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.

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.

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*.

FPU *floating point unit*.

GPC *Graphics Processing Cluster*.

GPGPU *General Purpose Computation on Graphics Processing Unit*.

GPU *Graphics Processing Unit*.

OpenACC® *Open Accelerators*.

OpenCL™ *Open Computing Language*.

OpenGL™ *Open Graphics Library*.

OpenMP® *Open Multi-Processing*.

SIMD *Single Instruction, Multiple Data*.

SIMT *Single Instruction, Multiple Threads*.

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“ [Rö95]. Es war nun möglich, die innere Beschaffenheit eines Objekts auf nichtinvasive Art und Weise – also ohne es zu beschädigen – festzustellen. 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.

1.2 Der Einsatz der Computertomographie am Helmholtz-Zentrum Dresden-Rossendorf

- Ausgangssituation am HZDR: Uraltes FDK-Programm braucht mehrere Tage für eine Rekonstruktion (schlecht)
- Gesamtziel: Dieses Programm soll durch ein schnelleres und möglichst gut optimiertes abgelöst werden

1.3 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

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

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 die Aufnahme der Projektionen mit Parallelstrahlen erklärt und die Aufnahme mit Fächerstrahlen abgeleitet. Zum Schluss wird das Prinzip der Fächerstrahlen auf den dreidimensionalen Raum ausgeweitet, also auf die Aufnahme mit Kegelstrahlen.

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 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 [Kak79]. Als Beispiel dienen ein Objekt, dargestellt als die Funktion $f(x, y)$, sowie Kurvenintegrale, dargestellt durch das Parameterpaar (α, t). Die Linie AB lässt sich dann durch die folgende Formel darstellen:

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

oder allgemein für beliebige, zu AB parallele, Linien:

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

Das zu $f(x, y)$ gehörige Linienintegral ist $P_\alpha(t)$:

