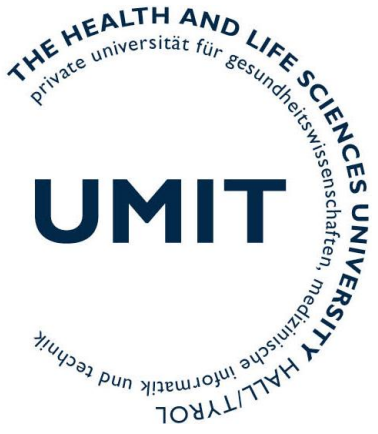

Algorithmische Optimierungen für iterative Dekonvolutionsverfahren

Bachelorarbeit
zur Erlangung des akademischen Grades

Bachelor of Science (BSc.)

im Rahmen des
Bachelorstudiums
Biomedizinische Informatik

vorgelegt von:
Ing. Martin Erler

betreut von:
a.o. Univ.-Prof. Dr. Martin Welk

an der:
UMIT - Private Universität für Gesundheitswissenschaften,
Medizinische Informatik und Technik

Inhaltsverzeichnis

1 Einleitung und Stand der Forschung

1.1 Dekonvolution in der Bildverarbeitung

Dekonvolutionsverfahren haben das Ziel die Faltungsoperation (Konvolution) umzukehren. Eine solche Faltungsoperation erfolgt implizit bei der Aufnahme eines Bildes. Jedes Mal wenn wir ein Foto aufnehmen, kann diese Bildaufnahme mathematisch als Konvolution modelliert werden. Diese Faltung kann auch mit einer ungewünschten Funktion ablaufen. Dann entsteht ein unscharfes Bild. Die Dekonvolution verfolgt das Ziel, diese Faltung mit der sog. Punktbildfunktion rückgängig zu machen. Wenn die Parameter bekannt sind, kann man also aus dem unscharfen Bild ein scharfes berechnen.

Unscharfe Bilder können auf verschiedene Weise entstehen: Wenn die Belichtungszeit im Verhältnis zur Objektbewegung zu lange dauert, entsteht Bewegungsunschärfe. Wir kennen das Problem, wenn wir im Dunkeln fotografieren und das Bild „verwackelt“ abgelichtet wird. Es kann ebenso passieren, dass die Entfernungseinstellung falsch vorgenommen wurde. Die Fokussierung klappt also nicht und das Ergebnis ist, dass das gewünschte Objekt unscharf dargestellt wird. Die wichtigsten Systemparameter sind bei der Bildaufnahme die Belichtungszeit, die Sensor- oder Filmempfindlichkeit, die Brennweite, der Brennpunkt oder Fokus und die Geschwindigkeit der abzubildenden Objekte.

Je genauer diese Parameter bekannt sind, desto besser läßt sich die Faltung durch die Dekonvolution rückgängig machen. Man unterscheidet verschiedene Typen der Dekonvolution. Eine wichtige Unterscheidung ist, ob und wie genau die Punktbildfunktion bekannt ist. Die blinde Dekonvolution hat das Ziel, die Faltung ohne genaue Kenntnis der Punktbildfunktion rückgängig zu machen. Die Art und das Ausmaß der Unschärfe sind also unbekannt. Eine weitere Unterscheidung der Verfahren bringt die Ortsvarianz der Punktbildfunktion. Die Unschärfe kann über das gesamte Bild gleich sein, oder auch nur einzelne Bildbereiche betreffen. Bei einer über den gesamten Bildbereich gleichbleibenden Unschärfe spricht man von einer ortsinvarianten Faltung, bzw. Dekonvolution. Bei einer ortsabhängigen Unschärfe, wenn also nur einzelne Bereiche oder Objekte unscharf dargestellt werden, muss die ortsvariante Dekonvolution angewendet werden.

Dekonvolutionsverfahren werden meist schrittweise, also iterativ durchgeführt. Es gibt aber auch nicht iterative Dekonvolutionsalgorithmen. Bei der Iteration wird die Schärfe dann Schritt für Schritt entfernt oder reduziert. Wenn der Algorithmus über Regularisierungsparameter verfügt, kann der Grad dieser Abminderung parametriert werden. Das macht dann Sinn, wenn ungewünschte Artefakte eine beliebig starke Schärfung verhindern.

1.2 Modellierung von Bildern und Unschärfe

1.2.1 Bildmodellierung

Bild als Funktion. Ein Bild läßt sich als Funktion beschreiben:

$$f : \Omega \to R \tag{1}$$

Dabei stellt Ω den Bildbereich oder die Bildebene dar. R beschreibt den Wertebereich. Wir beschränken uns auf Grauwertbilder.

Faltung. Wir führen nun Faltungsoperator ein. Dabei repräsentiert $f(x)$ das gefaltete Bild:

$$f(x) = \int_{\Omega} H(x, y) \cdot g(y) dy + n(x) \tag{2}$$

Als $g(x)$ bezeichnen wir das gedachte ideale Bild. Und $H(x, y)$ bezeichnen wir als Faltungskern, oder Punktbildfunktion (engl.: Point Spread Function, PSF). Sie beschreibt die Art und Weise, wie das ursprüngliche (gedachte) Bild verwischt worden ist. $n(x)$ beschreibt hier den Einfluss von Rauschen bei der Bildaufnahme. Wenn wir die Operation invertieren können, ist es also möglich mit Kenntnis der PSF auf das gedachte ideale Bild ohne Unschärfe zu gelangen. Da wir aber $n(x)$ nicht kennen, sprechen wir in der Folge von einem schlecht gestellten inversen Problem. Wenn die PSF über Ω gleich bleibt, liegt eine ortsunabhängige Faltung vor und wir erhalten (3).

$$f(x) = (g * h)(x) + n(x) \tag{3}$$

- f beaobachtetes, unscharfes Grauwertbild
- g gewünschtes, scharfes Grauwertbild
- n Bildstörungen (Rauschen)
- h ortsunabhängige Punktbildfunktion (PSF)

Diese Modellierung gilt nun auch nach der Abtastung (Sampling). Wir haben ein digitales Bild vorliegen. Dabei ist nun die Bildebene Ω diskretisiert und besteht aus Pixeln. R stellt den diskretisierten Wertebereich dar und wird bei uns auf 8 bit, also einen Bereich von $0 \ldots 255$ beschränkt.

1.2.2 Bewegungsunschärfe

Bewegungsunschärfe entsteht durch Objekt- oder Kamerabewegung während der Bildaufnahme. Wenn diese Bewegung im Verhältnis zur Belichtungszeit zu schnell abläuft, entsteht eine Unschärfe in Bewegungsrichtung.

Ortsvariante, -invariante Bewegungsunschärfe. Die Unschärfe kann ortsabhängig

Abbildung 1: Ausgangsbild

oder auch ortsinvariant sein. Bei der ortsinvarianten Unschärfe werden alle Pixel im Bild gleich verwischt. Ein praktisches Beispiel ist die Bildaufnahme ohne Stativ: Die Kamera bewegt sich während der Bildaufnahme (die Rotation wird ausgeschlossen, oder vernachlässigt). Bei einer ortsvarianten Unschärfe ändert sich der Faltungskern. Unschärfe variiert im Bild. Das kann durch Objektbewegung passiert sein: Beispielsweise wenn ein Auto durch das Bild fährt, kann es unscharf abgebildet werden. Die entstandene Unschärfe betrifft nur das Auto. Die anderen Pixel sind scharf abgebildet. Man spricht dann von einer ortsvarianten PSF.

Konstante oder veränderliche Unschärfe? In dem gezeigten Bild liegt also ein Boxfilter zugrunde. Alle Pixel des Faltungskerns haben denselben (konstanten) Grauwert. Die Kamera- oder Objektbewegung war in diesem Falle konstant. Wenn die Bewegung mit einer veränderlichen Geschwindigkeit v erfolgt, entsteht eine nicht konstante Bewegungsunschärfe.

Boxfilter 1D Der sogenannte eindimensionale Boxfilter kannn besonders effizient berechnet werden. Ein Beispiel eines solchen Filters zeigt die Abb. 2. Er ist konstant (alle Grauwerte haben den selben Wert) und verläuft exakt horizontal oder vertikal. Der Boxfilter wurde bereits im Jahre 1981 von McDonnell beschrieben [9].

1.2.3 Boxfilter 2D

Das Bild 3 zeigt das Ergebnis, bei einer Faltung mit einem zwei-dimensionalen Boxfilter. Alle besetzten Pixel der PSF haben den gleichen Grauwert. Im Gegensatz zu den anderen hier gezeigten Faltungskernen hat der 2D-Boxfilter eine geringere praktische Bedeutung. Die Unschärfe aus Bild 3 kommt physikalisch nicht vor und ist synthetisch erzeugt.

Abbildung 2: 1D Boxfilter mit 17 Pixel Ausdehnung in horizontaler Richtung, rechts unten ist der vergrößerte Faltungskern zu erkennen

Abbildung 3: 2D Boxfilter mit 17 Pixel Unschärfe, rechts daneben der Faltungskern, vergrößert

1.2.4 Lensblur, Defokussierung

Der Lensblurfilter ist von besonderer praktischer Bedeutung: Er generiert ein Bild, das einer fehlerhaften Fokussierung entspricht. Der Kern dieses Filters ist ein gefüllter Kreis. Das Bild 5 illustriert, wie der Radius und die entstehende Unschärfe optisch zusammenhängen. Die Form der PSF rührt von der Form der Blende her. Im Objektiv der Kamera hat die Blendenöffnung eine annähernd kreisrunde Öffnung. Je nach Objektiv kann die Blende aber auch einem Polygon ähnlich sein. Wenn man das optische System eines Objektivs weiter betrachtet, lässt sich festhalten, dass der Brennpunkt idR. über den Bildbereich gleich bleibt. Das gilt jedoch nicht für den Abstand der abgebildeten Objekte. Das hat die Folge, dann ein nahes Objekt scharf abgebildet werden kann, während ein Objekt im Hintergrund unscharf wird. Das Ausmaß dieser Schärfe wird als Tiefenschärfe bezeichnet. Die Tiefenschärfe wird bei geringer Blendenöffnung und Brennweite größer, während Objektive mit großer Brennweite und weiter Blendenöffnung eine große Unschärfe in der Tiefe mit sich bringen [3].

Abbildung 4: Unschärfe durch Defokussierung mit einem Radius von 8 Pixel synthetisch hergestellt, ortsinvariant; rechts: der zugehörige Kern, 5fach vergrößert

1.2.5 Allgemein dünn besetzter Kern

Allgemein dünn besetzte Kerne weisen viele Pixel auf, dessen Grauwerte den Wert Null haben. Die Gleichung 2 zeigt, dass bei der Faltung jeweils die Grauwerte der PSF als Faktor einfließen. Wenn nun viele Pixel der Punktbildfunktion den Grauwert Null besitzen, können sie in der Berechnung übersprungen werden. Die Abbildung 6 zeigt einen solchen Kern mit einem gefalteten Bild.

Solche Faltungskerne wirken anfänglich eher abstrakt, sie können aber praktische Bedeutung haben. Wir stellen uns einen Kern mit Bewegungsunschärfe vor, dessen Hauptrichtung

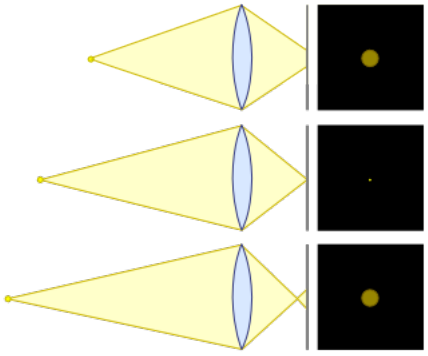

Abbildung 5: Kern und Unschärfe. Illustration aus Wikipedia Commons [1]

diagonal verläuft. Das ist dann ein Kern, der nicht mit dem 1D-Boxfilter berechnet werden kann. Eine weitere Anwendung können Objektive mit kodierter Blende sein. Dabei wird die Form der Blendenöffnung verändert (7). Der Vorteil einer solchen Blendenöffnung liegt darin, dass man die Größe des Kerns leichter bestimmen kann. Die Tiefenschärfe ändert sich ja mit dem Abstand des abgebildeten Objekts. Die Form der Blendenöffnung wird so gewählt, dass sich das gefaltete Ergebnis abhängig von der Größe des Kerns maximal ändert. Einen ähnlichen Ansatz verfolgt die Idee des kodierten Shutters [13]. Hier wird die Belichtung nicht kontinuierlich durchgeführt, sondern in variablen Pulsen. Diese Methode kann besonders bei periodischen Strukturen spezielle Vorteile bringen und die Dekonvolution stark verbessern. Hier gilt dasselbe wie bei der Bewegungsunschärfe. Wenn die Unschärfe in exakter horizontaler oder vertikaler Richtung verläuft kann die übliche Faltung effizient verwendet werden. Wenn die Bewegungsrichtung jedoch diagonal verläuft, kann ein Filter mit dünn besetzten Kernen uU. effizienter sein.

Abbildung 6: Unschärfe durch untypische PSF, rechts der passende Faltungskern (vergrößert), wenig Pixel besetzt

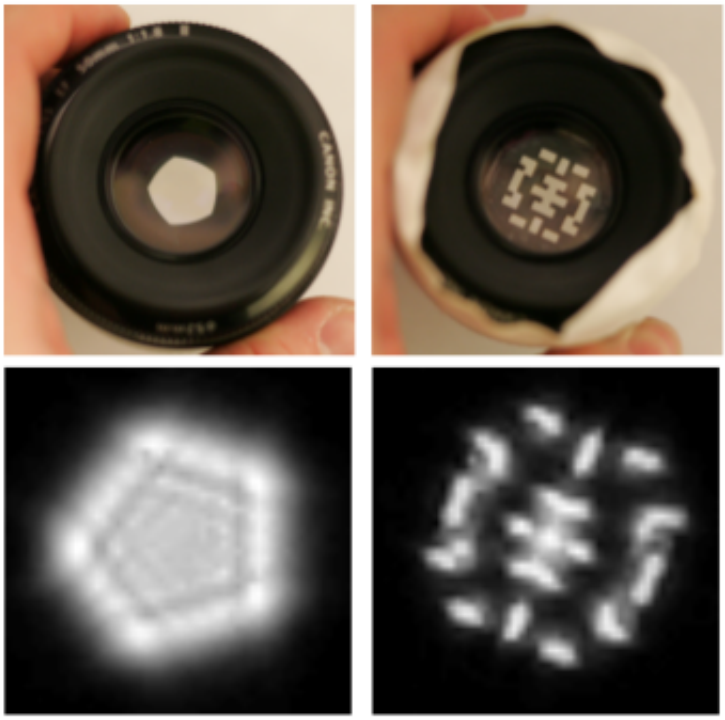

Abbildung 7: coded aperture, entnommen aus [6]

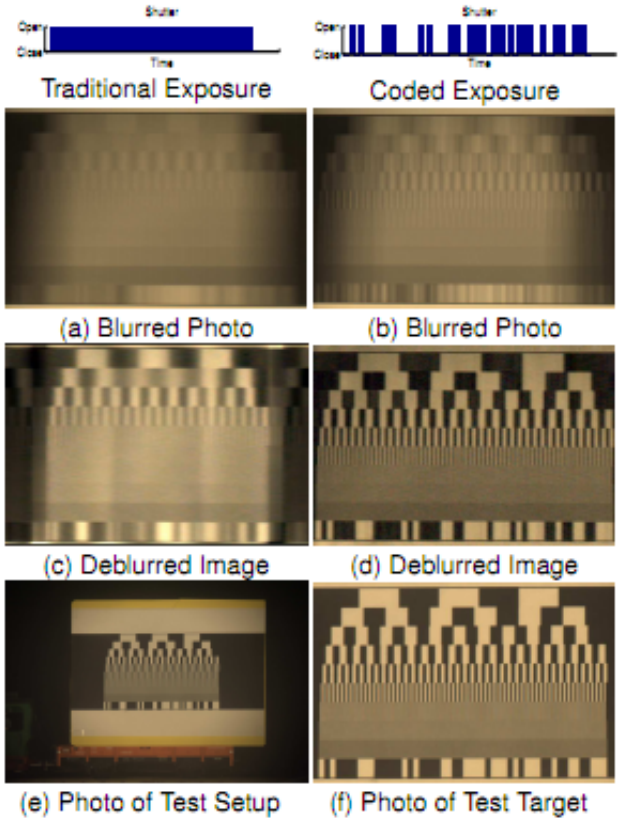

Abbildung 8: coded shutter, entnommen aus [13]

1.3 Übliche Faltungsmethoden

Die Faltung mit den zuvor kennengelernten Faltungskerne werden idR. allgemein mithilfe der Faltung im Orts- oder Fourierbereich durchgeführt. Wir kommen zunächst zur Faltung im Ortsbereich.

Ortsfaltung Die Faltungsfunktion (2) wird im Diskreten zur folgenden Gleichung:

$$(f * h)[n] = \sum_{m=-k}^{k} f[n-m] \cdot h[m] \tag{4}$$

Um ein Pixel des Ausgangsbildes zu erhalten müssen also die umliegenden Pixel des Eingangsbildes mit den Kernpixel multipliziert werden. Das Eingangsbild besitzt N Pixel und der Kern hat $K = 2k$ Pixel. Für die Berechnung des vollen Bildes resultiert daraus eine Laufzeit von $\mathcal{O}(N \cdot K)$. Sie ist also stark von der Kerngröße abhängig. Die Abb.9 schematisiert die Berechnungsvorschrift. Der Einfachheit halber wurden hier die Indizes im gerasterten Bildausschnitt von -4 bis 4 nummeriert.

