TECHNISCHE UNIVERSITÄT DRESDEN

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

Großer Beleg

Untersuchung der Parallelisierung des Feldkamp-Davis-Kress-Algorithmus mittels CUDA®

Jan Stephan
(Mat.-Nr.: 3755136)

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuer: Dr.-Ing. André Bieberle
Dr.-Ing. Guido Juckeland
Matthias Werner

Dresden, 21. März 2017

Inhaltsverzeichnis

Glossar

CUDA® NVIDIA® CUDA®, proprietäre Plattform für die Programmierung von Grafikkarten.

Device Beschleuniger, der einen Kernel ausführt. Im Zusammenhang mit CUDA® ist dieser typischerweise eine GPU.

DirectX® Microsoft® DirectX®, proprietäre Plattform für u.a. Grafikprogrammierung.

Grid Menge aller auf dem Device durch einen Kernel gestarteten Threads.

Host Gerät, das einen Kernel auf dem Device ausführt. Üblicherweise ist dies das Gerät, auf dem auch das Betriebssystem läuft, etwa ein Rechner oder ein Knoten auf einem Superrechner.

Kernel Programm, das auf einem Beschleuniger, wie etwa einer GPU, ausgeführt wird.

Pixel Punkt in einem zweidimensionalen Koordinatensystem, z.B. einem Bild oder einem Detektor.

Race Condition Paralleler, nicht (ausschließlich) lesender Zugriff mehrerer Threads auf das gleiche Datum. Da die Reihenfolge der Schreib- und Lesezugriffe ohne weitere Synchronisierungsmechanismen nicht definiert ist, besteht die Möglichkeit der Datenkorruption.

Stream Warteschlange auf einem CUDA®-Device. Operationen, wie z.B. Kernelaufrufe, werden innerhalb eines Streams sequentiell ausgeführt. Mehrere Streams können vom gleichen Device parallel abgearbeitet werden.

Voxel Punkt in einem dreidimensionalen Koordinatensystem, z.B. einem Volumen.

Abkürzungsverzeichnis

CPU *Central Processing Unit*.

cuFFT NVIDIA® CUDA® *Fast Fourier Transform*.

FDK-Algorithmus Feldkamp-Davis-Kress-Algorithmus.

FPGA *Field Programmable Gate Array*.

GPGPU *General Purpose Computation on Graphics Processing Unit*.

GPU *Graphics Processing Unit*.

HPC *High Performance Computing*.

OpenCL™ *Open Computing Language*.

OpenGL™ *Open Graphics Library*.

SM *Streaming Multiprocessor*.

Abbildungsverzeichnis

Quelltextverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Die Geschichte und Relevanz der Computertomographie

Die Geschichte der Computertomographie beginnt mit dem vom deutschen Physiker Wilhelm Conrad Röntgen entdeckten und später nach ihm benannten Verfahren der „X-Strahlen“ (vgl. [Rö95]). Plötzlich war es möglich, die innere Beschaffenheit eines Objekts auf nichtinvasive Art und Weise zu untersuchen. Die Bedeutung dieses Verfahrens insbesondere für die Anwendung in der Medizin war bereits Röntgens Zeitgenossen klar. So druckte die Wiener Zeitung „Die Presse“ am 05. Januar 1896 auf ihrer Titelseite unter der Überschrift „Eine sensationelle Entdeckung“: „[Man hat es] mit einem in seiner Art epochemachenden Ergebnisse der exacten Forschung zu thun, das sowol [sic] auf physikalischem wie auf medicinischem Gebiete ganz merkwürdige Consequenzen bringen dürfte.“ Für seine Entdeckung wurde Röntgen in der Folge unter anderem mit dem ersten Nobelpreis für Physik ausgezeichnet. Bis heute ist das Röntgenverfahren ein wichtiger Bestandteil der medizinischen Diagnostik und der Werkstoffprüfung. Führt man die Vorwärtsprojektion genügend oft in aufeinanderfolgenden Winkelschritten aus, bis man (idealerweise) einen Vollkreis abgefahren hat, so lässt sich aus den dabei entstandenen *Projektionen* der ursprünglich durchleuchtete Körper, den wir in der Folge als *Volumen* bezeichnen, rekonstruieren. Für jeden Punkt im Volumen (*Voxel*) kann anhand der Informationen aus den Projektionen der Absorptionsgrad berechnet und dadurch die innere Struktur des Volumens bestimmt werden. Dieser Zusammenhang wurde in den 60er Jahren des 20. Jahrhunderts durch den südafrikanisch-amerikanischen Physiker Allan McLeod Cormack festgestellt, der ebenfalls die dazu notwendigen mathematischen Grundlagen entwickelte (vgl. [Cor63] und [Cor64]); ihm war allerdings unbekannt, dass diese schon 1917 vom österreichischen Mathematiker Johann Radon gefunden wurden(vgl. [Cor79]). Mathematisch ist der Vorgang der *Rückprojektion* eine Anwendung der nach Radon benannten *Radon-Transformation* (vgl. [Rad17]).
Da die gefilterte Rückprojektion für jedes Voxel einzeln berechnet werden muss, ist sie für einen Menschen nicht in sinnvoller Zeit lösbar. Aus diesem Grund ist man für die Lösung des Gesamtproblems auf einen Computer angewiesen, woraus sich der Name des Verfahrens ableitet: *Computertomographie*. Die ersten bis zur Marktreife entwickelten Computertomographen wurden gegen Ende der 60er Jahre des 20. Jahrhunderts vom englischen Elektroingenieur Godfrey Hounsfield gebaut. Dieser entwickelte die für die Rückprojektion nötigen Algorithmen ebenfalls selbst, da ihm die Vorarbeiten von Cormack und Radon nicht bekannt waren. Für ihre voneinander unabhängigen Arbeiten erhielten Godfrey und Cormack 1979 den Nobelpreis für Physiologie oder Medizin, was die Bedeutung der Computertomographie insbesondere für die Medizin unterstreicht.

1.2 Aufgabenstellung

Der Feldkamp-Davis-Kress-Algorithmus (FDK-Algorithmus) ist ein weit verbreiteter Ansatz zur Rekonstruktion von kegelförmiger Computer-Tomographie. In diesem Beleg soll untersucht werden:

- Zusammenfassung des Forschungsstandes hinsichtlich der Parallelisierung / der Verwendung von CUDA®
- Gegenüberstellung verschiedener Optimierungsziele (Time-to-solution, Occupancy)
- Variantenvergleich verschiedener Implementierungsstrategien
- Implementierung und Analyse einer dieser Strategien

1.2.1 Forschungsstand

Aufgrund seiner geringen Komplexität und einfachen Implementierbarkeit ist der FDK-Algorithmus einer der beliebtesten Rückprojektionsalgorithmen für die Kegelstrahl-Computertomographie [XM04]. Der Vorteil des FDK-Algorithmus liegt außerdem darin, dass die gefilterte Rückprojektion für jedes Voxel individuell berechnet werden kann, das heißt ohne Abhängigkeiten zu anderen Voxeln. Dieser Umstand ermöglicht für die maschinelle Berechnung den maximalen Grad an Parallelität, der im englischen Sprachraum auch als *embarassingly parallel* bezeichnet wird, und macht den FDK-Algorithmus zu einem idealen Ziel für diverse Parallelisierungsansätze. Einige neuere Ansätze sollen im Folgenden vorgestellt werden.
Seit seiner Einführung ist der FDK-Algorithmus ein beliebtes Untersuchungsobjekt diverser Forschungsgruppen, die sich mit seiner Beschleunigung bzw. Parallelisierung mittels einer großen Variation von Architekturen, Plattformen und Programmiermodellen beschäftigen.
Xu et al. untersuchten bereits 2004, inwieweit sich der FDK-Algorithmus durch den Einsatz handelsüblicher Grafikkarten (*commodity graphics hardware*) beschleunigen lässt [XM04]. Dabei wurden die Schritte *Wichtung* und *Filterung* aufgrund ihrer geringen Komplexität ($\mathcal{O}(n^2)$) auf der *Central Processing Unit* (CPU) ausgeführt, während man die komplexere *Rückprojektion* ($\mathcal{O}(n^4)$) auf der *Graphics Processing Unit* (GPU) berechnete. Die Rückprojektion fand schichtweise statt, jeweils für eine Voxelebene entlang der vertikalen Volumenachse. In ihrem Fazit stellten die Autoren die Vermutung auf, dass der Abstand zwischen den Leistungen von CPUs und GPUs in der Zukunft zugunsten der GPUs immer größer werden würde: *Since GPU performance has so far doubled every 6 months (i.e., triple of Moore's law), we expect that the gap between CPU and GPU approaches will widen even further in the near future.*
Li et al. beschäftigten sich 2005 damit, wie man den FDK-Algorithmus mit einem *Field Programmable Gate Array* (FPGA) implementieren könnte. Dazu teilten sie das Ausgabevolumen, also die Zieldaten der Rückprojektion, in mehrere Würfel (*bricks*) auf, um zu einer optimalen Cachenutzung zu kommen. Der verwendete deterministische Aufteilungsalgorithmus hatte zur Folge, dass bei der Berechnung auf dem FPGA kein Cache-Verfehlen (*cache miss*) mehr auftrat.
Knaup et al. gingen 2007 der Frage nach, ob der FDK-Algorithmus durch die Eigenschaften der Cell-Architektur profitieren könne [KSBK07].
Scherl et al. unternahmen 2008 den Versuch, den FDK-Algorithmus mittels CUDA® zu beschleunigen [SKKH07]. Im Gegensatz zu der Gruppe um Xu et al. führten sie alle Schritte auf der GPU aus und führten die Rückprojektion projektionsweise durch, das heißt, dass jede Projektion einzeln in das Gesamtvolumen zurückprojiziert wurde. Diese Art der Datenverarbeitung ermöglichte es, die Schritte *Wichtung* und *Filterung* parallel zur Rückprojektion auszuführen. Zur Ausnutzung dieser Eigenschaft

und zur besseren Kapselung bzw. Modularisierung der Teilschritte entwickelten die Autoren daher eine Pipeline-Struktur zur parallelen Abarbeitung des Algorithmus, basierend auf dem von Mattson et al. vorgestellten Entwurfsmuster [MSM04].
Balász et al. versuchten 2009 das Gleiche mit der *Open Computing Language* (OpenCL™) [BG09].
Hofmann et al. untersuchten eventuelle Vorteile durch den Einsatz der neuen Koprozessoren vom Typ Intel® Xeon Phi™ ‚Knights Corner' [HTHW14].
Zhao et al. verfolgten die Absicht, eine Beschleunigung durch Ausnutzung geometrischer Zusammenhänge zu erreichen [ZHZ09]. Sie setzten dabei auf die Tatsache, dass ein einmal bestimmtes, also auf den Detektor projiziertes, Voxel durch Rotation in 90°-Schritten die rotierten Voxel ebenfalls genau bestimmt. Ist also für ein Voxel im Projektionswinkel 0° die zugehörige Detektorkoordinate gefunden, so kann diese Detektorkoordinate für die Projektionswinkel 90°, 180° und 270° und die entsprechenden Voxel wiederverwendet werden.

2 Grundlagen

Dieses Kapitel stellt das theoretische Fundament der späteren praktischen Arbeit vor. Zunächst werden die mathematischen und algorithmischen Grundlagen der Computertomographie erläutert, bevor die CUDA®-Plattform als technische Basis eingeführt wird.

2.1 Die Computertomographie

Dieser Abschnitt erläutert die für diese Arbeit relevanten theoretischen Grundlagen der Computertomographie. Es werden zunächst die mathematischen Voraussetzungen gezeigt, darauf aufbauend schließt sich die Beschreibung des FDK-Algorithmus an.

2.1.1 Mathematische Grundlagen der Computertomographie

In diesem Abschnitt werden die mathematischen Grundlagen der Computertomographie behandelt. Es werden zunächst die mathematischen Eigenschaften der Vorwärtsprojektion und das sich daraus ergebende Fourier-Schichten-Theorem erläutert. Anschließend wird das Prinzip der daraus abgeleiteten gefilterte Rückprojektion erläutert.

2.1.1.1 Projektionen

Schießt man einen Röntgenstrahl durch ein festes Objekt, wie beispielsweise biologisches Gewebe oder ein Metall, so wird dieser Strahl je nach Dichte des Materials entlang seiner Bahn abgeschwächt bzw. absorbiert. Mathematisch lässt sich ein Objekt daher als zwei- oder dreidimensionale Verteilung von Absorptionskonstanten verstehen, während die gesamte Abschwächung entlang einer Strahlbahn als Kurvenintegral dargestellt werden kann.
Die Grundlage der folgenden Ausführungen ist die Abbildung 2.1. Als Beispiel dienen ein Objekt, hier durch die Funktion $f(x, y)$ repräsentiert, sowie Kurvenintegrale, hier das Parameterpaar (α, t). Die Linie AB lässt sich dann durch die folgende Formel darstellen:

$$x \cdot \cos\theta + y \cdot \sin\theta = t_1$$

oder allgemein für beliebige, zu AB parallele, Linien:

$$x \cdot \cos\theta + y \cdot \sin\theta = t \tag{2.1}$$

Das zu $f(x, y)$ gehörige Kurvenintegral ist $P_\theta(t)$:

$$P_\theta(t) = \int\limits_{(\theta,t)\text{-Linie}} f(x, y)\,\mathrm{d}s \tag{2.2}$$

