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 ist es, einen bereits in Python implementierten Wellenfrontrekonstruktionsalgorithmus zu beschleunigen. Dieser berechnet aus zwei Bildern eines zu untersuchenden Objekts pixelweise die 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 Wellefront 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 sollen Optimierungen und Parallelisierungsmöglichkeiten für die kritischen Programmpfade ermittelt, implementiert und evaluiert werden.

Abstract

TODO

Inhaltsverzeichnis

Glossar

N_{Paare} Anzahl der Bildaare.

N_{corr} Anzahl Korrekturversuche.

N Anzahl der Bilder.

P_u Unterabtastperiode.

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.

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

In dieser Arbeit wird ein bereits in Python implementierter Wellenfrontrekonstruktionsalgorithmus für Manycore-Prozessoren beschrieben. Die Wellenfront wird aus zwei Bildern ein und desselben Targets rekonstruiert, die von zwei CCD-Sensoren aus unterschiedlicher Entfernung aufgenommen werden. Aus den Unterschieden der Bilder lässt sich der Verlauf der Wellenfornten herleiten, welcher wiederum Aufschluss über die dreidimensionale Struktur des Targets gibt.
In seiner momentanen Implementierung benötigt der Algorithmus 25 Sekunden pro Bildpaar **nachprüfen**. Da die CCD-Sensoren mit bis zu 25 Bildern pro Sekunde erhebliche Datenmengen erzeugen, ist es wünschenswert diese Daten in-time zu verarbeiten, sodass eine Speicherung der Daten entfällt.
Soll-Kriterien, die hierbei an die parallele Lösung gestellt werden, sind zuallererst die Korrektheit der parallelen Implementierung und ein erheblicher Speedup **von wie viel?**, wobei eine echtzeitfähige Implementierung angestrebt wird.

2 Evaluierung des Algorithmus

2.1 Verwandte Arbeiten

2.1.1 Wellenfrontrekonstruktionsalgorithmus

Guizar-Sicairos *et al.* zeigten 2011, dass die Messung von Wellenfronten zum Ausrichten von optischen Bauteilen in strahlen physischen Experimenten genutzt.[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] Eine auf einem beweglichen Sensor basierende Methode wurde 2012 von Bérujon *et al.* vorgestellt. [Bér+12] Auf dieser Methode basiert der hier behandelte Wellenfrontrekonstruktionsalgorithmus.

2.1.2 Python-Optimierungen

Es wurde von Oliphant und Perez *et al.* aufgezeigt, dass sich 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, indem den für Cython optimierten Python Code nach C/C++ übersetzt und diesen kompiliert. [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 [DPS05] vorgestellt und 2008 von Dalcín *et al.* auf MPI-2 erweitert. Die Verteilung der Daten auf verschiedene Rechenknoten kann erheblichen Einfluss auf die Leistung des Gesamtprogrammes haben. Hierzu verglichen Dalcin *et al.* Python-internen Pickle/cPickle Methode 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 man Python-Code effizient auf der Intel®Xeon Phi™Coprozessoren ausführen 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 in den Wellenfrontrekonstruktionsalgorithmus

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 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-Charge-coupled device (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 dem Papier "X-ray pulse wavefront metrology using speckle tracking" von Bérujon *et al.* entnommen wurde. [BZC15]
Die Röntgenstrahlen werde zuerst am zu untersuchenden Objekt absorbiert und gebrochen und treffen anschließend auf die Membran, wodurch das Röntgenstrahlbündel hinter dieser Membran das Fleckenmuster dieser 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 gebrochen und dort aufgenommen. Der Röntgen-Teil wird zum Szintillator des zweiten CCD-Sensors gebrochen, wo dieser in sichtbares Licht gewandelt wird, das mittels der 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 (parallel 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.

2.2.2 Kalibrierung der CCD-Sensoren

Während der Kalibrierung, gemäß Bérujon, findet zuerst eine Parameterinitialisierung statt auf welche der Kamerafehler der CCD-Sensoren in drei Phasen bestimmt wird ([Bér13], S. 76 ff., S. 194). Dazu wird, wie in Abbildung 2.2 (angefertigt gemäß DIN 66001 bzw. ISO 5807) gezeigt, zuerst der Median aus einer festen Anzahl dunkler Bilder aufgenommen. Anschließend findet eine Nullfeldkalibrierung bei ungestörter Wellenfront und zum Schluss eine Erkennung von Streueffekten im Detektor statt. Um letzteres zu erreichen werden für jeden CCD-Sensor die Anzahl der Bilder N^2 Bilder unter Bewegung der CCD-Sensoren aufgenommen, wobei N die hier gleich der Anzahl der Verschiebungen in X- und

