispielsweise OpenCL übertragen. Allerdings stellt dies kein Problem dar. Wie festgestellt wurde, lässt sich Volume-Rendering über Volume-Raytracing mit OpenCL deutlich direkter und effektiver realisieren als mit älteren Grafik-APIs. Durch eine schnelle Schnittpunktberechnung direkt im Volume-Raytracing Kernel lässt sich der Datentransfer von und in den langsamen globalen Speicher der Grafikkarte reduzieren. Da Berechnungen deutlich schneller sind als Zugriffe auf den globalen Speicher, kann hierdurch eine Leistungssteigerung erzielt werden. Des Weiteren sind in einer OpenCL-Implementation weniger API-Aufrufe nötig als beispielsweise in einer OpenGL-Implementation, die mehrere Renderschritte benötigt. Dies verringert die Kommunikation zwischen CPU und GPU und wirkt sich positiv auf die zu erzielende Leistung aus.

Die rein intensitätsbasierten Gütemaße lassen sich direkt von der parallelen Reduktion ableiten. Eine Realisierung dieser Klasse von Gütemaßen auf der GPU stellt daher keine Herausforderung dar, sobald deren zugrundeliegenden Algorithmen effektiv auf der GPU implementiert wurden. Die Klasse der räumlich intensitätsbasierten Gütemaße lässt sich ebenfalls recht einfach auf der GPU implementieren. Durch die Nutzung von Texturen kann hier die Leistung weiter gesteigert werden. Histogramm-basierte Gütemaße fallen etwas aus dem Rahmen. Zwar kommt auch bei dieser Klasse von Gütemaßen die parallele Reduktion zum Einsatz, da jedoch auf Histogrammen und Verbundhistogrammen gearbeitet wird, müssen diese zuvor erzeugt werden. Die effiziente Parallelisierung der Berechnung von Histogrammen ist nicht trivial. In der Literatur werden verschiedenste Ansätze beschrieben, die sich jedoch meist an Komplexität gegenseitig überbieten. Trotz der aufwändigen Implementationen stellt sich gerade bei einer hohen Anzahl von möglichen Werten meist keine zufriedenstellende Leistung ein. Die einfachste Möglichkeit der Berechnung von Histogrammen mit OpenCL besteht in der Verwendung von atomaren Schreibzugriffen, deren Leistungsfähigkeit auf GPUs der aktuellen Generation jedoch noch eingeschränkt ist.

Im Rahmen dieser Arbeit wurde eine OpenCL-Implementation der 2-D/3-D-Registrierung erstellt. Die erzielte Beschleunigung des Gesamtsystems lag im Bereich von Faktor 2 bis Faktor 77, abhängig vom verwendeten Testsystem und der Größe des verwendeten Datensets. Auf einem Testsystem benötigte die gesamte 2-D/3-D-Registrierung des voll auflösenden medizinischen Datensatzes auf der CPU 48,57 min. Die mit Hilfe von OpenCL auf der GPU laufende Implementation benötigte hingegen lediglich 41,31 s.

Wie der Vergleich der ATI- und NVIDIA-Hardware verdeutlichte, ist es ratsam, OpenCL-Implementationen bereits während der Entwicklung auf verschiedenen Systemen zu testen. Mit dieser Vorgehensweise lassen sich Leistungsanomalien sowie eventuell fehlerhaftes Verhalten der verwendeten OpenCL-Implementation aufdecken. Solange Leistung im Hintergrund steht, lassen sich mit GPGPU-Programmierschnittstellen Problemstellungen einfacher und vor allem direkter umsetzen. Sobald jedoch die Leistung maximiert werden muss, muss sich der Entwickler auch mit dem Thema der Hardwarearchitektur auseinandersetzen. Um eine optimale Leistung zu erzielen, muss sich ein Entwickler auf die Hardwarearchitektur eines Herstellers konzentrieren. Da die Hardwareentwicklung allerdings gerade auf dem Grafikkarten-Markt weiterhin rasant voranschreitet, besteht die Gefahr, dass gemachte Optimierungen bereits auf der nächsten Hardware-Generation keine Vorteile mehr bringen. Im schlimmsten Fall besteht die Gefahr, dass die für eine bestimmte Hardwarearchitektur optimierte Implementation in der nächsten Hardware-Generation sogar langsamer läuft. Daher ist eine Optimierung nur in der Abschluss-

phase eines Projektes zu empfehlen. Es lässt sich das Fazit ziehen, dass OpenCL zwar plattformunabhängig ist, die OpenCL-Leistung hingegen keineswegs plattformunabhängig ist.

Aufgrund der festgestellten Unterschiede zwischen ATI- und NVIDIA-Hardware, ergibt sich zum momentanen Zeitpunkt das Fazit, dass für GPGPU-Anwendungen NVIDIA-Hardware zu bevorzugen ist. Es bleibt die Frage offen, ob sich der aktuelle Zustand im Laufe der Zeit ändern wird, so dass GPGPU auch auf der Hardware anderer Hersteller eine vorzeigbare Leistung erreicht.