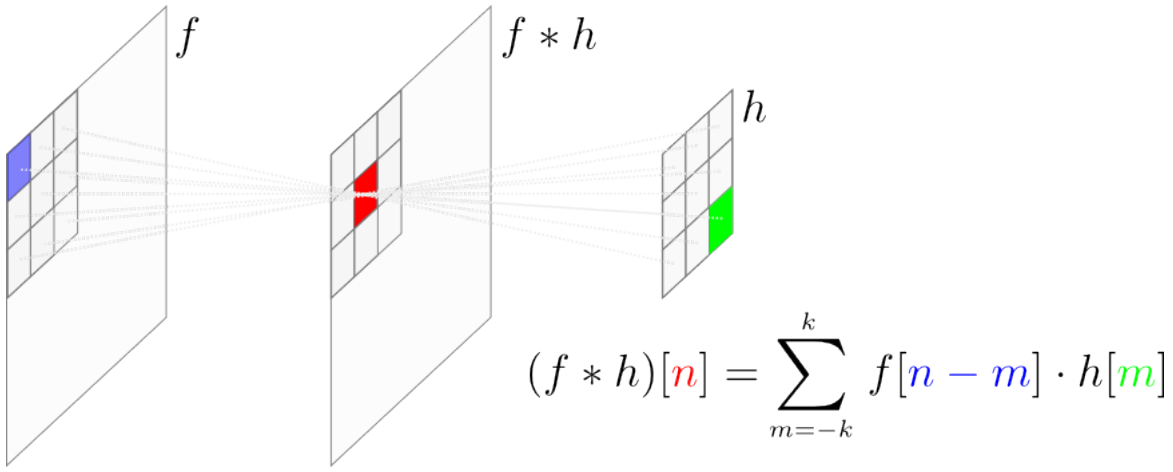

Abbildung 9: Illustration der Faltung im Ortsbereich

Häufig kommt ein punktgespiegelter Kern zur Anwendung: $h^*(x) = h(-x)$. In diesem Fall entspricht dann die Faltung der Korrelation:

$$(f * h^*)[n] = \sum_{m=-k}^{k} f[n+m] \cdot h[m] \tag{5}$$

Faltung im Fourierbereich. Die Faltung im Frequenzbereich macht sich das Faltungstheorem zu Nutze. Siehe dazu (6). Dabei wird aus einer Faltung im Ortsbereich eine Multiplikation im Fourierbereich. Die Laufzeit wird dadurch besonders bei größeren Kernen wesentlich reduziert. Die Laufzeit entspricht hier laut O-Notation dann $\mathcal{O}(N \log_2(N))$, sie ist also von der Kerngröße unabhängig.

$$f(x) * h(x) \rightarrow F(u) \cdot H(u) \tag{6}$$

Randbehandlung. Bei der Faltung im Orts- und Fourierbereich kann es zu Bereichsüberschreitungen bei Arrayzugriffen kommen. Wir betrachten zuerst die Faltung im Ortsbereich. Bereits wenn das erste Pixel des Ausgangsbild berechnet werden soll muss laut Berechnungsvorschrift auf ein Eingangspixel zugegriffen werden, das es gar nicht gibt:

$$(f * h)[0] = \sum_{m=-k}^{k} f[0-m] \cdot h[m] \tag{7}$$

Bei einer Kerngröße von 5 muss das Pixel $f[0-5]$ des Eingangsbildes definiert werden, da es ja nicht existiert. Dazu gibt es verschiedene Möglichkeiten. Man kann dieses Randpixel als 0 annehmen, oder auch mit einem anderen konstanten Wert festlegen. Es bietet sich auch an die Randpixel über den Rand hinaus zu spiegeln. Genauso ist es denkbar, dass man nicht auf das Pixel $f[0-5]$ zugreift, sondern auf das nächstgelegene: $f[0]$.

Nun betrachten wir die Faltung im Frequenzbereich. Die Fouriertransformation impliziert eine periodische Fortsetzung des Signals. Bei der Implementation tritt keine Bereichsüberschreitung bei Arrayzugriffen auf. Deshalb kann die periodische Fortsetzung ohne Implementationsaufwand umgesetzt werden. Durch den Rand entsteht dann jedoch auch ein unnatürlicher Frequenzgang, der dann bei der Dekonvolution zu „Ringing-Artefakten“ führen kann. Wenn diese Artefakte zu stark werden, können mit weiterem Rechenaufwand andere Randbehandlungen implementiert werde, wie zum Beispiel die Spiegelung am Bildrand.

1.4 Dekonvolutionsalgorithmen

Inverse Filterung, Wienerfilter [17]. Mit dem Faltungssatz wurde gezeigt, dass die Faltung im Ortsbereich zu einer Multiplikation im Frequenzbereich wird (6). Naheliegend scheint also diese Multiplikation durch eine Division mit dem bekannten Faltungskern umzukehren: $\hat{u} = \frac{\hat{f}(\omega)}{\hat{h}(\omega)}$. Da aber der Faltungskern Nullstellen besitzen kann, enstehen undefinierte Ergebnisse durch eine Division durch Null. Der inverse Filter ist also praktisch nicht realisierbar. Häufig verwendet wird jedoch der Wienerfilter:

$$\hat{u} = \frac{\hat{f} \cdot \bar{\hat{h}}}{|\hat{h}|^2 + K} \tag{8}$$

Dabei ist $\hat{f}$ das fouriertransformierte und unscharfe Eingangsbild. Das geschärfte Ausgangsbild wird im Frequenzbereich mit $\hat{u}$ dargestellt und der fouriertransformierte Faltungskern wird mit $\hat{h}$ ausgedrückt. Da diese Größen also fouriertransformiert wurden, erfolgen die Rechenoperationen nun im Komplexen. Die Division erfolgt jedoch mit einem reellen Divisor, dem quatratischen Betrag des fouriertransformierten Faltungskerns $|h^*|^2$. Dazu wird der Wert K addiert. Dieser Wert wirkt als Regularisierer und beseitigt das Problem der Nullstellen von $|\hat{h}|^2$. Die Multiplikation von $\hat{f} \cdot \bar{\hat{h}}$ ist hingegen eine Rechen-

operation mit Real- und Imaginärteil und zwar mit dem komplex Konjugierten von h^*. Der Wienerfilter kann mit K justiert werden: Wenn ein großes K gewählt wird, ist der Schärfungseffekt gering. Wird ein kleines K gewählt, so ist die Schärfung besser. Allerdings nimmt dann auch das Rauschen zu. Der Wienerfilter ist eine lineare Methode und wird durch die Fouriertransformation und komplexer Multiplikation effizient implementiert. Der Einsatz des Wienerfilters ist durch die Stärke des Rauschsignals begrenzt.

Richardson-Lucy [14, 7] Ein häufig verwendeter iterativer Dekonvolutionsalgorithmus ist die Methode von L. B. Lucy [7] und William Hadley Richardson [14], nun als RL bezeichnet:

$$u^{k+1} = \left(h^* * \left(\frac{f}{u^k * h} \right) \right) \cdot u^k \tag{9}$$

Mit der Berechnung von u^1 wird die Iteration gestartet und als Anfangswert wird $u^0 = f$ gewählt. Der Algorithmus erfodert die Division durch das Zwischenergebnis $u^k * h$. Dabei wird also mit dem bekannten Faltungskern gefaltet. Die zweite Faltung erfolgt dann mit dem punktgespiegelten Kern h^*, die falls der Kern punktsymmetrisch sein sollte ($h^*(x) = h(-x)$), einer Faltung mit dem Kern entspricht. Ein Wechsel in den Forierbereich ist hier nicht erforderlich. Die Faltungen können im Orts- oder Frequenzbereich berechnet werden. Der RL Algorithmus ist eine Fixpunktpunktiteration. Er verwendet eine Likelihood-Schätzung und basiert auch dem Poisson-Rauschmodell. Der Algorithmus erfordert als Parameter nur die Anzahl der Iterationen. RL erfordert positive Grauwerte des Eingangsbildes und arbeitet während der Iteration auch positivitätserhaltent. Es zeigt sich ein semi-konvergentes Verhalten. Dabei wird das Rauschen verstärkt und führt nach einer Konvergenzphase der Schärfe zur Divergenz hinsichtlich des Rauschverhaltens.

Robust Regularisierter Richardson-Lucy [4] Der RL-Algorithmus wird nun mit folgendem Funktional in Bezug gebracht:

$$E_{f,h}[u] := \int_\Omega \left(u * h - f - f \cdot \ln \frac{u * h}{f} \right) \mathrm{d}x \tag{10}$$

wobei es nun mit der Substitution $r_f(v) := v - f - f \cdot \ln(v/f)$ abgekürtzt geschrieben werden kann als:

$$E_{f,h}[u] := \int_\Omega \Phi(r_f(u * h)) + \alpha \Psi(|\nabla|^2) \mathrm{d}x \tag{11}$$

Dabei sind Φ und Ψ Straffunktionen und α ein Regularisierer, der abhängig vom Rauschpegel zu wählen ist.

Schließlich läßt sich aus dem Funktional (11) eine Euler-Lagrange Gleichung und eine weitere Fixpunktiteration, ähnlich zu (9) ableiten:

$$u^{k+1} = \frac{(\Phi'(r_f(u^k * h))\frac{f}{u^k * h}) * h^* + \alpha[\mathrm{div}(\Psi'(|\nabla u^k|^2)\nabla u^k)]_+}{\Phi'(r_f(u^k * h)) * h^* - \alpha[\mathrm{div}(\Psi'(|\nabla u^k|^2)\nabla u^k)]_-} u^k \tag{12}$$

wobei die Ausdrücke in den Klammern $[z]_\pm := \frac{1}{2}(z \pm |z|)$ sind. Diese Gleichung ist also wieder eine Iteration und wird „robust regularisierter Richardson Lucy-Algorithmus“ genannt [4]. Er ermöglicht die Dekonvolution bei gleichzeitiger Verwendung einer Regularisierung über die Straffunktionen und dem Regularisierungsparameter α.

1.5 Ansatz für Optimierungen

In [4] wurden bereits Möglichkeiten für eine schnelle Berechnung vorgestellt. Dabei wurde die Faltung im Fourierbereich verwendet. Sie stellt das Grundwerkzeug für eine effiziente Berechnung der Faltung dar. Die Schnelle Fourier-Transformation ist sehr gut parallelisierbar. In [4] wurde auch eine Variante implementiert, die zuerst ein heruntergerechnetes Ergebnis als Zwischenergebnis für die Iteration verwendet („course-to-fine“-Ansatz).

In [8] wurde der 1D-Boxfilter für die Vorwärtsfaltung in RL und RRRL angewandt. Der 1D-Boxfilter [9] ist eine Implementierung der Faltung mit einem ganz speziellen Faltungskern (wie bereits gezeigt im Kap. 1.2). Ebenso wurde im Paper [8] der RRRL eingesetzt, wobei als Anfangswert u^0 ein mit dem Wienerfilter vorberechnetes Bild zugeführt wurde.

Dass Faltungskerne separierbar sind und dadurch in ihrer Berechnung Speicher sparen, wurde von Lehmann et at. gezeigt [16]. Dieser Ansatz wurde bei der Implementation des separierten Boxfilters gewählt.

Wenn der Ansatz aus [8] nun aufgegriffen wird und dabei weitere Faltungsroutinen für spezielle Kernformen entwickelt werden, könnte uU. die Performance eines FFT erreicht oder übertroffen werden. Eine mit SIMD („single instruction on multiple data“) parallelisierte Implementierung wurde unter [15] vorgestellt. Hier wurde der SSE2-Befehlssatz verwendet.

Ein Auszug aus dem Profiler zeigt uns in Abb. 10 den Rechenaufwand der Faltung am Dekonvolutionsprozess eines RL-Algorithmus. Demnach verbaucht die Ortsfaltung über 90% der gesamten Berechnungszeit. Die Optimierung der Faltungsroutinen scheint also lohnend.

Wenn man ebenso bedenkt, dass bei der Iteration die Schärfung von Pixel zu Pixel unterschiedlich und mit steigender Iterationszahl langsamer [4] geschieht, kann auch die genaue Analyse der Konvergenz von konkreten Eingangsbildern eine Beschleunigung bringen.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.42     15.36    15.36      200    76.80    76.80  convolve
  0.58     15.45     0.09                             performRL_convolve
  0.00     15.45     0.00        1     0.00     0.00  transpose


                     Call graph


granularity: each sample hit covers 4 byte(s) for 0.06% of 15.45 seconds
```

Abbildung 10: Ausgabe von gprof zum RL mit Ortsfaltung

2 Zielsetzung

Für die iterativen Dekonvolutionsalgorithmen im Abschnitt 1.4 und die in Abschnitt 1.2 gezeigten Faltungskernen sollen schnelle Verfahren entwickelt werden. In diesem Zusammenhang sollen zwei Wege verfolgt werden: Die Optimierung der Faltungsoperation (siehe Abschnitt 2.1) und die Optimierung hinsichtlich des Konvergenzverhaltens einzelner Pixel (siehe Abschnitt 2.2).

2.1 Filterdesign

In jeder Iteration ist die Faltung mit dem bekannten Kern (der Punktbildfunktion) notwendig. Diese Faltung ist der mit Abstand aufwändigste Teil der Iteration und stellt somit den entscheidenden Schritt der Optimierung dar. Dazu sollen die nun folgenden Faltungskerne näher untersucht werden:

- 1D-Boxfilter (Abb. 2)
- 2D-Boxfilter (Abb. 3)
- Lensblur (Abb. 4)
- allgemein dünn besetzte Kerne (Abb. 6)

Der 1D-Boxfilter wurde im Zusammenhang mit RL und RRRL bereits verwendet und soll zum Vergleich zu den weiteren Filter herangezogen werden. Der 2D-Boxfilter wurde als Faltungsroutine schon untersucht und eingesetzt, im Zusammenhang mit iterativen Dekonvolutionsalgorithmen jedoch noch nicht. Er kann auf verschiedene Arten implementiert werden. Etwa durch Hintereinanderausführen von zwei 1D-Boxfiltern oder durch die Vorberechnung eines kumulierten Grauwertbildes.

Die Bedeutung des 2D-Boxfilters ist nicht so groß wie die der zwei folgenden: Der Lensblurfilter modelliert wie bereits gezeigt (1.2) die Defokussierungsunschärfe. Mit einem

besonders effizient implementierten Lensblur-Filter kann ein iterativer Dekonvolutionsalgorithmus die Defokussierungsunschärfe in Fotos schnell korrigieren. Der praktische Nutzen ist demnach sehr hoch.

Die allgemein dünn besetzten Kerne bieten ebenso vielfältige Einsatzmögklichkeiten. Wie in Kapitel 1.2 lassen sich mit einem solchen Filter diagonale Bewegungsunschärfen effizient entfernen.

Zu den genannten Filtern soll die schnellste Implementierung gefunden werden und die Rechenzeit abhängig von der Kerngröße ermittelt werden.

2.2 Analyse des Konvergenzverhaltens

Mit steigender Iterationszahl nimmt die Schärfung zu und der Anteil der Regularisierung sinkt [4]. Das wirkt sich negativ auf das Rauschverhalten aus und gleichzeitig weiß man nicht genau, nach wievielen Iterationen die gewünschte Schärfung eintritt. Ebenso ist anzunehmen, dass sich die Grauwerte mancher Pixel stärker und schneller ändern. Beispielsweise werden sich Pixel in der Nähe einer Kante anders verhalten, als die Pixel in einer homogenen Umgebung. Pixel mit einer schwachen bzw. langsamen Grauwertänderung können in den darauf folgenden Iterationen übersprungen werden, um Rechenzeit einzusparen. Gesucht ist ein Maß für die Konvergenz bei gleichbleibenden Ergebnissen. Ob Rechenzeit gespart wird, ist zu evaluieren.

3 Methoden

3.1 Softwareumgebung

Um eine effiziente, maschinennahe Implementierung zu ermöglichen wurde die Sprache C verwendet. Auf objektorientierte Programmierung wurde verzichtet. Durch die prozedurale Implementierung können unbeabsichtigte Aufrufe von Konstruktoren, Überladungen und ähnlicher C++ Elemente vermieden werden und der Fokus kann auf den effizienten Programmablauf gelegt werden. Ich folge dem Sprachstandard ANSI C.

Zur Implementierung: Es liegen bereits sämtliche Routinen zur Ein- und Ausgabe, sowie die Dekonvolutionsverfahren RL und RRRL vor. Die genannten Optimierungen (2.1 und 2.2) sollen implementiert und evaluiert werden.

3.1.1 System

Als Betriebssystem wurde Ubuntu 11.10 mit 32 Bit gewählt. Alle Implementierungen sind plattformunabhängig erfolgt. Also ist der Sourcecode ebenso unter Windows lauffähig. Die einzige Ausnahme ist die Zeitmessung. Hier wurde eine Linux-spezifische Systemfunktion verwendet.

3.1.2 Compiler und Linker

Es wurde GCC 4.6.1 gewählt. Der Compile- und Linkvorgang erfolgt in einem Aufruf. Es musste somit kein Makefile verwendet werden. Der Aufruf erfolgt folgendermaßen:

```
gcc ../main_rl.c ../pgmio.c ../functions.c -O2 -Wall -lm -o rl.out
```

Listing 1: Kompileraufruf

- **main_rl.c** stellt die main-Routine zur Verfügung. Hier werden die Routinen zur Dateiein- und ausgabe sowie die Algorithmen aufgerufen
- **pgmio.c** stellt die Dateiein- und ausgabe zur Verfügung
- **functions.c** stellt alle Algorithmen zur Verfügung; darunter sind die Dekonvolutionsalgorithmen, die FFT (Fast Fourier Transformation) und die Faltungsroutinen;

Der Schalter „-O2“ bewirkt die Optimierung mit Level 2 und faßt damit verschiedene Compiler-Schalter zusammen [12]. In der Entwicklung kam zusätzlich Eclipse mit der Erweiterung CDT (C/C++ Development Tooling) zum Einsatz. Entsprechende Projectfiles liegen vor.

Libraries: Der Einsatz von externen Libraries sollte möglichst vermieden werden. Eingebunden wurden die nun folgenden Headerfiles:

- stdlib.h

- stdio.h
- math.h
- sys/time.h spez. Linux-Zeitmessung

Eine einzige systemabhängige Funktion wurde verwendet: Die Datei „time.h“ musste für die Zeitmessung mit der Routine „gettimeofday()“ eingebunden werden. Wenn der Sourcecode unter einem anderen System lauffähig gemacht werden soll, wird eine equivalente Zeitmessung benötigt. Unter [2] sind verschiedene Varianten der Zeitmessung aufgeführt. Außerdem kann für Windows eine Funktion aus der WIN32 API und zwar die Routine „GetSystemTime(..)“ unter Einbindung der Headerdatei „Winbase.h“ verwendet werden [10].

3.1.3 Dateiformat

Wir wollen die Bilder in einem verlustfreien Format einlesen und ausgeben. Außerdem soll die Verwendung externer Libraries möglichst vermieden werden. Dabei bietet sich das Format „Portable-Greymap“ an. Es stellt eine kompressionsfreie und somit auch verlustfreie Dateienein- und -ausgabe für RGB-Bilder als auch für Grauwertbilder zur Verfügung.

pgm - portable greymap Die Spezifikation ist unter [11] zu finden. Das pgm-Format ermöglicht die Verarbeitung von 8-Bit-Grauwerten und 16-Bit. Die Daten können entweder als ASCII-Zeichen oder aber auch als Binärdaten gespeichert werden. Wir implementieren die Speicherung im Binärformat und 8 Bit. Im Listing 2 wird der relevante Codeauschnitt zu pgm - portable greymap 8bit gezeigt.

3.1.4 Datenorganisation und Pixelzugriff

Wenn man überlegt, wie die Daten eines Bildes zu speichern sind, würde man gleich auf ein zweidimensionales Array schließen, weil ein Bild ja auch zweidimensionale Daten repräsentiert. Warum das idR. weniger effizient ist als ein eindimensionales Array wird unter [2] erläutert. Wenn ein Speicherblock für ein eindimensionales Array alloziert wird, kann davon ausgegangen werden, dass ein zusammenhängender Block angelegt wird. Da die Bildgrößen zur Laufzeit nicht bekannt sind, wird das System die Speicherblöcke und Teile davon auf die verschiedenen Speicherstufen verschieben. Wie diese Stufen meist aufgebaut sind, zeigt die Abb. 11 aus dem Artikel [2].

Wie der Speicher konkret alloziert wurde, zeigt das Listing 3. Um darin auf ein einzelnes Pixel zuzugreifen, muss der Index anhand der Spalte und Zeile rückgerechnet werden. Um eine effiziente Implementierung zu gewährlisten, sollte ein Adresssprung auf eine andere Funktion möglichst vermieden werden. Die Berechnung des Index sollte also möglichst „inline“ erfolgen. Dazu kann eine Funktion zur Indexberechnung entweder als Inline-Funktion deklariert werden, oder die Berechnung direkt in der Index-Klammer erfolgen.

