TECHNISCHE UNIVERSITÄT DRESDEN

ZENTRUM FÜR INFORMATIONSDIENSTE
UND HOCHLEISTUNGSRECHNEN
PROF. DR. WOLFGANG E. NAGEL

Bachelor-Arbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Parallelisierung des Wellenfrontrekonstruktionsalgorithmus auf Multicore-Prozessoren

Jonas Schenke
(Geboren am 17. September 1995 in Burgstädt)

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Zweitgutachter: Prof. Thomas E. Cowan, PhD
Betreuer: Dr. Michael Bussmann, Matthias Werner

Dresden, 5. März 2018

Aufgabenstellung

- Evaluierung und Performance-Analyse des derzeit fast durchgängig seriellen Wellenfrontrekonstruktionsalgorithmus
- Parallelisierung der kritischen Pfade für Vielkernarchitekturen
- Performance-Messungen der parallelen Implementation
- Auswertung sowie Validierung der Ergebnisse

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Bachelor-Arbeit zum Thema:

Parallelisierung des Wellenfrontrekonstruktionsalgorithmus auf Multicore-Prozessoren

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 5. März 2018

Jonas Schenke

Kurzfassung

Ziel dieser Arbeit war es, einen bereits in Python implementierten Wellenfrontrekonstruktionsalgorithmus zu beschleunigen. Dieser berechnet aus zwei Bildern eines zu untersuchenden Objekts pixelweise die Fronten der elektromagnetischen Welle eines Röntgenlasers. Die Bilder werden dabei von zwei in einem festen zueinander Abstand stehenden, hochempfindlichen Röntgen-CCD-Sensoren aufgenommen. Die Kenntniss der Wellenfront ist die Grundlage zur Phasenrekonstruktion bei Röntgenstreubildern aus denen die Struktur von Proben abgeleitet werden kann, die mit Hilfe des Röntgenlasers untersucht werden. Auf Basis von Performance-Analysen der Python-Implementierung wurden Optimierungen und Parallelisierungsmöglichkeiten für die kritischen Programmabschnitte ermittelt, implementiert und evaluiert. Die zu verarbeitenden Bildpaare wurden hierfür auf mehrere Rechenkerngruppen verteilt, auf welchen die Verarbeitung dieser ebenfalls parallelisiert wurde. Zusätzlich zur Parallelisierung wurden verschiedene Methoden der Übersetzung von Python-Code und die Nutzung von bereits optimierten Funktionen betrachtet. Die hier präsentierte Lösung bietet einen Beschleunigungsfaktor zwischen 1,3 und vier gegenüber der Referenzimplementierung ohne die Zuhilfenahme weiterer Rechenkerne. Unter Nutzung von 120 Kernen wurden Beschleunigungsfaktoren von bis zu 133 gegenüber der Referenzimplementierung auf einem einzelnen Rechenkern erreicht. Darüber hinaus lässt sich das Problem mithilfe der neuen Lösung unabhängig von der Anzahl der Bildpaare über mehrere Rechner verteilen.

Abstract

TODO

Inhaltsverzeichnis

Glossar

N_{Paare} Anzahl der Bildpaare.

N_{Pos} Anzahl der Sensorposition.

N_{corr} Anzahl der Korrekturversuche.

N Anzahl der Bilder.

P_u Unterabtastperiode.

R_{Block} Auflösung der Blöcke.

R_{ROI} Auflösung der Region von Interesse.

R_{corr} Korrelationsgröße bzw. Korrelationsauflösung.

R_{grid} Gitterauflösung.

R Auflösung.

Abkürzungen

CCD Charge-Coupled Device.

CPU Central Processing Unit.

FFT Fast Fourier Transformation.

GB Gigabyte.

GHz Gigahertz.

GiB Gibibyte.

GPGPU General Purpose Computing on Graphics Processing Unit.

I/O Input/Output.

JIT just-in-time.

MPI Message Passing Interface.

ROI Region of Interest (Region von Interesse).

SSD Solid-State Drive.

1 Einleitung

Die Programmiersprache Python erfreut sich heutzutage besonders bei Physikern hoher Popularität. Sie ist einfach zu bedienen und bietet Zugriff auf eine große Ansammlung von physikalischer Hilfsbibliotheken, was unter anderem auch die Portierung von MATLAB-Code vereinfacht. Der hier zu parallelisierende Wellenfrontrekonstruktionsalgorithmus wurde auf diese Weise von MATLAB zu Python portiert. Die Aufgabe dieses Algorithmus ist es, aus zwei Bildern eines zu untersuchenden Objektes die Fronten der elektromagnetischen Welle eines Röntgenlasers zu rekonstruieren. Die Bilder werden dabei von zwei in einem festen zueinander Abstand stehenden, hochempfindlichen Röntgen-Charge-Coupled Device (CCD)-Sensoren aufgenommen. Diese Sensoren haben jeweils eine Auflösung von 2048 mal 2048 Pixeln und können bis zu zehn Bilder in einer Sekunde aufnehmen.
Der vorgegebene Python-Code benötigt jedoch pro Bildpaar mehr als 500 Sekunden auf einem Kerne eines Intel®Xeon® E5-2680 v3 Central Processing Units (CPUs). Selbst unter der Nutzung von 24 dieser Rechenkerne unterschreitet die Laufzeit pro Bildpaar die 100-Sekunden Grenze nicht.
Ziel dieser Arbeit ist es, die Laufzeit des Algorithmus näher an eine mögliche Echtzeitausführung des Programmes zu bringen. Dabei sollte die hier präsentierte Lösung eine möglichst gute Skalierung mit einerseits den Bildpaaren und andererseits mit den CPU-Kernen liefern. Auch eine einfach Erweiterbarkeit des Python-Codes ist erwünscht.
Um diese Kriterien erfüllen zu können, wurde daher zuerst der vorliegende Code bezüglich seiner Performance evaluiert. Diese Kenntnisse wurden anschließend für die Parallelisierung und weitere Optimierung genutzt. Hierbei wurden verschiedene Optimierungsmöglichkeiten betrachtet, welche sich vor allem mit der Nutzung bereits optimierter Bibliotheken und dem Übersetzen des Programmes in Maschinencode befassen. Zum Schluss wurden die Optimierungen bezüglich des erreichten Beschleunigungsfaktors gegenüber der Referenzimplementierung evaluiert.
Die hier verwendeten Datensätze wurden an der European Synchrotron Radiation Facility (ESRF) aufgenommen, wo auch die vorgegebene Python-Implementierung entwickelt wurde. Diese Arbeit wurde in der Abteilung für Strahlenphysik am Helmholtz-Zentrum Dresden - Rossendorf e. V. (HZDR) geschrieben.

2 Der Wellenfrontrekonstruktionsalgorithmus

2.1 Verwandte Arbeiten

2.1.1 Wellenfrontrekonstruktionsalgorithmus

In strahlenphysischen Experimenten spielt die Ausrichtung optischer Bauteile und deren Qualität eine entscheidende Rolle. Guizar-Sicairos *et al.* beschrieben 2011 Methoden zur Messung dieser Fehler und zur Rekonstruktion der Wellenfront [Gui+11]. Während der letzten zwei Jahrzehnte wurden hierzu verschiedene Techniken vorgestellt. Mercère *et al.* stellten 2003 ein Verfahren zur Ermittlung der Wellenfront unter Nutzung eines Hartmann-Sensors vor [Mer+03]. 2005 wurde von Weitkamp *et al.* eine Methode zur Wellenfrontrekonstruktion mittels eines Gitterinferometers vorgestellt [Wei+05]. Anand *et al.* präsentierten 2007 eine weitere Methode basierend auf der Verwendung einer Zufalls-Amplituden-Maske [Ana+07]. Diese umgeht das Problem der niedrigen räumlichen Auflösung eines Hartmann-Sensors und die aufwendige Kalibrierung der Optik eines Interferometers, indem mehrere Bilder in unterschiedlicher Entfernung vom zu untersuchenden Objekt gemacht werden. Dies bietet eine höhere Auflösung und die Möglichkeit, Ungenauigkeiten in der Optik algorithmisch zu korrigieren. Ein ähnlicher, robusterer Ansatz wurde 2012 von Bérujon *et al.* unter Zuhilfenahme des X-Ray-Speckle-Traking-Algorithmus verfolgt [Bér+12] und 2013 wurde von Bérujon eine Methode mit zwei Sensoren vorgestellt, welche simultan Bilder des zu untersuchenden Objektes aufnehmen können und somit schneller sind [Bér13]. Auf dieser Methode basiert der hier behandelte Wellenfrontrekonstruktionsalgorithmus.

2.1.2 Python-Optimierungen

Es wurde von Oliphant und Perez *et al.* aufgezeigt, dass die Programmiersprache Python aufgrund ihrer Simplizität und der Verfügbarkeit vieler Bibliotheken sich gut für wissenschaftliche Berechnungen eignet [Oli07; PGH11]. Da Python als Skriptsprache nicht besonders schnell ist, wurde sich bereits ausgiebig mit dem Thema Beschleunigung von Python-Code beschäftigt. Die Lösungsansätze reichen von der Verwendung optimierter Bibliotheken über das Kompilieren kompletter Programme bis hin zur Vektorisierung und Parallelisierung des Codes. 2009 wurde von Ben Asher *et al.* sogar eine auf Loop-Unrolling und Bytecode-Optimierung basierende Methode vorgestellt [BR09]. Illmer hat 2014 einen Überblick über verschiedene Optimierungen gegeben und deren Effektivität anhand des freewake-Algorithmus aufgezeigt [Ill14].

Kompilieren Behnel *et al.* zeigten, dass mithilfe des Cython-Compilers und dessen Erweiterungen für Python unter bestimmten Szenarien die 1000-fache Geschwindigkeit erreicht werden kann. Dies ist möglich, indem der für Cython optimierten Python Code nach C/C++ übersetzt und kompiliert wird [Beh+11]. Ein weiterer, nennenswerter Python-zu-C++-Compiler ist der Shed Skin-Compiler [Duf]. Lam *et al.* präsentierten 2015 numba, einen Python Just-In-Time-Compiler [LPS15]. Dieser unterstützt unter anderem auch Annotationen, mit denen sich einzelne Python-Funktionen unabhängig vom Rest des Programmes just-in-time in nativen Code übersetzen lassen.

Vektorisierung Wie man mittels Pythran und Boost.SIMD Python vektorisieren kann, wurde 2014 von Guelton *et al.* gezeigt [GFB14]. Eine weitere Möglichkeit bietet die numpy-Bibliothek, welche eine einfache Anwendung von Rechenoperationen auf ganze Arrays zulässt. Die wichtigsten Funktionen der numpy-Bibliothek wurden von Walt *et al.* dargelegt. Daily stellte 2009 eine Bibliothek vor, die ähnliche Verfahren über mehrere Rechner verteilen kann [Dai09]. Guelton *et al.* stellten 2013 einen Ansatz vor,

mit dem sich auch OpenMP in Python nutzen lässt [GBA13]. Eine MPI-1-Implementierung existiert ebenfalls und wurde 2005 von Dalcín *et al.* vorgestellt und 2008 von Dalcín *et al.* auf MPI-2 erweitert [DPS05; Dal+08]. Die Verteilung der Daten auf verschiedene Rechenknoten kann erheblichen Einfluss auf die Leistung des Gesamtprogrammes haben. Um Python-Objekte zwischen Rechner auszutauschen, müssen diese serialisiert werden, was mittels dem Python-internen Pickle/cPickle Modul erreicht werden kann. Hierzu verglichen Dalcin *et al.* die Pickle/cPickle Methoden mit einer in C implementierten Buffer-Variante [Dal+11]. Pickle/cPickle war dabei bis zu 30% langsamer als die in C implementierten Buffer-Variante.

Anwendungen Aufgrund der mannigfaltigen Möglichkeiten der Optimierung und der einfachen Nutzung, hält Python verstärkt Einzug auf Hochleistungsrechner. Klemm *et al.* präsentierten hierzu 2014, wie Python-Code effizient auf der Intel®Xeon Phi™Coprozessoren ausgeführt werden kann [KE14]. Wie Python für Hochenergiephysik verwendet werden kann, zeigten Sehrish *et al.* [Seh+17]. Hierbei wurden die Bibliotheken numpy, HDF5 und mpi4py intensiv genutzt. Das von Hand *et al.* vorgestellte nbodykit ist eine Ansammlung von Funktionen für Kosmologie-Simulationen, welche ebenfalls in Python entwickelt wurden und für den Gebrauch auf Hochleistungsrechnern optimiert wurde [HF17]. Wie 2011 von Enkovaara *et al.* gezeigt wurde, kann Python auch auf Hochleistungsrechnern für die Simulation elektronischer Strukturen verwendet werden [Enk+11]. Das Berechnen direkt numerischer Simulationen von Turbulenzverläufen kann laut Mortensen *et al.* ebenfalls performant mit Python durchgeführt werden [ML16].

2.2 Einführung

2.2.1 Versuchsaufbau

Ein allgemeiner Versuchsaufbau in der Strahlenphysik besteht aus einer Röntgenstrahlenquelle, einem zu untersuchenden Objekt, welches im Fokus der Röntgenstrahlen platziert ist und einem Aufnahmeaufbau, wie es von Bérujon beschrieben wurde ([Bér13], S. 31 ff.). Mittels der Belichtung des zu untersuchenden Objekts können anschließend physikalische Informationen über dieses herausgefunden werden.

Bérujon zeigte ebenfalls, dass die Bestimmung der geometrischen Struktur eines zu untersuchenden Objekts mittels eines solchen Versuchsaufbaus realisiert werden kann ([Bér13], S. 105 ff.). Der Aufnahmeaufbau, laut Bérujon *et al.*, besteht in diesem konkreten Fall aus einer Fleckenmembran und zwei hochempfindlichen Röntgen-CCD-Sensoren, die zwei Bilder des zu untersuchenden Objekts zeitgleich aus unterschiedlicher Entfernung durch die Fleckenmebran aufnehmen ([BZC15], S. 887). Ein solcher Versuchsaufbau wird in der Abbildung 2.1 dargestellt, welche aus der Veröffentlichung [BZC15] von Bérujon *et al.* entnommen wurde [BZC15].

Die Röntgenstrahlen werden zuerst am zu untersuchenden Objekt absorbiert und gebrochen und treffen anschließend auf die Membran, wodurch das Röntgenstrahlbündel hinter dieser Membran ein Fleckenmuster aufweist. Dieses Bündel trifft zuerst auf den Szintillator des ersten CCD-Sensors, wodurch die Röntgenstrahlen für die CCD-Sensoren sichtbar gemacht werden. Der sichtbare Teil dieses Bildes wird zum erstem CCD-Sensor reflektiert und dort aufgenommen. Der Röntgen-Teil wird zum Szintillator des zweiten CCD-Sensors gebrochen, wo dieser in sichtbares Licht gewandelt wird, das mittels einer Optik zum zweiten CCD-Sensor reflektiert und dort aufgenommen wird. ([BZC15], S. 886 ff.)

Aus der Verschiebung der Flecken lässt sich die Phase, und somit die Wellenfront des Lichtes rekonstruieren, wozu der Wellenfrontrekonstruktionsalgorithmus verwendet wird. Gemäß Bérujon besteht dieser aus zwei Teilen, welche im Folgenden genauer beschrieben werden sollen: einer Kalibrierung der CCD-Sensoren und der Hauptverarbeitungsroutine der Bilder ([Bér13], S. 194). Der Algorithmus soll online (zeitgleich zum Experiment) ausgeführt werden. Die CCD-Sensoren haben eine Auflösung von 2048x2048 Pixeln mit einer Farbtiefe von 16 Bit (Graustufen) und liefern bis zu 10 Bilder pro Sekunde.