Kapitel 7

Ausblick

Für die angefertigte Implementation war es wichtig, dass Komponenten getrennt zu testen waren, so das ein GPU/CPU-Vergleich der verschiedenen nötigen Schritte möglich wurde. Dies ermöglichte es ebenfalls, die Korrektheit der einzelnen Ergebnisse zu prüfen. Da OpenCL als Technologie noch jung ist und die OpenCL-Implementationen stellenweise noch kleinere Fehler aufweisen, war die Testbarkeit ein wichtiges Argument für diese das Softwaredesign betreffende Entscheidung. Wird auf eine strikte Trennung der einzelnen Teilschritte verzichtet, so wird es dagegen möglich, verschiedene Kernel zu einem einzigen zusammenfassen. Beispielsweise könnten direkt in dem Kernel, der das DRR in ein 8 bit Graustufenbild überführt, die Gütemaße SSD, SAD, SPD und SDT untergebracht werden. Teilschritte anderer Gütemaße ließen sich ebenfalls in diesen Kernel mit einbauen. Durch die Verschmelzung wären weniger API-Aufrufe nötig, wodurch die Kommunikation zwischen CPU und GPU weiter reduziert werden könnte. Ebenfalls könnte durch diese Änderung die Anzahl der Zugriffe auf den globalen Speicher reduziert werden. Durch eine Zusammenführung verschiedener Kernel könnten möglicherweise auch weitere Speicherlatenzen durch mehr Berechnungen in weniger Kernel verborgen werden. Daher wäre es interessant herauszufinden, ob das Zusammenfassen der Kernel einen messbaren Leistungsgewinn bringt.

Mit *NVIDIA SLI* und *ATI Crossfire* stehen Multi-GPU-Technologien zur Verfügung. Durch *Quad-SLI* besteht die Möglichkeit, bis zu vier Grafikkarten in einem System gleichzeitig zu betreiben. Über OpenCL ließe sich jede dieser Grafikkarten als eigenes Gerät ansprechen. Die im Rahmen dieser Arbeit erstellte Implementation für die 2-D/3-D-Registrierung medizinischer Datensätze könnte um Unterstützung für Multi-GPUs erweitert werden. Da jede Grafikkarte ihren eigenen Grafikspeicher besitzt, wäre es voraussichtlich am effektivsten, wenn jede GPU jeweils einen eigenen Schritt der Qualitätsprüfung unabhängig von den anderen Grafikkarten

ausführen würde. Durch diesen Ansatz wäre keine aufwändige Synchronisation der Berechnungen mehr nötig. Mehrere Qualitätsprüfungen müssten nebeneinander lauffähig sein, dafür müsste der Optimierungs-Algorithmus angepasst werden.

OpenCL ist auf Plattformunabhängigkeit ausgelegt, es geht sogar soweit, dass mehrere OpenCL-Implementationen unterschiedlicher Hersteller auf ein und demselben System installiert sein können. Über die am 29. Januar 2010 von Khronos festgelegte *cl_khr_icd*-Erweiterung besteht für Entwickler die Möglichkeit, auf den Installable Client Driver (ICD) Lademechanismus zuzugreifen, mit dem die verschiedenen Hersteller-Implementationen verwaltet werden. NVIDIA bietet derzeit keine CPU-Implementation von OpenCL an, über ICD wäre es allerdings möglich, die entsprechende Implementation von ATI Stream zu nutzen, so dass Entwicklern mit NVIDIA-Hardware kein Nachteil entstünde. Da die Hersteller diesen Mechanismus erst allmählich korrekt umsetzen, war es jedoch noch nicht möglich, dies an den vorhandenen Systemen auszuprobieren. Es wäre interessant zu prüfen, ob dies am Ende wirklich so wie vorgesehen funktioniert.

Ebenfalls wäre es interessant herauszufinden, ob die speziell auf GPGPU ausgerichtete NVIDIA Tesla für die 2-D/3-D-Registrierung geeigneter wäre, als eine traditionelle High-End-Grafikkarte wie beispielsweise die *GeForce 295 GTX*. Neben Grafikprozessoren mit heute üblichen Ansätzen existieren ausserdem weitere, auf extreme Parallelisierung ausgerichtete Hardware. Intels *Larrabee* verwendet beispielsweise einen Cluster von mehreren CPUs, die auf Pentium-Prozessoren basieren. Ein Kostenvergleich könnte aufzeigen, welche Beschleunigung durch Nutzung von schnellerer und teurere Hardware möglich wäre.

Zum Zeitpunkt der Anfertigung dieser Arbeit steht bereits die nächste GPU-Generation von NVIDIA in den Startlöchern. GPGPU relevante Details zur *Fermi* getauften Generation lassen sich in [NVI09c] und [NVI10] nachlesen. Neben den üblichen Geschwindigkeitssteigerungen werden ebenfalls einige für GPGPU wichtige Verbesserungen eingeführt. So wurde der maximal nutzbare lokale Speicher auf 48 KiB erhöht und atomare Schreibaktionen seien bis zu 20 mal schneller verglichen zu *GT200*-GPUs. Gerade diese zwei genannten Verbesserungen müssten die Berechnung von Verbundhistogrammen auf der GPU deutlich effektiver machen, als dies bisher möglich ist. Laut [Hal09] sind auf Fermi Fließkommazahlen mit doppelter Genauigkeit halb so schnell wie Fließkommazahlen mit einfacher Genauigkeit, damit seien Fließkommazahlen mit doppelter Genauigkeit bis zu 8 mal schneller als auf *GT200*-GPUs. Jeder CUDA-Kern der neuen Generation besitzt neben einer Einheit für Fließkommazahlen ebenfalls eine Einheit für 32 bit-Integer-Zahlen. Es stellt sich die spannende Frage, ob allein durch diese neue Hardwarearchitektur die 2-D/3-D-Registrierung noch weiter beschleunigt werden kann.

