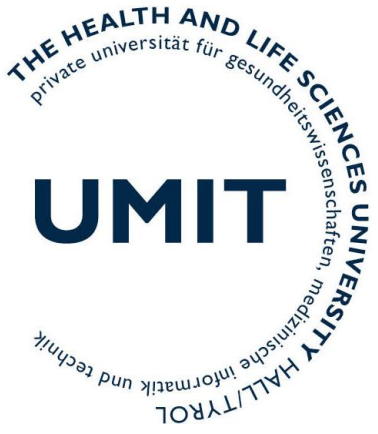

Algorithmische Optimierungen für iterative Dekonvolutionsverfahren

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 **4**

1.1 Dekonvolution in der Bildverarbeitung 4

1.2 Modellierung von Bildern und Unschärfe 4

1.2.1 Bildmodellierung 4

1.2.2 Bewegungsunschärfe 5

1.2.3 Boxfilter 2D 6

1.2.4 Lensblur, Defokussierung 7

1.2.5 Allgemein dünn besetzter Kern 8

1.3 Dekonvolutionsalgorithmen 9

1.4 Übliche Faltungsmethoden 10

1.5 Ansatz für Optimierungen 11

2 Zielsetzung **12**

2.1 Filterdesign 12

2.2 Analyse des Konvergenzverhaltens 13

3 Methoden **14**

3.1 Softwareumgebung 14

3.1.1 System 14

3.1.2 Compiler und Linker 14

3.1.3 Dateiformat 15

3.1.4 Datenorganisation und Pixelzugriff 16

3.2 Verwendete Hardware 16

3.3 Filterdesign 18

3.3.1 1D Box 18

3.3.2 2D Box 20

3.3.3 Lensblur, generischer Boxfilter 21

3.3.4 Listenfilter, allgemein dünn besetzte Kerne 24

3.3.5 Optimierung durch Paralellisierung: SIMD 24

3.4 Analyse des Konvergenzverhaltens 26

4 Ergebnisse **31**

4.1 Messergebnisse Faltungsoptimierung 31

4.1.1 Qualitative Betrachtung 31

4.1.2 Performancemessung 32

4.2 Messergebnisse der Konvergenzanalyse 41

5 Diskussion **46**

5.1 SIMD 46

6 Zusammenfassung/Abstract **47**

Abbildungsverzeichnis **51**

Tabellenverzeichnis **52**

Literaturverzeichnis **54**

Eidesstattliche Erklärung **55**

1 Einleitung und Stand der Forschung

1.1 Dekonvolution in der Bildverarbeitung

Dekonvolutionsverfahren in der Bildverarbeitung haben zum Ziel, Unschärfe in Bildern zu beseitigen. Es gibt verschiedene Arten von Unschärfe in Bilder. Viele davon sind systembedingt und hängen zusammen mit den optischen Parametern, Aufnahmedauer und Film- bzw. Sensorempfindlichkeit. Etwa bei der Bewegungsunschärfe bewegt sich das gesamte Bild oder Teile davon während der Aufnahmezeit. Die Defokussierungsunschärfe kann ebenso den gesamten Bildausschnitt oder nur Teile daraus betreffen. In diesem Falle wurden die optischen Parameter falsch gewählt...

1.2 Modellierung von Bildern und Unschärfe

1.2.1 Bildmodellierung

Bild als Funktion. Nach der Abtastung (Sampling) läßt sich das digitale Bild als Funktion beschreiben: Dabei stellt Ω den Bildbereich oder die Bildebene dar.

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

Sie ist bereits diskretisiert und besteht aus Pixeln. R stellt den Wertebereich dar. Er ist ebenso diskret und wird bei uns auf 8 bit, also einen Bereich von $0 \ldots 255$ beschränkt. Wir beschränken uns auf monochrome Bilder. Ein Wert aus R stellt einen Grauwert dar. Ein Grauwertbild wird dann als Funktion f beschrieben.

Faltung. Wir führen in Zusammenhang mit der diskreten Bildebene den 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 $H(x)$ 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)
- Ω Bildebene

Abbildung 1: Ausgangsbild

1.2.2 Bewegungsunschärfe

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

Ortsvariante, -invariante Bewegungsunschärfe. Die Unschärfe kann ortsabhängig oder auch ortsinvariant sein. Bei der ortsinvarianten Unschärfewerden alle Pixel im Bild gleich verwischt. Praktisches Beispiel ist die Bildaufnahme ohne Stativ: Die Kamera bewegt sich während der Bildaufnahme (Bildrotation 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.

In Bezug auf die später verwendeten Filter ist festzuhalten: Wenn sich diese PSF in eine Richtung ausdehnt sprechen wir in der Folge von einem eindimensionalen Faltungskern.

Liegt eine gleichmäßige, lineare Bewegung in x-Richtung vor, so entspricht das dem Ergebnisdes 1D-Boxfilters (siehe Abb. 2). Wir beschränken uns auf eine ortsinvariante Dekonvolution.

Abbildung 2: 1D Boxfilter mit 17 Pixel Ausdehnung in horizontaler Richtung, rechts unten: der Faltungskern (vergrößert)

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 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. **Ortsvariant?** Da die Schärfeebene im Bild variieren kann, resultiert daraus eine mögliche ortsvariante PSF. Eine ortsvariante Dekonvolution stellt eine große Herausforderung dar und setzt eine Tiefeninformation voraus. Wir werden nur die ortsvariante Dekonvolution behandeln und untersuchen eine mögliche Implementation der Faltung mit der Lensblur-PSF.

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

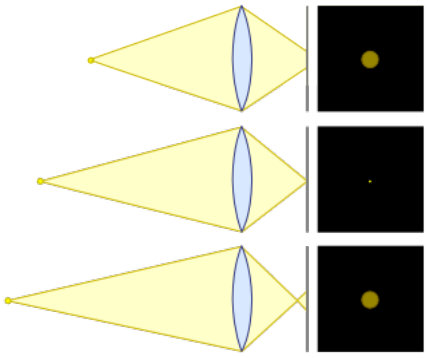

Abbildung 5: Kern und Unschärfe. Illustration aus Wikipedia Comons [2]

1.2.5 Allgemein dünn besetzter Kern

Mit dem nachfolgend beschriebenen Listenfilter können Faltungen mit allgemein dünn besetzten Kernen berechnet werden. Dabei kann jeder Kern verwendet werden. Aber besonders Kerne, die viele Pixel mit einem Grauwert von null aufweisen sind hier gemeint. Die Abbildung 6 zeigt ein Beispiel eines solchen Kerns.

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

1.3 Dekonvolutionsalgorithmen

Inverse Filterung, Wiener Filter. Dem Faltungssatz (9) folgend, kann die Faltungsoperation (3) im Fourier bereich mit einer Division invertiert werden: $\hat{u} = \frac{\hat{f}(\omega)}{\hat{h}(\omega)}$. Die Bildstörungen $n(x)$ werden dabei vernachlässigt. Und ein Problem tritt auf: Die Punktbildfunktion besitzt Nullstellen. Das führt uns zum Wiener Filter.

Der Wiener Filter[14] ist dem inversen Filter ähnlich, doch erscheint der Faktor $\bar{\hat{h}}$. Das ist der komplex konjugierte Faltungskern im Fourierbereich. Ebenso wurde der Divisor zu $|\hat{h}|^2 + K$, wobei das K nun als Regularisierer hinzugefügt wurde. Je kleiner K gewählt wird, desto stärker wird der Schärfungseffekt. Gleichzeitig wird aber das Rauschen verstärkt, basierend auf dem Gauss'schen Rauschmodell. Der Wiener Filter ist keine iterative, sondern eine lineare Methode. Obwohl laut Zielsetzung (Kapitel 2.1) nur iterative Methoden behandelt werden, sei der Wiener Filter hier kurz erwähnt, da er hinsichtlich der Laufzeit äußert effizient implementiert werden kann.

$$\hat{u} = \frac{\hat{f} \cdot \bar{\hat{h}}}{|\hat{h}|^2 + K} \tag{4}$$

Richardson-Lucy [12, 6] Der RL Algorithmus ist eine Fixpunktpunktiteration. In der Folge RL genannt.Er basiert auf einer Likelihood-Schätzung und basiert auch dem Poisson Rauschmodell. Der Algorithmus erfordert einen Parameter: die Anzahl der Iterationen. Als Anfangswert wird gewält: $u^0 = f$. RL erfordert positive Grauwerte des Eingangsbildes und arbeitet während der Iteration auch positivitätserhaltent. Es zeigt sich ein semikonvergentes Verhalten. Dh. dass die Schärfe nach einer Zahl von Iterationen zunimmt. Jedoch wird das Rauschen verstärkt und führt nach einer Konvergenzphase der Schärfe zur Divergenz hinsichtlich des Rauschverhaltens.

$$u^{k+1} = \left(h^* * \left(\frac{f}{u^k * h} \right) \right) \cdot u^k \tag{5}$$

Robust Regularisierter Richardson-Lucy [3] Ein Variationsansatz[16] bringt uns zur Notation (6). Hier ist ein Enerigefunktional notiert, das einen Datenterm und einen Glattheitsterm beinhaltet. Der Datenterm bestraft Abweichungen vom Modell $f = u * h$. Der Glattheitsterm bestraft Kontraste und führt damit zu einer Glattheit des Bildes. Der Glattheitsterm wird nun auf $\Phi(u*h - f ln \frac{u*h}{f})$ gesetzt. Und die Minimierung des Funktionals erfolgt durch eine Fixpunktiteration. Das führt uns zum Robust Regularisierten Richardos Lucy [3] Algorithmus. Siehe Gleichung (7). In der Folge RRRL genannt.

$$E[u] = \underbrace{\int_\Omega (f - u * h)^2 dx}_{Datenterm} + \underbrace{\alpha \int_\Omega |\nabla u|^2 dx}_{Glattheitsterm} \tag{6}$$

$$u^{k+1} = \frac{(\Phi'(r_f(u^k * h))\frac{f}{u^k * h}) * h^* + \alpha[\mathrm{div}(\Psi'(|\nabla u^k|^2)\nabla u^k)]_+}{\Phi'(r_f(u^k * h)) * h^* - \alpha[\mathrm{div}(\Psi'(|\nabla u^k|^2)\nabla u^k)]_-} u^k \tag{7}$$

1.4 Übliche Faltungsmethoden

Die zur Iteration bei RL (5)und RRRL (7) nötige Vorwärtsfaltung, dh. die Faltung mit dem bekannten Faltungskern der zur Unschärfe geführt hat, wird allgemein mit der Ortsfaltung oder der Faltung im Fourierbereich durchgeführt.

Ortsfaltung Die Faltungsfunktion (2) wird im Diskreten zur Gleichung (8). Um ein Pixel des Ausgangsbildes zu erhalten müssen also die umliegenden Pixel des Eingangsbildes mit den Kernpixel multipliziert werden. Für die Berechnung des vollen Bildes resultiert daraus eine Laufzeit von $\mathcal{O}(N^2K^2)$. Die Abb.**??** schematisiert die Berechnungsvorschrift.

$$(f * h)[n] = \sum_{m=0}^{N} f[n-m] \cdot h[m] \tag{8}$$

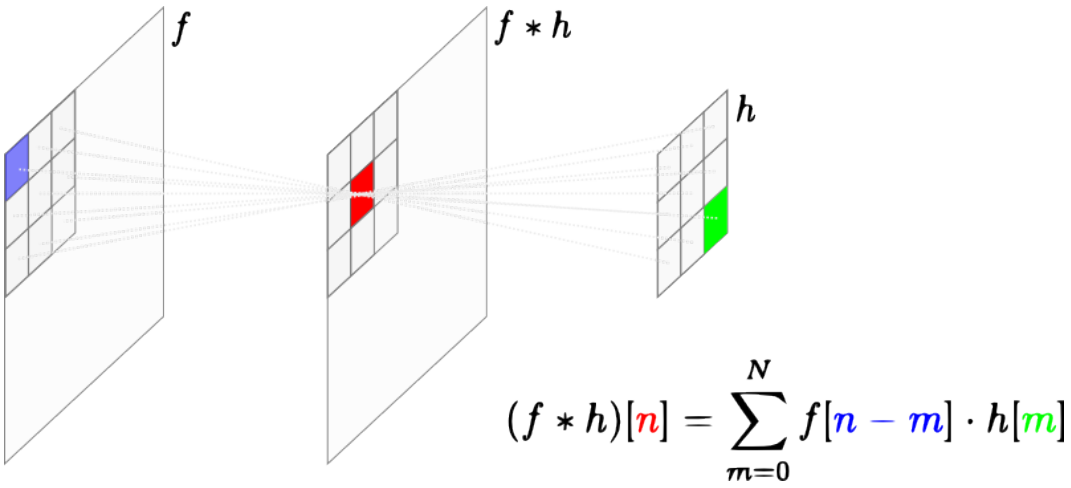

Abbildung 7: Illuistration der Faltung im Ortsbereich

Faltung im Fourierbereich. Die Faltung im Frequenzbereich macht sich das Faltungstheorem zu Nutze. Siehe dazu (9). 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.

$$f(x) * h(x) \rightarrow F(u) \cdot H(u) \tag{9}$$

Randbehandlung.*bei Ortsfaltung.*Bei der Faltung im Ortsbereich ist nicht klar, wie man an den Bildrändern faltet. Siehe dazu Abb. 7. Wenn das Pixel außerhalb des Bildbereichs liegt wählen wir das nächstliegende Randpixel. Wenn also unsere Bildebene Ω von

(0,0) bis (m,n) wird so gewählt: Bei Pixel(6,-7) wird das Pixel(6,0) am oberen Bildrand gewählt. Bei Pixel(m+1,n) wird das Pixel(m,n), also das Pixel in der rechten unteren Bildhälfte gewählt. Diese Methode wird in der Folge „Randbehandlung durch Umbiegen genannt“ und wurde mit der Funktion „getPixelIndexBend(...)“ implementiert.

bei Fourierfaltung. 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.