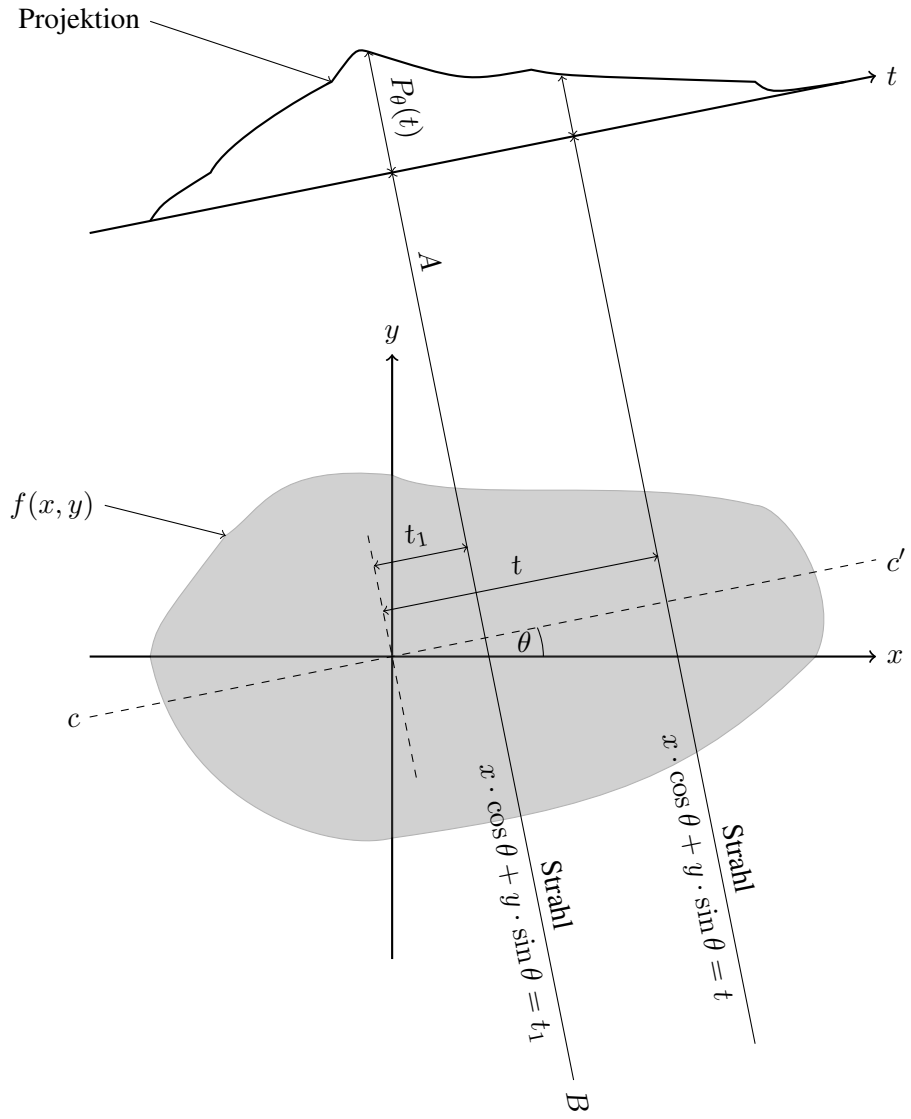

Abbildung 2.1: Zusammenhang zwischen Kurvenintegral und Projektion (Vorlage: [Kak79])

Zusammen mit der aus Formel 2.1 resultierenden Delta-Distribution lässt sich das Kurvenintegral wie folgt umschreiben:

$$P_\theta(t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y) \cdot \delta(x \cdot \cos\theta + y \cdot \sin\theta - t)\, \mathrm{d}x\, \mathrm{d}y \tag{2.3}$$

Die Funktion $P_\theta(t)$ ist die *Radon-Transformation* der Funktion $f(x,y)$ (vgl. [Rad17]).
Eine Projektion lässt sich als die Kombination einer Menge von Radon-Transformationen verstehen. Die (mathematisch) einfachste Projektion ist eine Sammlung von Parallelstrahlintegralen $P_\theta(t)$ unter einem konstanten Winkel θ. Man bezeichnet eine solche Projektion als *Parallelstrahlprojektion* (siehe Abbildung 2.2). In der Praxis kann eine Parallelstrahlprojektion durch die Bewegung einer Quelle-Detektor-Anordnung entlang paralleler Linien auf entgegengesetzten Seiten des Objekts aufgenommen werden.
Eine zweite Aufnahmemöglichkeit ist der Einsatz einer Quelle auf einer festen Position sowie einer Reihe von Detektoren entlang einer Linie auf der anderen Seite des Objekts (siehe Abbildung 2.3). Sol-

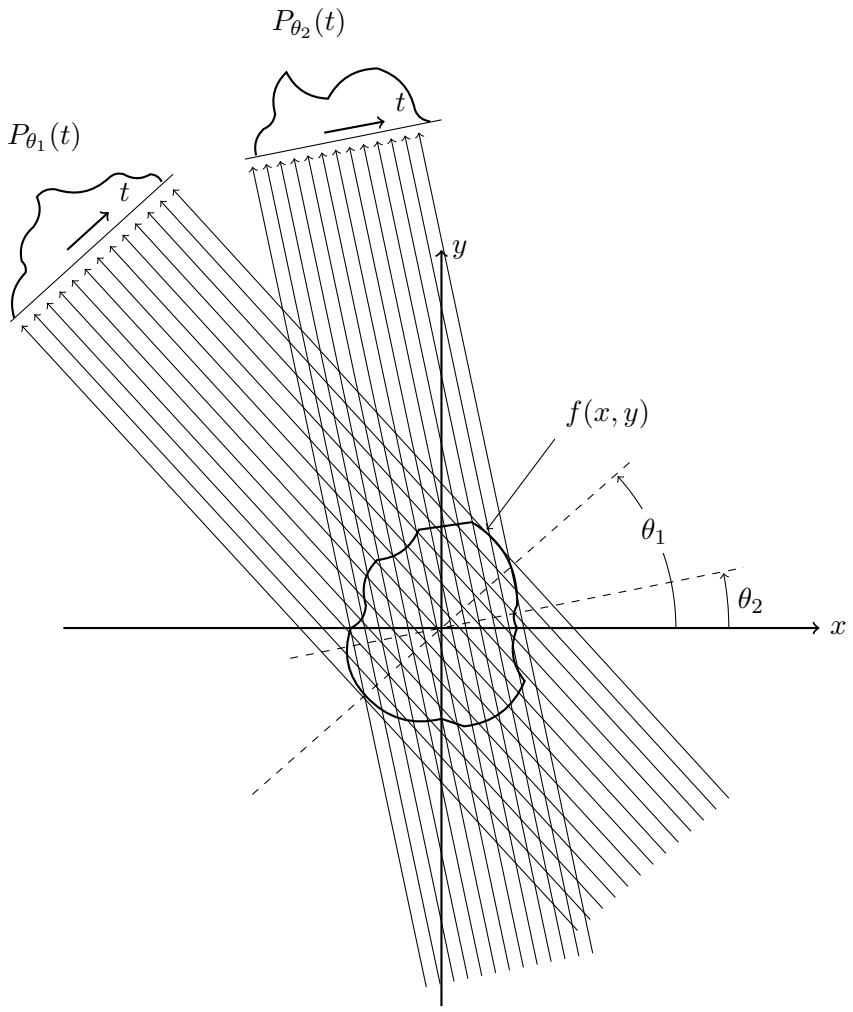

Abbildung 2.2: Parallelstrahlprojektionen (Vorlage: [RK82], S. 356)

cherart erzeugte Projektionen nennt man aufgrund der fächerförmigen Strahlen *Fächerstrahlprojektionen* (vgl. [KS88], S. 49 – 51).

2.1.1.2 Das Fourier-Scheiben-Theorem

Der Zusammenhang zwischen der Projektion und dem Objekt wird noch deutlicher, wenn man erstere einer eindimensionalen und letztere einer zweidimensionalen Fouriertransformation unterzieht. So stellten Rosenfeld und Kak in ihrer Arbeit fest, dass die Fouriertransformation einer Parallelstrahlprojektion eines Objekts $f(x, y)$, die unter dem Winkel θ aufgenommen wurde, einer Schicht des zweidimensional fouriertransformierten Objekts $F(u, v)$, entspricht. Mit anderen Worten ergibt die Fouriertransformation einer Projektion $P_\theta(t)$ die Werte von $F(u, v)$ entlang einer radialen Linie BB (siehe Abbildung 2.4, vgl. [RK82], S.366).
Rosenfeld und Kak haben diesen Zusammenhang auch mathematisch hergeleitet. Seien $F(u, v)$ die zweidimensionale Fouriertransformation des Objekts $f(x, y)$:

$$F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y) \cdot e^{-2\pi i \cdot (ux+vy)} \,\mathrm{d}x\,\mathrm{d}y \tag{2.4}$$

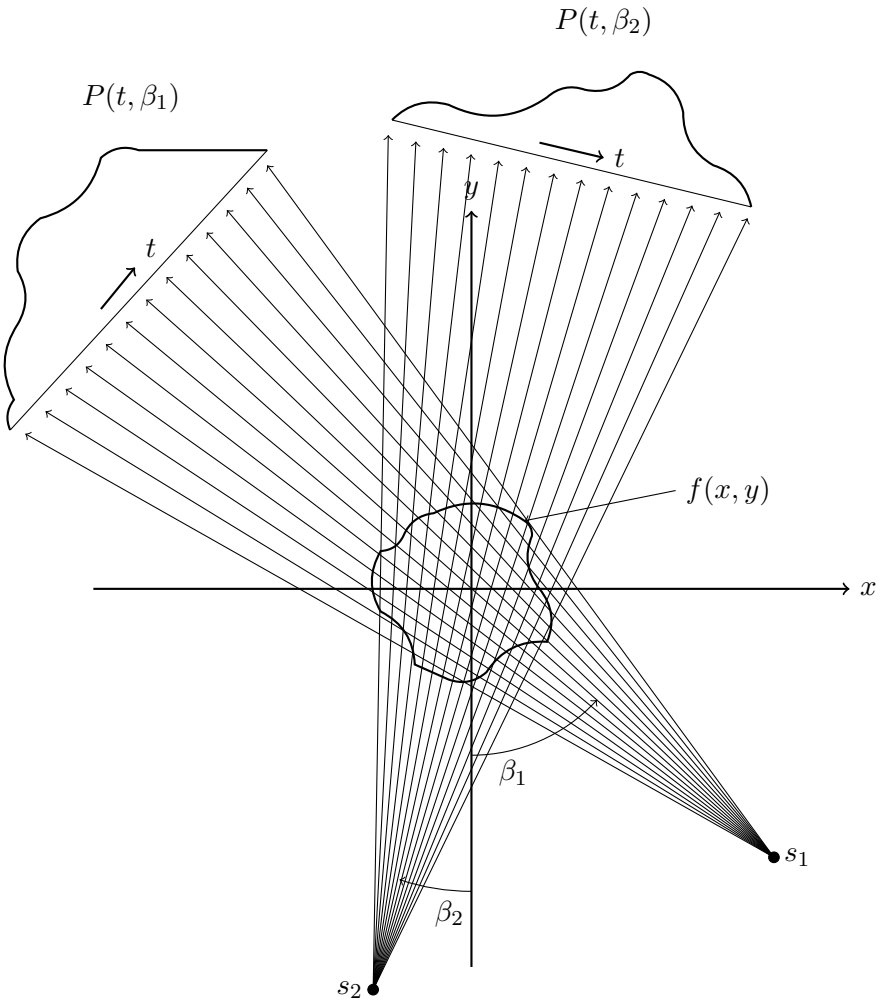

Abbildung 2.3: Fächerstrahlprojektionen (Vorlage: [RK82], S. 357)

und $S_\theta(w)$ die eindimensionale Fouriertransformation der Projektion unter dem Winkel θ $P_\theta(t)$:

$$S_\theta(w) = \int_{-\infty}^{\infty} P_\theta(t) \cdot e^{-2\pi i \cdot wt} \, \mathrm{d}t \tag{2.5}$$

Im Folgenden sei der Fall $\theta = 0$ betrachtet. Setzt man $v = 0$, so wird die Formel 2.4 vereinfacht:

$$F(u, 0) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \cdot e^{-2\pi i \cdot ux} \, \mathrm{d}x \, \mathrm{d}y \tag{2.6}$$

Der Phasenfaktor ist nun nicht mehr von y abhängig, wodurch das Integral in zwei Hälften geteilt werden kann:

$$F(u, 0) = \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} f(x, y) \, \mathrm{d}y \right] \cdot e^{-2\pi i \cdot ux} \, \mathrm{d}x \tag{2.7}$$

Der Term in eckigen Klammern entspricht der Gleichung für eine Projektion entlang konstanter x-Linien:

$$P_{\theta=0}(x) = \int_{-\infty}^{\infty} f(x, y) \, \mathrm{d}y \tag{2.8}$$

Durch Einsetzen in Gleichung 2.7 erhält man:

$$F(u, 0) = \int_{-\infty}^{\infty} P_{\theta=0}(x) \cdot e^{-2\pi i \cdot ux} \, \mathrm{d}x \tag{2.9}$$

Die rechte Seite dieser Gleichung stellt die eindimensional fouriertransformierte Projektion $P_{\theta=0}$ dar (vgl. Gleichung 2.5), es ergibt sich also der folgende Zusammenhang zwischen der transformierten Projektion und dem zweidimensional fouriertransformierten Objekt:

$$F(u, 0) = S_{\theta=0}(u) \tag{2.10}$$

Dieses Ergebnis ist die einfachste Form des Fourier-Scheiben-Theorems. Darüber hinaus ist es unabhängig von der konkreten Konstellation zwischen dem Objekt und dem Koordinatensystem der Projektion. Wird das (t, s)-Koordinatensystem um den Winkel θ rotiert, so ist die Fouriertransformation der Projektion gleich der zweidimensionalen Fouriertransformation des Objekts entlang einer radialen Linie, die um den Winkel θ rotiert wird (siehe Abbildung 2.4, vgl. [RK82], S. 366).
Aus den obigen Ergebnissen folgt, dass sich die Werte von $F(u, v)$ entlang radialer Linien dadurch bestimmen lassen, dass man die Projektionen des Objekts unter mehreren Winkeln $\theta_1, \theta_2, \ldots, \theta_k$ aufnimmt und diese fouriertransformiert. Für eine unendliche Anzahl von Projektionen wäre $F(u, v)$ somit in allen Punkten der uv-Ebene bestimmt, sodass die Objektfunktion $f(x, y)$ durch die inverse Fouriertransformation bestimmt werden kann. In der Praxis können allerdings nur endlich viele Projektionen aufgenommen werden, sodass $F(u, v)$ ebenfalls nur entlang einer endlichen Anzahl radialer Linien bestimmt ist (siehe Abbildung 2.5). Um $f(x, y)$ dennoch näherungsweise bestimmen zu können, müssen die restlichen Punkte anhand der bekannten Werte interpoliert werden, was typischerweise durch eine *nearest neigh-*

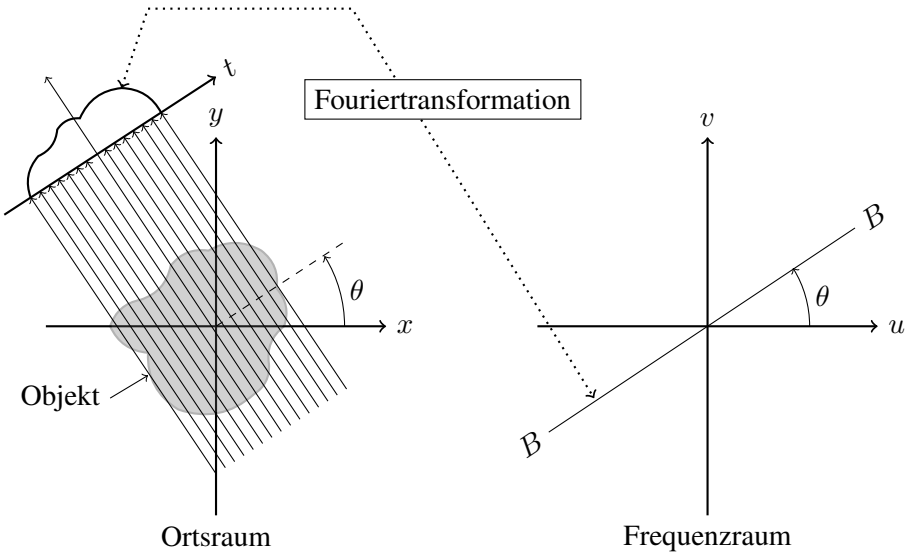

Abbildung 2.4: Das Fourier-Scheiben-Theorem (Vorlage: [KS88], S. 57)

bour oder eine lineare Interpolation geschieht. Da die Dichte der vorhandenen Punkte immer geringer wird, je weiter man sich vom Ursprung entfernt, nimmt der Interpolationsfehler bei größerer Distanz zu. Die Komponenten eines Bildes mit hoher Frequenz sind somit stärker von Fehlern betroffen als die niederfrequenten Anteile, was einen Qualitätsverlust des rekonstruierten Bildes zur Folge hat (vgl. [KS88], S. 59 – 60).