Anhang A

OpenCL-Eigenschaften der Testsysteme

A.1 NVIDIA GeForce 285 GTX

Der Hersteller NVIDIA ordnet der eigenen Hardware eine sogenannte *Compute Capability* zu. Die in [NVI09d] beschriebenen *Compute Capability* Versionen fassen für CUDA wichtige GPU-Eigenschaften zusammen. Die im Rahmen dieser Arbeit verwendete Grafikkarte *NVIDIA GeForce 285 GTX*[1] unterstützt *Compute Capability 1.3*. In Tabelle A.1 sind die über *clGetDeviceInfo* ermittelten OpenCL-Eigenschaften aufgelistet.

Eigenschaft	Wert
NAME	GeForce GTX 285
TYPE	CL_DEVICE_TYPE_GPU
VENDOR	NVIDIA Corporation
VENDOR_ID	4318
VERSION	OpenCL 1.0 CUDA
CL_DRIVER_VERSION	196.21
ADDRESS_BITS	32
AVAILABLE	Ja
COMPILER_AVAILABLE	Ja
ENDIAN_LITTLE	Ja
ERROR_CORRECTION_SUPPORT	Nein
Fortsetzung auf der nächsten Seite	

[1] Codename GT200b

Tabelle A.1 – Fortsetzung der vorherigen Seite

Eigenschaft	Wert
EXECUTION_CAPABILITIES	CL_EXEC_KERNEL
EXTENSIONS	cl_khr_byte_addressable_store cl_khr_gl_sharing cl_nv_compiler_options cl_nv_device_attribute_query cl_nv_pragma_unroll cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_fp64
GLOBAL_MEM_CACHE_SIZE	0 B
GLOBAL_MEM_CACHE_TYPE	CL_NONE
GLOBAL_MEM_CACHELINE_SIZE	0 B
GLOBAL_MEM_SIZE	2 147 483 648 B = 2 097 152 KiB = 2048 MiB
IMAGE_SUPPORT	Ja
IMAGE2D_MAX_HEIGHT	8192
IMAGE2D_MAX_WIDTH	8192
IMAGE3D_MAX_DEPTH	2048
IMAGE3D_MAX_HEIGHT	2048
IMAGE3D_MAX_WIDTH	2048
LOCAL_MEM_SIZE	16 384 B = 16 KiB
LOCAL_MEM_TYPE	CL_LOCAL
MAX_CLOCK_FREQUENCY	1476 MHz
MAX_COMPUTE_UNITS	30
MAX_CONSTANT_ARGS	9
MAX_CONSTANT_BUFFER_SIZE	65 536 B = 64 KiB
MAX_MEM_ALLOC_SIZE	536 870 912 B = 524 288 KiB = 512 MiB
MAX_PARAMETER_SIZE	4352 B = 4 KiB
MAX_READ_IMAGE_ARGS	128

Fortsetzung auf der nächsten Seite

Tabelle A.1 – Fortsetzung der vorherigen Seite

Eigenschaft	Wert
MAX_SAMPLERS	16
MAX_WORK_GROUP_SIZE	512
MAX_WORK_ITEM_DIMENSIONS	3
MAX_WORK_ITEM_SIZES	$512 \times 512 \times 64$
MAX_WRITE_IMAGE_ARGS	8
MEM_BASE_ADDR_ALIGN	256 bit = 32 B
MIN_DATA_TYPE_ALIGN_SIZE	16 B
PREFERRED_VECTOR_WIDTH_CHAR	1
PREFERRED_VECTOR_WIDTH_SHORT	1
PREFERRED_VECTOR_WIDTH_INT	1
PREFERRED_VECTOR_WIDTH_LONG	1
PREFERRED_VECTOR_WIDTH_FLOAT	1
PREFERRED_VECTOR_WIDTH_DOUBLE	1
PROFILE	FULL_PROFILE
PROFILING_TIMER_RESOLUTION	1000 ns
QUEUE_PROPERTIES	CL_QUEUE_OUT_OF_ORDER _EXEC_MODE_ENABLE CL_QUEUE_PROFILING_ENABLE
SINGLE_FP_CONFIG	CL_FP_INF_NAN CL_FP_ROUND_TO_NEAREST CL_FP_ROUND_TO_ZERO CL_FP_ROUND_TO_INF
HALF_FP_CONFIG	-
DOUBLE_FP_CONFIG	CL_FP_DENORM CL_FP_INF_NAN CL_FP_ROUND_TO_NEAREST CL_FP_ROUND_TO_ZERO CL_FP_ROUND_TO_INF
PROFILING_TIMER_RESOLUTION	32 ns
COMPUTE_CAPABILITY_NV	1.3

Fortsetzung auf der nächsten Seite

Tabelle A.1 – Fortsetzung der vorherigen Seite

Eigenschaft	Wert
REGISTERS_PER_BLOCK_NV	16384
WARP_SIZE_NV	32
GPU_OVERLAP_NV	Ja
KERNEL_EXEC_TIMEOUT_NV	Nein
INTEGRATED_MEMORY_NV	Nein

Tabelle A.1: OpenCL-Eigenschaften von Testsystem 1 (NVIDIA GeForce 285 GTX)

A.2 ATI Mobility Radeon HD 4850

Die Tabelle A.2 listet die über *clGetDeviceInfo* ermittelten OpenCL-Eigenschaften der verwendeten Grafikkarte *ATI Mobility Radeon HD 4850*[2] auf. Details über die ISA dieser Grafikkarte werden in [AMD09e] beschrieben. Ebenfalls finden sich in diesem Dokument Informationen zu Local Data Share (LDS), trotzdem wird bei OpenCL derzeit der lokale Speicher lediglich über den globalen Speicher emuliert.