```
...
// read 8bit values
int i;
for(i=0; i<size; i++){
  value = fgetc(fp);
  // check for error / EOF
  if((int)value == EOF){
      printf("\nError occured.);
      fclose(fp);
      return 0;
      }
      // save current pixel value in 1D array
      (*pixels)[i] = value;
}
...

 // write 8bit values
 for(i=0; i<size; i++){
  value = (*pixels)[i];
  // check if value is in correct range
  if(value < 0.0){
    c = 0;
  }
  else if(value > 255.0){
    c = 255;
  }
  else{
    c = (unsigned char) (value);
  }
  // write character
  fputc(c,fp);
}
...
```

Listing 2: PGM Dateien einlesen und schreiben

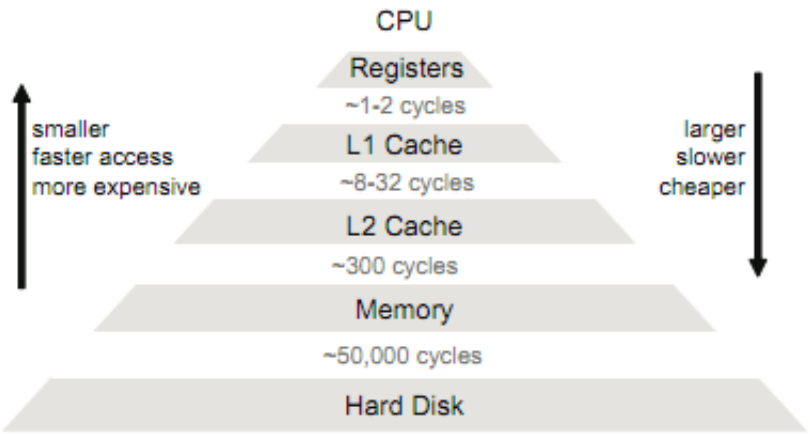

Abbildung 11: Hier sind die Speicherhirachie mit den Zugriffszeiten schematisch dargestellt. Die tatsächlichen Werte hängen von der verwendeten Hardware ab. Die Abbildung stammt aus [2].

```
Allokierung:
------------
float* g = (float*) malloc(unx*uny*sizeof(float));
// Die Groesse des erzeugten Bildes ist: unx*uny

Auf Pixel(10,0) zugreifen:
--------------------------
g[10] = ...

oder mit Makro:
#define getPixelIndex(row, col, nx, ny) (int)(((col) * (nx))+(row))
g[getPixelIndex(10,0,256,256)] = ...

// zusaetzliches Makro mit Bereichspruefung und Randbehandlung 'umbiegen'
#define getPixelIndexBend(row, col, nx, ny)\
  (int)(((col) < 0 ? \
  0 : ((col) >= (ny) ? \
  (ny)-1 : (col) ) * (nx)) + ((row) < 0 ? \
  0 : ((row) >= (nx) ? \
  (nx)-1 : (row))))

g[getPixelIndexBend(10,0,256,256)] = ...
```

Listing 3: Speicherorganisation und Pixelzugriff

Bei der Optimierung wurde ein Makro eingeführt, das die Berechnung am schnellsten ermöglicht: getPixelIndex(..) Siehe dazu Listing 3.

int vs. float. Die Verarbeitung von Gleitkommazahlen ist idR. aufwendiger als das Rechnen mit Ganzzahlen. Deshalb stellt sich die Frage, ob zur Bildberechnung ein Integer- oder Float-Format gewählt wird. Diese Frage läßt sich leicht beantworten: Da die iterativen Dekonvolutionsalgorithmen die Berechnung in Gleitkommazahlen vorraussetzen, kann nur ein Float-Typ verwendet werden. Beim Speichern der Pgm-Datei wird eine Rundung ausgeführt. Wir beschränken uns ja auf Bilder mit 8 Bit. Dennoch kann die Iteration nur mit Gleitkommawerten berechnet werden, da sich ja viele Pixel um weniger als eine Grauwertstufe pro Iteration ändern. Als durchgängiger Datentyp wurde also „Float“ gewählt.

3.2 Verwendete Hardware

Die Software wurde auf x86, also auf PC-Hardware implementiert. Zur Verfügung stand eine Intel-CPU: $Intel^{®}Pentium^{®}ProcessorP6100$ (3M Cache, 2.00 GHz). Sämtliche Optimierungen beziehen sich auf diese CPU, welche über zwei Kerne verfügt. Implementiert wurden jedoch keine parallelisierten Algorithmen, sodass nur ein Kern mit einem Thread genutzt wurde. Hinsichtlich der Optimierung ist speziell das Speichermanagement relevant. Der Pentium P6100 verfügt über eine dynamische Cache-Architektur [5]. Wenn eine andere CPU verwendet wird, sollte die Cache-Aufteilung berücksichtigt werden. Unter Umständen ergeben sich nichtlineare Laufzeitunterschiede durch eine Verlagerung mancher

Speicherblöcke.

Feature	Eigenschaft
No. of Cores	2
No. of Threads	2
Clockspeed	2 GHz
Intel Smart Cache	3 MB
Memory Types	DDR3-800/1066
Max Memory	8 GB

Tabelle 1: CPU-Eckdaten Intel P6100[5]

3.3 Filterdesign

3.3.1 1D Box

Der Boxfilter wurde von McDonnell bereits im Jahre 1981 beschrieben [9]. Die Methode ermöglicht eine besonders effiziente Implementierung der Faltung mit konstanten Grauwerten: Die Bewegungsunschärfe wurde in der Einleitung bereits beschrieben 1.2. Der 1D-Boxfilter wurde bereits im Zusammenhang mit Dekonvolutionsalgorithmen untersucht: [8]. Die Methode kann folgendermaßen zusammengefasst werden: Alle Pixel des Boxfilter-Kerns haben den selben Grauwert. Jedes erste Pixel in einer Zeile, bzw. Spalte wird normal summiert (die Komplexität sit linear zur Kerngröße und zum Eingangsbild). Durch die Größe des Boxfilters (bei einem 1D Boxfilter sind das $1 * k$ oder $k * 1$ Pixel) ergibt sich ein gleitendes Fenster. Die Summe dieses gleitenden Fensters kann schnell ermittelt werden: Wenn das Fenster um ein Pixel weiterverschoben wird, muss nur das überlappende Pixel zur Summe des vorhergehenden Pixels $(n-1)$ addiert werden (in der Abb. 12 mit „+“ markiert). Das überlappende Pixel an der auslaufenden Kernseite wird subtrahiert. (in der Abb. mit 12 mit „$-$“ markiert) Die Komplexität des Boxfilters in 1D beträgt: $\mathcal{O}(N + 2K)$, Wenn N die Länge einer Zeile, bzw. einer Spalte ist und K die Länge des Kerns. Das bedeutet, dass sich die Laufzeit linear zur Größe des Eingangsbildes verhält.

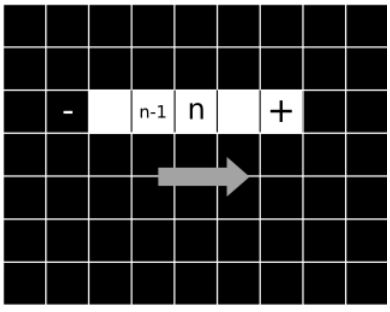

Abbildung 12: Illustration des 1D-Boxfilters

```
  for(x=0;x<nx;++x) {

    // calc the first valid sum of current row
    for(kerny=-halflengthm;kerny<=halflengthp;++kerny) {
         sum += upixel[getPixelIndexBend(x, kerny,nx,ny)];
    }

    // normalize the first pixel
    vpixel[getPixelIndexBend(x,0, nx, ny)] = sum * length_inv;

    // calc the following pixels just with adding
    // the next and subtracting the previous
    for(y=1;y<ny;++y) {

      // add the pixel next to the kernel
      sum += upixel[getPixelIndexBend(x,(int)y+(halflengthp),nx,ny)];

      // subtract one pixel before the kernelmask
      sum -= upixel[getPixelIndexBend(x,(int)y-(halflengthm)-1,nx,ny)];

      // normalize
      vpixel[getPixelIndexBend(x,y,nx,ny)] = sum * length_inv;

    }
    sum = 0;          // set the running sum to zero for the next column
  }
}
```

Listing 4: Codeauschnitt 1D Box

3.3.2 2D Box

Der Boxfilter mit einem Kern von $m * n$ Pixeln kann wie der 1D-Boxfilter effizient implementiert werden. Implementiert wurden der 2D-Boxfilter als separierter 1D-Boxfilter, nicht separierter 2D-Filter und kumulativer Boxfilter.

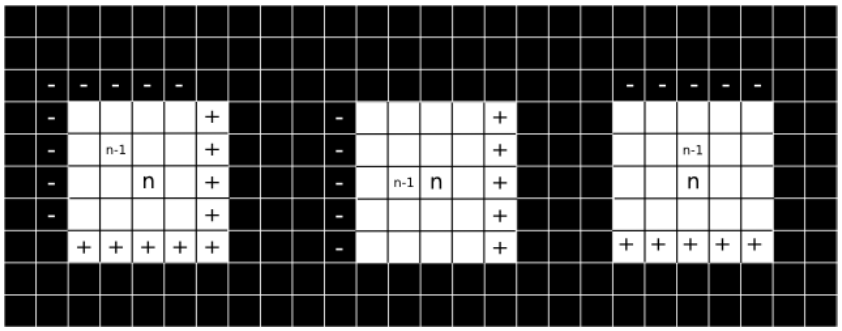

Abbildung 13: Illustration des 2D-Boxfilters

Nichtseparierter 2D-Boxfilter (zeilenweise) Der Boxfilter kann für jedes Pixel zeilen- oder spaltenweise berechnet werden. Wir behandeln das zeilenweise Vorgehen. Dazu wird erst das Pixel(0,0) voll berechnet. Voll berechnet bedeutet folgendes: Alle Grauwerte der Pixel in der Umgebung, die vom Kern aufgespannt wird, werden summiert. Danach wird die erste Spalte berechnet: Dazu gibt es zwei Möglichkeiten. Entweder wird die Spalte über ein gleitendes Fenster berechnet, oder die Berechnung erfolgt gleich wobei Pixel(0,0). Implementiert wurde die zweite Variante. Wenn das Bild eine Größe von $nx \cdot ny$ hat, haben wir nun ny-Pixel berechnet. Nun werden von der ersten Spalte ausgehend alle anderen Pixel zeilenweise berechnet. Diese Summation funktioniert nun über ein gleitendes Fenster. Es muss also jeweils nur das rechte Randpixel des aktuellen Fensters addiert und das linke Randpixel subtrahiert werden. Das zeigt die Abb. 13. Der Pseudocode fasst den Ablauf zusammen: Listing 5.

```
- Summe von Pixel(0,0) bilden
- Spalte(0) berechnen
- jede Zeile ausgehend von Pixel(0,n) berechnen
```

Listing 5: Pseudocode nicht separierter Boxfilter

Separierter 2D Boxfilter. Der 2D-Boxfilter ist separierbar. So kann die Faltung in zwei Schritten berechnet werden: $v = (u \otimes h_1) \otimes h_2, dim(h_1) = m \cdot 1, dim(h_2) = 1$. Das heißt also, dass ein Boxfilter mit der Kerngröße $hx \cdot hy$ mit einem 1D-Boxfilter berechnet werden kann: Erst ein horizontaler 1D-Boxfilter mit der Kernlänge hx und anschließend ein vertikaler 1D-Boxfilter mit der Kernlänge hy. Der Code vom 1D-Boxfilter wird somit

```
- berechne erste Zeile:
   von links nach rechts v(n,0) = v(n-1,0) + u(n,0)

- berechne erste Spalte:
   von oben nach unten v(0,n) = v(0,n-1) + u(0,n)

- jedes weitere Pixel:
von k=1 bis k=unx
  von l=1 bis l=uny
    v(k,l) = v(k,l-1) + v(k-1,l) - v(k-1,l-1) + u(k,l)
```

Listing 6: Pseudocode kumuliertes Grauwertbild

wiederverwendet und benötigt keine zusätzliche Implementation. Die Separierbarkeit von Faltungskernen wurde in [16] diskutiert.

Kumulierter 2D Boxfilter. Aus einem Bild mit m*n Pixeln kann ein Bild mit den kumulierten Grauwerten gebildet werden. Dabei entspricht der Grauwert des Pixels(k,l) der Summe aller Pixelgrauwerte der Fläche eines Rechtecks, welches aufgespannt wird durch die Eckpunkte Pixel(0,0) und Pixel(k,l). Das kumulierte Bild kann wie in Abb. 14 berechnet werden: Dabei werden die Pixelgrauwerte der ersten Zeile und der ersten Spalte aufsummiert. Alle weitere Summen werden daraus errechnet und zwar nach der Vorschrift, die im Listing 6 ersichtlich ist. Ähnlich funktioniert der darauffolgende Filter: Allerdings soll nun nicht die Summe des aufspannenden Rechtecks über dem Pixel, sondern die Grauwertsumme der aufgespannten Kernfläche ermittelt werden.

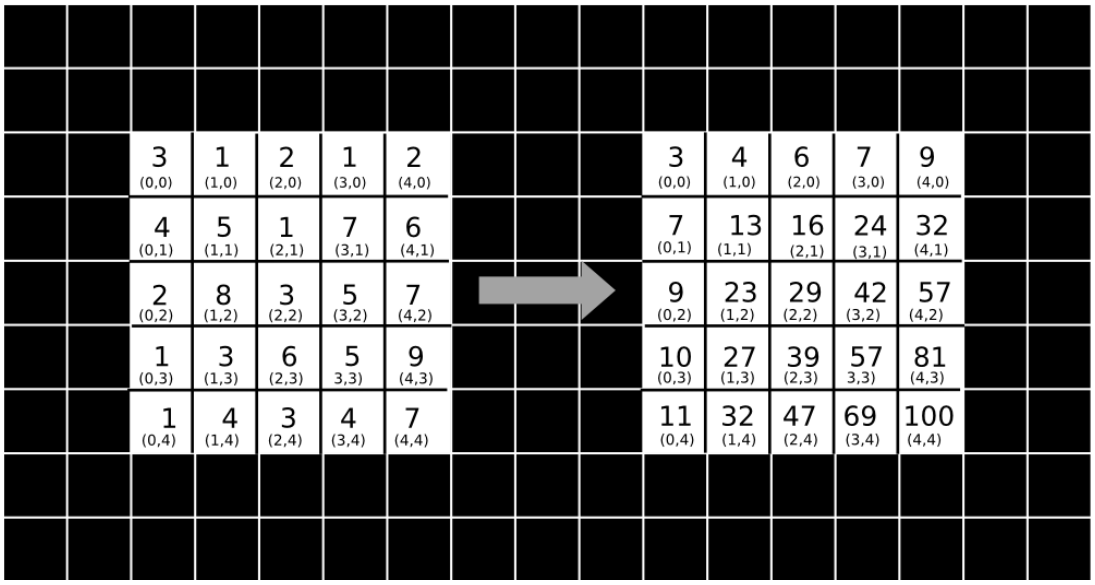

Abbildung 14: kumulierte Grauwerte

Das entstandene Bild hat dann einen Wertebereich von $0 \ldots (g_{max} \cdot m \cdot n)$. Der Datentyp des kumulierten Bildes muss demnach den auftretenden Wertebereich abbilden können. Bei

```
von k=0 bis k=unx
  von l=0 bis l=uny
    v(k,l) =  u(k+hnx,l+hny) - u(k+hnx,l-hny-1) - u(k-hnx-1,l+hny) +
              u(k-hnx-1,l-hny-1)
```

Listing 7: Pseudocode kumulierter Boxfilter

einer verarbeiteten Bildgröße ist das konkret: $256 \cdot 256 \cdot 256 = 2^8 \cdot 2^8 \cdot 2^8 = 2^{24}$. Single-Float Zahlen habe laut IEEE Standard 754-2008 eine Datenbreite von 32 Bit. Es werden 8 Bit für den Exponenten verwendet und 23 Bit für die Mantisse. Da wir aber eine Genauigkeit von 24 Bit genötigen, ist streng genommen der Datentyp „Float-Double“ notwendig.

Unser Datenarray für das kumulierte Grauwertbild muss demnach definiert werden, als: $double * meinKumuliertesBild$;. Bei den implementierten Zeitmessungen wurde jedoch ein Single-Float-Typ verwendet (da das Eingabebild eine mittlere Helligkeit aufweist, ist auch kein Werteüberlauf passiert).

3.3.3 Lensblur, generischer Boxfilter

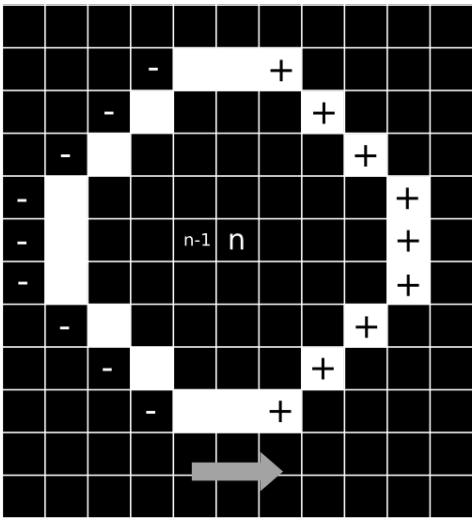

Abbildung 15: schneller Lensblur, Kern: Bresenham r=4px

Die Abbildung 15 illustriert die Methode, die in Listing 8 kurz beschrieben ist. Zu dem Algorithmus führen zwei Ansätze, die zum Teil bereits vorgestellt wurden. Der nun vorgestellte Faltungsalgorithmus wird aus drei einzelnen Teilen zusammengestellt und ermöglicht eine effiziente Berechnug des Lensblur. Wenn der zu faltende Kern über konstante Grauwerte verfügt und alle Pixel in einer Zeile zusammenhängen, kann der Algorithmus verwendet werden. Die Details zeigen die folgenden zwei Ansätze.

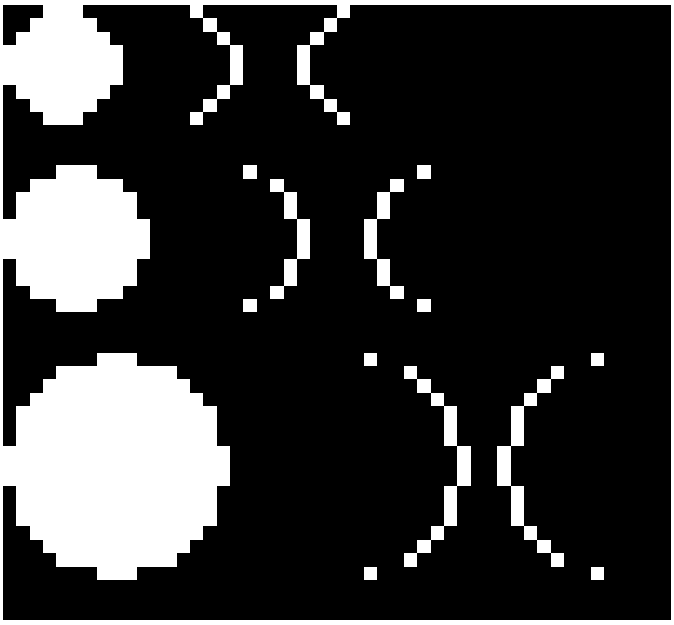

Abbildung 16: jeweils 3er Gruppen für den Lensblur Filter; mit ImageJ erzeugt, Bresenham mit einem Durchmesser von 9px, 11px und 17px, das entspricht einem Radius von 4px, 5px und 8 px