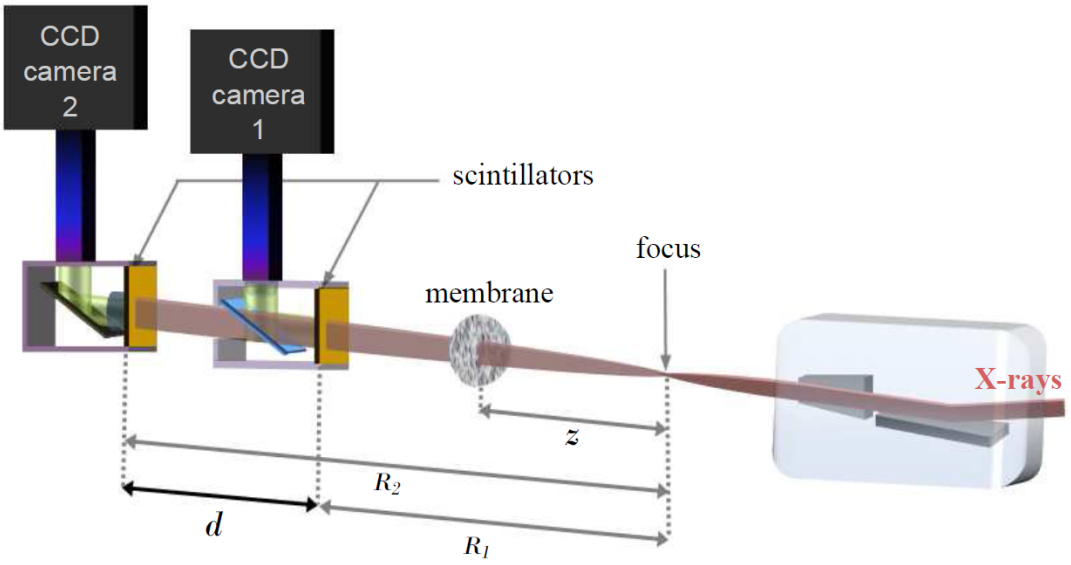


Abbildung 2.1: Versuchsaufbau ([BZC15], S. 891)

2.2.2 Kalibrierung der CCD-Sensoren

Während der Kalibrierung findet, gemäß Bérujon, zuerst eine Parameterinitialisierung statt, in welcher der Kamerafehler der CCD-Sensoren in drei Phasen bestimmt wird ([Bér13], S. 76 ff., S. 194). Dazu wird, wie in Abbildung 2.2 gezeigt, zuerst der Median aus einer festen Anzahl dunkler Bilder aufgenommen, was auch als Dunkelfeldkalibrierung bezeichnet wird. Anschließend findet eine Nullfeldkalibrierung statt, in welcher bei ungestörter Wellenfront der Verstärkungsfaktor der einzelnen Pixel ermittelt wird. Zum Schluss findet eine Erkennung von Streueffekten im Detektor statt. Um letzteres zu erreichen, werden für jeden CCD-Sensor ${N_{Pos}}^2$ Bilder unter Bewegung der CCD-Sensoren aufgenommen, wobei N_{Pos} hier gleich der Anzahl der Verschiebungen in X- und Y-Richtung ist. Dadurch lassen sich aus den vertikal und horizontal Gradienten benachbarter Motorpositionen die jeweilige Streuung der Sensoren mittels des Speckle-Trackings bestimmen. Aus diesen N_{Pos} Positionen ergibt sich somit, dass die Anzahl der Bilder N gleich $N_{Pos}*(N_{Pos}-1)*2*2$ ist, da jeweils horizontal und vertikal $N_{Pos}*(N_{Pos}-1)$ Bildpaarkombinationen existieren, welche in beide Richtungen miteinander verrechnet werden müssen. Da die Kalibrierung lediglich am Anfang des Experiments einmal durchgeführt werden muss, wird diese im weiteren Verlauf der Arbeit vernachlässigt.

Speckle-Tracking Ziel des Speckle-Trackings ist es, die Flecken (englisch: speckle) zwischen den CCD-Sensoren zu verfolgen, welche von der Fleckenmembran geworfen werden. Der von Bérujon beschriebene Speckle-Tracking-Algorithmus (dargestellt in Abbildung 2.4a) ermittelt zuerst die starre Verschiebung der beiden Sensoren zueinander ([Bér13], S. 76 ff., S. 194). Dies kann mittels festgelegter Werte, Korrelation oder Kreuzkorrelation geschehen. Anschließend werden die Bilder der Sensoren in 35x35 Pixel bzw. 10x10 Pixel große Teilbilder aufgeteilt, damit ein grober Gradient in einem ersten Durchlauf (gezeigt in Abbildung 2.4b) bestimmt werden kann. Diese Teilbilder liegen hierbei alle in der Region of Interest (Region von Interesse) (ROI), welche die möglicherweise fehlerbehafteten Bildränder abschneidet. Die Verschiebung der Teilbilder zwischen den beiden Bildern der Sensoren wird mithilfe des von Lewis beschriebenen Template-Matching-Prozesses ermittelt [Lew94]. Dieser sucht die Teilbilder des ersten Sensors mit denen des zweiten Sensors, wobei eine Übereinstimmungsmatrix entsteht. Anhand der Positionen der Maxima kann nun die Verschiebung abgelesen werden. Dieser Prozess kann laut Lewis durch die Kreuzkorrelation der beiden Teilbilder im Frequenzraum realisiert werden um die Komplexität des Algorithmus zu senken [Lew94]. Damit endet der erste Durchlauf. Im weite-

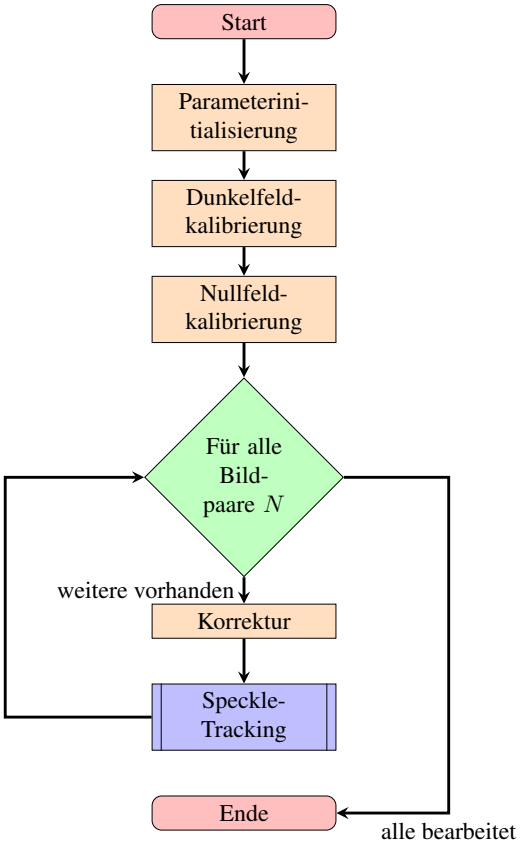


Abbildung 2.2: Programmablaufplan der Kalibrierung (nach ([Bér13], S. 85 ff., S. 194) und [CS17]; angefertigt gemäß DIN 66001 bzw. ISO 5807)

ren Verlauf werden zuerst stark abweichende Werte herausgefiltert und anschließend werden Ebenen an die horizontalen und vertikalen Verschiebungsmartizen angelegt. Diese sind zur Interpolation nötig, da die Verschiebungsmatrizen lediglich einen Pixel pro Teilbild beinhalten. Sobald die Ebene auf die volle Auflösung des Bildes interpoliert wurde, wird der zweite Durchlauf (gezeigt in Abbildung 2.4c) durchgeführt. Im diesem Durchlauf werden wieder beide Bilder in Teilbilder unterteilt, diesmal können sich die Teilbilder jedoch überlappen. Die konkrete Anzahl der Teilbilder hängt, gemäß Cojocaru *et al.*, von drei Variablen ab [CS17]: der Unterabtastperiode P_u, der Korrelationsgröße R_{corr} und der Gitterauflösung R_{grid}. Mittels der Unterabtastperiode lassen sich, wie in Abbildung 2.3 veranschaulicht, jeweils P_u Pixel überspringen, wodurch die effektive Auflösung der Region von Interesse R_{ROI} des Bildes auf Auflösung $\lfloor R_{ROI}/{P_u}^2 \rfloor$ reduziert wird. Die Korrelationsgröße R_{corr} gibt die Größe der Teilbilder an. Die Gitterauflösung bestimmt die Größe des Resultats, indem jeweils ein Teilbild um jeden R_{grid}-ten Punkt im bereits durch Unterabtastung verkleinerten Bild genutzt wird. Die Verschiebung der so berechneten Teilbilder wird dann mittels des Template-Matching-Prozesses bestimmt und auf Subpixel-Genauigkeit interpoliert. Teilbilder, die im zweiten Durchlauf nicht korrekt zugeordnet werden konnten, werden gesammelt und unter Verwendung von anderen Korrelationsgrößen erneut verarbeitet. Sollten diese Korrelation ebenfalls fehlschlagen, werden die Ergebnisse der fehlerhaften Teilbilder interpoliert.

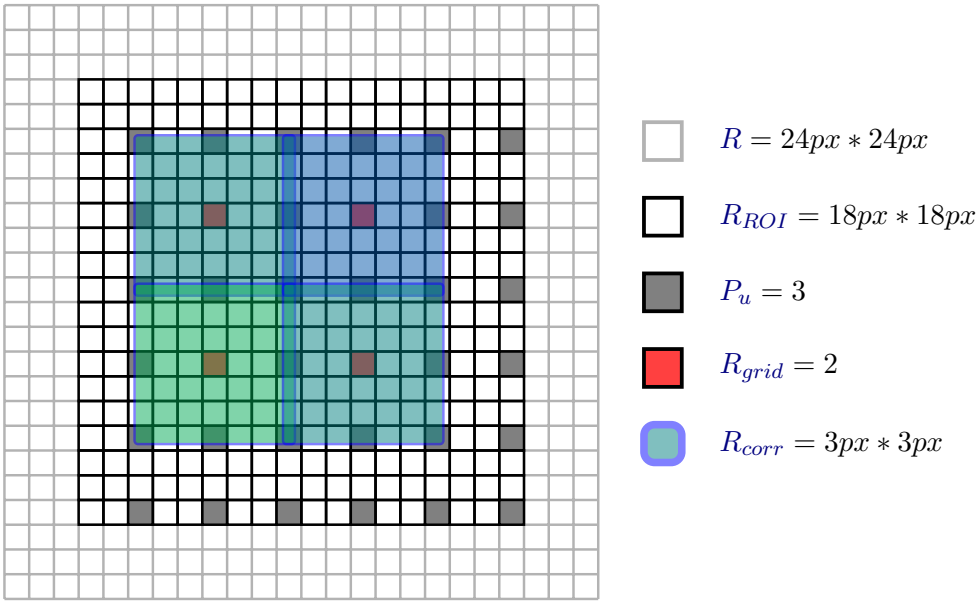
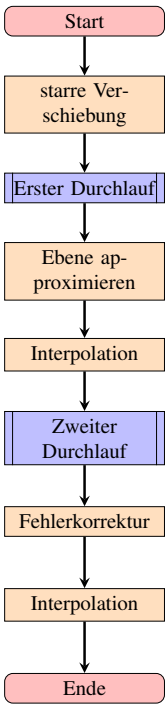


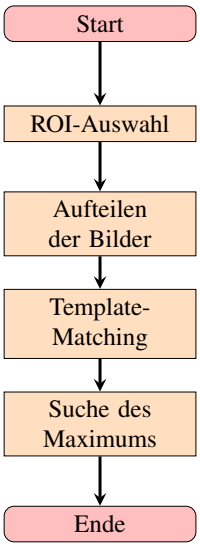
Abbildung 2.3: Übersicht der Template-Matching-Parameter im zweiten Durchlauf

2.2.3 Verarbeitungsroutine der Bilder

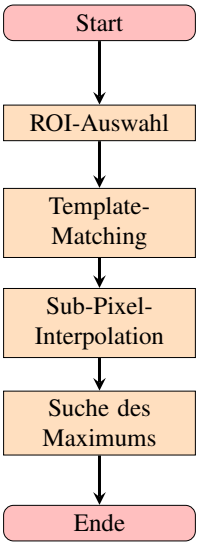
Die von Bérujon *et al.* beschriebene Hauptverarbeitungsroutine (siehe Abbildung 2.8a) ähnelt stark der Kalibrierung: Auch hier findet zuerst eine Parameterinitialisierung statt, welche von der Hauptschleife gefolgt wird ([BZC15], S. 891). Diese verarbeitet hier, im Gegensatz zur Kalibrierung, nur jedes Bildpaar. Ein Bildpaar ist exemplarisch in Abbildung 2.5 gezeigt. Innerhalb der Hauptschleife gibt es auch hier die Korrektur der Sensorfehler und das anschließende Speckle-Tracking, dessen Ergebnisse in Abbildung 2.6 gezeigt sind. Die Korrektur bezieht hierbei die von der Kalibrierung berechneten Ergebnisse, insbesondere die ermittelten Streueffekte der Sensoren. Anders als bei der Kalibrierung, werden hierbei jedoch in der Hauptschleife die beiden Gradientenmatrizen mittels des von Frankot *et al.* entwickelten Algorithmus zu einer Phasenmatrix integriert (gezeigt in Abbildung 2.8b), was effizient im Frequenz-



(a) Speckle-Tracking-Algorithmus (nach ([Bér13], S. 85 ff., S. 194) und [CS17])



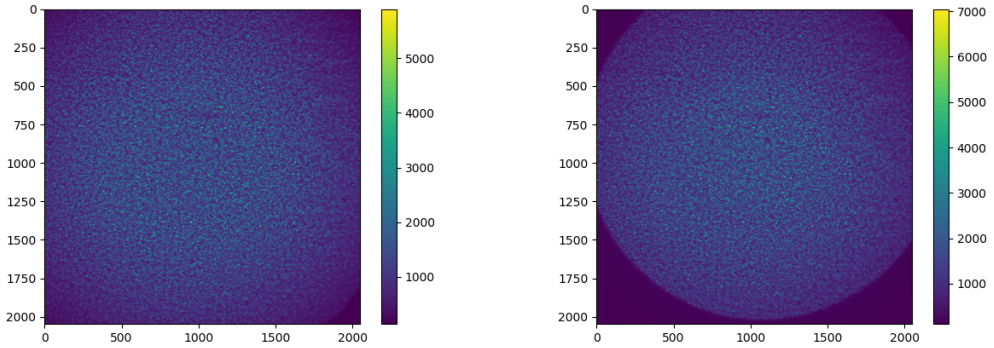
(b) Erster Durchlauf (nach [Bér+12] und [CS17])



(c) Zweiter Durchlauf (nach ([Bér13], S. 194), [Bér+12] und [Kov04])

Abbildung 2.4: Programmablaufpläne der Speckle-Tracking-Subroutinen (angefertigt gemäß DIN 66001 bzw. ISO 5807)

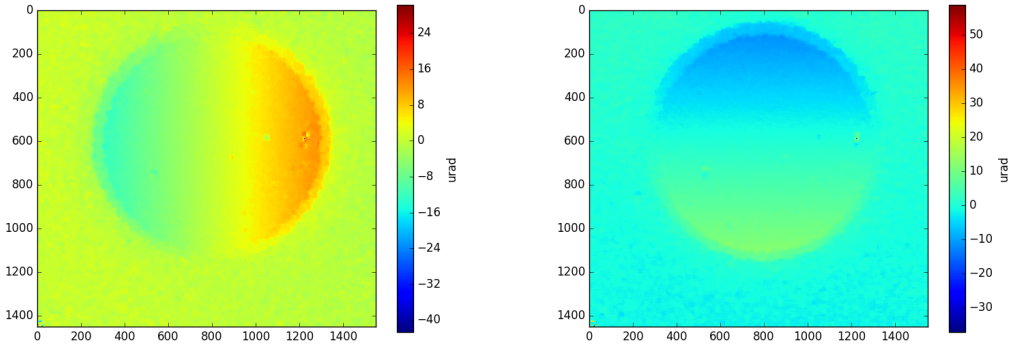
raum möglich ist [FC88]. Die zu in Abbildung 2.5 gezeigten Eingabebildern zugehörige Phasenmatrix ist in Abbildung 2.7 dargestellt.