Eigenschaft	Wert
NAME	ATI RV770
TYPE	CL_DEVICE_TYPE_GPU
VENDOR	Advanced Micro Devices, Inc.
VENDOR_ID	4098
VERSION	OpenCL 1.0 ATI-Stream-v2.0.1
CL_DRIVER_VERSION	CAL 1.4.519
ADDRESS_BITS	32
AVAILABLE	Ja
COMPILER_AVAILABLE	Ja

Fortsetzung auf der nächsten Seite

[2]Codename M98, basiert auf RV770

Tabelle A.2 – Fortsetzung der vorherigen Seite

Eigenschaft	Wert
ENDIAN_LITTLE	Ja
ERROR_CORRECTION_SUPPORT	Nein
EXECUTION_CAPABILITIES	CL_EXEC_KERNEL
EXTENSIONS	cl_khr_icd
GLOBAL_MEM_CACHE_SIZE	0 B
GLOBAL_MEM_CACHE_TYPE	CL_NONE
GLOBAL_MEM_CACHELINE_SIZE	0 B
GLOBAL_MEM_SIZE	134 217 728 B = 131 072 KiB = 128 MiB
IMAGE_SUPPORT	Nein
IMAGE2D_MAX_HEIGHT	0
IMAGE2D_MAX_WIDTH	0
IMAGE3D_MAX_DEPTH	0
IMAGE3D_MAX_HEIGHT	0
IMAGE3D_MAX_WIDTH	0
LOCAL_MEM_SIZE	16 384 B = 16 KiB
LOCAL_MEM_TYPE	CL_GLOBAL
MAX_CLOCK_FREQUENCY	500 MHz
MAX_COMPUTE_UNITS	10
MAX_CONSTANT_ARGS	8
MAX_CONSTANT_BUFFER_SIZE	65 536 B = 64 KiB
MAX_MEM_ALLOC_SIZE	134 217 728 B = 131 072 KiB = 128 MiB
MAX_PARAMETER_SIZE	1024 B = 1 KiB
MAX_READ_IMAGE_ARGS	0
MAX_SAMPLERS	0
MAX_WORK_GROUP_SIZE	256
MAX_WORK_ITEM_DIMENSIONS	3
MAX_WORK_ITEM_SIZES	$256 \times 256 \times 256$
MAX_WRITE_IMAGE_ARGS	0
MEM_BASE_ADDR_ALIGN	4096 bit = 512 B
MIN_DATA_TYPE_ALIGN_SIZE	128 B

Fortsetzung auf der nächsten Seite

Tabelle A.2 – Fortsetzung der vorherigen Seite

Eigenschaft	Wert
PREFERRED_VECTOR_WIDTH_CHAR	16
PREFERRED_VECTOR_WIDTH_SHORT	8
PREFERRED_VECTOR_WIDTH_INT	4
PREFERRED_VECTOR_WIDTH_LONG	2
PREFERRED_VECTOR_WIDTH_FLOAT	4
PREFERRED_VECTOR_WIDTH_DOUBLE	0
PROFILE	FULL_PROFILE
PROFILING_TIMER_RESOLUTION	1 ns
QUEUE_PROPERTIES	CL_QUEUE_PROFILING_ENABLE
SINGLE_FP_CONFIG	CL_FP_INF_NAN CL_FP_ROUND_TO_NEAREST
HALF_FP_CONFIG	-
DOUBLE_FP_CONFIG	-
PROFILING_TIMER_RESOLUTION	0 ns

Tabelle A.2: OpenCL CL_DEVICE_ Eigenschaften von Testsystem 2 (ATI Mobility Radeon HD 4850)

A.3 Intel Core 2 Quad CPU Q9000

Die OpenCL-Implementation von ATI unterstützt ebenfalls Berechnungen auf der CPU. Der Vollständigkeit halber wurden in Tabelle A.3 die über *clGetDeviceInfo* ermittelten OpenCL-Eigenschaften der verwendeten CPU aufgelistet.

Eigenschaft	Wert
NAME	Intel Core 2 Quad CPU Q9000 2,00 GHz
TYPE	CL_DEVICE_TYPE_CPU
VENDOR	GenuineIntel
VENDOR_ID	4098