2.1.1.3 Die gefilterte Rückprojektion

Das Fourier-Scheiben-Theorem stellt den Zusammenhang zwischen der fouriertransformierten Projektion und dem fouriertransformierten Objekt entlang einer radialen Linie her. Daraus folgt, dass man mit genügend vielen fouriertransformierten Projektionen, die unter unterschiedlichen Winkeln aufgenommen wurden, eine Annäherung an das zweidimensional fouriertransformierte Objekt und damit (durch die inverse Fouriertransformation) an das Objekt selbst erreichen kann. Dieses einfache Modell der Tomographie lässt sich jedoch nicht ohne Weiteres in dieser Form implementieren (vgl. [KS88], S. 60).
Ein bekannter Ansatz zur Lösung dieses Problems ist die gefilterte Rückprojektion, die sehr genau und schnell zu implementieren ist und sich aus dem Fourier-Scheiben-Theorem herleiten lässt (zur mathematischen Herleitung siehe [KS88], S. 63 – 68). Es ist dieser Schritt, der eine effiziente Implementierung der Rekonstruktion für Computer ermöglicht; Kak und Slaney stellen fest: „The derivation of this algorithm is perhaps one of the most illustrative examples of how we can obtain a radically different computer implementation by simply rewriting the fundamental expressions for the underlying theory“ (siehe [KS88], S. 60).
Die Grundlage der gefilterten Rückprojektion ist der Umstand, dass jede Projektion eine nahezu unabhängige Aufnahme des Objekts darstellt. Die einzige Gemeinsamkeit der unter verschiedenen Winkeln aufgenommenen Projektionen ist die Gleichheit im Punkt $F(0,0)$. Transformiert man eine einzelne Projektion und nimmt an, dass es keine anderen Projektionen gibt, so lässt sich durch die zweidimensionale inverse Fouriertransformation ein (verzerrtes) Objekt rekonstruieren. Durch das Aufsummieren mehrerer transformierter Projektionen nimmt die Qualität des rekonstruierten Objekts zu. Da die Fouriertransfor-

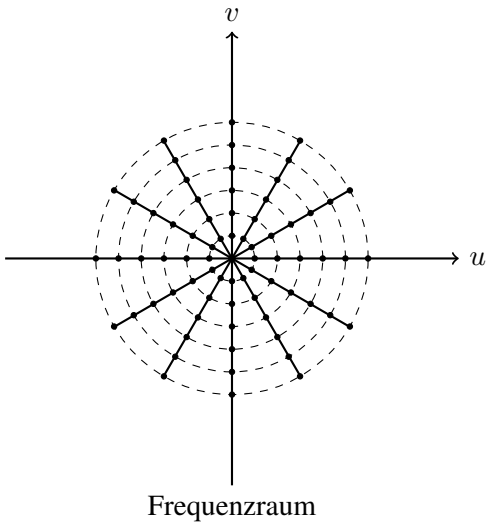

Abbildung 2.5: Radiale Linien, entstanden durch fouriertransformierte Projektionen (Vorlage: [KS88], S. 59)

mation eine lineare Operation ist, lässt sich dieses Aufsummieren der Projektionen auch im Ortsraum durchführen. In diesem Fall spricht man von der Rückprojektion (vgl. [KS88], S. 61).
Die Filterung lässt sich als Wichtung der Projektionen im Frequenzraum betrachten. Wie bereits in Abschnitt 2.1.1.2 erwähnt, nimmt die Anzahl der durch die (in endlicher Zahl vorliegenden) bekannten Punkte im Frequenzraum mit steigender Distanz zum Ursprung ab. Durch die Wichtung der höheren Frequenzen der Projektionen lassen sich die fehlenden Punkte approximieren, durch eine möglichst hohe Projektionszahl lässt sich der durch die fehlenden Informationen hervorgerufene Effekt weiter verringern (vgl. [KS88], S. 61).
Die gewichteten Projektionen lassen sich (nach der inversen Fouriertransformation) aufsummieren, um das Objekt zu rekonstruieren. Zusammengefasst ergibt sich für die gefilterte Rückprojektion der folgende Algorithmus:
Für alle Winkel θ:

1. Projektion $P_\theta(t)$ aufnehmen
2. $P_\theta(t)$ zu $S_\theta(w)$ fouriertransformieren
3. $S_\theta(w)$ filtern
4. $S_\theta(w)$ invers fouriertransformieren und zum bisherigen Ergebnis hinzu addieren

Gegenüber der in Abschnitt 2.1.1.2 erwähnten Interpolationsmethode bietet dieser Algorithmus den Vorteil, dass direkt nach dem Aufnehmen der ersten Projektion begonnen werden kann. Daneben ist es einfacher, im Ortsraum zu interpolieren, als wenn man dies im Frequenzraum täte, da im Ortsraum eine lineare Interpolation häufig genügt (vgl. [KS88], S. 62).

2.1.2 Der Feldkamp-Davis-Kress-Algorithmus

Der 1984 entwickelte FDK-Algorithmus [FDK84] ist eine spezielle Ausprägung der gefilterten Rückprojektion für die Computertomographie mit Kegelstrahlen. In diesem Abschnitt wird zunächst die zugrun-

deliegende Mathematik näher erläutert, bevor die einzelnen Schritte des FDK-Algorithmus detaillierter betrachtet werden.

2.1.2.1 Grundlagen

Die Idee des FDK-Algorithmus ist die Erweiterung der in Abschnitt 2.1.1.3 vorgestellten gefilterten Rückprojektion auf die dritte Dimension. Ausgangspunkt ist eine Fächerstrahlprojektion (siehe Abbildung 2.3), die jetzt im dreidimensionalen Raum betrachtet wird, wie in Abbildung 2.6 dargestellt. Ausgehend von der Schnittgeraden der mittleren Ebene ($z = 0$) mit der Detektorebene, also der Projektion, lassen sich die in der mittleren Ebene liegenden Punkte rekonstruieren. Verschiebt man nun die Schnittgerade (den Fächer) bei gleichbleibender Quellposition derart, dass sie parallel zur Schnittgeraden der mittleren Ebene mit dem Detektor bleibt (konstantes z, $z \neq 0$), so befinden sich die so erfassten Punkte ebenfalls innerhalb einer Ebene. Diese neue Ebene lässt sich ebenfalls als eine mittlere Ebene betrachten, die allerdings im Vergleich zur „ursprünglichen“ mittleren Ebene gekippt ist.
Um die Rückprojektion einer Fächerstrahlprojektion auf die gekippte Ebene anwenden zu können, müssen die durch die Verschiebung entstandenen Veränderungen berücksichtigt werden. Zum Aufnahmewinkel der Projektion kommt nun noch der Winkel zwischen der gekippten Ebene und der mittleren Ebene hinzu, außerdem hat sich für den gekippten Strahl die Distanz zwischen der Quelle und dem Detektor vergrößert. Berechnet man diese Faktoren mit ein, so lässt sich die Rückprojektion von Fächerstrahlprojektionen auch im dreidimensionalen Raum durchführen (zur mathematischen Herleitung siehe [FDK84], S. 614 – 615). Aufgrund der Kegelform der von der Quelle ausgehenden Strahlen bezeichnet man sie auch als *Kegelstrahlen* (englisch *cone-beam*; vgl. den Titel der Arbeit von Feldkamp, Davis und Kress, *Practical cone-beam algorithm*).
Es ist zu beachten, dass der FDK-Algorithmus keine exakte, sondern nur eine näherungsweise Lösung bietet: „No rigorous proof exists [...], since the result is approximate“ (siehe [FDK84], S. 614).

2.1.2.2 Geometrie

Basierend auf den oben genannten Faktoren lässt sich ein dreidimensionales Tomographiesystem wie in Abbildung 2.7 darstellen. Der Ausgangspunkt der Strahlung ist eine Quelle S (*source*), die das Volumen O (*object*) unter einem Drehwinkel α_p mit einem *kegelförmigen* Strahl durchleuchtet und auf einem Detektor mit $N_h \cdot N_v$ Pixeln abbildet. Dabei stellt d_{src} den Abstand zwischen der Quelle und dem Rotationsmittelpunkt, also dem Zentrum des durchleuchteten Volumens, dar, während d_{det} den Abstand zwischen dem Rotationsmittelpunkt und dem Detektor.

2.1.2.3 Wichtung

Aufgrund der Kegelform der Strahlung hat jeder Strahl, der das Volumen durchleuchtet, beim Auftreffen auf den Detektor einen im Vergleich zu den restlichen Strahlen unterschiedlich langen Weg zurückgelegt. Um die durch die Streckenunterschiede bedingten Absorptionsveränderungen auszugleichen, ist es erforderlich, jeden Punkt der Projektion (Pixel) zu wichten. Dafür wird jedes Pixel mit den Koordinaten (j, i) mit dem Wichtungsfaktor w_{ij} multipliziert, der auf den Abständen d_{src} und d_{det} sowie den vertikalen und horizontalen Distanzen des individuellen Pixels vom Ursprung des Detektorkoordinatensystems basiert:

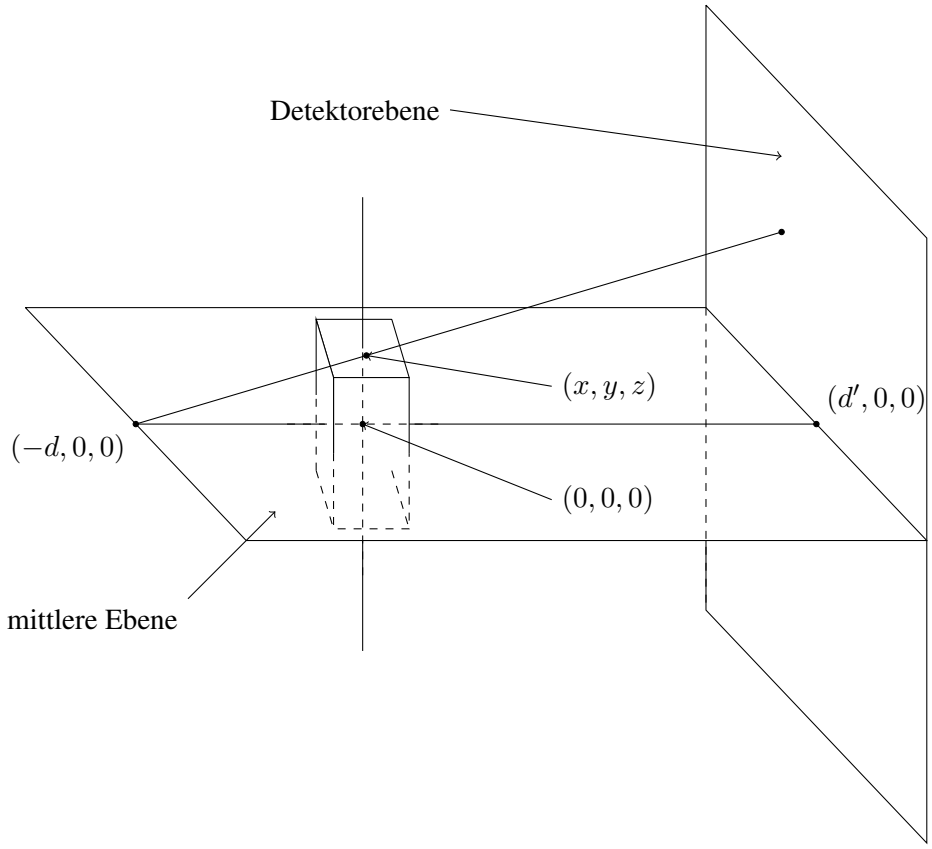

Abbildung 2.6: Schematische Darstellung eines 3D-Tomographiesystems (Vorlage: [FDK84])

