

BACHELORTHESES  
Anna Larissa Rauschelbach

# Beschleunigung eines Algorithmus zur Ermittlung der Punktspreizfunktion in medizinischen Röntgensystemen

---

FAKULTÄT TECHNIK UND INFORMATIK  
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering  
Department of Information and Electrical Engineering

Anna Larissa Rauschelbach

# Beschleunigung eines Algorithmus zur Ermittlung der Punktspreizfunktion in medizinischen Röntgensystemen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung  
im Studiengang *Bachelor of Science Regenerative Energiesysteme und Energie-  
management*  
am Department Informations- und Elektrotechnik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Robert Heß  
Zweitgutachter: Prof. Dr. Kolja Eger

Eingereicht am: 14. April 2022

**Anna Larissa Rauschelbach**

**Thema der Arbeit**

Beschleunigung eines Algorithmus zur Ermittlung der Punktspreizfunktion in medizinischen Röntgensystemen

**Stichworte**

Röntgen, Punktspreizfunktion, Laufzeitoptimierung

**Kurzzusammenfassung**

Im Rahmen dieser Ausarbeitung wird eine Software auf Engpässe untersucht und mittels Parallelisierung und Compileranweisungen beschleunigt. Hierzu werden die Grundlagen zur Thematik zusammengefasst, Anforderungen gestellt und ein Design entwickelt. Dieses Design wird umgesetzt und getestet. Zusätzlich wird das Laufzeitverhalten der beschleunigten Software untersucht.

**Anna Larissa Rauschelbach**

**Title of Thesis**

Speedup of an algorithm to determine the point spread function of a medical x-ray

**Keywords**

x-ray, point-spread-function, speedup, runtime optimization

**Abstract**

This report describes the runtime analysis of a given software which will be sped up using parallelization and compiler directives. The theory of the subject will be described, requirements will be defined and a design will be developed. This design will be implemented and tested. The runtime of sped up software will be analyzed.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen</b>	<b>2</b>
2.1	Grundlagen Röntgen . . . . .	2
2.1.1	Entstehung von Röntgenstrahlung . . . . .	3
2.1.2	Das Projektionsröntgen . . . . .	4
2.1.3	Aufbau und Funktionsweise einer Drehanoden-Röntgenröhre . . . . .	5
2.1.4	Das Qualitätskriterium der Brennfleckgröße . . . . .	6
2.1.5	Systemtechnische Betrachtung . . . . .	7
2.1.6	Der Siemensstern als Testmuster zur Prüfung der optischen Qualität eines Röntgensystems . . . . .	9
2.1.7	Die Bestimmung der PSF aus einer Sternrastreraufnahme . . . . .	10
2.2	Grundlagen der parallelen Programmierung . . . . .	12
2.2.1	Klassifikation nach Flynn's . . . . .	12
2.2.2	Befehlssatzarchitektur . . . . .	13
2.2.3	Pipelining . . . . .	14
2.2.4	Out-Of-Order Ausführung . . . . .	15
2.2.5	Superskalare Ausführung . . . . .	16
2.2.6	Spekulative Ausführung . . . . .	16
2.2.7	Aufbau und Arbeitsweise eines modernen Mehrkernprozessors . . . . .	17
2.3	Grundlagen OpenMP . . . . .	19
2.3.1	Das parallele Ausführungsmodell von OpenMP . . . . .	19
2.3.2	Parallelisieren von Schleifen . . . . .	20
2.4	Grundlagen der Software-Optimierung . . . . .	22
2.4.1	Sampling zur Leistungsanalyse von Software . . . . .	23
2.4.2	Optimierungseinstellungen des GNU Compilers . . . . .	23
2.5	Verwendete Software . . . . .	24
2.5.1	VSCoDe . . . . .	25

2.5.2	Very Sleepy . . . . .	25
2.5.3	OpenMP Bibliothek . . . . .	25
2.5.4	GNU Compiler . . . . .	25
2.5.5	Doxygen . . . . .	26
2.6	Verwendete Hardware . . . . .	26
<b>3</b>	<b>Anforderungen</b>	<b>27</b>
<b>4</b>	<b>Design</b>	<b>28</b>
4.1	Programmablaufanalyse . . . . .	28
4.2	Leistungsanalyse des Ausgangszustandes . . . . .	29
4.3	Beschleunigung durch Parallelisierung . . . . .	32
4.4	Beschleunigung durch Compiler-Optimierungen . . . . .	36
<b>5</b>	<b>Implementierung</b>	<b>38</b>
5.1	Implementierung der Parallelisierung . . . . .	38
5.1.1	Bestimmung der Koeffizientenmatrix . . . . .	38
5.1.2	Bestimmung der Konstanten . . . . .	42
5.1.3	Lösung des Gleichungssystems . . . . .	42
5.2	Parametersteuerung . . . . .	43
5.2.1	Ablaufpläne . . . . .	43
5.2.2	Anzahl der zu nutzenden Threads . . . . .	45
5.2.3	Bestimmung der Konstanten . . . . .	47
5.2.4	Method . . . . .	48
5.2.5	Automatisierung der Abweichung . . . . .	50
5.3	Compilerkonfigurationen . . . . .	51
<b>6</b>	<b>Validierung</b>	<b>53</b>
6.1	Untersuchung des Laufzeitverhaltens der parallelisierten Software . . . . .	53
6.1.1	Laufzeit bei variierender Bildgröße der simulierten Röntgenaufnahme	54
6.1.2	Laufzeit bei variierender Bildgröße der PSF . . . . .	56
6.1.3	Laufzeit bei variierenden OpenMP Schedules . . . . .	59
6.1.4	Laufzeit bei variierender Threadanzahl . . . . .	61
6.2	Weitere Beschleunigung durch Compilereinstellungen . . . . .	63
6.3	Gesamtbeschleunigung auf drei Systemen . . . . .	65
6.4	Test auf Linux Sub-System . . . . .	67
6.5	Zusammenfassung . . . . .	68

<b>7 Fazit &amp; Ausblick</b>	<b>69</b>
<b>Literaturverzeichnis</b>	<b>71</b>
<b>A Anhang</b>	<b>75</b>
A.1 Messwerte: Variierende Bildgröße conv . . . . .	75
A.2 Messwerte: Variierende Bildgröße psf . . . . .	82
A.3 Messwerte: Variierender Schedule . . . . .	85
A.4 Messwerte: Variierende Threadanzahl . . . . .	88
A.5 Messwerte: Variierende Optimierungslevel . . . . .	90
A.6 Code: Methoden zur Bestimmung der Koeffizientenmatrix . . . . .	93
A.7 Code: OpenMP Schedule . . . . .	96
A.8 Code: OpenMP Threadanzahl . . . . .	97
A.9 Code: Bestimmung der Konstanten . . . . .	97
A.10 Compilerkonfiguration . . . . .	98
<b>Selbstständigkeitserklärung</b>	<b>100</b>

# 1 Einleitung

In der Medizin ist das Projektionsröntgen das älteste und weit verbreitetste bildgebende Verfahren [4]. Die Bildqualität einer Röntgenaufnahme ist ein ausschlaggebender Qualitätsparameter eines Röntgensystem und lässt sich durch die Impulsantwort des Systems näher analysieren. Bei der Impulsantwort eines Röntgensystems wird von der Punktspreizfunktion gesprochen, mit welcher das abzubildende Objekt gefaltet wird [4]. Von Prof. Dr. Heß wurde ein Verfahren zur Bestimmung der Punktspreizfunktion aus Aufnahmen mit Sternrasterobjekten entwickelt [7]. Das Verfahren zur Bestimmung der Punktspreizfunktion ist in einer Software implementiert, die von Prof. Dr. Heß zur Verfügung gestellt wird. Aufgrund des Rechenaufwandes kommt es zu einer hohen Laufzeit, die es im Rahmen dieser Ausarbeitung zu beschleunigen gilt. Zur Beschleunigung der Software soll eine Leistungsanalyse des Ausgangszustandes der Software durchgeführt werden, anhand dessen sollen Engpässe gefunden und durch Parallelisierung auf dem Prozessor beschleunigt werden. Darüber hinaus soll das Laufzeitverhalten der seriellen und parallelen Ausführung der Software in Abhängigkeit einflussnehmender Parameter untersucht werden. Eine zusätzliche Beschleunigung soll über Compilereinstellungen implementiert und untersucht werden.

## 2 Grundlagen

In diesem Kapitel werden die Grundlagen für die Aufgabenstellung beschrieben. Im Folgenden werden die Grundlagen zu den Themen Röntgensysteme, parallele Programmierung und Software-Optimierung erläutert. Darüber hinaus werden die verwendete Software und Hardware beschrieben.

### 2.1 Grundlagen Röntgen

Am 8. November 1895 wurde von Wilhelm Conrad Röntgen eine bislang unbekannte, sehr durchdringungsfähige Art der Strahlung entdeckt, die später nach ihm benannte Röntgenstrahlung [1]. Die hohe Eindringtiefe der Röntgenstrahlung durchdringt neben Papier und Holz auch den menschlichen Körper. Mittels eines Fluoreszenzschirmes wurde diese Strahlung sichtbar gemacht. Die erste Röntgenaufnahme eines Menschen bildet die Hand von Wilhelm Conrad Röntgens' Frau ab [2.1]. Die Entdeckung der Röntgenstrahlung gilt als Meilenstein der Medizin und als Geburtsstunde der Radiologie.



Abbildung 2.1: Die erste Röntgenaufnahme, zu sehen ist die Hand von Wilhelm Conrad Röntgens' Frau [19]

### 2.1.1 Entstehung von Röntgenstrahlung

Die wesentlichen Bestandteile eines Röntgensystems [2.2] sind die Kathode und die Anode, die an die Beschleunigungsspannung  $U$  angeschlossen sind.

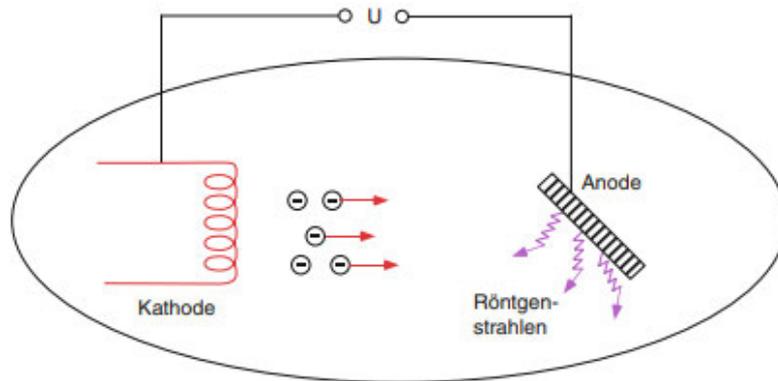


Abbildung 2.2: Schematische Darstellung der Erzeugung von Röntgenstrahlen [4]

Die Röntgenstrahlung entsteht durch das Abbremsen stark beschleunigter Elektronen im Anodenmaterial. Aus der Kathode treten Elektronen aufgrund thermoelektrischer Effekte aus und werden durch die Beschleunigungsspannung  $U$  in Richtung Anode beschleunigt. Die beschleunigten Elektronen treffen auf das Anodenmaterial und werden abgebremst, dabei entsteht Röntgenstrahlung und Wärme. Die im Anodenmaterial freigesetzte Röntgenstrahlung setzt sich aus der Bremsstrahlung und der charakteristischen Strahlung zusammen.

Dringen stark beschleunigte Elektronen in das Anodenmaterial ein, werden sie von den Feldern der Atome des Anodenmaterials abgelenkt und verlieren an Geschwindigkeit. Diese Abbremsprozesse verursachen einen kontinuierlichen Energieverlust in Form von Geschwindigkeit, bei der Strahlung emittiert wird. Diese Strahlung wird als Bremsstrahlung bezeichnet.

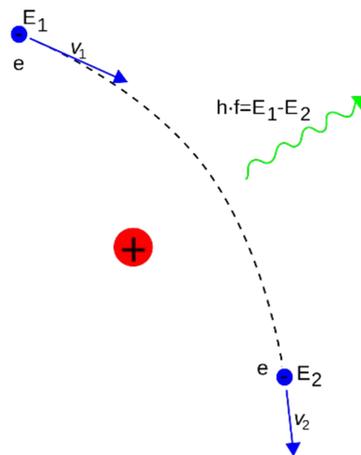


Abbildung 2.3: Schematische Darstellung der Erzeugung von Röntgenbremsstrahlung [17]

Wird ein Elektron durch den Zusammenstoß mit einem beschleunigten Elektron aus der inneren Schale des Atomkernes gestoßen, emittiert beim Nachrücken eines Elektrons aus einer höheren Schale Strahlung [2.4]. Diese Strahlung wird als charakteristische Strahlung bezeichnet. Da die Atome des Anodenmaterials eine hohe Ordnungszahl haben (beispielsweise Wolfram) wird beim Nachrücken der Elektronen Röntgenstrahlung emittiert [4, 34].

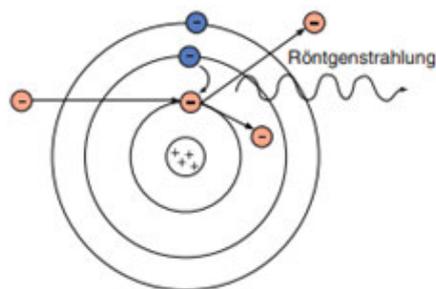


Abbildung 2.4: Schematische Darstellung der Erzeugung von charakteristischer Strahlung [4]

### 2.1.2 Das Projektionsröntgen

Bei dem Verfahren des Projektionsröntgen wird ein Körperteil mit gebündelten Röntgenstrahlen bestrahlt und das Integral über den linearen Schwächungskoeffizienten gemessen.

sen [14]. Aussagekräftige Röntgenbilder entstehen durch Differenzen in der Absorption der Röntgenstrahlen, Knochen absorbieren im Gegensatz zu Weichteilen mehr Strahlung, diese Differenz der Absorption wird in Form von Kontrast auf einem Röntgenbild deutlich[14]. Diese Methode der Bildgebung wurde kurz nach der Entdeckung der Röntgenstrahlung entwickelt und ist aufgrund der schnellen, zuverlässigen und aussagekräftigen Bildgebung eines der etabliertesten Verfahren in der Medizin.

### 2.1.3 Aufbau und Funktionsweise einer Drehanoden-Röntgenröhre

Für die Diagnostik sind Röntgensysteme geeignet, die für einen kurzen Zeitraum eine hohe Röntgenleistung erbringen können. Auf Grund dieser Anforderung werden in der Medizintechnik ausschließlich Drehanoden-Röntgenröhren [2.5] verwendet, die drehende Anode ermöglicht eine gleichmäßige Hitzeverteilung [4].

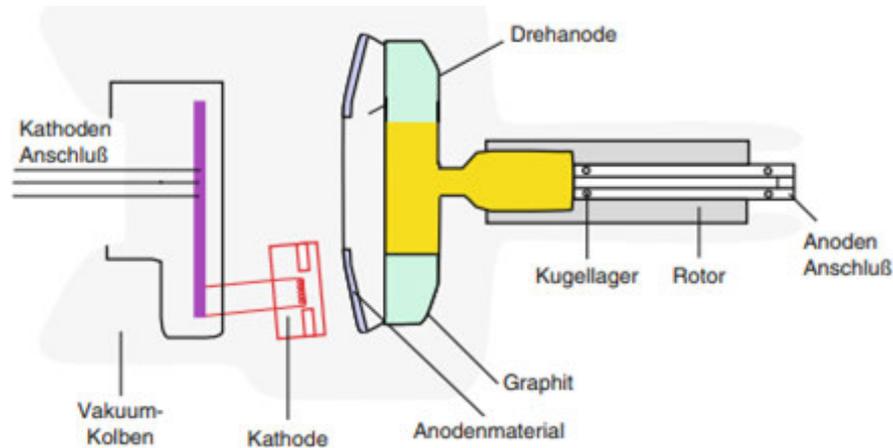


Abbildung 2.5: Schematische Darstellung einer Drehanoden-Röntgenröhre [4]

Der Aufbau einer Drehanoden-Röntgenröhre [2.5] besteht aus der Kathode und der Drehanode. Die Kathode besteht aus einem Vakuumpolben und einer Glühwendel, die aus Wolfram besteht. Die Kathode ist an eine Spannung  $U$ , die zwischen 80 kV und 150 kV liegt, angeschlossen. Durch den thermoelektrischen Effekt treten aus der Glühwendel Elektronen in den Vakuumpolben aus und werden in Richtung Anode beschleunigt. Die Drehanode besteht aus Molybdän, einem Graphitsockel und dem Anodenmaterial. Das Anodenmaterial besteht aus einer ca. 1 mm dicken Wolfram-Rhenium-Schicht. Treffen die beschleunigten Elektronen auf die Wolfram-Rhenium-Schicht, werden Röntgenstrahlen in

Form von Brems- und charakteristischer Strahlung emittiert. Beim Abbremsprozess der Elektronen im Anodenmaterial entsteht zudem Wärme. Um Hitzeschäden an der Anode zu vermeiden, ist diese an einem Drehlager befestigt, sodass die Wärme durch kontinuierliche Drehung auf der Anode verteilt wird.

### 2.1.4 Das Qualitätskriterium der Brennfleckgröße

In der Medizintechnik werden Anforderungen an ein Röntgensystem gestellt, um die bestmögliche Bildgebung zu erhalten. Dazu zählen unter anderem eine hohe Röntgenleistung und ein kleiner Brennfleck. Der thermische Brennfleck beschreibt die Stelle der Anode, in der die beschleunigten Elektronen auftreffen. Idealerweise ist der thermische Brennfleck scharf begrenzt, da er direkte Einflussnahme auf den optischen Brennfleck [2.6] nimmt und damit auf die Qualität der Röntgenaufnahme. Die Brennfleckgröße hängt von der Länge der Glühwendel sowie dem Anodenwinkel ab [2.6].

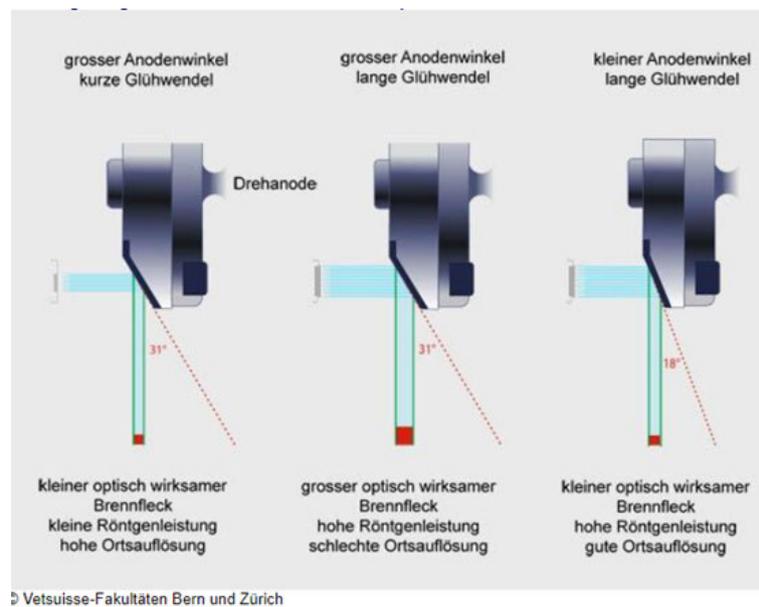


Abbildung 2.6: Schematische Darstellung des Strichfokusprinzips [36]

Ein großer thermischer und damit auch optischer Brennfleck kann in der Röntgenaufnahme für eine schlechte Ortsauflösung sorgen. Für eine gute Ortsauflösung wird ein kleiner optischer Brennfleck und eine hohe Röntgenleistung benötigt [36].

### 2.1.5 Systemtechnische Betrachtung

Die systemtheoretische Betrachtung eines Röntgensystems gibt Aufschluss über sein Verhalten. Ein abbildendes System hat einen Eingang und einen Ausgang. Ein Beispiel ist ein Röntgensystem mit Verstärkerfolie, der Eingang ist die Röntgendosis und der Ausgang der Film [4]. Im Allgemeinen lässt sich ein abbildendes System wie folgt beschreiben:

$$f(x, y) \rightarrow \boxed{\text{System}} \rightarrow g(x, y)$$

Lineare und verschiebungsinvariante Systeme lassen sich durch die Angabe der Impulsantwort  $h(x, y)$  charakterisieren, die Impulsantwort ist die Systemantwort auf einen Dirac-Impuls [4]. Somit ergibt sich für ein abbildendes System der Ausgang:

$$g(x, y) = f(x, y) * h(x, y)$$

Wobei  $g(x, y)$  die Bildaufnahme,  $f(x, y)$  das originale Abbild und  $h(x, y)$  die Impulsantwort beschreibt. Die Impulsantwort eines abbildenden Systems wird auch Punktspreizfunktion (engl.: Point-Spread-Function, kurz **PSF**) genannt. Die Abbildungseigenschaften eines abbildenden Systems lassen sich ausschließlich anhand der PSF charakterisieren [4]. Die PSF eines Röntgensystems gleicht dem optischen Brennfleck der Anode, Abbildung 2.7 zeigt eine berechnete PSF einer simulierten Röntgenaufnahme.

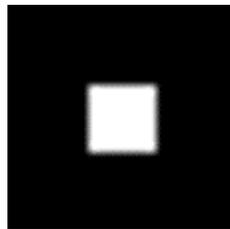


Abbildung 2.7: PSF, berechnet aus einer simulierten Röntgenaufnahme

Über die PSF lässt sich auf die Modulationstransferfunktion (kurz **MTF**) schließen. Die Modulationstransferfunktion beschreibt eine Kontrastübertragungsfunktion und wird zur Bestimmung der Abbildungsqualität eines abbildenden Systems genutzt.

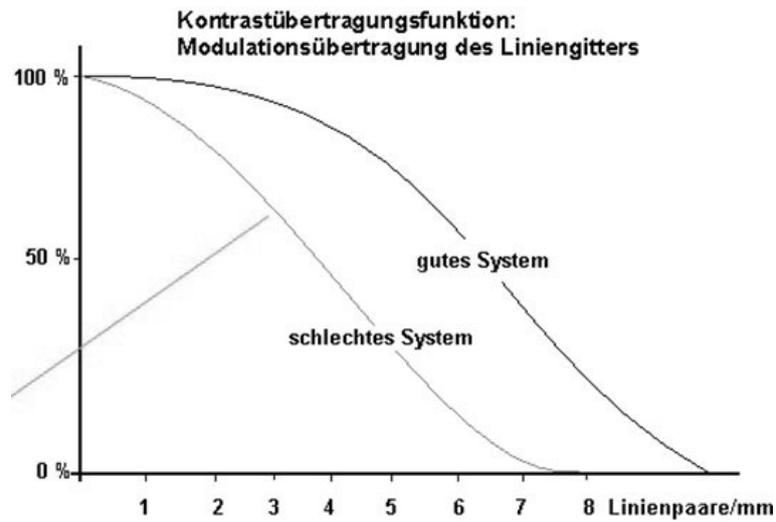


Abbildung 2.8: Kontrastübertragungsfunktionen eines guten und eines schlechten Systems [10]

Abbildung 2.8 zeigt die MTF, die Kontrastübertragung wird in Prozent angegeben und die Ortsfrequenz in Linienpaare/mm. Eine hohe Kontrastübertragung für eine hohe Linienpaarzahl pro Millimeter macht ein gutes System aus, folglich gibt der Kontrastverlust Aufschluss über die Abbildungsqualität eines abbildenden Systems. Da die Methoden zur direkten Messung der MTF nicht praktikabel sind [3], kann über die PSF die MTF berechnet werden. Zur Berechnung der MTF wird die gemessene PSF fouriertransformiert und der Absolutbetrag gebildet [4].

### 2.1.6 Der Siemensstern als Testmuster zur Prüfung der optischen Qualität eines Röntgensystems

Der Siemensstern [2.9] ist ein häufig verwendetes Testmuster zur Untersuchung des Auflösungsvermögens eines Röntgensystems.

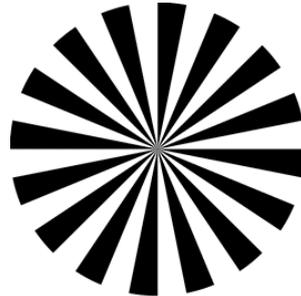


Abbildung 2.9: Siemensstern mit 16 schwarzen und 16 weißen Segmenten [26]

Der Siemensstern hat auf einen Mittelpunkt zulaufende Segmente, die abwechselnd schwarz und weiß sind. Die zur Mitte zunehmende Ortsfrequenz wird in Linienpaare pro Millimeter [Lp/mm] angegeben [26].

$$\text{Ortsfrequenz} \left[ \frac{LP}{mm} \right] = \frac{\text{Sektorenanzahl} [Lp]}{\pi \cdot d [mm]}$$

Insbesondere in dem mittleren Bereich ist das Auflösungsvermögen gut zu bestimmen, umso eindeutiger sich die Strukturdetails abzeichnen, umso höher ist das Auflösungsvermögen des Röntgensystems. Bei steigender Ortsfrequenz nimmt der Kontrast entsprechend der MTF des Systems ab und geht in einen Graubereich über, der sogenannte Grauring [2.10].

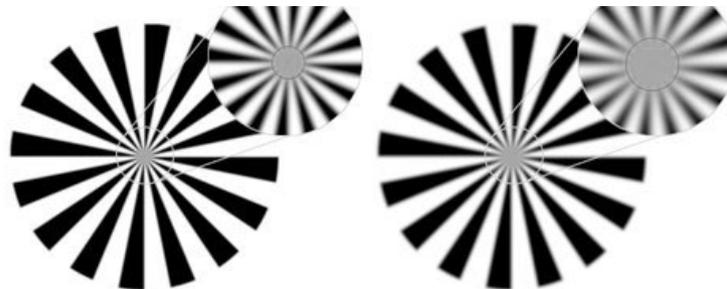


Abbildung 2.10: Siemenssterne mit Grauring für zwei Kontrastübertragungen [26]\*

Mittels des Durchmessers des Graurings kann das Auflösungsvermögen und der optimale Fokus eines Systems bestimmt werden.

### 2.1.7 Die Bestimmung der PSF aus einer Sternrastreraufnahme

Folgender Abschnitt basiert auf der Ausarbeitung „Ermittlung der PSF aus einer Sternrastreraufnahme mittels Methode kleinster Quadrate“ von Prof Dr. Heß [7] und beschreibt die theoretische Grundlage für den zu beschleunigenden Algorithmus. Zur Bestimmung der PSF wird ein ideales Sternraster und eine Bildaufnahme dieses Sternrasters benötigt, beide Abbildungen haben Auflösung von  $N$  Pixeln. Die Abbildungen lassen sich durch diskrete Funktionen beschreiben, wird die PSF mit der Abbildung des idealen Sterns gefaltet ergibt sich die Bildaufnahme des Sternrasters, veranschaulicht in Abbildung 2.11.