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

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

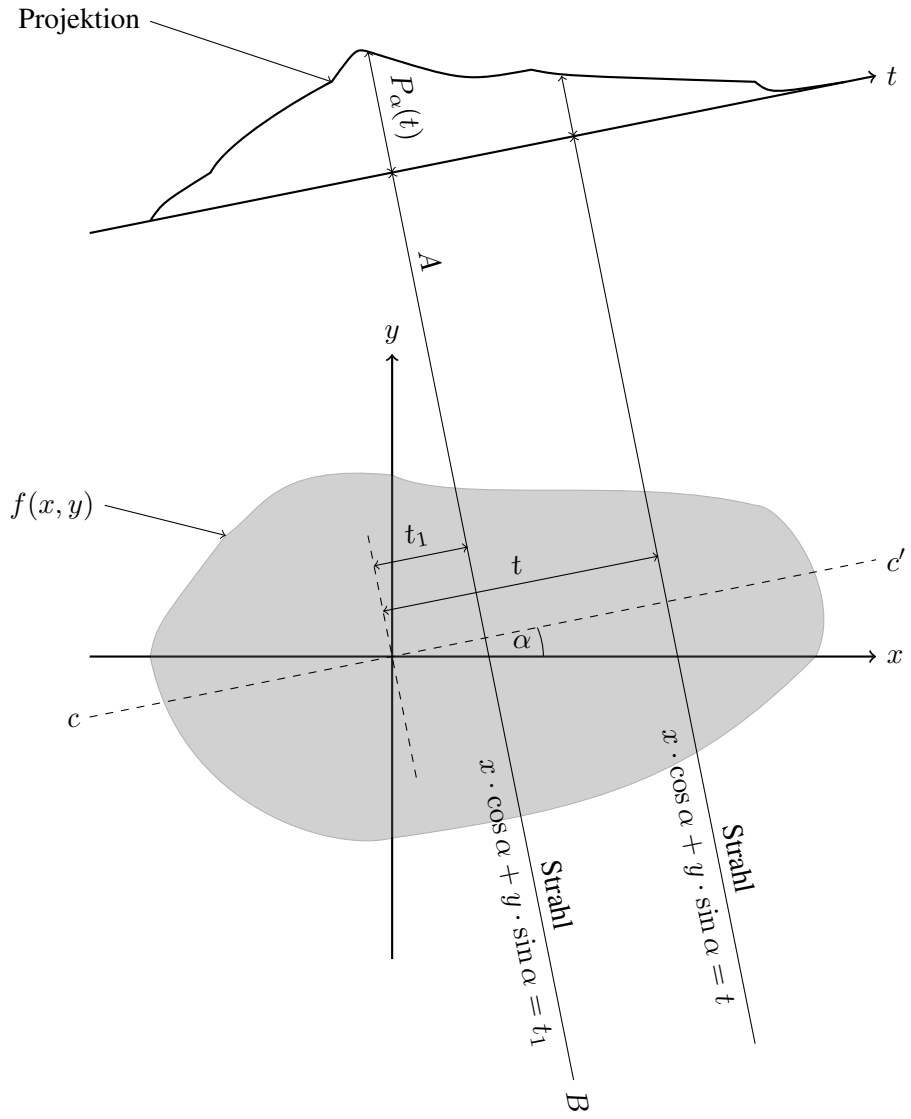

Abbildung 2.1: Zusammenhang zwischen Linienintegral und Projektion

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

Die Funktion $P_\alpha(t)$ ist die *Radon-Transformation* der Funktion $f(x,y)$ [Rad17].
Eine Projektion lässt sich als die Kombination einer Menge von Radon-Transformationen verstehen. Die (mathematisch) einfachste Projektion ist eine Sammlung von Parallelstrahlintegralen $P_\alpha(t)$ unter einem konstanten Winkel α. Man bezeichnet eine solche Projektion als *Parallelstrahlprojektion* (siehe Abbildung **??**). 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 **??**). Solcherart erzeugte Projektionen nennt man aufgrund der fächerförmigen Strahlen *Fächerstrahlprojektionen* [KS88].

2.1.1.2 Das Fourier-Schichten-Theorem

Betrachtet man die eindimensionale

2.1.1.3 Parallelstrahlen

2.1.1.4 Fächerstrahlen

2.1.1.5 Kegelstrahlen

2.1.2 Die prinzipielle Funktionsweise der Computertomographie

Am Anfang der Computertomographie steht das Röntgenverfahren, das 1895 vom deutschen Physiker Wilhelm Conrad Röntgen entdeckt wurde [Rö95]. Mit Hilfe einer Strahlungsquelle wird ein Objekt durchleuchtet und auf einem Film bzw. einem Detektor abgebildet; der dreidimensionale Körper wird also auf eine zweidimensionale Fläche projiziert. Diesen Schritt bezeichnet man als *Vorwärtsprojektion*. 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 [Cor63] [Cor64]; ihm war allerdings unbekannt [Cor79], dass diese schon 1917 vom österreichischen Mathematiker Johann Radon gefunden wurden [Rad17]. Mathematisch ist der Vorgang der *Rückprojektion* eine Anwendung der nach Radon benannten *Radon-Transformation*.
Ein Problem der Vorwärtsprojektion ist der Informationsverlust, der durch die mangelnde Tiefe des Films bzw. Detektors entsteht; die Tiefeninformationen werden auf die zweidimensionale Fläche „verschmiert“. Bei der Rückprojektion lässt sich dieser Verlust durch die Wahl eines geeigneten Bildfilters wiederum kaschieren, weshalb man auch von der *gefilterten Rückprojektion* spricht.
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 [Kal00]. 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.

2.1.3 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 zugrundeliegende Geometrie näher erläutert, bevor die einzelnen Schritte des FDK-Algorithmus detaillierter betrachtet werden.

2.1.3.1 Geometrie

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 bezeichnet (vgl. Abbildung 2.2).