1.5 Ansatz für Optimierungen

Da es nun in dieser Arbeit um die Performance von RL und RRRL geht, betrachten wir kurz die Ausgabe des Profilers. Der RL wurde unter Verwendung des Gnu Profiler untersucht [9]. Das Profiling weist darauf hin, dass über 90% der Rechenzeit für die Faltung aufgewendet werden. Es ist also naheliegend, primär die Faltungsoperation zu optimieren.

Faltung optimieren. In diesem Zusammenhang bieten sich zwei Wege an: Entweder es werden spezielle Faltungsroutinen für spezielle Kerne gesucht, oder die allgemeinem Faltungsroutinen werden optimiert. Etwa die Methode des Boxfilter wurde bereits im Jahre 1981 beschrieben [8] und konnte im Zusammenhang mit RL und RRRL angewendet werden [7]. Diese Arbeit verfolgt mit ihrer Zielsetzung ebenso den Weg der speziellen Kerne. In Kapitel 2.1 werden solche Kerne ausgewählt und die Methoden in Kapitel 3.3 beschreiben die Implementation.

Wenn man eine spezielle Hardwareplattform zugrunde legt, ergeben sich für allgemeine Faltungskerne technische Optimierungen. Unter Einsatz eines plattformspezifischen Befehlssatzes sind besondere Rechenoperationen zugänglich. SSE ist eine solche Technik und ermöglicht die Verarbeitung von mehreren Dateneinheiten in einem Rechenschritt (SIMD - single instruction, multiple data). In [13] wurden verschiedene Faltungskerne bereits implementiert. In dieser Arbeit wurde die allgemeine Faltung im Ortsbereich mit SSE implmentiert und nutzt dabei die „X86 Built-in Functions“. Unter [11] ist die Dokumentation dieser Wrapperfunktionen zugänglich.

Konvergenz analysieren. Ein anderer Optimierungsansatz bietet sich, wenn man das Konvergenzverhalten von RL und RRRL betrachtet. Es ist zu erwarten, dass sich manche Pixel während der Iteration kaum ändern. Trotzdem wird aber die volle Faltung ausgeführt. Wenn sich Pixel von der Operation auschließen ließen, könnte dadurch Rechenzeit eingespart werden.

```
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 8: Ausgabe von gprof zum RL mit Ortsfaltung

2 Zielsetzung

Für die iterative Dekonvolutionsalgorithmen in Kapitel 1.3 und den in Kapitel 1.2.2 gezeigten Faltungskernen sollen schnelle Verfahren entwickelt werden. In diesem Zusammenhang sollen zwei Wege verfolgt werden: Die Optimierung der Faltungsoperation (2.1) und die Optimierung hinsichtlich des Konvergenzverhaltens einzelner Pixel (2.2).

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

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

Der 2D Boxfilter kann auf mehrere Arten implementiert werden: entweder durch Hintereinanderausführen von horizontalem und vertikalem 1D Filter, oder aber durch Differenzenbildung am kumulierten Grauwertbild. Hier sollte die schnellere Variante gefunden werden.

Weiter soll der kreisrunde Kern untersucht werden. Dieser Kern modelliert eine Fokusunschärfe. Ähnlich wie beim Boxfilter stellt die Berechnung über laufende Summen einen

Ansatz dar. Der Kreis kann aus einem vorberechneten Array generiert werden. Gesucht ist die Zeitersparnis im Vergleich zur vollen Faltung im Ortsbereich.

Zu den dünn besetzten Kernen: Angenommen, ein Kern besteht aus m Pixeln. Davon haben n Pixel einen Grauwert größer null. Wenn nun n viel kleiner als m ist, kann man von einem dünn besetzten Kern sprechen. Um eine Faltung auszuführen, sind hier nicht m, sondern n Rechenschritte notwendig. Hinzu kommen jedoch Kontrollstrukturen, die einen höheren Rechenaufwand für jedes besetzte Pixel nach sich ziehen. Gesucht ist hier die Zeitersparnis im Vergleich zur vollen Faltung im Ortsbereich, abhängig von der Zahl n.

2.2 Analyse des Konvergenzverhaltens

Während der Iterationsschritte konvergieren manche Pixel schneller, manche weniger schnell. Ziel ist es, ein Maß für diese Konvergenz zu finden. Diese Pixel 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 prozeduale 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 mit C99 Erweiterung.

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. Auf das Linken von Objectfiles konnte durch die Verwendung von wenigen C-Files verzichtet werden. Der Aufruf erfolgt so:

```
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 Datei Ein- und Ausgabe sowie die Algorithmen aufgerufen
- **pgmio.c** stellt die Datei Ein- und Ausgabe zur Verfügung
- **functions.c** stellt alle Algorithmen zur Verfügung

Der Schalter „-O2“ bewirkt die Optimierung mit Level 2 und faßt damit verschiedene Compiler-Schalter zusammen [11]. In der Entwicklung kam zusätzlich Eclipse mit der Erweiterung CDT (C/C++ Development Tooling) zum Einsatz. Entsprechende Projectfiles liegen vor.

Libraries: Zum Einsatz kommen mit einer Ausnahme nur Standardlibraries.

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

```
...
// 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

Zu time.h: Benödigt von gettimeofday(). Ein Windows-Äquivalent zeigt uns die WIN32 API. Erreichbar unter http://msdn.microsoft.com/. Die Linuxfunktion kann durch „GetSystemTime(..)“ und Einbinden von „Windows.h“ ersetzt werden.

3.1.3 Dateiformat

Die Bilder sollen in einem unkomprimierten Format eingelesen werden. Falls möglich sollen die Bilddaten ohne proprietäre Bibliothek eingelesen werden. Wir beschränken uns auf Bilder in Graustufen.

pgm - portable greymap Das Portable Greymap Format stellt solch ein unkomprimiertes einfach zu handhabendes Bildformat dar. Die Spezifikation ist unter [10] 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.

```
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

3.1.4 Datenorganisation und Pixelzugriff

Die Bilder wurden in eindimensionalen Arrays gespeichert. Dabei wird ein zusammenhängender Speicherblock allokiert. Siehe dazu Listing 3. Um darin auf ein einzelnes Pixel zuzugreifen, muss der Index anhand der Spalte und Zeile rückgerechnet werden. Um eine effitiente Implementierung zu gewährlisten, sollte einen Adresssprung auf eine andere Funktion möglichst vermieden werde. Die Berechnung des Index sollte also möglichst „inline“ erfolgen. Dazu kann eine Funktione zur Indexberechnung entweder als Inline-Funktion deklariert werden, oder die Berechnung direkt in der Index-Klammer erfolgen. 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. Laut [4] ist die Typkonvertierung von int zu float, also das sog. Casting mit Rechenaufwand verbunden. Gleichzeitig bieten x86-Przessoren effiziente Gleitkomma-Register zur Berechnung. Die Verwendung von int brachte keine nennenswerte Beschleunigung. Als Datentyp wurde durchgängig float verwendet.

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

Optimierung beziehen sich auf diese CPU, welche über zwei Kerne verfügt. Implementiert wurden jedoch keine paralellisierten 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 Cach-Aufteilung berücksichtigt werden. Unter Umständen ergeben sich nicht lineare 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 ermöglicht eine besonders effiziente Implementierung: [8]. Dieser Filter wurde bereits im Zusammenhang mit Dekonvolutionsalgorithmen untersucht: [7]. 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 (Komplexität linear zur Kerngröße und Inputbild). Durch die Größe des Boxfilters (bei einem 1D Boxfilter: $1 * k$ oder $k * 1$) ergibt sich ein laufendes Fenster. Die Summe dieses laufenden 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 (Figure 9 mit „+“ markiert). Das überlappende Pixel an der auslaufenden Kernseite wird subtrahiert. (Figure 9mit „−“ markiert) Die Komplexität des Boxfilters in 1D beträgt: todo:Komplexität Formel.

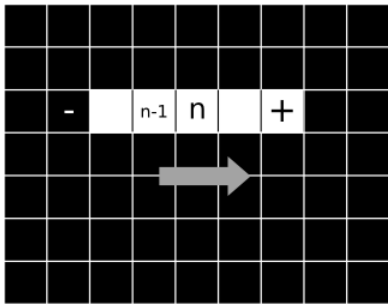

Abbildung 9: Illustration des 1D Boxfilters

```
  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 $m * n$ Pixel kann wie der 1D Boxfilter effizient implementiert werden. Implementiert wurde der 2D Boxfilter als separierter 1D Boxfilter, nicht separierte 2D Filter und kumulativer Boxfilter.

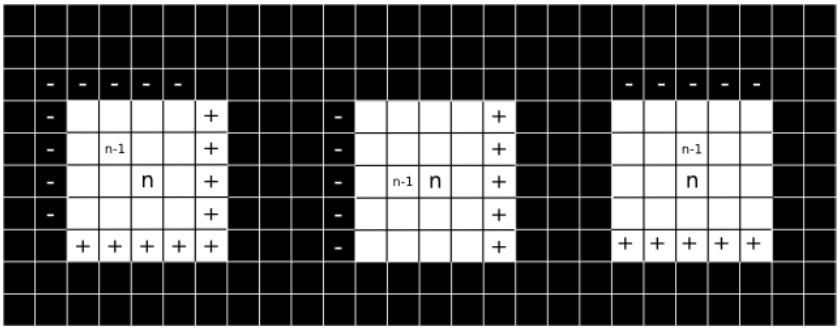

Abbildung 10: Illustration des 2D Boxfilters

Nichtseparierter 2D Boxfilter Der Boxfilter kann für jedes Pixel Zeilen- oder Spaltenweise berechnet werden. Der Pseudocode ist im Listing ersichtlich.

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

Listing 5: Pseudocode nicht separierter Boxfilter Boxfilter

Separierter 2D Boxfilter. Der 2D Boxfilter ist separierbar. So kann die Faltung in zwei Schritten berechnet werden:
$v = (u \otimes h_1) \otimes h_2, dim(h_1) = m \cdot 1, dim(h_2) = 1$

Kumulierter 2D Boxfilter. Grauwerten gebildet werden. Dabei entspricht der Grauwert des Pixels(k,l) der Summer aller Pixelgrauwerte der Fläche eines Rechtecks, welches aufgespannt wird durch: Pixel(0,0) und Pixel(k,l). Siehe dazu: Figure 11. Das kumulierte Bild kann so berechnet werden:

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 einer verarbeiteten Bildgröße ist das konkret: $256 \cdot 256 \cdot 256 = 2^8 \cdot 2^8 \cdot 2^8 = 2^{24}$. Bei einer mittleren Bildgröße sind also 32 Bit ausreichend. Wenn die Inputbildgröße auf 512 erhöht wird sind 26 Bit notwendig. Bei einer Genauigkeit von Single-Float auf einer x86-Plattform mit 32 bit ist float single also ausreichend. Laut dem IEEE Standard 754-2008