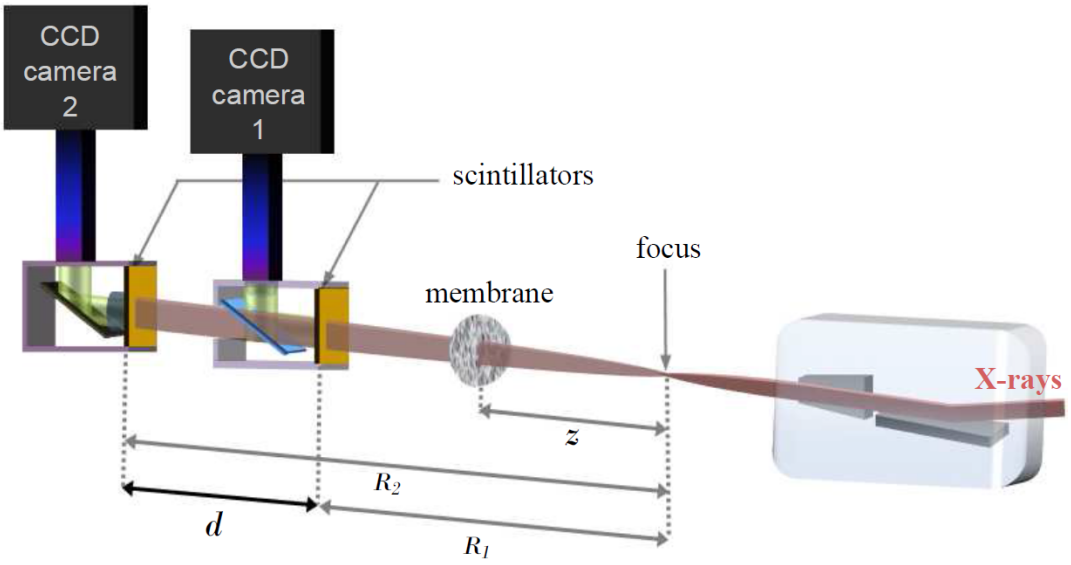


Abbildung 2.1: Versuchsaufbau ([BZC15], S. 891)

Y-Richtung ist. Dadurch lässt sich die horizontale und vertikale Streuung von den einzelnen Sensoren mittels des Speckle-Trackings bestimmen. Da die Kalibrierung lediglich am Anfang des Experiments ein mal durchgeführt werden muss, wird diese im weiteren Verlauf der Arbeit vernachlässigt.

Speckle-Tracking Ziel des Speckle-Trackings ist es, die Flecken (englisch: speckle) im Schatten zwischen den CCD-Sensoren nachzuverfolgen, welche von der Flecken Membran geworfen werden. Der von Bérujon beschriebene Speckle-Tracking-Algorithmus (dargestellt in Abbildung 2.3a) 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 2.3b) 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 weiteren 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 2.3c) 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, von Variablen drei ab [Coj17]: der Unterabtastperiode P_u, der Korrellationsgröße R_{corr} und der Gitterauflösung R_{grid}. Mittels der Unterabtastperiode lassen sich jeweils P_u Pixel überspringen, wodurch die effektive Auflösung Auflösung der Region von Interesse R_{ROI} des Bildes auf Auflösung $R/{P_u}^2$ 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 ermittelten Teilbilder wird dann mittels des Template-Matching-Prozesses ermittelt und auf Sub-Pixel-Genauigkeit interpoliert. Teilbilder, die im

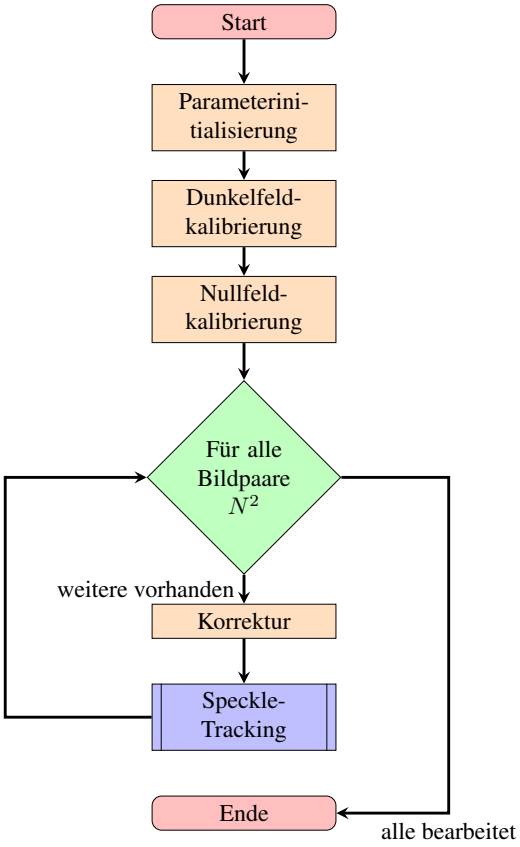
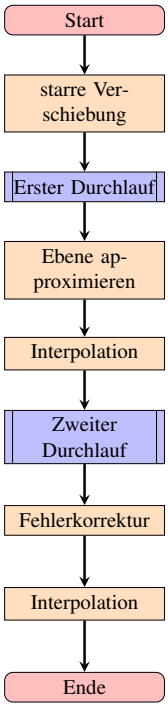
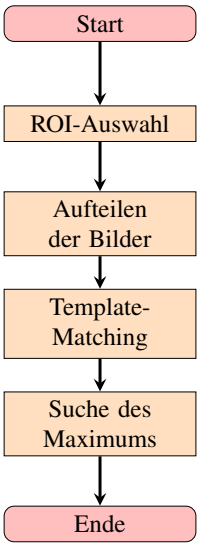