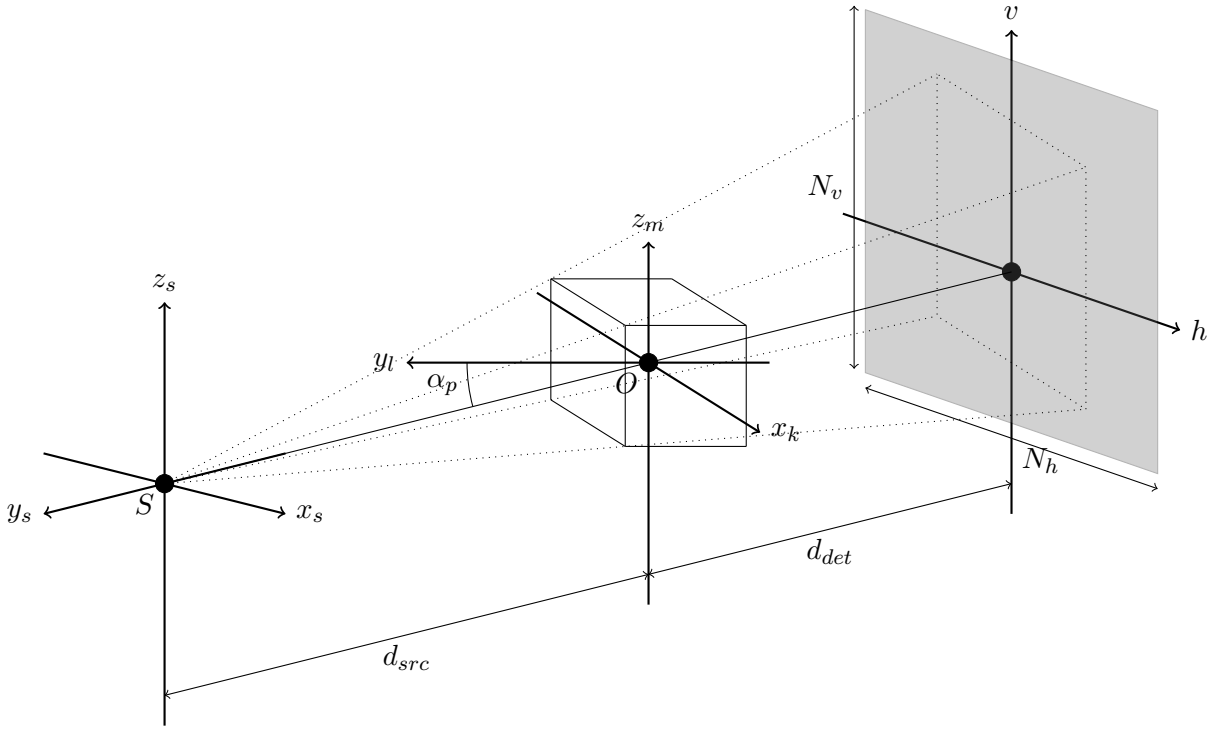

Abbildung 2.2: Geometrie der gefilterten Rückprojektion

2.1.3.2 Wichtung

Nach der Aufnahme wird jede Projektion gewichtet. Dafür wird jedes Pixel mit den Koordinaten (j, i) mit dem Wichtungsfaktor w_{ij} multipliziert.

$$w_{ij} = \frac{d_{det} - d_{src}}{\sqrt{(d_{det} - d_{src})^2 + h_j^2 + v_i^2}} \tag{2.4}$$

2.1.3.3 Filterung

Zum Ausgleich der durch die Vorwärtsprojektion verloren gegangenen Tiefeninformationen werden die Projektionen im nächsten Schritt zeilenweise gefiltert. Zu diesem Zweck müssen die Projektionen und der Filter allerdings mittels der diskreten Fouriertransformation in den komplexen Raum transformiert werden; zum Einsatz kommt dabei das Verfahren der schnellen Fouriertransformation (*fast Fourier transform*, FFT) nach Cooley und Tukey [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:

$$N_{hFFT} = 2 \cdot 2^{\lceil \log_2 N_h \rceil} \tag{2.5}$$

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

$$\begin{aligned} r(j) \text{ mit } j &\in \left[-\frac{N_{hFFT}-2}{2}, \frac{N_{hFFT}}{2} \right] \\ r(j) &= \begin{cases} \frac{1}{8} \cdot \frac{1}{\tau^2} & \text{wenn } j = 0 \\ 0 & \text{wenn } j \text{ gerade} \\ -\frac{1}{2j^2\pi^2\tau^2} & \text{wenn } j \text{ ungerade} \end{cases} \end{aligned} \tag{2.6}$$

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

$$\begin{aligned} p: & \text{ mit Nullen aufgefüllte Projektionszeile} \\ p(0 \dots N_{h-1}) &= \text{det}(0 \dots N_{h-1}) \\ p(N_h \dots N_{hFFT}) &= 0 \end{aligned} \tag{2.7}$$

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

$$\begin{aligned} R &= \text{FFT}(r) \\ P &= \text{FFT}(p) \\ F &= P \cdot R \quad \text{sowohl für den reellen als auch den imaginären Teil} \end{aligned} \tag{2.8}$$

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

$$\begin{aligned} f &= \text{IFFT}(F) \\ \text{gefilterte Projektionszeile}: & \; f(0 \dots N_{h-1}) \end{aligned} \tag{2.9}$$