```
berechne erste Spalte:
 - Falte mit jedem Pixel des Kreises:
   (256*58 Instruktionen)

berechne alle anderen Pixel:
 - schiebe das laufende Fenster
   um ein Pixel nach rechts:
   (255*256*16 Instruktionen)
 - normalisiere das aktuelle Pixel:
   (255*256 Instruktionen)
```

Listing 8: Pseudocode generischer Boxfilter

Erster Optimierungsansatz: Wenn wir den Kreis mit der Ortsfaltung berechnen, sind $m \cdot n \cdot d \cdot d$ Gleitkomma-Multiplikationen notwendig. Dabei hat der Kern eine Größe von $d \cdot d$ Pixeln. Es wird jedes Pixel des Eingangsbildes mit jedem Pixel des Kerns gefaltet. Wir beschränken uns auf die besetzten Pixel, dh. auf die Pixel mit einem Grauwert größer als Null. Das führt uns auch schon zur Methode des nächsten Kapitel: Listenfilter 3.3.4. Bei einem Kreis können wir uns also auf die besetzten Pixel beschränken, was einer Kreisfläche entspricht. Wir beschränken die Anzahl der Gleitkomma-Multiplikationen somit auf $r^2 \cdot \pi \cdot m \cdot n$. Ein Beispiel: Bei einem Faltungskern mit einem Durchmesser und einem Eingangsbild von $256^2 px$ sind nicht $256^2 \cdot 17^2$ notwendig, sondern nur mehr $256^2 \cdot 8.5^2 \cdot \pi$. Das entspricht einer Optimierung um einen Faktor $8.5^2 \cdot \pi / 17^2 \approx 227/289 \approx 0.78$. Wir sparen hier also etwa 20 Prozent ein. Allerding steht dem ein Overhead für die algorithmische Umsetzung der Pixelauswahl gegenüber.

Zweiter Optimierungsansatz: Jede Zeile des Kerns besteht aus zusammenhängenden Pixeln. Es ist also möglich hier den Ansatz des Boxfilter anzuwenden: Wir berechenen das erste Pixel jeder Zeile und schieben wieder ein „sliding-window“ nach rechts, wobei die laufende Summe mit den überlappenden Stellen berechnet werden kann. Siehe dazu Abbildung 15. Die Summe aller Grauwerte im Kreis kann gebildet werden durch Hinzufügen der mit „+“ markierten Pixel und durch Abziehen der mit „-“ markierten Pixel. Da die Grauwerte aller Pixel des Kreises gleich sind, brauchen wir außerdem nur einmal pro Eingangspixel normalisieren. Dazu ist eine Gleitkommamultiplikation nötig, während die laufenden Summen ohne Multiplikation auskommen.

Ansätze kombinieren: Wir können die beiden Ansätze gleichzeitig anwenden und kommen zu folgender Optimierung: Bei einem Kern mit Radius 4 und einem Eingangsbild mit $256 \cdot 256$px brauchen wir theoretisch folgende Rechenschritte (siehe Listing 8):
Die Kreisfläche beträgt 57 Pixel (siehe Abb. 15). Die erste Spalte müssen wir mit jedem Pixel des Kreises falten: $256 \cdot 57$ Additionen, 256-mal normalisieren (Multiplikation notwendig). Danach laufende Summe für jedes Pixel nach der ersten Spalte berechnen: $255 \cdot 256 \cdot 16$ und normalisieren: 256^2 mal. Macht also in Summe: $256 \cdot 58 + 255 \cdot 256 \cdot 17 = 1124608$ Instruktionen, während für die volle Ortsfaltung $256^2 \cdot 17^2 = 18939904$ nötig sind. Das entspricht weniger als 6 Prozent.
Demnach lassen sich also 94% der Laufzeit einsparen. Wieviel es dann im praktischen Falle sind, werden die Ergebnisse im Kapitel 4.1 zeigen.

Vom Lensblur zum generischen Boxfilter. Wie in Abb. 15 können wir mit dieser Methode einen Lensblur berechnen. Es ist aber gleichzeitig möglich, mit dem Algorithmus in Listing 8 weitere Kerne abzubilden. Einzige Voraussetzung: Die Pixel müssen konstant und in jeder Zeile zusammenhängend sein. Die Abbildung 17 zeigt eine Auswahl an möglichen Kernen.

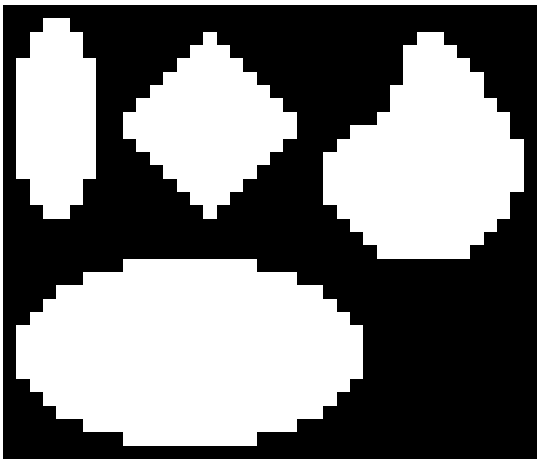

Abbildung 17: verschiedene Kerne für den generischen Boxfilter

3.3.4 Listenfilter, allgemein dünn besetzte Kerne

Bereits beim Lensblur-Filter des vorigen Kapitels wurde der Optimierungsansatz aufgegriffen: Wir beschränken uns auf jene Pixel, dessen Grauwert nicht Null ist. Der Kern besteht anfänglich aus n Pixeln. Diese Pixel haben jeweils einen Grauwert: $h(x)$. In der Implementation wird das in einem Array abgebildet (siehe Listing 9). Aus diesen n Pixel werden nun k Pixel, dessen Grauwert größer Null ist: $h(x) > 0$. Diese Pixel werden in drei Komponenten abgebildet: Einer x-Komponente, einer y-Komponente und einer Grauwert-Komponente. Diese Komponenten werden jeweils als Arrays implementiert. Eine Übersicht der neuen Datenstruktur bietet das Listing 9.
Während vorher bei der Ortsfaltung jedes Pixel des Eingangsbild mit allen Kernpixeln gefaltet wurden (also multipliziert), kann nun eine 3-Komponenten-Liste durchgearbeitet werden. Der Vorteil zeigt sich, wenn viele Pixel des Kernes nicht besetzt (gleich Null) sind. Der Listenfilter wurde noch in einer Variante implementiert, die alle Grauwerte des Kerns als konstant annimmt. Das Array mit den Grauwerten fällt dann weg und aus einer Multiplikation mit den Kernpixeln wird eine Addition. Diese Variante wird in der Folge als „Listenfilter konstant“ bezeichnet.

3.3.5 Optimierung durch Paralellisierung: SIMD

Wie in der Einleitung bereits erwähnt, kann die Optimierung sehr hardwarespezifisch erfolgen. In diesem Fall muss aber eine konkrete Hardwareplattform zu Grunde gelegt werden. In dieser Arbeit konzentrieren wir uns primär auf die nicht hardwarespezifische Optimierung. Ergänzend soll hier jedoch noch eine einfach zu implementierende Variante gezeigt werden: Der verwendete Comnpiler GCC bietet fertige Funktionen zur Verarbeitung mehrerer Datenblöcken in einem Rechenschritt [12]. Als Überbegriff ist wird hier „SIMD“ verwendet: „single instruction, multiple data“. SSE ist eine solche Technologie und ist auf der x86-Plattform weit verbreitet. GCC stellt dafür die „x86 build-in functions“ zur

```
ALTES DATENFORMAT:
- Floatarray. zb: float h[100];
- Inhalt: h = {1.2, 3.5, 90.2, 77.2, 66.1, ...};
- Zugriff:
h[10] // 10tes Pixel
oder h[getPixelIndex(10,0,nx,ny)] // 10tes Pixel der ersten Zeile

NEUES DATENFORMAT:
- drei Floatarrays mit x,y,g
zb:
int x[100];
int y[100];
float g[100];
- Inhalt:
x = {1, 2, 3, 5, 6, 7, ...}; // Liste der x-Koordinaten, 4tes Pixel=0
y = {0, 0, 0, 0, 0, 0, ...}; // Liste der y-Koordinaten
g = {1.2 , 10.5, 90.3, 37.4, 63.4, 72.1, ...}; // enthaelt Grauwerte!=0
- Zugriff:
g[10] // 10tes besetzes Pixel
bzw:
for(i=0;i<listsize;i++) { if(x==10 && y==0) nehme Wert von g[...]}
// zur Suche des 10ten Pixel muss die
Liste durchsucht werden!
```

Listing 9: Datenformat des Listfilter

```
for(kerny=0;kerny<hny;kerny++) {
  for(kernx=0;kernx<hnx-3;kernx=kernx+4) {

    uvec = __builtin_ia32_loadups(...); // load groups of four floats
    hvec = __builtin_ia32_loadups(...); // load groups of four floats
    // Multiply and Add the groups of four floats
    acc = __builtin_ia32_addps(acc, __builtin_ia32_mulps(uvec, hvec));