Abbildung 2.2: Programmablaufplan der Kalibrierung (nach ([Bér13], S. 85 ff., S. 194) und [Coj17])

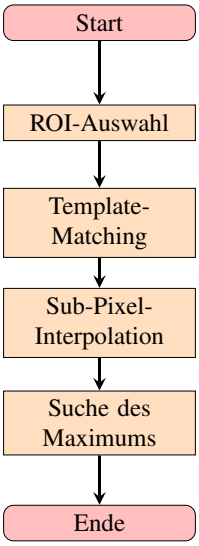
zweiten Durchlauf nicht korrekt zugeordnet werden konnten, werden gesammelt und unter Verwendung von anderen Korrelationsgrößen erneut verarbeitet. Sollten diese Korrelation ebenfalls fehlschlagen, werden die Ergebnisse der fehlerhaften Teilbilder interpoliert.



(a) Speckle-Tracking-Algorithmus (nach ([Bér13], S. 85 ff., S. 194) und [Coj17])



(b) Erster Durchlauf (nach [Bér+12] und [Coj17])



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

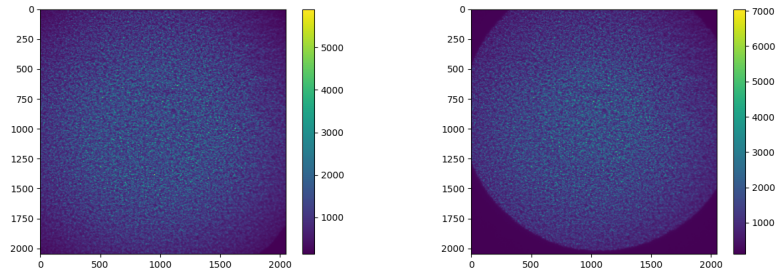
Abbildung 2.3: Programmablaufpläne der Speckle-Tracking-Subroutinen

2.2.3 Verarbeitungsroutine der Bilder

Die von Bérujon *et al.* beschriebene Hauptverarbeitungsroutine (siehe Abbildung 2.7a) 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 2.4 gezeigt. Innerhalb der Hauptschleife gibt es auch hier die Korrektur der Sensorfehler und das anschließende Speckle-Tracking, dessen Ergebnisse in Abbildung 2.5 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 2.7b), was effizient im Frequenzraum möglich ist [FC88]. Die zu in Abbildung 2.4 gezeigten Eingabebildern zugehörige Phasenmatrix ist in Abbildung 2.6 dargestellt.

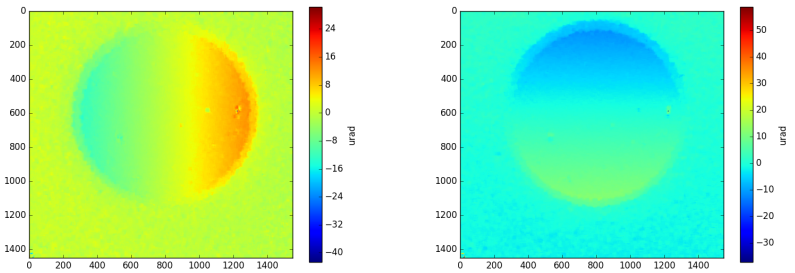
2.3 Überblick über den bereits bestehenden Code

Der bestehende Code wurde von Elena-Ruxandra Cojocaru entwickelt und ist auf GitHub verfügbar [Coj17]. Der Code ist in vier Dateien aufgeteilt. Die Datei *norm_xcorr.py* enthält 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



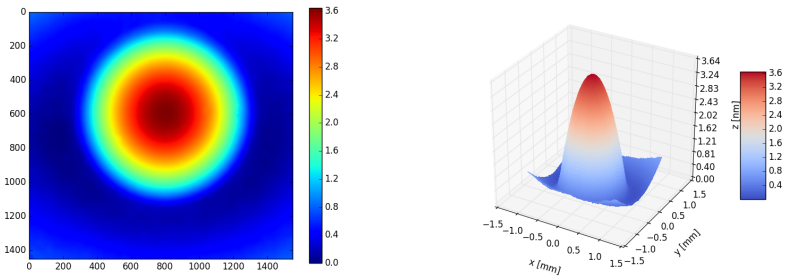
(a) Eingabebild des ersten Sensors (b) Eingabebild des zweiten Sensors