2.1.3.4 Rückprojektion

Die auf gefilterten Projektionen können nun nach dem folgenden Algorithmus für die Rückprojektion verwendet werden:
Für jede Projektion p mit dem Drehwinkel α_p:

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

$$\begin{aligned} s &= x_k \cos\alpha_p + y_l \sin\alpha_p \\ t &= -x_k \sin\alpha_p + y_l \cos\alpha_p \\ z &= z_m \end{aligned} \tag{2.10}$$

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

$$\begin{aligned} h' = y' = t \cdot \frac{d_{det} - d_{src}}{s - d_{src}} \\ v' = z' = z \cdot \frac{d_{det} - d_{src}}{s - d_{src}} \end{aligned} \tag{2.11}$$

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

$$det' = det(h', v') \tag{2.12}$$

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

$$\begin{aligned} vol_{klm} = vol_{klm} + 0,5 \cdot det' \cdot u^2 \\ \text{mit } u = \frac{d_{src}}{s - d_{src}} \end{aligned} \tag{2.13}$$

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

2.1.4 Bisherige Parallelisierungsansätze

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.

2.2 Die NVIDIA®-CUDA®-Plattform

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. Durch die Einführung von dezidierten Berechnungseinheiten für Gleitkommazahlen (*floating point unit* (FPU)) sowie den zunehmenden Funktionsumfang der auf den Grafikkarten verbauten Shader-Einheiten wurde es theoretisch möglich, Berechnungen, die vorher nur von CPUs ausgeführt werden konnten, nun auch von GPUs ausführen zu lassen. Dabei haben Grafikkarten gegenüber herkömmlichen Prozessoren den Vorteil, dass die auf ihnen ausgeführten Berechnungen aufgrund ihrer für die Computergrafik optimierten Bauweise – also die Manipulation vieler Pixel zur gleichen Zeit – automatisch *datenparallel* sind. *Datenparallelität* bezeichnet dabei die parallele Ausführung derselben Berechnung bzw. Anweisung auf verschiedenen Daten. Dem gegenüber steht die *Taskparallelität*, mit der eine parallele Abarbeitung verschiedener Aufgaben gemeint ist. *Taskparallelität* entspricht eher dem Programmiermodell der klassischen CPU, ist auf GPUs aufgrund der ihnen inhärenten datenparallelen Funktionsweise nur begrenzt anwendbar.
Die *datenparallele* Berechnung auf GPUs, meistens *General Purpose Computation on Graphics Processing Unit* (GPGPU) genannt, bietet sich insbesondere für die Verarbeitung großer Datenmengen an, wie sie zum Beispiel in der Wissenschaft häufig vorkommt. Am Anfang der 2000er Jahre gab es allerdings

keine komfortable Möglichkeit, die erhältlichen Grafikkarten direkt zu programmieren; man war daher gezwungen, die zu berechnenden Daten zunächst in Objekte der Computergrafik umzuwandeln (beispielsweise Texturen) und mit den Mitteln der bestehenden Computergrafik-Bibliotheken wie der *Open Graphics Library* (OpenGL™) oder DirectX® zu bearbeiten. Mit der Einführung von CUDA® entfiel diese Beschränkung, da es nun möglich war, speziell für die Grafikkarte geschriebene Programme auf dieser auszuführen, ohne den Umweg über Computergrafik-Bibliotheken gehen zu müssen.

2.2.2 Das CUDA®-Programmier- und Ausführungsmodell

2.2.2.1 Kernel und Device-Funktionen

Das Herzstück eines CUDA®-Programms ist der Kernel, also ein speziell für die GPU geschriebenes Programmstück. Ein Kernel wird in einem C- oder C++-Dialekt geschrieben und durch den CUDA®-eigenen Compiler in den Maschinencode der jeweiligen Zielarchitektur übersetzt. Dieser Kernel kann dann, ähnlich einer normalen Funktion, aus einem in C oder C++ geschriebenen Programm (dem Host) aufgerufen werden. Der so gestartete Kernel wird dann durch die CUDA®-Laufzeitumgebung auf der GPU (dem Device) ausgeführt. Da der Kernel nicht auf der CPU ausgeführt wird, ist sein Aufruf in Bezug auf den Host asynchron, das Ende der Berechnung wird auf der Hostseite also nicht abgewartet und erfordert eine manuelle Synchronisierung durch den Host:

```
auto main(int argc, char** argv) -> int
{
    kernel<<<...,...>>>(...);
    // Host-Programm wird nach Kernelaufruf weiter ausgeführt
    // Kernel-Ergebnis kann noch nicht verwendet werden
    cudaDeviceSynchronize();
    // Kernel-Ergebnis ab jetzt verwendbar
}
```