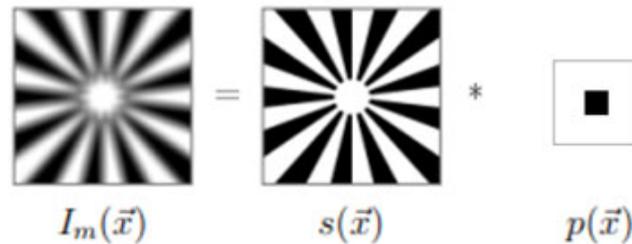


Abbildung 2.11: Das Gemessene Röntgenbild  $I_m(\vec{x})$  ergibt sich aus der Faltung des idealen Sterns  $s(\vec{x})$  und der PSF  $p(\vec{x})$

Wobei  $I_m(\vec{x})$  die Bildaufnahme des Sternrasters,  $s(\vec{x})$  das ideale Sternraster und  $p(\vec{x})$  die PSF beschreibt. Die PSF gilt es anhand des idealen Sterns und der Bildaufnahme zu bestimmen und wird als unbekannt angenommen. Die Bildaufnahme des Sternrasters  $I_m(\vec{x})$  lässt sich durch die Faltung des idealen Sternrasters  $s(\vec{x})$  mit der PSF  $p(\vec{x})$  berechnen:

$$I_m(\vec{x}) = s(\vec{x}) * p(\vec{x}) \quad (2.1)$$

$$= \sum_{k_j} s(\vec{k} - \vec{k}_j) \cdot p(\vec{k}_j) \quad (2.2)$$

mit  $\vec{k}$  = Wert im Idealen Stern und  $\vec{k}_j$  = Wert in der PSF.

Für die Bestimmung eines Pixels von  $I_m(\vec{k})$  wird die Summe über alle Pixel der PSF  $p(\vec{k}_j)$ , multipliziert mit bestimmten Werten des perfekten Sterns  $s(\vec{k} - \vec{k}_j)$  gebildet.  $I_b(k\vec{k})$

beschreibt das berechnete gefaltete Bild, um die Abweichung zwischen  $I_m(\vec{k})$  und  $I_b(\vec{k})$  zu minimieren wird die Standardabweichung der beiden Abbildungen beschrieben mit:

$$\begin{aligned} d &= \sqrt{\frac{1}{N}(I_b(\vec{k}) - I_m(\vec{k}))^2} \\ dN^2 &= \sum_{\vec{k}} (I_b(\vec{k}) - I_m(\vec{k}))^2 \\ &= \sum_{\vec{k}} \left( \sum_{k_j} s(\vec{k} - \vec{k}_j) \cdot (\vec{k}_j - I_m(\vec{k})) \right)^2 \end{aligned}$$

Wird die Standardabweichung nach dem Pixel  $p_i$  abgeleitet ergibt sich:

$$\begin{aligned} \frac{d}{dp_i} d^2 N &= \sum_{\vec{k}_j} p(\vec{k}_j) \sum_{\vec{k}} s(\vec{k} - \vec{k}_i) s(\vec{k} - \vec{k}_j) \\ &\quad - \sum_{\vec{k}} I_m(\vec{k}) s(\vec{k} - \vec{k}_i) = 0 \end{aligned}$$

Es ergibt sich ein Gleichungssystem mit dem Aufbau  $Ap = b$ . Das Gleichungssystem enthält  $M$  Gleichungen und pro Gleichung  $M$  Unbekannte, die aus der Anzahl der Pixel der PSF  $p(\vec{x})$  hervorgehen. Umgestellt ergibt sich für die Koeffizienten  $a_{ij}$ :

$$a_{ij} = \sum_{\vec{k}} s(\vec{k} - \vec{k}_i) s(\vec{k} - \vec{k}_j) \quad (2.3)$$

Und für die Konstanten  $b_i$ :

$$b_i = \sum_{\vec{k}} I_m(\vec{k}) s(\vec{k} - \vec{k}_i) \quad (2.4)$$

Das Gleichungssystem wird nach Gauss-Jordan gelöst und ergibt die Pixel der PSF. Die Bestimmung der PSF lässt sich in vier Schritte zusammenfassen:

1. Bestimmung der Position des Sternrasters in der Bildaufnahme  $I_m$
2. Generierung des idealen Sternrasters  $s$
3. Berechnung der Koeffizienten  $a_{ij}$  nach 2.3 und Konstanten  $b_i$  nach 2.4
4. Lösung des Gleichungssystems  $Ap = b$

## 2.2 Grundlagen der parallelen Programmierung

Die Parallelisierung von Software ermöglicht eine Erhöhung der Rechenleistung, da Prozessoren ohne weitere Anweisung Prozesse auf einem Prozessorkern ausführen. Ein Thread beschreibt einen Datenstrom eines Prozesses, der die Befehlsfolge abarbeitet [6]. Mehrkernprozessoren (engl.: Multi-Core Processor) eignen zur simultanen Ausführung von Threads eines Prozesses auf den Prozessorkernen, dieses Verfahren wird Multithreading genannt. Das folgende Kapitel behandelt die Grundlagen der parallelen Programmierung auf Software und Hardware Ebene.

### 2.2.1 Klassifikation nach Flynn's

Im Jahre 1972 schlug Michael J. Flynn eine Kategorisierung für die Klassifizierung der Anzahl paralleler Daten- und Kontrollflüssen vor [6]. Allgemein lassen sich Hardwarearchitekturen in folgende Klassen einteilen:

**Single Instruction, Single Data - SISD:** Die Single Instruction, Single Data Klasse (dt: Ein Befehl, ein Datenstrom) beschreibt sequenziell arbeitende Hardware, die weder Daten noch Kontrollflüsse parallel abarbeitet. Ein Beispiel ist ein Rechner, der nach der Von-Neumann Architektur aufgebaut ist [6].

**Single Instruction, Multiple Data - SIMD:** Die Single Instruction, Multiple Data Klasse (dt: Ein Befehl, mehrere Datenströme) beschreibt Hardware, die eine Anweisung auf mehreren Datenströmen ausführen kann. Grafikkarten, die über viele Rechenkerne verfügen, wenden einen Befehl auf beispielsweise einen Vektor an [6]. Durch Befehlssatzerweiterungen lassen sich auf einem Mehrkernprozessor SIMD Prozesse abarbeiten.

**Multiple Instruction, Single Data - MISD:** Bei der Klasse Multiple Instruction, Single Data (dt: Mehrere Befehle, ein Datenstrom) handelt es sich um ein theoretisches Konzept, es hat sich bislang nicht bei der Massenanfertigung von Hardware durchsetzen können, da für mehrere Befehle mehrere Datenströme benötigt werden. Die Umsetzung hat sich bislang als nicht effizient erwiesen [6].

**Multiple Instruction, Multiple Data - MIMD:** Die Klasse Multiple Instructions, Multiple Data (dt: Mehrere Befehle, mehrere Befehlsströme) beschreibt die Arbeitsweise moderner Mehrkernprozessoren. Mehrkernprozessoren sind imstande Befehle auf

mehreren Datenströmen auszuführen, diese Eigenschaft eines Prozessors wird als superskalar bezeichnet. Ein weiteres Beispiel sind PC-Cluster Systeme, die im Gegensatz zum Mehrkernprozessor einen verteilten Speicher haben [6].

### 2.2.2 Befehlssatzarchitektur

Die Befehlssatzarchitektur (engl: Instruction Set Architecture, ISA) beschreibt das Vokabular eines Prozessors, welches zur Kommunikation zwischen Software und Hardware genutzt wird. Die x86 Architektur gilt zu den weitverbreitetsten Befehlssatzarchitekturen und wird fortgehend weiterentwickelt und durch Befehlssatzerweiterungen ergänzt. Moderne Prozessoren arbeiten mit Befehlssätzen, die eine Länge von 64 Bit haben und sind in Abhängigkeit des Prozessormodells mit Befehlssatzerweiterungen kompatibel [2]. Folgende Befehlssatzerweiterungen sind für die parallele Programmierung von Interesse:

**Simultanes Multithreading - SMT:** Die SMT Befehlssatzerweiterung beschreiben eine Form des hardwareseitigen Multithreading, die mittels getrennter Pipelines mehrere Threads ausführt [29].

**Streaming SIMD Erweiterung - SSE:** Die SSE Befehlssätze ermöglichen die Parallelisierung auf Befehlsebene und ist speziell für Gleitkommazahl-Datentypen entwickelt. Mit der Befehlssatzerweiterung SSE können Befehle der Länge 128 Bit verarbeitet werden. Nach der Flynn'schen Klassifizierung entspricht die Parallelisierung der Klasse SIMD. Auf dieser Erweiterung aufbauend sind viele moderne Prozessoren mit den Erweiterungen SSE2, SSE3, SSE4.1, SSE4.2 und SSE4a [31] ausgestattet.

**Advanced Vector Erweiterung - AVX:** Bei AVX handelt es sich um eine Weiterentwicklung der Befehlssatzerweiterung SSE. Bei AVX werden 256 Bit Befehle verarbeitet, die modernsten Prozessoren sind imstande 512 Bit Befehle zu verarbeiten [32], dabei handelt es sich um die Befehlssatzerweiterung AVX512. Abbildung 2.12 zeigt den Unterschied zwischen der seriellen und der parallelen Ausführung von Befehlen durch Befehlssatzerweiterungen, wie SSE, AVX und AVX512.

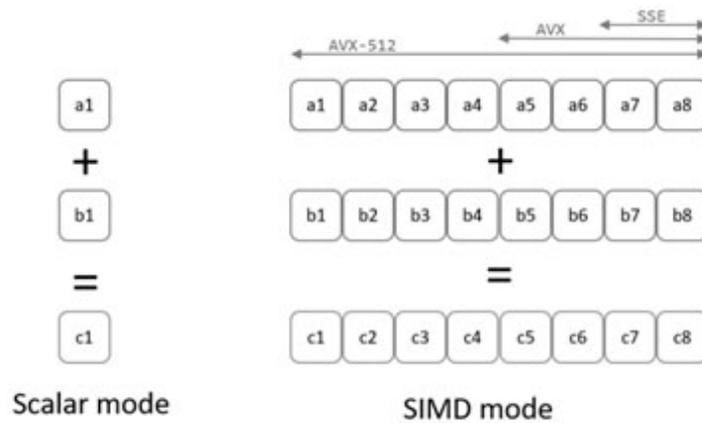


Abbildung 2.12: Beispiel für den Unterschied der seriellen und hardwareseitigen parallelen Ausführung von Rechenoperationen [2]

**Bit Manipulation Instruction – BMI:** Bei BMI handelt es sich um eine Befehlssatzerweiterung, die durch Bit Manipulationen Rechenaufgaben schneller oder mit reduzierter Codegröße ausführen kann, es gibt die Erweiterungen BMI1 und BMI2 [33, 20].

### 2.2.3 Pipelining

Die Pipeline-Architektur (dt: Fließband-Architektur) ermöglicht mehrere Befehlsabschnitte in einem Prozessortakt zu bearbeiten [2.13]. Hierzu wird ein Befehl in 5 Abschnitte unterteilt:

1. Holen
2. Decodierung
3. Ausführen
4. Speicherzugriff
5. Ergebnis schreiben



Abb. 6.7 Befehls-Pipelining

Abbildung 2.13: Beispiel für Befehls-Pipelining [6]

Im Takt  $t$  wird der erste Befehlsabschnitt *Holen* des Befehls 1 ausgeführt, im folgenden Takt  $t + 1$  geht der Befehl 1 decodiert während von Befehl 2 der Abschnitt *Holen* bearbeitet wird. Im Takt  $t + 4$  sind alle Register in Arbeit, da fünf Befehlsabschnitte gleichzeitig bearbeitet werden.

### 2.2.4 Out-Of-Order Ausführung

In der Praxis werden die Befehlsabschnitte Out-of-Order (dt.: nicht in Reihenfolge) ausgeführt (engl. Out-of-Order Execution, kurz: OOO). Bei der Out-of-Order Ausführung werden Befehle und Befehlsabschnitte nicht zwingend sequenziell abgearbeitet, sondern können in willkürlicher Reihenfolge abgearbeitet werden [2.14], die einzige Einschränkung stellen Abhängigkeiten zwischen den Befehlen dar. Die Out-Of-Order Ausführung trägt zu einer effizienten Auslastung des Prozessors bei. Die dynamische Abarbeitungsreihenfolge wird hardwareseitig durch Registerumbenennung [24] umgesetzt.

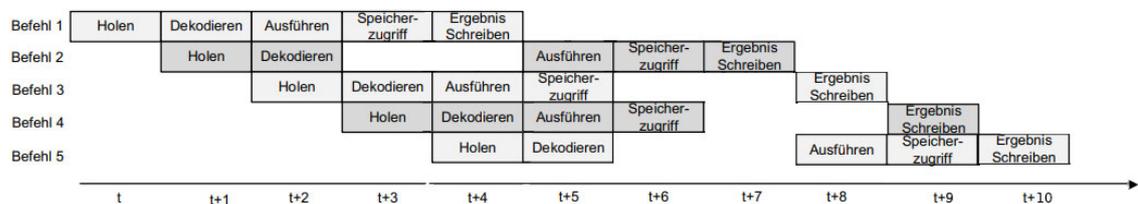


Abbildung 2.14: Beispiel für die Out-Of-Order Ausführung [6]

In Abbildung 2.14 ist ein Beispiel für die Out-Of-Order Ausführung von fünf Befehlen zu sehen. Befehl 1 wird ununterbrochen in fünf Takten abgearbeitet, Befehl 2 hingegen wird aufgrund eines Konflikts für zwei Takte ( $t + 3, t + 4$ ) angehalten. Befehl 3 weist in den Takten  $t + 4$  und  $t + 5$  keine Konflikte auf und kann vor Befehl 2 ausgeführt werden. Befehl 5 wird für 2 Takte ( $t + 6, t + 7$ ) angehalten, bevor er in  $t + 8$  weiter ausgeführt werden kann. Die Befehle werden in der sequenziellen Reihenfolge abgeschlossen, denn die Ergebnisse werden in Reihenfolge geschrieben.

### 2.2.5 Superskalare Ausführung

Moderne Prozessoren sind häufig imstande zwei bis sechs Befehle gleichzeitig zu bearbeiten, diese Eigenschaft nennt sich superskalar.



Abbildung 2.15: Beispiel für die superskalare Ausführung [6]

In Abbildung 2.15 wird die Befehlsausführung auf einem 2-Weg Superskalaren Prozessor dargestellt, Befehl 1 und 2 werden in den Takten  $t$  bis  $t + 4$  abgearbeitet, Befehl 3 und 4 werden in den Takten  $t + 1$  bis  $t + 5$  gleichzeitig abgearbeitet. Häufig werden die superskalare und die Out-of-Order Ausführung kombiniert [2].

### 2.2.6 Spekulative Ausführung

Bei der spekulativen Ausführung (engl. Branch Prediction) werden Befehlsabschnitte spekulativ gestartet, auch Sprungzielvorhersage genannt. Der Reorder-Buffer (ROB) überwacht die Zustände der Befehle und gleicht den spekulativen Ablauf mit dem tatsächlichen ab. Sobald festgestellt wird, dass die Spekulation falsch ist, wird der Ablauf abgebrochen [2].

### 2.2.7 Aufbau und Arbeitsweise eines modernen Mehrkernprozessors

Der Mehrkernprozessor verfügt über mehrere Prozessorkerne (engl. Cores), die es ermöglichen mehrere Prozesse gleichzeitig zu bearbeiten. Abbildung 2.16 zeigt den grundlegenden Aufbau eines Mehrkernprozessors:

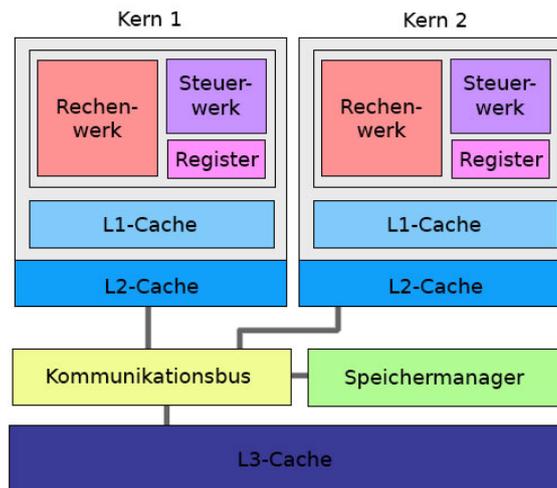


Abbildung 2.16: Schematische Darstellung des Aufbaus eines Mehrkernprozessors

Die Prozessorkerne verfügen über ein Rechenwerk, ein Steuerwerk und Register. Bei Mehrkernprozessoren verfügen die Prozessorkerne über eigene L1 und L2 Cache Speicher. Der Kommunikationsbus, der von allen Kernen genutzt wird, koordiniert mit dem Speichermanager den L3 Cache Speicher. Im Allgemeinen lässt sich eine CPU in Front-End und Back-End unterteilen. Im Folgenden wird die Funktion eines Mehrkernprozessors am Beispiel eines AMD Ryzen 7 3700X [2.17] genauer erläutert.

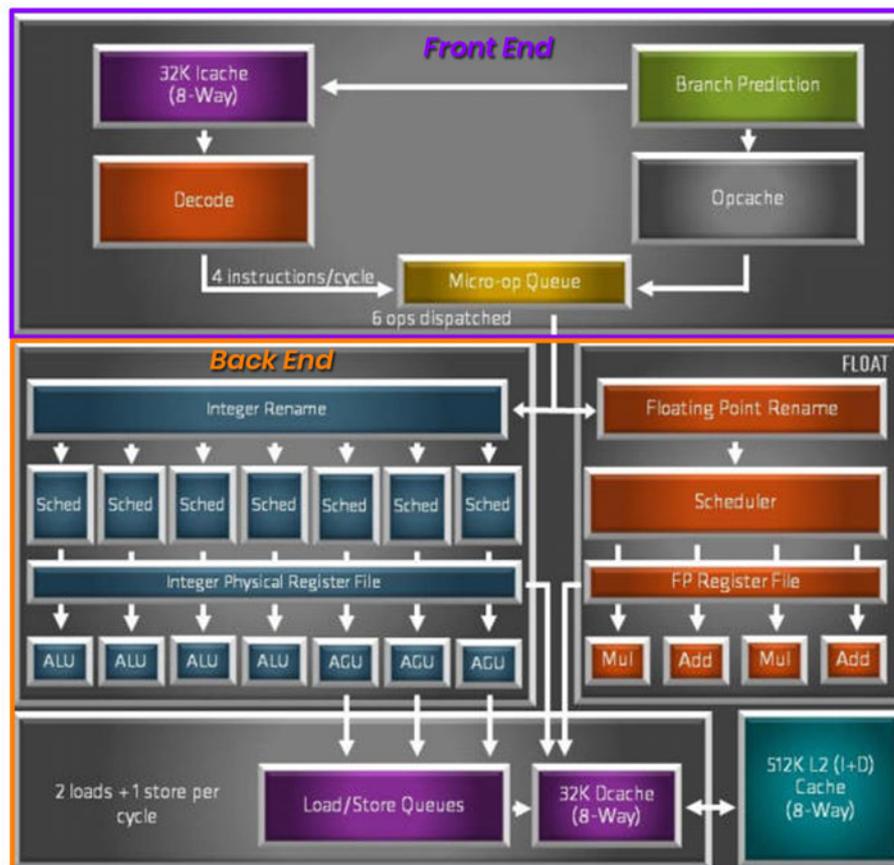


Abbildung 2.17: Aufbau eines AMD Ryzen 7 3700X Prozessors[18]

Das Front-End hat die Hauptaufgabe, Befehle zu laden und zu decodieren. Die decodierten Befehle werden an das Back-End weitergeleitet [13]. Genauer betrachtet werden aus dem L1-Befehls-Cache x86 Befehle geladen, decodiert und in Mikrobefehle (engl.: Micro-Ops) konvertiert. Die Mikrobefehle werden in die Mikrobefehl-Warteschlange (engl. Micro-Op Queue) eingereiht und von dort an das Back-End weitergegeben. Es gibt neben dem L1-Befehls-Cache einen OpCache, in dem die Übersetzung einiger Befehle in Mikrobefehle gespeichert sind. Der OpCache wird während des Ladens der Befehle auf Übersetzungen geprüft, die schnell geladen werden können, um den Decodiervorgänge zu sparen. Um die Pipeline effizient zu nutzen, werden von der Branch Prediction Unit Sprungvorhersagen

getätigt.

Die Hauptaufgabe des Back-Ends ist die Ausführung und das Abspeichern der Befehle [2]. Der Ryzen 7 3700X verfügt neben Recheneinheiten für Integer Berechnung auch gesonderte Recheneinheiten, die speziell für Gleitkommazahl Berechnungen vorhanden sind. Die Mikrobefehle werden in die Registerumbenennungseinheiten (engl.: Integer Rename, Float-Point Rename) geladen, dies erleichtert die Out-Of-Order Ausführung des Back-End [18]. Die Scheduler übernehmen Koordination der Mikrobefehle. Die Mikrobefehle werden an die arithmetisch-logischen Einheiten (ALU) weitergegeben und bearbeitet. Die Ergebnisse werden im L1-DCache oder im L2-Cache gespeichert [2].

### 2.3 Grundlagen OpenMP

Die OpenMP Bibliothek bietet eine Programmierschnittstelle zur Parallelisierung von C, C++ und Fortran Anwendungen. Die Parallelisierung mittels OpenMP ist besonders praktikabel, da die Umsetzung mit pragma-Direktiven und Bibliotheksfunktionen möglich ist und somit Änderungen am Quellcode meist minimal sind [16].

#### 2.3.1 Das parallele Ausführungsmodell von OpenMP

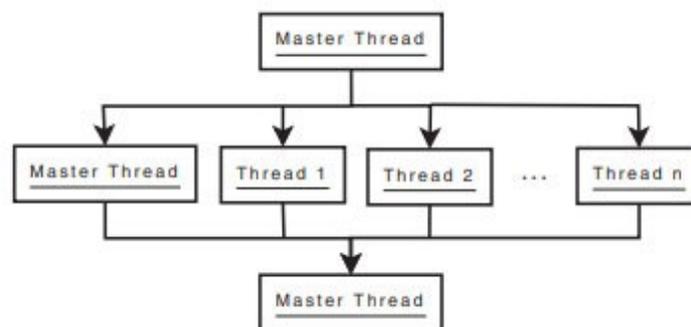


Abbildung 2.18: Das Fork-Join Ausführungsmodell von OpenMP [16]

Zu Beginn des Programms ist der Master Thread aktiv, trifft dieser auf eine OpenMP Direktive wird ein paralleler Abschnitt gestartet. Der Programmcode wird auf  $n$  Threads aufgeteilt (fork) und parallel ausgeführt. Ist der Abschnitt ausgeführt, werden die Threads synchronisiert und wieder zusammengeführt (join). Der langsamste Thread bestimmt

die Gesamtgeschwindigkeit eines parallelen Abschnittes, da die Threads ausschließlich synchronisiert werden können, wenn alle Threads ihren Abschnitt abgearbeitet haben.

### 2.3.2 Parallelisieren von Schleifen

Mit der pragma-Direktive `#pragma omp parallel for` können for-Schleifen parallelisiert werden. Die Iterationen der Schleife werden auf die verfügbaren Threads aufgeteilt und ausgeführt [16]. Im Folgenden werden die wichtigen Aspekte zur Parallelisierung von Schleifen mit OpenMP näher erläutert.

#### Datenzugriffsklauseln und Kommunikation der Threads

Als *shared* werden Variablen bezeichnet, auf die die Threads gemeinsam lesend und schreibend zugreifen. In einigen parallelen Abschnitten werden *private* Variablen benötigt, bei der jeder Thread eine private Kopie der Variable anlegt und mit dieser arbeitet. *Private* Variablen werden zufällig initialisiert, soll dies vermieden werden, kann eine *firstprivate* Variable genutzt werden, bei der von den privaten Kopien der letzte bekannte Wert vor dem parallelen Abschnitt übernommen wird [16].

#### Ablaufpläne

Die Aufteilung der Iterationen einer Schleife auf die Threads können mit Ablaufplänen (engl.: schedule) beeinflusst werden. Es wird zwischen statischen und dynamischen Ablaufplänen unterschieden. Bei einem statischen Ablaufplan wird jedem Thread eine bestimmte Anzahl an Iterationen zugeteilt. Die Zuteilung steht in direktem Zusammenhang mit der Anzahl der verfügbaren Threads  $p$  und der Anzahl der zu verteilenden Iterationen  $n$ . Bei einem dynamischen Ablaufplan werden die Iterationen anfangs nicht fest verteilt, sondern jedem Thread eine Anzahl  $c$  (Chunk) zugewiesen und sobald ein Thread seine Iterationen abgearbeitet hat, werden weitere Iterationen zugewiesen. Insgesamt sind die Threads dadurch besser ausgelastet und warten nicht auf jene Threads, die ihre Iterationen noch nicht abgearbeitet haben. Ein weiterer Ablaufplan ist *guided*, bei dem die Stückgrößen  $c$  bei jeder Zuweisung exponentiell kleiner wird. Ist der Ablaufplan auf *auto* gesetzt, wählt der Compiler einen Ablaufplan [16]. Eine Veranschaulichung der Ablaufpläne zeigt Abbildung 2.19.