Fortsetzung auf der nächsten Seite

Tabelle A.3 – Fortsetzung der vorherigen Seite

Eigenschaft	Wert
VERSION	OpenCL 1.0 ATI-Stream-v2.0.1
CL_DRIVER_VERSION	1.0
ADDRESS_BITS	64
AVAILABLE	Ja
COMPILER_AVAILABLE	Ja
ENDIAN_LITTLE	Ja
ERROR_CORRECTION_SUPPORT	Nein
EXECUTION_CAPABILITIES	CL_EXEC_KERNEL
EXTENSIONS	cl_khr_icd cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_int64_base_atomics cl_khr_int64_extended_atomics cl_khr_byte_addressable_store
GLOBAL_MEM_CACHE_SIZE	65 536 B = 64 KiB
GLOBAL_MEM_CACHE_TYPE	CL_READ_WRITE_CACHE
GLOBAL_MEM_CACHELINE_SIZE	64 B
GLOBAL_MEM_SIZE	3 221 225 472 B = 3 145 728 KiB = 3072 MiB
IMAGE_SUPPORT	Nein
IMAGE2D_MAX_HEIGHT	0
IMAGE2D_MAX_WIDTH	0
IMAGE3D_MAX_DEPTH	0
IMAGE3D_MAX_HEIGHT	0
IMAGE3D_MAX_WIDTH	0
LOCAL_MEM_SIZE	32 768 B = 32 KiB
LOCAL_MEM_TYPE	CL_GLOBAL
MAX_CLOCK_FREQUENCY	1995 MHz
MAX_COMPUTE_UNITS	4

Fortsetzung auf der nächsten Seite

Tabelle A.3 – Fortsetzung der vorherigen Seite

Eigenschaft	Wert
MAX_CONSTANT_ARGS	8
MAX_CONSTANT_BUFFER_SIZE	65 536 B = 64 KiB
MAX_MEM_ALLOC_SIZE	1 073 741 824 B = 1 048 576 KiB = 1024 MiB
MAX_PARAMETER_SIZE	4096 B = 4 KiB
MAX_READ_IMAGE_ARGS	0
MAX_SAMPLERS	0
MAX_WORK_GROUP_SIZE	1024
MAX_WORK_ITEM_DIMENSIONS	3
MAX_WORK_ITEM_SIZES	$1024 \times 1024 \times 1024$
MAX_WRITE_IMAGE_ARGS	0
MEM_BASE_ADDR_ALIGN	32 768 bit = 4096 B = 4 KiB
MIN_DATA_TYPE_ALIGN_SIZE	128 B
PREFERRED_VECTOR_WIDTH_CHAR	16
PREFERRED_VECTOR_WIDTH_SHORT	8
PREFERRED_VECTOR_WIDTH_INT	4
PREFERRED_VECTOR_WIDTH_LONG	2
PREFERRED_VECTOR_WIDTH_FLOAT	4
PREFERRED_VECTOR_WIDTH_DOUBLE	0
PROFILE	FULL_PROFILE
PROFILING_TIMER_RESOLUTION	1 ns
QUEUE_PROPERTIES	CL_QUEUE_PROFILING_ENABLE
SINGLE_FP_CONFIG	CL_FP_DENORM CL_FP_INF_NAN CL_FP_ROUND_TO_NEAREST
HALF_FP_CONFIG	-
DOUBLE_FP_CONFIG	-
PROFILING_TIMER_RESOLUTION	0 ns

Tabelle A.3: OpenCL CL_DEVICE_ Eigenschaften von Testsystem 2 (Intel Core 2 Quad CPU Q9000 2,00 GHz)

Anhang B

OpenCL-Werkzeuge

OpenCL ist eine noch recht junge Technologie, daher ist die Auswahl an Werkzeugen für Entwickler noch sehr eingeschränkt. ATI bietet bereits öffentlich verfügbare OpenCL-Werkzeuge an, weshalb es möglich war, diese im Rahmen der Arbeit zu testen. Aus diesem Grund werden diese Werkzeuge von ATI hier etwas ausführlicher beschrieben. Die OpenCL-Werkzeuge von NVIDIA befanden sich zum Zeitpunkt der Ausarbeitung dagegen noch in geschlossenen Beta-Phasen oder in öffentlich verfügbaren Beta-Phasen, die allerdings mit aktuellen Grafikkartentreibern noch nicht korrekt liefen. Der Vollständigkeit halber sollen diese kommenden Werkzeuge von NVIDIA jedoch nicht unerwähnt bleiben.

Stream KernelAnalyzer Mit der am 11. Februar 2010 erschienenen Version von *Stream KernelAnalyzer*[1] [AMD09a] von ATI steht ein einfaches aber hilfreiches Programm zur Verfügung, mit dessen Hilfe es möglich ist, OpenCL-Programme zu analysieren. Nachdem eine Datei mit OpenCL-Quellcode[2] in *Stream KernelAnalyzer* eingeladen wurde, lässt sich das OpenCL-Programm mit einem Knopfdruck für verschiedene GPUs von ATI übersetzen. Neben der Anzeige des erzeugten Objekt-Codes werden auch diverse Statistiken ausgegeben. Besonders hilfreich ist die Anzeige der Anzahl der pro Thread benötigten Register. Die Anzahl der Register ist einer der Einflussfaktoren dafür, wie viele Kernelinstanzen parallel laufen können, es sei jedoch angemerkt, dass eine Verringerung der Anzahl der benötigten Register pro Thread nicht gleichbedeutend mit besserer Leistung ist. *Stream KernelAnalyzer* zeigt ebenfalls eine Thread-Durchsatzrate an, die eine recht zuverlässige Aussage darüber trifft, ob sich eine gemachte Änderung positiv oder negativ auf die zu erwartende Leistung auswirkt. Ein weiteres Feld der Statistik gibt

[1] http://developer.amd.com/GPU/SKA/Pages/default.aspx, verwendete Version: v1.4.515
[2] *cl*-Dateiendung