...
```

Listing 10: verwendete x86 build-in-SSE-Erweiterungen

Verfügung. Sobald der Compilerschalter „-msse“ in der Kommandozeile übergeben wird, sind weitere Funktionen verfügbar. Das Listing 10 zeigt die verwendeten Routinen.

Da die Optimierung jedoch auf der parallelen Multiplikation von Eingangsbild und Kern in 4er Gruppen basiert, müssen diese Bilder jeweils in Vierergruppen aligniert sein. Das bedeutet, dass die Länge des berechneten Vektors (Bildes) ein vielfaches von 4 sein sollte. Unser Eingangsbild mit 256 mal 256 Pixeln ist bereits 4-aligned. Jedoch können wir nur mit einem Kern rechnen, der ebenfalls 4-aligned ist.

Spezielle Kernformen, wie zuvor gezeigt, können durch Parallelisierung leider nicht weiter beschleunigt werden. Näheres dazu in der Diskussion.

3.4 Analyse des Konvergenzverhaltens

Wie unter 2.2 bereits erwähnt wurde, nimmt die Schärfe des Bildes mit steigender Iterationszahl zu. Der Begriff der Schärfe ist nicht klar zu definieren. Wenn die Dekonvolution die Darstellung eines Fotos visuell verbessern soll, ist es schwierig eine Maßzahl für diese subjektive Eigenschaft zu finden. Daher verwenden wir einen praktischen Ansatz und iterieren so lange, bis das Ergebnis visuell ausreichend verbessert wurde. Zunächst soll die Konvergenz der Schärfe näher betrachtet werden. Da wir für die Schärfe keine Maßzahl verwenden, betrachten wir einfach den Grauwertverlauf eines Pixels während der Iteration. Wenn wir also ein Pixel isoliert betrachten lassen sich verschiedene Maßzahlen ableiten. Wir betrachten nun die Grauwertvarianz und die Grauwertdifferenz.

Betrachtung der Varianz. Die Abbildung 18 zeigt die Varianz als Grauwertbild. Auf ein Eingangsbild mit einem Lensblur von $17px$ Durchmesser wurde der Robust Regularisierte Richardson-Lucy Algorithmus angewandt. Die dunklen Stellen weisen auf eine starke Änderung während der Iteration hin. Es fällt also auf, dass es viele Bereiche gibt, in denen die Grauwertänderung nach der gesamten Berechnung nur gering ist.

Mit Hilfe des Verschiebungssatzes kann die Varianz aus den Zwischenergebnissen jeder Iteration berechnet werden:

$$\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\sum_{i=1}^{n} x_i^2\right) - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2 \tag{13}$$

Die Abbildung 19 stellt die Verteilung der Standardabweichung dar. Gezeigt wird die Werte nach 5, 50 und 100 Iteration bei Berechnung mittels RL-Algorithmus. Nach 100 Iterationen nehmen Standardabweichung und Varianz zu.

Grauwertänderung von einer Iteration zur nächsten. Nun wurde einfach die Differenz $|u_k - u_{k-1}|$ gebildet und ausgewertet. Im Diagramm der Abbildung 20 sieht man, dass nach der ersten Iteration besonders starke Änderungen auftreten. Wenige Pixel ändern sich innhalb einer Iteration um bis zu 20 Grauwerte. Weniger stark sind die Änderungen nach der zehnten Iteration. Nach 50 Iterationen ändern sich die Pixel schließlich um weniger als einen Grauwert pro Iteration.

Optimierung. Es wurden mit der Varianz und mit der Grauwertdifferenz zwei Maßzahlen gefunden. Nun soll anhand dieser Zahlen eine Optimierung entwickelt werden. Die Idee ist folgende: Jene Pixel, die sich nur schwach ändern, werden auch nicht berechnet und sparen dadurch Rechenzeit. Um dies zu ermöglichen, musste der Programmablauf des Dekonvolutionsalgorithmus und der Faltung angepasst werden. Die Abbildung 21 schematisiert die Anpassung am RL-Algorithmus: Wenn man einzelne Pixel in der Faltungsroutine ausläßt, muss ein Ersatzwert für die darauf folgende Faltung und Iteration zugewiesen werden. Die roten Linien sollen die Auswirkung auf die darauf folgenden Berechnungen zeigen. Das graue Pixel wurde hier nicht gefaltet. Die Lösung ist folgende: Nach jeder

Abbildung 18: Varianz bei RRRL, 0.000001, 0.1, 1000 Iterationen, invertierte Darstellung

Faltung wird das Zwischenergebnis eines jeden Pixels gespeichert. Wird ein Pixel in der aktuellen Iteration ausgelassen (nicht gefaltet), kann dessen Wert für die darauffolgenden Berechnungen aus dem Zwischenspeicher der vorhergehenden Iteration genommen werden. Im Programmablauf der Faltung muss eine if-Abfrage hinzugefügt werden. Sie prüft, ob die Änderung des jeweiligen Pixels für eine Bearbeitung ausreicht. Folgende Faltungsroutinen können mit dieser Methode angewendet werden: die Ortsfaltung und der Listenfilter. Diese zwei Routinen arbeiten Pixel für Pixel ab. Die Faltung im Fourierbereich scheidet für diese Optimierung aus, da alle Pixel in den Frequenzbereich zusammen transformiert werden müssen, um die Faltung durchzuführen. Die zuvor gezeigten Boxfilter (1D-Boxfilter, 2D-Boxfilter, generischer Boxfilter oder Lensblur) können mit dieser Optimierung ebenso nicht verwendet werden, da ihre Berechnung auf dem Konzept der „gleitenden“ Summen beruht und das Ausnehmen einzelner Pixel so nicht möglich ist. Eine gemeinsame Optimierung mit dem Listenfilter bietet sich jedoch an. Der Listenfilter hat die Eigenschaft, dass er nur die besetzten Pixel berechnet. Wenn also ein dünn besetzter Kern mit dieser „konvergenz-optimierten“ Methode berechnet wird, sind zusätzliche Einsparungen in der Rechenzeit möglich.

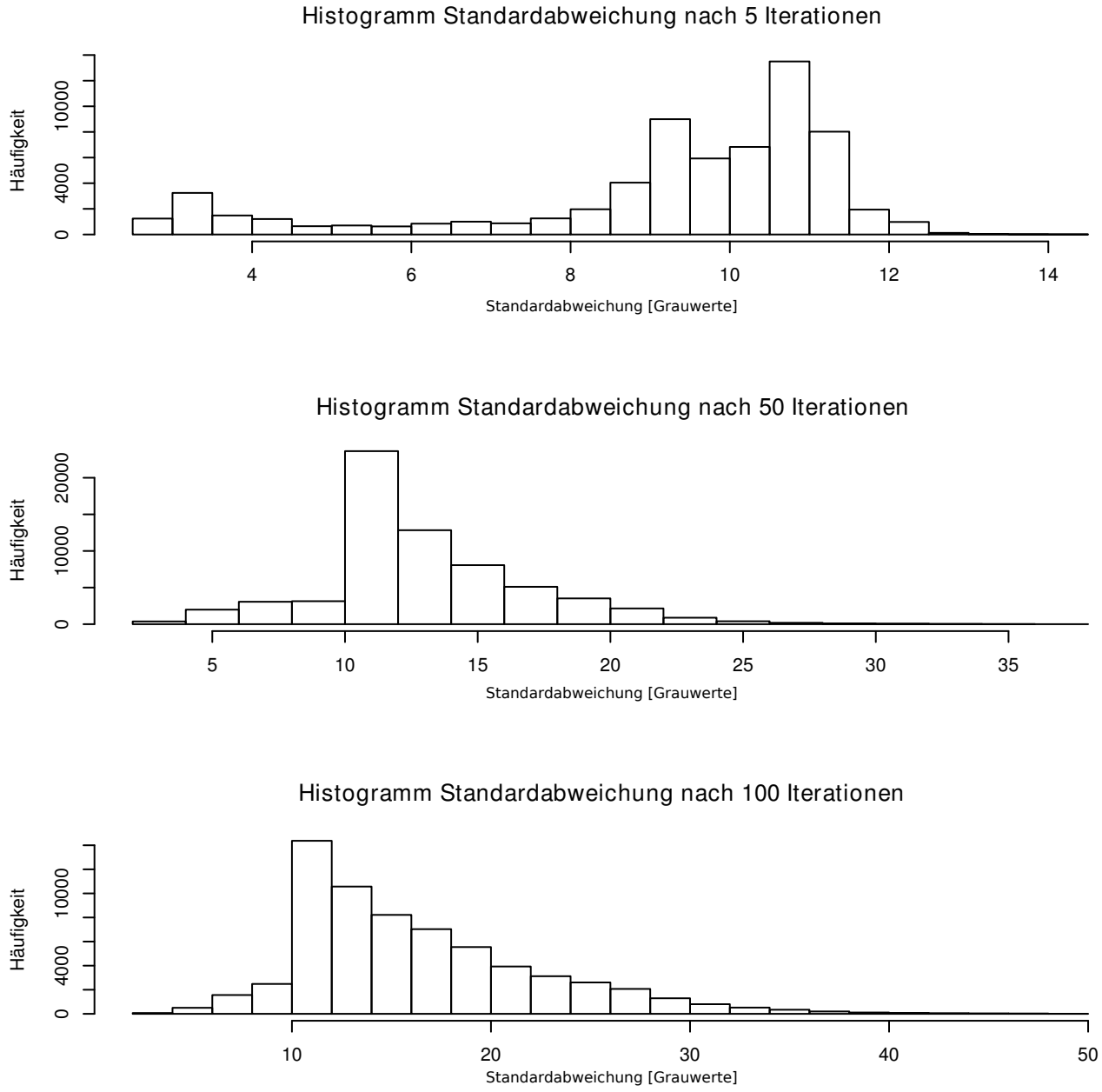

Abbildung 19: RL bei einem Lensblur von 17px, mit 5, 50 und 100 Iterationen. Standardabweichungen der Grauwerte eines jeden Pixels

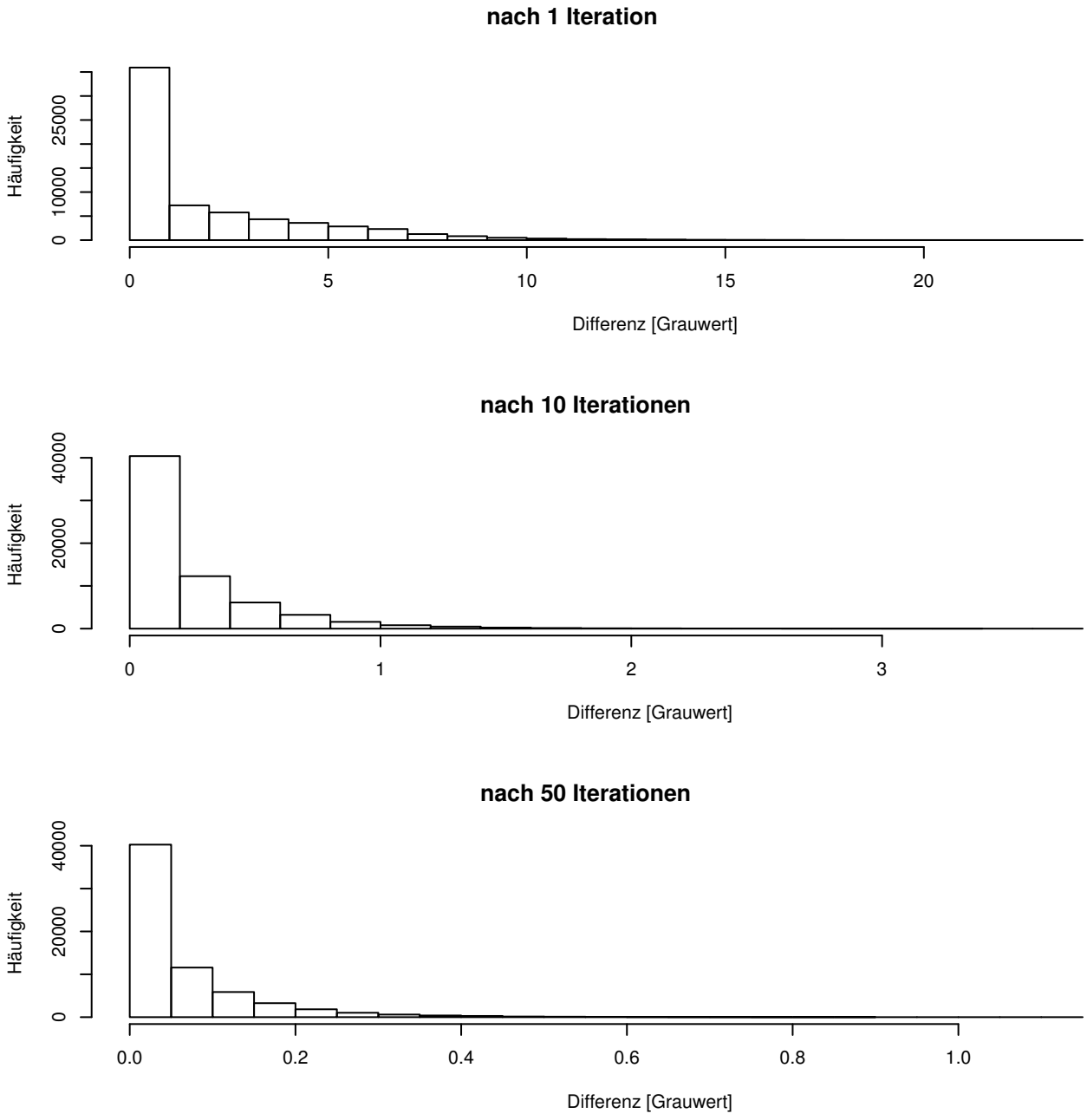

Abbildung 20: Grauwertänderung von einer Iteration zur nächsten;

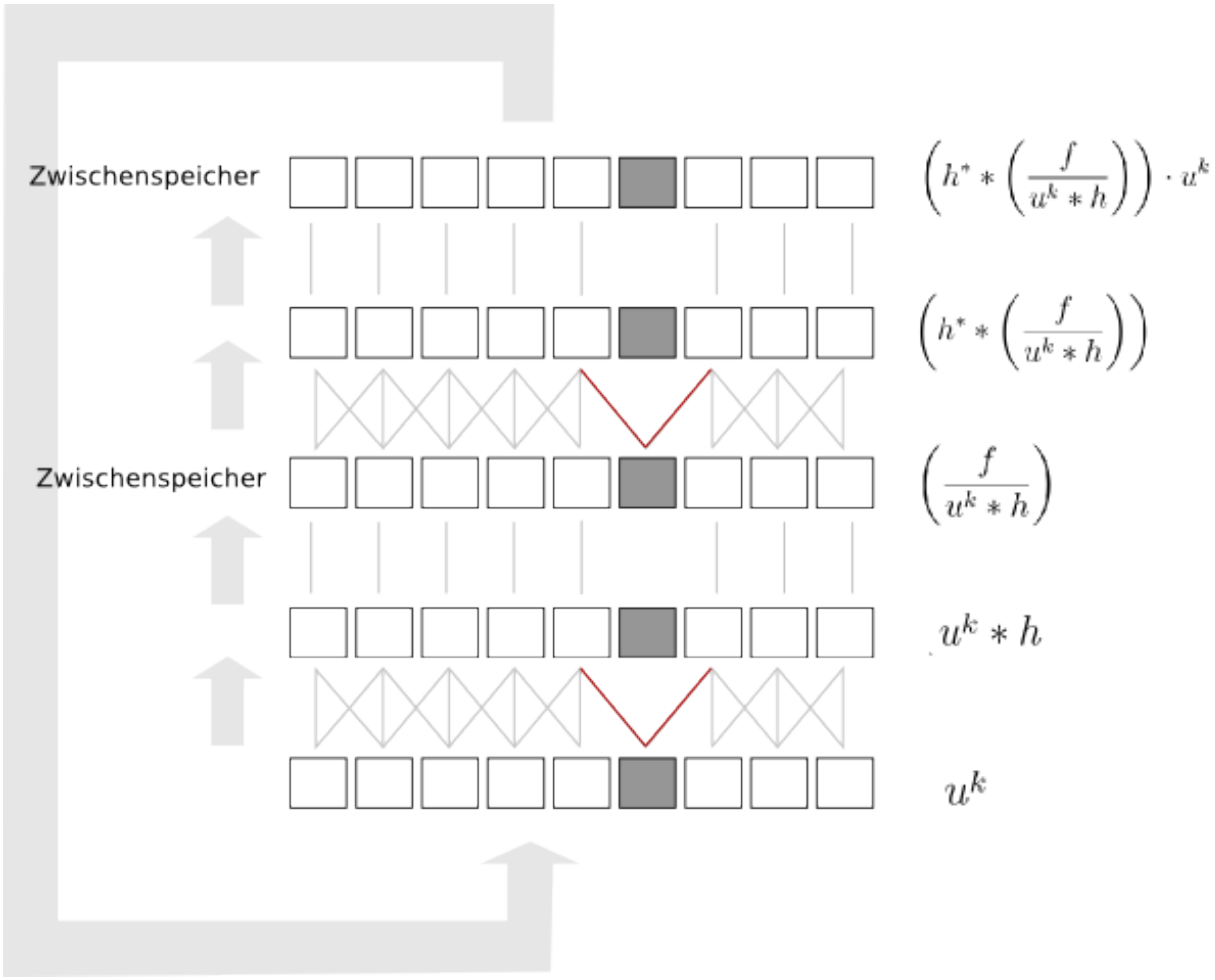

Abbildung 21: Ablaufdiagramm der optimierten Version

4 Ergebnisse

4.1 Messergebnisse Faltungsoptimierung

Das Ziel der optimierten Faltungsroutinen, ist dass die Rechenzeit bei gleichbleibender Qualität minimiert wird. Kurze Rechenzeit bei gleichbleibendem Ergebnis wird nun als gute Performance bezeichnet.

4.1.1 Qualitative Betrachtung

Um einen Vergleich der Performance verschiedener Faltungsroutinen zu ermöglichen, muss das berechnete Ergebnis übereinstimmen. Mit Rücksicht auf die Effizienz einzelner Filter wurden bei den Randbedingungen jedoch Abstriche von dieser exakten Übereinstimmung gemacht. Diese Unterschiede werden hier kurz betrachtet.

Ortsfaltung und Faltung im Frequenzbereich. Wie im Kapitel 1.3 beschrieben, werden die Ränder hier anders behandelt. Die Abb. 22 zeigt die Grauwertdifferenzen zwischen dem gefalteteten Bild im Orts- und Frequenzbereich.

Separierter 1D- und 2D-Boxfilter.Um den Boxfilter qualtitativ zu evaluieren, bietet sich ein Vergleich mit der Faltung im Ortsbereich an. Dazu wurde ein Bild mit dem separierten Boxfilter berechnet und dasselbe Bild mit dem Ortsfilter. Anschließend wurden beide Bilder verglichen und die Differenzen der Grauwerte berechnet. Der Vergleich zeigte keine Unterschiede.

Zeilenweiser Boxfilter. Der zeilenweise berechnete Boxfilter wurde ebenso mit der Faltung im Ortsbereich verglichen. Die Implementation zeigt keine Unterschiede.

Listenfilter. Der konstante Listenfilter und der nicht konstante Listenfilter wurden auf Unterschiede mit dem Ortsfilter verglichen. Auch hier konnten keine Unterschiede festgestellt werden.

Generischer Boxfilter. Um den generischen Boxfilter zu vergleichen, wurden drei passende Kerne erstellt. Diese Kerne entsprechen wieder einem 7 mal 7 Pixel großen Boxfilter. Somit kann wieder mit der normalen Ortsfaltung verglichen werden. Das berechnete Differenzbild zeigte, dass gegenüber dem Filter im Ortsbereich keine Unterschiede auftreten.

Kumulierter Boxfilter. Der kumulierte Boxfilter wurde auch mit der Faltung im Ortsbereich verglichen. Hier kann aber die Randbedingung aus Kapitel 1.3 „Umbiegen“ nicht ohne größeren Zusatzaufwand implementiert werden. Die Abb. 23 zeigt die Unterschiede des kumulierten Boxfilters zur Faltung im Ortsbereich. Hier wurde zum Vergleich ein Faltungskern von $17 * 17$px gewählt.

Ortsfaltung mit SIMD. Durch die 4fache Parallelisierung war die Implementierung derselben Randbedingung wie beim Ortsfilter nicht exakt möglich. Der Grund dafür liegt in der Verwendung des Makros „getPixelIndexBend“. Dieses Makro gibt bei Überschreitung

aus dem Bildbereich das am nächsten gelegene Randpixel zurück. Wenn vier Pixel in einer Rechenoperation verarbeitet werden, wird nur der jeweils erste Zeiger auf die Vierergruppe berechnet. Deshalb wird bei Überschreitung links und rechts nicht genau das Randpixel gewählt, sondern ein Pixel aus dem viel Pixel breiten Randbereich. Die Abb. 24 zeigt die Messung der Grauwertdifferenzen.

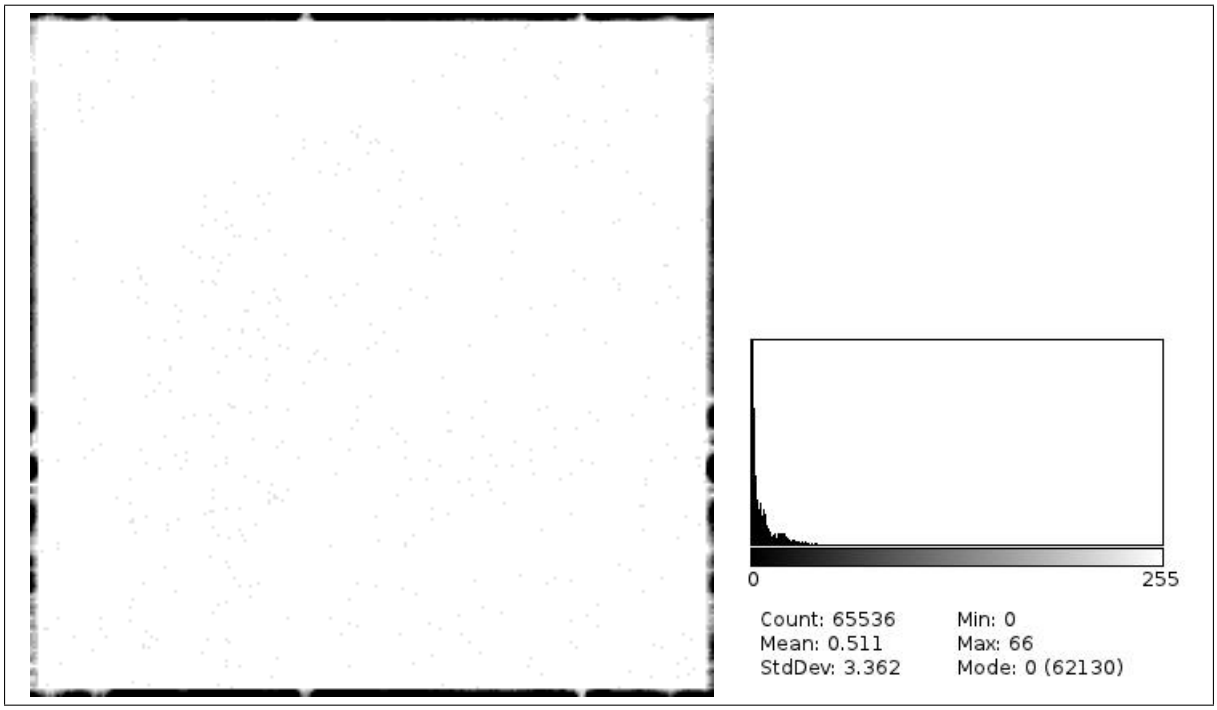