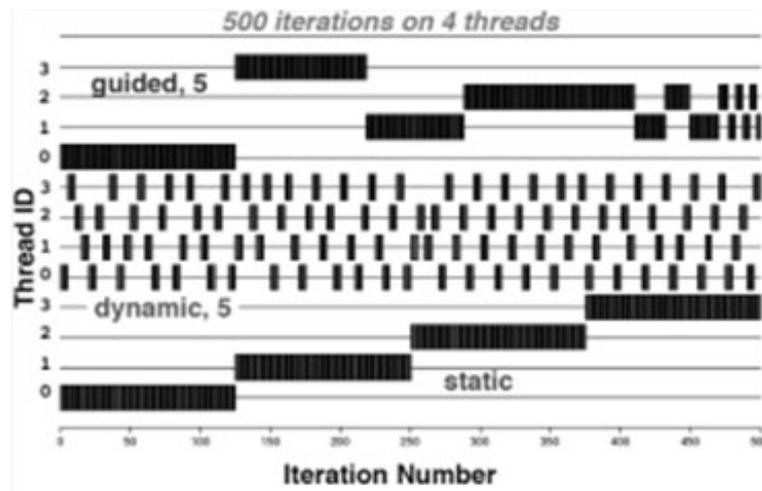


Abbildung 2.19: Ablaufpläne für 500 Iterationen auf 4 Threads [1]

### Umgang mit Datenabhängigkeiten

Datenabhängigkeiten können Ergebnisse von parallelen Programmen verfälschen, sie lassen sich in drei Kategorien einordnen [16, 25]:

**Echte Datenabhängigkeiten:** Eine echte Datenabhängigkeit ist gegeben, wenn eine Anweisung eine Speicherstelle ausliest, die von der vorherigen Anweisung beschrieben wurde:

$$a = b + c$$

$$d = a + 1$$

**Gegenabhängigkeiten:** Eine Gegenabhängigkeit ist gegeben, wenn eine Anweisung in eine Speicherstelle schreibt, die in der vorherigen Anweisung ausgelesen wurde:

$$a = b + c$$

$$b = d + 1$$

**Ausgabeabhängigkeiten:** Eine Ausgabeabhängigkeit besteht, wenn zwei Anweisungen aufeinander folgend in dieselbe Speicherstelle schreiben:

$$a = b + c$$

$$a = d + e$$

Datenabhängigkeiten sollten, wenn möglich, vor der Parallelisierung entfernt werden. Anderweitig können bestimmte Codeabschnitte innerhalb eines parallelen Abschnitts mit der pragma-Direktive `#pragma omp single` zwingend iterativ auf einem Thread abgearbeitet werden.

### Synchronisation

Bei der parallelen Ausführung von Programmen kann es zu Wettlaufsituationen kommen, bei denen von mehreren Threads auf die gleiche Speicherstelle zugegriffen wird. Um gemeinsam genutzte Daten konsistent zu halten, müssen die Daten synchronisiert werden. Die Synchronisation kann durch folgende Direktiven durchgeführt werden:

**# pragma omp critical:** Kritische Code-Abschnitte, die durch die *critical* Direktive eingeleitet werden, dürfen ausschließlich zu jedem Zeitpunkt von einem Thread ausgeführt werden.

**# pragma omp atomic** Mit der *atomic* Direktive lässt sich eine Speicherstelle für den parallelen Zugriff sperren. Es darf zu jedem Zeitpunkt niemals mehr als ein Thread auf die Variable zugreifen.

## 2.4 Grundlagen der Software-Optimierung

Die Effizienz von Software hängt von dem Betriebssystem bis hin zum Prozessor ab und lässt sich unter anderem anhand der Laufzeit bestimmen. Oft kann eine Leistungssteigerung durch das einfache Ausschöpfen der Hardwareressourcen und durch Compileroptimierungen mittels Compiler-Flags (dt: Compileranweisungen) erwirkt werden. Im Folgenden werden einige Begrifflichkeiten und Methoden zur Analyse und Leistungsmessung sowie der Optimierung von Software erläutert.

### 2.4.1 Sampling zur Leistungsanalyse von Software

Die Leistungsanalyse (engl. Profiling) von Software ist essenziell zur Softwareoptimierung. Zur Leistungsanalyse werden Daten von Soft- und Hardware gesammelt, wodurch sich Engpässe (engl.: Bottlenecks) identifizieren lassen. Eine Leistungsanalyse kann wie folgt ablaufen [2]:

- 1.Sampling:** Beim Sampling (dt. Stichprobennahme) wird die Ausführung der Software von einem Profiler (dt. Leistungsanalysetool) überwacht. Der Profiler sammelt Daten zu der Ausführung des Codes. Beim User-Based Sampling überwacht der Profiler die Software und sammelt periodisch Daten auf welche Art und Weise der Code ausgeführt wird. Das User-based Sampling liefert einen oberflächlichen Überblick zum Verhalten der Software, der es ermöglicht Engpässe zu identifizieren. Der Overhead beim User-Based Sampling beläuft sich auf ca. 5% [2].
- 2.Engpässe identifizieren:** Anhand der von Profiler gesammelten Daten können Optimierungspotentiale und Engpässe identifiziert werden [2].
- 3.Aufrufstapel analysieren:** Bei der Leistungsanalyse kann es dazu kommen, dass Engpässe durch verschiedene Aufrufquellen entstehen. Um diese näher zu untersuchen, werden Aufrufstapel (engl.: Call Stacks) vom Profiler angelegt. Aufrufstapel geben einen Einblick über das Aufrufverhalten einzelner Funktionen und deren Quellen [2].

### 2.4.2 Optimierungseinstellungen des GNU Compilers

Moderne Compiler verfügen über Optimierungseinstellungen, die mittels Compiler-Flags aktiviert werden. Ziel der Compiler Optimierungen ist die Reduzierung der auszuführenden Befehle. Folgende vordefinierte Optimierungslevel stellt der GNU Compiler zur Verfügung [5]:

- O0:** Bei -O0 handelt es sich um die Anweisung zur Standardkompilierung, es finden keine Optimierungen statt.
- O1:** Bei den -O1 Optimierungen werden Standardoptimierungen, die minimalen Mehraufwand an Daten und Kompilierzeit erzeugen, angewandt. Die Laufzeit und die Anwendungsgröße werden reduziert.

**O2:** Bei den -O2 Optimierungen werden zusätzlich zu den -O1 Optimierungen, Optimierungen wie beispielsweise die Out-Of-Order Ausführung angewandt. Die -O2 Optimierungen erzielen eine bessere Laufzeit, ohne die Anwendungsgröße zu vergrößern. Das Kompilieren dauert länger als bei dem Optimierungslevel O1

**O3:** Bei den -O3 Optimierungen werden zusätzlich zu den -O2 Optimierungen, Optimierungen wie beispielsweise die Aktivierung von AVX Befehlssatzerweiterungen angewandt. Es entsteht ein Mehraufwand an Daten und Kompilierzeit, die Laufzeit hingegen wird stark reduziert.

**Ofast:** Bei den -Ofast Optimierungen werden zusätzlich zu den -O3 Optimierungen, radikale Optimierungen angewandt, bei der die mathematische Korrektheit der Software nicht garantiert werden kann, es kommt zu einer höheren Kompilierzeit aber zu einer verbesserten Laufzeit.

**Os:** Bei den -Os Optimierungen liegt der Fokus auf der Reduzierung der Anwendungsgröße.

**Og:** Bei -Og Optimierungen liegt der Fokus auf schneller Kompilierzeit und dem Debuggen.

Weitere Compiler-Flags sind:

**march = native:** Ist die Compiler-Flag -march = native gesetzt werden Optimierungen vorgenommen, die auf dem Modell des Prozessors basieren. Dies birgt das Risiko, dass die Anwendung nicht auf Rechenystemen mit anderen Prozessoren läuft.

**DNDEBUG:** Die Ausgabe von Zusicherungen ist ausgeschaltet [15].

## 2.5 Verwendete Software

Im Folgenden wird auf die Software eingegangen, die zur Umsetzung der Aufgabenstellung genutzt wird.

### 2.5.1 VSCode

Bei VSCode handelt es sich um einen kostenfreien Quelltexteditor der Firma Microsoft. VSCode ist mit Windows und Linux Betriebssystemen kompatibel und unterstützt die meistgenutzten Programmiersprachen wie zum Beispiel C, C++ und Python [24]. VSCode bietet Programmierer\*innen einen schnellen, anpassbaren Quelltexteditor mit Kompilier- und Debugging Oberfläche für den alltäglichen Gebrauch. VSCode ist Open-Source, wird kontinuierlich weiterentwickelt und bietet einen Marketplace mit Erweiterungen [3]. Für die Funktionalität des Quellcodes werden die CMake, CMake Tools und die C++ Sprachpaket Erweiterungen benötigt. VSCode wird mit der Version 1.66 verwendet.

### 2.5.2 Very Sleepy

Der Profiler Very Sleepy ist ein Sampling Profiler, welcher den Zustand der Software und die Auslastung des Prozessors periodisch betrachtet [11]. Very Sleepy sammelt Daten wie beispielsweise inklusive und exklusive Laufzeit von Funktionen und Aufrufstapel.

### 2.5.3 OpenMP Bibliothek

Für die Parallelisierung der Software auf dem Prozessor wird die Bibliothek OpenMP genutzt. OpenMP ist mit den Programmiersprachen C, C++ und Fortran, den Betriebssystemen Windows und Linux und unter anderem dem GNU Compiler kompatibel [28].

### 2.5.4 GNU Compiler

Bei der GNU Compiler Collection (kurz: GCC) handelt es sich um einen Windows und Linux kompatiblen Compiler. GCC wurde erstmals im Jahr 1987 als C Compiler veröffentlicht und kann nach aktuellem Stand (04/2022) unter anderem die Programmiersprachen C, C++ und Fortran übersetzen [35]. Es wird die Version 11.2.0 64Bit des GNU Compilers verwendet.

### 2.5.5 Doxygen

Doxygen ist ein Software-Dokumentationswerkzeug, das aus C++ Code eine HTML Dokumentation anfertigen kann. Doxygen ist mit Windows und Linux Betriebssystemen kompatibel. Die Generierung einer Dokumentation umfasst unter anderem Call-Diagrammen und Klassendiagramme [27].

## 2.6 Verwendete Hardware

Die Laufzeitmessungen werden vorwiegend auf einem Rechner mit einem AMD Ryzen 7 3700X Prozessor vorgenommen, hier 8-Kerne-Messsystem genannt. Um das Verhältnis der Ergebnisse auf anderen Rechnern einschätzen zu können, wird die beschleunigte Version auf folgenden Rechnern getestet:

Bezeichnung	Prozessormodell	Kerne	log. Prozessoren	Takt[GHz]
2-Kerne Messsystem	Intel Core i3-6100U	2	4	2,3
6-Kerne Messsystem	AMD Ryzen 5 2600	6	12	3,4 - 3,9
8-Kerne Messsystem	AMD Ryzen 7 3700x	8	16	3,6 - 4,4

Tabelle 2.1: Messsysteme

Bei dem 2-Kerne-Messsystem handelt es sich um einen Laptop des Herstellers Acer aus dem Jahr 2015. Das 2-Kerne-Messsystem verfügt über einen Intel i3 Prozessor der sechsten Generation, welcher über 2 Kerne und 4 logische Prozessoren verfügt. Der Basistakt liegt bei 2,30 GHz, über einen Turbotakt verfügt der i3 Prozessor nicht [21].

Das 6-Kerne-Messsystem verfügt über einen AMD Ryzen 5 2600 Prozessor aus dem Baujahr 2019, es handelt sich um einen selbst zusammengestellten PC. Der AMD Ryzen 5 2600 verfügt über 6 Kerne, 12 logische Prozessoreinheiten und läuft in dem Basistakt 3,40 GHz. Der Turbotakt liegt bei 3,9 GHz [23].

Bei dem 8-Kerne-Messsystem handelt es sich um ein eigen konfigurierten Gaming-PC aus dem Jahr 2020. In diesem System ist ein AMD Ryzen 7 3700X Prozessor aus dem Jahr 2019 verbaut. Der Ryzen 7 3700X Prozessor verfügt über 8 Kerne und 16 logische Prozessoren. Der Basistakt liegt bei 3,60 GHz und kann bis maximal 4,40 GHz getaktet werden [22].

## 3 Anforderungen

Die Art der Anforderungen an die beschleunigte Software lassen sich in Leistung, Qualität und Komptabilität unterteilen. Die Leistungsanforderung dieser Ausarbeitung liegt in der Beschleunigung der Software. Die Beschleunigung sollte mindestens einen Faktor 10 auf den Messsystemen für die Bildgröße 3000x3000 Pixel und die Größe der PSF 30x30 Pixel erreichen. Eine optionale Zielsetzung ist die Beschleunigung für die oben genannten Bildgrößen auf unter 5 s auf den Messsystemen.

Die Qualität der Berechnung der PSF wird durch die Standardabweichung beschrieben. Die Standardabweichung der beschleunigten Software muss kleiner als  $1 \times 10^{-9}$  sein, um die Qualität der Berechnung gewährleisten zu können.

Die Maßnahmen zur Beschleunigung der Software müssen mit Linux- und Windows Betriebssystemen kompatibel sein. Des Weiteren müssen die Optimierungsmaßnahmen auf Intel-sowie AMD Prozessoren anwendbar sein.

## 4 Design

Im folgenden Kapitel wird der Ausgangszustand der von Prof. Dr. Heß zur Verfügung gestellten Software aufgenommen und im Rahmen der Beschleunigung der Laufzeit analysiert. Anhand der Analyse wird ein Design zur Implementierung der Parallelisierung mit OpenMP entwickelt. Darüber hinaus werden Compiler-Konfigurationen für die Compileroptimierungen definiert.

### 4.1 Programmablaufanalyse

Die Bestimmung der PSF lässt sich in folgende Schritte unterteilen [7]:

1. Bestimmung der Position des Sternrasters in der Bildaufnahme  $I_m$
2. Generierung des idealen Sternrasters  $s$
3. Berechnung der Koeffizienten  $a_{ij}$  nach 2.3 und Konstanten  $b_i$  nach 2.4
4. Lösung des Gleichungssystems  $Ap = b$

Die Programmablaufanalyse basiert auf der Ausgangsversion der Software von Prof. Dr. Heß. Das Call-Diagramm in Abbildung 4.1 zeigt die Funktionsaufrufe bei Ausführung der Software.

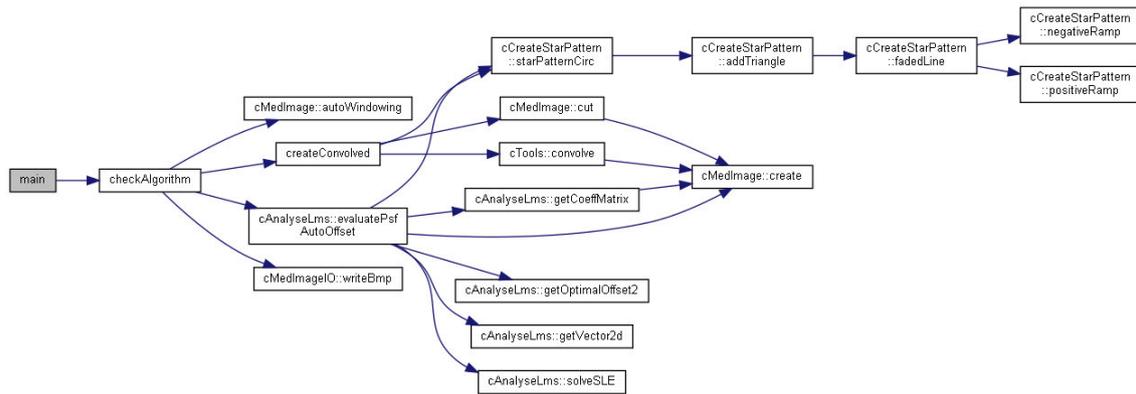


Abbildung 4.1: Call-Diagramm des Ausgangszustand, erstellt mit Doxygen

Die Bestimmung der Lage und Ausrichtung ist durch die Funktion *autoWindowing* implementiert. Im Rahmen dieser Ausarbeitung wird keine Bildaufnahme eines Röntgengerätes verwendet, stattdessen wird eine simulierte Röntgenaufnahme durch die Funktion *createConvolved* generiert. Die Bildgröße der simulierten Röntgenaufnahme kann in der Funktionsdefinition der Funktion *CheckAlgorithm* angepasst werden. Die Berechnung der PSF ist in der Klasse *AnalyseLMS* implementiert und durch die Funktion *evaluatePsfAutoOffset* zusammengefasst. Wie in Schritt zwei beschrieben wird der ideale Stern *s* aus der Bildaufnahme berechnet. Daraufhin folgt die Bestimmung der Koeffizienten, die Berechnung ist in der Funktion *getCoeffMatrix* mit zwei Methoden implementiert. Bei den Methoden zur Bestimmung der Koeffizientenmatrix handelt es sich um die direkte Implementierung und der optimierten Methode (Anhang A.6). Die Bestimmung der Konstanten ist in der Funktion *evaluatePsfAutoOffset* definiert. Sind die Koeffizienten und Konstanten bestimmt, wird das Gleichungssystem mit der Funktion *solveSLE* gelöst.

## 4.2 Leistungsanalyse des Ausgangszustandes

Die Leistungsanalyse wird mit Very Sleepy durchgeführt. Die Leistungsanalyse wird mit folgender Konfiguration durchgeführt:

Bildgröße <i>conv</i>	3000x3000 Pixel
Bildgröße <i>psf</i>	30x30 Pixel

Die Leistungsanalyse für die direkte Implementierung ergibt:

Functions						
Name	Excl...	Inclusive	% E...	% Inclus...	Module	Source File
cAnalyseLms::getCoeffMatrix	2390.88s	2653.75s	89.75%	99.61%	BA-OMP...	[unknown]
cMedImage<double>::operator[]	264.14s	264.14s	9.91%	9.91%	BA-OMP...	[unknown]
cAnalyseLms::evaluatePsfAutoOffset	6.69s	2663.30s	0.25%	99.97%	BA-OMP...	[unknown]
cAnalyseLms::solveSLE	1.66s	1.66s	0.06%	0.06%	BA-OMP...	[unknown]
cTools::convolve	0.26s	0.34s	0.01%	0.01%	BA-OMP...	[unknown]

Abbildung 4.2: Leistungsanalyse der direkten Implementierung

Es wird deutlich, dass die Funktion *getCoeffMatrix* einen Anteil von 89,75% der Gesamtlaufzeit von 2664,11s ausmacht. Darüber hinaus wird deutlich, dass 9,91% der Gesamtlaufzeit mit Klassenoperationen in *CMedImage* Objekten verbracht werden. Die Leistungsanalyse für die optimierte Methode ergibt:

Functions						
Name	Excl...	Inclusive	% Excl...	% Inclus...	Module	Source File
cAnalyseLms::getCoeffMatrix	7.62s	10.58s	37.16%	51.58%	BA-OMP...	[unknown]
cAnalyseLms::evaluatePsfAutoOffset	6.68s	20.10s	32.56%	98.02%	BA-OMP...	[unknown]
cMedImage<double>::operator[]	4.18s	4.18s	20.40%	20.40%	BA-OMP...	[unknown]
cAnalyseLms::solveSLE	1.67s	1.67s	8.14%	8.14%	BA-OMP...	[unknown]
cTools::convolve	0.26s	0.34s	1.26%	1.66%	BA-OMP...	[unknown]

Abbildung 4.3: Leistungsanalyse der optimierten Methode

Für die optimierte Methode macht die Funktion *getCoeffMatrix* einen Anteil von 37,16% der Gesamtlaufzeit von 20,52s aus. Die Funktion *evaluatePsfAutoOffset* hat eine exklusive Laufzeit von 6,68s, dies entspricht 32,56% der Gesamtlaufzeit. Die Funktion *solveSLE* macht mit 1,67s 8,14% der Gesamtlaufzeit aus.

Um das Laufzeitverhalten in Abhängigkeit der Eingangsgrößen *conv* und *psf* zu untersuchen, werden Messungen bei variierender Bildgröße der simulierten Röntgenaufnahme *conv* mit folgender Konfiguration vorgenommen:

Bildgröße <i>conv</i>	variiert
Bildgröße <i>psf</i>	30x30 Pixel
Methode	1,3
Stichprobenmenge	10 pro Bildgröße

Abbildung 4.4 zeigt die Laufzeit in Sekunden[s] für eine variierende Bildgröße der simulierten Röntgenaufnahme *conv*:

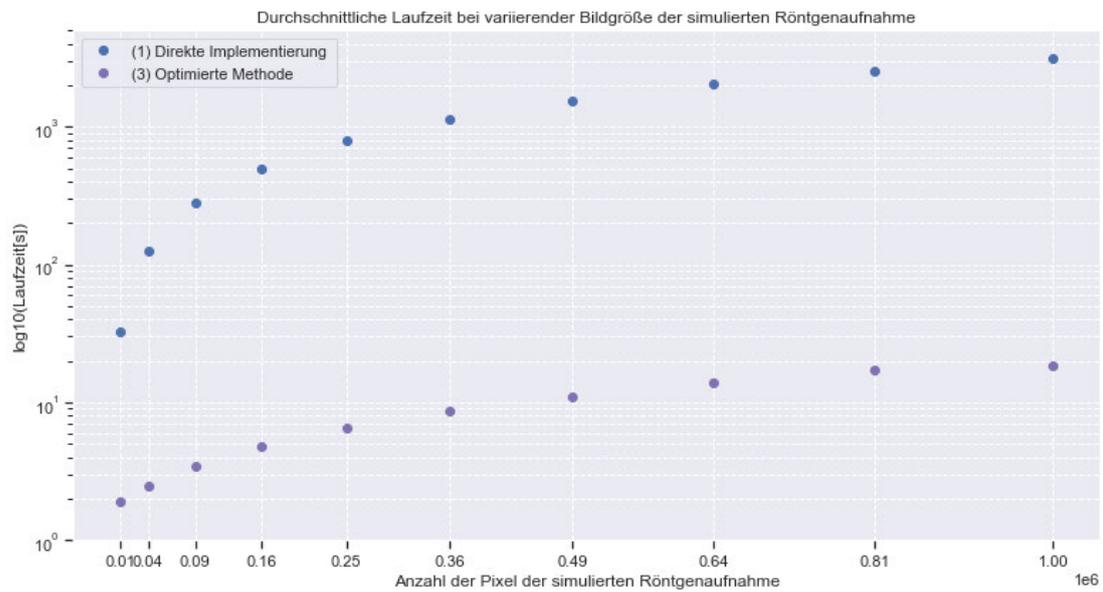


Abbildung 4.4: Laufzeitverhalten bei variierender Bildgröße der simulierten Röntgenaufnahme

Die genauen Messergebnisse befinden sich im Anhang A.1, einige Werte wurden aufgrund ihrer Dauer für die direkte Implementierung über die Steigung approximiert.

Die Messungen für eine variierende Bildgröße der PSF  $psf$  wird mit folgender Konfiguration vorgenommen:

Bildgröße $conv$	100x100 Pixel
Bildgröße $psf$	variiert
Methode	1,3
Stichprobenmenge	10 pro Bildgröße

Abbildung 4.5 zeigt die Laufzeit in Sekunden[s] für eine variierende Bildgröße der PSF  $psf$ :

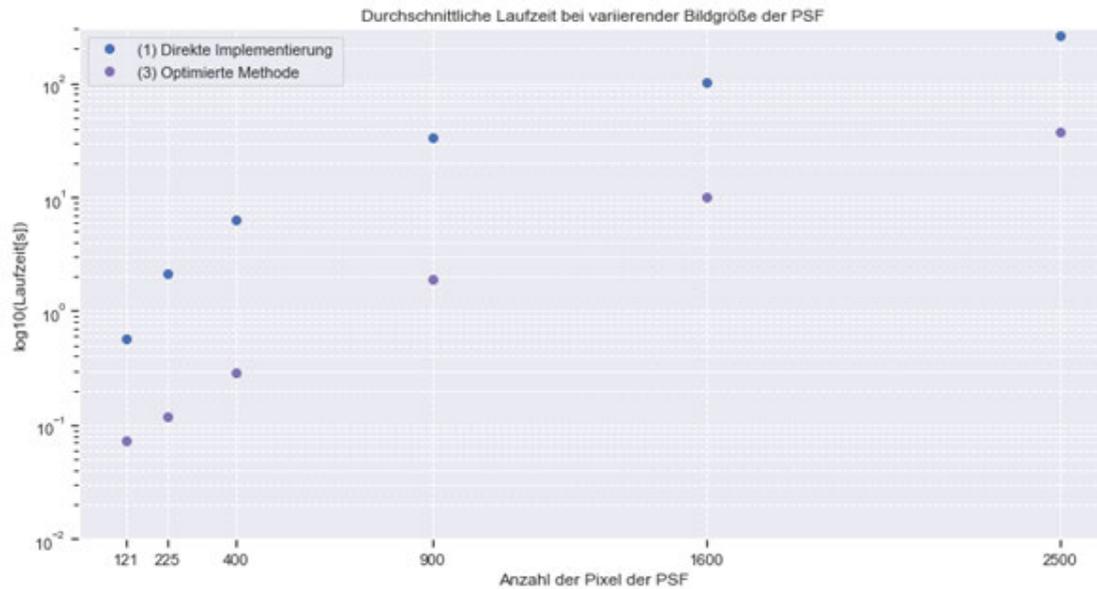


Abbildung 4.5: Laufzeitverhalten bei variierender Bildgröße der PSF

Es wird deutlich, dass die Laufzeit der Software direkt in Abhängigkeit mit der Anzahl der Gesamtpixel der simulierten Röntgenaufnahme und der Gesamtpixel der PSF steht. Die genauen Messergebnisse befinden sich im Anhang A.1 Aus den Leistungsanalysen geht hervor, dass die Funktion *getCoeffMatrix* ein Engpass ist und von einer Parallelisierung profitieren könnte. Zusätzlich könnte die Laufzeit durch die Parallelisierung der Funktionen *evaluatePsfAutoOffset* und *solveSLE* beschleunigt werden.

