LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

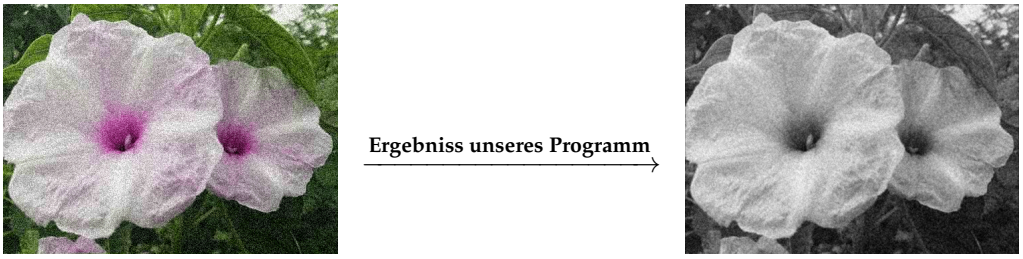
Gruppe 142 – Abgabe zu Aufgabe A220
Wintersemester 2023/24

Michael Ries Feres Ben Fraj Oussama Jeddou

1 Einleitung

Bei diesem Projekt handelt es sich mit dem Thema des Image Processing. Das Prinzip der Rauschunterdrückung in der Bildverarbeitung besteht darin, unerwünschtes Rauschen von einem Bild zu entfernen oder zu reduzieren, während gleichzeitig wichtige Merkmale erhalten bleiben. Bildrauschen kann durch verschiedene Faktoren wie Sensorfehler, Übertragungsfehler oder Umgebungsbedingungen verursacht werden. Das Ziel ist es, die visuelle Qualität und Interpretierbarkeit von Bildern zu verbessern, indem der Einfluss dieses Rauschens verringert wird.

Die folgende Arbeit beschäftigt sich genauer mit der Konvertierung eines farbigen Pixelbildes in Graustufen und die anschließende Entrauschung durch einen Laplace-Filter in Kombination mit einer Weichzeichnung. Die Zielsetzung in diesem Kontext ist ein Program welches 24bpp PPM P6 Farbbilder einlesen und dessen einzelne Pixel abspeichern kann um diese dann einer Methode Denoise zu geben welche die Umwandlung vornimmt. Für Denoise wurden zunächst die zugrundeliegenden Mathematischen Formeln für Graustufen umwandlung und Entrauschung analysiert. Auf dessen basis sind dann vier Algorithmen entwickelt worden, welche eine neue Menge von Pixel erzeugen mit denen sich das Ergebnisbild zusammensetzten lässt. Bei den verschiednene Implementationen wurde mit Veränderung der Berechnungsreihenfolge, Threads und SIMD versucht die Laufzeit des Algorithmuses zu verringern. Im folgenden werden die Konzepte und Ansätze der verschiedenen Implementationen anschaulich vorgestellt sowie die Korrektheit des Algorithmuses aufgezeigt. Zum Abschluss wird die Performanz der vier verschiedenen Implementationen gegenübergestellt und deutlich gemacht welche Laufzeitverbesserungen die verschiedenen Implementationen hervorbrachten.



(a) Original Bild (b) Bildergebnis

2 Lösungsansatz

2.1 Einführung

Die folgenden **4** Methoden umwadeln eines farbigen bildes in einem entrauschten Bild **Q'** indem man das originale und das weichgezeich-nete Bild basierend auf der Ausgabe des Laplace-Filters wie folgt kombiniert : $\mathbf{Q'_{(x,y)}} = \frac{|\mathbf{QL_{(x,y)}}|}{\mathbf{1020}} \times \mathbf{Q_{(x,y)}} + \left(\mathbf{1} - \frac{|\mathbf{QL_{(x,y)}}|}{\mathbf{1020}}\right) \times \mathbf{Qw_{(x,y)}}$