Abbildung 22: Grauwertdifferenz: Die linke Abbildung zeigt kaum sichtbare Randunterschiede der Faltung im Orts- und Frequenzbereich, sowie Rundungsdifferenzen in der Bildmitte; Darstellung 30fach verstärkt und invertiert, rechts: das Histogramm

Lokalisierung. Ein weiterer qualitativer Vergleich ist durch Prüfung der Lokalisierungsgenauigkeit möglich. In Abb. 25 wurde von ausgewählten Methoden jeweils der Punkt (103,103) farblich markiert. So ist eine einfache visuelle Kontrolle der Lokalisierung möglich. Geschärft wurde jeweils mit RRRL bei einer Unschärfe eins Boxfilter mit $7 \cdot 7$ px. Visuell lassen sich keine Differenzen feststellen.

4.1.2 Performancemessung

Um die Laufzeit der einzelnen Faltungsroutinen zu vergleichen, wurden diese einzeln gemessen und als Teil einer RL- sowie RRRL-Dekonvolution gemessen.

Faltungsroutinen einzeln gemessen. Es wurde jeweils das Bild mit dem Kameramann (Abb. 1) mit einem Kern von $17 \cdot 17$ Pixeln gefaltet. Das Diagramm aus Abb.26 zeigt die gemessenen Berechnungszeiten. Folgende Parameter wurden gewählt:

- convolve: Ortsfaltung. Kern $17 \cdot 17$px
- listFilter: Listenfilter, kreisrunder Kern mit 213 Elementen Listengröße

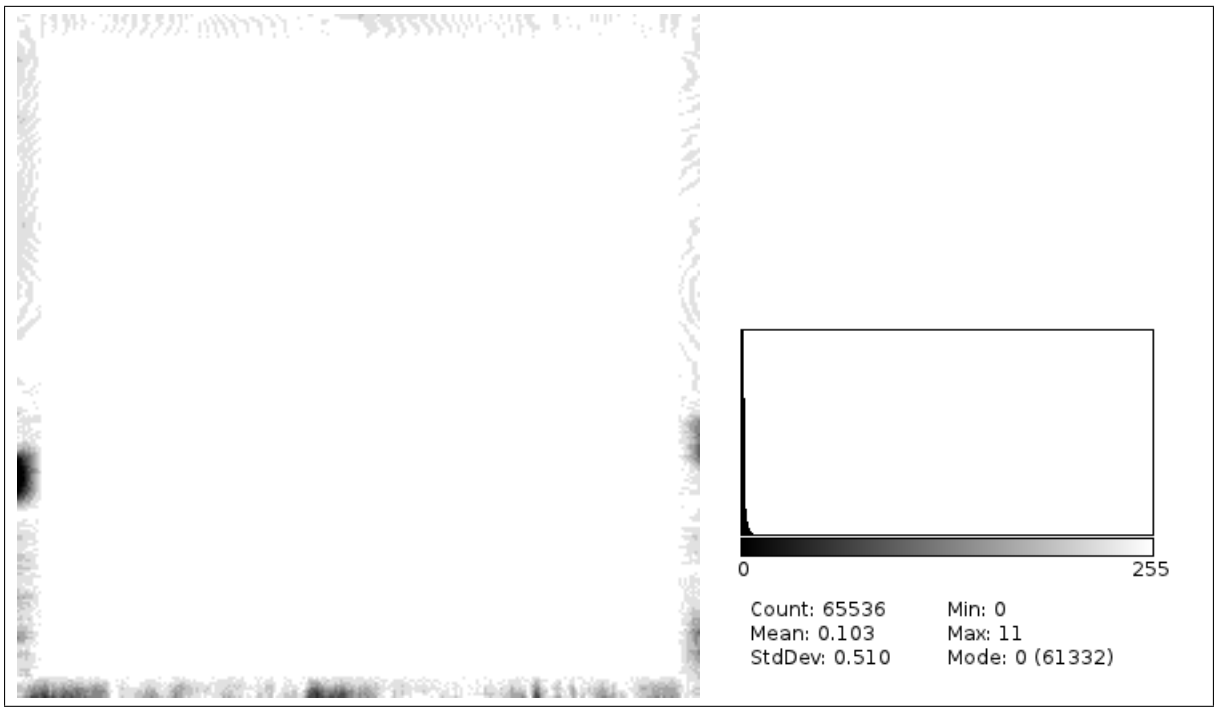

Abbildung 23: Grauwertdifferenz zwischen dem kumulierten Boxfilter und der Faltung im Ortsbereich. Die Kerngröße nimmt zum Rand hin ab, so dass dort eine Differenz entsteht; die Darstellung wurde invertiert und 30fach verstärkt

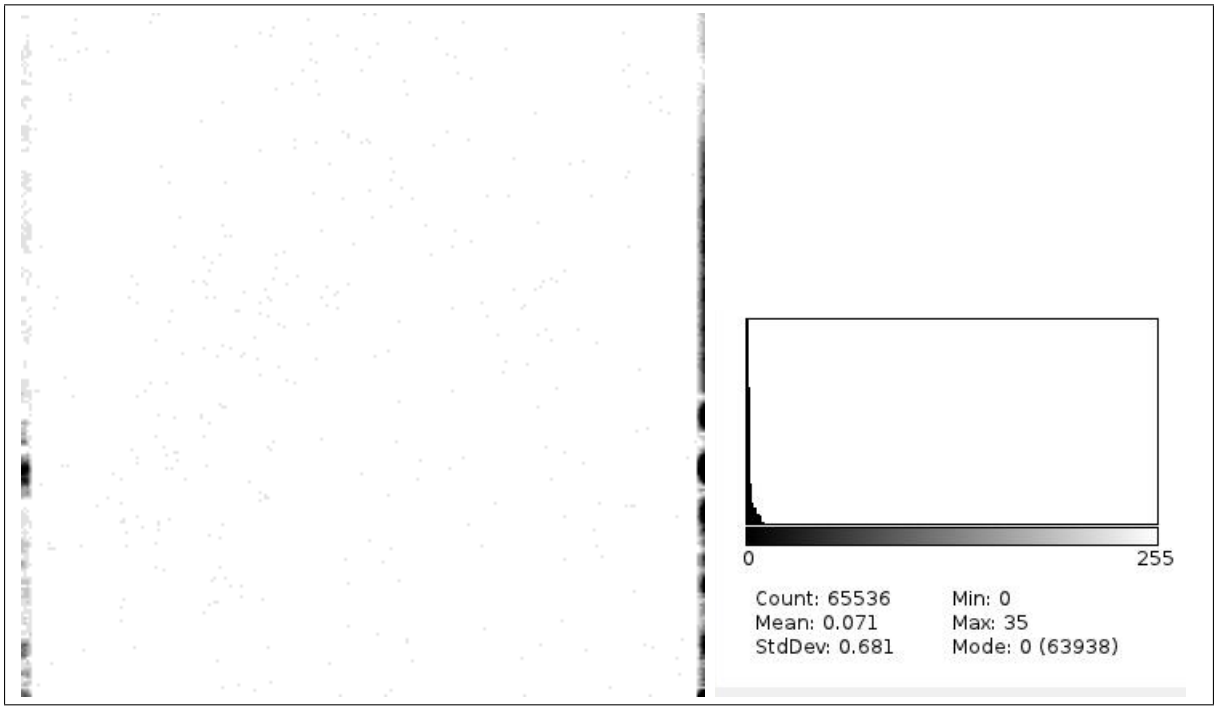

Abbildung 24: Grauwertdifferenz des Filters mit SIMD und ohne. 30fach verstärkt. Die Randbedingung weicht leicht ab und Rundungsfehler in der Mitte sind zu erkennen.

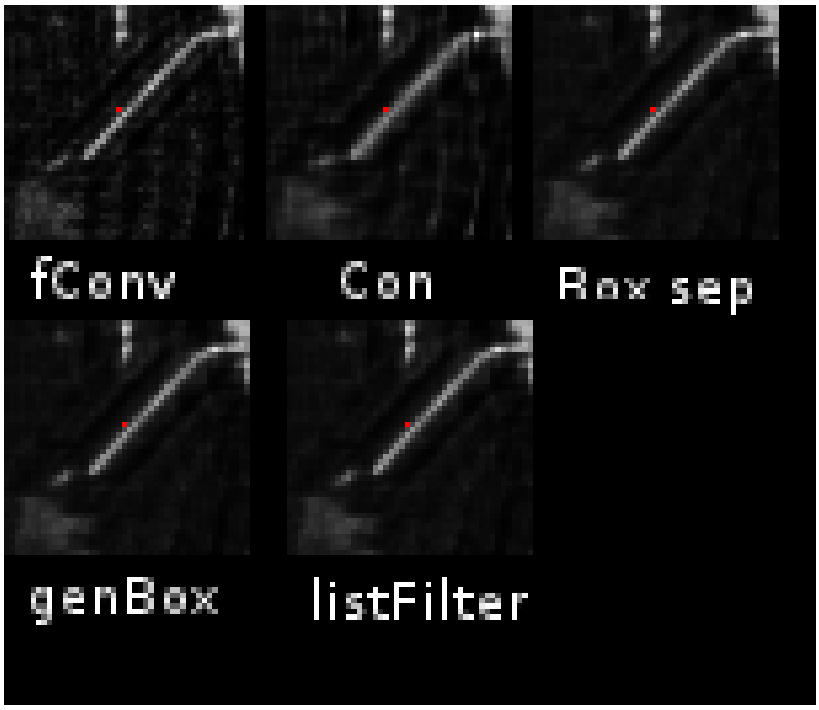

Abbildung 25: Lokalisierungsgenauigkeit. Der Punkt (103,103) wurde jeweils rot markiert. untersuchte Faltungen: F. im Frequenzbereich, Ortsbereich, separierter Boxfilter, generischer Boxfilter und der Listenfilter.

- convolve SIMD: SSE-optimierte Ortsfaltung mit einem Kern von $20 \cdot 20$ Pixel
- fconvolve: Faltung im Fourierbereich; die Kerngröße ist $17 \cdot 17$px
- genbox: generischer Boxfilter mit einem Lensblur von 17px Durchmesser. 213 Pixel besetzt
- box2D nicht separiert: Kerngröße $17 \cdot 17$px
- box2D separiert: Kerngröße $17 \cdot 17$
- box2D kumuliert: Kerngröße von $17 \cdot 17$px.
- box1D: Motionblur in horizontaler Richtung mit 17px Unschärfe

Die Routinen „listFilter“ sowie „convolve SIMD“ sind stark von dem gewählten Kern abhängig. Deshalb ist der direkte Vergleich zu den anderen Methoden bei einer Kerngröße von $17 \cdot 17$px nicht möglich. Beim Listenfilter ist weniger entscheidend, wie groß der Kern ist, sondern wieviel Pixel des Kerns besetzt (> 0) sind. Details werden später in der Performancemessung zum Listenfilter genannt. Die optimierte Ortsfaltung mit SIMD ist ebenso vom Kern abhängig: Die Kernlänge sollte stets ein Vielfaches von vier sein. Deshalb ist auch hier kein Vergleich mit einer Kerngröße von $17 \cdot 17$px möglich. Beim SIMD-Verfahren wurde deshalb eine Kerngröße von $20 \cdot 20$ gewählt. Näheres dazu dann im Abschnitt zur Laufzeitmessung SIMD.

Laufzeit Listenfilter Wie bereits erwähnt, ist der Listenfilter nicht von der Kerngröße abhängig, sondern von der Anzahl besetzter Pixel (> 0). Wenn also ein Kern vorliegt, der

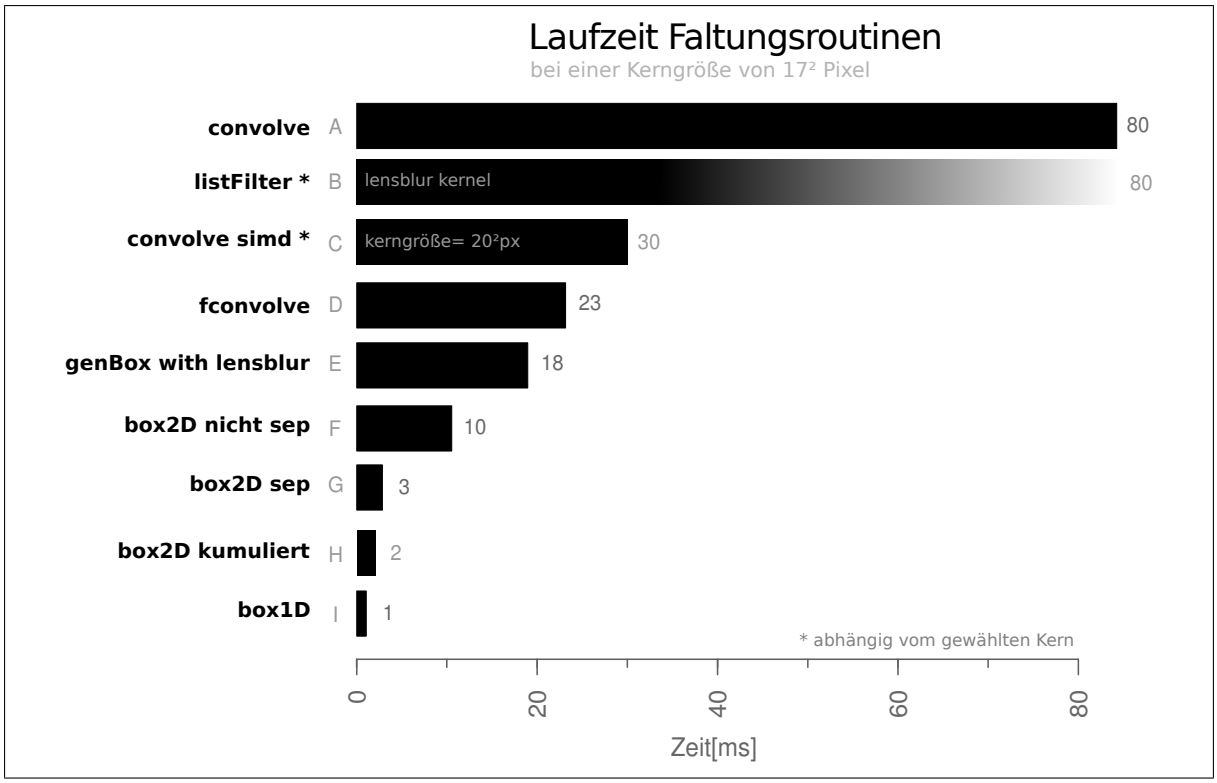

Abbildung 26: Zeitbedarf der Faltungsroutinen

dünn besetzt ist, dh. wenn viele Pixel gleich Null sind, soll der Listenfilter theoretisch schneller sein. Naheliegend scheint ein Vergleich Ortsfaltung zu Listenfilter, wobei beim Ortsfilter die Kerngröße und beim Listenfilter die Listengröße variiert. Das Diagramm von Abb. 27 zeigt einen solchen Vergleich. Mit der Speicherorganisation in drei Komponenten (Kapitel 3.3.4) entsteht ein Overhead. Das bedeutet, dass die Ortsfilterung mit einem Kern von 200 Pixel schneller ist, als der Listenfilter mit 200 Elementen. Die Messung zeigt: Die Ortfaltung braucht bei einem Kern von $1 \cdot 200$ Pixel etwa 48ms. Der Listenfilter zeigt eine Laufzeit von 87ms (bei 200 Elementen). Daraus resultiert ein Overhead von über 40 Prozent. Ein Vergleichsbeispiel: Wenn im optimalen Fall ein diagonaler Bewegungsunschärfekern vorliegt, sind mit dem Listenfilter nicht $10 \cdot 10$ Pixel zu berechnen (bei einer Kerngröße von $10 \cdot 10$px), sondern nur 10. Daraus resultiert in etwa ein Speedup um den Faktor 6 (inkl geschätzter Overhead).

Der konstante Listenfilter ist effizienter. Es müssen deutlich weniger Gleitpunkt-multiplikationen durchgeführt werden. Der konstante Listenfilter benötigt für 200 Listenelemente 74ms. Das bedeutet einen Overhead von etwa 35 Prozent. Die Vorteil der Gleitpunkt-Additionen statt -Multiplikationen ist von der verwendeten CPU abhängig.

Laufzeit SIMD Ortsfaltung. Wenn wir das Diagramm von Abb. 28 betrachten, können wir die Laufzeit der SIMD-optimierten Ortsfaltung sehen. Die auftretenden Peaks zeigen das nicht-deterministische Verhalten des Betriebssystems: Ubuntu ist kein Echt-

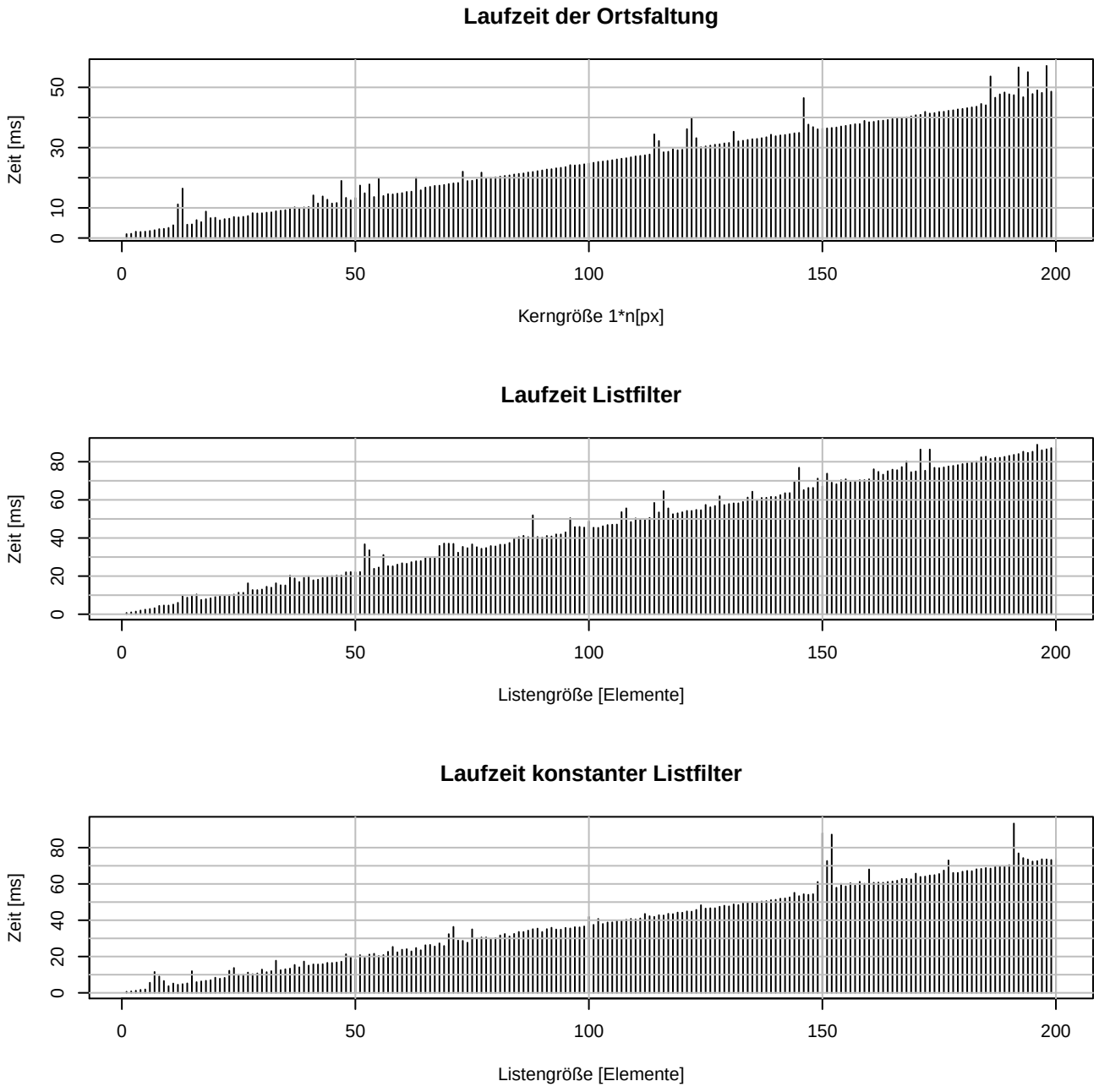

Abbildung 27: Zeitbedarf der Faltungsroutinen Listfilter und Ortsfaltung

zeitbetriebssystem und garantiert deshalb auch nicht die gleichmäßige Priorisierung eines Threads.

Außerdem: Die grüne Linie, also die optimierte Version zeigt stufenhafte Sprünge in Vierer-Schritten. Hier ist anzunehmen, dass die CPU bei vier-alignierter Kernlänge alle Rechenoperationen auf den SSE-Registern druchführen kann. Bei Kerngrößen, welche nicht durch vier teilbar sind, müssen immer ein, zwei oder drei Rechenschritte auf den normalen CPU-Registern einzeln ausgeführt werden. Die nächste Frage stellt sich also: Wie ist das Verhalten, wenn der Kern nicht die Form $1 \cdot n$ hat, sondern realistischerweise die Dimension $n \cdot n$? Ist eine Verschlechterung der Performance zu erwarten? Theoretisch sind bei einem Kern von beispielsweise $7 \cdot 7$ Pixel mindestens $3 \cdot 7 = 21$ nicht parallisierte Rechenschritte nötig. Den Unterschied zeigt Abb. 29. Die ungünstige Speicherausrichtung scheint demnach keinen großen Einfluss auf die Performance zu haben. Das unregelmäßige Ansteigen kann aber mit der Speicherauslagerung in Zusammenhang gebracht werden.

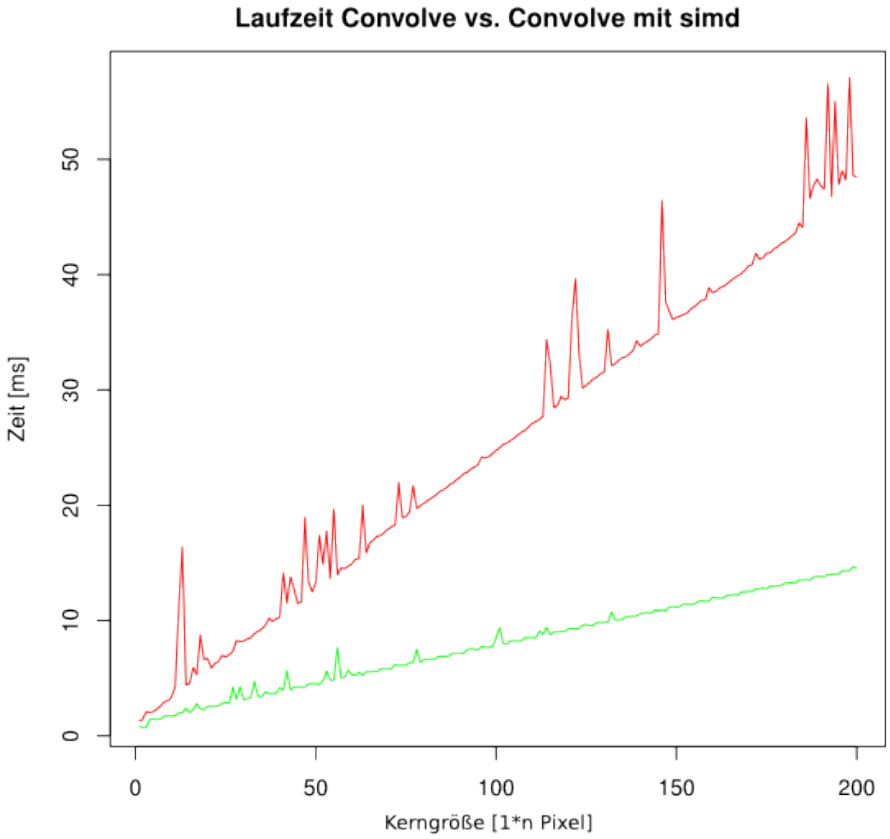

Abbildung 28: Zeitbedarf der gewöhnlichen Ortsfaltung und der SIMD-optimierten. Kerngröße steigt in der Form $1 \cdot n$ linear an

Faltungsroutinen mit Richardson-Lucy-Dekonvolution (RL). Die vorgestellten Faltungsroutinen wurden in den RL-Algorithmus integriert und getestet. Die nachfolgenden Abbildungen zeigen den Zeitbedarf der gesamten Dekonvolution bei 100 Iterationen. Die

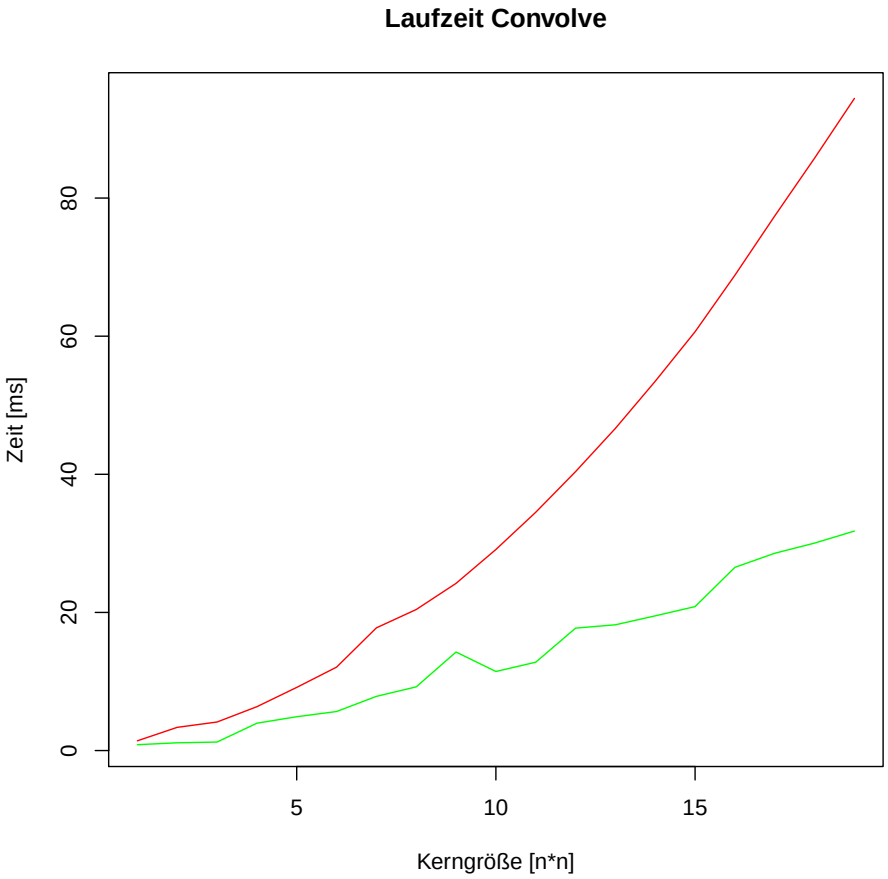

Abbildung 29: Zeitbedarf der gewöhnlichen Ortsfaltung und der SIMD-optimierten. Kerngröße steigt in der Form $n \cdot n$ quadratisch an

Abb. 30 zeigt die Ergebnisse nach Kerngrößen geordnet. Die Abb. 31 zeigt dieselben Ergebnisse, aber nach Faltungsmethoden gruppiert. Nachfolgend werden die Parameter der jeweiligen Faltungsfunktionen aufgelistet:

- convolve: Ortsfaltung
- listFilter const: Listenfilter mit Lensblur-Kern; Listengröße $\approx d^2/4 \cdot \pi$ Elemente. $d = 17px, 11px, 9px$
- convolve SIMD: Ortsfaltung mit SSE-Optimierung
- fconvolve: Faltung im Fourierbereich
- genBox: Faltung mit generischem Boxfilter. lensblur, dh. kreisrunder Kern. $d^2/4 \cdot \pi$ Elemente
- box2D nicht separiert: Boxkern mit jeweiliger Kerngröße.
- box2D kumuliert: kumulierter Boxfilter
- box2D separiert: Boxkern mit jeweiliger Kerngröße.
- box1D: Boxkern mit $1 \cdot n$ Pixel. $d = 17, 11, 9$

Die Ergebnisse der Ortsfaltung, Faltung im Frequenzbereich und des Boxfilters sind bekannt und dienen an dieser Stelle als Vergleich. Der 1D-Boxfilter ist bekannterweise der schnellste Filter. Er arbeitet aber auch mit dem speziellsten Filterkern: der konstanten Bewegungsunschärfe in horizontaler, bzw. vertikaler Richtung. Wie alle Boxfilter ändert er sich mit der Kerngröße nicht. Die Faltung im Fourierbereich ist ebenso von der Größe des Kerns unabhängig.

Faltungsroutinen mit RRRL. Bei der Faltung mit dem RRRL zeigen sich ähnliche Ergebnisse, wie bei der Faltung mit dem RL-Algorithmus. Hier sollte erwähnt werden, dass eine dritte Faltung durchgeführt werden muss. Das qualitative Ergebnis des RRRL ist mit dem des RL-Algorithmus nicht uneingeschränkt vergleichbar, da der RRRL eine zusätzliche Regularisierung zur Verfügung stellt. Als Parameter wurden gewählt: $\alpha = 0.000001$ und $\epsilon = 0.1$. Der RRRL wurde ebenfalls bei 100 Iterationen getestet. Hier fällt sofort auf, dass das Minimum des 1D-Boxfilters nicht so deutlich hervortritt, als beim RL. Der Grund dafür liegt im Mehraufwand der Berechung: Beim RRRL müssen noch zusätzliche Werte (zb. Φ) berechnet werden und der Anteil der Faltung an der gesamten Rechenzeit ist nicht mehr so hoch.