(a) Eingabebild des ersten Sensors

(b) Eingabebild des zweiten Sensors

Abbildung 2.5: Eingabebilder



(a) Horizontaler Gradient

(b) Vertikaler Gradient

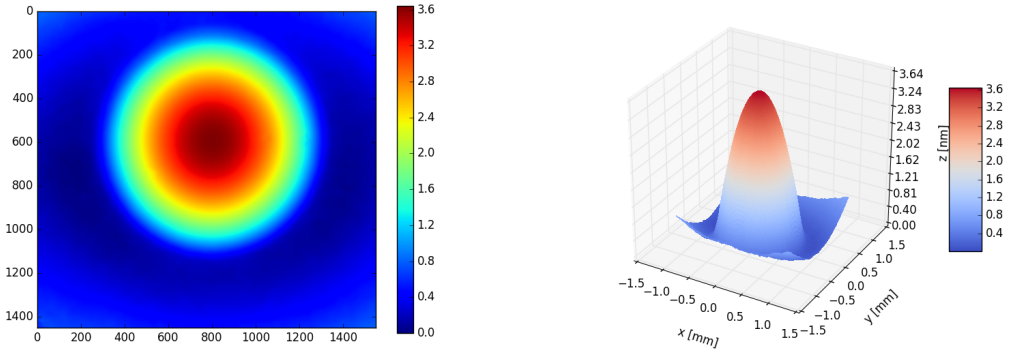
Abbildung 2.6: Gradientenbilder (Ausgabe des Speckle-Tracking)

2.3 Die wichtigsten Funktionen

Der bestehende Python-Code wurde von Elena-Ruxandra Cojocaru entwickelt und ist auf dem GitHub-Repository *Wavefront-Sensor* verfügbar [CS17]. Aufgrund der Abhängigkeit der Bibliothekt *EdfFile* zum Auslesen der Bilddaten, wurde der Code in Python 2.7 geschrieben. Der Code ist in vier Dateien aufgeteilt. Die Datei *norm_xcorr.py* nutzt die Schnittstelle zu der in OpenCV implementierten Template-Matching-Funktion [SA17]. Alle Helferfunktionen, insbesondere die des Speckle-Trackings und die der Integration der Gradienten, befinden sich in der *func.py*. Der Code der Kalibrierung befindet sich in der *detectorDistortion.py* Datei und der der Hauptroutine befindet sich in der Datei *wavefront.py*.

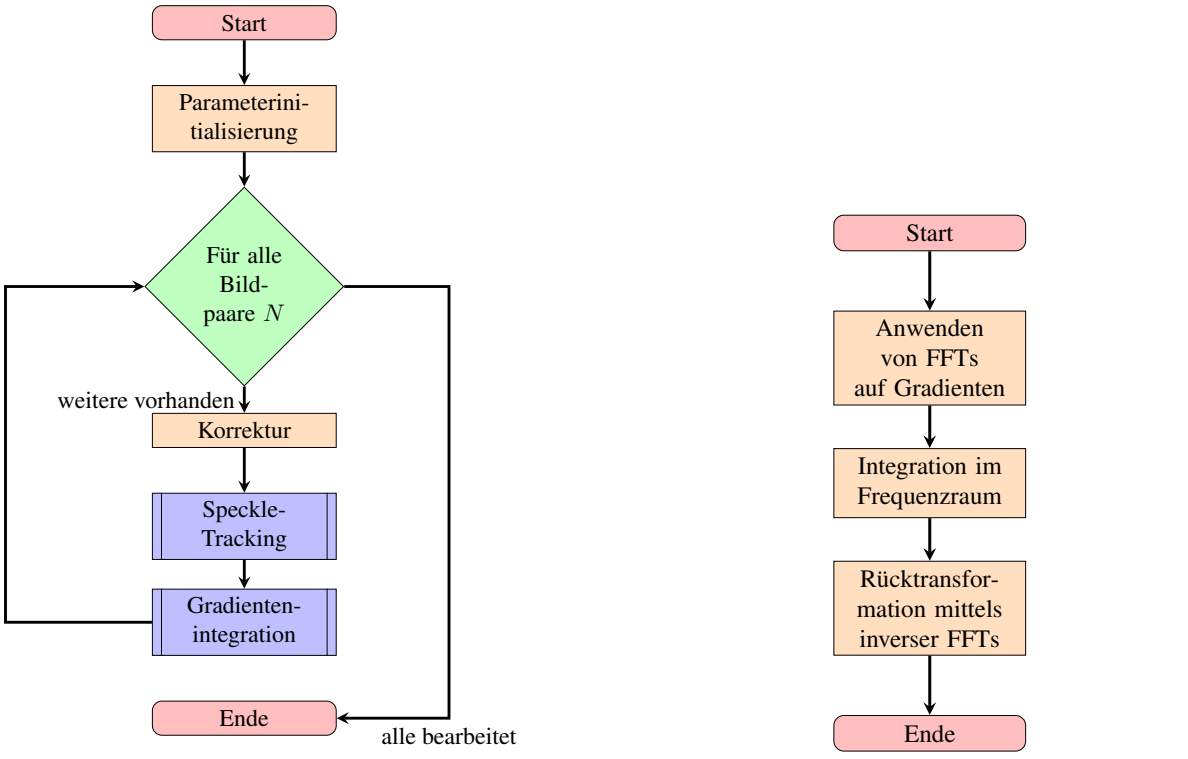
2.3.1 Überblick über die vorgegebene Python-Implementierung

Die Datei *norm_xcorr.py* enthält alle Funktionen, die für das Template-Matching benötigt werden. Darunter ist die Funktion `norm_xcorr()`, welche lediglich als Schnittstelle für die Template-Matchitg Funktion aus OpenCV dient. Sie nimmt als Argumente ein Template, ein Suchfeld und optional den Namen des gewünschten Algorithmus entgegen und gibt die Übereinstimmungsmarix, die Position und den



(a) 2D Repräsentation (b) 3D Repräsentation

Abbildung 2.7: Ausgabe des Algorithmus (Phasenmatrix)



(a) Hauptroutine (nach ([Bér13], S. 194) und [CS17]) (b) Integration der Gradienten gemäß [FC88] und [Kov04]

Abbildung 2.8: Programmablaufpläne der Hauptroutine (angefertigt gemäß DIN 66001 bzw. ISO 5807)

Wert des Maximums und des Minimums zurück. Die zweite Funktion in dieser Datei ist die `nxcorr_disp()`-Funktion, welche die Übereinstimmungsmatrix an relevanten Stellen, wo die Übereinstimmung am höchsten ist, auf ein Subpixel-Level verfeinert. Dazu nimmt diese Funktion das Ergebnis der `norm_xcorr()`-Funktion entgegen und gibt die Position sowie den Wert des neuen Maximums und das Signal-Rausch-Verhältnis zurück.
Die *func.py* enthält die meisten Helferfunktionen. Dazu gehören unter anderem Funktionen zur ROI-Auswahl und verschiedene Filterfunktionen. Des Weiteren befinden sich in dieser Datei auch die Funktionen `computerCorrField()` und `correction()`. `computerCorrField()` berechnet aus Bildern an einem vorgegebenen Pfad entweder das mediangefilterte Bild für die Dunkelfeld-Kalibrierung oder das durchschnittsgefilterte Bild für Nullfeld-Kalibrierung. Mithilfe der `correction()` Funktion können nun diese berechneten Korrekturbilder auf Eingabebilder mittels Subtraktion für die Dunkelfeld-Kalibrierung oder mittels Division für die Nullfeld-Kalibrierung angewendet werden. Zwei weitere wichtige Funktionen sind `firstPass()` und `cpCorr()`, welche den ersten (`firstPass()`) und zweiten Durchlauf (`cpCorr()`) implementieren. Diese sind damit Teil des Speckle-Trackings-Algorithmus, welcher in der `crossSpot4()`-Funktion implementiert ist. Die letzte wichtige Funktion im `func.py`-Skript ist die `frankot_chellappa()`-Funktion, welche aus Gradientenfeldern eine dreidimensionale Rekonstruktion berechnet. Dies wird nach dem Speckle-Tracking verwendet. Um dies effizient und schnell zu erreichen, werden die Eingabematrizen mittels Fast Fourier Transformations (FFTs) in den Frequenzraum transformiert, dort nach Formel 21 des von Frankot *et al.* beschriebenen Algorithmus addiert und wieder zurück transformiert ([FC88], S. 443).
Das letzte Skript, *wavefront.py*, beinhaltet den eigentlichen Rekonstruktionsalgorithmus, welcher sich besonders in der Initialisierungsphase sehr stark dem *detectorDistortion.py*-Skript ähnelt, welches die Kalibrierung derCCD-Sensoren implementiert. Die Hauptroutine wird jetzt, wie in Abbildung 2.8a veranschaulicht, nur einmal für jedes Bildpaar statt für jede Kombination aus Eingabebildern ausgeführt. Innerhalb dieser Routine wird das entsprechende Bildpaar samt Stellung Positionierungsmotoren der CCD-Sensoren geladen und mittels der `correction()`-Funktion korrigiert. Da die Pixelgröße zwischen den Sensoren unterschiedlich sein kann, wird hierfür eine Interpolation durchgeführt, die dieses Problem beheben soll. Bevor das Speckle-Tracking mittels der `crossSpot4()`-Funktion ausgeführt wird, können optional noch ein Gauß-Filter, ein Mittelwertfilter und ein Erosionsfilter angewendet werden. Danach werden die Zwischenergebnisse gespeichert und die endgültige Wellenfront wird mittels der `frankot_chellappa()`-Funktion rekonstruiert und gespeichert.

2.3.2 Die wichtigsten Funktionen

Die `norm_xcorr()`-Funktion ist die direkte Schnittstelle für die in OpenCV implementierte `matchTemplate()`-Funktion. Hierbei wird eine Matrix aus Übereinstimmungen mittels Faltung erstellt, die an jedem Pixel mit einem Wert zwischen -1 und 1 angibt, wie ähnlich sich die Bilder sind. Diese Funktion ist Teil des Speckle-Tracking-Algorithmus, der dafür zuständig ist, die Verschiebung der Wellenfront zwischen den beiden CCD-Sensoren zu ermitteln. Standardmäßig wird hierbei der Kreuzkorrelationskoeffizient verwendet.
Aufgabe dieser Funktion ist es, den Punkt der maximalen Übereinstimmung auf ein Subpixel-Level zu verfeinern. Dazu wird der Gradient der Pixelwerte in der Nachbarschaft des Maximums gebildet und anschließend interpoliert. Im Gesamtalgorithmus wird dieser Schritt direkt nach dem Template-Matching als Teil des Speckle-Trackings ausgeführt.
Die `frankot_chellappa()`-Funktion nutzt den von Frankot *et al.* vorgeschlagenen Algorithmus um aus vorgegebenen Gradientenfeldern ein dreidimensionales Bild zu rekonstruieren [FC88]. Der hier bereits vorliegende Code basiert auf dem Matlab-Code von Kovesi [Kov04]. Die `frankot_chellappa()`-Funktion wird nach dem Speckle-Tracking aufgerufen und nutzt die durch das Tracking ermittelten Gradientenmatrizen um die Wellenfront dreidimensional zu rekonstruieren.

3 Performance-Analyse der vorgegebenen Python-Implementierung

3.1 Komplexitätsanalyse

3.1.1 Speckle-Tracking

Der erste Schritt des Speckle-Trackings ist die Feststellung der starren Verschiebung. Werden hierfür feste Werte für diese angenommen, ist diese Komplexität konstant. Wird allerdings ein Korrelationsverfahren verwendet, so ist die Komplexität $\mathcal{O}\left(R \cdot log\left(R\right)\right)$ für die Auflösung R, da hierfür FFTs eingesetzt werden.
Der nächste Verarbeitungsschritt ist der erste Durchlauf. Hier werden die Eingabebilder zunächst durch die Selektion der ROI auf die Auflösung der Region von Interesse R_{ROI} zugeschnitten und anschließend in quadratische Blöcke mit konstanter Auflösung der Blöcke R_{Block} aufgeteilt. Daraufhin werden die Blöcke aus den Bildern des ersten Sensors in den Gegenstücken des zweiten Sensors mittels des Template-Matchings gesucht. Liegt ein Block teilweise außerhalb der ROI, so wird dieser hinein korrigiert. Der Durchlauf hat deshalb Komplexität:

$$\mathcal{O}\left(\frac{R_{ROI} \cdot R_{Block} \cdot log(R_{Block})}{R_{Block}}\right) = \mathcal{O}\left(R_{ROI} \cdot log\left(R_{Block}\right)\right)$$

Da die R_{Block} konstant und die Dauer der einzelnen Suchvorgänge somit ebenfalls als konstant angenommen werden können, liegt dieser Durchlauf in der linearen Komplexitätsklasse. Die anschließende Interpolation wird auf alle Pixel des Bildes angewandt und hat demzufolge eine Komplexität von $\mathcal{O}\left(R_{ROI}\right)$.
Im darauf folgenden zweiten Durchlauf werden Subbilder, wie in Abschnitt 2.2.2 beschrieben, generiert. Durch die Unterabtastperiode P_u wird R_{ROI}, und damit auch die Komplexität dieses Teils, mit dem Faktor ${P_u}^2$ verringert. Die Korrelationsgröße R_{corr} nimmt auf das Template-Matching Einfluss, dessen Komplexität dadurch bei $\mathcal{O}\left(R_{corr} \cdot log\left(R_{corr}\right)\right)$ liegt. Der Einfluss der Gitterauflösung R_{grid} ist, ähnlich zu P_u, eine Verringerung mit dem Faktor ${R_{grid}}^2$. Die Gesamtkomplexität der Template-Matchings im zweiten Durchlauf liegt somit bei:

$$\mathcal{O}\left(\frac{R_{ROI} \cdot R_{corr} \cdot log(R_{corr})}{\left({P_u}^2 \cdot R_{grid}\right)^2}\right)$$

Für die auf den Template-Matching-Prozess folgende Subpixel-Interpolierung werden neun Pixel in der Umgebung des Maximums jeder Übereinstimmungsmatrix interpoliert, womit dieser Schritt folgende Komplexität aufweist:

$$\mathcal{O}\left(\frac{R_{ROI}}{\left({P_u}^2 \cdot {R_{grid}}^2\right)}\right)$$

Am Ende des Speckle-Tracking-Algorithmus wird versucht, nicht zuordenbare Ergebnisse mit einer anderen Korrelationsgröße ${R_{corr}}'$ erneut zuzuordnen. Im schlimmsten Fall wird der zweite Durchlauf für die Hälfte der Subbilder N_{corr}-fach wiederholt, wobei N_{corr} die Anzahl der Korrekturversuche N_{corr} repräsentiert. Bei höheren Fehlerraten über 50% bricht das Programm ab. In der hier als Grundlage vorliegenden Implementierung ist N_{corr} gleich 6. Die Gesamtkomplexität des Speckle-Tracking-Algorithmus liegt damit in der Komplexitätsklasse:

$$\mathcal{O}\left(\frac{R_{ROI} \cdot R_{corr} \cdot log(R_{corr}) \cdot N_{corr}}{\left({P_u}^2 \cdot R_{grid}\right)^2}\right)$$

3.1.2 Integration der Gradienten

Um eine effiziente Integration der Gradienten zu ermöglichen, beruht der von Frankot *et al.* vorgeschlagene Algorithmus auf der Integration im Frequenzraum [FC88]. Hierzu werden zuerst die Gradientenbilder mittels FFTs in diesen Raum transformiert, dort in linearer Komplexität integriert und zum Schluss wieder zurück transformiert. Aufgrund der Verwendung von FFTs befindet sich dieser Algorithmus in der Komplexitätsklasse:

$$\mathcal{O}\left(R \cdot log\left(R\right)\right)$$

3.1.3 Verarbeitungsroutine der Bilder

Die Hauptroutine beginnt mit einer trivialen Parameterinitialisierung, die als linear angenommen werden kann. Auf diese folgt die Hauptschleife, welche für die Anzahl der Bildpaare N_{Paare} jeweils einmal ausgeführt wird. Hierzu werden zuerst die Sensorbilder mittels der bei der Kalibrierung ermittelten Werte mit einer Komplexität von $\mathcal{O}\left(R\right)$ korrigiert. Dies wird gefolgt vom Speckle-Tracking-Algorithmus und der Integration der Gradienten. Die höchste Komplexität hat hier der Template-Matching-Algorithmus, welcher damit die Komplexitätsklasse der Hauptroutine festlegt auf:

$$\mathcal{O}\left(\frac{N_{Paare} \cdot R_{ROI} \cdot R_{corr} \cdot log(R_{corr}) \cdot N_{corr}}{\left({P_u}^2 \cdot R_{grid}\right)^2}\right)$$

Diese Komplexitätsklasse liegt insbesondere für kleine N_{corr}, P_u und R_{grid} in der Oberklasse:

$$\mathcal{O}\left(N_{Paare} \cdot R_{ROI} \cdot R_{corr} \cdot log\left(R_{corr}\right)\right)$$

3.2 Performance-Messungen

Testsystem Alle Benchmarks liefen auf der *haswell*-Partition des Taurus-Supercomputers an der Technischen Universität Dresden. Jeder Knoten dieser Partition ist ausgestattet mit zwei Intel®Xeon® E5-2680 v3 CPUs. Diese haben jeweils zwölf Rechenkerne, die mit bis zu 2.50 Gigahertz (GHz) getaktet sind. HyperThreading war hierbei nicht aktiviert. Die Knoten haben 64 Gibibyte (GiB) (*haswell64*), 128 GiB (*haswell128*) oder 256 GiB (*haswell256*) Arbeitsspeicher zur Verfügung [Mar17]. Zusätzlich ist pro Rechenknoten eine 128 Gigabyte (GB) Solid-State Drive (SSD) installiert. Es wurde unter anderem Python 2.7.11 mit numpy 1.10.1 und OpenCV 3.1.0 verwendet. Eine komplette Liste aller geladenen Module lässt sich auf dem GitHub-Repository dieses Projektes finden [Sch18b].