- **Q** : Graustufenkonvertierung: Umwandlung jedes Pixel in Graustufen durch das Berechnen eines gewichteten Durchschnitts D mittels der Koeffizienten a, b und c und die Farbkanäle Rot(R), Grün(G) und Blau(B) : $\mathbf{Q_{(x,y)}} = \mathbf{D} = \frac{\mathbf{a}\times\mathbf{R}+\mathbf{b}\times\mathbf{G}+\mathbf{c}\times\mathbf{B}}{\mathbf{a}+\mathbf{b}+\mathbf{c}}$
- **QL** : Laplace-Filter : $\mathbf{QL} = \mathbf{ML} * \mathbf{Q}$, ML ein gegeben Faltungsmatrix
- **Qw**: Weichzeichnung des Bildes : $\mathbf{QL} = \mathbf{Mw} * \mathbf{Q}$, Mw ein gegeben Faltungsmatrix

Mit der Faltungsoperator * als: $\begin{bmatrix} M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 \\ M_7 & M_8 & M_9 \end{bmatrix}$ * $\begin{bmatrix} D_1 & D_2 & D_3 \\ D_4 & D_5 & D_6 \\ D_7 & D_8 & D_9 \end{bmatrix}$ = $\sum_{\mathbf{i=1}}^{\mathbf{9}} \mathbf{M_i} \cdot \mathbf{D_i}$

In allen nachfolgenden Implementierungen haben wir die Parameter der Denoise-Methode unverändert gelassen. Wir haben lediglich diejenigen Parameter nicht allokiert, die in einigen Methoden nicht verwendet werden.

2.2 Denoise Naive

Das ist die erste Idee, die uns eingefallen ist. Sie besteht aus **3 Schritten**:

1. **Berechnung des Graustufenbildes Q**
   Für jedes Pixel berechnen wir **Q(x, y) = D** Die Ergebnisse werden in einer Matrix **Q (tmp1)** gespeichert. Das ist moeglich da die Ergebnisse immer zwischen 0 und 255 liegen und somit mit $\mathbf{uint_8}$ dargestellt werden können.
2. **Berechnung der Weichzeichnung des Pixels Qw**
   Wir iterieren über alle Pixels und führen für jedes Pixel die oben genannte Faltungsoperation durch, um den Wert **QL(x, y)** zu berechnen. Die Ergebnisse werden in einer Matrix **Qw (tmp2)** gespeichert. Hier ist die Speicherung mit $\mathbf{uint_8}$ auch moeglich.
3. **Berechnung des entrauschten Bildes Q'**
   In diesem Abschnitt müssen wir zunächst den Laplace-Filter **QL(x, y)** für jedes Pixel **(x, y)** berechnen. Dies geschieht mithilfe desselben Verfahrens wie bei **QW**, jedoch ohne Speicherung, da das Ergebnis nun nicht mehr im uint8 Format dargestellt werden kann. Stattdessen verwenden wir den berechneten Wert direkt in der Berechnung von **Q'(x, y)** für jedes Pixel **(x, y)**. Das Ergebnis wird im **result** gespeichert. Um eine bessere Genauigkeit zu erzielen, haben wir die Funktion abs in der Implementierung durch eine Fallunterscheidung ersetzt.

Algorithm 1 Naive Algorithm

for (x, y) **in** Pixels **do**
    $Tmp_1[x + y \times \text{width}] \leftarrow Q_{(x,y)}$
end for

for (x, y) **in** Pixels **do**
    $Qw_{(x,y)} \leftarrow 0$
    for $i, j \leftarrow 0$ **to** 2 **do**
        $Qw_{(x,y)} \mathrel{+}= Mw[i \times 3 + j] \times Tmp_1[x + i - 1 + (y + j - 1) \times \text{width}]$
    end for
    $Tmp_2[x + y \times \text{width}] \leftarrow Qw_{(x,y)}$
end for

for (x, y) **in** Pixels **do**
    for $i, j \leftarrow 0$ **to** 2 **do**
        $QL_{(x,y)} \mathrel{+}= ML[i \times 3 + j] \times Tmp_1[x + i - 1 + (y + j - 1) \times \text{width}]$
    end for
    $result[x + y \times \text{width}] \leftarrow Q'_{(x,y)}$
end for

2.3 Denoise Optimised

Am Anfang berechnen wir das Graustufenbild **Q** wie im naiven Ansatz.
Der Unterschied liegt nun darin: Kombination der Berechnung des Laplace-Filters mit der Weichzeichnung des Bildes und des entrauschten Bildes, indem wir jedes Element in **QL**, **Qw** und **Q'** in nur einer Schleife berechnen. Durch diese Vorgehensweise erreichen wir eine bedeutende Optimierung im Vergleich zum naiven Ansatz: Wir sparen ein Schleifendurchlauf und ein Zwischenspeicher **(tmp2)**. Dies ermöglicht uns auch eine bessere Genauigkeit, da **Qw** bleibt Double und nicht mehr zu $\mathbf{uint_8}$ gecastet wird.