Auf dem Device werden mehrere Kernel in der Reihenfolge ihrer Aufrufe sequentiell abgearbeitet, sofern sie nicht auf mehrere Streams aufgeteilt werden (siehe Abschnitt 2.2.2.5):

```
auto main(int argc, char** argv) -> int
{
    kernel1<<<...,...>>>(...); // sofortige Ausführung
    kernel2<<<...,...>>>(...); // Ausführung nach Ende von kernel1
    kernel3<<<...,...>>>(...); // Ausführung nach Ende von kernel2
}
```

Die Deklaration eines Kernels ähnelt der Deklaration einer normalen C/C++-Funktion. Er muss allerdings stets den Rückgabetypen `void` haben, da der Kernel asynchron in Bezug auf den Host läuft und somit kein Ergebnis zurückgeben kann. Ein Kernel wird außerdem mit dem Schlüsselwort `__global__` als solcher markiert:

```
__global__ void foo(int* bar) { ... }
```

Neben dem Kernel gibt es Funktionen, die nur auf dem Device ausgeführt werden können, das heißt nur aus einem Kernel heraus. Diese Device-Funktionen können einen Rückgabetypen haben und müssen mit dem Schlüsselwort `__device__` markiert werden:

```
__device__ auto baz() -> foo { ... }
```

2.2.2.2 Parallelität

Jeder Kernel, der auf dem Host aufgerufen wird, wird gleichzeitig von mehreren CUDA®-Threads ausgeführt. Dieses Ausführungsmodell, das in Anlehnung an das von der klassischen CPU-Programmierung bekannte *Single Instruction, Multiple Data* (SIMD) von NVIDIA® als *Single Instruction, Multiple Threads* (SIMT) bezeichnet wird, sieht eine *datenparallele* (vgl. Abschnitt 2.2.1) Abarbeitung der Aufgabe vor; mehrere Threads führen also parallel die selbe Operation auf verschiedenen Daten aus.

```
auto main(int argc, char** argv) -> int
{
    kernel1<<<...,...>>>(...); // Ausführung durch mehrere Threads
    kernel2<<<...,...>>>(...); // Ausführung durch mehrere Threads
    kernel3<<<...,...>>>(...); // Ausführung durch mehrere Threads
}
```

Die Nähe zu SIMD wird insbesondere durch die bei Divergenzen entstehenden Probleme deutlich. Aufgrund architektonischer Besonderheiten (vgl. Abschnitt 2.2.2.3) kann es dazu kommen, dass Verzweigungspfade sequentiell abgearbeitet werden müssen:

```
__global__ void kernel(...)
{
    if(something)
        // alle Threads führen erst diesen Zweig aus...
    else
        // ... und dann diesen
}
```

Divergenzen können außerdem zu unterschiedlichen Thread-Laufzeiten führen, wenn die Abarbeitung der Verzweigungen unterschiedlich lange dauert. Die Laufzeit des Kernels richtet sich immer nach dem langsamsten Thread. Es entstehen also Wartezeiten für die schnelleren Threads, die dadurch nicht von anderen Kerneln für weitere Aufgaben herangezogen werden können.

2.2.2.3 Architektur

Der Aufbau einer CUDA®-fähigen GPU unterscheidet sich stark von der Struktur klassischer Prozessoren. Die gröbste Organisationseinheit auf einer GPU der neuesten Generation (Pascal™) ist ein *Graphics Processing Cluster* (GPC). Jeder GPC lässt sich unterteilen in bis zu zehn Einheiten des *Streaming Multiprocessor* (SM). Ein SM besteht wiederum aus 64 CUDA®-Kernen, womit sich eine Gesamtkernzahl

von bis zu 3840 Kernen pro Pascal™-GPU ergibt. Eine handelsübliche Intel®-Xeon®-CPU ist dagegen aus maximal 32 Kernen aufgebaut.