### 4.3 Beschleunigung durch Parallelisierung

Die Leistungsanalyse gibt Aufschluss über das Laufzeitverhalten der Software. Aus den Messungen geht hervor, dass die Funktionen *getCoeffMatrix*, *evaluateAutoOffset* und *solveSLE* das größte Beschleunigungspotential aufweisen. OpenMP bietet die Möglichkeit die Anzahl der Threads, die zur parallelen Berechnung genutzt werden sollen, sowie den Ablaufplan zu beeinflussen. Diese Parameter sollen zentral steuerbar sein, Abbildung 4.7 zeigt einen Zustandsautomaten der Software und die angedachte Platzierung der Parallelisierung:

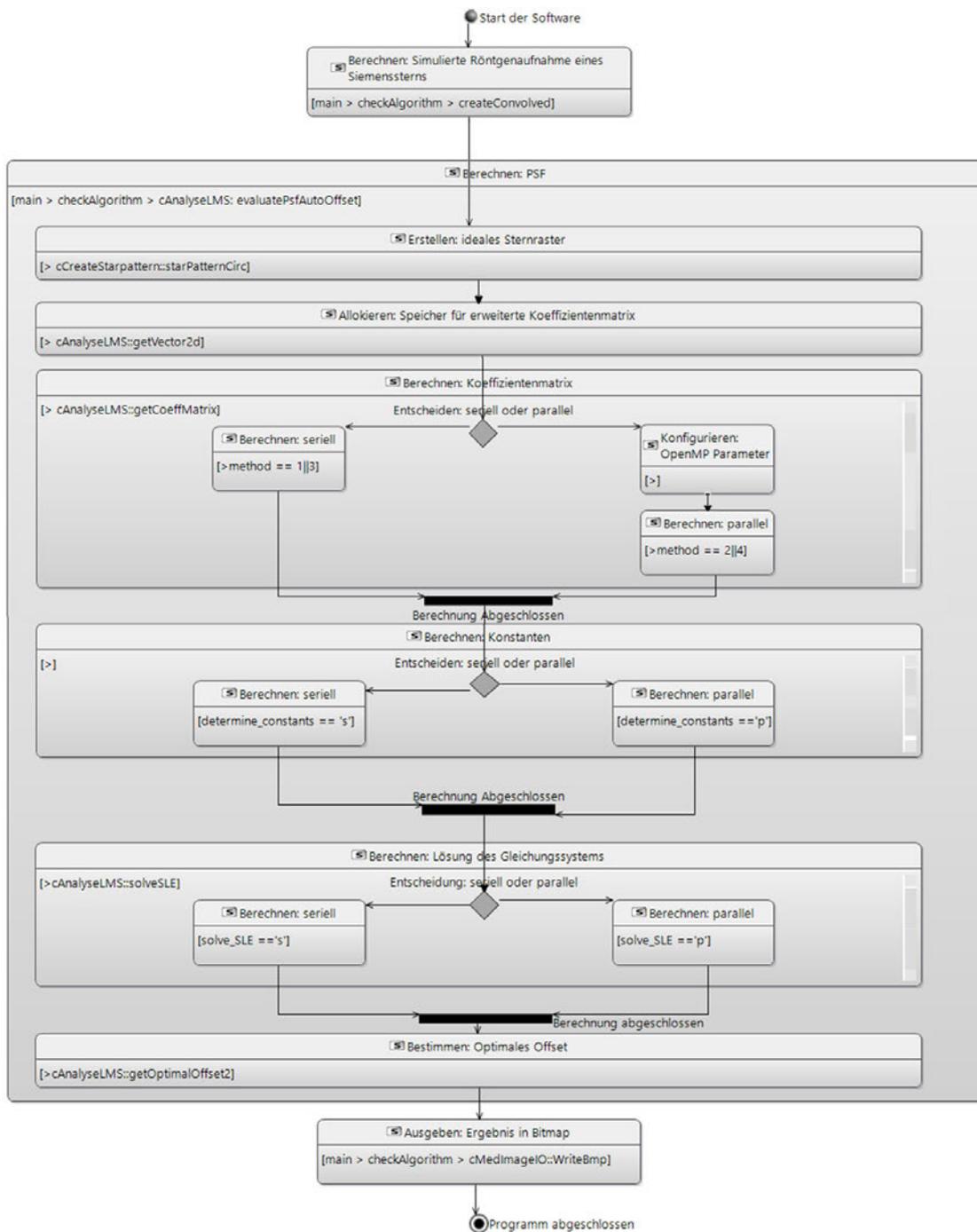


Abbildung 4.6: Design für die Parallelisierung

Für die Implementierung der Parallelisierung nach dem Design in Abbildung 4.7 werden zwei Parameter zur Steuerung der Parallelisierung eingeführt. Die Bestimmung der Koeffizientenmatrix wird für die direkte Implementierung sowie die optimierte Methode parallelisiert. Dazu werden für den Parameter *method* die parallelisierten Varianten implementiert. Die direkte Implementierung wird parallel berechnet, wenn der Parameter *method* den Wert 2 annimmt ( $method = 2$ ), die optimierte Methode wird zur Berechnung genutzt, wenn der Parameter *method* den Wert 4 annimmt ( $method = 4$ ). Für die Parallelisierung der Berechnung der Konstanten wird der Parameter *determine\_constants* eingeführt, der für die Zeichen 's' und 'p' gültig ist, wobei das Zeichen 's' die serielle Berechnung und das Zeichen 'p' die parallele Berechnung initiiert. Für die parallele Lösung des Gleichungssystems wird der Parameter *solve\_SLE* eingeführt, der für die Zeichen 's' für die serielle Berechnung und 'p' für die parallele Berechnung gültig ist. Um die Konfiguration der OpenMP Parameter zu ermöglichen, wird eine Abfrage der Anzahl der Threads und Ablaufpläne implementiert. Abbildung 4.7 zeigt ein Zustandsdiagramm der Konfiguration der OpenMP Parameter, die in der Funktion *getCoeffMatrix* vor der parallelen Berechnung implementiert wird:

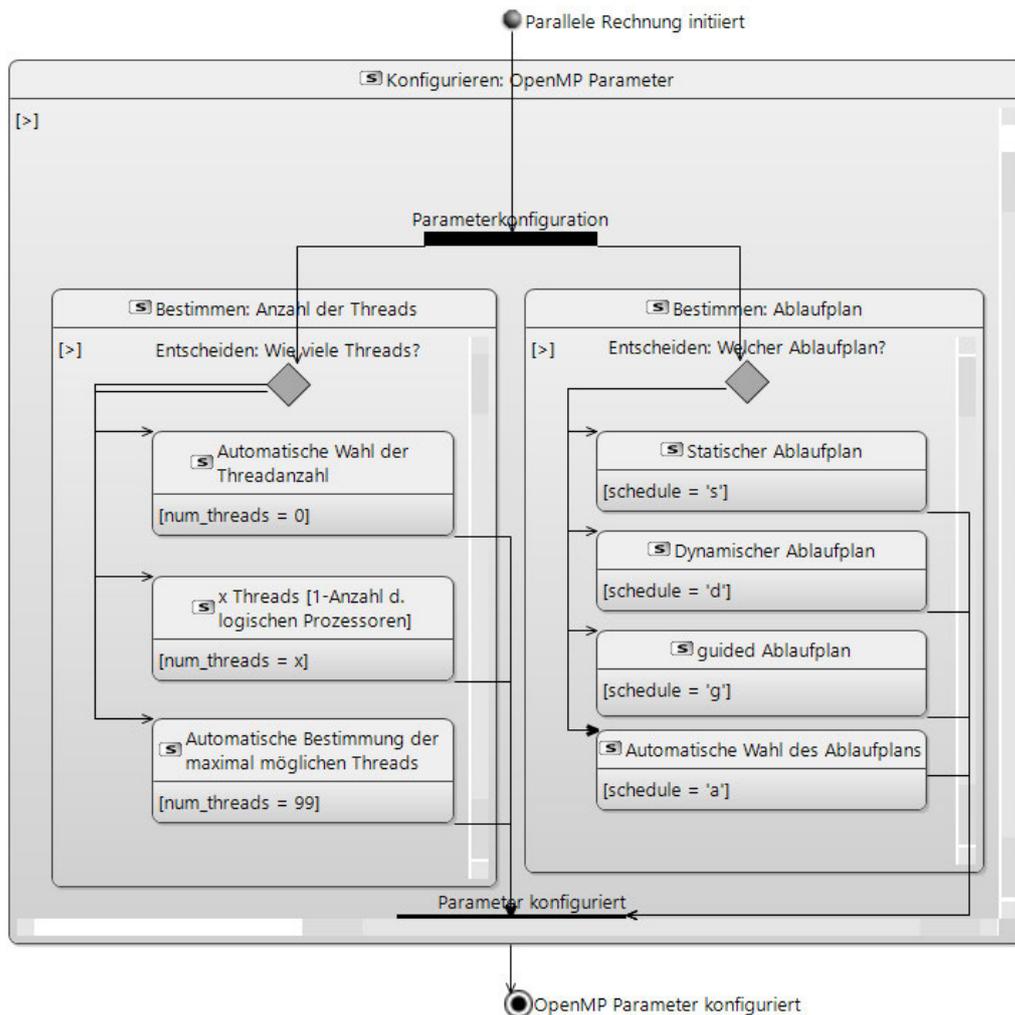


Abbildung 4.7: Design für die Wahl der Ablaufpläne

Um die Anzahl der Threads zur Parallelisierung zu beeinflussen, wird der Parameter *num\_threads* eingeführt für den die Werte 0, 99 und 1- Maximum der logischen Prozessoren des Systems gültig sind. Für den Wert *num\_threads* = 0 wird die Threadanzahl von OpenMP automatisch gewählt, für den Wert *num\_threads* = 99 automatisch das Maximum der möglichen Threads bestimmt und zur Berechnung genutzt. Für eine fest gewählte Anzahl der Threads kann der Parameter *num\_threads* die Werte 1 bis die Anzahl der logischen Prozessoren (Bsp.: 16 für das 8-Kerne System) gegeben werden. Zur Wahl des Ablaufplans wird der Parameter *schedule* eingeführt. Für *schedule* sind

die Zeichen 's' für einen statischen Ablaufplan, 'd' für einen dynamischen Ablaufplan, 'g' für einen guided Ablaufplan und 'a' für eine automatische Ablaufplanwahl zulässig. Ist die Konfiguration der Parameter abgeschlossen, werden die folgenden parallelisierten Abschnitte mit der gewählten Konfiguration bearbeitet.

Für eine einfache Anpassung der einflussnehmenden Parameter wird ein zentraler Punkt im Quellcode erstellt. Im Ausgangszustand lässt sich beispielsweise der Parameter *method*, der Einfluss auf das Verhalten der Funktion *getCoeffMatrix* nimmt, in der Funktion *evaluateAutoOffset* anpassen. Angedacht ist eine zentrale Anpassung der Parameter in der main Datei.

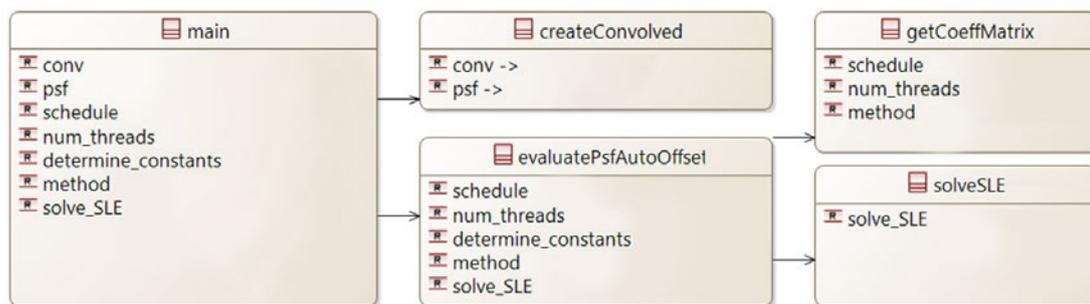


Abbildung 4.8: Design für die Parametersteuerung

## 4.4 Beschleunigung durch Compiler-Optimierungen

Neben der Parallelisierung einer Berechnung können Compilereinstellungen einen Beitrag zur Beschleunigung von Software leisten. Die Compiler-Optimierungen in Form von Compiler-Flags (dt. Marker) werden dem Compiler in den Konfigurationen übermittelt. Es werden verschiedene Optimierungslevel sowie einzelne Compiler-Flag Kombinationen für den GNU Compiler implementiert. Folgende Konfigurationen von Optimierungsanweisungen sollen implementiert und auf ihren zusätzlichen Laufzeiteinfluss untersucht werden:

Bezeichnung	Compiler-Flags
BA-OMP-Debug	Og
BA-OMP-O0	O0
BA-OMP-O1	O1, DNDEBUG
BA-OMP-O2	O2, DNDEBUG
BA-OMP-O3	O3, DNDEBUG
BA-OMP-Ofast	Ofast, DNDEBUG
BA-OMP-Ofinal	Ofast, DNDEBUG, march = native

## 5 Implementierung

Im folgenden Kapitel wird der Ausgangszustand der von Prof. Dr. Heß zur Verfügung gestellten Software mit parallelisierten Abschnitten, einer Parametersteuerung und Compilerkonfigurationen erweitert. Die Beschleunigung von Software kann auf viele Möglichkeiten erreicht werden, OpenMP bietet eine Allround-Bibliothek für die Implementierung von Multithreading auf dem Prozessor. Die Bibliothek benötigt keine zusätzlichen Builds, ist C++ kompatibel und ist fast ausschließlich durch pragma-Direktiven realisierbar, folglich fällt die Wahl für die Programmierschnittstelle zur Parallelisierung auf OpenMP. Für die Beschleunigung durch Compileroptimierungen werden die Optimierungslevel O0 bis Ofast implementiert, es handelt sich um vordefinierte Optimierungslevel des GNU Compilers. Die Optimierungslevel nehmen Einfluss auf die Laufzeit und sind einfach zu implementieren. Des Weiteren werden einige Compiler-Flags untersucht, die mit den Befehlssatzerweiterungen von Prozessoren die Laufzeit beeinflussen können. Im Folgenden wird die Implementierung der Parallelisierung, des Designs und der CompilerEinstellungen erläutert.

### 5.1 Implementierung der Parallelisierung

Die Parallelisierung auf dem Prozessor wird mittels der Bibliothek OpenMP realisiert. Aus der Programmablaufanalyse ist hervorgegangen, dass eine Parallelisierung der Funktionen *getCoeffMatrix*, *solveSLE* und *evaluatePsfAutoOffset* einen positiven Einfluss auf die Laufzeit haben könnte. Im Folgenden wird die Implementierung der Parallelisierung der Funktionen *getCoeffMatrix*, *solveSLE* und *evaluatePsfAutoOffset* erläutert:

#### 5.1.1 Bestimmung der Koeffizientenmatrix

Die Funktion *getCoeffMatrix*, die die Berechnung der Koeffizientenmatrix durchführt, ist im Ausgangszustand mit zwei Methoden zur Berechnung der Koeffizientenmatrix imple-

mentiert. Es handelt sich um die Optimierte Methode und die direkte Implementierung der Berechnung 2.3 die genutzt wird, sobald der Parameter *method* den Wert 1 annimmt. Folgender Quellcode zeigt die direkte Implementierung der Berechnung 2.3:

```

1 if (method == 1) {
2   for (unsigned r1 = 0; r1 < meas.nRow; r1++) { // Reihe der sim.
3     Roentgenaufnahme
4     cout << " " << r1 * 100.0 / meas.nRow << " % \r" << flush;
5     for (unsigned c1 = 0; c1 < meas.nCol; c1++) { // Spalten der sim.
6       Roentgenaufnahme
7       for (unsigned i = 0; i < (unsigned)M; i++) { // Reihe der Matrix m
8         double tmp = star[r1 + i / psfCols][c1 + i % psfCols]; // Wert fuer Berechnung in tmp
9         unsigned j = 0;
10        for (unsigned row = r1; row < r1 + psfRows; row++) { // Reihe der PSF
11          for (unsigned col = c1; col < c1 + psfCols; col++) { // Zeile der PSF
12            m[i][j++] += tmp * star[row][col]; }}}}} // Berechnung

```

Listing 5.1: Code für die direkte Implementierung

Bei der direkten Implementierung handelt es sich um fünf verschachtelte for-Schleifen, die über die Anzahl der Pixel der Abbildung der simulieren Röntgenaufnahme und die Anzahl der Pixel der PSF iterieren. Folgender Quellcode enthält die psarallelisierte Version der direkten Implementierung:

```

1 else if (method == 2) {
2   #pragma omp parallel for
3   for (unsigned i = 0; i < (unsigned)M; i++) { // Reihe Matrix m
4     vector<double> tmpvec; // temporaerer Rechenvektor
5     tmpvec.resize(M,0.0);
6     for (unsigned r1 = 0; r1 < meas.nRow; r1++) { // Reihe der sim. Roentgenaufnahme
7       cout << " " << r1 * 100.0 / meas.nRow << " % \r" << flush;
8       for (unsigned c1 = 0; c1 < meas.nCol; c1++) { // Spalten der sim. Roentgenaufnahme
9         double star1 = star[r1 + i / psfCols][c1 + i % psfCols]; //Wert fuer Berechnung in tmp
10        unsigned j = 0;
11        for (unsigned row = r1; row < r1 + psfRows; row++) { // Reihe der PSF
12          for (unsigned col = c1; col < c1 + psfCols; col++) { // Spalte der PSF
13            double star2 = star[row][col]; // Berechnung
14            tmpvec[j] += star1 * star2; // Zwischenspeichern in tmpvec
15            j++;
16          }}}
17        for (unsigned x=0; x<(unsigned)M; x++) // Ergebnis in Matrix m uebertragen
18        {
19          m[i][x] = tmpvec[x];
20        }}}

```

Listing 5.2: Code für die parallelisierte direkte Implementierung

Die parallelisierte direkte Implementierung wird genutzt, sobald der Parameter *method* auf den Wert '2' gesetzt wird (*method* = 2). Zur Parallelisierung wird die äußerste Schleife, die über die Anzahl der Reihen der Pixel der simulierten Röntgenaufnahme *meas* iteriert mit der pragma-Direktive *#pragma omp parallel for* versehen. Um simultane Speicherzugriffe der Threads zu vermeiden, wird ein temporärer Rechenvektor *tmpvec* der Klasse *vector* verwendet, von dem jeder Thread eine private Kopie inklusive Speicher angelegt.

Die optimierte Methode (*method* = 3) rechnet vorwiegend mit Objekten der Klasse *CMedImage*. Die Objekte der Klasse *CMedImage* allokierten Speicher über eine doppelte Zeichenkette [8], sprich Doppelzeiger. Da der OpenMP Standard private Kopien von Zeigern anlegen kann, diese Kopien aber allesamt auf dieselben Speicherstellen zeigen, kann es an dieser Stelle zu Problemen führen [9]. Die private Anweisung *#pragma omp parallel for private(Object)* funktioniert an dieser Stelle nicht, da es auf Grund überschneidender Speicherzugriffe zu Zugriffsverletzungen (engl.: Segmentation Errors) kommt. Mit folgender Implementierung konnte die Problematik umgangen werden:

```

1  else if (method == 4) {
2  #pragma omp parallel for
3  for (int dr = 0; dr < (int)psfRows; dr++) {           // Reihe der PSF
4  //omp_set_num_threads(16);
5  cMedImage<double> tmp;
6  tmp.create(2 * psfRows - 1, 2 * psfCols - 1);         // temporaere Matrix
7  vector<double> row;                                   // temporaerer Rechenvektor
8  row.resize(2 * psfCols - 1);
9  for (int dc = (dr > 0 ? -(int)psfCols + 1 : 0); dc < (int)psfCols; dc++) {
10 .
11 .
12 .
13 }}

```

Listing 5.3: Code für die parallelisierte optimierte Methode

Die parallelisierte Optimierte Methode wird genutzt, sobald der Parameter *method* auf '4' gesetzt wird (*method* = 4), der vollständige Code für die optimierte Methode und die parallelisierte optimierte Methode befinden sich im Anhang A.6. Für die Umsetzung der Parallelisierung muss innerhalb der parallelen Region, sprich der for-Schleife das *CMedImage* Objekt angelegt werden. So wird sichergestellt, dass jeder Thread eine private Kopie mit eigenem allokiertem Speicher erhält. Dasselbe Prinzip trifft auf den temporären Rechenvektor *row* zu, dieser wird innerhalb des parallelen Abschnitts angelegt, sodass jeder Thread eine eigene Kopie anlegt. Für die Parallelisierung wird die pragma-Direktive *#pragma omp parallel for* vor die äußerste Schleife gesetzt. Trifft der Compiler auf diese Direktive, wird die Parallelisierung initiiert. Die Iterationen der äußersten for-Schleife werden auf die Anzahl der Threads in Abhängigkeit des gewählten Ablaufplans aufgeteilt. Daraufhin werden in der äußersten Schleife die privaten Klassenobjekte initialisiert. Der temporäre Rechenvektor *row* enthält Zwischenergebnisse zur Berechnung der Summe. Der folgende Codeabschnitt zeigt die Übertragung der ersten Koeffizienten einer Reihe in die Ergebnismatrix *m* sowie die Berechnung der weiteren Koeffizienten der Reihe mittels Versatzberechnung.

```
1 else if (method == 4) {
2 ...
3 // first coefficient in row
4 double sum = 0.0;
5 for (int c = 0; c < psfCols - ABS(dc) - 1; c++)
6     sum += row[c];
7     sum += row[psfCols - 1];
8     unsigned i = r * psfCols + (dc < 0 ? -dc : 0);
9     unsigned j = (r + dr)*psfCols + (dc > 0 ? dc : 0);
10    m[i][j] = sum;
11    m[j][i] = sum;
12 // all other coefficients in row
13 for (int c = 1; c < psfCols - ABS(dc); c++) {
14     sum -= row[c - 1];
15     sum += row[psfCols + c - 1];
16     i++;
17     j++;
18     m[i][j] = sum;
19     m[j][i] = sum;
20 }
```

Listing 5.4: Übertragung der Ergebnisse und Versatzberechnung

Die Übertragung der Werte der Summe in die als *shared* betrachtete Ergebnismatrix  $m$  muss nicht geschützt werden, da der Iterator  $j$  von dem aktuellen Wert des Iterators der äußersten Schleife  $dr$  abhängt. Folglich hat der Iterator  $dr$  unter den Threads nie denselben Wert und es kommt nicht zu illegalen Speicherzugriffen, die Speicherstelle muss nicht geschützt werden.

### 5.1.2 Bestimmung der Konstanten

Die Bestimmung der Konstanten wird in der Funktion *evaluatePsfAutoOffset* vorgenommen.

```

1 for(unsigned r=0; r<image.nRow; r++) { // Reihe der sim. Roengenaufnahme
2   for(unsigned c=0; c<image.nCol; c++) { // Zeile der sim. Roengenaufnahme
3     for(unsigned i=0; i<M; i++) { // Reihe Matrix m
4       double tmp = star[r+i/psf.nCol][c+i*psf.nCol]; // Berechnung
5       m[i][M] += image[r][c]*tmp;
6       m[i][M+1] += tmp;
7     }}}

```

Listing 5.5: Serielle Bestimmung der Konstanten

Für die Parallelisierung wurde die Methode wie folgt modifiziert:

```

1 #pragma omp parallel for
2 for(unsigned i=0; i<M; i++) { // Reihe Matrix m
3   double tmpsum = 0; // temporaerer Parameter M
4   double tmpsumplusone = 0; // temporaerer Parameter M+1
5   for(unsigned r=0; r<image.nRow; r++) { // Reihe der sim. Roentgenaufnahme
6     for(unsigned c=0; c<image.nCol; c++) { // Zeile der sim. Roentgenaufnahme
7       double tmp = star[r+i/psf.nCol][c+i*psf.nCol]; //Berechnung
8       tmpsum += image[r][c]*tmp;
9       tmpsumplusone += tmp;
10    }
11  }
12  m[i][M] = tmpsum; // Uebertragung d. Ergebnisse in m
13  m[i][M+1] = tmpsumplusone;
14 }

```

Listing 5.6: Parallele Bestimmung der Konstanten

Es handelt sich im Ausgangszustand 5.5 um dreifach verschachtelte for-Schleifen die direkt in der Ergebnismatrix *m* arbeiten. Um illegale Speicherzugriffe zu umgehen, werden die Parameter *tmp* und *tmpplusone* eingeführt, die die Zwischenergebnisse der Aufsummierung abspeichern. Des Weiteren wird die Schleife mit dem Iterator *i* nach außen gesetzt und in dieser Schleife das Gesamtergebnis in die Ergebnismatrix *m* übertragen. Die äußere Schleife mit dem Iterator *i* wird mit der pragma-Direktive *#pragma omp parallel for* versehen, sodass jeder Thread in Abhängigkeit des gewählten Ablaufplans einen Zahlenbereich von *i* abarbeitet. Folglich werden keine illegalen Speicherzugriffe bei der Übertragung der Ergebnisse von *tmp* und *tmpplusone* in die Ergebnismatrix *m* getätigt.

### 5.1.3 Lösung des Gleichungssystems

Die Lösung des Gleichungssystems ist in der Funktion *solveSLE* definiert. Es handelt sich um eine direkte Implementierung des Gauss-Jordan Verfahrens:

```
1 // loop over rows in matrix
2 for (unsigned i=0; i<nRow; i++) {
3     // normalize current row
4     for (unsigned j=i+1; j<nCol; j++)
5         Ab[i][j] /= Ab[i][i];
6     Ab[i][i] = 1.0;
7
8     // change all elements above and below to zero
9     for (unsigned ii=0; ii<nRow; ii++) {
10        for (unsigned j=i + 1; ii!=i && j<nCol; j++)
11            Ab[ii][j] = Ab[ii][j] * Ab[i][i];
12        if (ii!=i) Ab[ii][i] = 0.0;
```

Listing 5.7: Ausgangszustand: Lösung des Gleichungssystems

Die Berechnung ist mit bis zu dreifach verschachtelte for-Schleifen realisiert. Die Berechnung weist Gegenabhängigkeiten auf, die es zunächst verhindern diese Funktion erfolgreich zu parallelisieren. Für eine Parallelisierung müssen die Datenabhängigkeiten entfernt werden, im Rahmen dieser Ausarbeitung wird die Funktion *solveSLE* nicht parallelisiert.

## 5.2 Parametersteuerung

Um Änderungen an der Konfiguration zentral vornehmen zu können, wurden drei neue Parameter zur Steuerung der Konfiguration implementiert. Der OpenMP Standard bietet die Möglichkeit Ablaufpläne und die Anzahl der genutzten Threads zu beeinflussen. Im Folgenden wird erläutert, wie Einfluss auf die Konfiguration der Software genommen werden kann.