Algorithm 2 Optimized Algorithm

for (x, y) **in** Pixels **do**
    $Tmp_1[x + y \times \text{width}] \leftarrow Q_{(x,y)}$
end for

for (x, y) **in** Pixels **do**
    for $i, j \leftarrow 0$ **to** 2 **do**
        $QL_{(x,y)} \mathrel{+}= ML[i \times 3 + j] \times Tmp_1[x + i - 1 + (y + j - 1) \times \text{width}]$
        $Qw_{(x,y)} \mathrel{+}= Mw[i \times 3 + j] \times Tmp_1[x + i - 1 + (y + j - 1) \times \text{width}]$
    end for
    result[x + y × width] ← Q'[pos]
end for

2.4 Denoise with SIMD

Ähnlich dem Optmized Ansatz aber mit SIMD.
Das Ziel dieses Programms ist die parallele Addition und Multiplikation sowohl der Faltungsmatrix **ML** und des Graustufenbildes **Q**, um den Laplace-Filter zu berechnen, als auch der Faltungsmatrix **Mw** und des Graustufenbildes **Q**, um die Weichzeichnung des Bildes zu berechnen.

Am Anfang berechnen wir das Graustufenbild wie im optimized Ansatz, aber die Speicherung in **tmp1** erfolgt anders. Tatsächlich füllen wir die erste Zeile, die erste Spalte sowie die letzte Zeile mit Nullen, und die verbleibenden Stellen mit dem berechneten Wert **D** jedes Pixels. Dadurch erhalten wir ein (2+height)x(1+width)+2 **tmp1** . Die Idee dahinter ist die Vermeidung von Segmentation Faults, wenn später parallel auf **tmp1** zugegriffen wird. Hier ist auch ein Cast von **tmp1** zu float* notwendig, um parallel auf 4 Elemente zugreifen zu können.

$$\begin{bmatrix} d_1 & d_2 & . & . \\ . & . & . & . \\ . & . & d_{n-1} & d_n \end{bmatrix} \xrightarrow{\textbf{Nullen hinzuefugen}} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & d_1 & d_2 & . & . \\ 0 & . & . & . & . \\ 0 & . & . & d_{n-1} & d_n \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & & & \end{bmatrix}$$

Im nächsten Schritt füllen wir **ML** und **Mw** nicht mit 9 Elementen, sondern mit 12, indem wir nach jedem 3 Element eine Null hinzufügen. Die Idee dahinter ist, dass wir später mit SIMD-Operationen auf 3 Elemente von **ML** (und auch **Mw**) parallel zugreifen und mit 3 Elementen von tmp1 multiplizieren möchten. Das ist schwieriger, wenn **ML** und **Mw** normal mit 9 Elementen gefüllt werden, da der Zugriff mit SIMD über **–m128** nur mit 4 Floats ($32Bit \times 4 = 128Bit$) erfolgt.

$$\begin{bmatrix} M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 \\ M_7 & M_8 & M_9 \end{bmatrix} \xrightarrow{\textbf{Nullen hinzuefugen}} \begin{bmatrix} M_1 & M_2 & M_3 & 0 \\ M_4 & M_5 & M_6 & 0 \\ M_7 & M_8 & M_9 & 0 \end{bmatrix}$$

Nun sind **Q**, **ML** und **Mw** bereit, und wir können damit mit SIMD arbeiten. Wir iterieren über alle Pixel, und für jedes Pixel **ML(x, y)** berechnen wir **QL(x, y)** mit 3 Zugriffen auf **tmp1** (statt 9), 3 Zugriffen auf **ML** (statt 9), 3 Multiplikationen (statt 9) und 2 Additionen (statt 8). Wichtig hier ist die Behandlung der Zugriffe auf **tmp1**, da wir Nullen an den Grenzen hinzugefügt haben. Daher ist das **(x, y)**-Element **tmp1 [1+x + (1+y)×(width+1)]** anstelle von **tmp1[x+y×width]**. Das gleiche Verfahren verwenden wir, um **Qw(x, y)** zu berechnen.