Abbildung 2.4: Eingabebilder



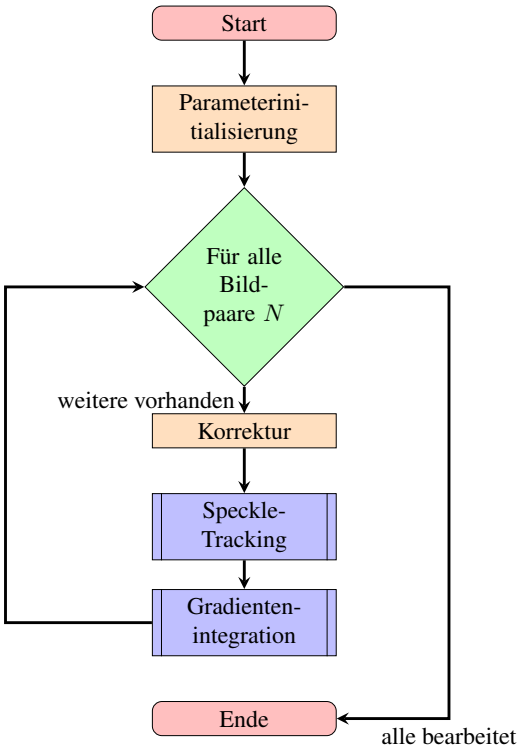
(a) Horizontaler Gradient (b) Vertikaler Gradient

Abbildung 2.5: Gradientenbilder

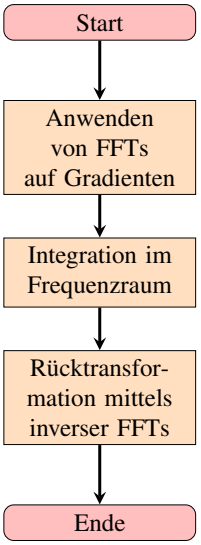


(a) 2D Repräsentation (b) 3D Repräsentation

Abbildung 2.6: Ausgabe des Algorithmus



(a) Hauptroutine (nach ([Bér13], S. 194) und [Coj17])



(b) Integration der Gradienten gemäß Frankot *et al.* (nach [FC88] und [Kov04])

Abbildung 2.7: Programmablaufpläne der Hauptroutine

der Kalibrierung befindet sich in der *detectorDistortion.py* Datei und der der Hauptroutine befindet sich in der Datei *wavefront.py*.

2.3.1 norm_xcorr.py

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-Matching 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 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 Sub-Pixel-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.

2.3.2 func.py

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_chelappa* 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 addiert und wieder zurücktransformiert.

2.3.3 wavefront.py

Das letzte Skript, *wavefront.py*, beinhaltet den eigentlichen Rekonstruktionsalgorithmus, welcher sich besonders in der Initialisierungsphase sehr stark dem *detectorDistortion.py*-Skript ähnelt. Die Hauptroutine wird, wie in Abbildung 2.7a veranschaulicht, jetzt nur ein mal für jedes Bildpaar statt für jede Kombination aus Eingabebildern ausgeführt. Innerhalb dieser Routine wird das entsprechende Bildpaar samt Motorpositionen geladen und mittels der *correction* - Funktion korrigiert. Da die Pixelgröße zwischen den Sensoren unterschiedlich sein kann, wird jetzt 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.

2.4 Die wichtigsten Funktionen

2.4.1 norm_xcorr

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 jeden 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.

2.4.2 nxcorr_disp

Aufgabe dieser Funktion ist es, den Punkt der maximalen Übereinstimmung auf ein Sub-Pixel-Level zu verfeinern. Dazu wird der Gradient der Pixelwerte in der Nachbarschaft des Maximums gebildet und anschließen interpoliert. Im Gesamtalgorithmus wird dieser Schritt direkt nach dem Template-Matching als Teil des Speckle-Trackings ausgeführt.

2.4.3 frankot_chelappa

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 dort durch das Tracking ermittelten Gradientenmatrizen um die Wellenfront dreidimensional zu rekonstruieren.

3 Performance-Analyse der derzeitigen Implementierung

3.1 Komplexität

3.1.1 Speckle-Tracking

Der erste Schritt des Speckle-Trackings ist die Feststellung der starren Verschiebung. Werden hierfür feste Werte angenommen, ist diese Komplexität konstant. Wird allerdings ein Korrelationsverfahren verwendet, so ist die Komplexität $\mathcal{O}(R \cdot log(R))$ 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, also auf die Auflösung der Region von Interesse R_{ROI} verkleinert und anschließend in Blöcke mit konstanter Größe aufgeteilt, welche dann ineinander mittels des Template-Matchings gesucht werden. Die Komplexität der einzelnen Suchvorgänge kann somit auch als konstant angesehen werden. Der Durchlauf braucht demzufolge $\mathcal{O}(R_{ROI})$ Schritte. Die anschließende Interpolation wird auf alle Pixel des Bildes angewandt und hat demzufolge eine Komplexität von $\mathcal{O}(R_{ROI})$.
Der zweite Durchlauf, der als nächstes folgt, werden Subbilder, wie 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 Korrellationsgröße R_{corr} legt nimmt auf das Template-Matching Einfluss, dessen Komplexität nun bei $\mathcal{O}(R_{corr} \cdot log(R_{corr}))$ 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:

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

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

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

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

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

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. 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 Hauptroutine

Die Hauptroutine beginnt mit einer trivialen Parameterinitialisierung, die als linear angenommen werden kann. Auf diese folgt die Hauptschleife, welche für die Anzahl der Bildaare N_{Paare} jeweils ein mal 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 und somit auch der gesamten Hauptroutine festlegt auf:

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

Dies liegt insbesondere für kleine P_u und R_{grid} in der Komplexitätsklasse:

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

3.2 Benchmark

3.2.1 Testsystem und Laufzeitumstände

Testsystem Alle Benchmarks liefen auf den *haswell*-Partition des Taurus-Supercomputers an der Technischen Universität Dresden. Jeder Knote dieser Partition ist ausgestattet mit zwei Intel®Xeon®E5-2680 v3 Central Processing Units (CPUs). Diese haben zwölf Rechenkerne, die mit bis zu 2.50 Gigahertz (GHz) getaktet sind. MultiThreading war hierbei nicht aktiviert. Die Knoten haben 64 Gibibyte (GiB) (*haswell64*), 128 GiB (*haswell128*) oder 256 GiB (*haswell256*) Arbeitsspeicher zur Verfügung. 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[1] finden.

Laufzeitumstände Jede Konfiguration, bestehend aus Datensatz und Kernanzahl, wurde nach vier Aufwärmiterationen fünf mal 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.

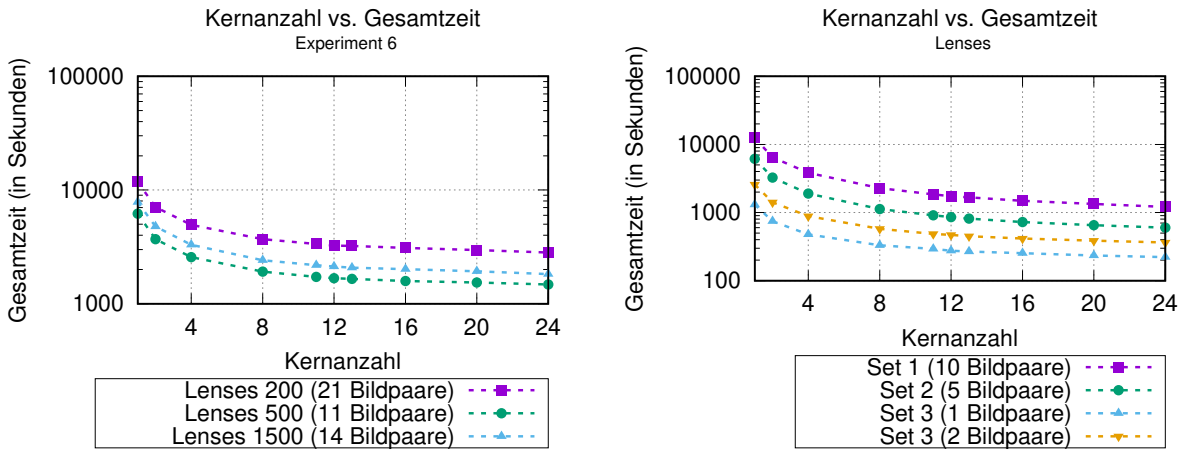
Datensätze Zur Leistungsfeststellung der vorliegenden Implementierung werden drei verschiedene Arten von Datensätzen verwendet: *detectorDistortion*, *Experiment 6*, *Lenses*. *detectorDistortion* wird zur Kalibrierung verwendet. Die Eigenschaften dieser Typen werden in Tabelle 3.1 gegenüber gestellt. Von diesem Datensatztyp gibt es einen Datensatz mit 25 Bildern. Es existieren drei *Experiment 6* Datensätze mit 21 (*Experiment 6 Lenses 200*), 11 (*Experiment 6 Lenses 500*) und 14 Bildpaaren (*Experiment 6 Lenses 1500*). Vom letzten Typ existieren vier Datensätze mit jeweils zehn (*Lenses Set 1*), fünf (*Lenses Set 2*), zwei (*Lenses Set 3*) und einem Bildpaar.

3.2.2 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.