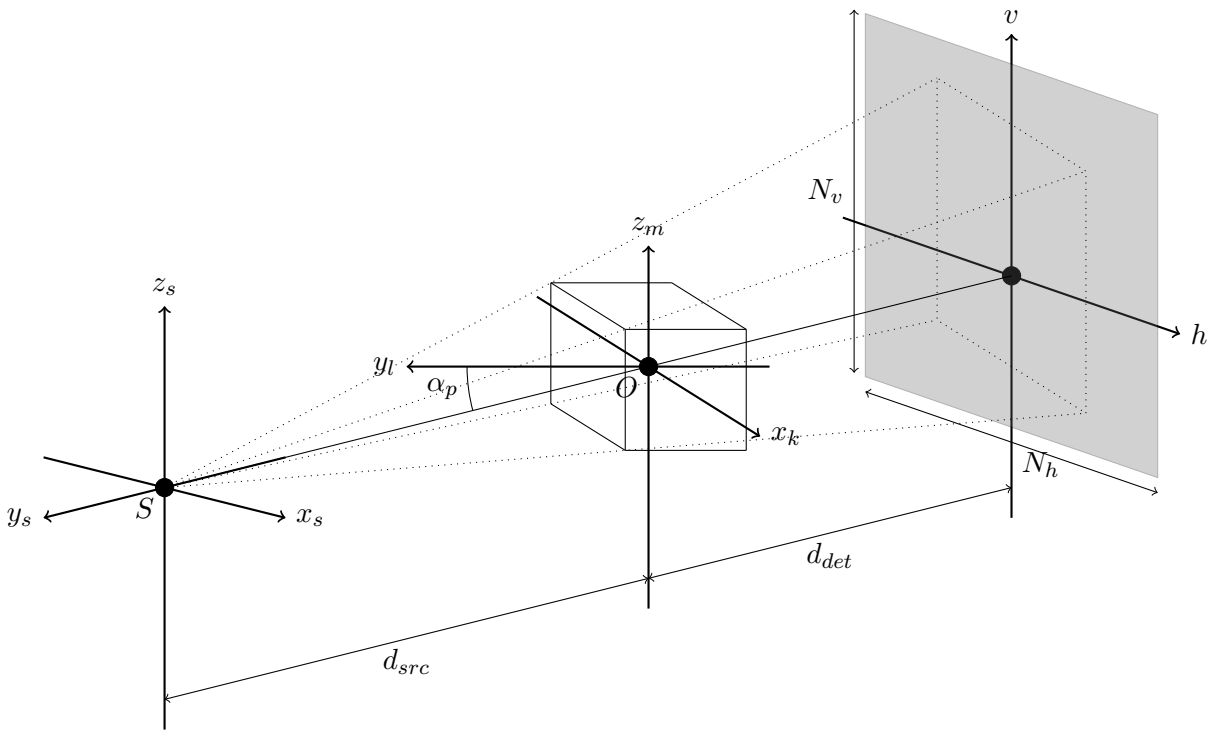

Abbildung 2.7: Geometrie der gefilterten Rückprojektion

$$w_{ij} = \frac{d_{det} - d_{src}}{\sqrt{(d_{det} - d_{src})^2 + h_j^2 + v_i^2}} \tag{2.11}$$

2.1.2.4 Filterung

Aufgrund der in Abschnitt **??** genannten Gründe müssen die gewichteten Projektionen vor der Rückprojektion gefiltert werden. Da es sich bei der Kegelstrahltomographie im Wesentlichen um eine erweiterte Fächerstrahltomographie handelt, genügt es, die Projektionen zeilenweise zu filtern, wie man es bei einem „Stapel“ von Fächerstrahlprojektionen tun würde. Zu diesem Zweck müssen die Projektionen und der zum Einsatz kommende Filter fouriertransformiert werden; zum Einsatz kommt dabei das Verfahren der schnellen Fouriertransformation (*fast Fourier transform*, FFT) nach Cooley und Tukey (vgl. [CT65]). Da dieses Verfahren nur mit einer Menge von Elementen funktioniert, die einer Zweierpotenz entspricht, müssen die Projektionszeilen und der Filter auf die nächste Zweierpotenz „aufgerundet“ werden. Dazu wird, ausgehend von der Länge einer Projektionszeile N_h, die Filterlänge N_{hFFT} berechnet:

$$N_{hFFT} = 2 \cdot 2^{\lceil \log_2 N_h \rceil} \tag{2.12}$$

Mit der so bestimmten Filterlänge lässt sich der Filter r erzeugen:

$$\begin{aligned} r(j) \text{ mit } j &\in \left[-\frac{N_{hFFT}-2}{2}, \frac{N_{hFFT}}{2} \right] \\ r(j) &= \begin{cases} \frac{1}{8} \cdot \frac{1}{\tau^2} & \text{wenn } j = 0 \\ 0 & \text{wenn } j \text{ gerade} \\ -\frac{1}{2j^2\pi^2\tau^2} & \text{wenn } j \text{ ungerade} \end{cases} \end{aligned} \tag{2.13}$$

Nun wird die zu filternde Zeile so lange mit 0 aufgefüllt, bis die erweiterte Zeile N_{hFFT} Pixel umfasst:

$$\begin{aligned} p &: \text{ mit Nullen aufgefüllte Projektionszeile} \\ p(0 \ldots N_{h-1}) &= \text{det}(0 \ldots N_{h-1}) \\ p(N_h \ldots N_{hFFT}) &= 0 \end{aligned} \tag{2.14}$$

Im Anschluss werden sowohl der Filter r als auch die erweiterte Projektionszeile p in den komplexen Raum transformiert und dort miteinander multipliziert:

$$\begin{aligned} R &= \text{FFT}(r) \\ P &= \text{FFT}(p) \\ F &= P \cdot R \quad \text{sowohl für den reellen als auch den imaginären Teil} \end{aligned} \tag{2.15}$$

Die so gefilterte Projektionszeile F wird dann mit der inversen schnellen Fouriertransformation (IFFT) in den reellen Raum zurücktransformiert und von den „aufgefüllten“ Elementen bereinigt:

$$\begin{aligned} f &= \text{IFFT}(F) \\ \text{gefilterte Projektionszeile} &: f(0 \ldots N_{h-1}) \end{aligned} \tag{2.16}$$

2.1.2.5 Rückprojektion

Die gefilterten Projektionen können nun für die Rückprojektion verwendet werden. Dazu werden für jede vorhandene Projektion p, die unter dem Drehwinkel α_p aufgenommen wurde, die folgenden Schritte ausgeführt:

- berechne für jede Voxelkoordinate (x_k, y_l, z_m) deren rotierte Position (s, t, z):

$$\begin{aligned} s &= x_k \cos\alpha_p + y_l \sin\alpha_p \\ t &= -x_k \sin\alpha_p + y_l \cos\alpha_p \\ z &= z_m \end{aligned} \tag{2.17}$$

- projiziere die rotierte Voxelkoordinate (s, t, z) auf den Detektor:

$$\begin{aligned} h' = y' &= t \cdot \frac{d_{det} - d_{src}}{s - d_{src}} \\ v' = z' &= z \cdot \frac{d_{det} - d_{src}}{s - d_{src}} \end{aligned} \tag{2.18}$$

- interpoliere das Detektorsignal bei (h', v'):

$$det' = det(h', v') \tag{2.19}$$

- führe die Rückprojektion für jedes Voxel vol_{klm} aus:

$$\begin{aligned} vol_{klm} &= vol_{klm} + 0,5 \cdot det' \cdot u^2 \\ \text{mit } u &= \frac{d_{src}}{s - d_{src}} \end{aligned} \tag{2.20}$$

Nach dem Abschluss der Rückprojektion erhält man ein Volumen, dessen Voxel Aufschluss über seine innere Struktur geben.

2.2 Die NVIDIA®-CUDA®-Plattform

In diesem Abschnitt wird die NVIDIA®-CUDA®-Plattform näher vorgestellt. Eingegangen wird zunächst auf die historische Entwicklung, die die Einführung von CUDA® begünstigte bzw. erforderte. Im Anschluss daran wird das CUDA®-Programmiermodell vorgestellt, das als technische Basis für die in Kapitel 3 beschriebene Implementierung dient.

2.2.1 Programmierbare Grafikkarten

Als NVIDIA® im Jahre 2006 seine *Compute-Unified-Device-Architecture*-Plattform (CUDA®) vorstellte, die die direkte Programmierung der NVIDIA®-Grafikkarten ermöglichte, folgte die Firma damit einer Entwicklung, die in den ersten Jahren des neuen Jahrtausends begonnen hatte. Mit der Einführung der NVIDIA® GeForce 3 im Jahre 2001 und der parallelen Veröffentlichung von DirectX® 8 bzw. *Open Graphics Library* (OpenGL™) *Vertex-Shader*-Erweiterungen hatten Anwendungsentwickler erstmals Zugriff auf die *Shader*-Einheiten für die *Vertex-* und *Transform-&-Lighting*-Berechnung. Spätere GPUs, die mit DirectX® 9 kompatibel waren, gestatteten eine noch flexiblere Programmierung, indem sie Entwicklern Zugriff auf die *Pixel-Shader* erlaubten und die Nutzung von Texturen im *Vertex-Shader* zuließen. Die 2002 vorgestellte GPU ATI Radeon 9700 verfügte über einen programmierbaren 24bit-Fließkommazahl-*Pixel-Shader*-Prozessor, der mit DirectX® 9 und OpenGL™ gesteuert werden konnte,

die GeForce® FX bot sogar 32bit-Fließkommazahl-Pixel-Prozessoren. Diese programmierbaren Prozessoren waren Teil einer Entwicklung, die zu einer allmählichen Vereinheitlichung der auf der GPU verbauten Funktionseinheiten führte: während es auf den GeForce®-Serien 6800 und 7800 noch getrennte Prozessoren für die *Vertex*- und *Pixel*-Berechnung gab, wurde in der 2005 erschienenen XBox 360 eine „vereinheitlichte“ Grafikkarte verbaut, deren Prozessoreinheiten sowohl für die *Vertex*- als auch für die *Pixel*-Berechnung geeignet waren (vgl. [KH13], S. 28 – 29).
Durch diese Vereinheitlichung der GPU-Prozessoren glich die Hardware mehr und mehr den aus dem *High Performance Computing* (HPC) bekannten Parallelrechnern. Mit der Verfügbarkeit DirectX®-9-kompatibler GPUs wurde diese Entwicklung zunehmend auch in Forschungskreisen bekannt und man begann zu untersuchen, inwiefern sich die neue Hardware zur Lösung von berechnungsintensiven Problemen aus den Bereichen der Natur- und Ingenieurswissenschaften einsetzen ließ. Die zu diesen Zwecken nicht konstruierten GPUs ließen sich jedoch nur über die vorhandenen Schnittstellen zur Grafikprogrammierung ansteuern. Zur Nutzung der Hardwareressourcen musste ein Programmierer daher das zu lösende Problem zunächst auf computergrafische Operationen abbilden, sodass die Berechnung dann mit OpenGL™ oder DirectX® durchgeführt werden konnte. Um beispielsweise eine mathematische Funktion mehrfach auszuführen, musste diese erst in ein *Pixel-Shader*-Programm umgeschrieben werden, während die zugehörigen Eingabedaten als Texturen vorzuliegen hatten und die Ausgabedaten im der jeweiligen Grafikbibliothek eigenen Pixelformat zurückgegeben wurden. Die Umschreibung in einen für die Verwendung von *Pixel-Shadern* geeigneten Algorithmus hatte außerdem den gravierenden Nachteil, dass Zugriffe auf beliebige Stellen im Speicher nicht möglich waren. Da ein *Pixel-Shader* als Ausgabedatum die Farbe eines Pixels zurückliefert, besteht dazu aus Sicht der Grafikbibliothek auch keine Notwendigkeit, da die Position des zugehörigen Pixels ja bereits bekannt ist (vgl. [KH13], S. 33).
Diese Beschränkungen erwiesen sich für generische numerische Berechnungen bald als zu restriktiv. Erschwerend kam hinzu, dass jeder Entwickler, der sich mit diesen technischen Limitierungen abfinden konnte, zusätzlich noch Wissen über die Funktionsweise von OpenGL™ und DirectX® benötigte, um das zu lösende Problem mit Hilfe von GPUs bewältigen zu lassen. Eine flächendeckende Akzeptanz von GPUs als Beschleunigern war unter Forschern daher in den ersten Jahren des *General Purpose Computation on Graphics Processing Unit* (GPGPU) nicht gegeben (vgl. [SK11], S. 6).
Die Situation änderte sich, als NVIDIA® 2006 die GeForce® 8800 GTX der Öffentlichkeit präsentierte. Diese GPU war nicht nur zum damals neuen DirectX® 10 kompatibel, sondern auch die erste Grafikkarte, die mit CUDA® ohne den Umweg über DirectX® oder OpenGL™ direkt programmierbar war (vgl. [SK11], S. 7). Programmierer hatten nun die Möglichkeit, *datenparallele* Aspekte (vgl. Abschnitt 2.2.2.1) des zu lösenden Problems zu deklarieren und von der GPU ausführen zu lassen. Die *Shader*-Prozessoren hatten sich zu komplett programmierbaren Prozessoren entwickelt und verfügten über einen Instruktionsspeicher, einen Instruktions-Cache und eine Instruktionskontrollogik. Diesen zusätzlichen Hardwareoverhead konnte NVIDIA® dadurch reduzieren, dass mehrere Prozessoren sich den Instruktionscache und die -kontrolllogik teilten. Diese Struktur funktioniert aufgrund des Hardware-Fokus auf die parallele Berechnung von Pixeln für GPUs gut. Zusätzlich war es nun möglich, auf beliebige Teile des Speichers zuzugreifen. Aus Sicht des Entwicklers lag nun ein Programmiermodell vor, das eine Hierarchie paralleler Threads, Synchronisierungsmechanismen und Barrieren sowie atomare Operationen bot und das Ganze in eine an C bzw. C++ angelehnte Sprache einbettete (vgl. [KH13], S. 35).

2.2.2 Das CUDA®-Programmiermodell

In diesem Abschnitt wird das der CUDA®-Plattform zugrundeliegende Programmiermodell näher vorgestellt. Das gedankliche Fundament dieses Modell ist die *Datenparallelität*, die in Abschnitt 2.2.2.1 erläutert wird. Die konkreten Programmierkonzepte schließen sich in den danach folgenden Abschnitten an.