Der letzte Schritt ist die Berechnung von **Q'(x, y)**. Das ist identisch mit der Denoise-Optimized-Implementierung.

2.5 Denoise with Threading

Ähnlich dem naiven Ansatz, jedoch aufgeteilt in **vier Threads**: Zwei Produzenten und ein Konsument. Die Graustufenwerte sind bereits berechnet und in **tmp1** gespeichert.

1. **Thread 1** (calculate Ql): Berechnet den Laplace-Filter durch das Durchlaufen aller Pixel und setzt das Ergebnis an der richtigen Stelle in **Ql** (**Ql[x + y**$\times$**width]** ist der Laplace-Filter des Pixels mit den Koordinaten **(x, y)**).

2. **Thread 2** (calculate Qw): Berechnet die Weichzeichnung des Bildes durch das Durchlaufen aller Pixel, berechnet **Qw** gemäß dem obigen Algorithmus und speichert das Ergebnis an der richtigen Stelle in **Qw** (ähnlich **Ql**).

   $\Rightarrow$ **Thread 1** und **Thread 2** arbeiten parallel, um Laplace-Filter und Weichzeichnungswerte zu berechnen und speichern in jedem Schritt die berechneten Werte jedes Pixels in **Ql** und **Qw**.

3. **Thread 3**(calculate result): Dieser Thread durchläuft alle Pixel, wartet auf die Berechnung von **Qw** und **Ql** und berechnet dann das Ergebnis **Q'**. Die Werte werden im Puffer **result** gespeichert.

Hier wird das "busy waiting"durch eine While-Schleife verwendet: Diese Wartezeit erfordert eine :
• For-Schleife am Anfang, um die initialen Werte in **Qw** und **Ql** auf -1 zu setzen (die Werte von Laplace-Filter und Weichzeichnung werden auf keinen Fall -1 sein).
• Die While-Schleife wartet darauf, dass **Qw** und **Ql** gesetzt sind (Qw[index] und Ql[index] sind nicht mehr -1).

Die Kosten des busy waiting sind durch die Gleichung :
$\Rightarrow$ Cost(busy waiting) = Cost(for loop) + Cost(wait)- O(n) + k = O(n) ;
n = n die Anzahl der Pixel
$\Rightarrow$ zusätzlichen Speicherverbrauch = 0
$\Rightarrow$ jedoch werden CPU Ressourcen beansprucht.

Es gibt auch andere Wartungs- und Koordinationsmethoden wie Warten mit Bedingungen/Signalen: Funktioniert in Verbindung mit Mutex und erfordert für jedes Pixel 2 Bedingungen und 2 Mutexe (für **QL** und **Qw**).

$\Rightarrow$ Kosten des Wartens mit Bedingungen = Kosten der Initialisierung + Kosten der Nutzung + Kosten der Zerstörung = O(n); *n die Anzahl der Pixel*.
$\Rightarrow$ Zusätzlicher Speicherverbrauch = Speicher(Mutex) + Speicher(Bedingung)=
$2\times n\times$sizeof(pthread mutex t) + $2\times n\times$sizeof(pthread cond t) = $2\times n\times 40 + 2\times n\times 48$ Bytes = 176n Bytes ; *für ein Bild mit 500 Breite und 500 Höhe würde dies etwa 42 MB betragen*.

2.6 Ausgabe Bild Format

Das denoisierte Bild, das unser Programm zurückgeben muss, hat die Form eines Graustufenbildes, wofür jedes Bild nur ein Byte pro Pixel benötigt. Daher ist das geeignete Dateiformat aus der Netpbm-Familie das PGM P5-Bildformat [2] . Diese Wahl ermöglicht nicht nur eine effiziente Speicherung, sondern spart auch sowohl Zeit als auch Speicherplatz bei der Erstellung und Ausgabe der Datei.