Konfigurationen Jede Konfiguration, bestehend aus Datensatz und Kernanzahl, wurde nach vier Aufwärmiterationen fünfmal ausgeführt. Hierbei wurden jeweils die reinen Ausführungszeiten des gesamten Skripts und einzelner Funktionen erfasst. Aus allen vorliegenden Zeiten wurde Input/Output (I/O)-Zeiten herausgerechnet. Die Laufzeit mit den entsprechenden Datensätzen wurde auf unterschiedlich vielen Kernen von eins bis 24 gemessen. Jeder Benchmark lief exklusiv auf einem Knoten.

Zur Leistungsfeststellung der vorliegenden Implementierung werden zwei verschiedene Arten von Datensätzen verwendet: *Experiment 6* und *Lenses*. Die Bildpaare des *Experiment 6*-Datensatzes wurden, wie in Abbildung 2.1 gezeigt, mit zwei Sensoren gleichzeitig aufgezeichnet, währenddessen die Bildpaare des *Lenses*-Datensatzes aus Bildern ein und desselben Sensors stammen, wobei anfangs ein Bild als Referenz aufgenommen wurde. Die Eigenschaften dieser Typen werden in Tabelle 3.1 gegenüber gestellt. Die einzelnen Datensätze mit deren Anzahl der Bilder ist in Tabelle 3.2 zu finden.

	Experiment 6	Lenses
R_{ROI}	Sensor 1: $550px * 550px$ Sensor 2: $1450px * 1450px$	$1450px * 1550px$
R_{grid}	1	1
R_{corr}	$91px * 91px$	$41px * 41px$
P_u	1	1
Pixelgröße	unterschiedlich	gleich

Tabelle 3.1: Parameter der Datensatztypen

	Experiment 6			Lenses			
	Lenses 200	Lenses 500	Lenses 1500	Set 1	Set2	Set 3	
N_{Paare}	21	11	14	10	5	1	2

Tabelle 3.2: Anzahl der Bildpaare der Datensätze

3.2.1 Laufzeiten

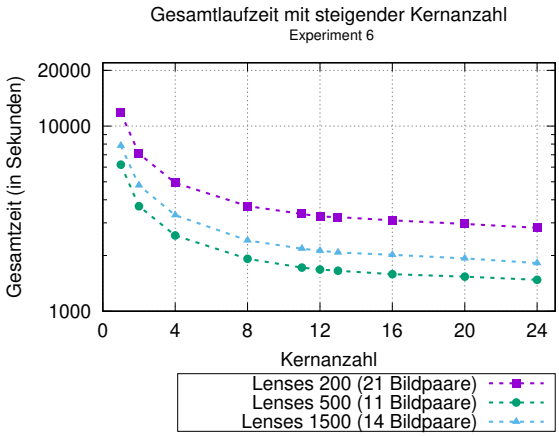
Die Laufzeiten der Konfigurationen, dargestellt in Abbildung 3.1, variieren untereinander stark und reichen von ca. dreieinhalb Stunden für den *Lenses Set 1*-Datensatz auf einem Kern bis hin zu ca. vier Minuten für den *Lenses Set 3* Datensatz mit einem Bild auf 24 Kernen. Die Messpunkte sind hierbei dick hervorgehoben und die Skala ist logarithmisch eingeteilt.

Der Speedup des Programmes skaliert mit der Anzahl der Prozessorkerne nicht linear und flacht schnell ab. Der Speedup-Faktor für die *Experiment 6*-Datensätze konvergiert gegen vier. Bei den *Lenses*-Datensätzen hingegen wird bei 24 Kernen ein Speedup von mehr als zehn erreicht. In den auf Abbildung 3.2 visualisierten Graphen ist eine starke Skalierung deutlich erkennbar.

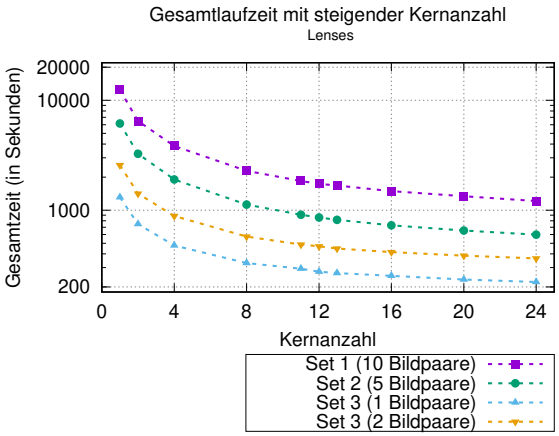
Um Engpässe und besonders rechenaufwendige Funktionen zu identifizieren, wurde das Programm mit Zeitmessern versehen, die Ausführungszeiten und Aufrufanzahl protokolliert haben. Anschließend wurde es auf einem Rechenkern unter denselben Bedingungen, wie die anderen Konfigurationen, ausgeführt und die Zeiten wurden gemessen. Ein Überblick über das Gesamtprogramm mit seinen Subroutinen und deren Anteil an der Gesamtlaufzeit ist in Abbildung 3.3 zu sehen.

Hierbei ist eindeutig zu sehen, dass die meiste Zeit für das Speckle-Tracking benötigt wird. Um weitere Informationen über die Laufzeiten der einzelnen Speckle-Tracking-Schritte zu gewinnen, wurde dieses ebenfalls mit Zeitmessern versehen. Die zeitliche Aufteilung dieser zeigt Abbildung 3.4, dass hierbei der zweite Durchlauf am meisten Zeit benötigt.

Die kumulative Zeit der fünf rechenaufwendigsten Funktionen aller Konfigurationen, dargestellt in Abbildung 3.5, liegt jeweils bei über 95% der Gesamtzeit.