```
- 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:
  v(k,l) = v(k,l-1) + v(k-1,l) - v(k-1,l-1) + u(k,l)
```

Listing 6: Pseudocode kumulierter Boxfilter

für Gleitkommazahlen [1] entspricht das einem Zahlenbereich von$1 \cdot 10^{-45}$ bis $3.403 \cdot 10^{38}$. Mit unserem GCC reicht damit die Deklaration unseres Datenarray mit
$float * meinKumuliertesBild$;

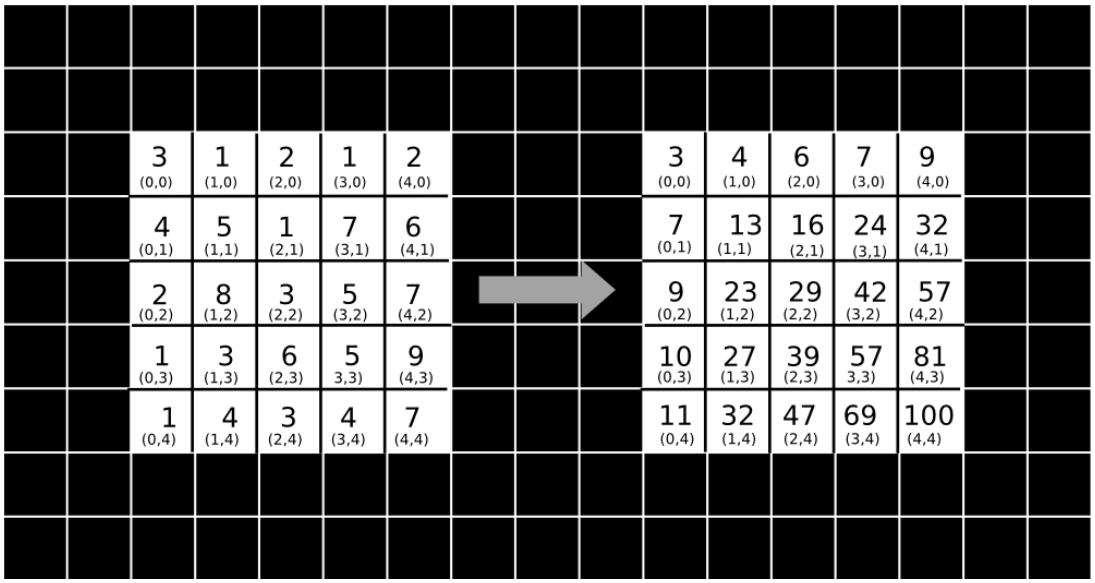

Abbildung 11: kumulierte Grauwerte

3.3.3 Lensblur, generischer Boxfilter

Die Abbildung 12 illustriert die Methode, die in Listing 7 kurz beschrieben ist. Zu dem Algorithmus führen zwei Ansätze, die zum Teil bereits vorgestellt wurden. Der nun vorgestellte Faltungsalgorithmus wird aus drei einzelnen Teile 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.

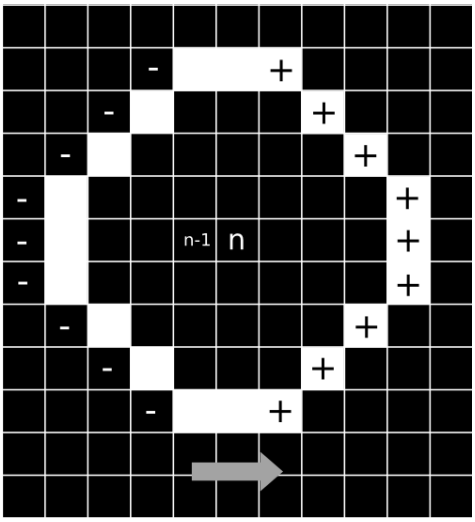

Abbildung 12: schneller Lensblur, Kern: Bresenham r=4px

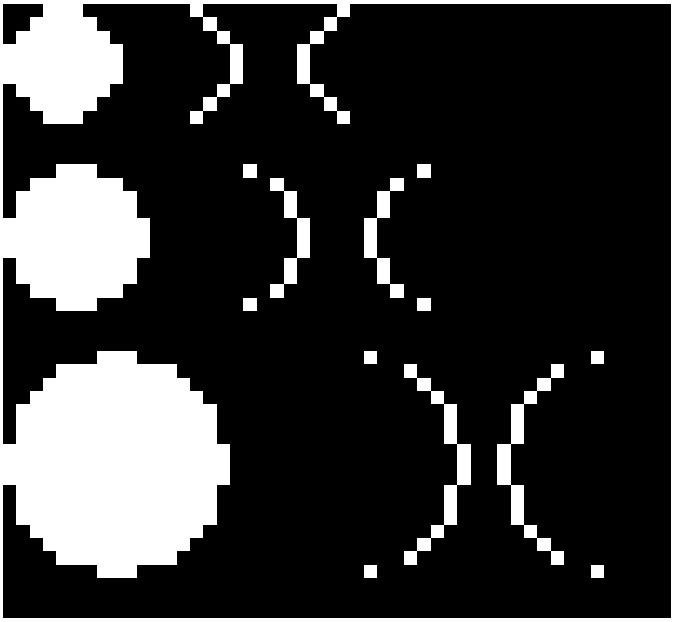

Abbildung 13: jeweils 3er Gruppen für den Lensblur Filter; mit Imagej erzeugt, Bresenham mit 9,11 und 17 Pixel Durchmesser, entspricht einem Radius von 4,5 und 8 Pixel

```
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 7: 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, sonder 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 schon mal etwa 20 Prozent ein (wohlgemerkt ohne Overhead).
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 12. 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):
Die Kreisfläche beträgt 57 Pixel (siehe Abb. 12). 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 nötig sind: $256^2 \cdot 17^2 = 18939904$
Das entspricht weniger als 6 Prozent.
Wir sollten also theoretisch 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 wir in Abb. 12 können wir mit dieser Methode einen Lensblur berechnen. Es ist aber gleichzeitig möglich, mit dem Algorithmus in Listing 7 weitere Kerne abzubilden. Einzige Vorraussetzung: Die Pixel müssen konstant und in jeder Zeile zusammenhängend sein. Die Abbildung 14 zeigt eine Auswahl an möglichen Kernen.

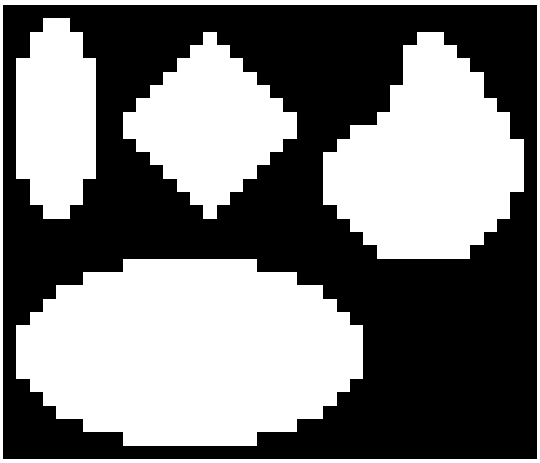

Abbildung 14: verschiedene Kerne für den generischen Boxfilter

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 abgebilded (siehe Listing. XX). 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. XX.
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 [11]. 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 8: 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 9: verwendete x86 build-in SSE Erweiterungen

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

Da die Optimierung jedoch auf der parallelen Multiplikation von Eingangbild und Kern in 4er Gruppen basiert, müssen diese Bilder jeweils 4-aligned sein. 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.

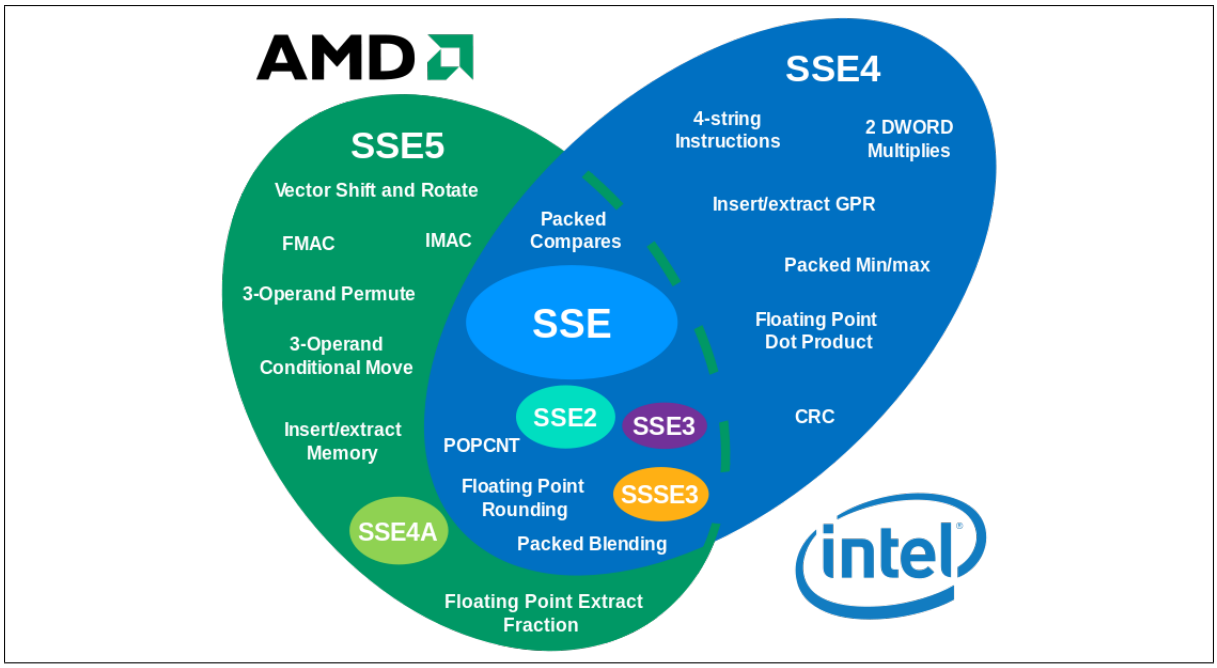

Abbildung 15: Architektur der SSE Befehlssätze von AMD und Intel X86 CPUs [15]

3.4 Analyse des Konvergenzverhaltens

Als erster Schritt soll eine Maßzahl gefunden werden, mit der sich die relevanten Pixel intentifizieren lassen. Relevant heißt in diesem Zusammenhang, dass nur Pixel berechnet werden sollen, die sich auch ändern. Es wurden zwei Maßzahlen berechnet. Zunächst wird die Varianz erhoben und danach betrachten wir die Grauwertänderung von einer Iteration zur nächsten.

Betrachtung der Varianz. Die Abbildung 28 zeigt die Varianz als Grauwertbild. Auf ein Eingangsbild mit einem Lensblur von $17px$ Durchmesser wurde der Robust Regularisierte Richardson-Lucy Algorithmus angewandt. Die dunkeln 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 Zwischenergebniss jeder Iteration berechnet werden:

$$\sum_{i=1}^{n}(x_i - \overline{x})^2 = \left(\sum_{i=1}^{n} x_i^2\right) - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2 \tag{10}$$

Die Abbildung 17 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 Stadardabweichung 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 18 sieht man, dass einer Iteration besonders starke Änderungen auftreten.

Abbildung 16: Varianz bei RRRL, 0.000001, 0.1, 1000 Iterationen, invertierte Darstellung

Wenige Pixel ändern sich innhalb einer Iteration um bis zu 20 Grauwerte. Weniger stark sind die Änderungen nach der zehnten Iteration. Nach 50 Iterationen änder sich die Pixel schließlich um weniger als einem Grauwert pro Iteration.

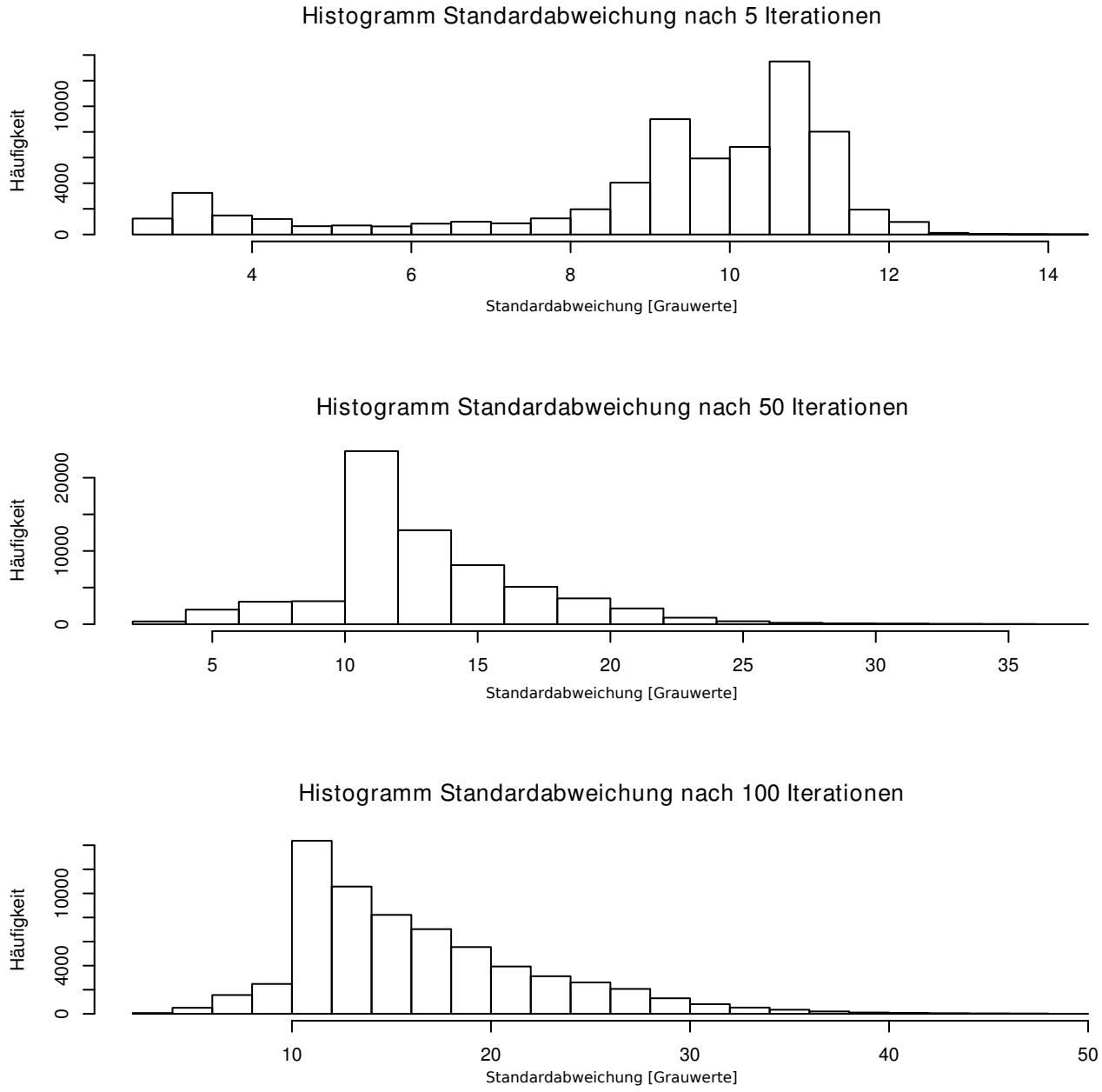

Abbildung 17: RL bei einem Lensblur von 17px, mit 5, 50 und 100 Iterationen. Standardabweichungen der geänderten Grauwerte eines jeden Pixels