### 5.2.1 Ablaufpläne

Für die Wahl des OpenMP Ablaufpläne wurde ein Parameter *schedule* in die Funktionen *evaluatePsfAutoOffset* und *getCoeffMatrix* eingeführt. Der Parameter *schedule* lässt sich in der main in den Funktionen *analyseParameter* und *CheckAlgorithm* setzen. Bei dem Parameter *schedule* handelt es sich um den Datentyp `char`, für den folgende Zeichen gültig sind:

's' static, statischer Ablaufplan

'd' dynamic, dynamischer Ablaufplan

'g' guided, guided Ablaufplan

'a' auto, automatische Ablaufplanwahl

Der Ablaufplan lässt sich mit der Bibliotheksfunktion `omp_set_schedule(omp_sched_t kind, int chunk size)` setzen [13]. Der Ablaufplan wird in der Funktion `getCoeffMatrix` gesetzt. Abbildung 5.1 veranschaulicht die Implementierung:

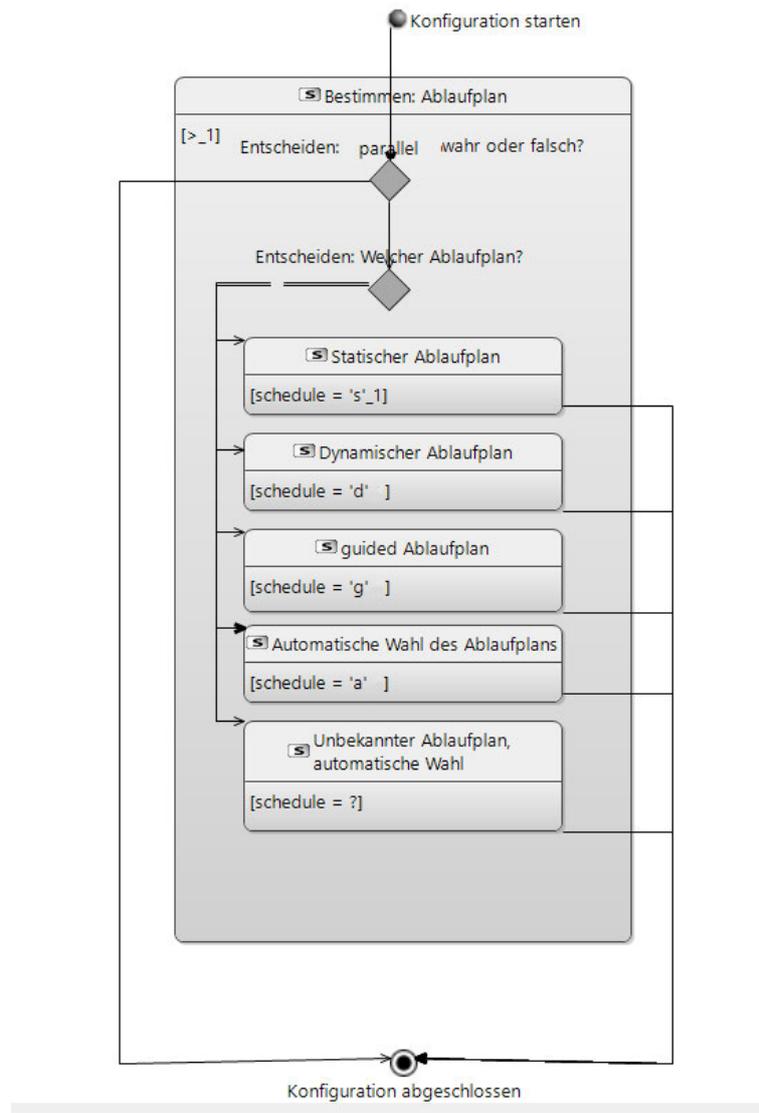


Abbildung 5.1: Ablauf für die Wahl des Ablaufplans

Für die automatische Einstellung des Ablaufplans wurde eine if-else Anweisung in Abhängigkeit des Parameters `schedule` und dem lokalen Parameter `parallel` implementiert, der

überprüft, ob es sich um eine parallele Berechnung handelt. Der Parameter *parallel* ist vom Datentyp `bool`, ist der Parameter *method* gleich zwei oder vier gesetzt (`method == 2||4`) wird *parallel* wahr. Ist *parallel* falsch, bedeutet dies, dass die Bestimmung der Koeffizienten seriell stattfindet. Sobald *parallel* wahr ist, wird der Parameter *schedule* mit den if-else Anweisungen verglichen, für jeden Ablaufplan gibt es eine passende Anweisung. Wird *schedule* auf ein nicht oben genanntes Zeichen gesetzt, wird der Ablaufplan von OpenMP automatisch bestimmt. Um sicherzugehen, dass der gewünschte Ablaufplan ausgeführt wird, wird beim Ausführen der Software ausgegeben, mit welchem Ablaufplan gearbeitet wird.

### 5.2.2 Anzahl der zu nutzenden Threads

Die Anzahl der genutzten Threads muss nicht zwingend definiert sein, OpenMP kann automatisch bestimmen, wie viele Threads initialisiert werden sollen. Um das Variieren der Threadanzahl zu ermöglichen, wurde der Parameter *num\_threads* eingeführt. Für den Parameter *num\_threads* sind folgende Werte gültig:

'0' automatische Wahl der Anzahl

'1-x' gültig sind Zahlen von 1 bis zur Anzahl der logischen Kerne des Prozessors x

'99' automatische Bestimmung der maximalen Threadanzahl

Abbildung 5.2 veranschaulicht die Implementierung der Konfiguration der Threadanzahl:

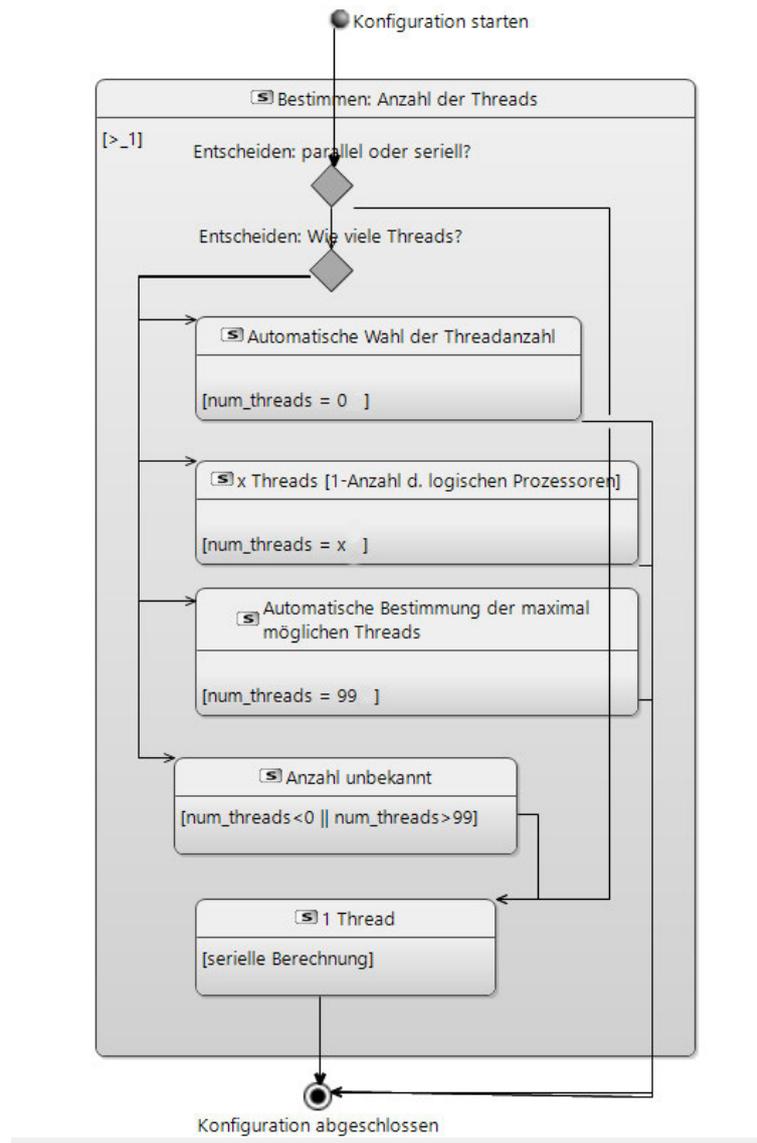


Abbildung 5.2: Ablauf für die Wahl der Threadanzahl

Der Parameter *num\_threads* kann in der main für die Funktionen *checkAlgorithm* und *AnalyseParameter* gesetzt werden und wird an die Funktionen *evaluateAutoOffset* und *getCoeffMatrix* weitergegeben. In der Funktion *getCoeffMatrix* lässt sich die Anzahl der Threads mit der Bibliotheksfunktion *omp\_set\_num\_threads(intnum\_threads)* setzen [12]. Für den automatischen Ablauf wurde eine if-else Anweisung implementiert (Anhang A.8), die in Abhängigkeit des Parameters *num\_threads* die Threadzahl setzt. Anmer-

kung: Für den Fall, dass dem Parameter *num\_threads* der Wert 1 (*num\_threads* = 1) gegeben wird, handelt es sich um eine serielle Ausführung, die als solche nicht erkannt wird.

### 5.2.3 Bestimmung der Konstanten

Um die Bestimmung der Konstanten seriell oder parallel ablaufen zu lassen, wurde der Parameter *determine\_constants* eingeführt. Es handelt sich um einen Parameter des Datentyps `char`, für den folgende Zeichen gültig ist:

's' serielle Berechnung

'p' parallele Berechnung

Der Parameter *determine\_constants* kann in der `main` in den Funktionen *checkAlgorithm* und *analyseParameter* gesetzt werden. Das Zeichen des Parameters *determine\_constants* wird an die Funktion *evaluatePsfAutoOffset* weitergegeben. Abbildung 5.3 veranschaulicht die Implementierung zur Parallelisierung der Bestimmung der Konstanten.

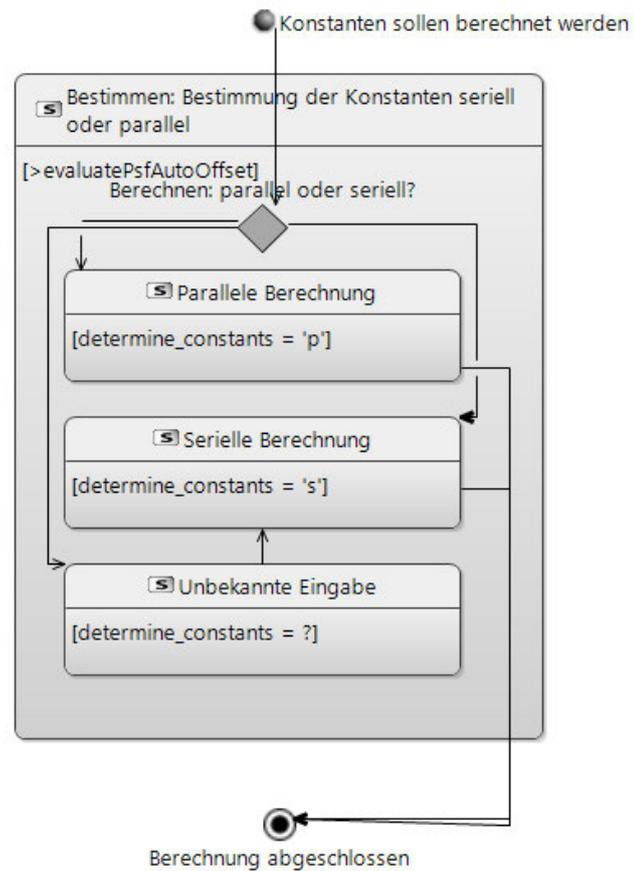


Abbildung 5.3: Ablauf für die Wahl der Bestimmung der Konstanten

In der Funktion *evaluatePsfAutoOffset* wird über eine if-else Anweisung in Abhängigkeit des Parameters *determine\_constants* die Berechnung entweder mit der pragma-Direktive *#pragma omp parallel for* parallelisiert (Anhang A.9) oder seriell ausgeführt.

#### 5.2.4 Method

Für die Bestimmung der Methode zur Berechnung der Koeffizientenmatrix wurde der Parameter *method* für die Funktionen *evaluatePsfAutoOffset* und *CheckAlgorithm* erweitert. Der Parameter kann in der main angepasst werden und wird über die Funktionen

*checkAlgorithm* und *evaluatePsfAutoOffset* an die Funktion *getCoeffMatrix* weitergegeben. Für den Parameter *method* sind folgende Werte zulässig:

- '1' Direkte Implementierung
- '2' Direkte Implementierung, parallelisiert
- '3' Optimierte Methode
- '4' Optimierte Methode, parallelisiert

Ist *method* auf den Wert '2' oder '4' gesetzt, handelt es sich um die parallele Berechnung mittels OpenMP, für die Werte *method* = 1 oder *method* = 3 handelt es sich um die serielle Berechnung der Koeffizientenmatrix. Die Bestimmung der Methode zur Bestimmung der Koeffizientenmatrix wird im Ausgangszustand über eine if-else Anweisung umgesetzt A.6. Für die Parallelisierung wurde die Anweisung um die zwei parallelisierten Methoden erweitert. Abbildung 5.4 veranschaulicht die Implementierung.

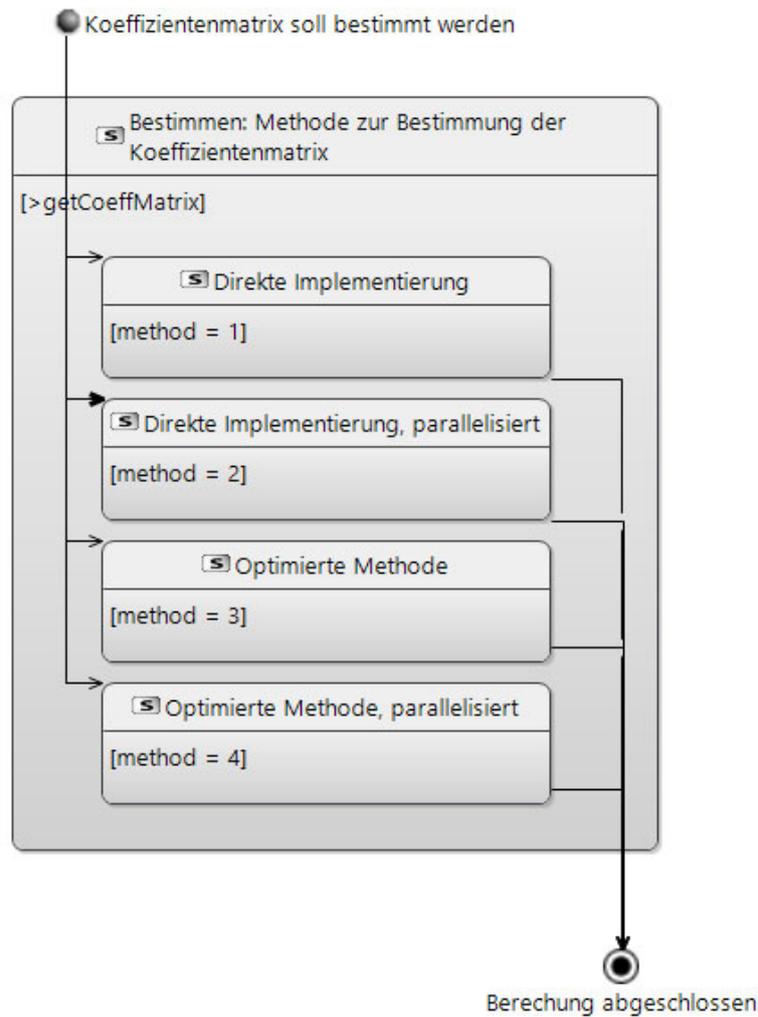


Abbildung 5.4: Ablauf für die Wahl der Methode zur Bestimmung der Koeffizientenmatrix

### 5.2.5 Automatisierung der Abweichung

Im Ausgangszustand wurde die Berechnung der Abweichung hart codiert, um die Berechnung zu automatisieren, wurden vier Parameter zur Berechnung eingeführt. Für die Abweichung werden die Dimension der PSF, die Dimension des Kerns, mit dem das ideale Sternraster gefaltet wird und die PSF benötigt. Folgende Double Parameter wurden für die Berechnung der Abweichung eingeführt:

**Dpsfrows:** Anzahl der Reihen der PSF

**Dpsfcols:** Anzahl der Zeilen der PSF

**Krows** Anzahl der Reihen des Kernels

**Kcols:** Anzahl der Zeilen des Kernels

Folgender Quellcode 5.8 zeigt die Implementierung der automatisierten Berechnung der Standardabweichung:

```
1 double stdev = 0;
2 for(unsigned r=0; r<psf.nRow; r++) { //Reihe der PSF
3   for(unsigned c=0; c<psf.nCol; c++) { //Zeile der PSF
4     //double tmp = psf[r][c] - (r>=9 && r<16 && c>=9 && c<16 ? 1 : 0); Ausgangszustand
5     double tmp = psf[r][c] - (r>=((dPSFRows/2)-(krows/2)) && r<(((dPSFRows/2)+(krows/2)) && c
6       >=((dPSFCols/2)-(kcols/2)) && c<(((dPSFCols/2)+(kcols/2)) ? 1 : 0); // automatisiert
7     stdev += tmp*tmp; // Berechnung
8   }
9 }
10 stdev /= psf.nRow*psf.nCol;
11 stdev = sqrt(stdev);
12 cout << " stdev: " << stdev << endl;
```

Listing 5.8: Automatisierten Berechnung der Standardabweichung

Die Parameter werden von Integer auf Double gecastet, da mit Gleitkommazahlen gerechnet wird. Für die Berechnung wird die PSF mit der Dimension des Kernels verrechnet und geprüft, ob der Wert näher an null oder eins liegt. Das Ergebnis wird durch die Anzahl der Pixel der PSF geteilt und quadriert, um die Standardabweichung zu erhalten. Umso kleiner die Abweichung, umso genauer ist die Rekonstruktion der PSF gelungen. Anzumerken ist, dass die Klammern um die Erstellung des gefalteten Bildes entfernt werden mussten, da sonst kein Zugriff auf die Dimension des Kernels möglich ist.

## 5.3 Compilerkonfigurationen

Für die Laufzeitoptimierung durch Compilereinstellungen werden acht Compilerkonfigurationen implementiert. Die Konfigurationen kompilieren das Basisprogramm für einen 64Bit Befehlssatz, sie unterscheiden sich in ihren Compiler-Flags. Der GNU Compiler verfügt über vordefinierte Optimierungslevel, die mittels Compiler-Flags aktiviert werden können. Für die fünf Optimierungslevel O0,O1,O2,O3 und Ofast werden jeweils eine Konfiguration angelegt, des Weiteren gibt es eine Konfiguration mit der Flag -march=native, die mit Befehlssatzerweiterungen arbeitet, die in Abhängigkeit des Prozessors variieren

können. Des Weiteren wurde eine Konfiguration zum Debuggen implementiert. Im Folgenden sind die Konfigurationen genauer aufgeführt:

Bezeichnung	Compiler-Flags
BA-OMP-Debug	Og
BA-OMP-O0	O0
BA-OMP-O1	O1, DNDEBUG
BA-OMP-O2	O2, DNDEBUG
BA-OMP-O3	O3, DNDEBUG
BA-OMP-Ofast	Ofast, DNDEBUG
BA-OMP-Ofinal	Ofast, DNDEBUG, march = native

Die Build-Konfigurationen sind als `preLaunchTask` in `tasks.json` definiert und als Ausführungsanweisungen `launch.json` eingepflegt, sodass die Konfigurationen über das 'Run and Debug' Menü von VSCode gewählt und gestartet werden können. Ein Beispiel für die Implementierung einer Konfiguration befindet sich im Anhang A.10. Wichtig ist, dass der Pfad zum Compiler bei Systemwechsel angepasst wird.

## 6 Validierung

Die Laufzeit der parallelisierten Software hängt von der Konfiguration der Parameter ab. Im Folgenden wird die Laufzeit in Abhängigkeit der einflussnehmenden Parameter untersucht. Zusätzlich wird untersucht, wie Optimierungseinstellungen die Laufzeit beschleunigen können. Es wird geprüft, inwiefern die Anforderungen und die Implementierung des Designs erfolgt sind.

### 6.1 Untersuchung des Laufzeitverhaltens der parallelisierten Software

Die Bibliothek OpenMP ermöglicht die Einflussnahme auf das Verhalten der Parallelisierung, es können vier Ablaufpläne und die Threadanzahl durch Anweisungen gesetzt werden. Im Folgenden wird die Laufzeit der parallelisierten Software auf Variation der Bildgröße der simulierten Röntgenaufnahme *conv*, der Bildgröße der PSF *psf*, der Ablaufpläne *schedule* und der Threadanzahl *num\_thread* untersucht. Des Weiteren wird das Laufzeitverhalten auf zwei zusätzlichen Rechensystemen überprüft. Als Gegenwert der Messungen wird die theoretische Beschleunigung betrachtet. Für die Berechnung der theoretischen Beschleunigung in Abhängigkeit der Prozessorkerne gilt [30]:

$$S_p = \frac{T_1}{T_1((1 - f) + \frac{f}{p})}$$

mit:

$p$  : Anzahl der Prozessorkerne

$S_p$  : Beschleunigung

$T_1$  : Laufzeit[s] der seriellen Ausführung

$f$ : Anteil der Anwendung, die parallelisiert werden kann

Die theoretische Beschleunigung ergibt für die Prozessorkernanzahl von 2,4,6 und 8 und einen parallelisierbaren Anteil  $f = 0,98$ :

Anzahl der Kerne	2	4	6	8
Beschleunigung	1,96	3,77	5,45	7,01

### 6.1.1 Laufzeit bei variierender Bildgröße der simulierten Röntgenaufnahme

Für die Untersuchung der Laufzeit in Abhängigkeit der Bildgröße der simulierten Röntgenaufnahme wird die Bildgröße von der Dimension 100x100 Pixel bis 1000x1000 Pixel in Hunderterschritten betrachtet. Die Laufzeit wird für die parallelisierten Methoden ( $method = 2$ || $method = 4$ ) sowie parallele Bestimmung der Konstanten gemessen und mit dem Ausgangszustand ins Verhältnis gesetzt. Für jede Bildgröße werden zehn Stichproben genommen und die durchschnittliche Laufzeit berechnet, die genauen Messdaten befinden sich im Anhang A.1. Zusammengefasst lautet die Konfiguration für die Leistungsmessung:

Bildgröße conv	variiert
Bildgröße psf	30x30 Pixel
schedule	static
num_threads	16
determine_constants	's' für method 1&3, 'p' für method 2&4
method	1,2,3,4
Optimierungslevel	O0
Stichprobenmenge	10 pro Bildgröße

Abbildung 6.1 veranschaulicht die durchschnittlichen Messwerte:

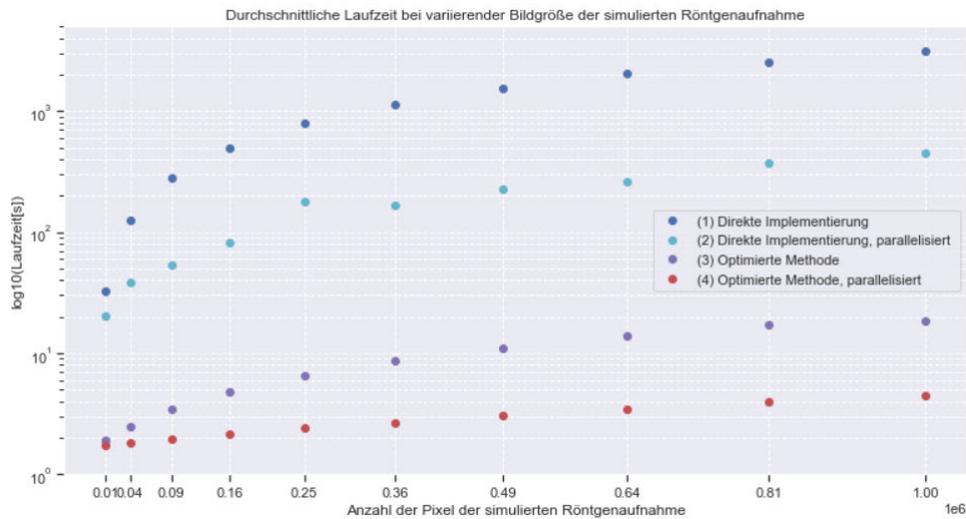


Abbildung 6.1: Laufzeitverhalten bei variierender Bildgröße der simulierten Röntgenaufnahme

Der Graph 6.1 stellt die Laufzeit auf logarithmischer Achse im Verhältnis der Anzahl der Pixel der simulierten Röntgenaufnahme dar. In der Programmablaufanalyse konnte bereits festgestellt werden, dass die Laufzeit in Zusammenhang mit der Bildgröße der simulierten Röntgenaufnahme sowie der Bildgröße der PSF steht. Diese Annahme bestätigt sich ebenfalls für die parallelisierte Berechnung. Es wird deutlich, dass die Parallelisierung eine Beschleunigung für die direkte Implementierung sowie die optimierte Methode bewirkt hat. Anzumerken ist, dass die Bestimmung der Konstanten für die Messungen der parallelen Berechnung ebenfalls parallel berechnet werden. Für die gemessene Beschleunigung gilt [30]:

$$S_p = \frac{T_1}{T_p} \quad (6.1)$$

mit:

$S_p$ : Beschleunigung

$T_1$ : Laufzeit[s] der seriellen Ausführung

$T_p$ : Laufzeit[s] der parallelen Ausführung

Werden für die Bildgröße 1000x1000 Pixel die durchschnittlichen Laufzeiten betrachtet, ergibt sich für die Beschleunigung der direkten Implementierung:

$$S_p = \frac{3162}{450} = 7,02$$

und für die optimierte Methode:

$$S_p = \frac{20,59}{4,49} = 4,58$$

### 6.1.2 Laufzeit bei variierender Bildgröße der PSF

Für die Untersuchung der Laufzeit in Abhängigkeit der Bildgröße der PSF wird die Dimension der Bildgrößen 11x11 Pixel, 15x15 Pixel und 20x20-50x50 Pixel in Zehnerschritten betrachtet. Aufgrund der Größe des Kernels von 7x7 Pixel wird mit der Bildgröße 11x11 Pixel begonnen. Da eine zu kleine Bildgröße der PSF zu fehlerhaften Berechnungen führt, ist die Mindestgröße der PSF 11x11 Pixel für einen Kernel von 7x7 Pixeln.