	Hauptroutine	
	Experiment 6	Lenses
R_{ROI} (in Pixel)	Sensor 1: 550 x 550 Sensor 2:1450 x 1450	1450 x 1550
R_{grid}	1	1
R_{corr}	91	41
P_u	1	1
Pixelgröße	unterschiedlich	gleich

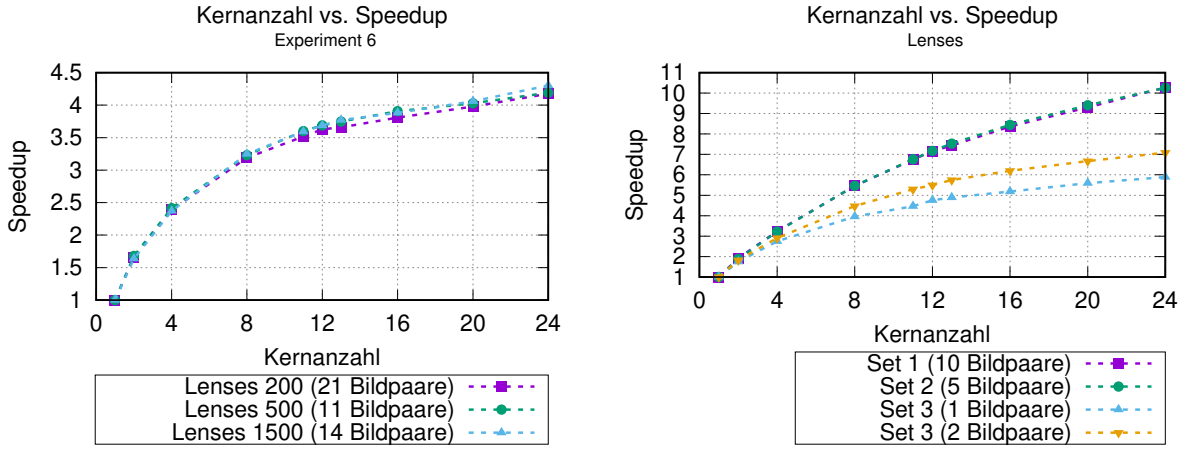
Tabelle 3.1: Parameter der Datensätze



(a) Experiment 6

(b) Lenses

Abbildung 3.1: Gesamtlaufzeiten



(a) Experiment 6

(b) Lenses

Abbildung 3.2: Speedup

Der Speedup des Programmes skaliert mit der Anzahl der Prozessorkerne nicht linear und flacht schnell ab. Der Speedup-Faktor für den *Experiment 6* Datensätze übersteigt fünf nicht. 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. Ein Überblick über das Gesamtprogramm ist in Abbildung 3.3 zu sehen.

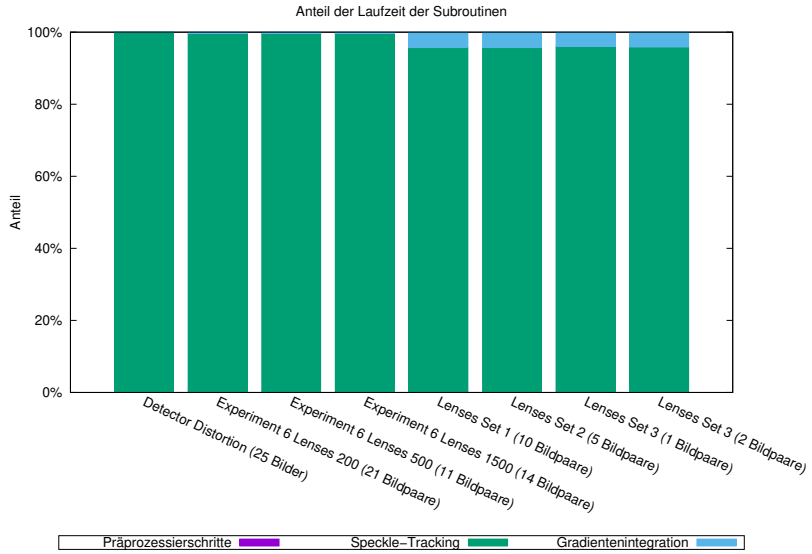


Abbildung 3.3: Anteile der Laufzeiten

[1] https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/blob/f2fc5c2e5f8b4ef0e9b3ca3a4e770db67f230588/doc/loaded_libs.txt

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 in Abbildung 3.4, dass hierbei der zweite Durchlauf am meisten Zeit benötigt.