2.2.2.1 Datenparallelität

Moderne Programme bearbeiten häufig große Datenmengen und benötigen deswegen auf herkömmlichen Rechnern lange Ausführungszeiten. Viele Anwendungen arbeiten dabei mit Daten, die Vorgänge der echten Welt ab- oder nachbilden, wie beispielsweise einfache Bilder oder Filme oder die Bewegungen von Flüssigkeiten und Gasen unter bestimmten Umständen. Fluglinien müssen ihre Flüge planen, wozu die Daten einer Vielzahl anderer Flüge, Besatzungen und Flughäfen herangezogen werden, die voneinander unabhängig sind. Diese Unabhängigkeit der Daten ist die Voraussetzung für datenparallele Algorithmen.
Als einfaches Beispiel für einen datenparallelen Algorithmus lässt sich die Addition zweier Vektoren A und B betrachten:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

In diesem Beispiel wird c_1 berechnet, indem a_1 und b_1 addiert werden, und c_3 ergibt sich aus der Addition von a_3 und b_3. Die Additionen sind voneinander unabhängig und können parallel berechnet werden. Sehr große Vektoren führen in diesem Beispiel daher zu einem hohen Maß an Datenparallelität.
Der Datenparallelität steht das Konzept der *Task-Parallelität* gegenüber. Die Task-Parallelität kommt vor allem in komplexeren Programmen zur Anwendung, in denen es mehrere unterschiedliche Aufgaben gibt, die parallel bearbeitet werden können. Vorstellbar ist z.B. ein Programm zur Filterung von Bildern, in denen ein Task das die Bilder nacheinander lädt, der zweite die Bilder nacheinander filtert und ein weiterer die gefilterten Bilder wieder abspeichert (vgl. [KH13], S. 42 – 43).

2.2.2.2 Parallele Ausführung

Die Ausführung eines CUDA®-Programms beginnt auf dem Host. Wird ein Kernel aufgerufen, so wird auf dem Device eine Anzahl an Threads gestartet. Die Summe aller auf dem Device gestrateten Threads bezeichnet man als *Grid*. Haben alle Threads eines Kernels ihre Ausführung beendet, wird auch das Grid beendet. Das Starten eines Kernels erzeugt typischerweise eine große Zahl von Threads. Im Gegensatz zur CPU, auf der das Erzeugen eines Threads in der Regel mehrere tausend Takte benötigt, ist der gleiche Vorgang auf der GPU in einigen wenigen Takten zu bewältigen (vgl. [KH13], S. 44 – 45).
Das Beispiel der Vektoraddition aus Abschnitt 2.2.2.1 lässt sich in C++ wie in Quelltext 2.1 dargestellt implementieren.

```
auto vec_add(const std::int32_t* A, const std::int32_t* B,
             std::int32_t* C, std::size_t size) -> void
{
    for(auto i = 0u; i < size; ++i)
        C[i] = A[i] + B[i];
}
```

Quelltext 2.1: Vektoraddition mit C++

In CUDA® würde dagegen jeder Thread ein Element des Ausgabevektors C berechnen, wie in Quelltext 2.2 gezeigt.

```
__global__ void vec_add(const std::int32_t* A, const std::int32_t* B,
                        std::int32_t* C, std::size_t size)
{
    auto i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < size)
        C[i] = A[i] + B[i];
}
```

Quelltext 2.2: Vektoraddition mit CUDA®

Die Berechnung der Indices erfolgt in diesem Beispiel nicht über eine fortlaufend inkrementierte Variable, wie es in C++ der Fall wäre, sondern über die Hilfsvariablen `blockIdx`, `blockDim` und `threadIdx`. Wird ein CUDA®-Kernel ausgeführt, so entsteht auf dem Device ein Thread-Grid. Diese Threads werden in einer zweistufigen Hierarchie angeordnet. Jedes Grid besteht aus mehreren Thread-Blöcken, die der Einfachheit halber in der Folge als *Blöcke* bezeichnet werden. Alle Blöcke innerhalb eines Grids haben die gleiche Größe, können jedoch maximal 1024 Threads umfassen; die Thread-Zahl der Blöcke wird auf dem Host vor der Ausführung des Kernels definiert. Für ein gegebenes Grid kann die Anzahl der Threads pro Block durch die Variable `blockDim` abgefragt werden, während der individuelle Block, zu dem ein bestimmter Thread gehört, durch die Variable `blockIdx` identifiert werden kann. Innerhalb eines Blocks kann ein einzelner Thread über die Variable `threadIdx` bestimmt werden. Für den ersten Thread des Blocks `0` hat `threadIdx` also den Wert `0`, für den zweiten `1`, für den dritten `2`, usw. Durch die Kombination der drei Variablen, wie sie in Quelltext 2.2 dargestellt ist, lässt sich somit jeder Thread innerhalb des Grids eindeutig identifizieren. Da CUDA® das Starten von zwei- und dreidimensionalen Kerneln unterstützt, kann über die Felder `x`, `y` und `z` die Zahl der Blöcke bzw. Threads in der jeweiligen Dimension bestimmt werden (vgl. [KH13], S. 54).

Ein weiterer Unterschied zur herkömmlichen C++-Programmierung ist das Schlüsselwort `__global__` vor der Deklaration. Dieses Schlüsselwort dient zur Identifizierung einer Funktion als Kernel, also einer Funktion auf dem Device, die vom Host aus aufgerufen werden kann. Daneben verwendet CUDA® die Schlüsselworte `__device__` und `__host__`, die anzeigen, auf welcher Hardware die jeweilige Funktion ausgeführt werden kann bzw. soll. Eine `__device__`-Funktion kann dabei nur innerhalb eines Kernels oder einer weiteren `__device__`-Funktion aufgerufen werden, eine `__host__`-Funktion nur auf dem Host – sie entspricht also einer normalen C++-Funktion (vgl. Tabelle 2.1 und [KH13], S.55).

	Ausgeführt auf dem	Aufgerufen vom
`__device__ float device_func()`	Device	Device
`__global__ void kernel_func()`	Device	Host
`__host__ float host_func()`	Host	Host

Tabelle 2.1: CUDA®-Schlüsselworte für die Funktionsdeklaration

Bezeichnung	Ort	Sichtbarkeit	Zugriff
Register	On-Chip	pro Thread	lesen / schreiben
`__shared__`	On-Chip	pro Block	lesen / schreiben
`__global__`	Off-Chip	pro Kernel	lesen / schreiben
`__constant__`	Off-Chip	pro Kernel	nur lesen
`__local__`	Off-Chip	pro Thread	lesen / schreiben
Textur	Off-Chip	pro Kernel	nur lesen

Tabelle 2.2: Die verschiedenen Varianten des CUDA®-Speichers

2.2.2.3 Architektur

2.2.2.4 Speicher

Wie auf einem klassischen Rechner mit seinem Hauptspeicher, diversen Caches und Prozessorregistern gibt es auch auf einer CUDA®-fähigen GPU verschiedene Möglichkeiten zur Zwischenspeicherung von Daten.
Die verschiedenen Speichervarianten sind in Tabelle 2.2 zusammengefasst.

2.2.2.5 Streams

3 Umsetzung

In diesem Kapitel wird eine mögliche parallelisierte Implementierung des FDK-Algorithmus vorgestellt. Dazu wird im ersten Abschnitt ein Variantenvergleich vorgenommen, an den sich im zweiten Abschnitt die konkrete Implementierung anschließt.

3.1 Variantenvergleich

Dieser Abschnitt befasst sich mit dem Variantenvergleich verschiedener denkbarer Parallelisierungsstrategien. Zunächst werden einige Ansätze aus der Literatur besprochen, die im Hinblick auf die Parallelisierung mit GPUs von besonderem Interesse sind, sowie auf deren Schwächen eingegangen. Danach wird das allen denkbaren Ansätzen gemeine Problem des begrenzten GPU-Speichers erläutert, woran sich dessen Abschwächung durch den Einsatz mehrerer GPUs anschließt.

3.1.1 Bestehende Parallelisierungsstrategien und ihre Grenzen

Von den in Abschnitt **??** genannten Ansätzen in der Literatur sind aufgrund ihrer Umsetzung für GPUs die Strategien von Xu et al. (vgl. [XM04]), Scherl et al. (vgl. [SKKH07]) und Zhao et al. (vgl. [ZHZ09]) von besonderem Interesse für diese Arbeit.
Da Xu et al. 2004 mit ihrer Arbeit Neuland betraten, standen ihnen viele Methoden und Technologien, die seitdem entwickelt wurden, noch nicht zur Verfügung. Die 2004 erschienenen GPUs hatten im Vergleich zu heutigen Grafikkarten sehr viel weniger Speicher; das damals beste verfügbare Produkt von NVIDIA®, die GeForce® 6800 Ultra, konnte lediglich mit 512 MiB Speicher aufwarten und war nur über OpenGL™ oder DirectX® indirekt programmierbar (vgl. [NVIa]). Den begrenzten Speicher versuchte die Gruppe durch eine Aufteilung des Volumens und eine schichtweise Rekonstruktion desselben unter Einbeziehung aller Projektionen möglichst effizient zu nutzen (vgl. [XM04]). Aufgrund des technischen Fortschritts stehen uns heute andere Möglichkeiten zur Lösung dieses Problems offen; so bietet etwa die NVIDIA® GeForce® GTX 1080 mit 8 GiB Speicher und der Möglichkeit der direkten Programmierung mittels CUDA® oder OpenCL™ ganz andere Nutzungs- und Berechnungsmöglichkeiten als ihre frühen Vorgänger (vgl. [NVIb]). Insbesondere ist es möglich, das ganze Volumen oder größere Teile davon während der Berechnung im Speicher zu halten und dadurch häufige Kopien zwischen CPU-Speicher und GPU-Speicher zu vermeiden.
Die Forschungsgruppe um Scherl baute auf der Idee, das Volumen im Speicher zu halten, auf und ging im Gegensatz zu Xu et al. den Weg, jede Projektion einzeln in dieses Volumen zu projizieren (vgl. [SKKH07]). Zur Trennung bzw. Kapselung der einzelnen Schritte entwickelten sie in einem vorherigen Schritt (vgl. [SHKH08]) eine Pipeline-Struktur (nach Mattson et al., vgl. [MSM04]). Jeder Schritt des FDK-Algorithmus wird dabei in einer eigenen Stufe (*stage*) ausgeführt, die in einem separaten Thread ausgeführt wird. Zur Kommunikation der Ergebnisse der einzelnen Stufen werden thread-sichere

Puffer verwendet, auf die die Eingabe- bzw. Ausgaberoutinen der Stufen zugreifen. Die eigentliche Rekonstruktion erfolgt entlang der z-Achse und der y-Achse: Jede z-Schicht im Volumen wird entlang der x-Achse nocheinmal in Spalten aufgeteilt. Ein zweidimensionaler Kernel iteriert dann über alle Voxel in y-Richtung (vgl. Abbildung 3.1). Der Nachteil des von Scherl et al. gewählten Ansatzes liegt in der Pipeline-Struktur, die durch die wesentlich längere Laufzeit der Rückprojektion nicht optimal ist. So schreiben Mattson et al.: „If the stages in the pipeline vary widely in computational effort, the slowest stage creates a bottleneck for the aggregate throughput“ ([MSM04], S. 106).
Die Grenzen bei dem vorgeschlagenen Verfahren der Gruppe um Zhao et al. sind vor allem praktischer Natur. Das von ihnen vorgestellte Modell sieht vor, Symmetrien auszunutzen und dadurch Rechenzeit einzusparen (vgl. Abbildung 3.2). Sie machen sich dabei den Umstand zunutze, dass die auf den Detektor projizierte Koordinate eines Voxels der um 90° rotierten Projektionskoordinate des um den gleichen Betrag rotierten Voxels entspricht. Auf diese Weise lassen sich durch eine Berechnung die Detektorkoordinaten von vier Voxeln finden, was eine Verkürzung der Rechenzeit verspricht (vgl. [ZHZ09]). In der Praxis scheitert dieses Verfahren an dem mechanischen Aufbau üblicher CT-Scanner. Da entweder Quelle und Detektor oder aber das Untersuchungsobjekt rotiert werden müssen, kommt es durch Fehler in der Mechanik häufig dazu, dass Aufnahmen doppelt gemacht oder übersprungen werden; auch kann es passieren, dass die Winkelschritte zwischen zwei Aufnahmen nicht immer einheitlich sind.

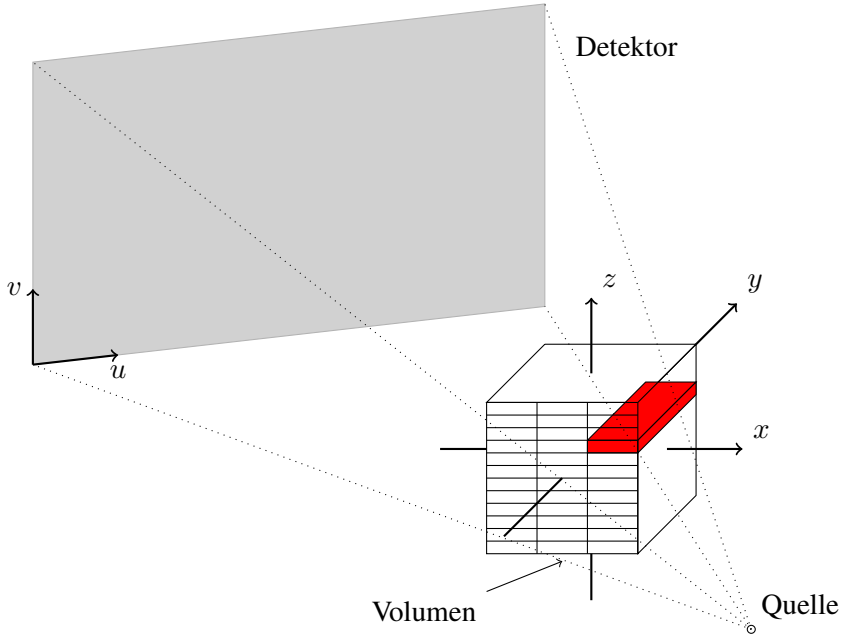

Abbildung 3.1: Aufteilung des Volumens nach Scherl et al. (Vorlage: [SKKH07])

3.1.2 Das Problem des GPU-Speichers

Den oben vorgestellten Ansätzen ist gemein, dass sie alle mit dem limitierten Speicher einer GPU arbeiten mussten. Während der technische Fortschritt die Speichergrenze seitdem weiter nach oben verschoben hat, hat sich das zugrundeliegende Problem dennoch nicht wesentlich verändert. Mit modernen GPUs wie der NVIDIA® GeForce® GTX 1080 ist es inzwischen zwar möglich, komplette Volumen oder wenigstens große Teile davon während der Rückprojektion im GPU-Speicher zu halten und so die Zahl der Kopien zwischen Host und Device zu reduzieren; der parallel stattfindende Fortschritt bei der

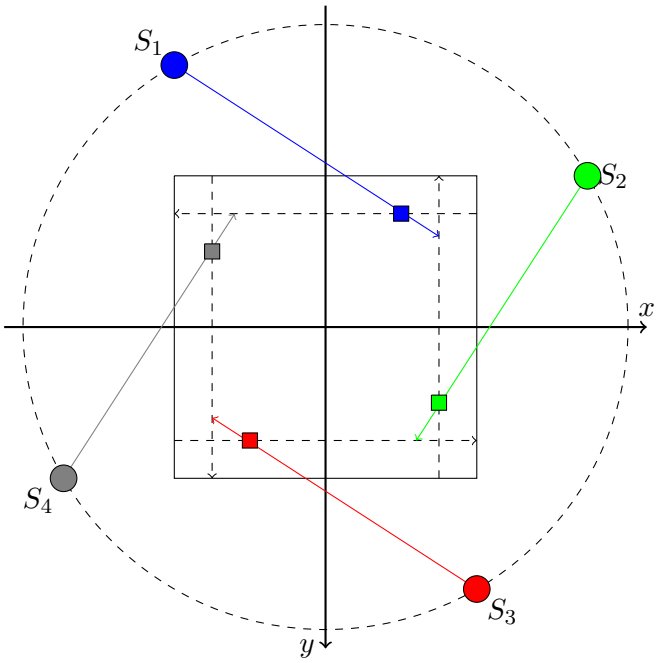

Abbildung 3.2: Symmetrien bei der Rückprojektion nach Zhao et al. (Vorlage: [ZHZ09])

Computertomographie, insbesondere im Hinblick auf die Detektorauflösung und die dadurch produzierte Datenmenge, erhöht allerdings auch weiterhin den für die Berechnung erforderlichen Speicher. Im Gegensatz zu den klassischen CPUs, deren Speicher sich theoretisch nahezu unbegrenzt erhöhen lässt, ist für die Parallelisierung mit GPUs eine Strategie zur Verwaltung des zur Verfügung stehenden Speichers also weiterhin unabdingbar.