2.7 Nicht benutzte Methode für die Berechnung von GraustufenBild Q

2.7.1 Arrange-then-SIMD

Wir ordnen die Puffer der Pixel **(img)** um, indem wir die roten Werte aller Pixel von 0 bis width×height -1, die grünen Werte aller Pixel von width × height bis 2×width×height-1 und die blauen Werte aller Pixel an den verbleibenden Stellen platzieren. Dann können wir mithilfe von SIMD-Anweisungen Graustufenwerte von 16 Pixeln gleichzeitig berechnen, indem wir in **–mm128** einmal 16 $uint_8$ laden. Diese Methode erfordert eine Schleife von 0 bis 3×width×height, um die Umordnung der Pixel im Puffer img durchzuführen. In der Realität resultiert diese Methode in einer Laufzeit, die fast fünfmal so hoch ist wie die unserer verwendeten Methode. Ab etwa **10.000 Pixeln** wird sie sogar fast 15-mal langsamer (**33 ns** für diese Methode und nur **2 ns** für die normale Methode).

2.7.2 Threads-geteilt

Im Zusammenhang mit Threads entstand die Idee, die Berechnung der Graustufen auf verschiedene Threads zu verteilen. Theoretisch betrachtet ist dies eine vielversprechende Optimierung. In der Praxis zeigt sich jedoch, dass diese Methode erst ab einer sehr großen Anzahl von Pixeln (**1.000.000 Pixel**) Verbesserungen bringt. Im Gegensatz dazu ist sie bei einer geringeren Anzahl von Pixeln (weniger als **500.000**) deutlich langsamer (viermal langsamer bei **90.000 Pixeln**). Dies erweist sich als gänzlich unpraktisch und stellt keine tatsächliche Optimierung dar.

2.7.3 Caching-Style

Während der Berechnung von **Qw** und **QL** benötigen wir für jedes Pixel den Q-Wert von 9 Pixeln. Daher entstand die Idee des Caching-Stils: Mit einem leeren initialen Puffer wird der Q-Wert nur berechnet, wenn er benötigt wird und nicht bereits im Puffer vorhanden ist (Miss). Anschließend wird der korrekte Platz im Puffer gespeichert. Wenn der Q-Wert im Puffer gefunden wird, wird er gelesen und nicht erneut berechnet.
Da alle Q-Werte angewendet werden, bringt diese Methode keine Verbesserung der Laufzeit. In der Realität zeigt diese Methode keine Optimierung; im Gegenteil, die Berechnung ist leicht langsamer. Für **1.000.000 Pixel** beträgt der Unterschied etwa **5 ns** pro Pixel.

3 Korrektheit

3.1 Warum die Kombination aus Laplace-Filter und Weichzeichnung?

Laplace-Filter im Zusammenhang mit Weichzeichnung des Bildes können zu einem denoisierten Bild führen, aber es ist wichtig zu verstehen, warum dies der Fall ist.
Der Laplace-Filter ist ein Hochpassfilter, der auf hochfrequente Details im Bild empfindlich reagiert. Er verstärkt Kanten und feine Strukturen, während er niedrige Frequenzen unterdrückt. Weichzeichnung hingegen reduziert die Hochfrequenzkomponenten im Bild und glättet es.
[1] Die Kombination von Laplace-Filter und Weichzeichnung bietet Vorteile bei der Rauschunterdrückung. Durch den Laplace-Filter werden Kanten betont, während die Weichzeichnung homogene Bereiche glättet. Dies ermöglicht eine ausgewogene Rauschreduktion und Kantenbewahrung. Die Methode ist robuster gegenüber verschiedenen Arten von Bildrauschen und optimiert die Gesamtbildqualität im Vergleich zur alleinigen Anwendung einer Weichzeichnung.
Die unten stehenden Bilder, die von unserem Programm erstellt wurden, zeigen den Unterschied zwischen einem entrauschten Bild (a), einem weichgezeichneten Bild (b) und der Kombination von Laplace mit Weichzeichnung (c):