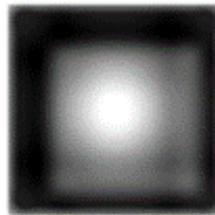


Abbildung 6.2: PSF für eine Bildgröße 10x10 Pixel und einer Kernelgröße 7x7 Pixel

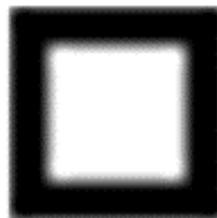


Abbildung 6.3: PSF für eine Bildgröße 11x11 Pixel

Für jede betrachtete Bildgröße der PSF werden fünf Stichproben gemessen und die durchschnittliche Laufzeit berechnet, die genauen Messdaten befinden sich im Anhang A.2. Folgende Konfiguration wird für die Leistungsmessungen verwendet:

Bildgröße conv	100x100 Pixel
Bildgröße psf	variiert
schedule	static
num_threads	16
determine_constants	's' für method 1&3, 'p' für method 2&4
method	1,2,3,4
Optimierungslevel	O0
Stichprobenmenge	5 pro Bildgröße

Abbildung 6.4 veranschaulicht die durchschnittlichen Messwerte:

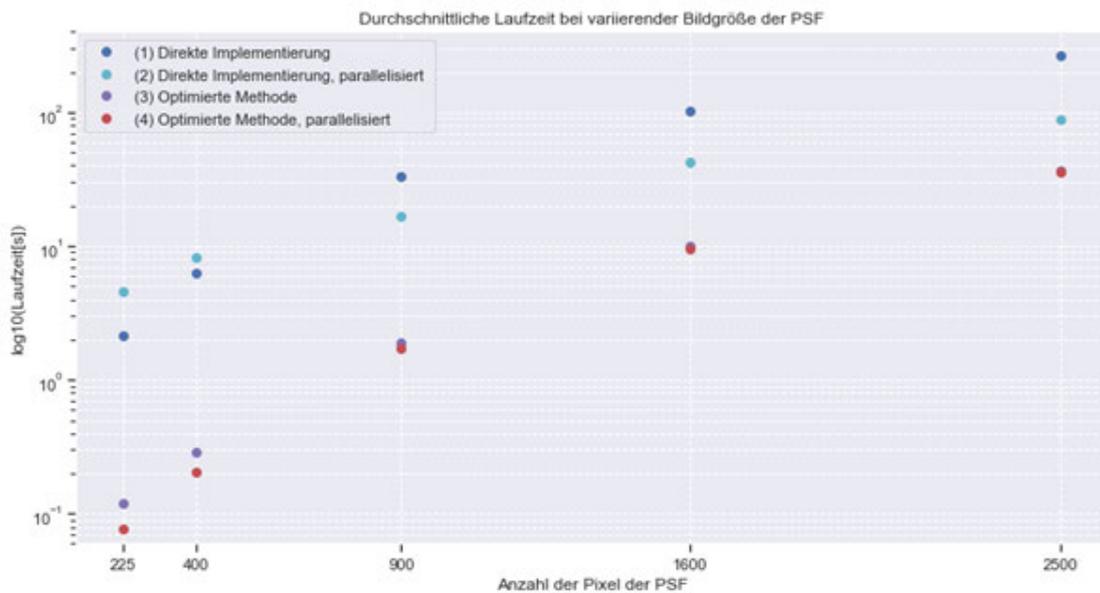


Abbildung 6.4: Laufzeitverhalten bei variierender Bildgröße der PSF

Die im Graph 6.4 dargestellten Messungen beschreiben die durchschnittlichen Laufzeiten in Abhängigkeit der Anzahl der Pixel der PSF. Die Messungen umfassen die serielle Berechnung ( $method = 1$  |  $method = 3$ ,  $determine\_constants = 's'$ ), sowie die parallele Bestimmung der Koeffizientenmatrix ( $method = 2$  |  $method = 4$ ,  $determine\_constants = 'p'$ ). Es wird deutlich, dass die direkte Implementierung bei einer kleinen Bildgröße der

PSF für die serielle Berechnung schneller ist als die parallele Berechnung der direkten Implementierung. Es geht hervor, dass bei einer kleinen Bildgröße der PSF der Verwaltungs- und Synchronisationsaufwand der parallelen Berechnung höher als der Rechenaufwand der seriellen Berechnung ist. Die Bildgröße, ab der der Rechenaufwand den Synchronisationsaufwand der parallelisierten direkten Implementierung übersteigt, liegt zwischen den Bildgrößen 20x20 Pixel und 30x30 Pixel. Die optimierte Methode ist für alle betrachteten Bildgrößen der PSF schneller als die serielle Berechnung. Es wird deutlich, dass die serielle und parallele Berechnung der optimierten Methode bei wachsender Bildgröße der PSF keine Beschleunigung zu verzeichnen ist. Im Folgenden wird die Zusammensetzung der Laufzeit der Funktionen genauer betrachtet:

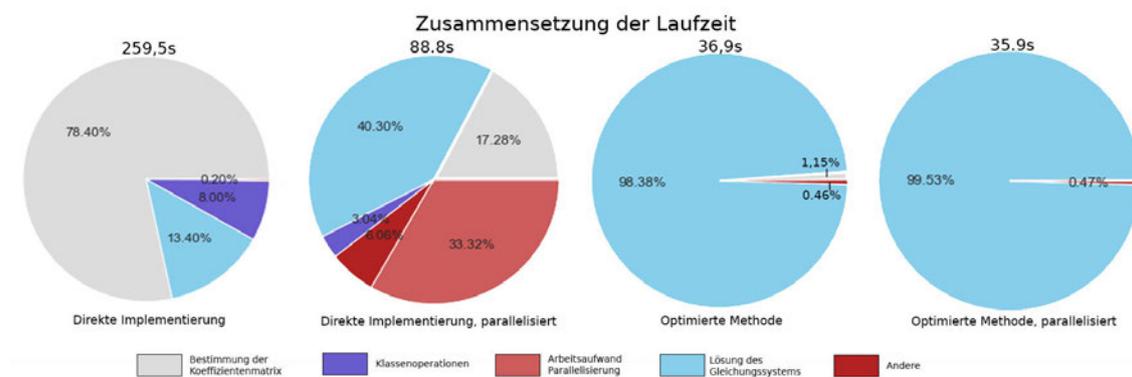


Abbildung 6.5: Zusammensetzung der Laufzeit bei variierender Bildgröße der PSF

Die Zusammensetzung der Laufzeit wird für eine Bildgröße der simulierten Röntgenaufnahme von 1000x1000 Pixel und eine Bildgröße der PSF für 50x50 Pixel betrachtet. Für die direkte Implementierung wird deutlich, dass die Parallelisierung der Bestimmung der Koeffizientenmatrix die Beschleunigung verursacht. Bei der optimierten Methode beläuft sich die Differenz von der seriellen zur parallelen Berechnung auf 1s, da die Lösung des Gleichungssystems für die serielle Berechnung bereits 96,38% der Laufzeit ausmacht. Die parallelisierte optimierte Methode löst zu 99,53% der Laufzeit das Gleichungssystem. Folglich ist die fehlende Beschleunigung darauf zurückzuführen, dass die Funktion zur Lösung des Gleichungssystems *solveSLE* im Umfang dieser Ausarbeitung nicht parallelisiert werden konnte.

### 6.1.3 Laufzeit bei variierenden OpenMP Schedules

Die OpenMP Bibliothek stellt vier verschiedene Ablaufpläne zur Verfügung, die in Abhängigkeit der Rechenaufgabe unterschiedliche Beschleunigungen erzielen. Im Folgenden wird die Laufzeit in Abhängigkeit der OpenMP Ablaufpläne untersucht. Dazu werden die Ablaufpläne static, dynamic, guided und auto mit zwanzig Stichproben pro Ablaufplan untersucht. Mit folgender Konfiguration werden die Messungen vorgenommen:

Bildgröße conv	3000x3000 Pixel
Bildgröße psf	30x30 Pixel
schedule	variiert
num_threads	16
determine_constants	'p'
method	4
Optimierungslevel	O0
Stichprobenmenge	20 pro Bildgröße



Abbildung 6.6: Laufzeitverhalten bei variierenden Ablaufplänen

Für eine genauere Betrachtung wurde die Achse startend bei 25,5 s skaliert

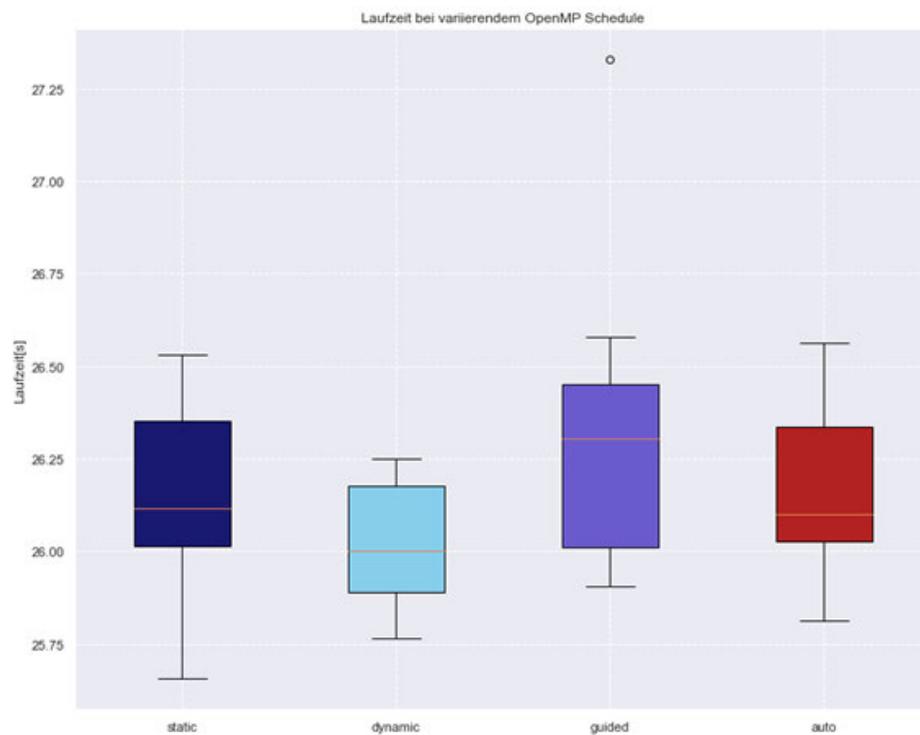


Abbildung 6.7: Laufzeitverhalten bei variierenden Ablaufplänen, skaliert

Aus den Messungen geht hervor, dass die Ablaufpläne bei genannter Konfiguration eine Laufzeit zwischen 25,6 s und 27,5 s haben, die Messdaten befinden sich im Anhang A.3. Der Ablaufplan static hat für eine Stichprobenmenge von 20 Leistungsmessungen eine Laufzeit im Median von ca. 26,10 s, die minimale Laufzeit beträgt 25,66 s und die maximale Laufzeit 26,5 s. Die mittleren 50% der Messungen, dargestellt als Interquartilbereich, liegen zwischen den Laufzeiten 26,00 s und 26,34 s. Für die Stichproben gibt es keine Ausreißer. Der Ablaufplan dynamic hat für eine Stichprobenmenge von 20 Leistungsmessungen eine Laufzeit im Median von ca. 26,00 s, die minimale Laufzeit beträgt 25,79 s und die maximale Laufzeit beträgt 26,25 s. Das untere Quartil liegt bei einer Laufzeit von 25,90 s, das obere Quartil bei einer Laufzeit von 26,20 s. Für die Stichproben gibt es keine Ausreißer. Der Ablaufplan guided hat für eine Stichprobenmenge von 20 Leistungsmessungen eine Laufzeit im Median von ca. 26,30 s, die minimale Laufzeit beträgt 25,95 s und die maximale Laufzeit beträgt 27,32 s, bei diesem Messwert handelt es sich um einen Ausreißer. Die größte Laufzeit exklusive Ausreißer liegt bei 26,60 s. Der Inter-

quartilbereich liegt zwischen den Laufzeiten 26,00 s und 26,4 s. Der Ablaufplan auto hat für eine Stichprobenmenge von 20 Durchläufen eine Laufzeit im Median von ca. 26,08 s, die minimale Laufzeit beträgt 25,81 s und die maximale Laufzeit beträgt 26,56 s. Das untere Quartil liegt bei einer Laufzeit von 26,05 s, das obere Quartil bei einer Laufzeit von 26,35 s. Es wird deutlich, dass der Median des Ablaufplans guided am höchsten ist, des Weiteren ist die Streubreite des Interquartilbereichs im Vergleich zu den anderen Ablaufplänen größer. Die Spannbreite des Ablaufplans static am größten. Für den Ablaufplan auto kann nicht genau bestimmt werden, welche Ablaufpläne für die Stichproben wie oft gewählt wurden. Das Laufzeitverhalten zeigt ein Mischbild der Ablaufpläne guided, dynamic und static. Der Ablaufplan dynamic weist die geringste Spannbreite und den kleinsten Median für die gegebene Konfiguration auf. Aufgrund dessen wird in aufbauenden Messungen der Ablaufplan dynamic genutzt. In Abbildung 6.6 wird deutlich, dass die Laufzeiten der Ablaufpläne ähnlich verlaufen, im Maximum weichen diese bei der angegebenen Konfiguration maximal, Ausreißer nicht berücksichtigt, 0,94 s voneinander ab. Bei einer durchschnittlichen Gesamtlaufzeit von 26,5 s macht das 3,6% der Laufzeit aus, folglich haben die Ablaufpläne einen geringfügigen Einfluss auf die Laufzeit.

#### 6.1.4 Laufzeit bei variierender Threadanzahl

Bei der Variation der Threadanzahl sollt das Verhalten der Laufzeit bei variierender Threadanzahl untersucht werden. Die Laufzeit soll für 2, 4, 8 und 16 Threads untersucht werden. Mit folgender Konfiguration werden die Messungen vorgenommen:

Bildgröße conv	3000x3000 Pixel
Bildgröße psf	30x30 Pixel
schedule	dynamic
num_threads	variiert
determine_constants	'p'
method	4
Optimierungslevel	00
Stichprobenmenge	10 pro Threadanzahl

Abbildung 6.8 veranschaulicht die durchschnittliche Laufzeit in Sekunden[s] bei variierender Threadanzahl:

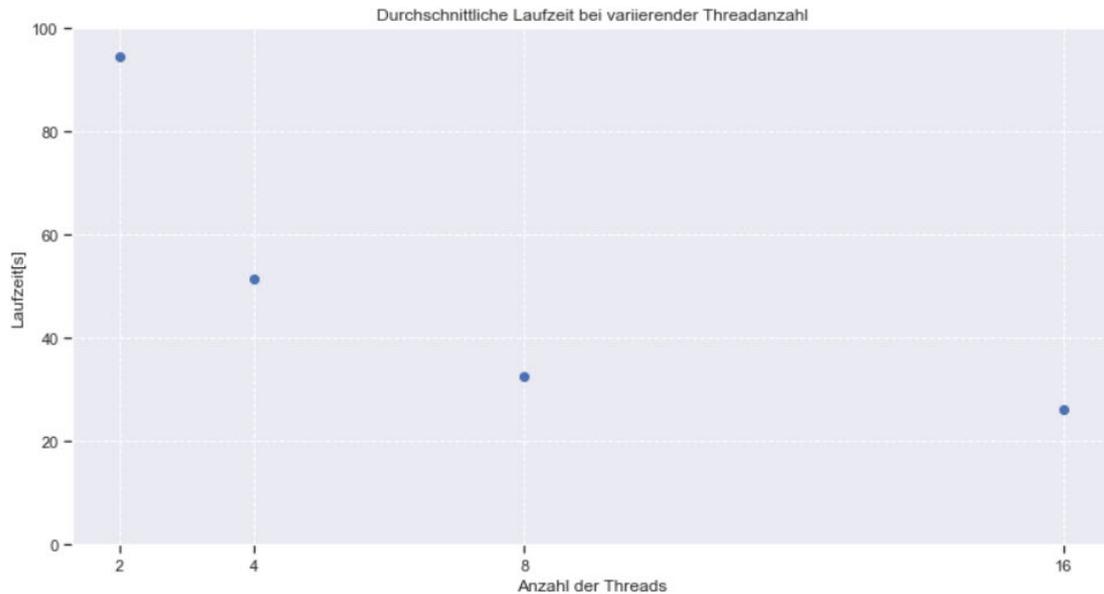


Abbildung 6.8: Laufzeitverhalten bei variierender Threadanzahl

Es wird deutlich, dass die Laufzeit nicht in linearem Zusammenhang zur Anzahl der Threads steht. Folgende Laufzeitbeschleunigung mit 6.1 kann im Vergleich zur seriellen Berechnung erreicht werden:

<i>num_threads</i>	1	2	4	6	8
durchschnittliche Laufzeit[s]	171	94,44	51,43	32,49	26,20
Beschleunigung		1,82	3,35	5,29	6,56

Dieses Phänomen ist auf die zusätzlichen Verwaltungsaufwand zur Synchronisation der Threads zurückzuführen. Der mit der Anzahl der Threads wachsende Verwaltungsaufwand verrechnet sich mit der Laufzeitbeschleunigung durch Parallelisierung, es folgt eine gegen einen unbekanntem Wert konvergierende Kurve. Dieser unbekanntem Wert kann durch die Betrachtung der Laufzeit über  $\frac{1}{\text{Anzahl der Threads}}$  approximiert werden. Für die Betrachtung der Laufzeit über  $\frac{1}{\text{Anzahl der Threads}}$  ergibt sich folgender Graph 6.9:

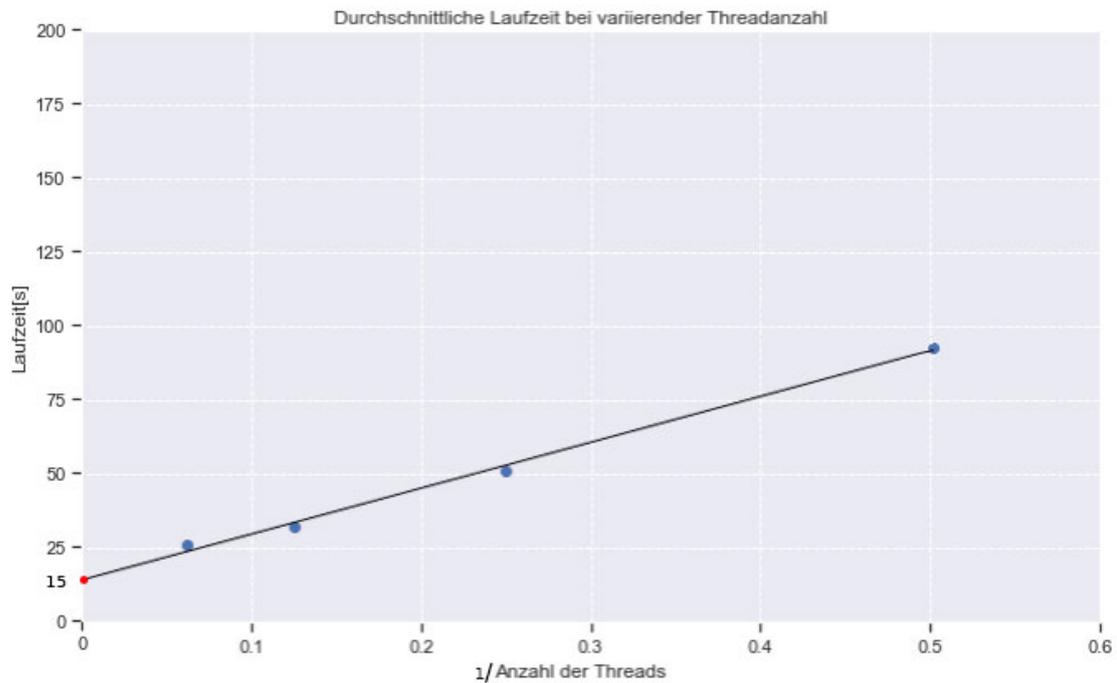


Abbildung 6.9: Approximierung der theoretischen Laufzeit für eine unendliche Threadanzahl

Die Approximierung der Grenzlafzeit erfolgt über eine Gerade. Die Gerade wird anhand der Messwerte bestimmt und bis zur theoretischen Threadanzahl von unendlich, hier 0.0 auf der X-Achse, gezogen. Der abzulesende Wert für die minimale Laufzeit bei einer unendlichen Threadanzahl beläuft sich auf ca. 15 s.

## 6.2 Weitere Beschleunigung durch Compilereinstellungen

Neben der Beschleunigung durch Parallelisierung lässt sich die Laufzeit ebenso durch Compilereinstellungen beeinflussen. Die Laufzeit in Abhängigkeit der Optimierungslevel von dem GNU Compiler werden für die Optimierungslevel O1, O2, O3, Ofast und das eigen implementierte Level Ofinal untersucht und mit dem Ausgangszustand O0 verglichen. Mit folgender Konfiguration werden die Messungen vorgenommen:

Bildgröße conv	3000x3000 Pixel
Bildgröße psf	30x30 Pixel
schedule	dynamic
num_threads	16
determine_constants	'p'
method	4
Optimierungslevel	variiert
Stichprobenmenge	10 pro Threadanzahl

Abbildung 6.9 veranschaulicht die Messdaten für die Optimierungslevel und dessen durchschnittliche Laufzeit:

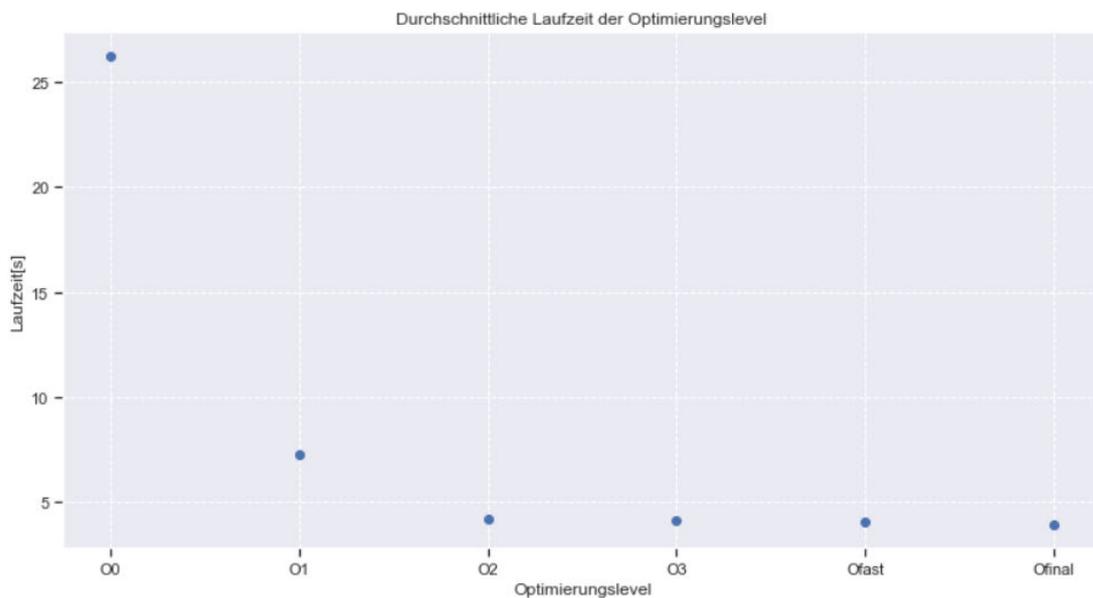


Abbildung 6.10: Laufzeit für verschiedene Optimierungslevel

Für den Ausgangszustand wird die parallelisierte Software betrachtet, die mit dem Optimierungslevel O0 kompiliert wurde. Bereits vom Ausgangszustand zum Optimierungslevel O1 ist eine Laufzeitbeschleunigung von 26,23 s auf 7,24 s zu verzeichnen, dies entspricht einer Beschleunigung des Faktors 3,63. Das Optimierungslevel O2 ist mit einer Laufzeit von 4,22 s um Faktor 6,22 schneller als der Ausgangszustand. Die Laufzeit beträgt für das Optimierungslevel O3 4,11 s und weist eine Beschleunigung des Faktors 6,38 zum Ausgangszustand auf. Das Optimierungslevel Ofast beschleunigt die Laufzeit vom Ausgangszustand um einen Faktor 6,44 auf 4,07 s. Die zusätzliche Konfiguration Ofinal

hat eine durchschnittliche Laufzeit von 3,93 s und ist im Verhältnis zum Ausgangszustand O0 um den Faktor 6,67 schneller. Die genauen Messdaten befinden sich im Anhang A.5, für die Laufzeiten und Beschleunigungen ergibt sich:

Optimierungslevel	O0	O1	O2	O3	Ofast	Ofinal
durchschnittliche Laufzeit[s]	26,23	7,24	4,22	4,11	3,93	
Beschleunigung		3,62	6,22	6,38	6,44	6,67

Es wird deutlich, dass die Optimierungslevel einen großen Einfluss auf die Beschleunigung von Software haben. Für die oben genannte Konfiguration, kompiliert auf dem 8-Kerne-Messsystem, ist eine zusätzliche Laufzeitbeschleunigung vom Faktor 6,67 im Durchschnitt möglich.