3.1.3 Heterogene GPU-Systeme und effiziente Arbeitsteilung

Ein naheliegender Weg zur Umgehung des begrenzten GPU-Speichers ist der parallele Einsatz mehrerer GPUs. Dieser Ansatz wirft jedoch einige neue Probleme auf.
Am vordringlichsten ist die Frage zu beantworten, in welcher Form die Berechnung auf die verfügbaren GPUs verteilt werden soll. Während dies für den Fall, dass mehrere gleichartige GPUs verwendet werden sollen, relativ einfach ist, wird die Lastverteilung beim Einsatz unterschiedlich leistungsstarker Grafikkarten deutlich schwieriger. Es ist zu vermeiden, dass die schnelleren GPUs auf die langsameren warten müssen, da so Rechenzeit verschenkt wird.
In beiden Fällen kommt hinzu, dass die vorhandenen Grafikkarten möglichst parallel mit Daten befüllt und zur Berechnung gebracht werden sollen, die vorgesehenen Schritte der Rückprojektion also weitestgehend gleichzeitig ausgeführt werden.

3.1.4 Lösungsvorschlag

Aus den oben genannten Faktoren beim Einsatz von GPUs ergeben sich die folgenden Anforderungen an eine effiziente Parallelisierungsstrategie des FDK-Algorithmus:

1. der auf GPUs beschränkte Speicher muss möglichst effizient genutzt werden, häufige Kopien sollten also vermieden werden.

2. Die Optimierung der Rückprojektionen sollte die höchste Priorität genießen, woraus folgt, dass
3. die Rückprojektion die zur Verfügung stehenden GPUs möglichst stark auslasten sollte.
4. Die Wartezeit zwischen zwei Rückprojektionen sollte möglichst minimal sein, also nicht auf die Wichtung und Filterung warten müssen.
5. Die Parallelisierung sollte sich möglichst effizient über mehrere GPUs skalieren lassen.

Mit der Umsetzung der Punkte 2. und 3. beschäftigt sich detailliert der nächste Abschnitt, ein Lösungsvorschlag für die restlichen Punkte findet sich in der im Folgenden skizzierten Programmstruktur.
Ein typischer Projektionsdatensatz umfasst 1440 Projektionen, die jeweils eine Auflösung von 1024 x 1024 Pixeln haben; jeder Pixel ist dabei im 4-Byte-Fließkommazahlen-Format gespeichert. Für den kompletten Datensatz ergibt sich dadurch eine Datenmenge von 5,625 GiB. Das daraus resultierende Volumen besitzt (abhängig von den konkreten Geometrieparametern der CT-Anlage) etwa 1070 x 1070 x 1033 Voxel (vgl. Abschnitt 3.2.4), die das gleiche Datenformat wie die Pixel aufweisen. Die Datengröße des Volumens beträgt 4,4 GiB und die kumulierte Datenmenge damit 10,25 GiB. Ausgehend von der Annahme, dass eine GPU des Typs GeForce® GTX 1080 zur Verfügung steht, ist allein mit den zu verarbeitenden Daten der verfügbare GPU-Speicher nicht ausreichend, da die GTX 1080 über lediglich 8 GiB Speicher verfügt (siehe [NVIb]). Sowohl die Projektionen als auch das komplette Volumen im Speicher zu halten ist demnach kein praktikabler Ansatz. In Abschnitt 3.1.1 wurden mehrere Ansätze der Literatur vorgestellt, die dieses Problem zu umgehen versuchen.
Der erste Ansatz von Xu et al. sieht eine Aufteilung des Volumens in mehrere Segmente vor. Jedes Teilvolumen wird dann durch die Iteration über den kompletten Projektionsdatensatz berechnet (vgl. [XM04]). Allerdings muss bei diesem Ansatz zusätzlich zur Aufteilung des Volumens ein Weg gefunden werden, alle Projektionen effizient auf der GPU abzubilden, da – wie oben gezeigt – der Projektionsdatensatz tendenziell größer ist als das komplette Volumen. Dies erzeugt zusätzliche Komplexität und damit unerwünschten *overhead*.
Scherl et al. schlagen den umgekehrten Weg vor, sodass das ganze Volumen bzw. ein möglichst großer Teil davon im Speicher gehalten wird. Die Projektionen werden dann einzeln geladen und in das Volumen zurückprojiziert (vgl. [SKKH07]). Gegenüber der von Xu et al. gewählten Variante bietet dies den Vorteil, dass nicht mehr alle Projektionen im Speicher gehalten werden bzw. im richtigen Iterationsschritt vorhanden sein müssen; stattdessen werden die Projektionen sequentiell geladen und verarbeitet. Im idealen Fall (das ganze Volumen passt in den Speicher) entfällt zudem die Notwendigkeit für Teilvolumen, sodass es genau einen Kopiervorgang bezüglich des Volumens am Ende der Berechnung gibt.
Für den Fall, dass das Volumen nicht in Gänze in den GPU-Speicher passt, schlagen Zhao et al. eine Aufteilung entlang der z-Achse vor (vgl. [ZHZ09]). Das Volumen wird dabei in N gleich große Teilvolumen zerlegt, die einzeln in den GPU-Speicher passen. Jede Projektion wird dann ebenfalls in N Teilprojektionen unterteilt, sodass jede Teilprojektion geometrisch einem Teilvolumen entspricht. Dieser Ansatz hat jedoch den Nachteil, dass die für das aktuell zu rekonstruierende Teilvolumen nicht benötigten Teilprojektionen verwaltet bzw. zwischengespeichert werden müssen, was ebenfalls zusätzliche Komplexität in den Algorithmus einführt.
Aufgrund der geschilderten Vorteile im Hinblick auf den begrenzt verfügbaren GPU-Speicher ist es daher sinnvoll, dem Ansatz von Scherl et al. zu folgen und die Rückprojektion sequentiell für jede Pro-

jektion durchzuführen, während das ganze Volumen im Speicher gehalten wird. Sollte der vorhandene Speicher nicht ausreichen, muss das Volumen in mehrere Teilvolumen zerlegt werden, die dann einzeln rekonstruiert und zum Schluss wieder zusammengesetzt werden. Abweichend von Zhao et al. ist eine entsprechende Unterteilung der Projektionen aus zweierlei Gründen nicht notwendig. Einerseits besitzt eine einzelne Projektion des obigen Beispiels eine im Vergleich zum Volumen vernachlässigbare Datengröße von 4 MiB, sodass der durch eine Aufteilung eingesparte Speicher keinen nennenswerten Mehrwert bietet, andererseits ist der Zeitaufwand der Wichtungs- und Filteroperationen verglichen mit der Rückprojektion derart gering, dass die Durchführung dieser Operationen auf weniger Pixeln in Bezug auf die Gesamtlaufzeit keine bemerkenswerte Zeitersparnis verspricht. Es ist hierbei zu beachten, dass dadurch alle Projektionen für jedes Teilvolumen neu geladen werden müssen, sobald die Berechnung des vorherigen Teilvolumens abgeschlossen ist. Bei einer Datenübertragung über das Netzwerk oder einer langsamen Festplatte kann sich dieser Faktor schnell zu einem Flaschenhals entwickeln, sofern der (gegebenenfalls gewichtete und gefilterte) Projektionsdatensatz nicht auf andere Art und Weise zwischengespeichert werden kann.

Eine weitere Herausforderung ist die Minimierung der Wartezeit zwischen zwei aufeinanderfolgenden Rückprojektionen. Theoretisch wäre es hier denkbar, zwei oder mehr Rückprojektionen verschiedener Projektionen gleichzeitig durchzuführen. Praktisch ergäbe sich daraus die Notwendigkeit der Synchronisierung zwischen den verschiedenen Rückprojektionsoperationen, um Datenkorruption durch *race conditions* zu vermeiden. Es ist daher der einfachere Weg, die Rückprojektionen in sequentieller Reihenfolge durchzuführen. Um die Wartezeiten zwischen zwei Rückprojektionen zu minimieren, sollten möglichst keine weiteren Operationen in der Zwischenzeit ausgeführt werden. Idealerweise werden dann – neben dem der Rückprojektion – nur noch das Entnehmen der aktuellen Projektion vom Eingabepuffer sowie das Übergeben dieser Projektion an den Rückprojektions-Kernel ausgeführt. Demzufolge muss die beschriebene Ausführungskette in einem eigenen Thread ausgeführt werden, um die restlichen Operationen, insbesondere die Wichtung und die Filterung der Projektionen, durch Nebenläufigkeiten zu kaschieren.

Schwieriger ist eine effiziente Skalierung des FDK-Algorithmus auf heterogenen GPU-Systemen, also Systemen mit unterschiedlich leistungsstarken Grafikkarten. In der Literatur gibt es in jüngerer Zeit Ansätze zur besseren (nicht FDK-spezifischen) Lastverteilung durch den Einsatz von *machine-learning*-Techniken (vgl. [LHCH14]), eine Implementierung eines solchen Verfahrens geht allerdings über den Fokus dieser Arbeit hinaus und bedarf daher einer gesonderten Untersuchung. Für den Fall, dass der implementierte Algorithmus auf einem System mit mehreren GPUs zum Einsatz kommt, wird daher die zu erledigende Arbeit statisch über alle verfügbaren GPUs verteilt. Zunächst wird das Gesamtvolumen in N gleich große Teilvolumen zerlegt, wobei N die Zahl der im System vorhandenen GPUs bezeichnet. Ist die Zahl der Schichten im Gesamtvolumen nicht glatt durch N teilbar, wird der Rest an das unterste Teilvolumen angehängt. Im Anschluss daran wird überprüft, ob diese Teilvolumen gemeinsam mit ein wenig zusätzlichem Puffer für Projektions- und Berechnungsdaten in den kleinsten vorhandenen GPU-Speicher passen. Ist dies nicht der Fall, werden alle Teilvolumen so lange durch zwei geteilt, bis die so entstandenen kleineren Teilvolumen klein genug für den Speicher sind. Für jede vorhandene Grafikkarte wird dann ein separater Thread gestartet, der die Schritte *Projektion laden – Wichtung – Filterung – Rückprojektion* auf dieser GPU ausführt (siehe Quelltext 3.1). Hierbei ist zu beachten, dass aufgrund der oben geschilderten Vorgehensweise zur Reduktion der Wartezeiten die Rückprojektion nochmals in

einem weiteren Thread läuft, sodass die Anzahl der insgesamt gestarteten Threads $2 \cdot N$ beträgt.

```
// Diese Funktion läuft in einem separaten Thread pro GPU
auto fdk(task_queue& queue, // enthält ausstehende Teilvolumen
         int device) -> void
{
    cudaSetDevice(device);

    while(!queue.empty()) // solange es unbearbeitete Teilvolumen gibt
    {
        auto t = queue.pop(); // hole Teilvolumen-Informationen
        auto s = source(...); // konfiguriere Laden der Projektionen

        auto v = make_volume(...); // erzeuge Teilvolumen auf Device

        while(!source.drained()) // für jede Projektion
        {
            auto p = source.load_next(); // lade nächste Projektion
            auto d_p = copy_h2d(p); // kopiere Projektion auf Device
            weight(d_p, ...); // wichte Projektion
            filter(d_p, ...); // filtere Projektion

            /* Rückprojektion. Startet intern einen weiteren Thread,
               der die übergebenen Projektionen weiter verarbeitet */
            backproject(d_p, v, ...);
        }

        save_subvolume(v, ...); // Teilvolumen abspeichern
    }
}
```

Quelltext 3.1: Funktionsabfolge des implementierten FDK-Algorithmus

3.2 Implementierung und Optimierung

In diesem Abschnitt werden die Implementierung und die Optimierung des FDK-Algorithmus beschrieben. Zunächst werden die Implementierungs- und Optimierungsziele sowie die Einflüsse der realen Welt auf das Modell vorgestellt und im Anschluss daran die Umsetzung der Stufen *Wichtung* und *Filterung* gezeigt. Der Abschnitt schließt mit der Implementierung der Rückprojektion.