(a) Noised (b) Weichzeichnung (c) Laplace-Filter

3.2 Wahl von Graustufengewichtfaktoren a, b und c

Das Menschliche Visuelle System **(HVS)** bezieht sich auf das visuelle System des Menschen, einschließlich der anatomischen Strukturen des Auges und des zugehörigen neuronalen Netzwerks im Gehirn. Es spielt eine entscheidende Rolle in der Interpretation von visuellen Informationen.
Das HVS hat eine relevante Beziehung zur Umwandlung eines Bildes in Graustufen, da diese Umwandlung ausschließlich auf die Luminanz fokussiert, wodurch die für das Auge relevanten Helligkeitsinformationen erhalten bleiben. [4]
Typischerweise werden die Gewichtungsfaktoren a, b und c so gewählt, dass die Helligkeit des resultierenden Graustufenbildes dem visuellen Empfinden entspricht.
Unsere Forschung führte uns zur Verwendung von: **ITU-R** Recommend **BT. 709** [3]

$$\Rightarrow a = 0.2126 \; ; b = 0.7152 \; ; c = 0.0722$$

3.3 Korretheit der Implementierung

3.3.1 Mathematische Überprüfung der Korrektheit

Zu Beginn haben wir eine naive Implementierung erstellt und um ihre Korrektheit zu testen, haben wir einen Bildpuffer von 9 Pixeln erstellt und die von unserer Methode ausgegebenen Werte mit den Werten verglichen, die manuell mithilfe der zugrundeliegenden mathematischen Formel berechnet wurden. Dies gab uns einen ersten Eindruck, dass unser Programm auf dem richtigen Weg ist.
Nach und nach vergrößerten wir unseren Bildpuffer auf bis zu 100 Pixel und stellten fest, dass alle Ergebnisse mit der mathematischen Berechnung identisch waren.
An diesem Punkt angelangt, führten wir den folgenden Korrektheitstest durch.

3.3.2 Visuelle Überprüfung der Korrektheit

Durch den Vergleich mit dem originalen Graustufenbild und unserem erstellten Bild mithilfe menschlicher Augen haben wir festgestellt, dass die Entschärfung des Bildes deutlich verbessert wurde. Sowohl die Bildqualität als auch der Farbkontrast bleiben nahezu unverändert. Bei der Analyse verschiedener Bildgrößen sowie unterschiedlicher Farbstärken und Rauschtypen (wie Salt and Pepper oder Gaussian Noise) haben wir herausgefunden, dass die Implementierung korrekt ist.

3.3.3 Methodenvergleichstest

Nachdem wir bewiesen hatten, dass unsere naive Version sowohl mathematisch als auch visuell korrekt ist, begannen wir mit der Optimierung, indem wir drei weitere Methoden mit unterschiedlichen Techniken erstellten, um die Laufzeit zu verbessern.
Um sicherzustellen, dass die drei neuen Programme ebenfalls korrekt sind, haben wir ihre Ergebnisse sowohl mit der naiven Version als auch untereinander verglichen. Dabei stellten wir fest, dass alle Ergebnisse identisch waren, mit einer Fehlerquote von 1 in den Ergebnissen. Diese Fehlerquote ist auf die Interaktion mit Gleitkommazahlen und Casts zurückzuführen, die in einigen Fällen verwendet werden, wie z.B tmp1 in SIMD und tmp2 in Threads. Dies ist jedoch vernachlässigbar und stellt kein Problem dar, da es keinen Einfluss auf unser resultierendes Bild hat

$\Rightarrow$ Mit diesen drei Methoden konnten wir sicherstellen, dass unsere vier verschiedenen Implementierungen identische, korrekte Ergebnisse lieferten und mit den erwarteten Ergebnissen übereinstimmten.

4 Performanzanalyse

4.1 Theroetische Laufzeitanalyse

Im folgenden Abschnitt seien λ_α die benötigte Zeit zur Berechnung von α für ein Pixel.

1. **Graustufenberechnung**
   In allen Methoden bleibt die Graustufenberechnung durch eine Schleife über die Anzahl der Pixel gleich. Es entsteht jedoch zusätzlicher Aufwand für SIMD, um Nullen mit einer Schleife von 2×width hinzuzufügen, sowie einer Bedingung in der Haupt-Schleife.

   $$\Rightarrow T_{graustufe} = \lambda_Q \times n$$
   $$\Rightarrow T_{graustufe}(SIMD) = \lambda_Q \times n + k \times n;\ \text{k : zusätzliche Laufzeit um Nullen Hinzufügen}$$

2. **Berechnung von Qw, QL und Q'**
   Im naiven Algorithmus werden Qw, QL und Q' durch zwei Schleifen über alle Pixel berechnet.