### 6.3 Gesamtbeschleunigung auf drei Systemen

Aus den vorherigen Laufzeituntersuchungen geht hervor, dass die Threadanzahl, der Ablaufplan und die Compilereinstellungen einen Einfluss auf die Laufzeit haben. Da die Rechenleistung anderer Systeme stark von dem primären 8-Kerne-Messsystem abweichen kann, wird die Gesamtbeschleunigung auf den Systemen 2-Kerne-Messsystem und 6-Kerne-Messsystem gemessen. Im Folgenden wird die Gesamtbeschleunigung auf diesen Messsystemen geprüft. Folgende Konfigurationen werden zur Bestimmung der Gesamtbeschleunigung genutzt:

Bildgröße conv	3000x3000 Pixel
Bildgröße psf	30x30 Pixel
schedule	dynamic
num_threads	systemabhängig
determine_constants	'p'
method	3,4
Optimierungslevel	O0,O2,Ofast

Die Messung für das 2-Kerne-Messsystem für eine Taktgeschwindigkeit von 2,3 GHz ergibt:

Konfiguration	seriell O0	parallel O0	parallel O2	parallel Ofinal
Laufzeit[s]	422	232	33,42	34
Beschleunigung		1,81	12,62	12,41

Die Messung für das 6-Kerne-Messsystem für eine Taktgeschwindigkeit von 3,5 GHz ergibt:

Konfiguration	seriell O0	parallel O0	parallel O2	parallel Ofinal
Laufzeit[s]	254	47	8,03	7,53
Beschleunigung		5,4	31,63	33,73

Die Messung für das 8-Kerne-Messsystem für eine Taktgeschwindigkeit von 4,1 GHz ergibt:

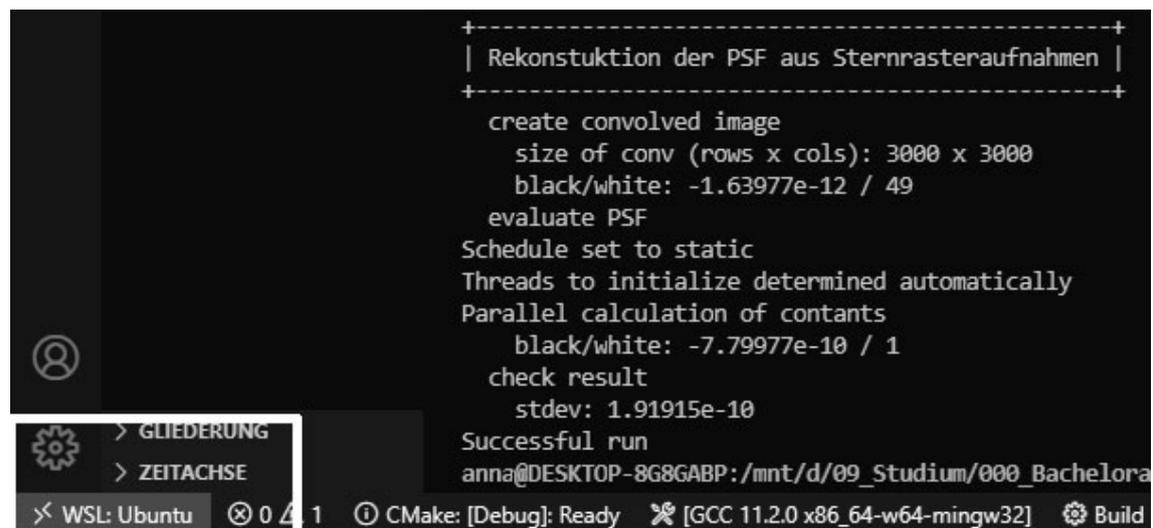
Konfiguration	seriell O0	parallel O0	parallel O2	parallel Ofinal
Laufzeit[s]	172	26,33	4,22	3,92
Beschleunigung		6,53	40,75	43,87

Es wird deutlich, dass sich die serielle Ausführung auf den Messsystemen Linear zu der Taktgeschwindigkeit der Prozessoren verhält. Das 2-Kerne-Messsystem erreicht durch Parallelisierung eine Beschleunigung des Faktors 1,81, die Gesamtbeschleunigung durch Parallelisierung und Compileroptimierungen ist für das Optimierungslevel O2 am höchsten. Die Anwendung lässt sich auf dem 2-Kerne-Messsystem für die genannte Konfiguration, mit dem Optimierungslevel O2 kompiliert, um den Faktor 12,62 beschleunigen. Das 2-Kerne-Messsystem profitiert nicht von den Optimierungen der Level O3 und höher. Die Beschleunigung, die durch Compiler-Optimierungen erzielt werden kann, lässt sich unter anderem auf die Out-Of-Order Ausführung, das simultane Multithreading auf Hardwareebene und Advanced Vector Befehlssatzerweiterung AVX zurückführen. Das 6-Kerne-Messsystem erreicht durch eine Parallelisierung mit OpenMP eine Beschleunigung des Faktors 5,4. Die parallelisierte Software, mit der Konfiguration Ofinal kompiliert, lässt sich um den Faktor 33,76 beschleunigen. Das 6-Kerne-Messsystem profitiert über das Optimierungslevel O2 hinaus von prozessorspezifischen Optimierungen der Konfiguration Ofinal. Das 8-Kerne-Messsystem weist für die Parallelisierung eine Beschleunigung vom Faktor 6,53 auf. Die mit der Konfiguration Ofinal kompilierte Software lässt sich auf

dem 8-Kerne-Messsystem um den Faktor 43,87 beschleunigen. Das 8-Kerne-Messsystem profitiert über das Optimierungslevel O2 hinaus von prozessorspezifischen Optimierungen der Konfiguration Ofinal. Im Gegensatz zum 2-Kerne-Messsystem verfügen das 6-Kerne-Messsystem und das 8-Kerne-Messsystem über weitere Befehlssatzerweiterungen, wie AVX2, Bit Manipulation Instructions und SSE4.a [21, 23, 22]. Diese Befehlssatzerweiterungen führen unter anderem zu einer kleinen Beschleunigung über das Optimierungslevel O2 hinaus.

## 6.4 Test auf Linux Sub-System

Um die Kompatibilität von Linux Systemen zu testen wurde ein Ubuntu Sub-System auf dem 8-Kerne-Messsystem installiert. Folgende Abbildung zeigt die Ausführung des Codes auf einem Linux Sub-System:



```
+-----+
| Rekonstruktion der PSF aus Sternrasteraufnahmen |
+-----+
create convolved image
  size of conv (rows x cols): 3000 x 3000
  black/white: -1.63977e-12 / 49
evaluate PSF
Schedule set to static
Threads to initialize determined automatically
Parallel calculation of contants
  black/white: -7.79977e-10 / 1
check result
  stdev: 1.91915e-10
Successful run
anna@DESKTOP-8G8GABP:/mnt/d/09_Studium/000_Bachelora
```

Abbildung 6.11: Funktionstest auf einem Linux Sub-system

Folglich ist die Parallelisierte Version Linux kompatibel.

## 6.5 Zusammenfassung

Zusammengefasst liegt die durch Parallelisierung erzielte Beschleunigung leicht unter den erwarteten Werten 6.1. Dies ist unter anderem auf die ungenaue Bestimmung des parallelisierbaren Anteils zurückzuführen. Zusätzlich ist anzumerken, dass die Messsysteme durch Hintergrundprozesse beeinflusst werden können und nie ausschließlich die Software ausführt. Die Software lässt sich mittels Parallelisierung und Compileroptimierungen um mindestens Faktor 12,62 für den Intel i3-6100U Prozessor aus dem Baujahr 2015 beschleunigen, folglich ist die Leistungsanforderung für die Beschleunigung auf dem 2-Kerne-Messsystem erfüllt. Das 6-Kerne-Messsystem und das 8-Kerne-Messsystem werden der Leistungsanforderung des Beschleunigungsfaktors ebenfalls gerecht. Die Leistungsanforderung der Laufzeit von unter fünf Sekunden ist ausschließlich auf dem 8-Kerne-Messsystem erfüllt. Für Systeme, die neuer als das 2-Kerne-Messsystem sind, ist folglich mindestens eine Gesamtbeschleunigung vom Faktor 12 zu erwarten. Die Standardabweichung aller Messungen liegt bei maximal  $1,9076 \times 10^{-10}$  und erfüllt somit die Qualitätsanforderung einer maximalen Standardabweichung von  $1 \times 10^{-9}$ . Die Software lässt sich auf Intel sowie AMD Prozessoren kompilieren und beschleunigen. Die Kompatibilität mit Linux-Systemen wurde auf einem Sub-System getestet und ist gewährleistet. Eine Einschränkung der Beschleunigung ist die Abhängigkeit der Anzahl der Prozessorkerne, es kann keine pauschale Beschleunigung genannt werden, die für alle Systeme gültig ist. Des Weiteren konnte die Funktion *solveSLE* auf Grund von Datenabhängigkeiten nicht parallelisiert werden. Eine weitere Einschränkung bildet die Übertragbarkeit der Anwendung, die mit Ofinal kompiliert wurde, auf andere Systeme. Auf Grund der prozessorspezifischen Optimierungen kann nicht gewährleistet werden, dass die Anwendung auf anderen Systemen ausführbar ist. Um sicherzugehen, dass die Anwendung systemübergreifend ausführbar ist, sollte mit dem Optimierungslevel O2 kompiliert werden.

## 7 Fazit & Ausblick

Im Rahmen dieser Ausarbeitung wurde die Software zur Bestimmung der PSF eines Röntgensystems mittels Parallelisierung und CompilerEinstellungen beschleunigt. Die Software wurde auf Engpässe untersucht und an entsprechenden Stellen parallelisiert. Die Funktion *solveSLE* konnte aufgrund von Datenabhängigkeiten im Rahmen der Ausarbeitung nicht parallelisiert werden. Für die einfache Konfiguration der OpenMP Parameter wurde ein zentraler Abschnitt implementiert, in dem die Anzahl der Threads, der Ablaufplan, die Methode zur Bestimmung der Koeffizientenmatrix und die Parallelisierung der Bestimmung der Konstanten gesteuert werden können. Die Parallelisierung mittels OpenMP wurde für die genannten Parameter sowie variierende Bildgrößen der simulierten Röntgenaufnahme und der PSF auf das Laufzeitverhalten untersucht. Die Laufzeituntersuchungen ergaben, dass die Anzahl der Threads, die Methode zur Bestimmung der Koeffizientenmatrix, die Bildgröße der simulierten Röntgenaufnahme und die Bildgröße der PSF allesamt einen signifikanten Einfluss auf die Laufzeit nehmen. Ausschließlich die Laufzeituntersuchung der Ablaufpläne ergab eine geringere Einflussnahme auf die Laufzeit. Die Beschleunigung durch Parallelisierung beläuft sich auf den Faktor 1,81 für einen 2-Kern Prozessor, den Faktor 5,40 für einen 6-Kern Prozessor und den Faktor 6,53 für einen 8-Kern Prozessor. Wichtig ist, dass für die parallelen Abschnitte jeder Thread über eine eigene Kopie der Rechenobjekte verfügt, da es anderweitig zu Zugriffverletzungen kommt. Über die Parallelisierung hinaus konnte die Software durch Compileroptimierungen beschleunigt werden. Dazu wurden sieben Compiler-Konfigurationen implementiert, wobei es sich um eine Debugkonfiguration, eine Konfiguration ohne Optimierungen und fünf Konfigurationen für die Optimierungslevel O1, O2, O3, Ofast und Ofinal handelt. Die Laufzeit wurde in Kombination der Parallelisierung mit OpenMP untersucht. Die maximale Beschleunigung der Software mittels Parallelisierung und Compileroptimierungen beläuft auf 12,61 für den 2-Kern Prozessor, 33,73 für den 6-Kern Prozessor und 43,87 für den 8-Kern Prozessor. Die Anwendungen, die mit der Compiler-Konfiguration Ofinal kompiliert werden, sind nicht garantiert auf andere Systeme übertragbar. Um sicherzustellen, dass die Anwendung systemübergreifend läuft, sollte mit der Konfiguration O2

kompiliert werden. Die Beschleunigung steht im Zusammenhang mit dem Prozessormodell und kann von System zu System variieren. Die Beschleunigungen, die im Rahmen dieser Ausarbeitung erzielt wurden, nutzen die Ressourcen des Prozessors effizient aus und ermöglichen eine Mindestbeschleunigung vom Faktor 12 auf Prozessoren gleichwertig oder neuer als der Intel i3 6100U. Für weitere Optimierung der Laufzeit kann eine Auslagerung der Berechnung auf den Prozessoren der Grafikkarte eine Option zur weiteren Beschleunigung der Berechnung sein, da Grafikkarten über ein Vielfaches an Rechenkernen im Gegensatz zu einem Mehrkernprozessor verfügen. Des Weiteren würde die Laufzeit durch die Parallelisierung der Funktion *solveSLE* profitieren, für die zunächst bestehende Datenabhängigkeiten entfernt werden müssen. Von Interesse könnten Laufzeituntersuchungen sein, die einerseits die Software für die 32 Bit Befehlssatzlänge untersuchen oder das Laufzeitverhalten für den Fall, dass alle Parameter von Datentyp `double` auf `float` geändert werden. Da `float` weniger Nachkommastellen hat, könnten Berechnungen möglicherweise schneller, folglich aber auch ungenauer durchgeführt werden. Für diese Untersuchung wäre ebenfalls der Einfluss dieser Änderung auf die Standardabweichung von Interesse.

# Literaturverzeichnis

- [1] AIKEN, Prof.: *OpenMP - Lecture 13*. – URL <https://web.stanford.edu/class/cs315b/lectures/lecture13.pdf>. – (accessed: 11.04.2022)
- [2] BAKHVALOV, Denis: *Performance Analysis and Tuning on Modern CPUs*. easy-perf.net, 2020. – URL [https://book.easyperf.net/perf\\_book](https://book.easyperf.net/perf_book). – ISBN 979-8575614234
- [3] CODE, Visual S.: *Hundreds of programming languages supported*. 2022. – URL <https://code.visualstudio.com/docs/languages/overview>. – (accessed: 11.04.2022)
- [4] DÖSSEL, Olaf: *Bildgebende Verfahren in der Medizin*. Springer, 2016. – URL <https://link.springer.com/book/10.1007/978-3-642-54407-1>. – ISBN 978-3-642-54406-4
- [5] GENTOO.ORG: *GCC optimization*. 2022. – URL [https://wiki.gentoo.org/wiki/GCC\\_optimization](https://wiki.gentoo.org/wiki/GCC_optimization). – (accessed: 11.04.2022)
- [6] HARTMUT ERNST, Gerd B.: *Grundkurs Informatik*. Springer, 2020. – URL <https://link.springer.com/book/10.1007/978-3-658-30331-0>. – ISBN 978-3-658-30331-0
- [7] HESS, Prof. Dr. R.: *Ermittlung der PSF aus einer Sternrasteraufnahme mittels Methode kleinster Quadrate*
- [8] HESS, Prof. Dr. R.: *Programmieren II*. 2020. – URL <http://rrhess.de/pdf/Skript-PR-II.pdf>. – (accessed: 11.04.2022)
- [9] MACLAREN, Nick: *Introduction to OpenMP - Critical Guidelines*. 2011. – URL [https://www-uxsup.csx.cam.ac.uk/courses/moved.OpenMP/paper\\_6.pdf](https://www-uxsup.csx.cam.ac.uk/courses/moved.OpenMP/paper_6.pdf). – (accessed: 11.04.2022)

- [10] MBQ: *Modulationsübertragungsfunktion, grafische Darstellung.* – URL [https://commons.wikimedia.org/wiki/File:MBq\\_MTF.jpg](https://commons.wikimedia.org/wiki/File:MBq_MTF.jpg). – (accessed: 11.04.2022)
- [11] MITTON, Richard: *Very Sleepy.* 2021. – URL <http://www.codersnotes.com/sleepy/>. – (accessed: 11.04.2022)
- [12] OPENMP.ORG: *omp set num threads.* 2018. – URL <https://www.openmp.org/spec-html/5.0/openmpsul10.html#x147-6380003.2.1>. – (accessed: 11.04.2022)
- [13] OPENMP.ORG: *omp set schedule.* 2018. – URL <https://www.openmp.org/spec-html/5.0/openmpsul21.html>. – (accessed: 11.04.2022)
- [14] PROF. DR, Dieter S.: *Skript Medizinphysik: Röntgendiagnostik.* – URL [https://qnap.e3.physik.tu-dortmund.de/suter/Vorlesung/Medizinphysik\\_09/6\\_Roentgen.pdf](https://qnap.e3.physik.tu-dortmund.de/suter/Vorlesung/Medizinphysik_09/6_Roentgen.pdf). – (accessed: 11.04.2022)
- [15] RORDRIGUES, Caio: *CPP/C++ Compiler Flags and Options.* 2021. – URL <https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html>. – (accessed: 11.04.2022)
- [16] SIMON HOFFMANN, Rainer L.: *OpenMP.* Springer, 2008. – URL <https://link.springer.com/book/10.1007/978-3-540-73123-8>. – ISBN 978-3-540-73122-1
- [17] TREX2001: *22. Dezember 1895 - Erste Röntgenaufnahme einer Hand.* – URL <https://de.wikipedia.org/wiki/Datei:Bremstrahlung.svg#/media/Datei:Bremstrahlung.svg>. – (accessed: 11.04.2022)
- [18] TWEAKPC.DE: *AMD Ryzen 3000 CPUs die Technik im Detail.* – URL [https://www.tweakpc.de/hardware/tests/cpu/amd\\_ryzen\\_7\\_3700x\\_ryzen\\_9\\_3900x/s02.php](https://www.tweakpc.de/hardware/tests/cpu/amd_ryzen_7_3700x_ryzen_9_3900x/s02.php). – (accessed: 11.04.2022)
- [19] WDR.DE: *22. Dezember 1895 - Erste Röntgenaufnahme einer Hand.* – URL <https://www1.wdr.de/stichtag/stichtag-erste-roentgenaufnahme-100.html>
- [20] WIKI, ChessProgramming: *Bit Manipulation Instruction Sets — Wikipedia, die freie Enzyklopädie.* 2019. – URL [https://de.wikipedia.org/w/index.php?title=Bit\\_Manipulation\\_Instruction\\_Sets&oldid=194980433](https://de.wikipedia.org/w/index.php?title=Bit_Manipulation_Instruction_Sets&oldid=194980433). – (accessed: 11.04.2022)

- [21] WIKICHIP: *Core i3-6100U - Intel*. 2017. – URL [https://en.wikichip.org/wiki/intel/core\\_i3/i3-6100u](https://en.wikichip.org/wiki/intel/core_i3/i3-6100u). – (accessed: 11.04.2022)
- [22] WIKICHIP: *Ryzen 7 3700X - AMD*. 2020. – URL [https://en.wikichip.org/wiki/amd/ryzen\\_7/3700x](https://en.wikichip.org/wiki/amd/ryzen_7/3700x). – (accessed: 11.04.2022)
- [23] WIKICHIP: *Ryzen 5 2600 - AMD*. 2021. – URL [https://en.wikichip.org/wiki/amd/ryzen\\_5/2600](https://en.wikichip.org/wiki/amd/ryzen_5/2600). – (accessed: 11.04.2022)
- [24] WIKIPEDIA: *Registerumbenennung* — *Wikipedia, die freie Enzyklopädie*. 2017. – URL <https://de.wikipedia.org/w/index.php?title=Registerumbenennung&oldid=163487681>. – (accessed: 11.04.2022)
- [25] WIKIPEDIA: *Datenabhängigkeit* — *Wikipedia, die freie Enzyklopädie*. 2018. – URL <https://de.wikipedia.org/w/index.php?title=Datenabh%C3%A4ngigkeit&oldid=183776833>. – (accessed: 11.04.2022)
- [26] WIKIPEDIA: *Siemensstern* — *Wikipedia, die freie Enzyklopädie*. 2020. – URL <https://de.wikipedia.org/w/index.php?title=Siemensstern&oldid=199812038>. – (accessed: 11.04.2022)
- [27] WIKIPEDIA: *Doxygen* — *Wikipedia, die freie Enzyklopädie*. 2021. – URL <https://de.wikipedia.org/w/index.php?title=Doxygen&oldid=216010254>. – (accessed: 11.04.2022)
- [28] WIKIPEDIA: *OpenMP* — *Wikipedia, die freie Enzyklopädie*. 2021. – URL <https://de.wikipedia.org/w/index.php?title=OpenMP&oldid=217641231>. – (accessed: 11.04.2022)
- [29] WIKIPEDIA: *Simultaneous Multithreading* — *Wikipedia, die freie Enzyklopädie*. 2021. – URL [https://de.wikipedia.org/w/index.php?title=Simultaneous\\_Multithreading&oldid=209006969](https://de.wikipedia.org/w/index.php?title=Simultaneous_Multithreading&oldid=209006969). – (accessed: 11.04.2022)
- [30] WIKIPEDIA: *Speedup* — *Wikipedia, die freie Enzyklopädie*. 2021. – URL <https://de.wikipedia.org/w/index.php?title=Speedup&oldid=207722806>. – (accessed: 11.04.2022)
- [31] WIKIPEDIA: *Streaming SIMD Extensions* — *Wikipedia, die freie Enzyklopädie*. 2021. – URL [https://de.wikipedia.org/w/index.php?title=Streaming\\_SIMD\\_Extensions&oldid=215304905](https://de.wikipedia.org/w/index.php?title=Streaming_SIMD_Extensions&oldid=215304905). – (accessed: 11.04.2022)

- [32] WIKIPEDIA: *Advanced Vector Extensions* — *Wikipedia, die freie Enzyklopädie*. 2022. – URL [https://de.wikipedia.org/w/index.php?title=Advanced\\_Vector\\_Extensions&oldid=221403719](https://de.wikipedia.org/w/index.php?title=Advanced_Vector_Extensions&oldid=221403719). – (accessed: 11.04.2022)
- [33] WIKIPEDIA: *Bit-Twiddling*. 2022. – URL <https://www.chessprogramming.org/Bit-Twiddling#BitManipulation>. – (accessed: 11.04.2022)
- [34] WIKIPEDIA: *Bremsstrahlung* — *Wikipedia, die freie Enzyklopädie*. 2022. – URL <https://de.wikipedia.org/w/index.php?title=Bremsstrahlung&oldid=221845714>. – (accessed: 11.04.2022)
- [35] WIKIPEDIA: *GNU Compiler Collection* — *Wikipedia, die freie Enzyklopädie*. 2022. – URL [https://de.wikipedia.org/w/index.php?title=GNU\\_Compiler\\_Collection&oldid=220258265](https://de.wikipedia.org/w/index.php?title=GNU_Compiler_Collection&oldid=220258265). – (accessed: 11.04.2022)
- [36] ZÜRICH, Vetsuisse-Fakultäten B. und: *Strichfokus-Prinzip*. – URL <https://vetsuisse.com/vet-impl/lernmodule/htmls/slide.html?radiosurfvet|radgeneral|technics|roentgentube|6>. – (accessed: 11.04.2022)

# A Anhang

## A.1 Messwerte: Variierende Bildgröße conv

Für die Konfiguration:

Bildgröße conv	variiert
Bildgröße psf	30x30 Pixel
schedule	static
num_threads	16
determine_constants	's' für method 1&3, 'p' für method 2&4
method	1,2,3,4
Optimierungslevel	O0
Stichprobenmenge	10 pro Bildgröße

Bildgröße [px]	method = 1		method = 3		method = 2	
	Laufzeit [s]	∅Laufzeit[s]	Laufzeit [s]	∅Laufzeit[s]	Laufzeit [s]	∅ Laufzeit[s]
100x100	32.187	32.469	1.890	1.910	20.890	20.164
100x100	34.078		1.938		20.344	
100x100	33.703		1.906		19.453	
100x100	33.094		1.907		20.094	
100x100	32.890		1.891		20.266	
100x100	30.156		1.922		19.531	
100x100	30.391		1.906		20.344	
100x100	33.953		1.922		20.250	
100x100	31.766		1.907		20.031	
100x100	32.469		1.906		20.438	
200x200	130.297	124.3985	2.500	2.4922	38.922	38.220
200x200	118.766		2.484		38.953	
200x200	131.516		2.500		38.922	
200x200	115.578		2.485		38.765	
200x200	126.875		2.500		39.156	
200x200	121.219		2.484		37.921	
200x200	118.515		2.484		38.297	
200x200	121.515		2.500		36.984	
200x200	127.610		2.516		37.141	
200x200	132.094		2.469		37.141	
300x300	299.860	277.8939	3.453	3.4565	59.641	53.888
300x300	277.625		3.438		60.766	
300x300	258.016		3.438		58.812	
300x300	273.922		3.485		60.000	
300x300	274.656		3.453		59.172	
300x300	271.172		3.438		61.078	
300x300	276.437		3.469		59.500	
300x300	275.297		3.422		60.250	
300x300	278.297		3.438		59.656	
300x300	293.657		3.531		60.078	
400x400	525.360	498.5922	4.750	4.8092	81.812	82.396
400x400	525.078		4.828		82.500	
400x400	469.578		4.812		83.703	
400x400	471.437		4.734		83.438	
400x400	478.890		4.750		83.109	
400x400	463.657		4.750		81.000	
400x400	531.172		4.734		82.531	
400x400	502.141		4.797		82.641	
400x400	493.203		5.109		80.937	
400x400	525.406		4.828		82.293	
500x500	793.485	791.097625	6.515	6.4735	117.688	117.683
500x500	767.375		6.391		117.906	
500x500	833.890		6.453		118.093	
500x500	713.047		6.469		117.907	
500x500	783.187		6.469		117.828	
500x500	806.125		6.515		117.250	
500x500	826.656		6.469		116.953	
500x500	805.016		6.469		117.282	

500x500		6.485		118.391	
500x500		6.500		117.531	
600x600	1138	8.547	8.5657	165.313	166.942
600x600		8.531		166.547	
600x600		8.657		166.188	
600x600		8.469		166.844	
600x600		8.500		167.578	
600x600		8.500		167.907	
600x600		8.579		165.750	
600x600		8.562		167.672	
600x600		8.687		166.609	
600x600		8.625		169.016	
700x700		10.984	10.9718	221.406	225.350
700x700	1549	11.015		225.422	
700x700		10.953		224.671	
700x700		10.906		224.797	
700x700		11.016		225.344	
700x700		11.078		225.781	
700x700		10.860		225.906	
700x700		11.000		227.531	
700x700		10.969		227.156	
700x700		10.937		225.484	
800x800	2024	13.890		289.516	262.516
800x800		13.703		292.843	
800x800		13.859		292.797	
800x800		13.765	13.8373	291.625	
800x800		13.828		291.984	
800x800		13.922		288.079	
800x800		13.922		290.922	
800x800		13.657		292.672	
800x800		13.937		294.719	
800x800		13.890		292.546	
900x900	2571	16.984	17.0218	361.797	370.542
900x900		17.328		370.375	
900x900		17.000		371.203	
900x900		16.953		371.250	
900x900		17.141		370.031	
900x900		17.000		368.563	
900x900		16.906		373.078	
900x900		16.953		374.703	
900x900		16.922		369.859	
900x900		17.031		374.563	
1000x1000	3162.56	20.485	20.587	451.343	450.367
1000x1000		20.656		448.610	
1000x1000		20.610		452.656	
1000x1000		20.672		452.109	
1000x1000		20.641		447.922	
1000x1000		20.375		451.266	
1000x1000		20.640		449.891	
1000x1000		20.609		452.297	

1000x1000		20.594		449.844
1000x1000		171.875	172.5187	447.735
3000x3000	28463	176.531		
3000x3000		170.265		
3000x3000		172.375		
3000x3000		170.750		
3000x3000		172.938		
3000x3000		172.188		
3000x3000		173.203		
3000x3000		172.703		
3000x3000		172.359		

\*Approximiert

method = 4	
Laufzeit [s]	∅ Laufzeit[s]
1.703	1.725
1.719	
1.734	
1.735	
1.719	
1.735	
1.718	
1.718	
1.719	
1.750	
1.812	1.8124
1.797	
1.813	
1.797	
1.812	
1.796	
1.813	
1.828	
1.828	
1.828	
1.954	1.9535
1.954	
1.922	
1.953	
1.938	
1.954	
1.953	
1.985	
1.985	
1.937	
2.125	2.1281
2.140	
2.125	
2.125	
2.125	
2.125	
2.125	
2.125	
2.125	
2.141	
2.391	2.3921
2.406	
2.375	
2.375	
2.375	
2.406	
2.390	
2.406	