4.2 Messergebnisse der Konvergenzanalyse

Messungen bei 100 Iterationen und verschiedener Optimierungsparametern. Berechnet wurde das Bild des Kameramanns (Abb. 1) mit einem Lensblur von 17px Durchmesser. In zwei Messreihen wurden der optimierte RL und der optimierte RRRL mit der gewöhnlichen Ortsfaltung verwendet. Alle 10 Iterationen wurde die Faltung ohne

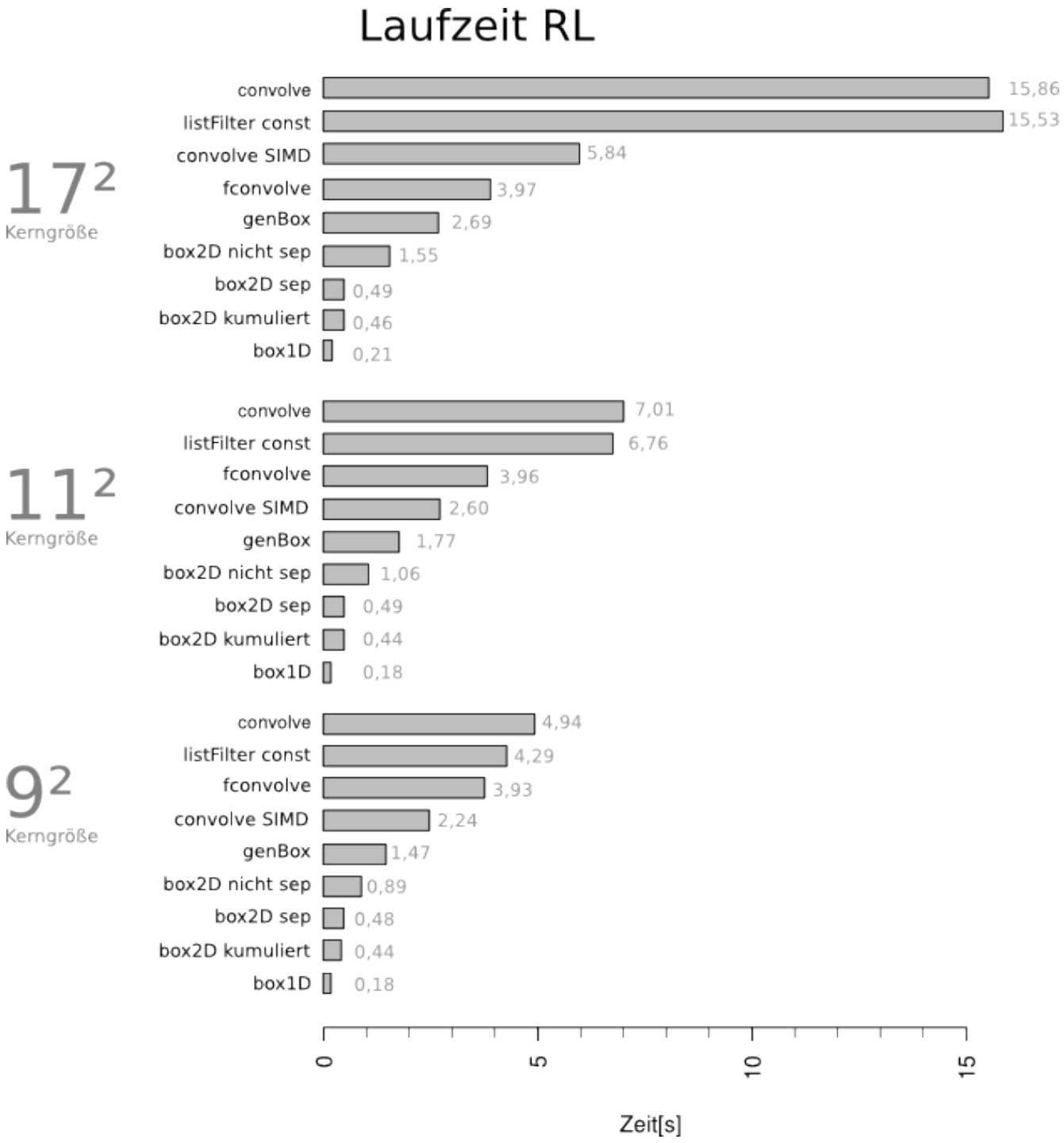

Abbildung 30: Zeitbedarf RL mit verschiedenen Faltungsroutinen, gruppert nach Kerngröße, 100 Iterationen

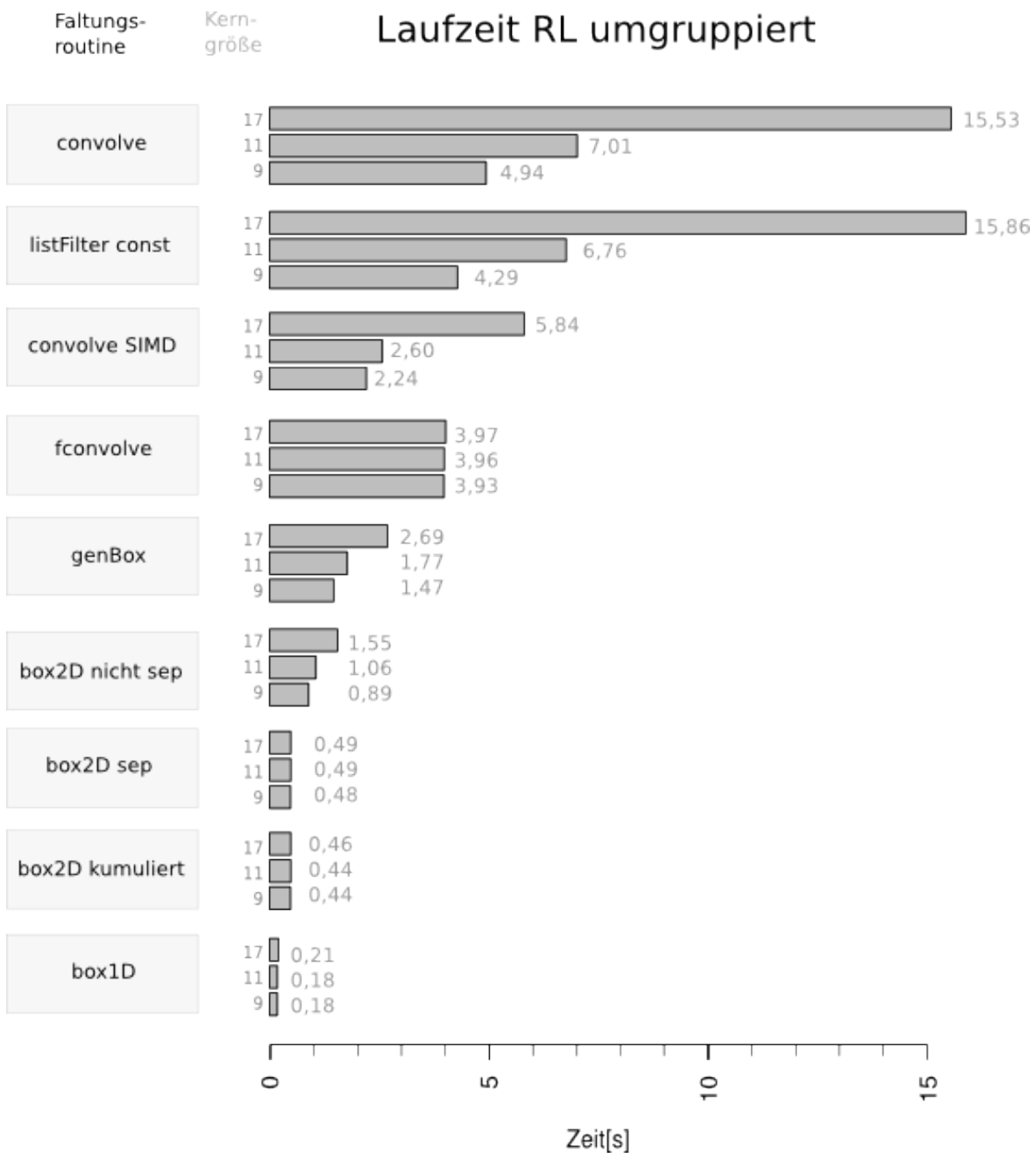

Abbildung 31: Zeitbedarf RL mit verschiedenen Faltungsroutinen, gruppert nach Faltungsroutinen, 100 Iterationen

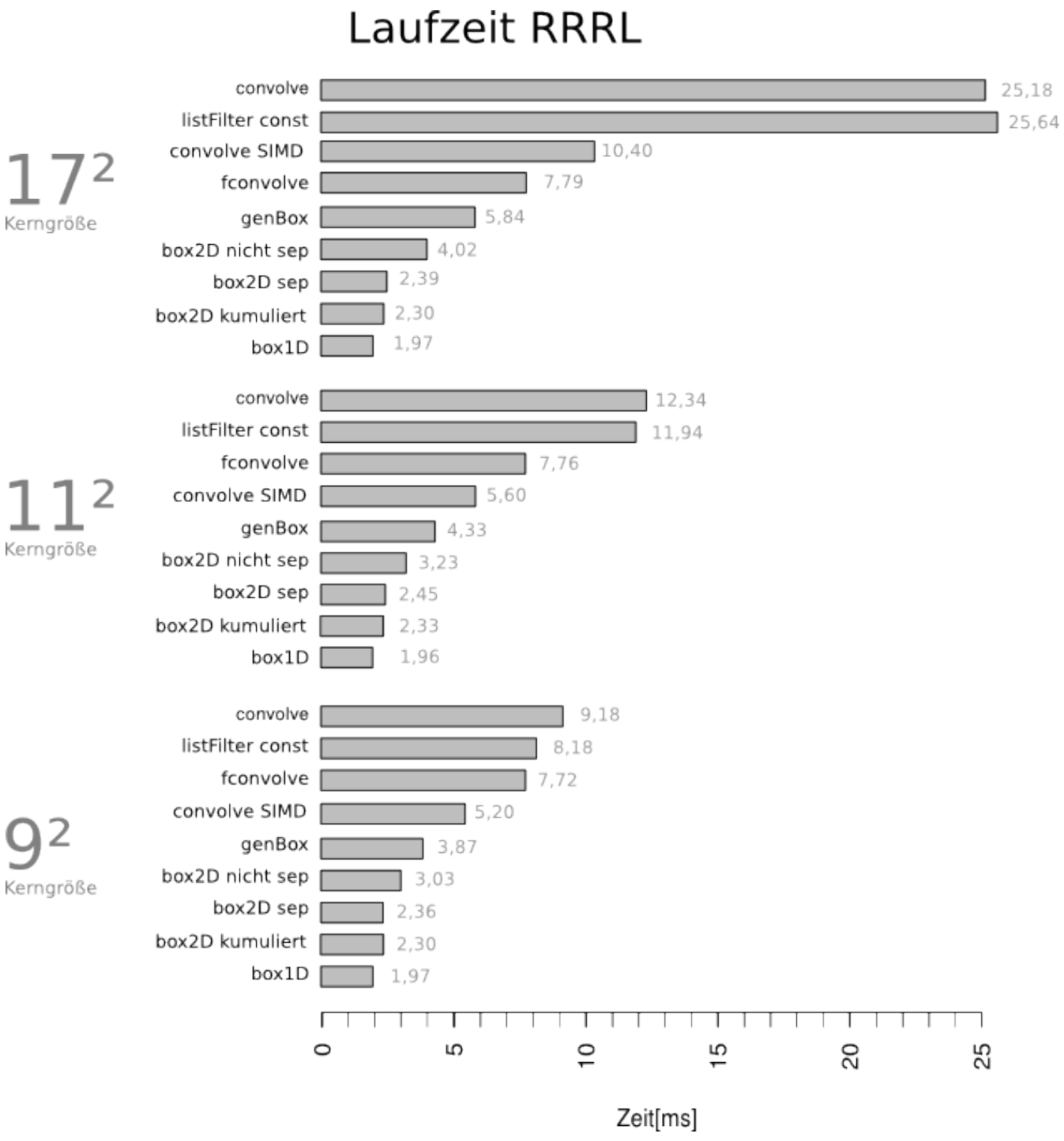

Abbildung 32: Zeitbedarf RRRL mit verschiedenen Faltungsroutinen, Parameter: 100 Iterationen, Alpha = 0.000001 , Epsilon = 0.1

Optimierung berechnet. Im Interfall $0.01 \ldots 0.09$ treten jeweils geringe qualitative Einbußen statt. Im Intervall $0.1 \ldots 0.5$ ist das Ergebnis visuell bereits deutlich verschlechtert. Hier sind immerhin deutliche Einsparungen in der Rechenzeit möglich und zwar bis zu einem Fünftel der ursprünglichen Rechenzeit (Tab. 2).
Die Tabelle 3 zeigt die Messreihe mit dem RRRL-Algorithmus. Als Regularisierungsparameter wurden gewählt: $\alpha = 0.000001, \epsilon = 0.1$.

Rechenaufwand der zusätzlichen Kontrollstrukturen - der Overhead. Die optimierte Version der Faltungsroutine beinhaltet eine if-Klausel. Diese Abfrage prüft, ob der Schwellwert des jeweiligen Pixels erreicht wurde. Diese Kontrollstrukturen bringen also einen Overhead mit sich. Zur Veranschaulichung beinhaltet hier jede Tabelle zwei Vergleichswerte der nicht optimierten Version:

- performRL/performRRRL: Hier wurde die ursprüngliche Version, ohne Optimierung und ohne Kontrollstrukturen gemessen (ohne Overhead)
- $k = -10$: Hier wurde die optimierte Version mit Kontrollstrukturen gemessen. Das k wurde so niedrig gewählt, dass alle Pixel in jeder Iteration voll gefaltet wurden (also mit Overhead).

Das Ergebnis zeigt, dass praktisch kein Overhead entsteht! Auffällig ist, dass die Zeiten der ursprünglichen Routine und des optimierten Algorithmus gleich sind. Das ist optimal, weil bereits bei einem kleinen k-Wert die Rechenzeit verkürzt wird.

Anzahl der ignorierten Pixel. Um ein Maß für die Optimierung zu finden wurde ausgewertet, wieviele Pixel bei der Faltung übersprungen, also nicht gefaltet worden sind. Die Werte stehen jeweils in der zweiten Spalte in den Tabellen 2, 3 und 4. Die Diagramme in Abb. 33 zeigen den Verlauf dieses Anteils. Der Verlauf des oberen Diagramms nähert sich auf den Endwert von 0.23 ein (siehe Tab. 4). Das heißt, dass das Verhältnis der gefalteten Pixel zu den ignorierten Pixeln bei 0.23 liegt. In anderen Worten kann man also sagen, dass hier nur ein Viertel aller Pixel tatsächlich berechnet wurden und die anderen ignoriert. Daraus entsteht laut Tabelle 4 aber dann eine Beschleunigung um das Fünffache.

Nähere Betrachtung bei 500 Iterationen. Der RL wurde nun bei 500 Iterationen getestet. Bei dieser Iterationszahl ist das Ergebnis der Schärfung bereits sehr deutlich zu sehen. Die Tabelle zeigt eine Übersicht der Messung.

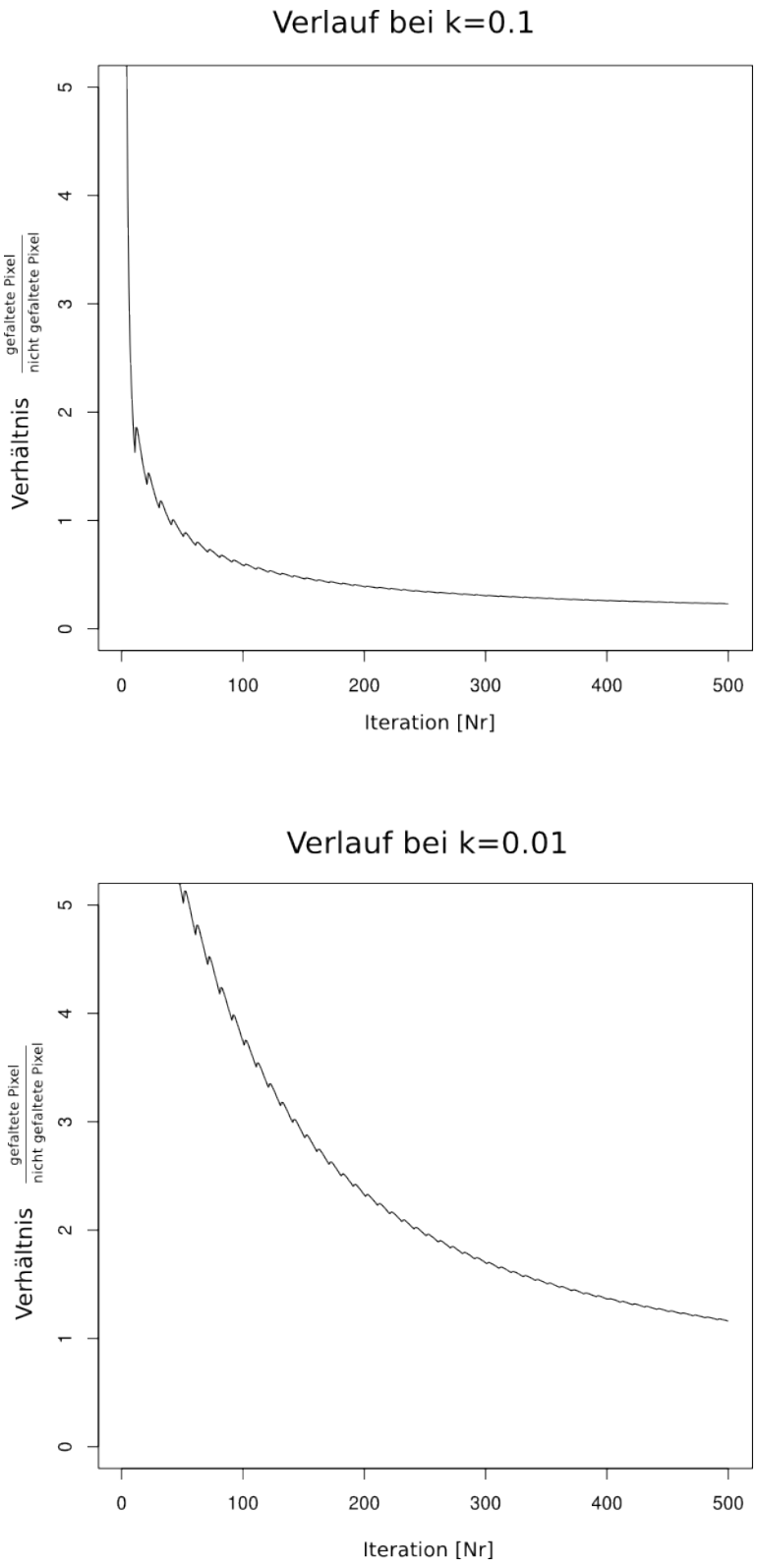

Abbildung 33: RL: Anteil gefalteter Pixel, 500 Iterationen, bei k=0.01 und k=0.1

k-Wert	Verhältnis Faltung	Zeit [s]	SNR[dB]	Speedup
performRL	1/0	15.43	17.25	1.0
-10	1/0	15.51	17.25	1.0
0.01	3.74	12.29	17.25	1.27
0.02	2.04	10.48	17.25	1.49
0.03	1.46	9.34	17.24	1.66
0.04	1.16	8.42	17.23	1.84
0.05	0.97	7.76	17.21	2.00
0.06	0.85	7.23	17.18	2.15
0.07	0.75	6.79	17.16	2.28
0.08	0.68	6.47	17.13	2.40
0.09	0.63	6.11	17.11	2.54
0.1	0.59	5.86	17.08	2.65
0.2	0.37	4.31	16.87	3.60
0.3	0.28	3.59	16.69	4.32
0.4	0.24	3.16	16.52	4.91
0.5	0.22	2.89	16.37	5.37
blurred image		12.86		

Tabelle 2: RL mit Optimierung: Die Laufzeiten für verschiedene k-Werte und Signal-Rausch-Verhältnisse, 100 Iterationen

5 Diskussion

In der Arbeit wurden neue Methoden zur Faltung mit speziellen Faltungskernen entwickelt. Diese Faltungsroutinen dienen als Werkzeug für die Berechnung iterativer Dekonvolutionsverfahren, und zwar des Richardson-Lucy [14], [7] und des robust regularisierten Richardson-Lucy-Algorithmus [4]. Ebenso konnte eine Möglichkeit gefunden werden, bei der sich die Berechnung auf die relevanten Pixel begrenzen läßt. So kann Rechenzeit gespart und der Dekonvolutionsalgorithmus beschleunigt werden. Im Folgenden werden die Vor- und Nachteile der entwickelten Verfahren aufgezeigt.

5.1 Eigenschaften der entwickelten Faltungsroutinen

Bei den neu entwickelten Filtern wurden spezielle Faltungskerne zugrunde gelegt. Die Methoden wurden auf einer Plattform getestet, sie wurden aber nicht hardwarespezifisch optimiert. Das bedeutet, dass der Einsatz weniger von der verwendeten Hardware abhängig ist, sondern vielmehr vom vorliegenden Faltungskern. Mit dem SIMD-optimierten Ortsfilter wurde außerdem eine Methode implementiert, die eine parallelisierte Berechnung ermöglicht. Der Vorteil der Parallelisierung ist, dass der Filter vom Faltungskern unabhängig ist. Jedoch ist für die Implementierung der parallelisierten Berechnung eine bestimmte Hardwareplattform (mit mehreren Rechenkernen bzw. speziellen Befehlssätzen wie SSE) notwendig. Wenn die Implementierung dann aber an die jeweilige Hardwareplatt-

k-Wert	Verhältnis Faltung	Zeit [s]	SNR[dB]	Speedup
perform RRRL	1/0	24.79	16.67	1
-10	1/0	24.83	16.67	1
0.01	4.36	20.55	16.67	1.20
0.02	2.41	18.08	16.66	1.37
0.03	1.72	16.50	16.66	1.50
0.04	1.39	15.12	16.65	1.63
0.05	1.18	14.34	16.64	1.72
0.06	1.04	13.46	16.61	1.84
0.07	0.94	12.90	16.59	1.92
0.08	0.86	12.36	16.56	2.00
0.09	0.80	11.90	16.53	2.08
0.1	0.76	11.66	16.52	2.12
0.2	0.53	9.64	16.37	2.57
0.3	0.46	8.81	16.29	2.81
0.4	0.42	8.38	16.21	2.95
0.5	0.39	8.03	16.10	3.08
blurred image			12.86	

Tabelle 3: RRRL mit Optimierung: Die Laufzeiten für verschiedene k-Werte und Signal-Rausch-Verhältnisse, 100 Iterationen, $\alpha = 0.000001, \epsilon = 0.1$

k-Wert	Verhältnis Faltung	Zeit [s]	SNR[dB]	Speedup
perfornRL	1/0	77.5	19.33	1
-10	1/0	77.9	19.33	1
0.01	1.15	42.2	19.29	1.85
0.1	0.23	15.2	18.40	5,13
blurred image			12.86	

Tabelle 4: RL bei 500 Iterationen.

form angepasst wird, sind sehr effiziente Implementierungen möglich, wie der Vergleich der Ortsfaltung mit der SIMD-optimierten Ortsfaltung gezeigt hat. Die meisten hier entwickelten Verfahren sind jedoch nicht weiter parallelisierbar. Deshalb sollte zuerst die Verwendung der gewöhnlichen Ortsfaltung und der Faltung im Fourierbereich mit angemessener Anpassung an die Hardware in Betracht gezogen werden. Falls eine aufwendige Anpassung an die Hardware nicht möglich oder gewünscht ist, kommt dann aber eine der folgenden Methoden in Frage:

Mit dem generischen Boxfilter wurde eine Methode gefunden, mit der sich verschiedene Faltungskerne effizient berechnen lassen. Die Rechengeschwindigkeit ist bei einer Kerngröße von $17 \cdot 17$px höher als die implementierte Faltung im Fourierbereich. Die Voraussetzung für die Verwendung des generischen Boxfilters ist, dass die Pixel des zugehörigen Faltungskerns in einer Zeile zusammenhängen und konstant sind. Dann lassen sich verschiedene Kerne falten: Eine typische Anwendung wird der Lensblurfilter sein. Der Kern des Lensblurfilter

Abbildung 34: RL ohne Optimierung, 500 Iterationen

ist ein gefüllter Kreis und modelliert die Unschärfe durch Defokussierung. In der Fotografie kommt das Problem der fehlerhaften Fokussierung sehr häufig vor und könnte deshalb eine typische Anwendung sein. Häufig ist die Form der Blende nicht exakt kreisförmig. Auch diesen Fall kann der Filter abdecken und Faltungen mit konvexen Polygonen berechnen.

Der entwickelte Listenfilter kann mit allen Faltungskernen umgehen. Allerdings bringt er einen Mehraufwand in der Berechnung mit sich. Die Verwendung des Filters lohnt sich erst, wenn weniger als halb so viele Pixel des Kerns besetzt sind (Der gemessene Overhead ist etwa 50% gegenüber der Ortsfaltung). Ein typischer Anwendungsfall für diesen Listenfilter kann die unregelmäßige Bewegungsunschärfe sein. Hier kann der Filter seine Stärke ausspielen, weil die PSF der Bewegungsunschärfe idR. dünn besetzt ist.