3.2.1 Implementierungs- und Optimierungsziele

Von den in Abschnitt 3.1.4 genannten Zielen einer Implementierung des FDK-Algorithmus sind im Hinblick auf die Nutzung von CUDA® die zusammenhängenden Faktoren *Optimierung* und *GPU-Auslastung* von großer Bedeutung. Da die Rückprojektion im Vergleich zur Wichtung und Filterung einen weitaus höheren Rechenaufwand erfordert, ist ihrer Optimierung die höchste Priorität einzuräu-

men, sodass der gesamte Algorithmus für derzeit gängige Datensätze der Computertomographie – also bis zu einer Projektionsauflösung von 1024 x 1024 Pixeln – auf neuer Hardware in wenigen Minuten lösbar ist. Da die CUDA®-Plattform den parallelen Einsatz mehrerer GPUs unterstützt, ist die Implementierung so zu konzipieren, dass sie auf einer oder mehreren Grafikkarten korrekte Ergebnisse bei gleichzeitiger Beschleunigung des Gesamtalgorithmus berechnet.

3.2.2 Optimierungsüberlegungen

Die Optimierung eines Kernels fußt im Wesentlichen auf zwei Dingen:

1. Die Auslastung der Grafikkarte muss möglichst hoch sein, um die vorhandenen Rechenkapazitäten effektiv nutzen zu können. Bedingung für eine effektive Verteilung des Kernels ist ein möglichst geringer Registerverbrauch innerhalb eines *Streaming Multiprocessor* (SM)s, sodass die maximale Zahl an Kernen genutzt werden kann.

2. Die langsamen Zugriffe auf den globalen Speicher müssen möglichst effizient sein, um den Algorithmus nicht auszubremsen.

Da, wie wir gesehen haben, die gefilterte Rückprojektion von einer Reihe geometrischer Parameter abhängt, ist die Zahl der benötigten Register für die Übergabe dieser Parameter sowie zur Speicherung von Zwischenergebnissen potentiell hoch. Ein großer Teil dieser Parameter ist jedoch für die (teils komplette, teils projektionsweise) Rückprojektion konstant, wie zum Beispiel die Abstände zwischen dem Objekt, der Quelle und dem Detektor oder die geometrischen Eigenschaften des Detektors. Da sich diese Parameter auch während der Berechnung nicht ändern können, empfiehlt es sich, diese in den konstanten Speicher der Grafikkarte auszulagern, auf den verhältnismäßig schnell zugegriffen werden kann.
Anders verhält es sich mit den Projektionen und dem Volumen. Da während der Ausführung auf beide Speicherbereiche schreibend zugegriffen werden muss, ist es sinnvoll, diese Zugriffe möglichst optimal zu gestalten. Da die Projektionen der Kegelstrahltomographie üblicherweise zweidimensional sind und das zu rekonstruierende Volumen dreidimensional ist, können die für zwei- bzw. dreidimensionale Operationen optimierten Speichervarianten der CUDA®-Plattform verwendet werden, die man mit `cudaMallocPitch` bzw. `cudaMalloc3D` alloziiert. Zusätzlich ist es nach der Wichtung und der Filterung für die Rückprojektion nicht mehr erforderlich, die Projektionen zu verändern. Durch deren Umwandlung in Texturen bzw. deren Verschiebung in den für Texturzugriffe optimierten Teil des globalen Speichers, der einen speziellen Cache besitzt, kann der Zeitverlust beim lesenden Zugriff auf die Projektionen minimiert werden.

3.2.3 Einflüsse der realen Welt

Die in Kapitel 2 gezeigten Überlegungen zur gefilterten Rückprojektion und dem FDK-Algorithmus sind in dieser Form rein theoretischer Natur. Die Anwendung dieser Modelle auf die reale Welt ist mit einigen Schwierigkeiten bzw. Einflüssen verbunden, die im Folgenden näher vorgestellt werden sollen.

3.2.3.1 Detektorgeometrie

Der Detektor übt aufgrund der durch ihn gewonnenen Informationen (in Form der Projektionen) einen großen Einfluss auf die gefilterte Rückprojektion aus. Seinem Aufbau muss daher bei der Implementie-

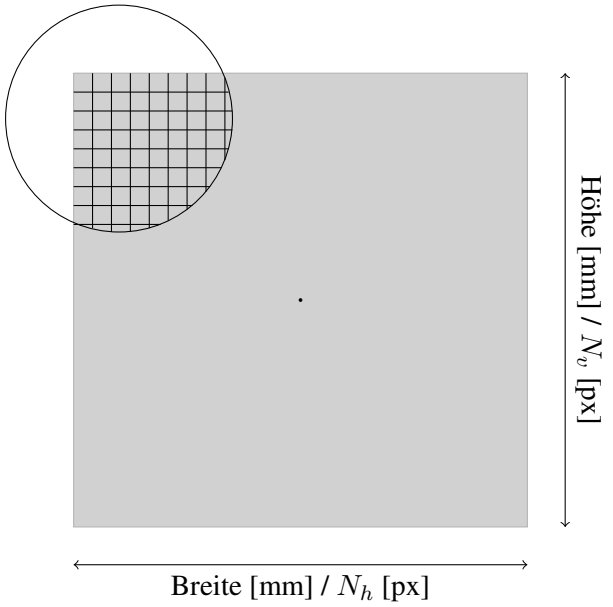

Abbildung 3.3: Detektorgeometrie

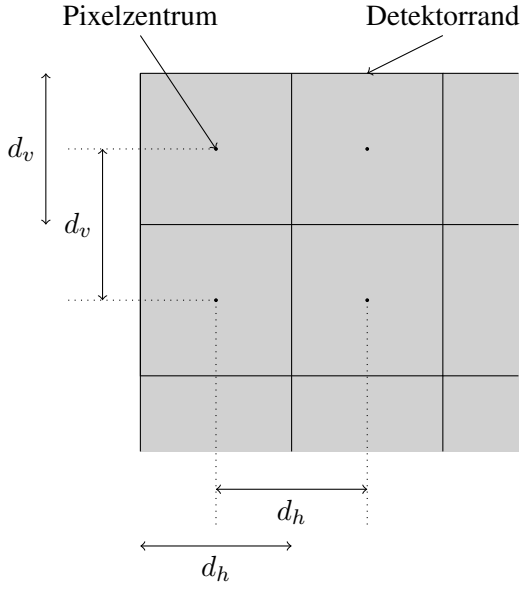

Abbildung 3.4: Pixelgeometrie

rung des FDK-Algorithmus besondere Aufmerksamkeit zukommen.
Der Detektor hat eine feste Breite und Höhe (gemessen in Millimetern) und besteht aus einer zweidimensionalen Anordnung von Pixeln (siehe Abbildung 3.3). In der horizontalen Richtung verfügt er über N_h Pixel, in der vertikalen Richtung sind es N_v Pixel.
Jedes Pixel hat eine physische Breite d_h und Höhe d_v (gemessen in Millimetern); äquivalent lassen sich diese Ausdehnungen als horizontale (vertikale) Abstände zwischen den Pixelzentren betrachten (siehe Abbildung 3.4).
Spannt man nun über dem Detektor ein Koordinatensystem auf (mit dessen Zentrum als Ursprung, siehe Abbildung 3.5), wobei h_{min}, h_{max}, v_{min} und v_{max} den horizontalen (vertikalen) Abstand von den Detektorrändern bis zur Detektormitte in Millimetern angeben, so ergeben sich die folgenden Zusammenhänge:

$$\begin{aligned} h_{max} - h_{min} &= N_h \cdot d_h \\ h_{min} + \frac{N_h \cdot d_h}{2} &= 0 \end{aligned} \tag{3.1}$$

$$\begin{aligned} v_{max} - v_{min} &= N_v \cdot d_v \\ v_{min} + \frac{N_v \cdot d_v}{2} &= 0 \end{aligned} \tag{3.2}$$

3.2.3.2 Verschiebungen

In einem idealen Modell sind die Strahlungsquelle und der Detektor genau aufeinander ausgerichtet, das heißt, dass der Mittelpunkt der Strahlungsquelle und der Mittelpunkt des Detektors auf derselben Achse liegen. Durch den mechanischen Aufbau einer realen Computertomographie-Anlage und deren händischer Justierung kommt es allerdings zu sowohl einer horizontalen Verschiebung Δh als auch einer vertikalen Verschiebung Δv dieser Achse (siehe Abbildung 3.6). Von der Strahlungsquelle ausgehend

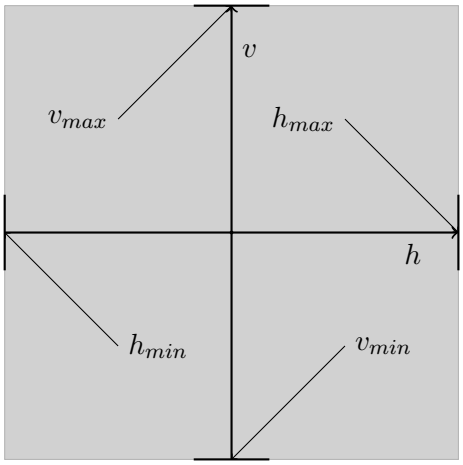

Abbildung 3.5: Detektorkoordinatensystem

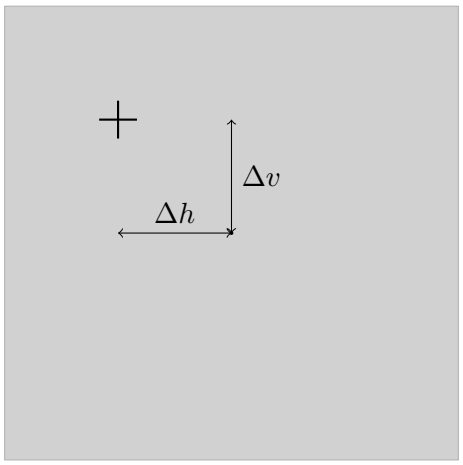

Abbildung 3.6: Verschiebungsgeometrie

trifft sie somit nicht mehr auf das Zentrum des Detektors, sondern auf einen anderen Teil. Nimmt man Bezug auf die Detektorgeometrie, so müssen diese Verschiebungen entsprechend berücksichtigt werden, da ansonsten ein verfälschtes Ergebnis berechnet wird. Dazu müssen die Formeln 3.1 und 3.2 wie folgt umgeschrieben werden:

$$h_{min} + \frac{N_h \cdot d_h}{2} + \Delta h = 0 \tag{3.3}$$

$$v_{min} + \frac{N_v \cdot d_v}{2} + \Delta v = 0 \tag{3.4}$$

3.2.3.3 Fehlende Projektionen

Es leuchtet ein, dass Projektionen, die im Abstand von 180° aufgenommen wurden, das durchleuchtete Objekt spiegelverkehrt darstellen. In der Theorie würde es also ausreichen, einen Halbkreis um das Objekt abzufahren, um alle erforderlichen Informationen für die Rückprojektion zu gewinnen. In der Praxis kann es aufgrund mechanischer Fehler bei der Rotation des Quelle-Detektor-Aufbaus allerdings dazu kommen, dass einzelne Projektionen übersprungen werden oder die Winkelabstände zwischen zwei Projektionen verschieden groß sind. Das Abfahren eines Vollkreises dient dazu, die so entstandenen Fehler durch Redundanzen zu minimieren.

3.2.4 Geometrische Berechnungen

- Berechnung der Volumengeometrie
- Aufteilung in Teilvolumen

3.2.5 Implementierung der Vorstufen

3.2.5.1 Wichtung

Die Grundlage der Wichtungsoperation ist die in Abschnitt 2.1.2.3 vorgestellte Formel 2.11:

$$w_{ij} = \frac{d_{det} - d_{src}}{\sqrt{(d_{det} - d_{src})^2 + h_j^2 + v_i^2}}$$

Es ist leicht zu sehen, dass sich der Wichtungsfaktor w_{ij} zwar pro Pixel ändert, aber nicht von der konkreten Projektion abhängig ist. Es ist daher möglich, die Berechnung der Wichtungsfaktoren am Anfang des Programms genau einmal durchzuführen und in einer Wichtungsmatrix `m` zu speichern (siehe Quelltext 3.2). Die Berechnung der Wichtungsmatrix hängt von mehreren geometrischen Parametern ab (vgl. Abschnitt 2.1.2.2 und Abbildungen 3.3, 3.4, 3.5, 3.6):

- `dim_x`: Anzahl der Pixel in horizontaler Richtung. Entspricht der Anzahl der Detektorpixel in horizontaler Richtung N_h
- `dim_y`: Anzahl der Pixel in vertikaler Richtung. Entspricht der Anzahl der Detektorpixel in vertikaler Richtung N_v
- `h_min`: horizontaler Abstand vom Detektorrand zum Detektorzentrum in mm.
- `v_min`: vertikaler Abstand vom Detektorrand zum Detektorzentrum in mm.
- `d_sd`: Abstand von der Quelle zum Detektor. Entsprich der Differenz der Strecken d_{det} (Abstand zwischen dem Objekt und dem Detektor) und d_{src} (Abstand zwischen der Quelle und dem Objekt) bzw. der Summe ihrer Beträge:

  $$d_{det} - d_{src} = |d_{det}| + |d_{src}|$$

- `l_px_row`: horizontale Länge eines Pixels, also der horizontale Abstand zwischen den Mittelpunkten zweier aufeinanderfolgender Pixel. Entspricht der horizontalen Länge eines Detektorpixels d_h.
- `l_px_col`: vertikale Länge eines Pixels, also der vertikale Abstand zwischen den Mittelpunkten zweier aufeinanderfolgender Pixel. Entspricht der vertikalen Länge eines Detektorpixels d_v.

```
__global__ void matrix_generation_kernel(float* m,
    std::uint32_t dim_x, std::uint32_t dim_y, std::size_t pitch,
    float h_min, float v_min, float d_sd, float l_px_row,
    float l_px_col)
{
    auto s = blockIdx.x * blockDim.x + threadIdx.x;
    auto t = blockIdx.y * blockDim.y + threadIdx.y;

    if((s < dim_x) && (t < dim_y))
    {
        auto row = reinterpret_cast<float*>(
            reinterpret_cast<char*>(m) + t * pitch);

        // Detektorkoordinaten in mm
        const auto h_s = (l_px_row / 2.f) + s * l_px_row + h_min;
        const auto v_t = (l_px_col / 2.f) + t * l_px_col + v_min;

        // berechne Wichtungsfaktor
        row[s] = d_sd * rsqrtf(d_sd * d_sd + h_s * h_s + v_t * v_t);
    }
}
```