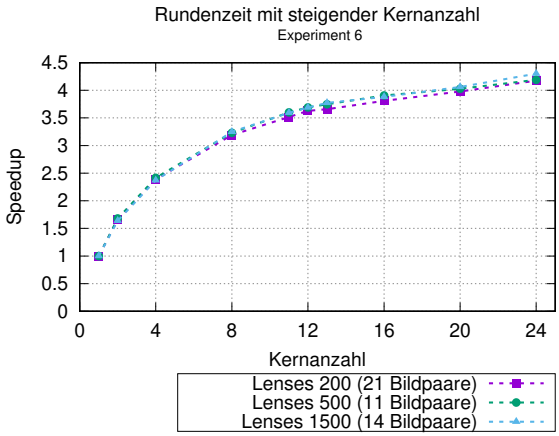


(a) Experiment 6

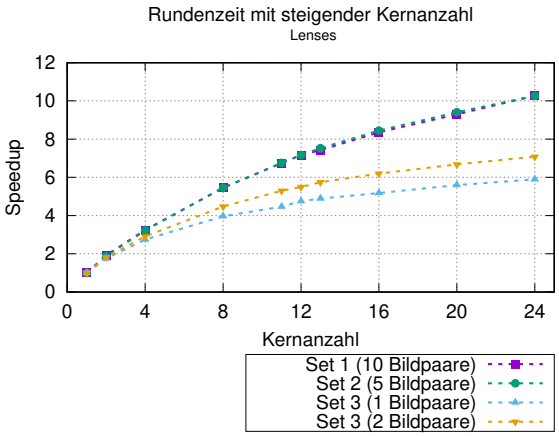


(b) Lenses

Abbildung 3.1: Gesamtlaufzeiten

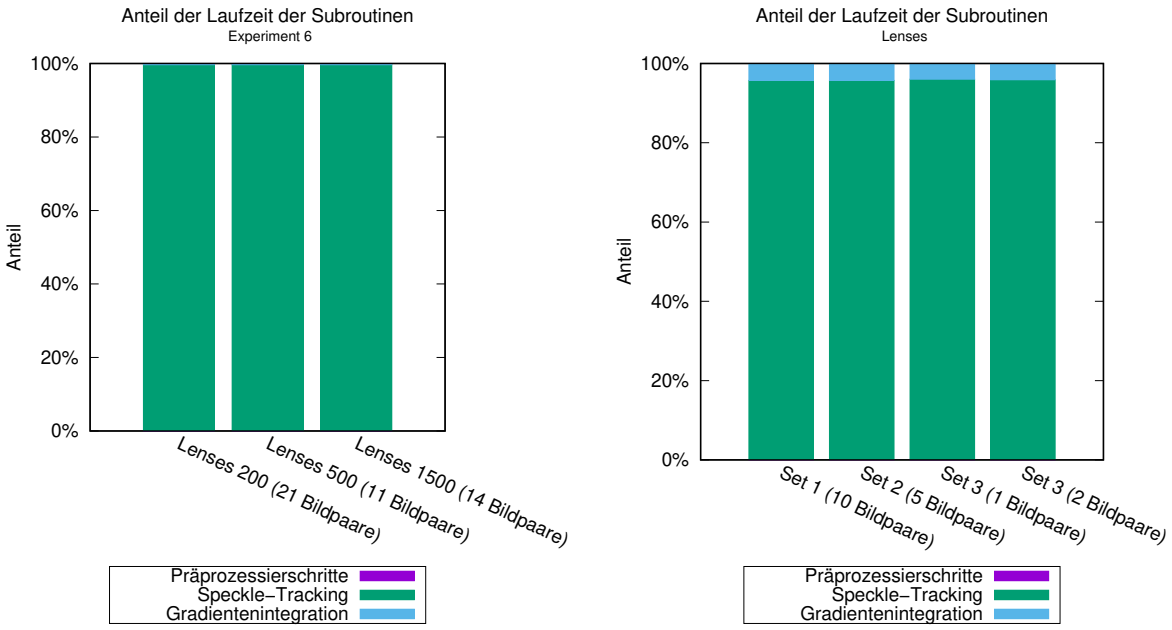


(a) *Experiment 6*



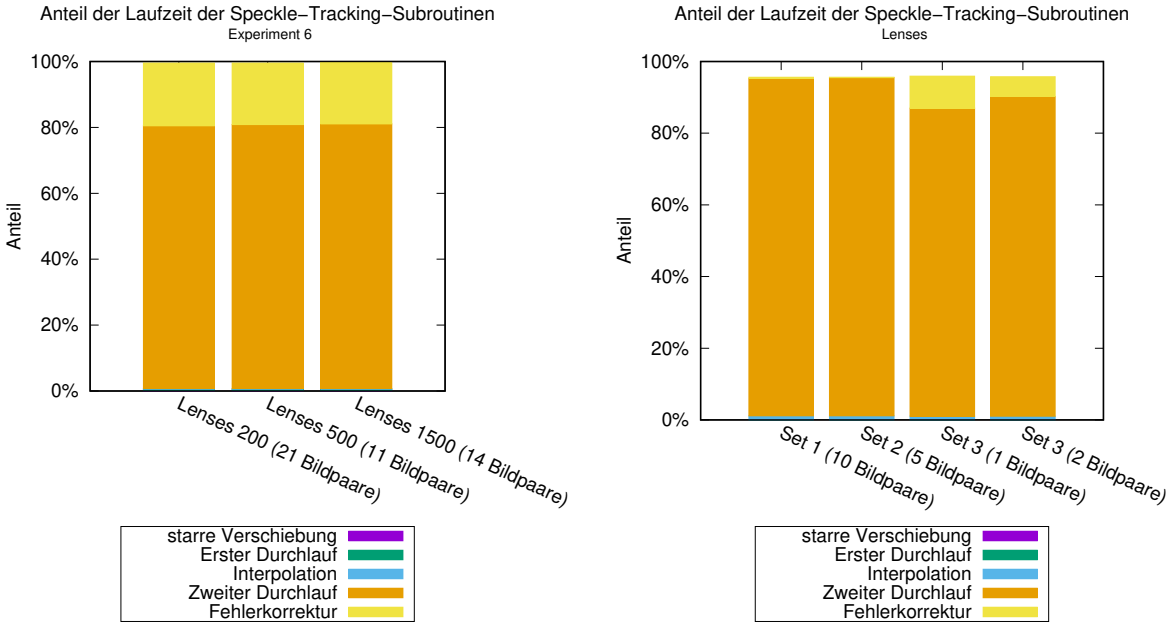
(b) *Lenses*

Abbildung 3.2: Speedup



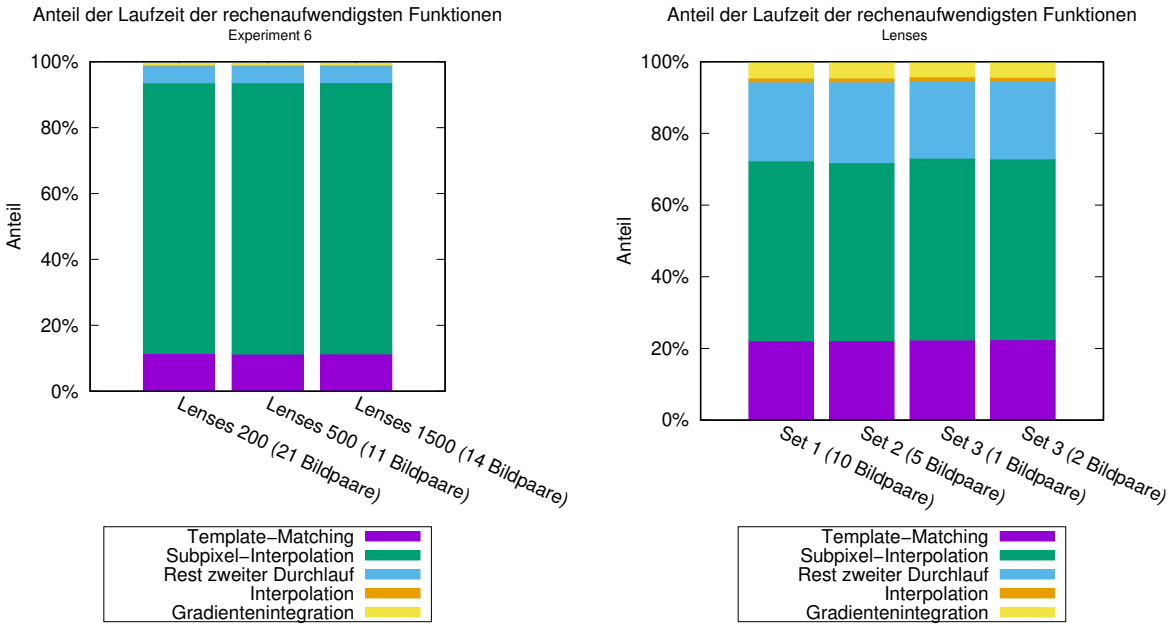
(a) Experiment 6 (b) Lenses

Abbildung 3.3: Anteile der Laufzeiten



(a) Experiment 6 (b) Lenses

Abbildung 3.4: Anteile der Laufzeiten des Speckle-Tracking-Algorithmus



(a) *Experiment 6* (b) *Lenses*

Abbildung 3.5: Anteile der Laufzeiten der langsamsten Funktionen

3.3 Performance-Engpässe

Der Grund der langen Rechenzeiten des Template-Matchings und der Subpixel-Interpolation liegt in der hohen Anzahl der Aufrufe dieser begründet. Der zweite Durchlauf allein wird im *Experiment 6 Lenses 200*-Datensatz über 5.3 Millionen mal aufgerufen. In jedem dieser Aufrufe wird das Template-Matching und die Subpixel-Interpolation jeweils einmal genutzt. Hinzu kommt, dass, bis auf das Template-Matching, der zweite Durchlauf nur geringen Gebrauch von bereits optimierten Bibliotheken wie numpy macht und somit der Python-Overhead starken Einfluss auf die Laufzeiten hat. Innerhalb des Speckle-Trackings ist der Aufruf des zweiten Durchlaufes mittels der joblib parallelisiert. Diese nutzt standardmäßig die multiprocessing-Bibliothek, welche für jeden Thread einen Fork der gesamten Python-Umgebung erstellen muss [Gri+18]. Die hohe Rechenzeit der Gradienten-Integration ist im Aufruf dieser auf die Größe des Gesamtbildes begründet. Insgesamt hat das Programm eine schlechte CPU-Auslastung von lediglich durchschnittlich 19,635% [Sch18a], wodurch häufig einige Kerne nicht oder nur wenig genutzt werden.

3.4 Prüfen der Ergebnisse

Um die Korrektheit der Ergebnisse nach der Optimierung sicherstellen zu können, wurden die Ergebnisse der Referenzimplementierung unter Eingabe aller Datensätze gespeichert. Diese werden als Referenz für einen bitweisen Abgleich verwendet. Auf Fließkommazahlen basierende Ungenauigkeiten werden hierbei bereits als Fehler gewertet. Nach den initialen Benchmarks wurde diese Methode zur Sicherstellung der Datenintegrität eingesetzt und für geeignet befunden.

4 Parallelisierung der kritischen Abschnitte

Um der optimalen Leistung nah zu kommen, wurde bei der Implementierung ein iterativer Ansatz gewählt. Hierzu wurden zuerst die Möglichkeiten der Parallelisierung und anschließend die, der Optimierung in Python betrachtet.

4.1 Parallelisierung

4.1.1 Parallelisierung der Verarbeitung einzelner Bildpaare mittels MPI

Da die zu bearbeitenden Bildpaare voneinander unabhängig sind, lassen diese sich trivial parallelisieren. Der Vorteil dieses Ansatzes liegt besonders in seiner simplen Implementierung und erwarteten linearen Skalierung begründet. Dieser Ansatz bringt allerdings auch Nachteile mit sich: Einige Message Passing Interface (MPI)-Implementierungen erlauben das Erstellen neuer Threads nur, wenn dies beim Installieren der Bibliotheken angegeben wird [Pro17] und es ist nur eine schwache Skalierung zu erwarten. Sofern kein Multithreading innerhalb von MPI möglich ist, limitiert die Anzahl der Bildpaare die Parallelisierungsmöglichkeit stark. In der auf dem GitHub-Branch *mpi* verfügbaren Implementierung dieses Ansatzes wird pro Bildpaar ein Kern genutzt [CS17]. Die Initialisierung wird dabei vom Wurzelknoten übernommen, dessen Aufgabe es auch ist, die Bildpaare auf weitere Kerne zu verteilen. Zum Schluss werden die Ergebnisse wieder beim Wurzelknoten gesammelt und gespeichert.
Die parallele Bearbeitung eines Bildpaares wird, wie bereits zuvor, von der joblib-Bibliothek übernommen, welche die Nutzung von Multithreading in Python ermöglicht. Hierbei wurde nicht sichergestellt, dass unter Verwendung mehrerer Rechenknoten die Bildpaare gleichmäßig auf die diese verteilt wurden.

4.1.2 Parallelisierung innerhalb der Verarbeitung einzelner Bildpaare mittels MPI

Eine sinnvolle Erweiterung zur oben beschriebenen Methode ist das Ersetzen der genutzten Multithreading-Bibliothek mittels MPI, sodass selbst die Berechnung eines einzelnen Bildpaares über Rechnergrenzen hinweg möglich ist. In diesem Zuge wurde auch die Fehlerkorrektur am Ende des Speckle-Trackings parallelisiert, indem die zu korrigierende Bildausschnitte auf mehrere Kerne verteilt wurden. Zusätzlich ermöglicht diese Implementierung den Einsatz eines Tracing-Programmes, wie SCORE-P [Knü+12]. Dies war aufgrund der unterliegenden multiprocessing-Bibliothek zuvor nicht möglich. Ein hoher Speedup wird insbesondere für wenige zu korrigierende Bildausschnitte nicht erwartet.
Im Konkreten werden die Bildpaare auf CPU-Kern Gruppen verteilt. Einer dieser Kerne innerhalb der Gruppe agiert hierbei als Hauptkern und ist dafür verantwortlich das Bildpaar zu verarbeiten, wobei dieser Aufgaben mittels eines MPI-Kommunikators an die anderen Rechenkerne der Gruppe verteilen kann. Die Aufgabe des Wurzelkerns ist es, die Bildpaare gleichmäßig auf die Gruppen zu verteilen. Dies ist in der Abbildung 4.1 gezeigt. Sollten mehr Bildpaare als Rechenkerne vorhanden sein, werden mehrere Bildpaare von eine Kern hintereinander verarbeitet.
Die Programmierschnittstelle wurde so entworfen, dass die Verteilung der Bildpaare auf die Kerne und das Parallelisieren innerhalb dieser für den Programmierer transparent geschieht. Hierbei gibt der Nutzer lediglich drei Funktionszeiger an: einen für die Initialisierungsfunktion, einen für die Hauptverarbeitungsroutine und einen für die Endfunktion. Die Initialisierungsfunktion generiert hierbei zwei Listen: eine mit globalen Parametern, die später an alle Kerne gesendet wird, und eine mit lokalen Parametern, die auf die Kerne verteilt wird. Die so verteilten Daten werden der Hauptverarbeitungsroutine übergeben,

welche die Daten parallel verarbeitet und das Ergebnis zurückgibt. Dieses wird anschließend auf dem Wurzelknoten gesammelt und der Endfunktion übergeben.
Innerhalb der Hauptverarbeitungsroutine bietet eine Schnittstelle auf die CPU-Kern-Gruppe eine weitere Parallelisierungsmöglichkeit. Diese Schnittstelle wurde hierbei ähnlich zur joblib-Implementierung entworfen. Auch hier werden wieder ein Funktionszeiger und lokale sowie globale Parameter entgegengenommen, die analog zu der oben beschriebenen Methode in der Gruppe verteilt werden. Eine exemplarische Implementierung eines Programmes, welches jedes Element in einer Matrix mit einem bestimmten Wert multiplizieren und anschließend einen weiteren aufaddieren soll, ist in Listing 4.1 gezeigt. Ein mittels der joblib implementiertes, funktional äquivalentes Programm ist in Listing 4.2 dargestellt. Die äquivalente Ausgabe der beiden Programme ist in Listing 4.3 dokumentiert. Die Implementierung dieses Ansatzes ist auf dem GitHub-Branch *mpi-advanced* zu finden [CS17].

Listing 4.1: Parallelisierung mittels der in MPI implementierten Version

```
from distributor import Distributor
def initFn():
    global_args = (1, 2)
    local_args = [[1, 5], [2, 6], [3, 7], [4, 8]]
    return (global_args, local_args)
def addFn(global_args, local_args):
    (multiply, add) = global_args
    return local_args * multiply + add
def mainFn(global_args, local_args, dist):
    return dist.parallel(addFn, global_args, local_args)
def exitFn(global_args, result):
    print("The result is", result)
    return 0
dist = Distributor(initFn, mainFn, exitFn)
```

Listing 4.2: Parallelisierung mittels der joblib-Bibliothek

```
from joblib import Parallel, delayed
import multiprocessing
def addFn(array, multiply, add):
    result = []
    for element in array:
        result += [element * multiply + add]
    return result
matrix = [[1, 5], [2, 6], [3, 7], [4, 8]]
multiply = 1
add = 2
n_jobs = len(matrix)
result = Parallel()(delayed(addFn)(matrix[k], multiply, add) for k in range(
    n_jobs))
print("The result is", result)
```

Listing 4.3: Ausgabe der Programme

```
The result is [[3, 7], [4, 8], [5, 9], [6, 10]]
```

4.2 Optimierung der Performance-Engpässe in Python

Wie in Abbildung 3.5 zu sehen ist, werden über 95% der Rechenzeit in den fünf langsamsten Funktionen verbracht. Unter diesen ist insbesondere die `nxcorr_disp()`-Funktion, welche komplett in Python implementiert ist. Um die Leistung solcher Funktionen zu verbessern werden im folgenden Methoden zur Optimierung des bestehenden Codes betrachtet mit besonderem Augenmerk auf der Beschleunigung der langsamsten Funktionen.

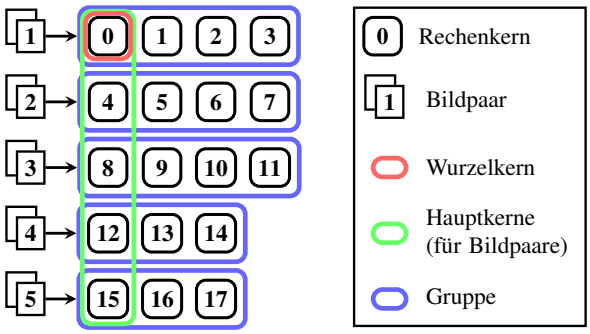


Abbildung 4.1: Verteilung von fünf Bildpaaren auf 18 Rechenknoten

4.2.1 Nutzen bereits optimierter Funktionen

Einige Teile des Codes können durch bereits in Python oder einer optimierten Bibliothek enthaltenen Funktion ersetzt werden, womit der Interpretieraufwand erheblich reduziert wird. Dies gehört damit zu einer der grundlegenden Optimierungsmöglichkeiten. Zusätzlich dazu enthalten diese Funktionen bereits Plattform-spezifische Optimierungen. In der Funktion `nxcorr_disp()` lassen sich Code-Abschnitte mit diesem Verbesserungspotential finden. Das Listing 4.4 zeigt den Code aus dieser Funktion zum Ermitteln eines Maximums einer Matrix in reinem Python-Code, wobei die äußeren Zeilen und Spalten vernachlässigt werden. Listing 4.5 zeigt einen funktional äquivalenten Code unter Nutzung von bereits optimierten Funktionen. `lengthX` und `lengthY` geben hier die Dimensionen der Eingabematrix `nxcorr` an.

Listing 4.4: Finden des Maximums einer Matrix

```
for i in range(1, lengthY - 1):
    for j in range(1, lengthX - 1):
        if(nxcorr[i, j] > maxValue):
            maxValue = nxcorr[i, j]
            maxI = i
            maxJ = j
```

Listing 4.5: Finden des Maximums einer Matrix mittels NumPy und OpenCV

```
nxcorr_small = nxcorr[1:-1, 1:-1]
(_, maxValue, _, (maxJ, maxI)) = cv2.minMaxLoc(nxcorr_small)
maxI += 1
maxJ += 1
```

Des Weiteren befindet sich in der `nxcorr_disp()`-Funktion die Berechnung des Signal-Rausch-Verhältnisses (gezeigt im Listing 4.6), was allerdings im weiteren Verlauf des Programmes nicht wieder verwendet wird und deshalb entfernt werden kann.

Listing 4.6: Berechnung des Signal-Rausch-Verhältnisses

```
avg = 0.0
count = 0
for i in range(lengthY):
    for j in range(lengthX):
        if((i is not maxI) and (j is not maxJ)):
            avg = avg + abs(nxcorr[i,j])
            count = count + 1
avg = avg / float()count)
SNr = maxValue / avg
```

Nach dem Anwenden dieser Änderungen befindet sich keine in Python implementierte Schleife mehr in der Funktion. Angesichts der hohen Aufrufzahl von `nxcorr_disp()` und dem Entfernen großer Codeanteile ist ein hoher Beschleunigungsfaktor zu erwarten.
Zum Schluss wurde die Evaluierung des zu verwendenden Korrelationsalgorithmus in der `norm_xcorr()`-Funktion, gezeigt in Listing 4.7, durch eine direkte Variable, gezeigt in Listing 4.8, ersetzt. Eine Version des Codes mit allen hier vorgeschlagenen Optimierungen ist auf dem Branch `intrinsics` des GitHub-Repositorys zu finden [CS17].

Listing 4.7: Evaluierung der Korellationsmethode

```
1 def norm_xcorr(template, searchArea, method='cv2.TM_CCOEFF_NORMED'):
2     meth = eval(method)
3     #...
```

Listing 4.8: Übergabe der Korellationsmethode als Variable

```
1 def norm_xcorr(template, searchArea, method=cv2.TM_CCOEFF_NORMED):
2     meth = method
3     #...
```

4.2.2 Kompilieren

Eine weitere Möglichkeit der Minimierung des Python-Engpasses ist die Übersetzung des Codes in nativen Maschinencode. Die möglichen Ansätze hierbei reichen von der Übersetzung des gesamten Programmes über die Übersetzung einzelner Funktionen, die in Python dann als Modul geladen werden können, bis hin zur Nutzung eines just-in-time (JIT) Compilers, welcher annotierte Funktionen bei dessen ersten Aufruf in nativen Maschinencode übersetzt.

Gesamtes Programm

Die einfachste Möglichkeit der Übersetzung ist es, das gesamte Programm in Maschinencode zu übersetzten. Hierzu kann die Bibliothek cython [Beh+17] genutzt werden, welche Python-Code in C übersetzt, was anschließend in Maschinencode übersetzt werden kann. Hierzu wurde nur die Datei *waveFront.py* unübersetzt gelassen, da diese keine rechenaufwendigen Funktionen enthält und diese nur aus anderen Dateien, insbesondere der *func.py* aufruft. Die benötigten Installationsdateien sind auch dem GitHub-Branch *compiled* verfügbar [CS17].

Einzelne Funktionen

numba numba ist eine Optimierungsbibliothek für Python, welche unter anderem die Möglichkeit bietet, Funktionen JIT zu übersetzen sowie CUDA und OpenCL zu nutzen [LPS15]. Funktionen können zur JIT-Übersetzung mittels der Annotation `@jit` markiert werden. Sobald während der Ausführung diese Funktion erreicht wird, übersetzt numba den Code in eine Intermediate-Repräsentation, welche anschließend von LLVM weiter in Maschinencode übersetzt wird. Beim nächsten Aufruf wird sofort der Maschinencode verwendet. Ein permanentes Speichern des Übersetzungsergebnisses ist mittels der Annotiation `@jit(cached = True)` möglich. Auf dem `numba`-Branc befindet sich eine Version des Codes, in dem diese Annotationen genutzt werden [CS17]. Annotiert wurden hierbei Funktionen zur Auswahl der ROI und die `nxcorr_disp()`-Funktion.