Auskunft darüber, wo ein Flaschenhals zu erwarten ist. Bei den OpenCL-Programmen der erstellten Implementation wird die Leistung laut *Stream KernelAnalyzer* meistens durch die Arithmetic Logic Unit (ALU)-Operationen begrenzt. Lediglich ein einziges OpenCL-Programm hatte - ausschließlich auf der *Radeon HD 5870 - Global Fetch* als Flaschenhals. OpenCL-Quellcode kann in diesem Werkzeug direkt bearbeitet werden, Fehler werden dem Entwickler in lesbarer Form mitgeteilt und Auswirkungen auf die Leistung lassen sich sofort erkennen. Mit Hilfe dieses Werkzeugs stellte sich an einigen Stellen im Nachhinein heraus, dass vermeintliche Optimierungen in der Implementation sich in Wirklichkeit negativ auf die Leistung auswirkten, auch wenn diese Auswirkungen meist nur sehr gering waren. Dies zeigt jedoch, dass der Einsatz von Analysewerkzeugen gerade in der Optimierungsphase eines Projektes sehr hilfreich sein kann.

ATI Stream Profiler Bei *ATI Stream Profiler*[3] handelt es sich um eine Erweiterung für *Microsoft Visual Studio 2008*, die im *ATI Stream SDK v2.01* enthalten ist oder separat heruntergeladen werden kann. Mit Hilfe dieses Werkzeugs ist es möglich, direkt von *Visual Studio* aus die eigene Anwendung mit einem OpenCL-Profiler zu starten, um die Leistung zu analysieren. Im Gegensatz zu *Stream KernelAnalyzer* werden die Statistiken dabei anhand der erstellten OpenCL-Anwendung erstellt und nicht nur durch die Analyse des OpenCL-Quellcodes. Das Werkzeug ist in der Lage, die vom OpenCL-Kernel benötigte Zeit zu messen, den Datentransfer von und zur GPU festzustellen, sowie weitere Daten zu sammeln. Alle Messergebnisse werden gemeinsam in eine *CSV*-Datei[4] geschrieben. Des Weiteren werden die übersetzten IL- und ISA-Codes ausgegeben.

OpenCL Visual Profiler Der *OpenCL Visual Profiler*[5] von NVIDIA ist eine eigenständige Anwendung, die es ermöglicht, die Leistung von OpenCL-Anwendungen zu analysieren. Im Gegensatz zu *Stream KernelAnalyzer* analysiert das Programm nicht den OpenCL-Quellcodes, sondern arbeitet mit jeweils übergebenen ausführbaren Anwendungen.

NVIDIA Parallel Nsight *NVIDIA Parallel Nsight*[6], auch *Nexus* genannt, ist eine Erweiterung für *Microsoft Visual Studio 2008* [NVI09c]. Mit diesem Werkzeug soll es möglich sein, CUDA C, OpenCL, DirectCompute, Direct3D und OpenGL GPU-Codes zu analysieren und ebenfalls zu debuggen.

[3] http://developer.amd.com/GPU/STREAMPROFILER/Pages/default.aspx, verwendete Version: 1.1
[4] engl. Character Separated Values (CSV)
[5] http://developer.nvidia.com/object/opencl-download.html
[6] http://developer.nvidia.com/object/nexus_features.html

Literaturverzeichnis

[AMD07] Advanced Micro Devices, I. A.: *AMD stellt ersten Stream-Prozessor mit Fließkomma-Technologie in doppelter Genauigkeit vor*, AMD Pressemitteilung, 2007, http://www.amd.com/de/press-releases/Pages/Press_Release_121775.aspx, Letzter Zugriff 15.02.2010.

[AMD09a] Advanced Micro Devices, I. A.: *ATI Stream Computing User Guide*, AMD Developer Central, 2009, rev1.4.0a, April 2009, http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf, Letzter Zugriff 27.02.2010.

[AMD09b] Advanced Micro Devices, I. A.: *A Brief History of General Purpose (GPGPU) Computing*, AMD Developer Central, 2009, http://ati.amd.com/technology/streamcomputing/gpgpu_history.html, Letzter Zugriff 15.02.2010.

[AMD09c] Advanced Micro Devices, I. A.: *Compute Abstraction Layer (CAL) Technology - Intermediate Language (IL)*, AMD Developer Central, 2009, v.2.0 February 2009, http://developer.amd.com/gpu_assets/Intermediate_Language_Specification--Stream_Processor.pdf, Letzter Zugriff 13.02.2010.

[AMD09d] Advanced Micro Devices, I. A.: *Porting CUDA to OpenCL*, AMD Developer Central, 2009, http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx#four, Letzter Zugriff 12.02.2010.

[AMD09e] Advanced Micro Devices, I. A.: *R700-Family Instruction Set Architecture*, AMD Developer Central, 2009, Revision 1.0 March 2009, http://developer.amd.com/gpu_assets/R700-Family_Instruction_Set_Architecture.pdf, Letzter Zugriff 13.02.2010.

[Ast82] Astola, J.; Virtanen, I.: *Entropy correlation coefficient, a measure of statistical dependence for categorized data*, *Proceedings of the University of Vaasa*, Bd. 44, 1982.

[ATI01] ATI, : *SMARTSHADER TECHNOLOGY WHITE PAPER*, ATI Website, 2001, http://ati.amd.com/products/pdf/smartshader.pdf, Letzter Zugriff 17.02.2010.

[Fer04] Fernando, R.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Addison-Wesley Professional, 2004, Part VI: Beyond Triangles.