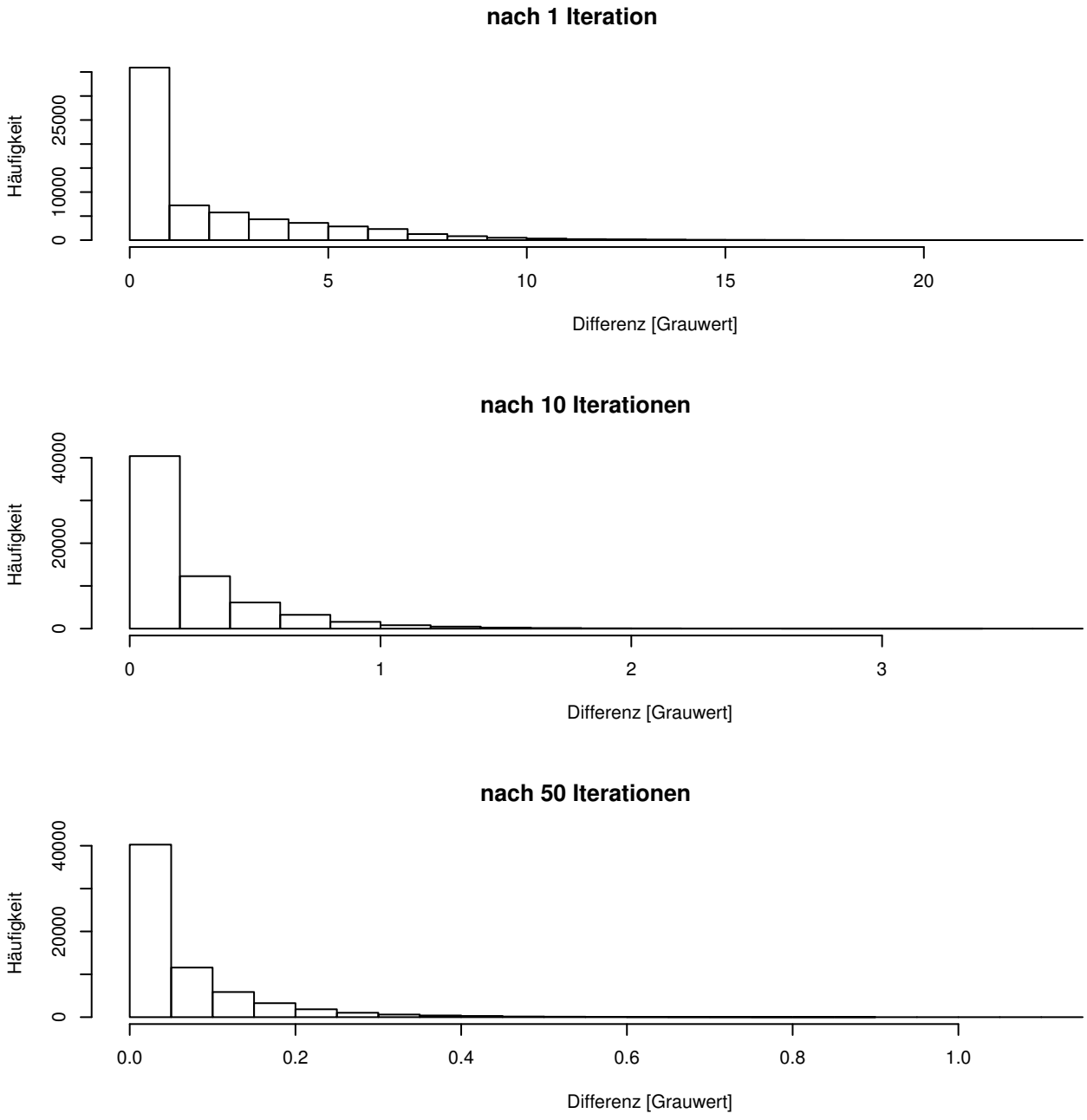

Abbildung 18: Grauwertänderung von einer Iteration zur nächsten;

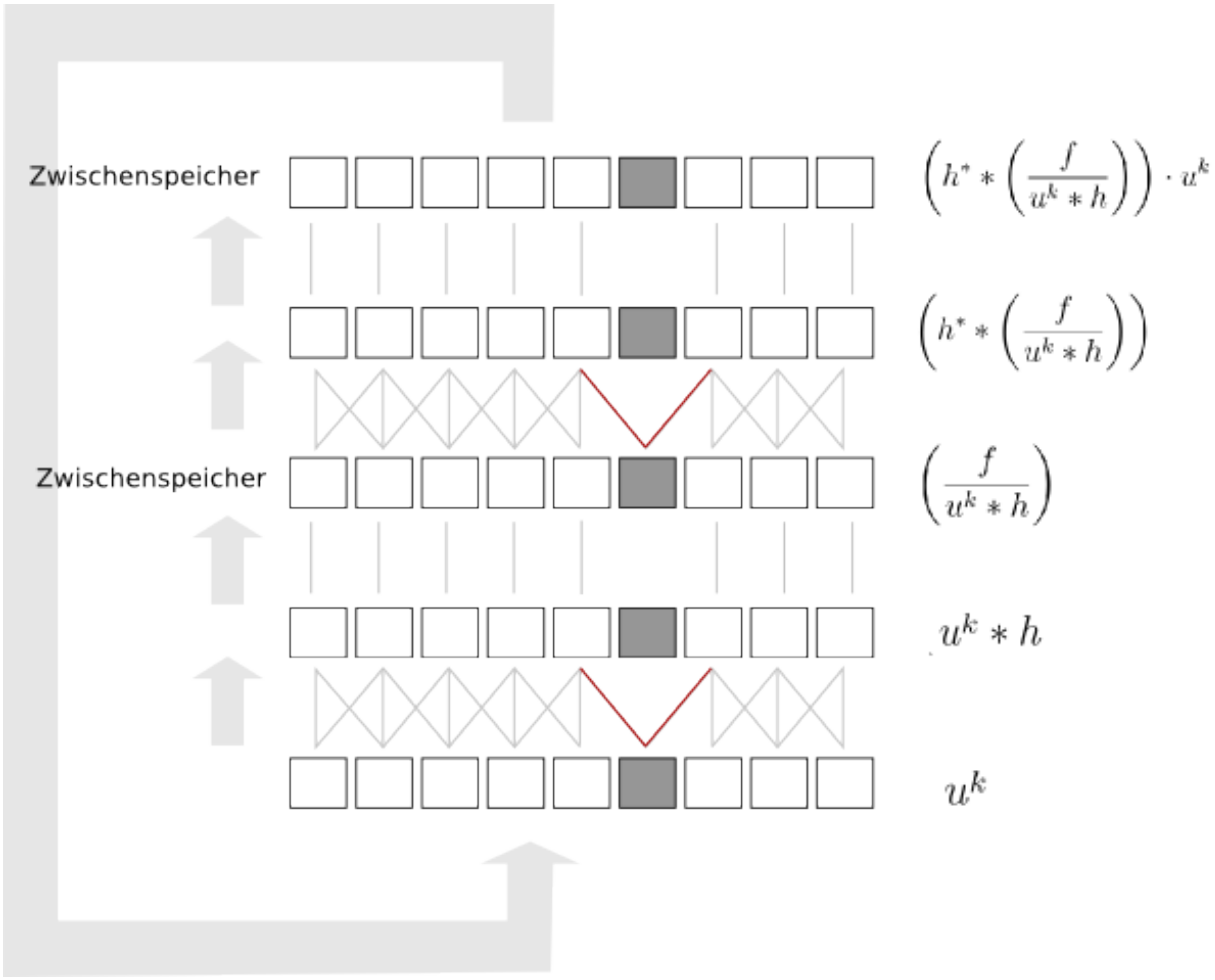

Abbildung 19: Ablaufdiagramm der optimierten Version

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 zu ermöglichen, muss das qualitative Ergebnis übereinstimmen. Die verglichenen Faltungsroutinen führen also zum selben qualitativen Ergebnis. Dennoch wurden verschiedene Randbedingungen implementiert. Diese qualitativen Unterschiede werden hier kurz betrachtet.

Ortsfaltung und Faltung im Frequenzbereich. Wie im Kapitel 1.4 beschrieben, werden die Ränder hier anders behandelt. Die Abb. 20 zeigt die Grauwertdifferenzen zwischen dem gefalteteten Bild im Orts- und Frequenzbereich.

Boxfilter, 1D, 2D separiert.Um den Boxfilter qualtitativ zu evaluieren, bietet sich ein Vergleich mit der Faltung im Ortsbereich an. Das Ergebnis des Grauwertvergleichs ist in Abb. 21 zu sehen. Hier treten keine Unterschiede auf.

Zeilenweiser Boxfilter. Der zeilenweise berechnete Boxfilter wurde ebenso mit der Faltung im Ortsbereich verglichen. Die Implementation zeigt keine qualitativen Unterschiede. Siehe dazu Abb. 22.

Listenfilter. Der konstante Listenfilter und der nicht konstante Listenfilter wurden auf qualitative Unterschiede mit dem Ortsfilter verglichen. Es konnten dabei keine Unterschiede gemessen werden. Die Abb. 23 zeigt, dass die Grauwerte aller Pixel übereinstimmen.

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 Ergebnisbild in Abb. 24 zeigt, 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 1.4 „Umbiegen“ nicht implementiert werden.

todo: ändern!!! Es werden die Pixel über den Rändern mit 0 angenommen (Dirichlet Rand). Daras entstehen schwarze Ränder. Bei mehrfacher Faltung wandert der schwarze Rand weiter zur Bildmitte, sodass das Bild nach einigen Iterationsschritten mit RL oder RRRL vollkommen schwarz wird. Die Implementation des kumulierten Boxfilter mit Dirichlet Rand zur iterativen Dekonvolution ist deshalb nicht möglich und wird bei der Performancemessung ausgenommen!

Ortsfaltung mit SIMD. Es konnte noch eine parallesisierte Version des Filters im

Ortsbereich implementiert werden. Durch die 4fache Parallelisierung konnte die Randbedingung nicht exakte nachgebildet werden. Daraus resultiert eine leiche Differenz an den Rändern. Die Abb. 26 zeigt die Messung der Grauwertdifferenzen. Außerdem konnte mit nicht alignierten Kernen zwar eine erfolgreiche Dekonvolution durchgeführt werden. Jedoch ergaben sich leichte Pegelunterschiede bei nicht alignierten Kernen.