Cython Eine weitere weitaus mächtigere Methode zur Übersetzung einzelner Funktionen bietet die Cython-Bibliothek, welche bereits genutzt wurde, um das gesamte Programm zu übersetzen. Diese führt die Möglichkeit der Typisierung ein und lässt die direkte Einbindung von C zu. In der auf dem GitHub-Branch *cython* verfügbaren Version wurden Cython genutzt, um die Funktionen `norm_xcorr()` (siehe Listing 4.9) und `nxcorr_disp()` (siehe Listing 4.10) zu übersetzen [CS17]. Hierbei wurden alle

Variablen mit Typen versehen und es wurde die von Cython bereitgestellte Schnittstelle zu NumPy genutzt. Auf dem Branch *compiled-advanced* wurde die Übersetzung des gesamten Programmes mit den hier optimierten Funktionen zusammengeführt [CS17].

Listing 4.9: Die in Cython optimierte Funktion norm_xcorr()

```
def norm_xcorr(np.ndarray[float, ndim=2] template, np.ndarray[float, ndim=2]
    searchArea, int method = cv2.TM_CCOEFF_NORMED):
    cdef double minVal = 0.0, maxVal = 0.0
    cdef tuple minLoc, maxLoc
    cdef np.ndarray[float, ndim=2] corr = cv2.matchTemplate(searchArea, template
        , method)
    minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(corr)
    return corr, minVal, maxVal, minLoc, maxLoc
```

Listing 4.10: Die in Cython optimierte Funktion nxcorr_disp()

```
def nxcorr_disp(np.ndarray[float, ndim=2] nxcorr):
    cdef int lengthY = nxcorr.shape[0]
    cdef int lengthX = nxcorr.shape[1]
    # find maximum
    nxcorr_small = nxcorr[1:-1, 1:-1]
    cdef int maxI, maxJ
    (_, _, _, (maxJ, maxI)) = cv2.minMaxLoc(nxcorr_small)
    maxI += 1
    maxJ += 1
    # calculate offset using gaussian subpixel interpolation
    cdef double dy = (nxcorr[maxI + 1, maxJ] - nxcorr[maxI - 1, maxJ]) / 2.0
    cdef double dyy = nxcorr[maxI + 1, maxJ] + nxcorr[maxI - 1, maxJ] - 2.0 *
        nxcorr[maxI, maxJ]
    cdef double dx = (nxcorr[maxI, maxJ + 1] - nxcorr[maxI, maxJ - 1]) / 2.0
    cdef double dxx = (nxcorr[maxI, maxJ + 1] + nxcorr[maxI, maxJ - 1] - 2.0 *
        nxcorr[maxI, maxJ])
    cdef double dxy = (nxcorr[maxI + 1, maxJ + 1] - nxcorr[maxI + 1, maxJ - 1] -
         nxcorr[maxI - 1, maxJ + 1] + nxcorr[maxI - 1, maxJ - 1]) / 4.0
    # calculate normalization factor
    cdef double det = 0.0
    if((dxx * dyy - dxy * dxy) != 0.0):
        det = 1.0 / (dxx * dyy - dxy * dxy)
    # calculate new subpixel indices
    cdef double ix = - (dyy * dx - dxy * dy) * det + maxJ - (lengthX//2)
    cdef double iy = - (dxx * dy - dxy * dx) * det + maxI - (lengthY//2)
    out = []
    out.append(iy)
    out.append(ix)
    return out
```

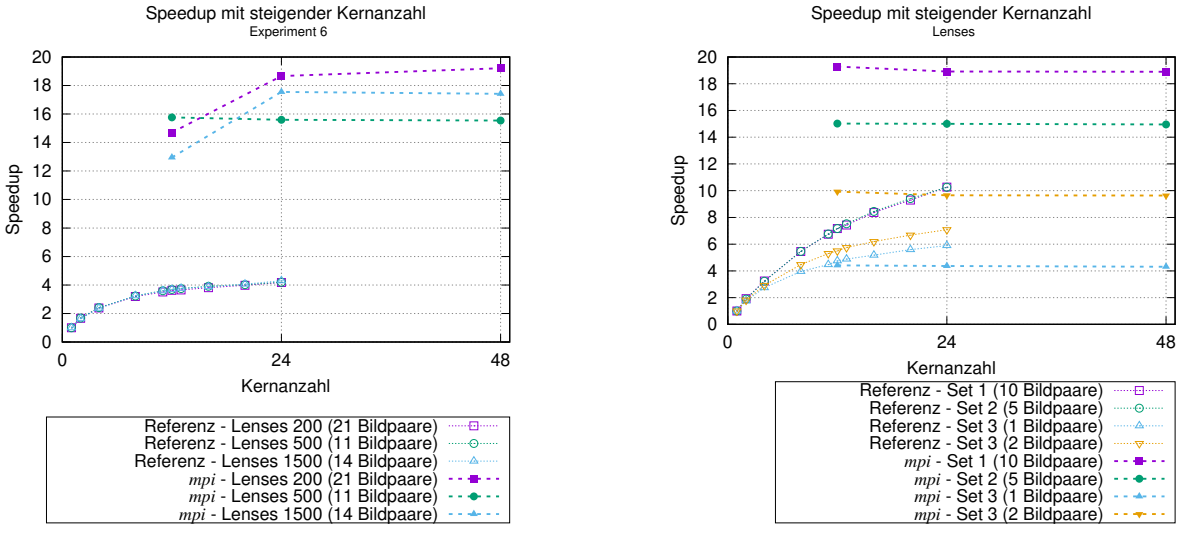
5 Performance-Messungen der parallelen Implementation

Das Testsystem ist dasselbe geblieben, wie bei der Performance-Analyse. Diesmal wird allerdings in vielen Benchmarks mehr als ein Knoten beansprucht. Auch die Datensätze und die geladenen Module haben sich nicht geändert. Wie auch bei der Performance-Analyse, sind aus alle in diesem Kapitel angegeben Zeiten die I/O-Zeiten herausgerechnet. Ebenfalls wurde hier auch wieder nach vier Aufwärmiterationen für fünf Iterationen des Programmes die Zeit gemessen.

5.1 Evaluierung der Optimierungen

5.1.1 Parallelisierung

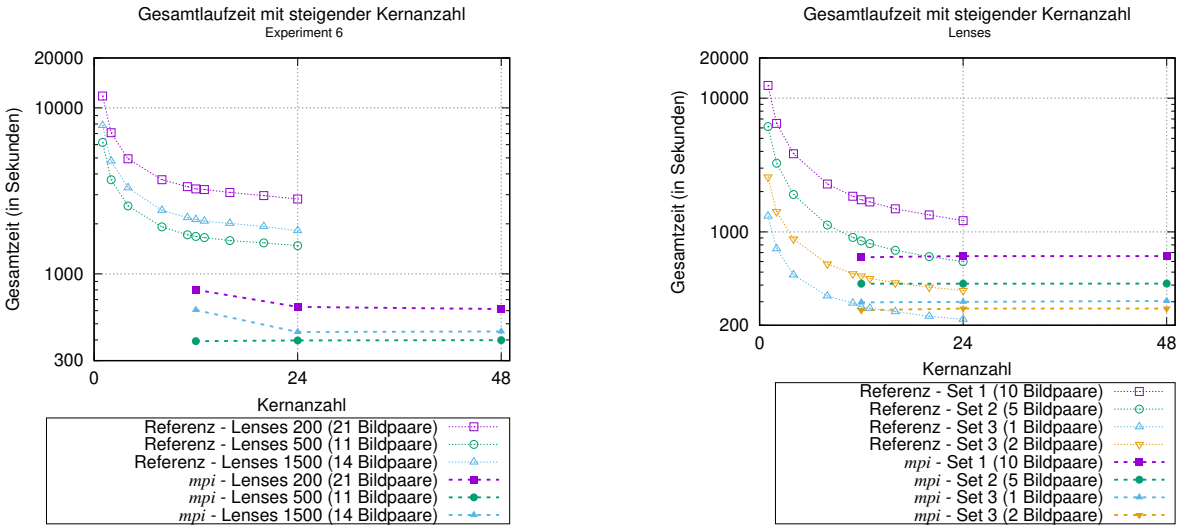
Auf der Abbildung 5.1 ist für die auf dem GithHub-Branch *mpi* verfügbare Version [CS17] deutlich ein Beschleunigungsfaktor von ca. zwei bis vier gegenüber der vorgegebenen Implementierung erkennbar. Anzumerken ist hierbei, dass als Referenz für den Speedup die Laufzeit der vorgegebenen Implementierung auf eine Kern genutzt wurde. Ebenfalls wird ersichtlich, dass diese Lösung nur mit der Anzahl der Eingabebildpaare skaliert, weshalb nur die *Experiment 6 Lenses 200* und *Lenses 500* mit 24 Kernen schneller ist als mit zwölf. Wie in Abbildung 5.2 zu sehen ist, liegen alle Laufzeiten dieser Version unter 1.000 Sekunden. Auch hier sind die Messpunkte wieder dick hervorgehoben und die Skala der Gesamtzeitgraphen ist logarithmisch eingeteilt. Ein Rechenknoten besitzt 24 Kerne, wodurch die Einteilung der X-Achse die Grenze der Rechenknoten verdeutlicht.



(a) Experiment 6

(b) Lenses

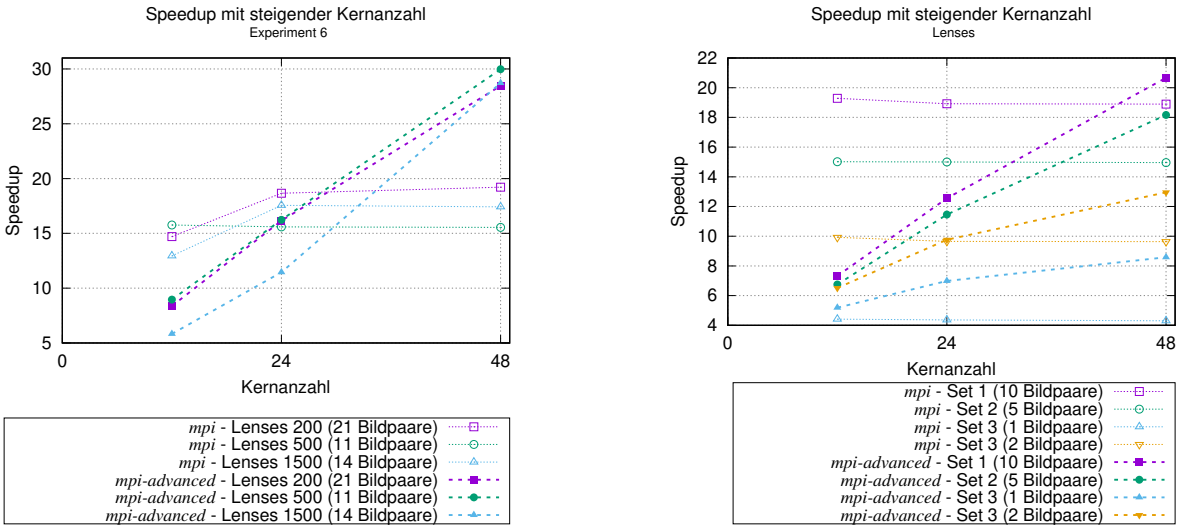
Abbildung 5.1: Speedup der *mpi* Implementierung gegenüber des vorgegebenen Python-Codes



(a) Experiment 6 (b) Lenses

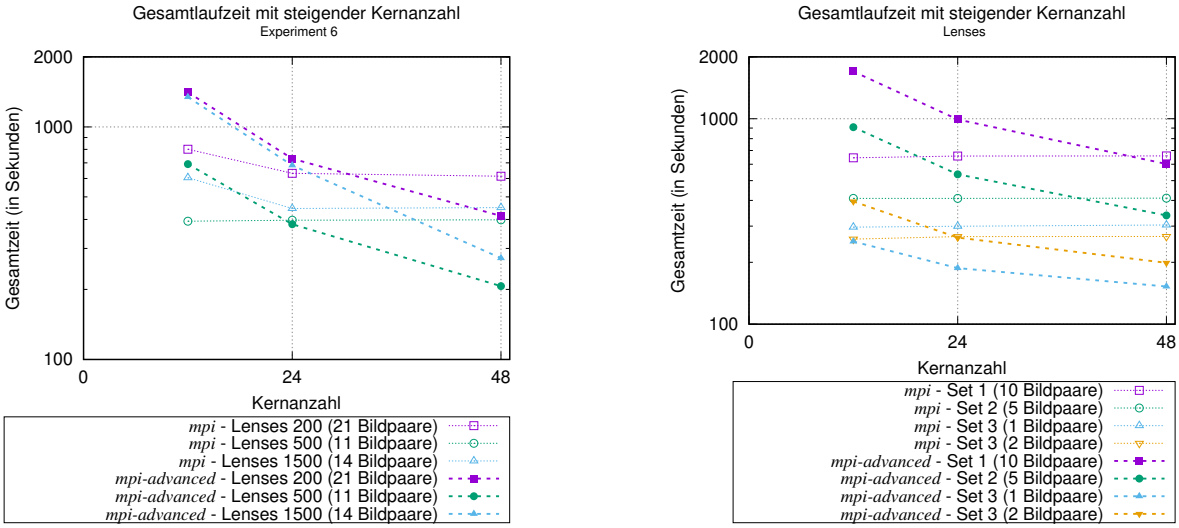
Abbildung 5.2: Gesamtlaufzeit der *mpi* Implementierung gegenüber des vorgegebenen Python-Codes

Wie in Abbildung 5.3 zu sehen ist, schneidet die auf dem Branch *mpi-advanced*-[CS17] verfügbare Version für 24 Kerne schlechter ab, als die *mpi* Implementierung. Dies kann in der effizienteren Verteilung der Daten mittels der joblib-Bibliothek begründet liegen, da diese den in Linux effizient implementierten Fork-Befehl nutzt, wohingegen die *mpi-advanced*-Implementierung die Daten direkt kopiert [Gri+18]. Im Gegensatz zu der *mpi*-Implementierung, skaliert diese Version aber mit der Anzahl der verfügbaren CPU-Kerne und nicht nur mit der Anzahl der Bildpaare. Anzumerken ist hier ebenfalls, dass die *mpi*-Implementierung bei wenigen Kernen zwar schneller ist, aber deutlich mehr Arbeitsspeicher benötigt. Währenddessen die *mpi-advanced*-Version mit 64 GiB auskommt, benötigt die *mpi*-Version mehr als 128 GiB.
Auch in der Abbildung 5.3 wurde wieder die auf einem Kern ausgeführte vorgegebene Implementierung als Referenz genutzt.