   $$\Rightarrow T_{Ql+Qw+Q'}(Naive) = (\lambda_{Qw} + \lambda_{Speicherung_{Qw}}) \times n + (\lambda_{QL} + \lambda_{Q'} + \lambda_{Zugrif_{Qw}}) \times n$$

   Im optimierten Ansatz werden alle drei in nur einer Schleife berechnet. QL und Qw werden in nur einer 3x3-Schleife berechnet. Wir sparen damit ein $9 \times \lambda_{Zugrif_Q}$ in jeder Pixel.

   $$\Rightarrow T_{Ql+Qw+Q'}(Optimised) = (\lambda_{Qw} + \lambda_{QL} - 9 \times \lambda_{Zugrif_Q} + \lambda_{Q'}) \times n$$

   Im SIMD-Ansatz wird die Berechnung von QL und Qw durch die Verwendung von SIMD geteilt, aber die Berechnung von Q' bleibt gleich.

   $$\Rightarrow T_{Ql+Qw+Q'}(\text{SIMD}) = \left(\frac{\lambda_{Qw} + \lambda_{QL} - 9 \times \lambda_{Zugrif_Q}}{3} + \lambda_{Q'}\right) \times n$$

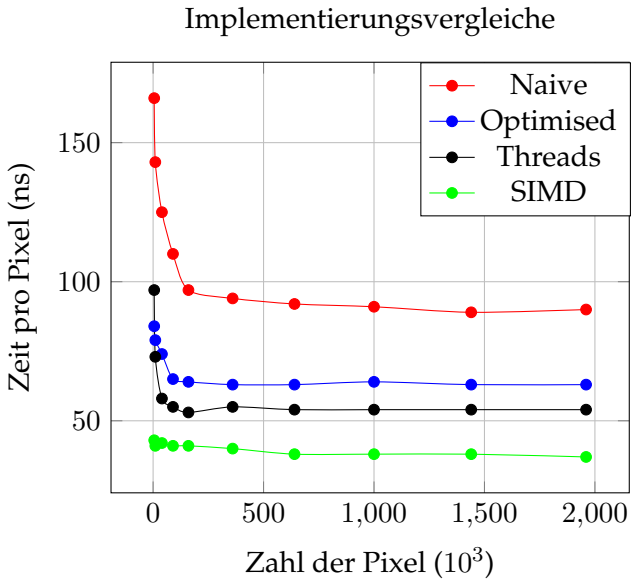
   Im Thread-Ansatz werden sie auf 3 Threads aufgeteilt, die theoretisch parallel arbeiten können. Hierbei wird die Laufzeit dieser Berechnung gleich einer Schleife, allerdings benötigt es zusätzlich eine Schleife, um -1 in den QL- und Qw-Puffern zu initialisieren.

   $$\Rightarrow T_{Ql+Qw+Q'}(Threads) = (max(\lambda_{QL}, \lambda_{Qw}, \lambda_{Q'}) + \lambda_{Speicher}) \times n + \lambda_{init} \times n$$

→ Was deutlich zu bemerken ist, dass die Zeit für ein einzelnes Pixel in allen Ansätzen konstant bleibt. Theoretisch ist der naive Ansatz am langsamsten, und die Zeit wird durch SIMD fast gedrittelt, während die anderen Ansätze deutlich schneller als der naive Ansatz bleiben.

4.2 Reelle Laufzeitanalyse

Für unsere Laufzeittests haben wir jede Methode 10 Mal für jede Messung ausgeführt. Um eine Vielzahl von Fällen abzudecken, beispielsweise mit 1960×10^3, 1440×10^3, 5×10^3 Pixeln usw, haben wir direkt mit zufällig gefüllten Bildpuffern getestet, ohne echte Bilder zu verwenden.
Getestet wurde auf einem System (MSI modern A10RB) mit einem Intel Core i5-10210U, up to 4.1 Ghz ,16 GB Arbeitsspeicher, Ubuntu 22.04.2 LTS. Kompiliert wurde mit gcc mit der Option-O0.



→ Im realen Laufzeitvergleich zeigt der Graph, wie erwartet, dass alle Kurven ab einem Zahl von Pixel nahezu horizontal verlaufen, was darauf hinweist, dass die Zeit pro Pixel bei allen Ansätzen konstant ist.