Mit dem kumulierten 2D-Boxfilter wurde eine Methode entwickelt, mit der sich der Boxfilter noch schneller berechnen läßt. Der kumulierte Boxfilter ist knapp schneller als der separierte. Dieser Zeitvorteil kann bei steigender Bildgröße noch deutlicher hervortreten.

5.2 Vor- und Nachteile der konvergenzorientierten Optimierung

Die iterativen Dekonvolutionsverfahren RL und RRRL konnten durch Selektion einzelner relevanter Pixel beschleunigt werden. Dabei konnte bei 500 Iterationen die Rechengeschwindigkeit um das Fünffache erhöht werden. Natürlich bringt das eine Verschlechterung des Ergebnis mit sich. Dennoch kann bei der Wahl des geeigneten Parameter ein Kompromiss zwischen Qualität und Performance individuell gewählt werden. Für manche Anwendung kann es sinnvoll sein, eine rasche und grobe Vorberechung auszuführen. Gleich-

Abbildung 35: RL mit Optimierung, 500 Iterationen, $k = 0.01$

zeitig ist die Beschleunigung ohne großer Qualitätseinbußen um den Faktor 2 möglich. Die Anwendungen können also vielfältig sein. Seine besondere Stärke spielt die konvergenzoptimierte Version in Verbindung mit dem Listenfilter aus. Speziell der Listenfilter ermöglicht die isolierte Faltung einzelner ausgewählter Pixel. Der konvergenzoptimierte RL/RRRL mit dem Listenfilter ausgeführt kann demnach Beschleunigungen über das Fünffache gegenüber der gewöhnlichen Ortsfaltung bringen.

Abbildung 36: RL mit Optimierung, 500 Iterationen, $k = 0.1$

6 Zusammenfassung/Abstract

Deutsch

Itertive Dekonvolutionsverfahren ermöglichen die Rekonstruktion von unscharfen Bildern. Häufige Ursachen von Unschärfe sind ungünstige Lichtverhältnisse, zu schnelle Bewegung oder fehlerhafte Fokussierung. Wenn die genaue Form und das Ausmaß der Unschärfe bekannt sind, kann das Bild mit Dekonvolutionsalgorithmen geschärft werden. Die iterativen Verfahren wie der Richardson-Lucy-Algorithmus oder der robust regularisierte Richardson-Lucy-Algorithmus sind jedoch rechenintensiv und benötigen oft mehrere hundert Iterationen, um ein hochwertiges Ergebnis zu berechnen. Die meiste aufgewendete Rechenzeit wird dabei in die Faltung der Zwischenergebnisse investiert. Dazu wurden nun mehrere Verfahren vorgestellt, die eine effiziente Berechnung bei speziellen Faltungskernen ermöglichen. Diese speziellen Faltungskerne stehen in direktem Bezug zu der entstandenen Unschärfe: Der vorgestellte Lensblur-Filter ermöglicht so die effiziente Dekonvolution von defokussierten Bildern. Die Form der Filtermaske kann dabei an die exakte Form der Blendenöffnung angepasst werden. Der Listenfilter bietet eine schnelle Berechnung von unregelmäßigen Bewegungsunschärfen. Speziell, wenn die Unschärfe diagonal verläuft, erzielt der Filter die beste Performance. Der Rechenzeit ist dabei linear abhängig von der Anzahl der besetzten Pixel, mit einem gemessenen Overhead von 50%. Das bedeutet bei einem diagonal verlaufenden Unschärfekern von $10 \cdot 10$px eine Beschleunigung um mehr als den Faktor 6 im Vergleich zu der Faltung im Ortsbereich. Neben der Optimierung der Faltungsopertionen wurde ein weiterer Ansatz verfolgt: Bei der Iteration des

Richardson-Lucy (RL) und robust regularisierten RL ändern sich manche Pixel stärker. Andere Pixel werden durch die Dekonvolution kaum verändert. Das sind etwa Pixel, die in einer homogenen Umgebung liegen. Im vorgestellten Verfahren wurden anhand zweier verschiedener Maßzahlen (Varianz und Grauwertdifferenz) die relevanten Pixel selektiert. Die Berechnung wurde nur auf die selektierten Pixel angewandt. So konnte die Rechenzeit um mehr als den Faktor 5 verringert werden. Dabei ist es möglich, einen Kompromiss zwischen Qualität und Effizienz durch entsprechende Wahl des Schwellwerts zu finden.

English Iterative deconvolution methods allow to reconstruct unsharp images. Frequent causes of unsharp images are bad lighting conditions, fast movement or wrong focussing. These unsharp images can be sharpened with deconvolution algorithms, if the blurring parameters are known. The deconvolution algorithms Richardson-Lucy and the robust regularized Richardson Lucy are such methods. But for a high quality result, they require over hundred iterations, so that the calculation time becomes the limiting factor. The main time is spent on convolution. This thesis show some new methods for convolving images with special convolution kernels. The kernels are directly associated to the type of blurring, for example the lens blur kernel. It describes the blur, that comes along with images which have a wrong focus. The developed filter performs a convolution with this lens blur kernel even faster than the convolution in frequency domain with a FFT. The fast lens blur filter can be adopted to the exact shape of the aperture. Another special filter that was introduced is the so called list filter. It allows the fast convolution with kernels, in which most grey values are zero. Especially with motion blurs that drifts diagonally, this filter is most effective. The calculation time of the list filter depends on the number of kernel pixels that are not null. So in case of a diagonal motion blur, the list filter only calculates 10 Pixel instead of $10 \cdot 10$ Pixel. This causes an acceleration of six times with an already included calculation overhead of 50%.

With the mentioned convolution methods an optimized version of the Richardson-Lucy (RL) and the robust regularized RL algorithms have been introduced, with the following approach: Some pixels may not be changed during the convolution. The neighbor pixels of that ones are mostly uniform. The introduced technique performs the convolution only on the affected pixels. That selected pixels have been identified with two values: The variance and the difference of the grey values. So the calculation time could be accelerated with a factor of 5 compared to a standard space domain convolution. This optimization of course affects the quality. But the user can find a trade-off in quality and performance, simply with adjusting the threshold.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literatur

[1] Wikipedia Commons Brion Vibber. Circles_of_confusion_lens_diagram.png, 2005.

[2] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. How to write fast numerical code: A small introduction. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 196–259. Springer Berlin Heidelberg, 2008.

[3] Dr. Roland Mühle. Physikalische und technische grundlagen der fotografie. *Vorlesungsskript*, 2000.

[4] Ahmed Elhayek, Martin Welk, and Joachim Weickert. Simultaneous interpolation and deconvolution model for the 3-d reconstruction of cell images. In Rudolf Mester and Michael Felsberg, editors, *Pattern Recognition*, volume 6835 of *Lecture Notes in Computer Science*, pages 316–325. Springer Berlin / Heidelberg, 2011.

[5] Intel. Intel® pentium® processor p6100 (3m cache, 2.00 ghz). `http://ark.intel.com`, 2012. [Online; accessed 1-Sep-2012].

[6] Anat Levin, Rob Fergus, Frédo Durand, and William T. Freeman. Image and depth from a conventional camera with a coded aperture. *ACM Trans. Graph.*, 26(3), July 2007.

[7] Lucy. An iterative technique for the rectification of observed distributions, June 1974.

[8] Welk Raudaschl Schwarzbauer Erler Läuter. fast and robust image deconvolution, submitted. 2012.

[9] M.J. McDonnell. Box-filtering techniques. *Computer Graphics and Image Processing*, 17(1):65 – 70, 1981.

[10] Charles Petzold. *Programming Windows, 5th Edition*. Microsoft Press, 2011.

[11] Jef Poskanzer. specification pgm. 1991. http://netpbm.sourceforge.net/doc/pgm.html.

[12] GNU project. Gcc, the gnu compiler collection. `gcc.gnu.org`, 2011. [Online; accessed 2-Sep-2012].

[13] Jack Tumblin Ramesh Raskar, Amit Agrawal. Coded exposure photography: Motion deblurring using fluttered shutter. *Mitsubishi Electric Research Laboratories*, 2006.

[14] William Hadley Richardson. Bayesian-based iterative method of image restoration. *J. Opt. Soc. Am.*, 62(1):55–59, Jan 1972.

[15] Felsberg Skoglund. Fast image processing using sse2, 2005.

[16] E. Pelikan R.Repges T. Lehmann, W. Oberschelp. *Bildverarbeitung fur die Medizin.* Springer-Verlag, 1997.

[17] Norbert Wiener. *Extrapolation, Inerpolation, and Smoothing of Stationary Time Series.* The M.I.T. Press, 1975.

Lebenslauf / CV

(kurzer tabellarischer Lebenslauf, max. eine Seite) (Am Ende der gesamten Arbeit:) Hiermit erkläre ich an Eides statt, die Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet zu haben.