(a) Experiment 6 (b) Lenses

Abbildung 5.3: Speedup der *mpi-advanced* Implementierung gegenüber der *mpi*-Version



(a) Experiment 6

(b) Lenses

Abbildung 5.4: Gesamtlaufzeit der *advanced-mpi* Implementierung gegenüber des vorgegebenen Python-Codes

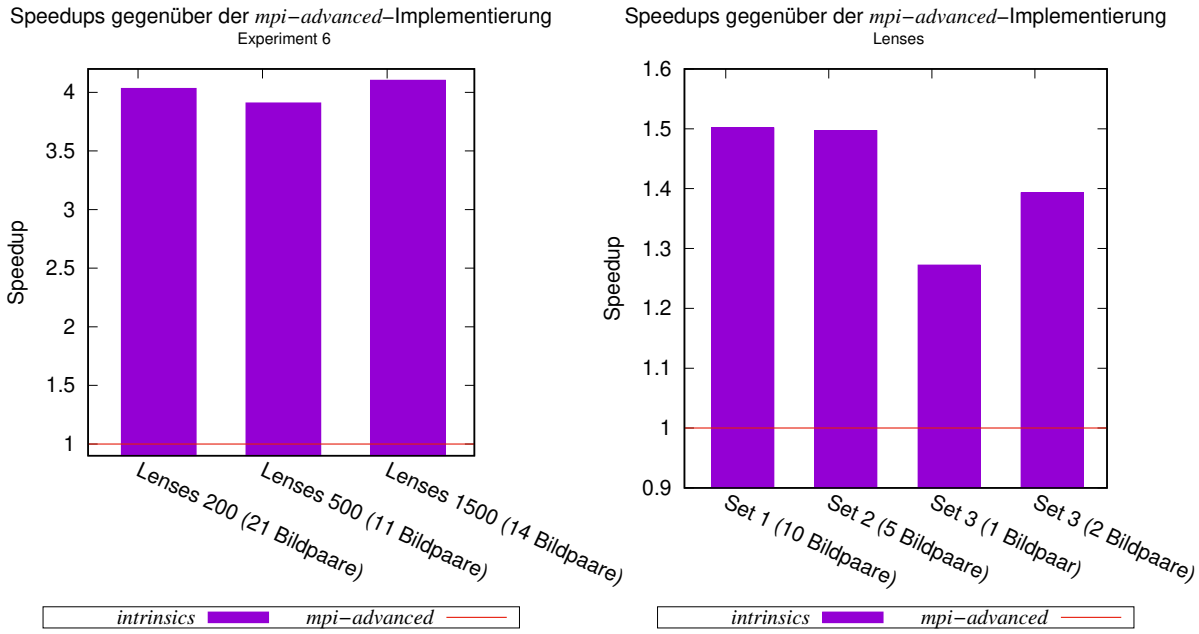
5.1.2 Optimierung von Python Engpässen

Nutzen bereits optimierter Funktionen

Wie in Abbildung 5.5 zu sehen ist, wurde mittels der Nutzung von bereits optimierten Funktionen ein Beschleunigungsfaktor von ca. vier für die *Experiment 6*-Datensätze und ein Faktor zwischen 1,2 und 1,5 für die *Lenses*-Datensätze erreicht. Der Speedup bei diesen Datensätzen war nicht so hoch, da diese weniger Bildpaare beinhalten und die Rechenzeit pro Bildpaar deutlich kleiner war, wodurch der Overhead durch das Senden der Daten einen größeren Einfluss auf die Gesamtlaufzeit hat. Die geringere Rechenzeit pro Bildpaar kommt durch die deutlich kleinere ROI, eine kleinere Korrelationsgröße R_{corr} und weniger zu korrigierenden Punkten zustande. Zusätzlich dazu entfällt die Notwendigkeit der Interpolation zwischen zwei verschiedenen Pixelgrößen, da hierfür nur ein Sensor im Einsatz war.

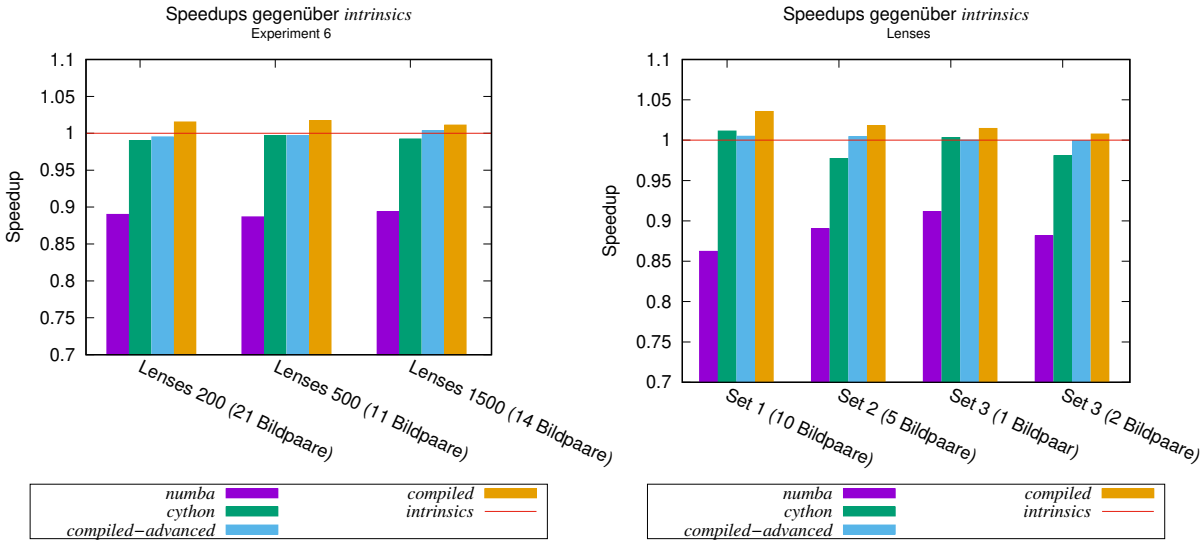
Kompilieren

Die Abbildung 5.6 zeigt den Beschleunigungsfaktor der verschiedenen übersetzten Versionen gegenüber der Implementierung, welche bereits optimierte Funktionen nutzt. Hierbei ist deutlich ersichtlich, dass die Übersetzung des gesamten Programmes, welche auf dem *compiled*-Branch verfügbar ist, immer schneller als die *intrinsics*-Implementierung läuft. Der Grund für die höhere Leistung der *compiled*-Implementierung ist hierbei nicht bekannt. Die *numba*-Version hingegen schnitt immer deutlich schlechter, als die anderen Versionen ab. Die Laufzeiten der *cython* und der *compiled-advanced*-Versionen liegen nahe beieinander und bieten keinen Geschwindigkeitszuwachs gegenüber der *intrinsics*-Implementierung.



(a) Experiment 6 (b) Lenses

Abbildung 5.5: Speedup der *intrinsics* Implementierungen gegenüber der *mpi-advanced*-Implementierung mit zwölf Kernen



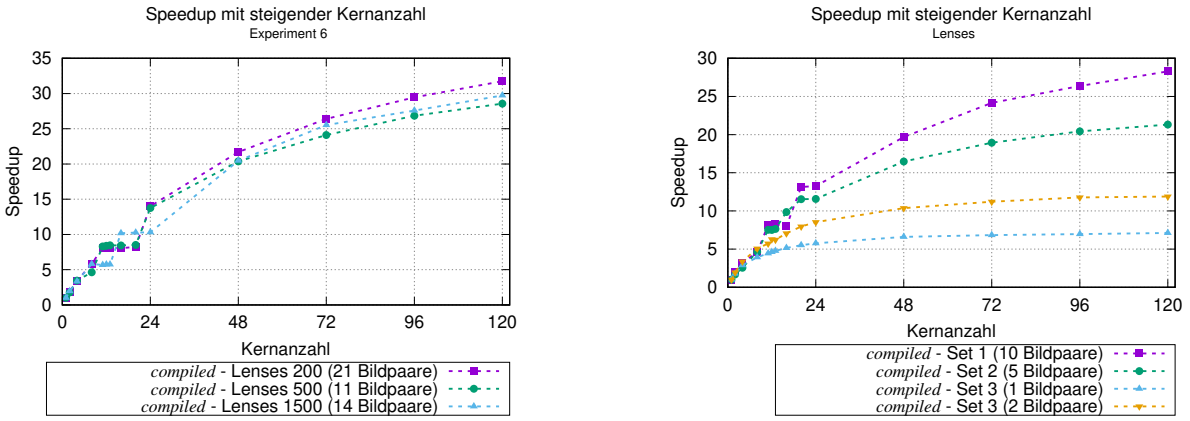
(a) Experiment 6 (b) Lenses

Abbildung 5.6: Speedups der einzelnen Implementierungen gegenüber der *intrinsics* Implementierung mit zwölf Kernen

Die bereits gute Performance der *intrinsics*-Implementierung liegt in der intensiven Nutzung optimierter Funktionen begründet, welche von einer Übersetzung der Python-Codes unberührt bleiben. Die Performance der *numba*-Implementierung ist deutlich schlechter, da beim Aufruf einer JIT-kompilierten Funktion diese erst in einer Liste aus übersetzten Funktionen gesucht werden muss, bevor diese aufgerufen werden kann [PLA17]. Dieser Effekt wird dadurch verstärkt, dass die übersetzten Funktionen eine geringe Laufzeit haben, aber dafür sehr oft aufgerufen werden. Der zweite Durchlauf allein, und damit auch die `nxcorr_disp()`-Funktion, wird im *Experiment 6 Lenses 200*-Datensatz über 5.3 Millionen mal aufgerufen.

5.2 Skalierung

Auf der Abbildung 5.7 ist eine lineare Skalierung gut erkennbar, bis die Anzahl der CPU-Kerne die Anzahl der Bildpaare übersteigt. Anschließend stagniert der Beschleunigungsfaktor, da einige Bildpaare zwar mit mehr Kernen schneller bearbeitet werden können, aber das Programm auf die Fertigstellung der restlichen Bildpaare warten muss, die nur einen Kern zur Verfügung haben. Ein ähnliches Laufzeitverhalten lässt sich jedes mal beobachten, wenn die Kernanzahl ein Vielfaches der Bildpaaranzahl erreicht. Da die Verarbeitung eines einzelnen Bildpaares nicht komplett parallelisiert werden kann, flacht der Graph mit steigender Kernanzahl ab. Die Gesamtlaufzeiten im Bezug auf die vorgegebene Implementierung ist in Abbildung 5.8 zu sehen.

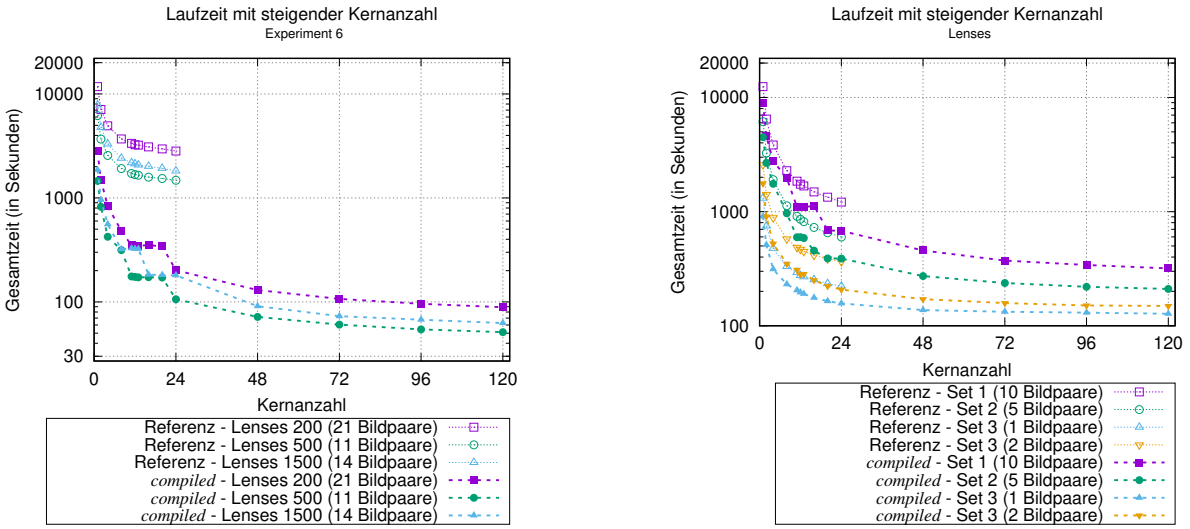


(a) Experiment 6

(b) Lenses

Abbildung 5.7: Speedup der *compiled* Implementierung

Der Speedup-Graph in Abbildung 5.7 weißt eine starke Skalierung auf, jedoch skaliert das Laufzeitverhalten schwach besser. Solang mehr Bildpaare als CPU-Kerne an der Berechnung beteiligt sind, skaliert das Programm nahezu linear. Das Effizienzoptimum wird somit erreicht, wenn die Anzahl der CPU-Kerne gleich der Anzahl der Bildpaare ist.
Eine Sättigung tritt bei allen Datensätzen ein, sobald mehr als 20 CPU-Kerne pro Bildpaar eingesetzt werden. An diesem Punkt wird die meiste Rechenzeit in nicht parallelisierten Abschnitten des Speckle-Trackings und in der Gradientenintegration verbracht.



(a) Experiment 6

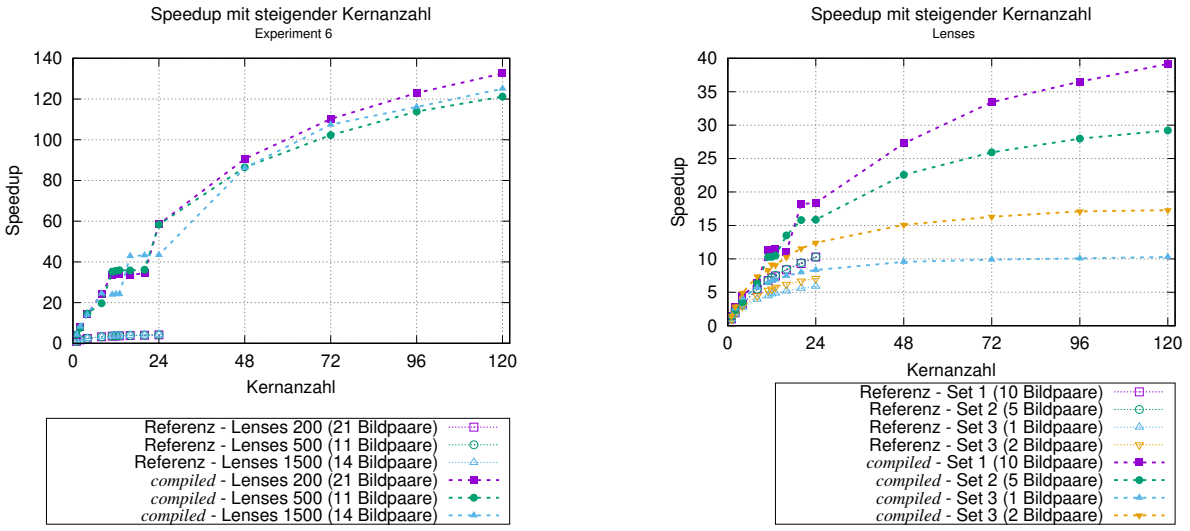
(b) Lenses

Abbildung 5.8: Gesamtlaufzeit der *compiled* Implementierung gegenüber des vorgegebenen Python-Codes

6 Auswertung

6.1 Wertung des Ergebnisses

Die Abbildung 6.1 zeigt einen deutlich erhöhten Speedup gegenüber der vorgegebenen Python-Implementierung. Für die *Experiment 6*-Datensätze erreicht dieser einen Speedup von über 130 und für die *Lenses*-Datensätze liegt der maximale Beschleunigungsfaktor bei bis zu 40 für den *Set 1*-Datensatz. Bei der Wahl größerer Datensätze sind größere Beschleunigungsfaktoren zu erwarten, da das Effizienzmaximum in den hier gezeigten Benchmarks weit überschritten wurde.



(a) Experiment 6

(b) Lenses

Abbildung 6.1: Speedup der *compiled* Implementierung gegenüber des vorgegebenen Python-Codes

Trotz dessen, dass mit 120 CPU-Kernen bereits ein Speedup von über 130 erreicht wurde, müsste die Anzahl der Bildpaare und der Kerne bei weitem höher sein, um die Echtzeitfähigkeit des Programmes sicher zu stellen. Dies ist in Größenordnungen von 2400 Bildpaaren und 2400 CPU-Kernen zu erwarten, was bereits 100 Taurus-Knoten wären.