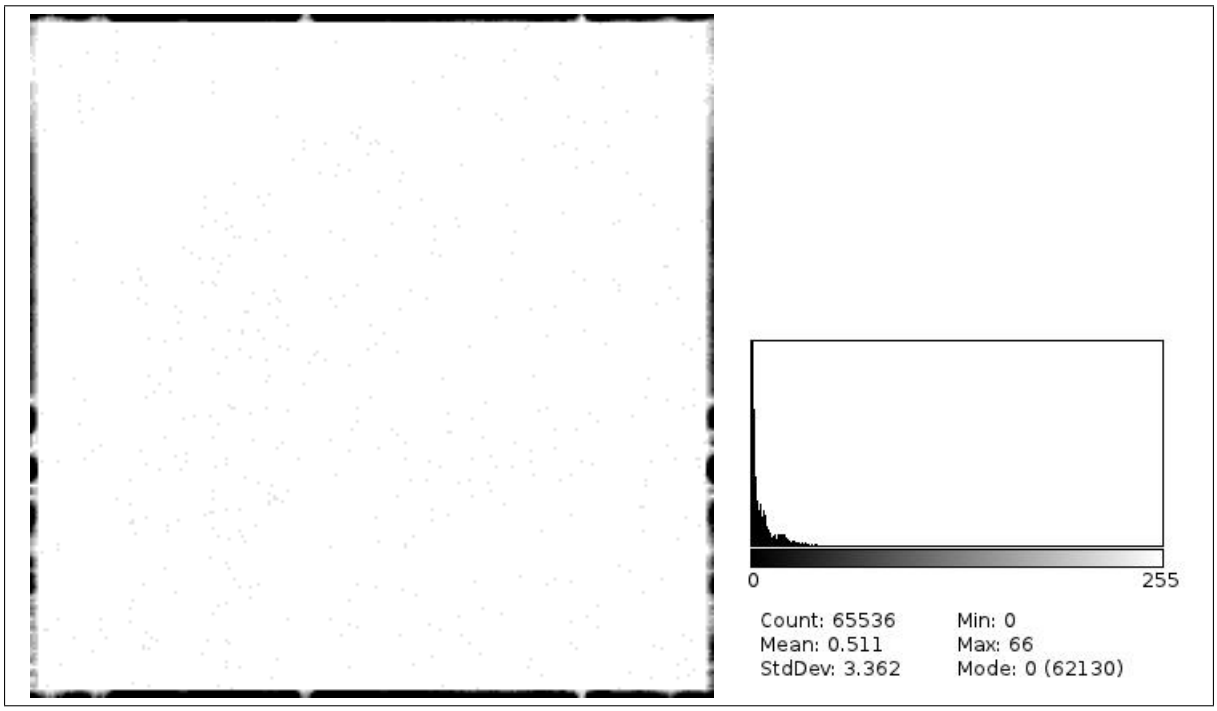

Abbildung 20: Grauwertdifferenz: kaum sichtbare Randunterschiede der Faltung im Orts- und Frequenzbereich, sowie Rundungsdifferenzen in der Bildmitte; Darstellung 30fach verstärkt und invertiert, rechts: das Histogramm

Lokalisierung. Ein weiterer qualitativer Vergleich ist durch Prüfung der Lokalisierungsgenauigkeit möglich. In Abb. 27 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.29 zeigt die gemessenen Berechnungszeiten. Es ist in eine obere und in eine untere Hälfte geteilt: Für die Routinen in der unteren Hälfte gelten besondere Vorraussetzungen. Folgende Parameter wurden gewählt:

- convolve: Ortsfaltung. Kern $17 \cdot 17$px

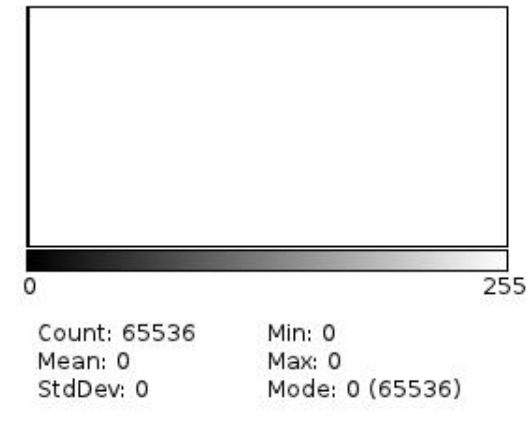

Abbildung 21: Grauwertdifferenz: das Histogramm zeigt die Unterschiede zwischen Ortsfaltung und separierten Boxfilter, inkl. Box 1D. keine Unterschiede

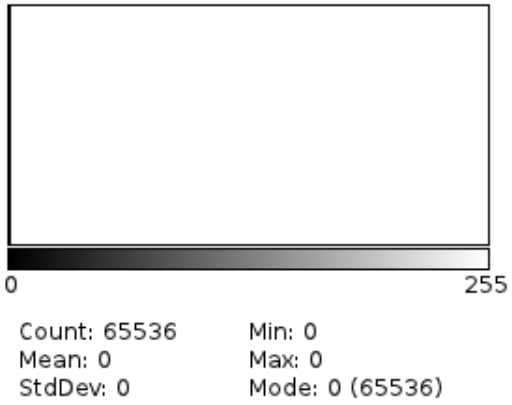

Abbildung 22: Grauwertdifferenz: das Histogramm zeigt die Unterschiede zwischen zeilenweisen Boxfilters und der Ortsfaltung. keine Differenz

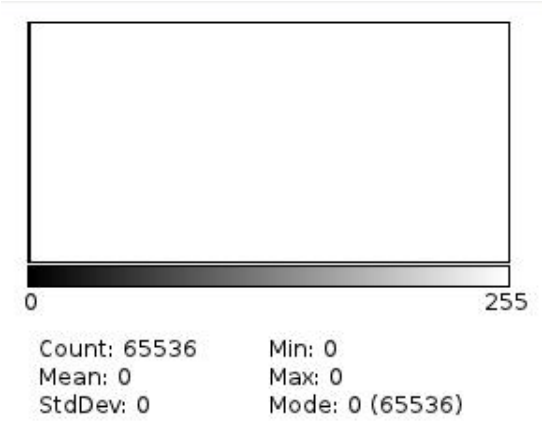

Abbildung 23: Grauwertdifferenz zwischen Listenfilter, bzw. konstanten Listenfilter und der Ortsfaltung. keine Unterschiede

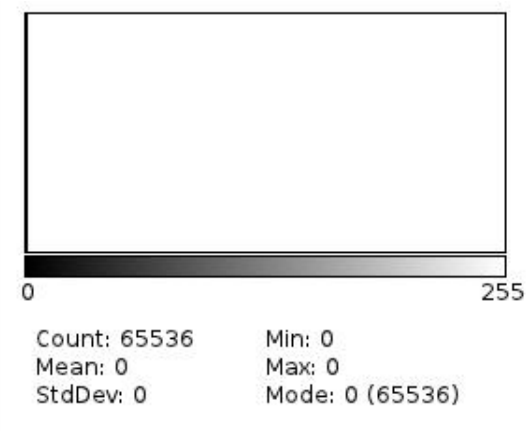

Abbildung 24: Grauwertdifferenz zwischen generischen Boxfilter und der Ortsfaltung. Dazu wurden drei Teilkerne für den generischen Boxfilter erstellt.

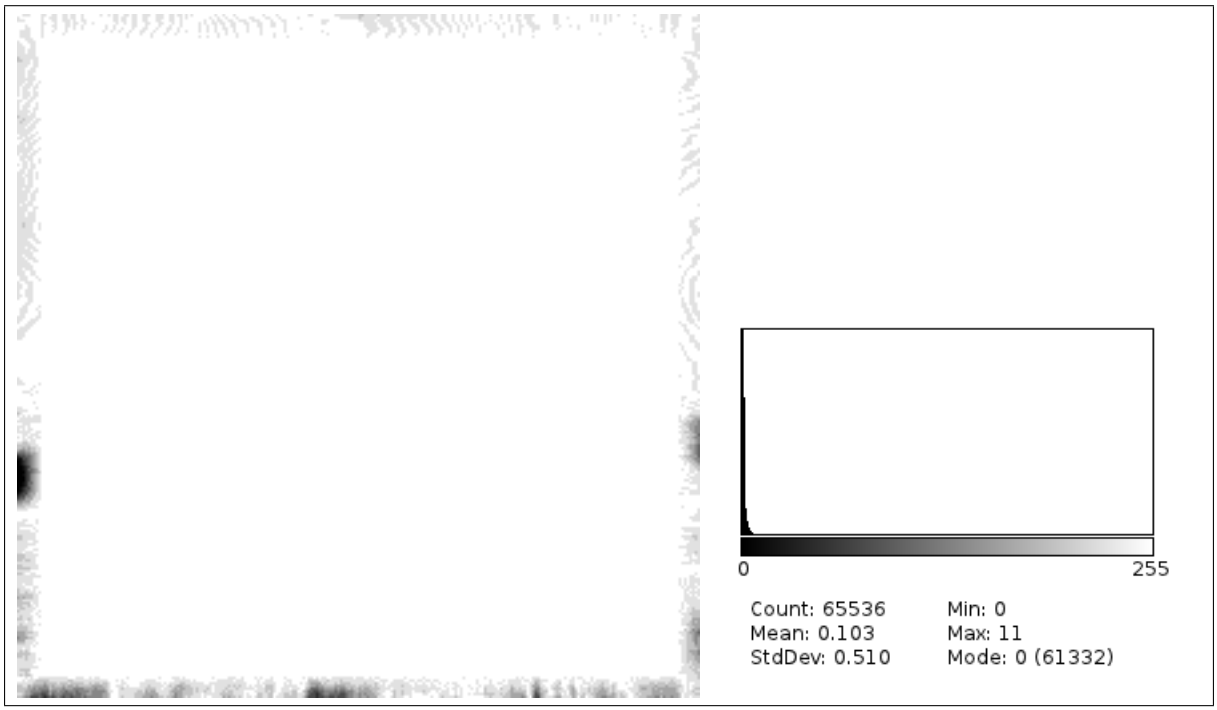

Abbildung 25: Grauwertdifferenz zwischen den kumulierten Boxfilter und der Faltung im Ortsbereich; Die Kerngröße nimmt zum Rand hin ab, so dass dort eine Differenz entsteht; die Darstellung wurde invertiert und 30fach verstärkt

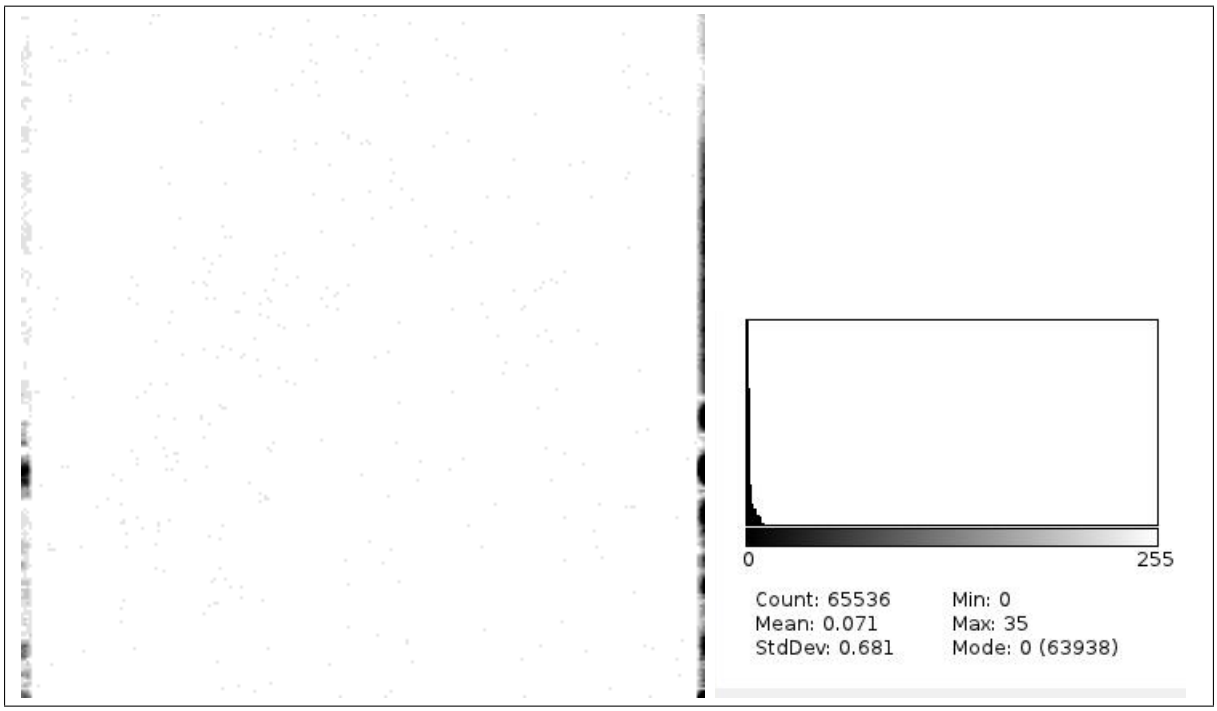

Abbildung 26: Grauwertdifferenz des Filters mit SIMD und ohne. 30fach verstärkt. Die Randbedingung weicht leicht ab und Rundungsfehler in der Mitte sind zu erkennen.

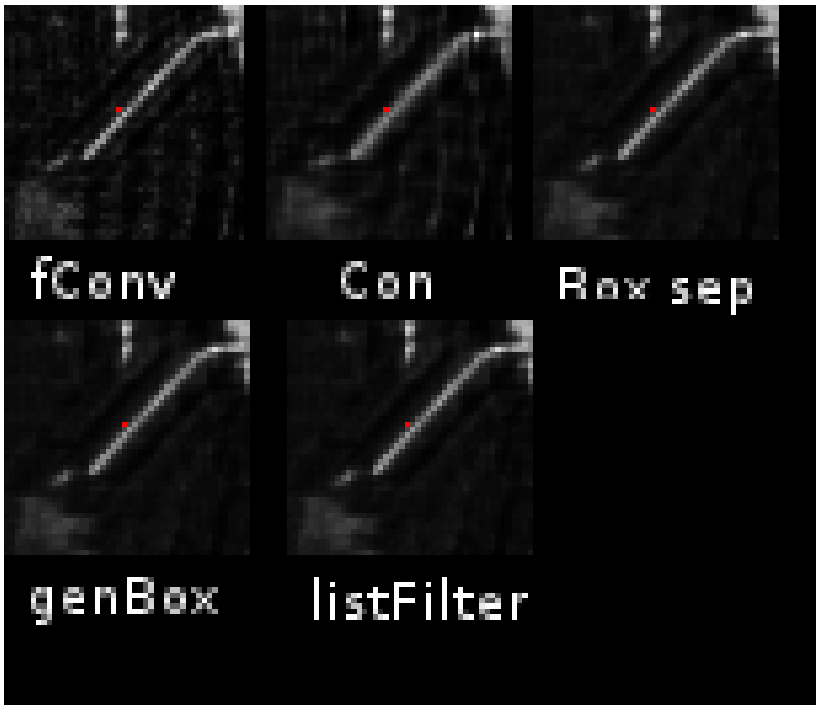

Abbildung 27: Lokalisierungsgenauigkeit. Der Punkt (103,103) wurde jeweils rot markiert. untersuchte Faltungen: F. im Frequenzbereich, Ortsbereich, separierter Boxfilter, generischer Boxfilter und der Listenfilter.