2.406	
2.391	
2.672	2.6827
2.719	
2.671	
2.672	
2.688	
2.703	
2.672	
2.734	
2.656	
2.640	
3.141	3.0861
3.078	
3.015	
3.110	
3.079	
3.110	
3.094	
3.094	
3.062	
3.078	
3.390	3.4266
3.469	
3.422	
3.422	
3.453	
3.438	
3.422	
3.375	
3.469	
3.406	
4.094	4.0032
4.000	
4.032	
3.953	
4.000	
3.985	
4.047	
3.984	
4.000	
3.937	
4.500	4.4876
4.610	
4.484	
4.515	
4.500	
4.454	
4.484	
4.485	

4.422

4.422

## A.2 Messwerte: Variierende Bildgröße psf

Für die Konfiguration:

Bildgröße conv	100x100 Pixel
Bildgröße psf	variiert
schedule	static
num_threads	16
determine_constants	's' für method 1&3, 'p' für method 2&4
method	1,2,3,4
Optimierungslevel	00
Stichprobenmenge	5 pro Bildgröße

PSF Größe [Px]	method = 1		method = 3		method = 2	
	Laufzeit [s]	∅Laufzeit[s]	Laufzeit [s]	∅Laufzeit[s]	Laufzeit [s]	∅Laufzeit[s]
225	2.015	2.1344	0.109	0.1188	4.391	4.553
225	2.188		0.125		4.484	
225	2.281		0.125		4.718	
225	2.016		0.110		4.422	
225	2.172		0.125		4.750	
400	6.328	6.275	0.296	0.2872	8.172	8.285
400	6.703		0.281		8.782	
400	6.109		0.282		8.000	
400	6.172		0.281		8.469	
400	6.063		0.296		8.000	
900	32.187	33.1904	1.890	1.9064	1.984	16.565
900	34.078		1.938		20.375	
900	33.703		1.906		20.312	
900	33.094		1.907		19.719	
900	32.89		1.891		20.437	
1600	105.672	102.7502	9.906	9.925	42.000	42.284
1600	100.141		9.953		41.922	
1600	105.032		9.937		43.047	
1600	101.14		9.922		42.250	
1600	101.766		9.907		42.203	
2500	266.563	264.3592	37.000	36.9032	89.812	88.369
2500	262.562		36.859		88.515	
2500	267.515		36.891		87.703	
2500	265.656		36.844		87.047	
2500	259.5		36.922		88.766	
121	0.578	0.575	0.078	0.0718	2.704	2.628
121	0.578		0.078		2.515	
121	0.578		0.078		2.688	
121	0.594		0.078		2.515	
121	0.547		0.047		2.718	

method = 4	
Laufzeit [s]	∅Laufzeit[s]
0.078	0.078
0.078	
0.078	
0.094	
0.062	
0.203	0.203
0.219	
0.203	
0.219	
0.172	
1.718	1.719
1.734	
1.703	
1.735	
1.703	
9.453	9.482
9.469	
9.516	
9.454	
9.516	
36.046	36.141
36.110	
36.109	
36.516	
35.922	
0.063	0.066
0.078	
0.078	
0.078	
0.032	

### A.3 Messwerte: Variierender Schedule

Für die Konfiguration:

Bildgröße conv	3000x3000 Pixel
Bildgröße psf	30x30 Pixel
schedule	variiert
num_threads	16
determine_constants	'p'
method	4
Optimierungslevel	O0
Stichprobenmenge	20 pro Bildgröße

schedule	Laufzeit [s]	ØLaufzeit[s]
static	26.422	26.15765
static	26.328	
static	26.219	
static	26.375	
static	26.531	
static	26.344	
static	26.391	
static	26.187	
static	26.125	
static	26.469	
static	25.656	
static	26.109	
static	25.843	
static	25.953	
static	25.969	
static	26.062	
static	26.015	
static	26.062	
static	26.078	
static	26.015	
dynamic	26.250	26.0312
dynamic	25.797	
dynamic	26.250	
dynamic	26.218	
dynamic	25.766	
dynamic	26.219	
dynamic	25.891	
dynamic	25.843	
dynamic	25.906	
dynamic	26.172	
dynamic	26.093	
dynamic	25.985	
dynamic	26.031	
dynamic	25.985	
dynamic	25.969	
dynamic	26.015	
dynamic	25.812	
dynamic	25.891	
dynamic	26.187	
dynamic	26.347	
guided	26.031	26.475
guided	26.578	
guided	26.453	
guided	26.516	
guided	26.297	
guided	26.453	
guided	26.313	
guided	26.328	
guided	27.328	

guided	26.453	
guided	26.015	
guided	26	
guided	26	
guided	25.969	
guided	26.343	
guided	26.454	
guided	26.078	
guided	25.953	
guided	26.047	
guided	25.906	
auto	26.359	26.3031
auto	26.328	
auto	26.390	
auto	26.375	
auto	26.297	
auto	26.375	
auto	26.563	
auto	26.094	
auto	26.125	
auto	26.125	
auto	25.953	
auto	25.969	
auto	26.063	
auto	25.906	
auto	26.062	
auto	26.015	
auto	26.109	
auto	25.812	
auto	26.032	
auto	26.062	

## A.4 Messwerte: Variierende Threadanzahl

Für die Konfiguration:

Bildgröße conv	3000x3000 Pixel
Bildgröße psf	30x30 Pixel
schedule	dynamic
num_threads	variiert
determine_constants	'p'
method	4
Optimierungslevel	O0
Stichprobenmenge	10 pro Threadanzahl

num_threads	Laufzeit [s]	ØLaufzeit[s]
2	94.422	94.4408
2	96.750	
2	95.204	
2	91.937	
2	91.359	
2	95.079	
2	96.688	
2	92.672	
2	97.219	
2	93.078	
4	51.844	51.3423
4	49.938	
4	49.672	
4	53.031	
4	51.219	
4	50.906	
4	51.782	
4	50.282	
4	52.515	
4	52.234	
8	32.157	32.4955
8	32.172	
8	32.188	
8	34.016	
8	31.735	
8	32.329	
8	33.640	
8	32.140	
8	31.859	
8	32.719	
16	26.050	26.1989
16	26.219	
16	26.219	
16	26.328	
16	26.360	
16	26.297	
16	26.125	
16	26.141	
16	26.125	
16	26.125	

## A.5 Messwerte: Variierende Optimierungslevel

Für die Konfiguration:

Bildgröße conv	3000x3000 Pixel
Bildgröße psf	30x30 Pixel
schedule	dynamic
num_threads	16
determine_constants	'p'
method	4
Optimierungslevel	variiert
Stichprobenmenge	10 pro Threadanzahl

Optimierungslevel	Laufzeit [s]	ØLaufzeit[s]
O0	26.328	26.227
O0	26.219	
O0	26.219	
O0	26.328	
O0	26.360	
O0	26.297	
O0	26.125	
O0	26.141	
O0	26.125	
O0	26.125	
O1	7.218	7.236
O1	7.203	
O1	7.219	
O1	7.765	
O1	6.891	
O1	7.078	
O1	7.156	
O1	7.391	
O1	7.125	
O1	7.313	
O2	4.422	4.224
O2	4.032	
O2	4.110	
O2	4.234	
O2	4.344	
O2	4.172	
O2	4.141	
O2	4.125	
O2	4.218	
O2	4.437	
O3	4.062	4.111
O3	4.203	
O3	4.156	
O3	4.141	
O3	4.156	
O3	4.234	
O3	4.016	
O3	3.969	
O3	4.047	
O3	4.125	
Ofast	4.047	4.077
Ofast	3.907	
Ofast	3.938	
Ofast	4.250	
Ofast	3.968	
Ofast	4.109	
Ofast	4.219	
Ofast	3.985	
Ofast	4.171	

Ofast	4.172	
Ofinal	3.890	3.930
Ofinal	3.782	
Ofinal	4.219	
Ofinal	4.016	
Ofinal	3.765	
Ofinal	3.829	
Ofinal	4.094	
Ofinal	3.938	
Ofinal	3.969	
Ofinal	3.797	

## A.6 Code: Methoden zur Bestimmung der Koeffizientenmatrix

```

1  if (method == 1) {
2      for (unsigned r1 = 0; r1 < meas.nRow; r1++) {
3          cout << "      " << r1 * 100.0 / meas.nRow << " %      \r" << flush;
4          for (unsigned c1 = 0; c1 < meas.nCol; c1++) {
5              for (unsigned i = 0; i < (unsigned)M; i++) { // row of matrix
6                  double tmp = star[r1 + i / psfCols][c1 + i % psfCols];
7                  unsigned j = 0;
8                  for (unsigned row = r1; row < r1 + psfRows; row++) {
9                      for (unsigned col = c1; col < c1 + psfCols; col++) {
10                         m[i][j++] += tmp * star[row][col];
11                     }
12                 }
13             }
14         }
15     }
16 }
17 // method 2 - Direkte Implementierung, parallelisiert
18 else if (method == 2) {
19     #pragma omp parallel for
20     for (unsigned i = 0; i < (unsigned)M; i++) {
21         vector<double> tmpvec;
22         tmpvec.resize(M,0.0); // temporaerer Rechenvektor
23         for (unsigned r1 = 0; r1 < meas.nRow; r1++) {
24             cout << "      " << r1 * 100.0 / meas.nRow << " %      \r" << flush;
25             for (unsigned c1 = 0; c1 < meas.nCol; c1++) {
26                 // row of matrix
27                 double star1 = star[r1 + i / psfCols][c1 + i % psfCols];
28                 unsigned j = 0;
29                 for (unsigned row = r1; row < r1 + psfRows; row++) {
30                     for (unsigned col = c1; col < c1 + psfCols; col++) {
31                         double star2 = star[row][col];
32                         tmpvec[j] += star1 * star2;
33                         j++;
34                     }
35                 }
36             }
37         }
38         for(unsigned x=0; x<(unsigned)M; x++)
39             {
40                 m[i][x] = tmpvec[x];
41             }
42     }
43 }
44 }
45
46 // method 3 - Optimierte Methode by Hess
47 else if (method == 3) {
48     cMedImage<double> tmp; // temporary image for calculations
49     vector<double> row; // row vector for calculations
50     tmp.create(2 * psfRows - 1, 2 * psfCols - 1);
51     row.resize(2 * psfCols - 1);
52     for (int dr = 0; dr < (int)psfRows; dr++) {
53         for (int dc = (dr > 0 ? -(int)psfCols + 1 : 0); dc < (int)psfCols; dc++) {
54
55             // set temporary matrix to zero
56             tmp = 0;
57             // fill four corners of temporary matrix
58             for (int r = 0; r < psfRows - dr - 1; r++) {
59                 for (int c = 0; c < psfCols - ABS(dc) - 1; c++) {
60                     tmp[r][c] = star[r][c + (dc < 0 ? -dc : 0)]
61                         * star[r + dr][c + (dc > 0 ? dc : 0)];
62                     tmp[r + psfRows][c]
63                         = star[r + meas.nRow][c + (dc < 0 ? -dc : 0)]
64                         * star[r + dr + meas.nRow][c + (dc > 0 ? dc : 0)];

```

```

65     tmp[r][c + psfCols]
66     = star[r][c + (dc < 0 ? -dc : 0) + meas.nCol]
67     * star[r + dr][c + (dc > 0 ? dc : 0) + meas.nCol];
68     tmp[r + psfRows][c + psfCols]
69     = star[r + meas.nRow][c + (dc < 0 ? -dc : 0) + meas.nCol]
70     * star[r + dr + meas.nRow][c + (dc > 0 ? dc : 0) + meas.nCol];
71     }
72     }
73     // fill top and bottom middle part of temporary matrix
74     for (int r = 0; r < psfRows - dr - 1; r++) {
75         for (int c = psfCols - ABS(dc) - 1; c < (int)meas.nCol; c++) {
76             tmp[r][psfCols - 1] += star[r][c + (dc < 0 ? -dc : 0)]
77             * star[r + dr][c + (dc > 0 ? dc : 0)];
78             tmp[r + psfRows][psfCols - 1]
79             += star[r + meas.nRow][c + (dc < 0 ? -dc : 0)]
80             * star[r + dr + meas.nRow][c + (dc > 0 ? dc : 0)];
81         }
82     }
83     // fill left and right middle part of temporary matrix
84     for (int c = 0; c < psfCols - ABS(dc) - 1; c++) {
85         for (int r = psfRows - dr - 1; r < (int)meas.nRow; r++) {
86             tmp[psfRows - 1][c] += star[r][c + (dc < 0 ? -dc : 0)]
87             * star[r + dr][c + (dc > 0 ? dc : 0)];
88             tmp[psfRows - 1][c + psfCols]
89             += star[r][c + (dc < 0 ? -dc : 0) + meas.nCol]
90             * star[r + dr][c + meas.nCol + (dc > 0 ? dc : 0)];
91         }
92     }
93     // fill middle element of matrix
94     for (int r = psfRows - dr - 1; r < (int)meas.nRow; r++) {
95         for (int c = psfCols - ABS(dc) - 1; c < (int)meas.nCol; c++) {
96             tmp[psfRows - 1][psfCols - 1] += star[r][c + (dc < 0 ? -dc : 0)]
97             * star[r + dr][c + (dc > 0 ? dc : 0)];
98         }
99     }
100
101     // prepare row vector
102     for (int c = 0; c < psfCols - ABS(dc) - 1; c++) {
103         row[c] = tmp[psfRows - 1][c];
104         row[psfCols + c] = tmp[psfRows - 1][psfCols + c];
105         for (int r = 0; r < psfRows - dr - 1; r++) {
106             row[c] += tmp[r][c];
107             row[psfCols + c] += tmp[r][psfCols + c];
108         }
109     }
110     row[psfCols - 1] = tmp[psfRows - 1][psfCols - 1];
111     for (int r = 0; r < psfRows - dr - 1; r++)
112     row[psfCols - 1] += tmp[r][psfCols - 1];
113
114     // evaluate coefficients row by row
115     for (int r = 0; r < psfRows - dr; r++) {
116
117         // from second row on adjust row vector
118         if (r > 0) {
119             for (int c = 0; r > 0 && c < psfCols - ABS(dc) - 1; c++) {
120                 row[c] += tmp[psfRows + r - 1][c] - tmp[r - 1][c];
121                 row[psfCols + c] += tmp[psfRows + r - 1][psfCols + c] - tmp[r - 1][psfCols + c];
122             }
123             row[psfCols - 1] += tmp[psfRows + r - 1][psfCols - 1] - tmp[r - 1][psfCols - 1];
124         }
125
126         // first coefficient in row
127         double sum = 0.0;
128         for (int c = 0; c < psfCols - ABS(dc) - 1; c++)
129             sum += row[c];
130         sum += row[psfCols - 1];
131         unsigned i = r * psfCols + (dc < 0 ? -dc : 0);
132         unsigned j = (r + dr) * psfCols + (dc > 0 ? dc : 0);
133         m[i][j] = sum;

```

```

134     m[j][i] = sum;
135
136     // all other coefficients in row
137     for (int c = 1; c < psfCols - ABS(dc); c++) {
138         sum -= row[c - 1];
139         sum += row[psfCols + c - 1];
140         i++;
141         j++;
142         m[i][j] = sum;
143         m[j][i] = sum;
144     }
145 }
146 }
147 }
148 }
149
150 // method 4 - Optimierte Methode by Hess, parallelisiert
151 else if (method == 4) {
152     #pragma omp parallel for
153     for (int dr = 0; dr < (int)psfRows; dr++) {
154         //omp_set_num_threads(16);
155         cMedImage<double> tmp;
156         tmp.create(2 * psfRows - 1, 2 * psfCols - 1); // temporaere Matrix fuer Berechnungen,
// jeder Thread legt eine private Kopie an
157         vector<double> row;
158         row.resize(2 * psfCols - 1); // Row Vektor fuer beschleunigte Berechnung, jeder Thread
// legt eine private Kopie an
159         for (int dc = (dr > 0 ? -(int)psfCols + 1 : 0); dc < (int)psfCols; dc++) {
160             // set temporary matrix to zero
161             tmp = 0;
162             // fill four corners of temporary matrix
163             for (int r = 0; r < psfRows - dr - 1; r++) {
164                 for (int c = 0; c < psfCols - ABS(dc) - 1; c++) {
165                     tmp[r][c] = star[r][c + (dc < 0 ? -dc : 0)]
166                         * star[r + dr][c + (dc > 0 ? dc : 0)];
167                     tmp[r + psfRows][c]
168                         = star[r + meas.nRow][c + (dc < 0 ? -dc : 0)]
169                         * star[r + dr + meas.nRow][c + (dc > 0 ? dc : 0)];
170                     tmp[r][c + psfCols]
171                         = star[r][c + (dc < 0 ? -dc : 0) + meas.nCol]
172                         * star[r + dr][c + (dc > 0 ? dc : 0) + meas.nCol];
173                     tmp[r + psfRows][c + psfCols]
174                         = star[r + meas.nRow][c + (dc < 0 ? -dc : 0) + meas.nCol]
175                         * star[r + dr + meas.nRow][c + (dc > 0 ? dc : 0) + meas.nCol];
176                 }
177             }
178             // fill top and bottom middle part of temporary matrix
179             for (int r = 0; r < psfRows - dr - 1; r++) {
180                 for (int c = psfCols - ABS(dc) - 1; c < (int)meas.nCol; c++) {
181                     tmp[r][psfCols - 1] += star[r][c + (dc < 0 ? -dc : 0)]
182                         * star[r + dr][c + (dc > 0 ? dc : 0)];
183                     tmp[r + psfRows][psfCols - 1]
184                         += star[r + meas.nRow][c + (dc < 0 ? -dc : 0)]
185                         * star[r + dr + meas.nRow][c + (dc > 0 ? dc : 0)];
186                 }
187             }
188             // fill left and right middle part of temporary matrix
189             for (int c = 0; c < psfCols - ABS(dc) - 1; c++) {
190                 for (int r = psfRows - dr - 1; r < (int)meas.nRow; r++) {
191                     tmp[psfRows - 1][c] += star[r][c + (dc < 0 ? -dc : 0)]
192                         * star[r + dr][c + (dc > 0 ? dc : 0)];
193                     tmp[psfRows - 1][c + psfCols]
194                         += star[r][c + (dc < 0 ? -dc : 0) + meas.nCol]
195                         * star[r + dr][c + meas.nCol + (dc > 0 ? dc : 0)];
196                 }
197             }
198             // fill middle element of matrix
199             for (int r = psfRows - dr - 1; r < (int)meas.nRow; r++) {
200                 for (int c = psfCols - ABS(dc) - 1; c < (int)meas.nCol; c++) {

```

```

201     tmp[psfRows - 1][psfCols - 1] += star[r][c + (dc < 0 ? -dc : 0)]
202     * star[r + dr][c + (dc > 0 ? dc : 0)];
203 }
204 }
205
206 // prepare row vector
207 for (int c = 0; c < psfCols - ABS(dc) - 1; c++) {
208     row[c] = tmp[psfRows - 1][c];
209     row[psfCols + c] = tmp[psfRows - 1][psfCols + c];
210     for (int r = 0; r < psfRows - dr - 1; r++) {
211         row[c] += tmp[r][c];
212         row[psfCols + c] += tmp[r][psfCols + c];
213     }
214 }
215 row[psfCols - 1] = tmp[psfRows - 1][psfCols - 1];
216 for (int r = 0; r < psfRows - dr - 1; r++)
217 row[psfCols - 1] += tmp[r][psfCols - 1];
218
219 // evaluate coefficients row by row
220 for (int r = 0; r < psfRows - dr; r++) {
221     // from second row on adjust row vector
222     if (r > 0) {
223         for (int c = 0; r > 0 && c < psfCols - ABS(dc) - 1; c++) {
224             row[c] += tmp[psfRows + r - 1][c] - tmp[r - 1][c];
225             row[psfCols + c] += tmp[psfRows + r - 1][psfCols + c] - tmp[r - 1][psfCols + c];
226         }
227         row[psfCols - 1] += tmp[psfRows + r - 1][psfCols - 1] - tmp[r - 1][psfCols - 1];
228     }
229     // first coefficient in row
230     double sum = 0.0;
231     for (int c = 0; c < psfCols - ABS(dc) - 1; c++)
232         sum += row[c];
233     sum += row[psfCols - 1];
234     unsigned i = r * psfCols + (dc < 0 ? -dc : 0);
235     unsigned j = (r + dr)*psfCols + (dc > 0 ? dc : 0);
236     m[i][j] = sum;
237     m[j][i] = sum;
238     // all other coefficients in row
239     for (int c = 1; c < psfCols - ABS(dc); c++) {
240         sum -= row[c - 1];
241         sum += row[psfCols + c - 1];
242         i++;
243         j++;
244         m[i][j] = sum;
245         m[j][i] = sum;
246     }
247 }
248 }
249 }
250 }
251 }

```

Listing A.1: Code für die Bestimmung der Koeffizientenmatrix

## A.7 Code: OpenMP Schedule

```

1  bool parallel = (method == 2) || (method == 4);
2
3  //Setting schedule for OpenMP
4  if((schedule == 's') && parallel)
5  {
6      omp_set_schedule(omp_sched_static,0); // Schedule = static, chunk size = automatic
7      cout << "Schedule set to static" << endl;

```

```
8 }else if((schedule == 'd') && parallel)
9 {
10     omp_set_schedule(omp_sched_dynamic,0); // Schedule = dynamic, chunk size = automatic
11     cout << "Schedule set to dynamic"<< endl;
12 }else if((schedule == 'g') && parallel)
13 {
14     omp_set_schedule(omp_sched_guided,0); // Schedule = guided, chunk size = automatic
15     cout << "Schedule set to guided"<< endl;
16
17 }else if((schedule == 'a') && parallel)
18 {
19     omp_set_schedule(omp_sched_auto,0); // Schedule = auto, chunk size = no meaning
20     cout << "Schedule set to auto"<< endl;
21 }else if(parallel == false){
22     cout << "No parallelization"<< endl;
23 }else{
24     cout << "Unknown OMP schedule, schedule will be determined automatically"<< endl;
25 }
```

Listing A.2: Code für die Bestimmung des OpenMP Schedules

### A.8 Code: OpenMP Threadanzahl

```
1 bool parallel = (method == 2) || (method == 4);
2 ...
3 // setting thread count to calculate with
4 if (num_threads == 0 && parallel)
5 {
6     cout << "Threads to initialize determined automatically"<< endl;
7 } else if(parallel && num_threads < 99)
8 {
9     omp_set_num_threads(num_threads);
10    cout << "Calculation done by " << num_threads << " threads"<< endl;
11 }else if (parallel && num_threads == 99)
12 {
13     int max_threads = omp_get_max_threads();
14     omp_set_num_threads(max_threads);
15     cout << "Calculation done by maximum of " << max_threads << " threads"<< endl;
16 }else {
17     cout << "Program runs on one thread"<< endl;
18 }
```

Listing A.3: Code für die Bestimmung der Threadanzahl

### A.9 Code: Bestimmung der Konstanten

```
1 // determine constants either serial or parallel
2 if (determine_constants == 'p')
3 { cout << "Parallel calculation of constants" << endl;
4     #pragma omp parallel for
5     for(unsigned i=0; i<M; i++) {
6         double tmpsum = 0;
7         double tmpsumplusone = 0;
8         for(unsigned r=0; r<image.nRow; r++) {
9             for(unsigned c=0; c<image.nCol; c++) {
10                double tmp = star[r+i/psf.nCol][c+i%psf.nCol];
11                tmpsum += image[r][c]*tmp;
```

```

12     tmpsumplusone += tmp;
13     }
14     }
15     m[i][M] = tmpsum;
16     m[i][M+1] = tmpsumplusone;
17     }
18 }else if(determine_constants == 's'){
19     cout <<"Serial calculation of constants" << endl;
20     for(unsigned r=0; r<image.nRow; r++) {
21         for(unsigned c=0; c<image.nCol; c++) {
22             for(unsigned i=0; i<M; i++) {
23                 double tmp = star[r+i/psf.nCol][c+i%psf.nCol];
24                 m[i][M] += image[r][c]*tmp;
25                 m[i][M+1] += tmp;
26             }
27         }
28     }
29 }else
30 {
31     cout <<"Error: Unknown determine_constants Value" << endl;
32     cout <<"Constants will be determined unparallelized" << endl;
33     for(unsigned r=0; r<image.nRow; r++) {
34         for(unsigned c=0; c<image.nCol; c++) {
35             for(unsigned i=0; i<M; i++) {
36                 double tmp = star[r+i/psf.nCol][c+i%psf.nCol];
37                 m[i][M] += image[r][c]*tmp;
38                 m[i][M+1] += tmp;
39             }
40         }
41     }
42 }

```

Listing A.4: Code für die Bestimmung der Threadanzahl

## A.10 Compilerkonfiguration

Beispiel für Konfiguration Ofinal in tasks.json:

```

1 {
2     "type": "process",
3     "label": "BA-OMP-Ofinal-64Bit",
4     "command": "C:\\msys64\\mingw64\\bin\\x86_64-w64-mingw32-g++.exe",
5     "args": [
6         "-Ofast",
7         "-fopenmp",
8         "-march=native",
9         "-DNDEBUG",
10    "main.cpp",
11    "AnalyseLMS.cpp",
12    "CreateStarPattern.cpp",
13    "Tools.cpp",
14    "random.cpp",
15    "-o",
16    "${workspaceFolder}\\BA-OMP-Ofinal-64Bit.exe"
17    ],
18    "options": {
19        "cwd": "${workspaceFolder}"
20    },
21    "problemMatcher": [
22        "$gcc"
23    ],
24    "group": {

```

```
25     "kind": "build",
26     "isDefault": true
27   },
28   "detail": "Vom Debugger generierte Aufgabe."
29 },
```

Listing A.5: Konfiguration in tasks.json

### Beispiel für Konfiguration Ofinal in launch.json

```
1 {
2   "name": "BA-OMP-Ofinal-64Bit",
3   "type": "cppdbg",
4   "request": "launch",
5   "preLaunchTask": "BA-OMP-Ofinal-64Bit",
6   "program": "${workspaceFolder}/BA-OMP-Ofinal-64Bit.exe",
7   "args": [],
8   "stopAtEntry": false,
9   "cwd": "${fileDirname}",
10  "environment": [],
11  "externalConsole": false,
12  "MIMode": "gdb",
13  "miDebuggerPath": "C:\\msys64\\mingw64\\bin\\gdb.exe",
14  "setupCommands": [
15    {
16      "description": "Automatische Strukturierung und Einrueckung fuer \"gdb\" aktivieren",
17      "text": "-enable-pretty-printing",
18      "ignoreFailures": true
19    }
20  ],
21 },
```

Listing A.6: Konfiguration in launch.json

## **Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit**

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

\_\_\_\_\_

Ort	Datum	Unterschrift im Original
-----	-------	--------------------------