Quelltext 3.2: Generierung der Wichtungsmatrix

Bei der Wichtung einer Projektion `p` kann der jeweilige Wichtungsfaktor aus der generierten Matrix `m` ausgelesen und auf das zugehörige Pixel angewendet werden (siehe Quelltext 3.3). Die so gewichtete Projektion wird dann im folgenden Schritt gefiltert.

```
__global__ void weighting_kernel(float* p, const float* m,
    std::uint32_t dim_x, std::uint32_t dim_y, std::size_t pitch,
    std::size_t m_pitch)
{
    auto s = blockIdx.x * blockDim.x + threadIdx.x;
    auto t = blockIdx.y * blockDim.y + threadIdx.y;

    if((s < dim_x) && (t < dim_y))
    {
        auto p_row = reinterpret_cast<float*>(
            reinterpret_cast<char*>(p) + t * pitch);
        auto m_row = reinterpret_cast<const float*>(
            reinterpret_cast<const char*>(m) + t * m_pitch);

        // Wichtung
        p_row[s] *= m_row[s];
    }
}
```

Quelltext 3.3: Wichtung einer Projektion

3.2.5.2 Filterung

Dem in Abschnitt 2.1.2.4 vorgestellten Algorithmus entsprechend, folgt die Implementierung des Filterschrittes dem nachstehenden Schema:

1. einmalige Erzeugung und Fouriertransformation des Filters
2. zeilenweise Fouriertransformation der Projektion
3. Anwendung des Filters auf die jeweilige Projektionszeile im komplexen Raum
4. inverse zeilenweise Fouriertransformation der Projektion

Die Implementierung der Filtergenerierung entspricht der Formel 2.13 und kann dem im Anhang befindlichen Quelltext B.1 entnommen werden. Dieser Filter wird dann zeilenweise auf jede Projektion angewendet. Dazu werden der Filter und die einzelnen Projektionszeilen mit der NVIDIA® CUDA® *Fast Fourier Transform* (cuFFT)-Bibliothek zunächst fouriertransformiert. Im komplexen Raum werden dann die einzelnen Elemente der transformierten Projektionszeile mit den korrespondierenden Elementen des transformierten Filters multipliziert (siehe Quelltext 3.4). Ist dieser Vorgang abgeschlossen, wird die Projektion wieder zurücktransformiert und normalisiert (siehe den angehängten Quelltext B.2). Die Projektion ist dann bereit für die Rückprojektion.

```
__global__ void filter_application_kernel(
    cufftComplex* __restrict__ data,
    const cufftComplex* __restrict__ filter,
    std::uint32_t filter_size, std::uint32_t data_height,
    std::size_t pitch)
{
    auto x = blockIdx.x * blockDim.x + threadIdx.x;
    auto y = blockIdx.y * blockDim.y + threadIdx.y;

    if((x < filter_size) && (y < data_height))
    {
        auto row = reinterpret_cast<cufftComplex*>(
            reinterpret_cast<char*>(data) + y * pitch);

        row[x].x *= filter[x].x;
        row[x].y *= filter[x].y;
    }
}
```

Quelltext 3.4: Filterung einer Projektion

3.2.6 Implementierung der gefilterten Rückprojektion

Die Implementierung der gefilterten Rückprojektion soll möglichst optimal auf der GPU laufen. Um dies zu erreichen,

- welche Konstanten und Variablen gibt es
- welche Schwierigkeiten können auftreten

Die Rückprojektion erfolgt für jede Projektion p einzeln und für ein auf dem Device erstelltes dreidimensionales Volumen (vgl. Abschnitt 2.2.2.4). Der Implementierung (siehe Quelltext 3.5) liegt dabei die in Abbildung 2.7 gezeigte Geometrie zugrunde. Für jedes Voxel im Volumen müssen zunächst die zugehörigen Projektionskoordinaten unter dem Winkel α_p bestimmt werden.

```
__global__ void backprojection_kernel(
    float* vol, std::size_t vol_pitch, cudaTextureObject_t proj,
    float angle_sin, float angle_cos)
{
    auto k = blockIdx.x * blockDim.x + threadIdx.x;
    auto l = blockIdx.y * blockDim.y + threadIdx.y;
    auto m = blockIdx.z * blockDim.z + threadIdx.z;

    if((k < consts.vol_dim_x) && (l < consts.vol_dim_y) &&
       (m < consts.vol_dim_z)) {
        // berechne gegenwärtige Schicht und Zeile
        auto slice_pitch = vol_pitch * consts.vol_dim_y;
        auto slice = reinterpret_cast<char*>(vol) + m * slice_pitch;
        auto row = reinterpret_cast<float*>(slice + l * vol_pitch);

        // lade alten Wert aus dem globalen Speicher
        auto old_val = row[k];

        // Koordinatensystemursprung in Volumenmittelpunkt verschieben
        auto x_k = vol_centered_coordinate(k, consts.vol_dim_x,
                                           consts.l_vx_x);
        auto y_l = vol_centered_coordinate(l, consts.vol_dim_y,
                                           consts.l_vx_y);
        auto z_m = vol_centered_coordinate(m, consts.vol_dim_z,
                                           consts.l_vx_z);

        // Koordinaten rotieren
        auto s = x_k * angle_cos + y_l * angle_sin;
        auto t = -x_k * angle_sin + y_l * angle_cos;

        // projiziere rotierte Koordinaten auf Detektor
        auto factor = consts.d_sd / (s + consts.d_so);
        auto h = proj_pixel_coordinate(t * factor, consts.proj_dim_x,
             consts.l_px_x, consts.delta_s) + 0.5f;
        auto v = proj_pixel_coordinate(z_m * factor,
             consts.proj_dim_y, consts.l_px_y, consts.delta_t) + 0.5f;

        // lies Projektionswert an dieser Stelle
        auto det = tex2D<float>(proj, h, v);

        // Rückprojektion
        auto u = -(consts.d_so / (s + consts.d_so));
        row[k] = old_val + 0.5f * det * u * u;
    }
}
```

Quelltext 3.5: Rückprojektion

4 Analyse

4.1 Leistungsmessungen

4.1.1 Übersicht

Messungen des Gesamtprogramms (Datentransfers, alle Stufen)
grobe Messungen der Teilstufen (Wichtung, Filterung, Rückprojektion)
Ergebnis: Rückprojektion braucht am längsten

4.1.2 Rückprojektion im Detail

detaillierte Messungen der Rückprojektion (Registerverbrauch, GPU-Auslastung, etc)

4.1.3 Vergleich mit der Literatur

5 Fazit

- Faktencheck: Wurden die in der Einleitung genannten Ziele erreicht?

5.1 Zusammenfassung

5.2 Ausblick

Einer gesonderten Untersuchung bedarf, wie in Abschnitt 3.1.4 geschildert, das Potential einer effizienteren Lastverteilung durch den Einsatz von *machine-learning*-Techniken.

Literaturverzeichnis

[BG09] BALÁZS, D. ; GÁBOR, J.: A programming model for GPU-based parallel Computing with scalability and abstraction. In: *SCCG '09 Proceedings of the 25th Spring Conference on Computer Graphics* Spring Conference on Computer Graphics, 2009, S. 103–111

[Cor63] CORMACK, A. M.: Representation of a Function by Its Line Integrals, with Some Radiological Applications. In: *Journal of Applied Physics* 34 (1963), S. 2722

[Cor64] CORMACK, A. M.: Representation of a Function by Its Line Integrals, with Some Radiological Applications. II. In: *Journal of Applied Physics* 35 (1964), S. 2908

[Cor79] CORMACK, A. M.: *Early Two-Dimensional Reconstruction and Recent Topics Stemming from It (Nobel Lecture)*. Dezember 1979

[CT65] COOLEY, J. W. ; TUKEY, J. W.: An Algorithm for the Machine Calculation of Complex Fourier Series. In: *Mathematics of Computation* 19 (1965), April, Nr. 90, S. 297–301

[FDK84] FELDKAMP, L. A. ; DAVIS, L. C. ; KRESS, J. W.: Practical cone-beam algorithm. In: *Journal of the Optical Society of America A* 1 (1984), Februar, Nr. 6, S. 612–619

[HTHW14] HOFMANN, J. ; TREIBIG, J. ; HAGER, G. ; WELLEIN, G.: Performance Engineering for a Medical Imaging Application of the Intel Xeon Phi Accelerator. In: *2014 Workshop Proceedings* International Conference on Architecture of Computing Systems, 2014

[Kak79] KAK, A. C.: Computerized tomography with x-ray emission and ultrasound sources. In: *Proceedings of the IEEE* Bd. 67, 1979, S. 1245–1272

[KH13] KIRK, D. B. ; HWU, W. W.: *Programming Massively Parallel Processors*. 2nd Ed. Morgan Kaufmann, 2013. – ISBN 978–0–124–15992–1

[KS88] KAK, A. C. ; SLANEY, M.: *Principles of Computerized Tomographic Imaging*. IEEE Press, 1988. – ISBN 978–0–879–42198–4

[KSBK07] KNAUP, M. ; STECKMANN, S. ; BOCKENBACH, O. ; KACHELRIESS, M.: Tomographic image reconstruction using the cell broadband engine (CBE) general purpose hardware. In: *Proceedings Electronic Imaging, Computational Imaging V* Bd. 6498 SPIE, 2007, S. 1–10

[LHCH14] LIN, C. ; HSIEH, C. ; CHANG, H. ; HSIUNG, P.: Efficient Workload Balancing on Heterogeneous GPUs using Mixed-Integer Non-Linear Programming. In: *Journal of applied research and technology* 12 (2014), Nr. 6, S. 1176–1186. – ISSN 1665–6423

[MSM04] MATTSON, T. G. ; SANDERS, B. ; MASSINGILL, B.: *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004. – ISBN 978–0–321–22811–6

[NVIa] NVIDIA®: *Enthusiast*. https://www.nvidia.com/page/geforce_6800.html, . – Online; zuletzt abgerufen am 02. März 2017

[NVIb] NVIDIA®: *GeForce GTX 1080 Graphics Card*. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/, . – Online; zuletzt abgerufen am 02. März 2017

[Rad17] RADON, J.: Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten. In: *Berichte über die Verhandlungen der Königlich Sächsischen Gesellschaft der Wissenschaften zu Leipzig, Mathematisch-physische Klasse* Bd. 69 Königlich Sächsische Gesellschaft der Wissenschaften zu Leipzig, 1917, S. 262–277

[RK82] ROSENFELD, A. ; KAK, A. C.: *Digital Image Processing*. Bd. 1. 2nd Ed. Academic Press, 1982. – ISBN 978–0–125–97301–4

[Rö95] RÖNTGEN, W. C.: Über eine neue Art von Strahlen. Vorläufige Mittheilung. In: *Aus den Sitzungsberichten der Würzburger Physikalisch-medicinischen Gesellschaft 1895* Würzburger Physikalisch-medicinische Gesellschaft, 1895, S. 137–147

[SHKH08] SCHERL, H. ; HOPPE, S. ; KOWARSCHIK, M. ; HORNEGGER, J.: Design and implementation of the software architecture for a 3-D reconstruction system in medical imaging. In: *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08* Institute of Electrical and Electronic Engineers, 2008, S. 661–668

[SK11] SANDERS, J. ; KANDROT, E.: *CUDA by Example*. 2nd Printing. Addison-Wesley, 2011. – ISBN 978–0–131–38768–3

[SKKH07] SCHERL, H. ; KECK, B. ; KOWARSCHIK, M. ; HORNEGGER, J.: Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA). In: *IEEE Nuclear Science Symposium Conference Record* Institute of Electrical and Electronics Engineers, 2007, S. 4464–4466

[XM04] XU, F. ; MÜLLER, K.: Ultra-Fast 3D Filtered Backprojection on Commodity Graphics Hardware. In: *IEEE International Symposium on Biomedical Imaging: Nano to Macro* Institute of Electrical and Electronics Engineers, 2004, S. 571–574

[ZHZ09] ZHAO, X. ; HU, J. ; ZHANG, P.: GPU-based 3D cone-beam CT image reconstruction for large data volume. In: *Journal of Biomedical Imaging* 2009 (2009), Nr. 8

A Grundlagen

B Umsetzung

B.1 Implementierung und Optimierung

B.1.1 Implementierung der Vorstufen

B.1.1.1 Filterung

```
__global__ void filter_creation_kernel(float* __restrict__ r,
    const std::int32_t* __restrict__ j, std::uint32_t size, float tau)
{
    auto x = blockIdx.x * blockDim.x + threadIdx.x;

    if(x < size)
    {
        if(j[x] == 0)
            r[x] = (1.f / 8.f) * (1.f / powf(tau, 2.f));
        else
        {
            if(j[x] % 2 == 0)
                r[x] = 0.f;
            else
                r[x] = -(1.f / (2.f * powf(j[x], 2.f)
                        * powf(M_PI, 2.f)
                        * powf(tau, 2.f)));
        }
    }
}
```

Quelltext B.1: Filtergenerierung

Quelltext B.2: Projektionsnormalisierung

B.1.2 Implementierung der gefilterten Rückprojektion

```
__device__ auto vol_centered_coordinate(unsigned int coord,
                                        std::uint32_t dim,
                                        float size)
    -> float
{
    auto size2 = size / 2.f;
    return -(dim * size2) + size2 + coord * size;
}
```

Quelltext B.3: Koordinatensystemtransformation im Volumen

```
__device__ auto proj_pixel_coordinate(float coord, std::uint32_t dim,
                                      float size, float offset)
    -> float
{
    auto size2 = size / 2.f;
    auto min = -(dim * size2) - offset;
    return (coord - min) / size - (1.f / 2.f);
}
```

Quelltext B.4: Koordinatensystemtransformation auf dem Detektor

C Analyse

Danksagung

Für die fachliche Betreuung bei der Erstellung dieser Arbeit bedanke ich mich recht herzlich bei Herrn Dr.-Ing. Stephan Boden von der AREVA-Stiftungsprofessur für bildgebende Messverfahren für die Energie- und Verfahrenstechnik.

Erklärungen zum Urheberrecht

Die in dieser Arbeit verwendeten Grafiken wurden – soweit nicht anders angegeben – von mir persönlich entweder ohne Vorlage oder aber nach einer am jeweiligen Erscheinungsplatz zitierten Vorlage erstellt. Für die von mir ohne Vorlage erstellten Grafiken behalte ich mir alle Rechte vor.
Ich versichere ferner, diese Arbeit eigenständig angefertigt und aus anderen Werken übernommene Zitate und Gedankengänge entsprechend kenntlich gemacht zu haben.