→ Die naiven Ansatz ist am langsamsten, mit **90-95 ns** pro Pixel, während der SIMD-Ansatz ist wie erwartet am schnellsten mit **37-40 ns** pro Pixel (fast ein Drittel des Naiven wie in der theoretische Teil).
→ Was uns überrascht hat, ist dass die Thread-Ansatz sowohl theoretisch als auch praktisch deutlich schnell ist, mit einer konstanten Zeit von **53-55 ns** pro Pixel. Sie sind sogar schneller als Optimised mit **63-65 ns** pro Pixel

→ Es ist auch zu bemerken, dass die Kurven für Naive, Optimised und Threads von einem anfänglich höheren Wert fallen: von **166 ns** bei Naiv, **84 ns** bei Optimiert und **97** ns bei Threads. Dies liegt an der Zeit, die diese Methoden für die Initialisierung benötigen. Diese Zeit ist nahezu konstant und wird für mehrere Punkte vernachlässigbar.

4.3 Speicherverbrauch

In dieser Tabelle betrachten wir *n=width×height= Zahl von Pixeln*

	Naive	**Optimised**	**SIMD**	**Threads**
img	3n	3n	3n	3n
result	n	n	n	n
Tmp1	n	n	4n	n
Tmp2	n	0	0	4n
Zusätzlicher Puffer	0	0	0	Qw= 4n
Summe	6n	5n	8n	13n

→ Wir stellen fest, dass SIMD und Threads zwar die besten in Bezug auf die Zeit sind, aber in Bezug auf den Speicher wirklich aufwändig sind. Die Erklärung dafür ist:
SIMD benötigt einen Zwischenspeicher für Floats (4n), und Threads benötigen sogar zwei Zwischenspeicher für Floats, während Naive und Optimized nur $\mathbf{uint_8}$ verwenden.
→ Wir kommen zu dem Schluss, dass Optimized am besten geeignet ist, wenn wir ein Gleichgewicht zwischen Laufzeit und Speicherverbrauch erreichen möchten.

5 Zusammenfassung und Ausblick

Das durchgeführte Projekt erläutert und zeigt die Graustufenkonvertierung samt Entrauschung mit Laplace-Filter und Weichzeichnung. Mit diesem Verfahren kann ein Farbiges RGB Bild im 24bpp PPM P6 Format in ein entrauschte Bild des PGM P5 Formates verwandelt werden.
Implementiert wurden dazu vier Algorithmen. Drei dieser Algorithmen nehmen unterschiedliche Optimierungen vor, mit dem Ziel der Laufzeitverbesserung, während der erste eine simple und naive Implementierung zur referenz darstellt. In der Arbeit wurde auch gezeigt das der Algorithmus sich stets an die zugrundeliegenden Mathematischen Berechnung hält und Korrekte Ergebnisse hervorbringt.

Es ist jedoch wichtig zu beachten, dass die Kombination aus Laplace-Filter und Weichzeichnung nicht immer perfekt ist und stark von den spezifischen Merkmalen des Rauschens und des Bildinhalts abhängt. In einigen Fällen kann es zu unerwünschten Artefakten führen oder wichtige Details im Bild beeinträchtigen.
Es gibt auch viele andere Denoising-Techniken, die speziell für Rauschunterdrückung entwickelt wurden und möglicherweise bessere Ergebnisse in Bezug auf die Erhaltung von Details und die Reduzierung von Rauschen. Der am häufigsten verwendete Algorithmus sind z.B Gausscher Weichzeichner (Gaussian Blur) und Medianfilter.
Die Auswahl des besten Entrauschungsalgorithmus hängt von den Merkmalen des Rauschens in Ihren Bildern und Ihren spezifischen Anforderungen ab.

Literatur

[1] MATHIEU AUBRY, SYLVAIN PARIS, SAMUEL W. HASINOFF, JAN KAUTZ, and FREDO DURAND. *Fast Local Laplacian Filters: Theory and Applications*. École normale supérieure ENS, 2014.

[2] Jef Poskanzer. *pgm - Netpbm grayscale image format*, 1989.

[3] ITU Radiocommunication Sector. *Recommendation ITU-R BT.709-6 (06/2015) Parameter values for the HDTV standards for production and international programme exchange*. ITU Radiocommunication Sector, 2015.

[4] Wikipedia. Luma and luminance errors. *Wikipedia*, 2023.