2.2.2.4 Speicher

Im Gegensatz zu CPU-Kernen verfügen CUDA®-Kerne über keine eigenen Register oder Caches. Stattdessen verfügt jeder SM über eine feste Registerzahl sowie eine gewisse Cache-Größe, die auf die benötigten Kerne pro Kernel aufgeteilt werden. Ferner verfügt

2.2.2.5 Streams

2.2.2.6 Beispiel

Die Funktionsweise der CUDA®-Plattform soll im Folgenden anhand eines kleinen Beispiels veranschaulicht werden. Die folgende, in C++ geschriebene Funktion summiert die Vektoren `A` und `B` der Länge `size` und speichert das Ergebnis im gleichlangen Vektor `C`:

```
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];
}
```

Portiert man diese Funktion auf die CUDA®-Plattform, so erhält man den folgenden Kernel:

```
__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];
}
```

2.2.3 Alternativen zu CUDA®

2.2.3.1 OpenCL™

Während CUDA® ein großer Fortschritt im Bereich des GPGPU war, so blieb es aufgrund der Tatsache, dass es ein proprietäres Produkt von NVIDIA® ist, stets auf Grafikkarten und Beschleuniger dieses Herstellers beschränkt. Der Computerhersteller Apple®, der in der zweiten Hälfte der 2000er Jahre Grafikkarten des NVIDIA®-Konkurrenten AMD verbaute und seinen Kunden ebenfalls einen einfachen Zugang zu GPGPU ermöglichen wollte, entwickelte daher eine eigene, wenn auch stark an CUDA® angelehnte, GPGPU-Plattform, die OpenCL™ genannt wurde, und übergab diese kurz darauf dem Industriekonsortium Khronos zur Standardisierung. Im Jahre 2009 wurde dann mit OpenCL™ 1.0 die erste hardware- und herstellerunabhängige GPGPU-Plattform herausgegeben, die unter anderem von den drei

größten GPU-Herstellern NVIDIA®, AMD und Intel® unterstützt wurde. Theoretisch konnte dasselbe Programm nun ohne Änderungen mit einer beliebigen GPU beschleunigt werden. In der Praxis hinkte die Unterstützung sukzessiver Versionen des Standards seitens einiger Hersteller jedoch der Entwicklung der Hardware hinterher. Während das 2010 erschienene OpenCL™ 1.1 noch relativ zeitnah von allen Herstellern unterstützt wurde, wurde die 2011 veröffentlichte Version OpenCL™ 1.2 erst 2015 offiziell auf NVIDIA®-GPUs verfügbar; das 2013 erschienene OpenCL™ 2.0 wird von NVIDIA® erst seit Anfang 2017 zu Evaluierungszwecken unterstützt. Die Firma Apple®, die OpenCL™ ursprünglich entwickelt hatte, unterstützt bis heute nur OpenCL™ 1.2. Für eine wirklich plattformunabhängige Programmierung ist man also gezwungen, die verwendeten OpenCL™-Befehle auf den kleinsten gemeinsamen Nenner der gewünschten Zielplattformen zu beschränken; bei GPUs waren dies über einen langen Zeitraum die Befehle der Version OpenCL™ 1.1.
Neben der unterschiedlichen Versionsunterstützung durch die Hersteller ist die Portabilität außerdem durch unterschiedliche Leistungen der Kernel auf verschiedenen, aber technisch vergleichbaren Plattformen – wie etwa GPUs der gleichen Generation von NVIDIA® und AMD – beschränkt [DWL+12]. Durch den Einsatz selbstoptimierender Techniken (*auto-tuning*) ist dieses Problem aber möglicherweise lösbar [DWL+12] [FVS11].
Eine weitere Schwierigkeit beim Einsatz von OpenCL™ ist die unterschiedliche Leistung funktionsgleicher CUDA®- und OpenCL™-Kernel auf NVIDIA®-Hardware. Fang et al. stellten fest, dass ein von CUDA® zu OpenCL™ übertragener Kernel ohne weitere Optimierungen bis zu 30% mehr Zeit benötigte als das CUDA®-Original [FVS11]. Karimi et al. kamen in ihrer Untersuchung zu dem Ergebnis, dass OpenCL™-Kernel auf dem gleichen Beschleuniger 13% bis 63% langsamer sind als ihre CUDA®-Gegenstücke [KDH10].

```
__kernel void vec_add(__constant int* A, __constant int* B, int*
 ↪ C, size_t size)
{
    size_t i = get_global_id(0);
    if(i < size)
        C[i] = A[i] + B[i];
}
```

2.2.3.2 OpenMP®

Parallel zu der Weiterentwicklung der Grafikkarten fand auch bei den herkömmlichen Prozessoren durch die allmähliche Erhöhung der Kernanzahl eine Hinwendung zu parallelen Ausführungsmodellen statt. Ein Ansatz zur Vereinfachung der Programmierung vieler CPU-Kerne ist das seit 1997 gemeinschaftlich entwickelte *Open Multi-Processing* (OpenMP®). Im Gegensatz zu den Programmiermodellen von CUDA® oder OpenCL™, die separate Programme auf den Beschleunigern starten, wird OpenMP® direkt mit Compiler-Direktiven in das eigentliche Programm eingebettet.