- fconvolve: Faltung im Fourierbereich. Kern $17 \cdot 17$px
- generischer Boxfilter: Lensblur von 17px Durchmesser. 213 Pixel besetzt.
- box2D nicht separiert: Kern $17 \cdot 17$px
- box1D: Motinblur in horizontaler Richtung mit 17px Unschärfe
- listFilter: Kreisrunder Kern mit 213 Elemente Listengröße. lensblur.
- convolve simd: Kern mit $20 \cdot 20$ Pixel.
- box2D kumuliert: Mit einer Kergröße von $17 \cdot 17$px. Zur Dekonvolution jedoch nicht geeignet.

Die Routinen „listFilter“ sowie „convolve simd“ sind stark von der 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 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.

Abbildung 28: Varianz bei RRRL, 0.000001, 0.1, 1000 Iterationen

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 dünn besetzt ist, dh. wenn viele Pixel gleich Null sind, soll der Listenfilter theoretisch

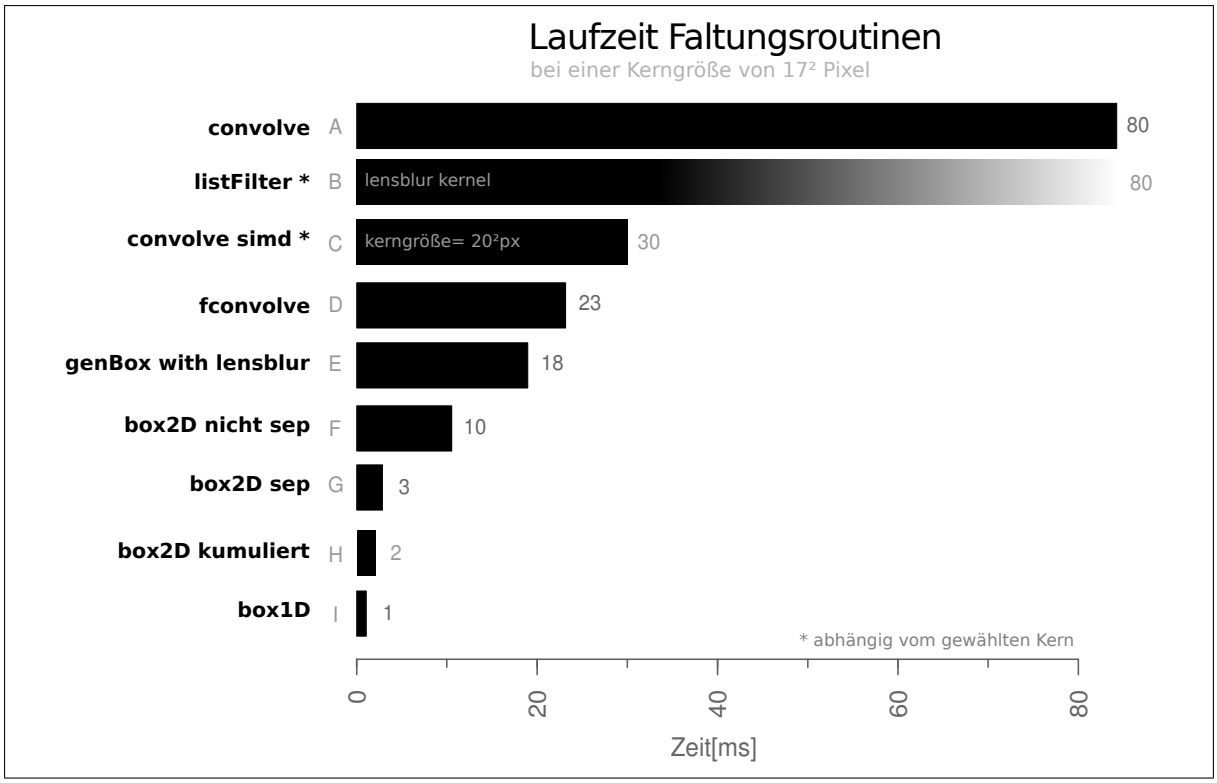

Abbildung 29: Zeitbedarf der Faltungsroutinen

schneller sein. Naheliegend scheint ein Vergleich Ortsfaltung zu Listenfilter, wobei beim Ortsfilter die Kerngröße und beim Listenfilter die Listengröße variiert. Das Diagramm von Abb. 30 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 Elemente. Die Messung zeigt: Die Ortfaltung brauch 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.

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

Laufzeit SIMD Ortsfaltung. Nachfolgend wird die hardwarespezifische Implementation der Ortfaltung näher betrachtet werden. Durch die Verwendung von SSE können vier float-Werte in einem Schritt multipliziert und addiert (SSE2) werden. Die Abb. 31 zeigt den Vergleich zwischen nicht-optimierter (rot) und optimierter Version (grün). Die Kerngröße steigt linear an: $1 \cdot n$. In der Anwendung wird ein Kern von 200 mal 1 Pixel kaum vorkommen, der Vergleich ermöglicht trotzdem einen sehr guten Vergleich. Dabei zeigt sich das lineare Ansteigen mit der Pixelgröße.

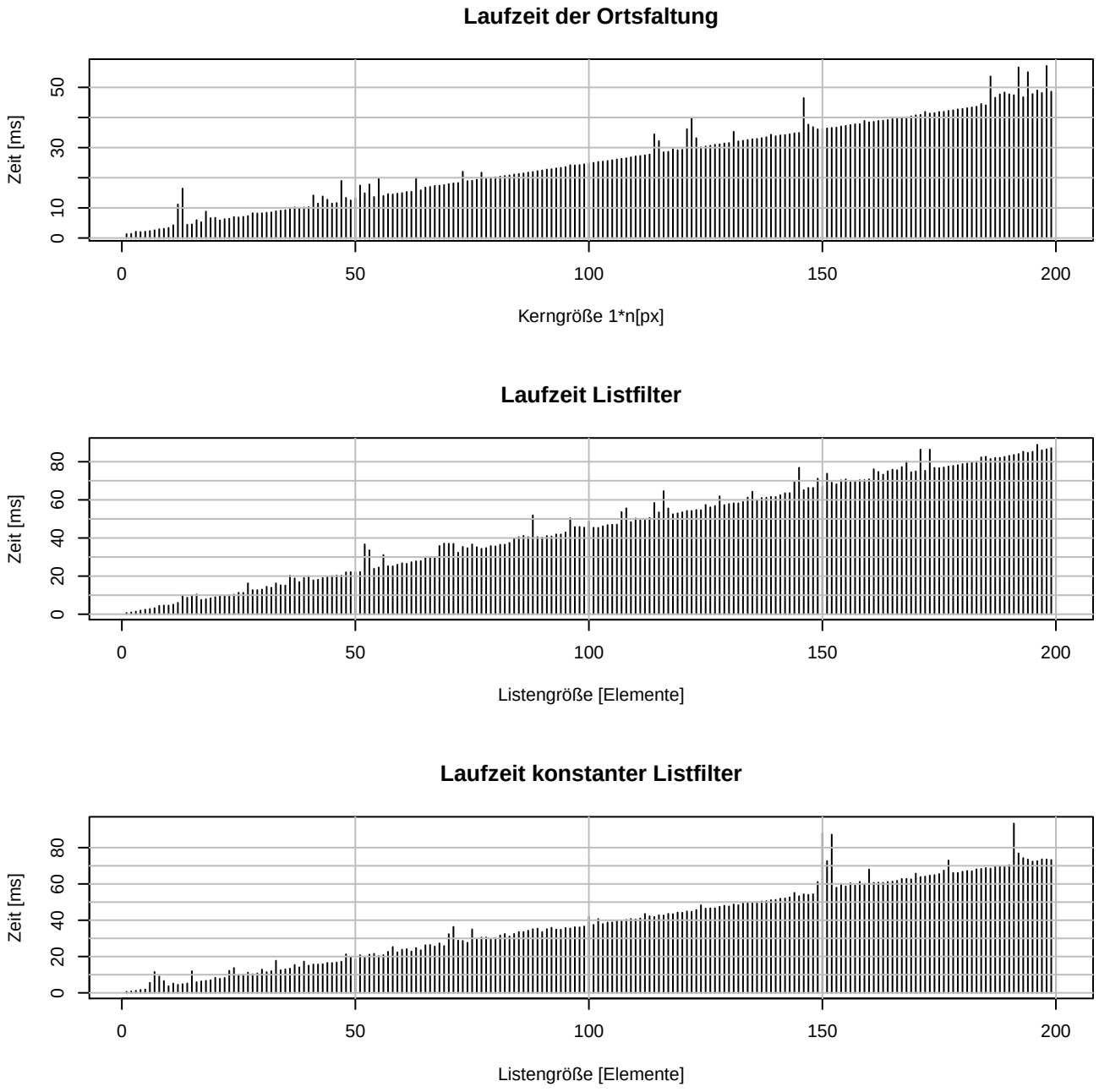

Abbildung 30: Zeitbedarf der Faltungsroutinen Listfilter und Ortsfaltung

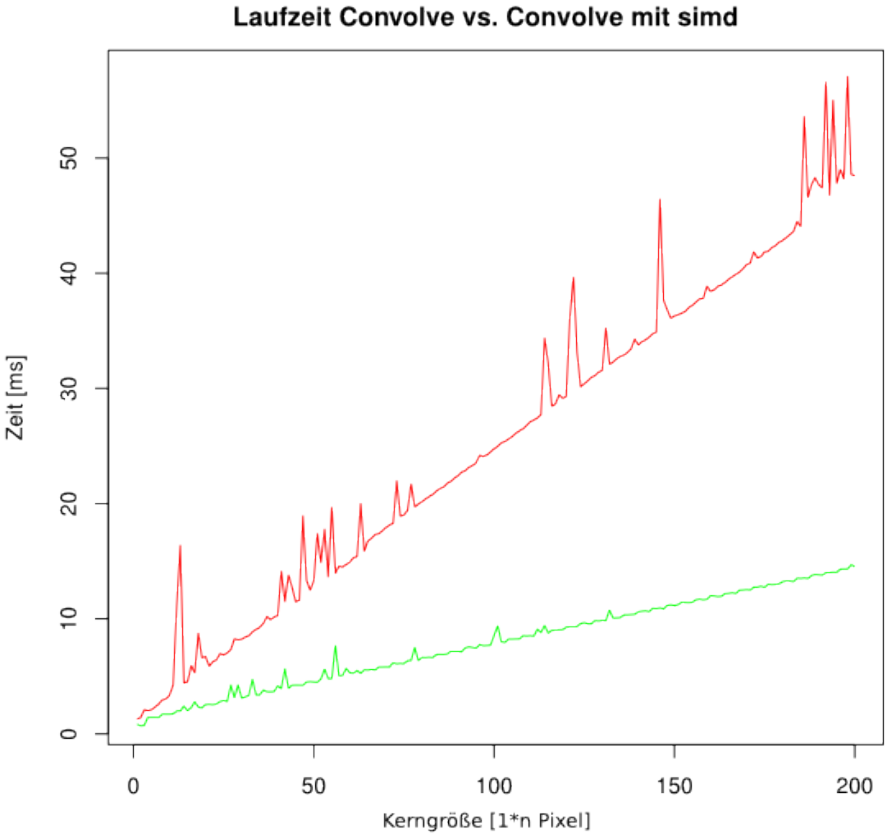

Abbildung 31: Zeitbedarf der gewöhnlichen Ortsfaltung und der SIMD-optimierten. Kerngröße steigt in der Form $1 \cdot n$ linear an

Faltungsroutinen mit Richarson 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 Abb. 33 zeigt die Ergebnisse nach Kerngrößen geordnet. Die Abb. 34 zeigt die selben 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. $d = 17, 11, 9$
- fconvolve: Faltung im Fourierbereich
- genBox: Faltung mit generischen Boxfilter. lensblur, dh. kreisrunder Kern. $d^2/4 \cdot \pi$ Elemente
- box2D nicht separiert: Boxkern mit jeweiliger Kerngröße.