[Gro09] Group, K. O. W.: *The OpenCL Specification*, Khronos Group Website, 2009, Version: 1.0, Document Revision: 48, http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf, Letzter Zugriff: 11.02.2010.

[Hal09] Halfhill, T. R.: *Looking Beyond Graphics - NVIDIA's Next-Generation CUDA Compute and Graphics Architecture, Code-Named Fermi, Adds Muscle for Parallel Processing*, NVIDIA Website, 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_LookingBeyondGraphics.pdf, Letzter Zugriff 15.02.2010.

[Har07] Harris, M.: *Optimizing Parallel Reduction in CUDA*, NVIDIA Website, 2007, Folien einer Präsentation, http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf, Letzter Zugriff 04.03.2010.

[Kay86] Kay, T. L.; Kajiya, J. T.: *Ray tracing complex scenes*, in *SIGGRAPH*, 1986, S. 269–278.

[Krü03] Krüger, J.; Westermann, R. (Hrsg.): *Acceleration Techniques for GPU-based Volume Rendering*, Proceedings of the 14th IEEE Visualization 2003 (VIS'03), IEEE Computer Society, Washington, DC, USA, 2003.

[Kre08] Kreuder, F.: *2D-3D-Registrierung mit Parameterentkopplung für die Patientenlagerung in der Strahlentherapie*, *Karlsruhe Transactions on Biomedical Engineering*, Bd. 7, 2008.

[Kub08] Kubias, A.: *Effiziente, adaptive 2D/3D-Registrierung: Überblick und neue Ansätze zu Registrierungsverfahren in der medizinischen Bildverarbeitung*, VDM Verlag Dr. Müller, 2008.

[Mae97] Maes, F.; Collignon, A.; Vandermeulen, D.; Marchal, G.; Suetens, P.: *Multimodality Image Registration by Maximization of Mutual Information*, *IEEE TRANSACTIONS ON MEDICAL IMAGING*, Bd. 16, no. 2, 1997, S. 187–198.

[Mic09] Microsoft, : *Timeout Detection and Recovery of GPUs through WDDM*, Windows Hardware Developer Central, 2009, http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx, Letzter Zugriff: 11.02.2010.

[Ngu07] Nguyen, H.: *GPU Gems 3*, Addison-Wesley Professional, 2007, Part VI: GPU Computing.

[NVI06] NVIDIA, : *NVIDIA präsentiert neue Computing-Technologie*, NVIDIA Pressemitteilung, 2006, http://www.nvidia.de/object/IO_37510.html, Letzter Zugriff 15.02.2010.

[NVI07a] NVIDIA, : *NVIDIA bietet kostenlose Beta-Version seiner neuen GPU-Computing-Technologie*, NVIDIA Pressemitteilung, 2007, http://www.nvidia.de/object/IO_39977.html, Letzter Zugriff 15.02.2010.

[NVI07b] NVIDIA, : *NVIDIA CUDA 1.0: Development Kit für GPU-Programmierung*, NVIDIA Pressemitteilung, 2007, http://www.nvidia.de/object/IO_44264.html, Letzter Zugriff 15.02.2010.

[NVI08] NVIDIA, : *Neue Tesla-Serie von NVIDIA verdoppelt Performance beim Supercomputing*, NVIDIA Pressemitteilung, 2008, http://www.nvidia.de/object/io_1213787517370.html, Letzter Zugriff 15.02.2010.

[NVI09a] NVIDIA, : *NVIDIA Compute - PTX: Parallel Thread Execution*, NVIDIA Developer Website, 2009, ISA Version 1.4, 31.03.2009, http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf, Letzter Zugriff 12.02.2010.

[NVI09b] NVIDIA, : *NVIDIA OpenCL JumpStart Guide*, NVIDIA Developer Website, 2009, Version 0.9 - April 2009, http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf, Letzter Zugriff 12.02.2010.

[NVI09c] NVIDIA, : *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, NVIDIA Website, 2009, V1.1, http://www.nvidia.com/attach/2954622?type=support&primitive=0, Letzter Zugriff 12.02.2010.

[NVI09d] NVIDIA, : *OpenCL Programming Guide for the CUDA Architecture*, NVIDIA Developer Website, 2009, Version: 2.3 - 8/27/2009, http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf, Letzter Zugriff 12.02.2010.

[NVI10] NVIDIA, : *NVIDIA GF100*, NVIDIA Website, 2010, V1.4, http://www.nvidia.com/attach/2986289?type=support&primitive=0, Letzter Zugriff 12.02.2010.

[Pha05] Pharr, M.; Fernando, R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley Professional, 2005, Part IV - General-Purpose Computation on GPUs: A Primer.

[Sch07] Scheuermann, T.; Hensley, J.: *Efficient Histogram Generation Using Scattering on GPUs*, Paper presented at the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07), Seattle, Washington, April 30-May 02, 2007, 2007, http://developer.amd.com/media/gpu_assets/GPUHistogramGeneration_preprint.pdf, Letzter Zugriff: 17.02.2010.

[Sha48] Shannon, C. E.: *A Mathematical Theory of Communication*, *Bell Syst. Tech. J.*, Bd. 27, 1948, S. 379–423, 623–656.