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

2.2.3.3 OpenACC®

An OpenMP® angelehnt ist *Open Accelerators* (OpenACC®), welches die GPGPU-Programmierung mit Compiler-Direktiven ermöglicht. Syntax und Funktionsweise ähneln stark der von OpenMP®, haben jedoch den Nachteil, dass eine auf bestimmte Hardware zugeschnittene Optimierung nicht mehr so einfach möglich ist.

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

3 Umsetzung

3.1 Variantenvergleich

3.1.1 Bestehende Parallelisierungsstrategien und ihre Grenzen

Von den in Abschnitt 2.1.4 genannten Ansätzen in der Literatur sind aufgrund ihrer Umsetzung für GPUs die Strategien von Xu et al. [XM04], Scherl et al. [SKKH07] und Zhao et al. [ZHZ09] von besonderem Interesse für diese Arbeit.

Da Xu et al. 2004 mit ihrer Arbeit [XM04] 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 [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. 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 [NVIb]. Insbesondere ist es möglich, das ganze Volumen oder 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 [SKKH07] baute auf der Idee, das Volumen im Speicher zu halten, auf und ging stattdessen den Weg, jede Projektion einzeln in dieses Volumen zu projizieren. Zur Trennung bzw. Kapselung der einzelnen Schritte entwickelten sie in einem vorherigen Schritt [SHKH08] eine Pipeline-Struktur (nach Mattson et al. [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 Grenzen bei dem vorgeschlagenen Verfahren der Gruppe um Zhao et al. [ZHZ09] sind vor allem praktischer Natur. Das von ihnen vorgestellte Modell sieht vor, Symmetrien auszunutzen und dadurch Rechenzeit einzusparen. 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. 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.

3.1.2 Das Problem des GPU-Speichers

3.1.3 Heterogene GPU-Systeme und effiziente Arbeitsteilung

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

- Wartbarkeit
- Portabilität
- sinnvolle Geschwindigkeit
- nutzbar auf Laptop / Workstation / GPU-Cluster

3.2.2 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.

3.2.2.1 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 Implementierung 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 3.1). 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 3.2).
Spannt man nun über dem Detektor ein Koordinatensystem auf (mit dessen Zentrum als Ursprung, siehe Abbildung 3.3), 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:

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

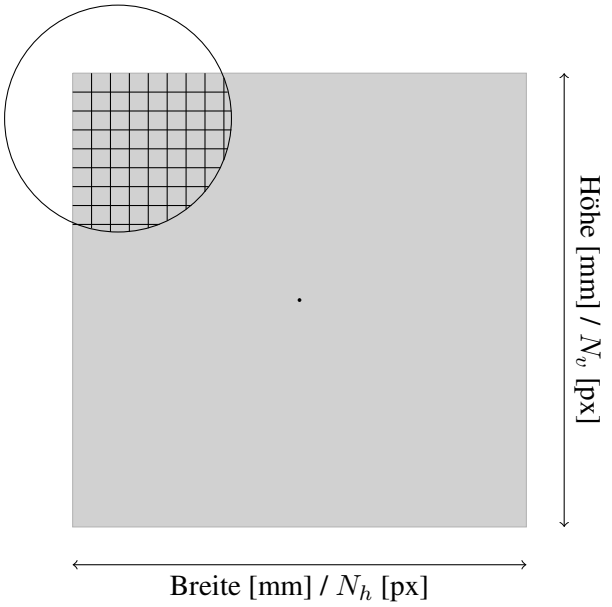

Abbildung 3.1: Detektorgeometrie

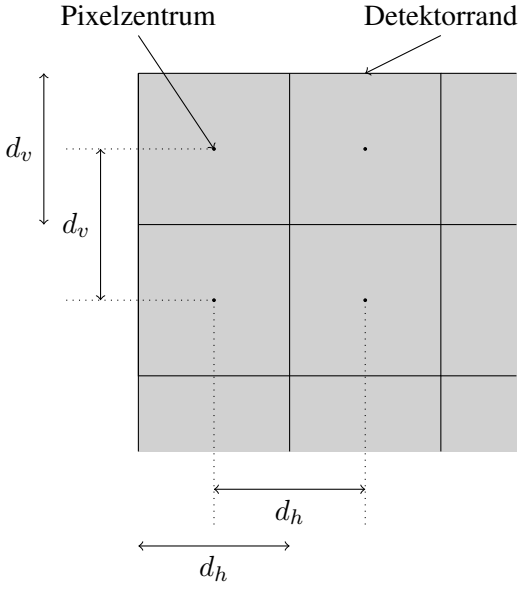

Abbildung 3.2: Pixelgeometrie

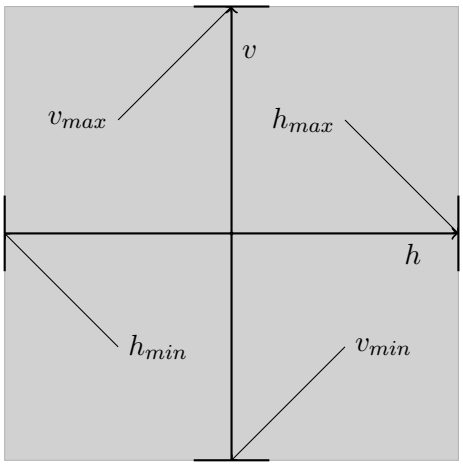

Abbildung 3.3: Detektorkoordinatensystem

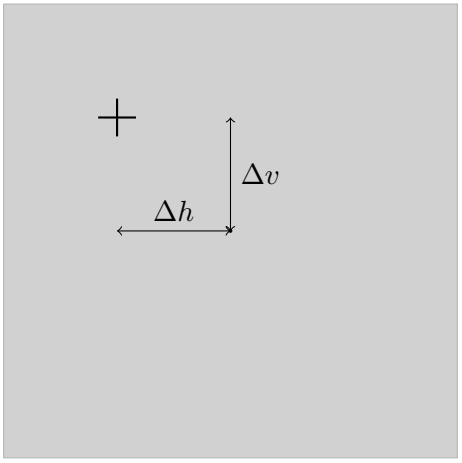

Abbildung 3.4: Verschiebungsgeometrie

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

3.2.2.2 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 3.4). Von der Strahlungsquelle ausgehend 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:

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

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

3.2.2.3 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.

3.2.3 Geometrische Berechnungen

- Berechnung der Volumengeometrie
- Aufteilung in Teilvolumen

3.2.4 Implementierung der Vorstufen

3.2.4.1 Wichtung

Die Grundlage der Wichtungsoperation ist die in Abschnitt 2.1.3.2 vorgestellte Formel 2.4:

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

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 3.1). Die Berechnung der Wichtungsmatrix hängt von mehreren geometrischen Parametern ab (vgl. Abschnitt 2.1.3.1 und Abbildungen 3.1, 3.2, 3.3, 3.4):

- `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 3.1: 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 3.2). 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 3.2: Wichtung einer Projektion

3.2.4.2 Filterung

Dem in Abschnitt 2.1.3.3 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 2.6 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 3.3). 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 3.3: Filterung einer Projektion

3.2.5 Implementierung der gefilterten Rückprojektion

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

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?

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

[DWL+12] DU, P. ; WEBER, R. ; LUSZCZEK, P. ; TOMOV, S. ; PETERSON, G. ; DONGARRA, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. In: *Parallel Computing* 38 (2012), August, Nr. 8, S. 391–407

[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

[FVS11] FANG, J. ; VARBANESCU, A. L. ; SIPS, H.: A Comprehensive Performance Comparison of CUDA and OpenCL. In: *2011 International Conference on Parallel Processing*, 2011, S. 216–225

[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

[Kal00] KALENDER, W. A.: *Computertomographie: Grundlagen, Gerätetechnologie, Bildqualität, Anwendungen*. Publicis MCD Verlag, 2000. – ISBN 978–3–895–78082–0

[KDH10] KARIMI, K. ; DICKSON, N. G. ; HAMZE, F.: A Performance Comparison of CUDA and OpenCL. In: *arXiv* arXiv:1005.2581v3 (2010)

[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

[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

[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

[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

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.