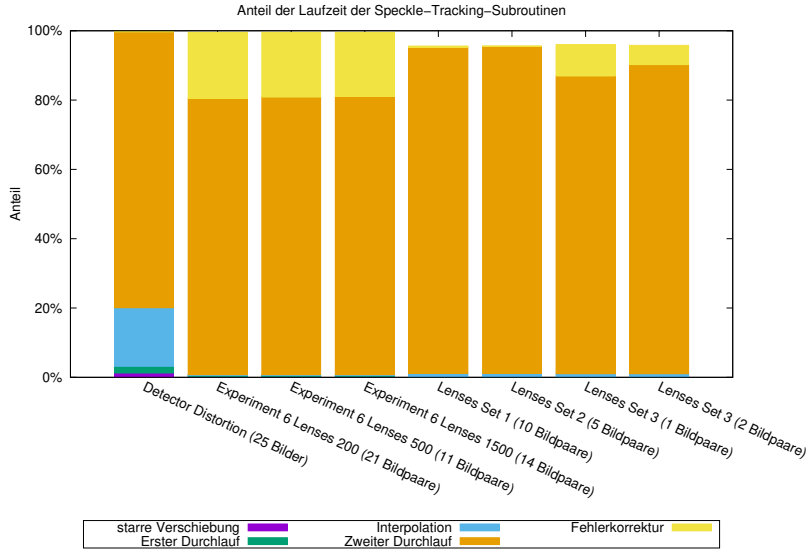


Abbildung 3.4: Anteile der Laufzeiten des Speckle-Tracking-Algorithmus

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

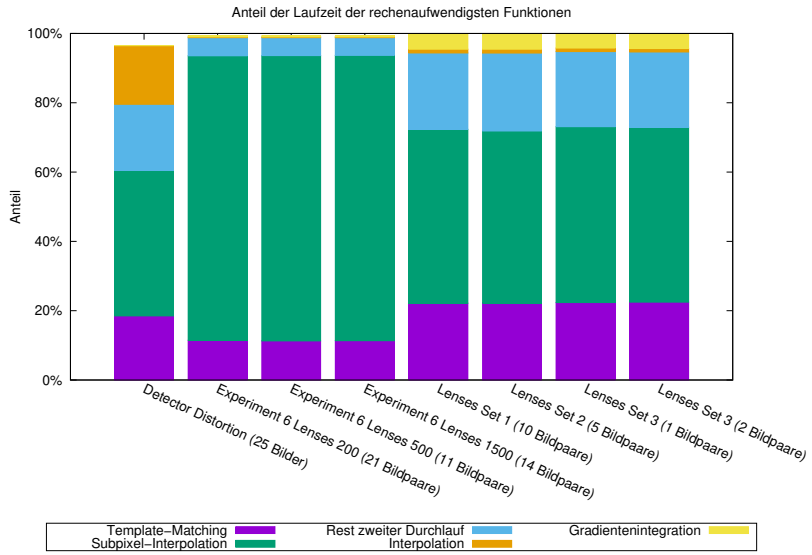


Abbildung 3.5: Anteile der Laufzeiten der langsamsten Funktionen

3.3 Grund der 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 ein mal 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 Pyhton-Overhead hinzukommt.
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, was zu einem erheblichen Overhead führt[2].
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% [3], wodurch häufig einige Kerne nicht oder nur wenig genutzt werden.

[2] `https://pythonhosted.org/joblib/parallel.html`
[3] `https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/blob/f2fc5c2e5f8b4ef0e9b3ca3a4e770db67f230588/doc/cpu_util.md`

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 einige Nachteile mit sich: Message Passing Interface (MPI)-Implementierungen erlauben das Erstellen neuer Threads nur, wenn man dies beim Installieren der Bibliotheken angibtThreads Nachweiß einfügen und es ist nur eine schwache Skalierung zu erwarten. Sofern kein Multithreading innerhalb von MPI möglich ist, limitiert die Anzahl der Bildpaare die Parallelisierungsmöglichkeiten stark, weshalb hier aus Zeitgründen auf intensives Testen dieser Version verzichtet wird. Die Implementierung dieses Ansatzes ist auf dem GitHub-Branch *mpi*[1] verfügbar.

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[2]. 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 auch CPU-Kern Gruppen verteilt. Einer dieser Kerne agiert hierbei als Hauptkern und ist dafür verantwortlich das Bildpaar zu verarbeiten, wobei dieser Aufgaben mittels eines MPI-Kommunikators an die anderen Rechenkerne verteilen kann. Dies ist in der Abbildung Abbildung einfügen gezeigt. Die Schnittstelle wurde hierbei ähnlich zur joblib-Implementierung entworfen. Sollten mehr Bildpaare als Rechenkerne vorhanden sein, werde mehrere Bildpaare von eine Kern hintereinander verarbeitet. Die Programmierschnittstelle wurde so entworfen, dass die Verteilung der Bildpaare auf die Kernen und das Parallelisieren innerhalb dieser für den Programmierer transparent geschieht. Die Implementierung dieses Ansatzes ist auf GitHub auf dem *mpi-advanced*-Branch[3] zu finden.
Schemata einfügen

4.2 Optimierung der Python-Engpässe

Wie in Abbildung 3.5 zu sehen ist, wird über 95% der Rechenzeit in den fünf langsamsten Funktionen verbracht. Unter diesen ist insbesondere die *nxcorr_disp*-Funktion, welche komplett in reinem Python

[1] `https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/tree/mpi`
[2] `http://www.vi-hps.org/projects/score-p/`
[3] `https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/tree/mpi-advanced`

implementiert ist. Um die Leistung solcher Funktionen zu verbessern werden im folgenden Methoden zur Optimierung des bestehenden Codes betrachtet mit besonderem Augenmerk auf der Optimierung der langsamsten Funktionen.

4.2.1 Nutzen von 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 sind diese Funktionen bereits für optimale Leistung optimiert. In der Funktion *nxcorr_disp* lassen sich solche Code-Abschnitte finden. Die Code-Auflistung 4.1 zeigt den Code zum Ermitteln eines Maximums einer Matrix in reinem Python-Code, wobei die äußeren Zeilen und Spalten vernachlässigt werden. Auflistung 4.2 zeigt einen funktional äquivalenten Code unter Nutzung von bereits optimierten Funktionen.