6.2 Verbesserungsmöglichkeiten

Im Verlauf der Implementierung wurden viele der trivialsten Optimierungsmöglichkeiten implementiert. Während der letzten Iterationen der Implementierungen wurde allerdings auch weiteres Potential sichtbar. Beispielsweise ließe sich die Knoten interne Kommunikation mit Intrakommunikatoren oder geteiltem Speicher optimieren um den Kopieraufwand der Daten zu senken.
Obwohl in dieser Arbeit bereits viele Optimierungen implementiert wurden und ein hoher Speedup erreicht wurde, gibt es immer noch weiteres Potential. Die Verwendung der FFTW[1]-Bibliothek ist eine

[1] http://www.fftw.org/

davon. Mithilfe dieser ist es möglich, die Gradientenintegration zu beschleunigen indem beim Start des Programmes sogenannte *Wisdoms* erstellt werden, welche die Transformation eines Bildes in den Frequenzraum erheblich beschleunigen könnte.
Des Weiteren ist es möglich für die am häufigsten aufgerufenen Funktionen Datenblöcke zu bilden und diese zu verarbeiten, sodass der Overhead der Funktionsaufrufe sinkt. Dies bildet ebenfalls eine gute Grundlage um diese Datenblockverarbeitung mittels numpy, numexpr oder numba weiter zu beschleunigen. Insbesondere kann hierbei auch die CUDA und OpenCL Schnittstelle von numba genutzt werden, um Berechnungen mittels General Purpose Computing on Graphics Processing Units (GPGPUs) auszulagern.
Eine weitere Verbesserungsmöglichkeit liegt in einer Verbesserung des Belastungsausgleiches. Hierbei bestünde die Möglichkeit die Bildpaare in Packete zusammenzufassen und diese hintereinander auszuführen, sodass ein durchgängiger Betrieb möglich wäre.
Auch Optimierungen am Algorithmus sind noch denkbar. Da aus dem Ergebnis des Template-Matchings nur die Position und der Wert des Maximums benötigt werden, ließe sich Bild und Template mit durch Skalierung verminderter Auflösung matchen. Hierbei entstandene Ungenauigkeiten können behoben werden, indem in der Region des Maximums ein genaueres Match durchgeführt wird. Da das Match mit reduzierter Auflösung eine Heuristik über die Maximal mögliche Übereinstimmung ist, kann anschließend in diesem der zweit höchste Wert betrachtet werden. Ist dieser größer als das Ergebnis des genauen Matches, besteht die Möglichkeit, dass dieses falsch ist und es kann auf den in der OpenCV-Bibliothek implementierten Template-Matching-Algorithmus zurückgreifen. Ist die Auflösung des Templates und des Bildes jedoch nicht durch ein und denselben Skalierungsfaktor teilbar, so kann die Skalierung auf eine niedrigere Auflösung bereits erhebliche Zeit in Anspruch nehmen.
Zu guter Letzt besteht auch noch die Möglichkeit der Optimierung des Kalibrierungsteiles des Programmes.

Literatur

[Ana+07] ANAND, Arun; PEDRINI, Giancarlo; OSTEN, Wolfgang; ALMORO, Percival. Wavefront sensing with random amplitude mask and phase retrieval. *Opt. Lett.* 2007, Jg. 32, Nr. 11, S. 1584–1586. Abgerufen unter DOI: 10.1364/OL.32.001584.

[Beh+11] BEHNEL, Stefan; BRADSHAW, Robert; CITRO, Craig; DALCIN, Lisandro; SELJEBOTN, Dag Sverre; SMITH, Kurt. Cython: The Best of Both Worlds. *Computing in Science & Engineering*. 2011, Jg. 13, Nr. 2, S. 31–39. Abgerufen unter DOI: 10.1109/MCSE.2010.118.

[Beh+17] BEHNEL, Stefan; BRADSHAW, Robert; DALCÍN, Lisandro; FLORISSON, Mark; MAKAROV, Vitja; SELJEBOTN, Dag Sverre. *Cython: C-Extensions for Python*. 2017. Auch verfügbar unter: https:/http://cython.org/. Abgerufen: 24. Februar 2018.

[BR09] BEN ASHER, Yosi; ROTEM, Nadav. The Effect of Unrolling and Inlining for Python Bytecode Optimizations. In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. Haifa, Israel: ACM, 2009, 14:1–14:14. SYSTOR '09. ISBN 978-1-60558-623-6. Abgerufen unter DOI: 10.1145/1534530.1534550.

[Bér13] BÉRUJON, Sébastien. *Métrologie en ligne de faisceaux et d'optiques X de synchrotrons*. 2013. Auch verfügbar unter: http://www.theses.fr/2013GRENY010. Dissertation. Université de Grenoble. Thèse de doctorat dirigée par Ziegler, Eric et Sawhney, Kawal Physique Grenoble 2013.

[Bér+12] BÉRUJON, Sébastien; ZIEGLER, Eric; CERBINO, Roberto; PEVERINI, Luca. Two-Dimensional X-Ray Beam Phase Sensing. *Phys. Rev. Lett.* 2012, Jg. 108, S. 158102. Abgerufen unter DOI: 10.1103/PhysRevLett.108.158102.

[BZC15] BÉRUJON, Sébastien; ZIEGLER, Eric; CLOETENS, Peter. X-ray pulse wavefront metrology using speckle tracking. *Journal of Synchrotron Radiation*. 2015, Jg. 22, Nr. 4, S. 886–894. ISSN 1600-5775. Abgerufen unter DOI: 10.1107/S1600577515005433.

[CS17] COJOCARU, Elena-Ruxandra; SCHENKE, Jonas. *Wavefront-Sensor*. 2017. Auch verfügbar unter: https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/. (Privat).

[Dai09] DAILY, Jeffrey A. *GAiN: Distributed Array Computation with Python*. 2009. Abgerufen unter DOI: 10.2172/1006323. Dissertation.

[Dal+11] DALCIN, Lisandro D.; PAZ, Rodrigo R.; KLER, Pablo A.; COSIMO, Alejandro. Parallel distributed computing using Python. *Advances in Water Resources*. 2011, Jg. 34, Nr. 9, S. 1124–1139. ISSN 0309-1708. Abgerufen unter DOI: https://doi.org/10.1016/j.advwatres.2011.04.013. New Computational Methods and Software Tools.

[DPS05] DALCÍN, Lisandro; PAZ, Rodrigo; STORTI, Mario. MPI for Python. *Journal of Parallel and Distributed Computing*. 2005, Jg. 65, Nr. 9, S. 1108–1115. ISSN 0743-7315. Abgerufen unter DOI: https://doi.org/10.1016/j.jpdc.2005.03.010.

[Dal+08] DALCÍN, Lisandro; PAZ, Rodrigo; STORTI, Mario; DELÍA, Jorge. MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*. 2008, Jg. 68, Nr. 5, S. 655–662. ISSN 0743-7315. Abgerufen unter DOI: https://doi.org/10.1016/j.jpdc.2007.09.005.

[Duf] DUFOUR, Mark. *Shed Skin-An experimental (restricted) Python to C++ compiler (2009-09-30)*.

[Enk+11] ENKOVAARA, Jussi; ROMERO, Nichols A.; SHENDE, Sameer; MORTENSEN, Jens J. GPAW - massively parallel electronic structure calculations with Python-based software. *Procedia Computer Science*. 2011, Jg. 4, S. 17–25. ISSN 1877-0509. Abgerufen unter DOI: https://doi.org/10.1016/j.procs.2011.04.003. Proceedings of the International Conference on Computational Science, ICCS 2011.

[FC88] FRANKOT, R. T.; CHELLAPPA, R. A method for enforcing integrability in shape from shading algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1988, Jg. 10, Nr. 4, S. 439–451. ISSN 0162-8828. Abgerufen unter DOI: 10.1109/34.3909.

[Gri+18] GRISEL, Olivier; VAROQUAUX, Gael; BESSON, Lilian; ESTÈVE, Loïc; ZEILEMAKER, Niels. *Embarrassingly parallel for loops*. 2018. Auch verfügbar unter: https://github.com/joblib/joblib/blob/master/doc/parallel.rst#old-multiprocessing-backend. Abgerufen: 24. Februar 2018.

[GBA13] GUELTON, Serge; BRUNET, Pierrick; AMINI, Mehdi. Compiling Python modules to native parallel modules using Pythran and OpenMP annotations. *Python for High Performance and Scientific Computing*. 2013, Jg. 2013.

[GFB14] GUELTON, Serge; FALCOU, Joël; BRUNET, Pierrick. Exploring the Vectorization of Python Constructs Using Pythran and Boost SIMD. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. Orlando, Florida, USA: ACM, 2014, S. 79–86. WPMVP '14. ISBN 978-1-4503-2653-7. Abgerufen unter DOI: 10.1145/2568058.2568060.

[Gui+11] GUIZAR-SICAIROS, Manuel; NARAYANAN, Suresh; STEIN, Aaron; METZLER, Meredith; SANDY, Alec R.; FIENUP, James R.; EVANS-LUTTERODT, Kenneth. Measurement of hard x-ray lens wavefront aberrations using phase retrieval. *Applied Physics Letters*. 2011, Jg. 98, Nr. 11, S. 111108. Abgerufen unter DOI: 10.1063/1.3558914.

[HF17] HAND, Nick; FENG, Yu. Nbodykit: A Python Toolkit for Cosmology Simulations and Data Analysis on Parallel HPC Systems. In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. Denver, CO, USA: ACM, 2017, 7:1–7:10. PyHPC'17. ISBN 978-1-4503-5124-9. Abgerufen unter DOI: 10.1145/3149869.3149876.

[Ill14] ILLMER, Joachim. *Parallele Python-Programmierung auf Multi-Core-Architekturen und Grafikkarten für numerische Algorithmen aus der Strömungstechnik und den Materialwissenschaften*. 2014. Auch verfügbar unter: http://elib.dlr.de/92043/. Bachelor's. Duale Hochschule Baden-Würtemberg, Mannheim. Betreuer: Dr.-Ing. Achim Basermann (DLR, Simulations- und Softwaretechnik), M. Sc. Melven Röhrig-Zöllner (DLR, Simulations- und Softwaretechnik), Prof. Dr. Harald Kornmayer (Duale Hochschule Baden-Württemberg, Mannheim).

[KE14] KLEMM, Michael; ENKOVAARA, Jussi. pyMIC: A Python offload module for the Intel Xeon Phi coprocessor. *Proceedings of PyHPC*. 2014.

[Knü+12] KNÜPFER, Andreas *et al.* Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In: BRUNST, Holger; MÜLLER, Matthias S.; NAGEL, Wolfgang E.; RESCH, Michael M. (Hrsg.). *Tools for High Performance Computing 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 79–91. ISBN 978-3-642-31476-6.

[Kov04] KOVESI, Peter. *FRANKOTCHELLAPPA*. 2004. Auch verfügbar unter: http://www.peterkovesi.com/matlabfns/Shapelet/frankotchellappa.m. Abgerufen: 12. Januar 2018.

[LPS15] LAM, Siu Kwan; PITROU, Antoine; SEIBERT, Stanley. Numba: A LLVM-based Python JIT Compiler. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Austin, Texas: ACM, 2015, 7:1–7:6. LLVM '15. ISBN 978-1-4503-4005-2. Abgerufen unter DOI: 10.1145/2833157.2833162.

[Lew94] LEWIS, J.P. Fast Template Matching. 1994, Jg. 95.

[Mar17] MARKWARDT, Ulf. *HardwareTaurus - Compendium - Foswiki*. 2017. Auch verfügbar unter: https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus. Abgerufen: 24. Februar 2018.

[Mer+03] MERCÈRE, Pascal *et al.* Hartmann wave-front measurement at 13.4 nm with λEUV/120 accuracy. *Opt. Lett.* 2003, Jg. 28, Nr. 17, S. 1534–1536. Abgerufen unter DOI: 10.1364/OL.28.001534.

[ML16] MORTENSEN, Mikael; LANGTANGEN, Hans Petter. High performance Python for direct numerical simulations of turbulent flows. *Computer Physics Communications*. 2016, Jg. 203, S. 53–65. ISSN 0010-4655. Abgerufen unter DOI: https://doi.org/10.1016/j.cpc.2016.02.005.

[Oli07] OLIPHANT, T. E. Python for Scientific Computing. *Computing in Science Engineering*. 2007, Jg. 9, Nr. 3, S. 10–20. ISSN 1521-9615. Abgerufen unter DOI: 10.1109/MCSE.2007.58.

[PGH11] PEREZ, F.; GRANGER, B. E.; HUNTER, J. D. Python: An Ecosystem for Scientific Computing. *Computing in Science Engineering*. 2011, Jg. 13, Nr. 2, S. 13–21. ISSN 1521-9615. Abgerufen unter DOI: 10.1109/MCSE.2010.119.

[PLA17] PITROU, Antoine; LAM, Siu Kwan; ANDERSON, Todd A. *Just-in-Time compilation*. 2017. Auch verfügbar unter: https://github.com/numba/numba/blob/master/docs/source/reference/jit-compilation.rst. Abgerufen: 24. Februar 2018.

[Pro17] PROJECT, The Open MPI. *MPI_Init_thread(3) man page (version 2.0.4)*. 2017. Auch verfügbar unter: https://www.open-mpi.org/doc/v2.0/man3/MPI_Init_thread.3.php. Abgerufen: 24. Februar 2018.

[Sch18a] SCHENKE, Jonas. *CPU Utilization*. 2018. Auch verfügbar unter: https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/blob/f2fc5c2e5f8b4ef0e9b3ca3a4e770db67f230588/doc/cpu_util.md. (Privat).

[Sch18b] SCHENKE, Jonas. *Loaded Libraries*. 2018. Auch verfügbar unter: https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/blob/cb7fd24ea64ad0fef0af93ba0dfb9f04f6487382/doc/loaded_libs.txt. (Privat).

[Seh+17] SEHRISH, S.; KOWALKOWSKI, J.; PATERNO, M.; GREEN, C. Python and HPC for High Energy Physics Data Analyses. In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. Denver, CO, USA: ACM, 2017, 8:1–8:8. PyHPC'17. ISBN 978-1-4503-5124-9. Abgerufen unter DOI: 10.1145/3149869.3149877.

[SA17] SHABUNIN, Maksim; ALEKHIN, Alexander. *Template Matching*. 2017. Auch verfügbar unter: https://github.com/opencv/opencv/blob/master/doc/py_tutorials/py_imgproc/py_template_matching/py_template_matching.markdown. Abgerufen: 21. Januar 2018.

[WCV11] WALT, Stéfan van der; CHRIS COLBERT, S; VAROQUAUX, Gael. The NumPy Array: A Structure for Efficient Numerical Computation. 2011, Jg. 13, S. 22–30.

[Wei+05] WEITKAMP, Timm; NÖHAMMER, Bernd; DIAZ, Ana; DAVID, Christian; ZIEGLER, Eric. X-ray wavefront analysis and optics characterization with a grating interferometer. *Applied Physics Letters*. 2005, Jg. 86, Nr. 5, S. 054101. Abgerufen unter DOI: 10.1063/1.1857066.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Erklärungen zum Urheberrecht

Zur Implementierung des Wellenfrontrekonstruktionsalgorithmus wurden viele Open Source-Bibliotheken verwendet. Folgende Bibliotheken wurden hier besonders intensiv genutzt:

- Cython (Apache Lizenz[1])
- EdfFile.py (GPL Version 2[2])
- joblib (BSD[3])
- matplotlib (matplotlib Lizenz[4])
- mpi4py (BSD[3])
- numba (BSD[3])
- NumPy (BSD 2.0[5])
- OpenCV (BSD[3])
- Pillow (Standard PIL Lizenz[6])
- Python (Python Software Foundation License[7])
- SciPy (SciPy Lizenz[8])

[1] https://www.apache.org/licenses/LICENSE-2.0
[2] https://www.gnu.org/licenses/gpl-2.0.html
[3] https://opensource.org/licenses/BSD-2-Clause
[4] https://matplotlib.org/users/license.html
[5] https://opensource.org/licenses/BSD-3-Clause
[6] http://www.pythonware.com/products/pil/license.htm
[7] https://www.python.org/download/releases/2.7/license/
[8] https://scipy.org/scipylib/license.html