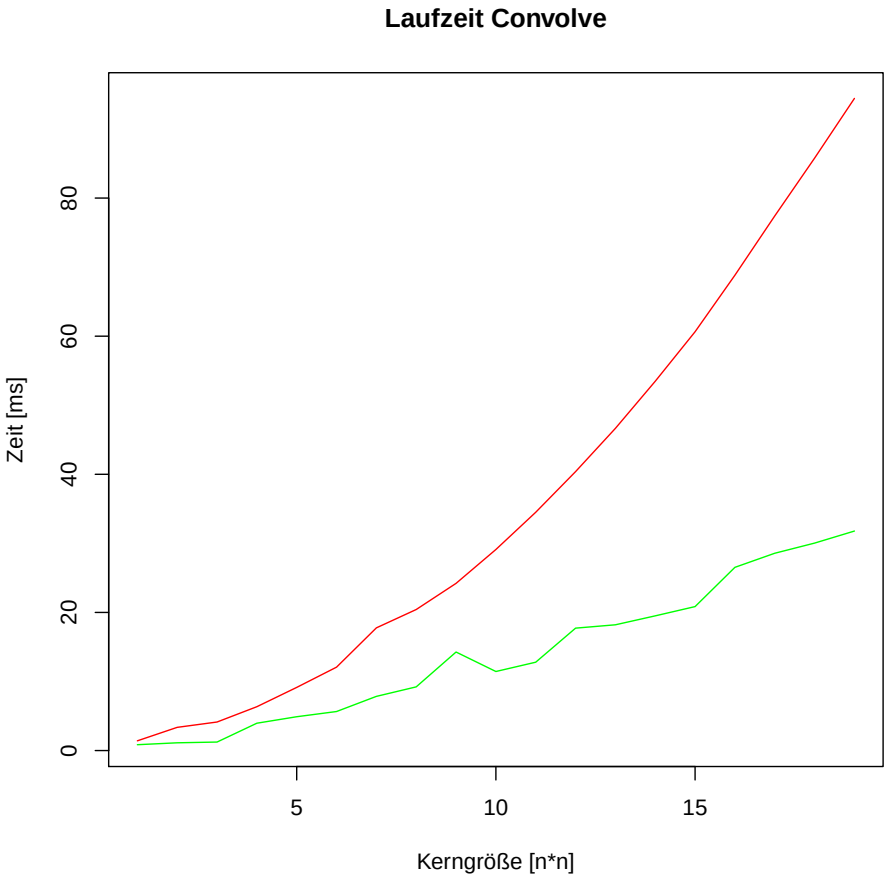

Abbildung 32: Zeitbedarf der gewöhnlichen Ortsfaltung und der SIMD-optimierten. Kerngröße steigt in der Form $n \cdot n$ quadratisch an

- box2D separiert: Boxkern mit jeweiliger Kerngröße.
- box1D: Boxkern mit $1 \cdot n$ Pixel. $d = 17, 11, 9$

Faltungsroutinen mit RRRL.

4.2 Messergebnisse der Konvergenzanalyse

Messungen bei 100 Iterationen und verschiedener Optimierungsparameter. 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 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 wurder gewählt: $\alpha = 0.000001, \epsilon = 0.1$.

Rechnenaufwand 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! Auffälig 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. 38 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 Pixel 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.

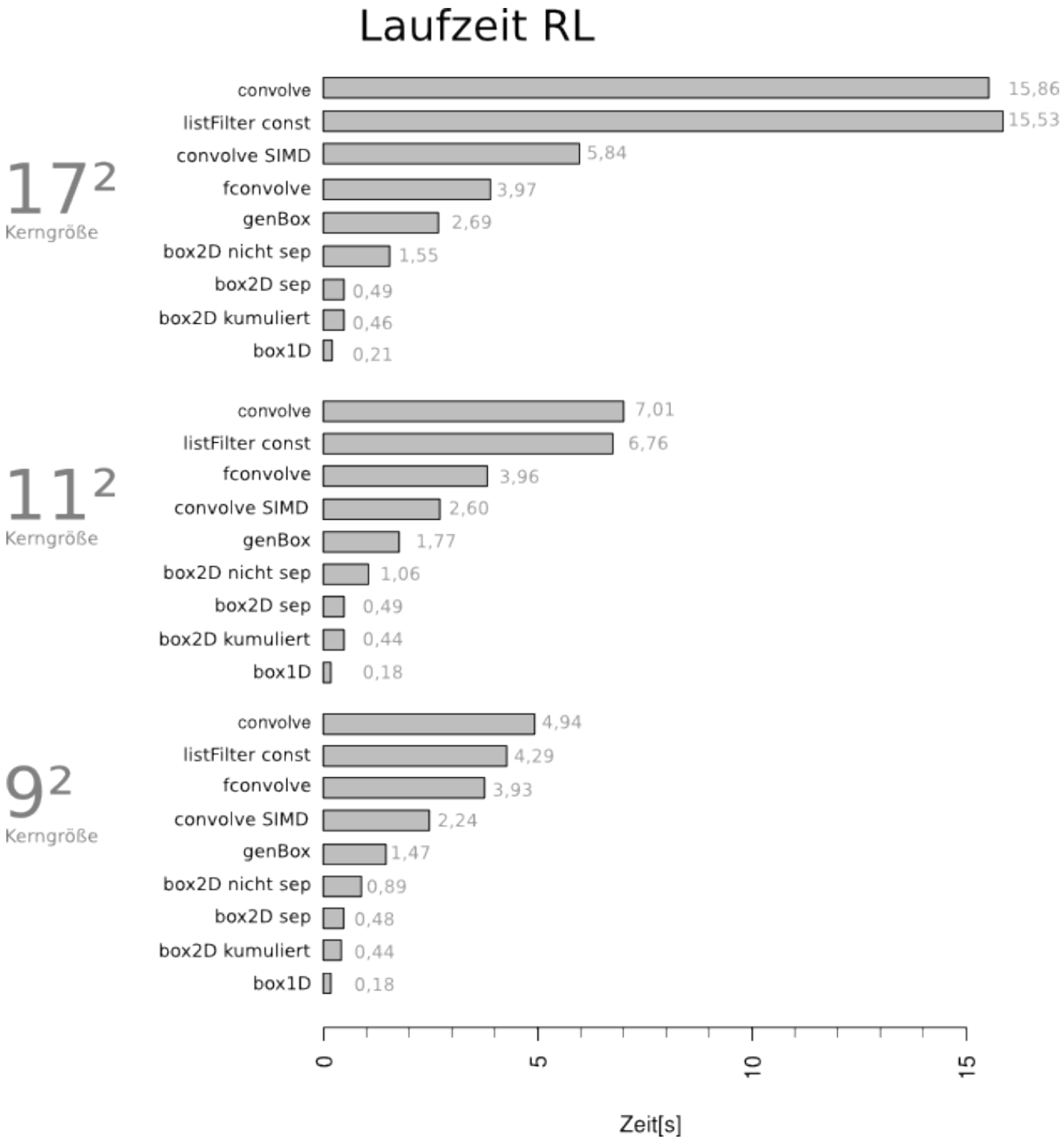

Abbildung 33: Zeitbedarf RL mit verschiedenen Faltungsroutinen, gruppert nach Kerngröße, 100 Iterationen

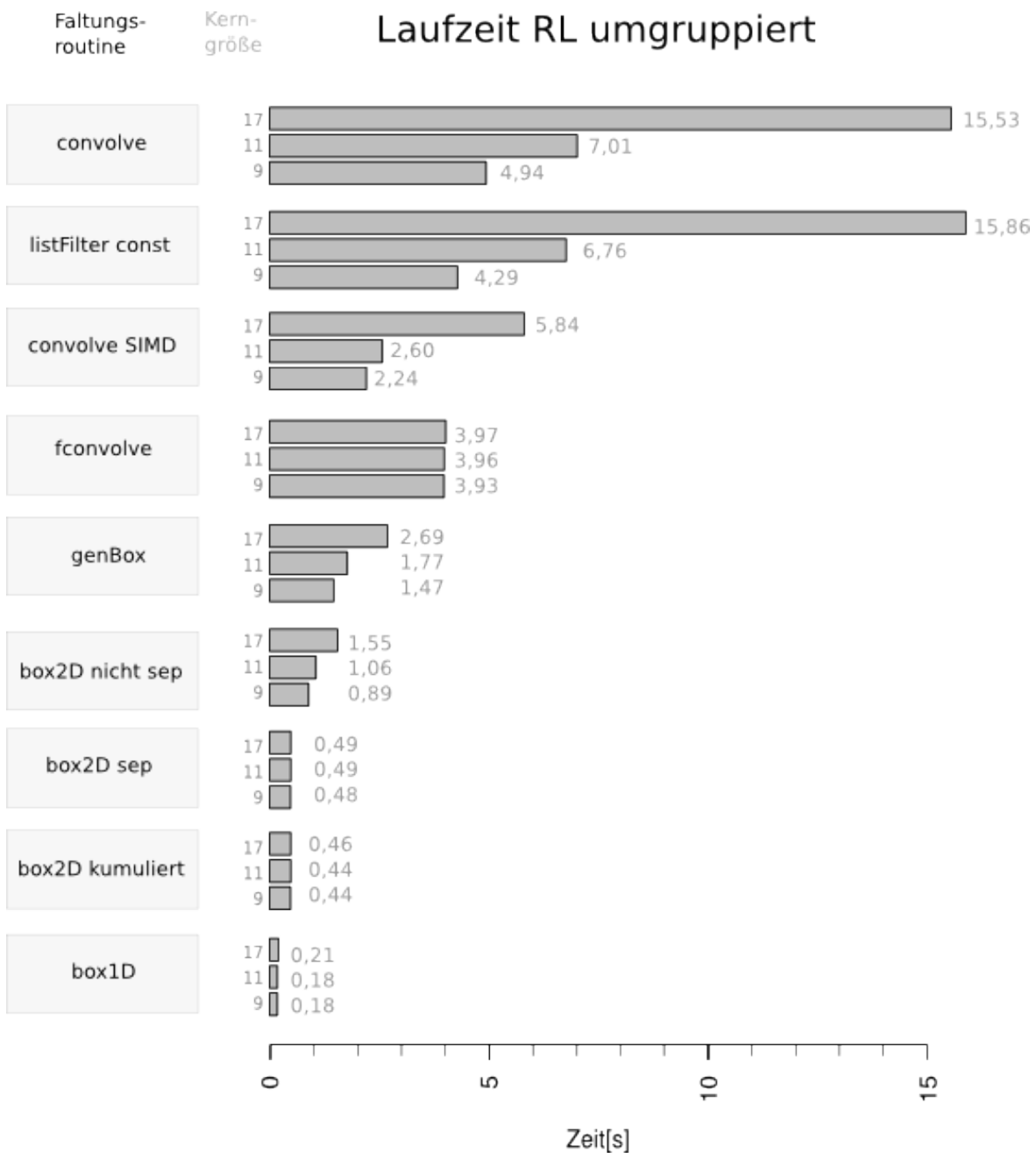

Abbildung 34: Zeitbedarf RL mit verschiedenen Faltungsroutinen, gruppert nach Faltungsroutinen, 100 Iterationen

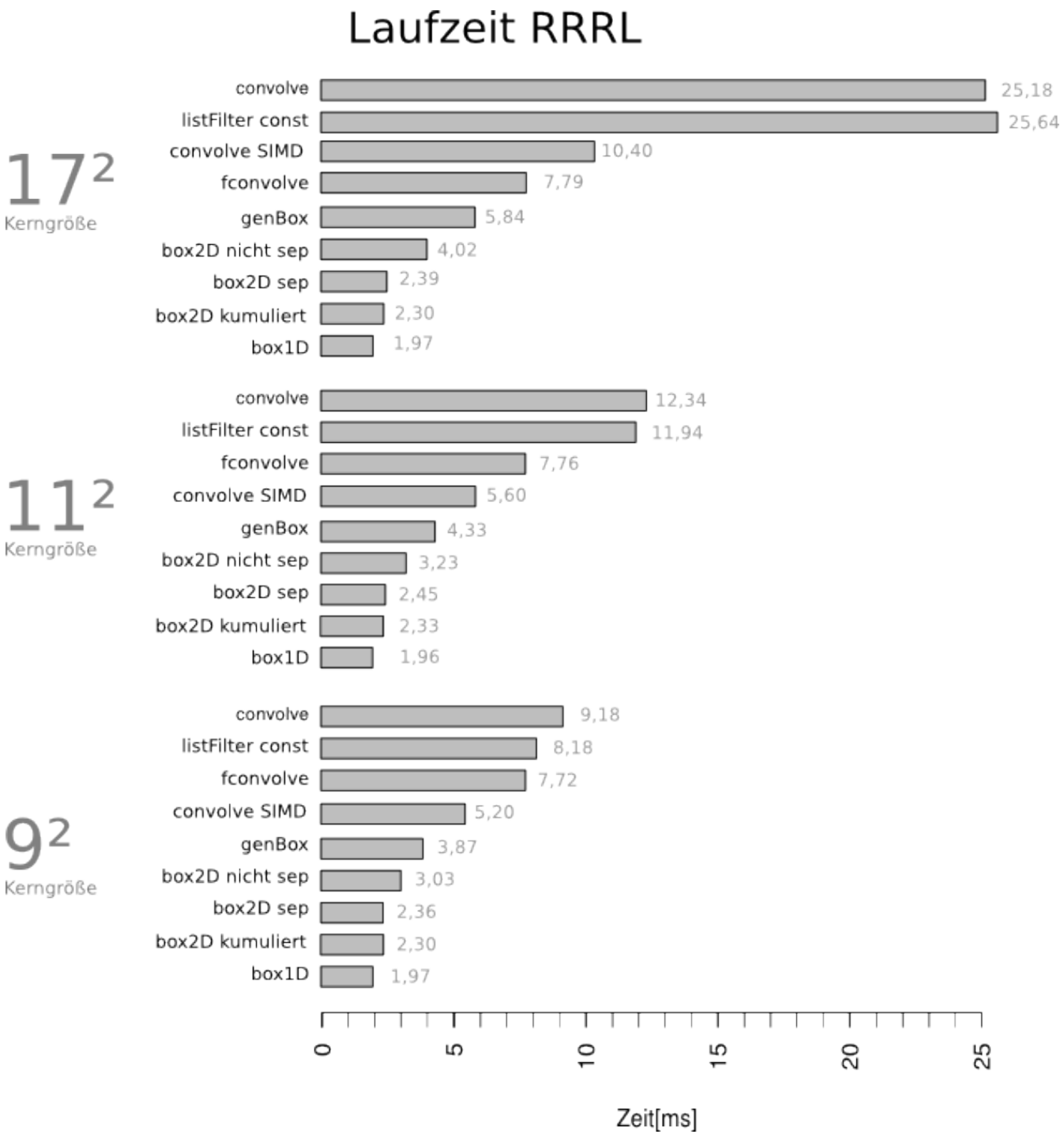

Abbildung 35: Zeitbedarf RRRL mit verschiedenen Faltungsroutinen, Parameter: 100 Iterationen, Alpha = 0.000001 , Epsilon = 0.1

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

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.