Listing 4.1: 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 4.2: 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 in Auflistung 4.3), was allerdings im weiteren Verlauf des Programmes nicht wieder verwendet wird.

Listing 4.3: 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
```

Nachdem diesen Änderungen befindet sich keine in Python implementierte Schleife mehr in der Funktion. Angesichts der hohen Aufrufzahl der *nxcorr_disp* und dem Entfernen großer Codeanteile ist ein hoher Beschleunigungsfaktor zu erwarten. Die hier angegebenen Verbesserungen sind auf dem Branch *intrinsics* des GitHub-Repositorys[4] zu fintrinsicsnden.

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 Pro-

[4] https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/tree/intrinsics

grammes über die Übersetzung einzelner Funktionen, die in Python dann als Modul geladen werden können, bis hin zur Nutzung eines just-in-time 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[5] 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*[6] verfügbar.

Einzelne Funktionen

numba numba[7] ist eine Optimierungsbibliothek für Python, welche unter anderem die Möglichkeit bietet, Funktionen just-in-time (JIT) zu übersetzen und CUDA und OpenCL zu nutzen.
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*-Branch[8] befindet sich eine Version des Codes, in dem diese Annotationen genutzt werden.

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*[9] verfügbaren Version wurden Cython genutzt, um die Funktionen *norm_xcorr* und *nxcorr_disp* zu übersetzen. Hierbei wurden alle Variablen mit Typen versehen und es wurde die von Cython bereitgestellte Schnittstelle zu NumPy genutzt. Zusätzlich wurden ebenfalls Cython-Annotationen verwendet, die NumPy's Grenzfallbehandlung abschalten.

[5] `http://cython.org/`
[6] `https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/tree/compiled`
[7] `https://numba.pydata.org/`
[8] `https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/tree/numba`
[9] `https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/tree/cython`

5 Performance-Messungen der parallelen Implementation

5.1 Evaluierung der Optimierungen

5.1.1 Parallelisierung

- Speedup: 2-5x (geschätzt; ohne mehr nodes) - linear mit Anzahl der Nodes bis Anzahl der Bildpaare

5.1.2 Optimierung von Python Engpässen

Nutzen bereits optimierter Funktionen

- Speedup: 2.5x

Kompilieren

Gesamtes Programm - kompilieren des kompletten Projektes mit Cython - funktioniert, aber Ergebnisse bringen nicht gewünschten Speedup bzw. nur manchmal –> Ansatz verworfen

numba - kaum speedup - cpython compiler besser, da er direkt für python gemacht wurde - bereits viel durch intrinsics optimiert

Cython - ein wenig besser als intrinsics, da typinformationen fest einprogrammiert sind

5.2 Einfluss der Parameter

- Algorithmus in verschiedenen Konfigurationen benchmarken (Anzahl der Nodes, verschiedene Datensets, ...) - Performance Sweet-Spot finden

5.3 Skalierung

5.3.1 Skalierungsfaktor

- auf Skalierungsfaktor eingehen und diesen in Bezug zu Parametern und verwendeter Hardware setzen

5.3.2 Sättigung

- Sättigungspunkt -grund beschreiben - Sättigungsgrund: -> Framepaketgröße: Rechenaufwand » Kopieraufwand -> keine weitere Optimierung des Kopieraufwandes möglich -> Anzahl der Nodes: Kopieraufwand übersteigt Rechenaufwand bzw. nähert sich an -> nur begrenzte Menge an Datenverfügbar

6 Analyse sowie Validierung der Ergebnisse

6.1 Speedup

- Ergebnisse werten (im Vergleich zu seriell) -> parallele Implementierung deutlich schneller dank hoher Auslastung der Nodes und Aufteilen der Frames in Framepakete, welche parallel verarbeitet werden

6.2 Wertung des Ergebnisses

- Analyse des erreichten Ergebnisses mit Rückblick auf die Möglichkeiten - Analyse des noch nicht genutzten Potentials mit Rückblick auf die Möglichkeiten

6.3 Verbesserungsmöglichkeiten

-> Anpassung des template-matching Prozesses (nur Punkt mit größter Übereinstimmung suchen mittels Runterskalierung) -> Parallelisierung von detectorDistorion.py -> Nutzen von FFTW mit Wisdoms für Frankot_chellappa -> Implementierung der rechenaufwendigen Funktionen für GPGPUs -> SIMD-Verarbeitung der Bildpaare

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.

[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.

[Coj17] COJOCARU, Elena-Ruxandra. *Wavefront-Sensor*. 2017. Auch verfügbar unter: https://github.com/ComputationalRadiationPhysics/Wavefront-Sensor/. Abgerufen: 20. Januar 2018.

[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.

[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.

[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.

[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.

[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.

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