[Sha09] Shams, R.; Sadeghi, P.; Kennedy, R.; Hartley, R.: *Parallel Computation of Mutual Information on the GPU with Application to Real-Time Registration of 3D Medical Images*, *College of Engineering and Computer Science (CECS), The Australian National University, Canberra, ACT 0200, Australia*, 2009.

[Wan09] Wang, P.: *OpenCL Optimization*, NVIDIA Website, 2009, Folien einer Präsentation, http://www.nvidia.com/content/GTC/documents/1068_GTC09.pdf, Letzter Zugriff 20.02.2010.

Bilderverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Hilfsmittelverzeichnis

Schriftliche Ausarbeitung:

- Von Prof. Dr.-Ing. Frank Deinzer bereitgestelltes LaTeX-Template (*BA-MT-Vorlage_Ver.2.zip*)
- LaTeX für Windows: MiKTeX 2.8 (http://miktex.org/)
- Grafischer LaTeX-Editor: Texmaker 1.9.2 (http://www.xm1math.net/texmaker/)
- Notepad++ v5.5.1 (http://notepad-plus.sourceforge.net/de/site.htm)

Grafiken:

- Microsoft Office Visio 2007
- Bullzip PDF Printer 7.1.0.1136 (http://www.bullzip.com/)
- Inkscape 0.47 (http://www.inkscape.org/)
- Paint.NET v3.5.1 (http://www.getpaint.net/index.html)
- GIMP 2.6.8 (http://www.gimp.org/)
- OpenOffice.org 3.2 (http://de.openoffice.org/)

Implementierung:

- Von Prof. Dr.-Ing. Frank Deinzer und Siemens bereitgestellte exemplarische CPU-Implementierung der 2-D/3-D-Registrierung
- Entwicklungsumgebung: Visual Studio Team System 2008
- ATI Stream SDK v2.01 (http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx)

- GPU Computing SDK 3.0 Beta1 (http://forums.nvidia.com/index.php?showtopic=149959~und~http://www.nvidia.com/object/cuda_opencl.html)
- The OpenG Extension Wrangler Library (GLEW) 1.5.2 (http://glew.sourceforge.net/)

Quellcode Dokumentation:

- Doxygen 1.6.2 (http://www.doxygen.org/)
- Graphviz 2.26.2 (http://www.graphviz.org/)
- Microsoft HTML Help Compiler 4.74.8702

Abkürzungsverzeichnis

ROI Region Of Interest, Deutsch: Bereich von Interesse
DRR Digitally Reconstructed Radiograph, Deutsch: Künstliche Rückprojektion
FLL Fluoroscopic image, Deutsch: Durchleuchtungsbild
SSD Sum of Squared Differences, Deutsch: Summe der quadratischen Abstände
SPD Sum of Positive Differences, Deutsch: Summe der positiven Differenzen
SAD Sum of Absolute Differences, Deutsch: Summe der absoluten Abstände
SDT Sum of the absolute values of Differences above a Threshold, Deutsch: Summe der absoluten Abstände oberhalb eines Schwellwertes
GC Gradient Correlation, Deutsch: Gradientenkorrelation
NCC Normalized Cross Correlation, Deutsch: Normalisierte Kreuzkorrelation
JE Joint Entropy, Deutsch: Verbundentropie
MI Mutual Information, Deutsch: Transinformation/gegenseitige Information
NMI Normalized Mutual Information, Deutsch: Normalisierte Transinformation
ECC Entropy Correlation Coefficient, Deutsch: Entropie Korrelationskoeffizient
MRT Magnet-Resonanz-Tomographie
CT Computer-Tomographie
ALU Arithmetic Logic Unit, Deutsch: arithmetisch-logische Einheit
DSP Digital Signal Processing, Deutsch: digitale Signalverarbeitung
WDDM Windows Display Driver Model
ISA Instruction Set Architecture, Deutsch: Befehlssatzarchitektur
JIT Just In Time
API Application Programming Interface, Deutsch: Schnittstelle zur Anwendungsprogrammierung
SDK Software Development Kit, Deutsch: Softwareentwicklungssystem
CPU Central Processing Unit, Deutsch: Hauptprozessor
GPU Graphics Processing Unit, Deutsch: Grafikprozessor

GPGPU General-Purpose Computing on Graphics Processing Units, Deutsch: Allgemeine Berechnungen auf Grafikprozessoren

CSV Character Separated Values, Deutsch: Durch ein Zeichen getrennte Werte

RAM Random-Access Memory, Deutsch: Speicher mit wahlfreiem/direktem Zugriff

OS Operating System, Deutsch: Betriebssystem

PC Personal Computer, Deutsch: Einzelplatzrechner

WHQL Windows Hardware Quality Labs

OpenCL Open Computing Language

ICD Installable Client Driver

CU Compute Unit, Deutsch: Recheneinheit

PE Processing Element, Deutsch: Ausführendes Element

ICD Installable Client Driver

OpenGL Open Graphics Library

GLSL OpenGL Shading Language

GLEW The OpenG Extension Wrangler Library

NVIDIA NVIDIA Corporation

CUDA Compute Unified Device Architecture

PTX Parallel Thread eXecution assembly language

SM Streaming Multiprocessor, Deutsch: Streaming-Multiprozessor

T&L Transform and Lighting, Deutsch: Umformung und Ausleuchtung

ATI ATI Technologies Inc.

LDS Local Data Share

CTM Close To Metal

IL Intermediate Language, Deutsch: Zwischensprache

CAL Compute Abstraction Layer