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.

5 Diskussion

5.1 SIMD

Wenn wir das Diagramm von Abb. 31 betrachten, können wir die Laufzeit der SIMD-optimierten Ortsfaltung sehen. Die auftretenden Peaks zeigen das nicht-deterministische Verhalten des Betriebssystems: Ubuntu ist kein Echtzeitbetriebssystem 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 nicht 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, sonder realistischer Weise

Abbildung 36: RL ohne Optimierung, 500 Iterationen

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. 32. 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.

6 Zusammenfassung/Abstract

Abbildung 37: RL mit Optimierung, 500 Iterationen, $k = 0.01$

Abbildungsverzeichnis

1 Ausgangsbild 5
2 1D Boxfilter mit 17 Pixel Ausdehnung in horizontaler Richtung, rechts unten: der Faltungskern (vergrößert) 6
3 2D Boxfilter mit 17 Pixel Unschärfe, rechts daneben der Faltungskern, vergrößert 6
4 Unschärfe durch Defokussierung mit einem Radius von 8 Pixel synthetisch hergestellt, ortsinvariant; rechts: der zugehörige Kern, 5fach vergrößert . . 7
5 Kern und Unschärfe. Illustration aus Wikipedia Comons [2] 7
6 Unschärfe durch untypische PSF, rechts der passende Faltungskern (vergrößert), wenig Pixel besetzt 8
7 Illuistration der Faltung im Ortsbereich 10
8 Ausgabe von gprof zum RL mit Ortsfaltung 12
9 Illustration des 1D Boxfilters 18
10 Illustration des 2D Boxfilters 20
11 kumulierte Grauwerte 21
12 schneller Lensblur, Kern: Bresenham r=4px 22
13 jeweils 3er Gruppen für den Lensblur Filter; mit Imagej erzeugt, Bresenham mit 9,11 und 17 Pixel Durchmesser, entspricht einem Radius von 4,5 und 8 Pixel 22
14 verschiedene Kerne für den generischen Boxfilter 24

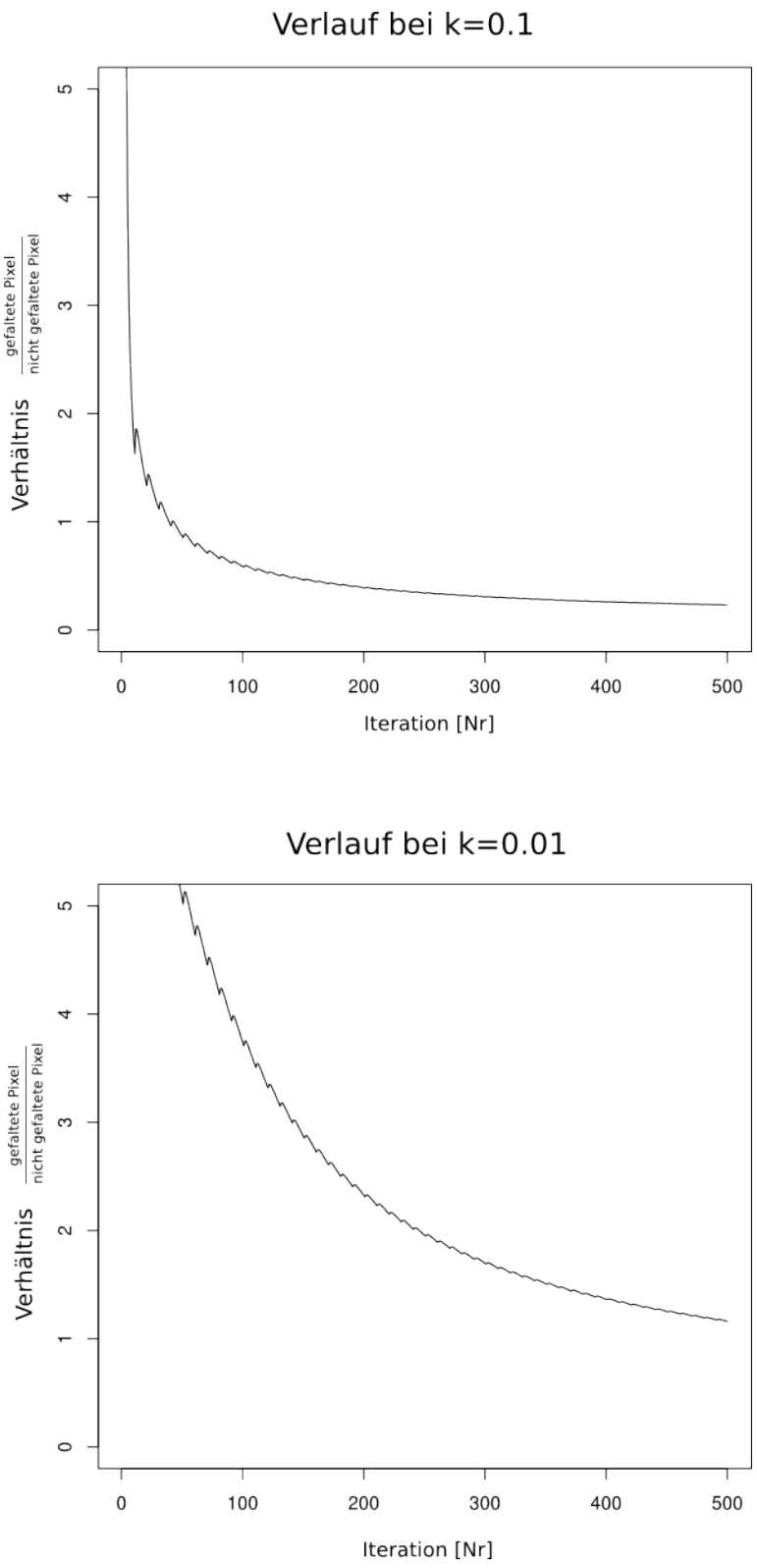

Abbildung 38: RL: Anteil gefalteter Pixel, 500 Iterationen, bei k=0.01 und k=0.1

Abbildung 39: RL mit Optimierung, 500 Iterationen, $k = 0.1$

15 Architektur der SSE Befehlssätze von AMD und Intel X86 CPUs [15] . . . 26
16 Varianz bei RRRL, 0.000001, 0.1, 1000 Iterationen, invertierte Darstellung 27
17 RL bei einem Lensblur von 17px, mit 5, 50 und 100 Iterationen. Standardabweichungen der geänderten Grauwerte eines jeden Pixels 28
18 Grauwertänderung von einer Iteration zur nächsten; 29
19 Ablaufdiagramm der optimierten Version 30
20 Grauwertdifferenz: kaum sichtbare Randunterschiede der Faltung im Orts- und Frequenzbereich, sowie Rundungsdifferenzen in der Bildmitte; Darstellung 30fach verstärkt und invertiert, rechts: das Histogramm 32
21 Grauwertdifferenz: das Histogramm zeigt die Unterschiede zwischen Ortsfaltung und separierten Boxfilter, inkl. Box 1D. keine Unterschiede 33
22 Grauwertdifferenz: das Histogramm zeigt die Unterschiede zwischen zeilenweisen Boxfilters und der Ortsfaltung. keine Differenz 33
23 Grauwertdifferenz zwischen Listenfilter, bzw. konstanten Listenfilter und der Ortsfaltung. keine Unterschiede 33
24 Grauwertdifferenz zwischen generischen Boxfilter und der Ortsfaltung. Dazu wurden drei Teilkerne für den generischen Boxfilter erstellt. 34
25 Grauwertdifferenz zwischen den kumulierten Boxfilter und der Faltung im Ortsbereich; Die Kerngröße nimmt zum Rand hin ab, so dass dort eine Differenz entsteht 34

26 Grauwertdifferenz des Filters mit SIMD und ohne. 30fach verstärkt. Die Randbedingung weicht leicht ab und Rundungsfehler in der Mitte sind zu erkennen. 35
27 Lokalisierungsgenauigkeit. Der Punkt (103,103) wurde jeweils rot markiert. untersuchte Faltungen: F. im Frequenzbereich, Ortsbereich, separierter Boxfilter, generischer Boxfilter und der Listenfilter. 35
28 Varianz bei RRRL, 0.000001, 0.1, 1000 Iterationen 36
29 Zeitbedarf der Faltungsroutinen 37
30 Zeitbedarf der Faltungsroutinen Listfilter und Ortsfaltung 38
31 Zeitbedarf der gewöhnlichen Ortsfaltung und der SIMD-optimierten. Kerngröße steigt in der Form $1 \cdot n$ linear an 39
32 Zeitbedarf der gewöhnlichen Ortsfaltung und der SIMD-optimierten. Kerngröße steigt in der Form $n \cdot n$ quadratisch an 40
33 Zeitbedarf RL mit verschiedenen Faltungsroutinen, gruppert nach Kerngröße, 100 Iterationen 42
34 Zeitbedarf RL mit verschiedenen Faltungsroutinen, gruppert nach Faltungsroutinen, 100 Iterationen 43
35 Zeitbedarf RRRL mit verschiedenen Faltungsroutinen, Parameter: 100 Iterationen, Alpha = 0.000001 , Epsilon = 0.1 44
36 RL ohne Optimierung, 500 Iterationen 47
37 RL mit Optimierung, 500 Iterationen, $k = 0.01$ 48
38 RL: Anteil gefalteter Pixel, 500 Iterationen, bei k=0.01 und k=0.1 49
39 RL mit Optimierung, 500 Iterationen, $k = 0.1$ 50

Tabellenverzeichnis

1 CPU Eckdaten Intel P6100[5] 17
2 RL mit Optimierung: Die Laufzeiten für verschiedene k-Werte und Signal-Rausch-Verhältnisse, 100 Iterationen 45
3 RRRL mit Optimierung: Die Laufzeiten für verschiedene k-Werte und Signal-Rausch-Verhältnisse, 100 Iterationen, $\alpha = 0.000001, \epsilon = 0.1$ 46
4 RL bei 500 Iterationen. 46

Listings

1 Kompileraufruf 14
2 PGM Dateien einlesen und schreiben 15
3 Speicherorganisation und Pixelzugriff 16
4 Codeauschnitt 1D Box 19
5 Pseudocode nicht separierter Boxfilter Boxfilter 20
6 Pseudocode kumulierter Boxfilter 21
7 Pseudocode generischer Boxfilter 22
8 Datenformat des Listfilter 25
9 verwendete x86 build-in SSE Erweiterungen 25

Literatur

[1] IEEE Standards Association. Ieee 754-2008: Standard for floating-point arithmetic, 2008.

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

[3] 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.

[4] Agner Fog. Optimizing software in c++ - an optimization guide for windows, linux and mac platforms. `http://agner.org/`, 2011. [Online; accessed 1-Sep-2012].

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

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

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

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

[9] Graham Kessler McKusick. gprof: a call graph execution proler. *SIGPLAN, Symposium on Compiler Construction*, 1982.

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

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

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

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

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

[15] user:Shadim wikipedia commons. Sse architecture relations. `http://upload.wikimedia.org/wikipedia/de/a/aa/Sse-architecture-relations.svg`, 2012. [Online; accessed 2-Oct-2012].

[16] Y.L. You, W. Xu, A. Tannenbaum, and M. Kaveh. Behavioral analysis of anisotropic diffusion in image processing. *Image Processing, IEEE Transactions on*, 5(11):1539–1553, 1996.

Abkürzungen) Einleitung und Stand der Forschung Zielsetzung Methoden Ergebnisse Diskussion Zusammenfassung (deutsch und englisch!) Literatur 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.