LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 120 – Abgabe zu Aufgabe A207
Wintersemester 2023/24

Tim Knothe Paul Reisenberg

1 Einleitung

1.1 Einführung

Im Folgenden befassen wir uns mit der Umsetzung der Projektaufgabe: *Helligkeit und Kontrast* im Rahmen des Moduls Grundlagenpraktikum: Rechnerarchitektur an der Technischen Universität München. Bei der Aufgabe handelt es sich um eine Problemstellung im Bereich *Image Processing*, welche durch eine Implementierung in C gelöst werden soll. Die Implementierung dient dazu, ein farbiges Bild in Graustufen zu konvertieren und anschließend Helligkeit und Kontrast anzupassen. Die Konvertierung in Graustufen wird beispielsweise in der Computertomografie eingesetzt [1].

1.2 Aufgabenstellung

Wir definieren einen Farbpixel an der Position (x, y) des Bildes P als einen Vektor mit drei Elementen für die Farbkanäle Rot (R), Grün (G) und Blau (B):

$$P_{(x,y)} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \quad (x, y) \in \mathbb{D} = \{0, \ldots, \text{Breite} - 1\} \times \{0, \ldots, \text{Höhe} - 1\}$$

Wir betrachten zunächst die Graustufenkonvertierung: Die Umwandlung eines Pixels in Graustufen geschieht durch das Berechnen eines gewichteten Durchschnitts D mittels der Koeffizienten a, b und c:

$$D = \frac{a \cdot R + b \cdot G + c \cdot B}{a + b + c} \tag{1}$$

Der Pixel in Graustufen und somit das entsprechende Graustufenbild Q ist dann gegeben durch

$$Q_{(x,y)} = D$$

Zur Berechnung von einem Bild, mit neuer Helligkeit verschiebt man den Graustufenwert um eine Konstante $l \in [-255, 255]$, wobei positive Zahlen bedeuten, dass das Bild aufgehellt wird und negative Zahlen bedeuten, dass das Bild abgedunkelt wird. Das heißt, für jeden Graustufenpixel gilt dann

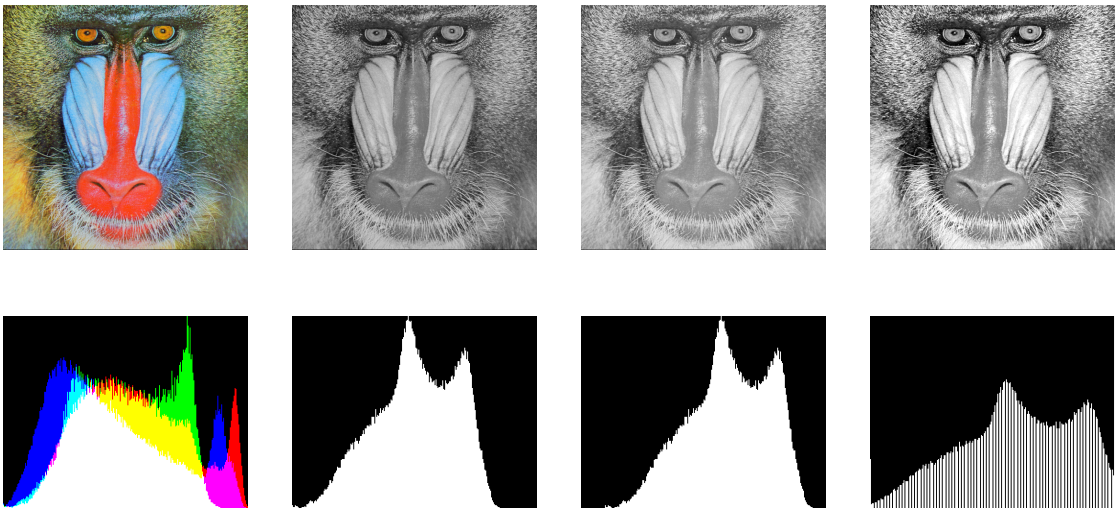
$$Q'_{(x,y)} = \text{clamp}_0^{255}\left(Q_{(x,y)} + l\right)$$

Soll der Kontrast angepasst werden, werden die Graustufenwerte relativ zum Mittelwert um den Faktor k/σ gestaucht oder gestreckt, (wobei im Fall $k = \sigma = 0$ gilt: $k/\sigma = 0$) um ein kontrastärmeres ($k = 0$) bzw. kontrastreicheres Bild ($|k| = 255$) zu erhalten:

$$Q''_{(x,y)} = \text{clamp}_0^{255}\left(\frac{k}{\sigma} \cdot Q'_{(x,y)} + \left(1 - \frac{k}{\sigma}\right) \cdot \mu\right) \tag{2}$$

1.3 Beispielhafte Bildverarbeitung

Im Folgenden wird eine beispielhafte Anwendung des Programms gezeigt, bei der Helligkeit und Kontrast eines Bildes angepasst wurden (Optionen: '–brightness 25' & '–contrast 61'). In der oberen Reihe sind die Bilder nach den jeweiligen Verarbeitungsschritten zu sehen: Original, Graustufenkonvertierung, Helligkeitsanpassung und Kontrastanpassung. In der unteren Reihe werden die Spektren der jeweiligen Bilder als Histogramme dargestellt. Diese zeigen, wie oft die Farb- bzw. Graustufenwerte im Bereich $[0, 255]$ im jeweiligen Bild vorkommen.



(a) Originalbild (b) Graukonvertierung (c) Helligkeitsanpassung (d) Kontrastanpassung

Bei der Helligkeitsanpassung lässt sich erkennen, wie das gesamte Spektrum nach rechts entlang der x-Achse verschoben wird, womit das Bild heller wird. Bei der Kontrastanpassung hingegen wird deutlich, wie sich die Graustufen um den Mittelwert ausbreiten und somit den Kontrast des Bildes erhöhen.

2 Lösungsansatz

2.1 Auswahl des Bildformats

Im Rahmen dieses Projekts wird das 24bpp PPM (P6) Bildformat verwendet, welches Bilder mit 24 Bit pro Pixel (bpp) repräsentiert. Dieses Format speichert Farbbilder, wobei jeder Pixel durch drei Bytes repräsentiert wird – eines für jede Farbkomponente (Rot, Grün und Blau). Sollten in einer Eingabedatei mehrere Bilder enthalten sein, wird nur das erste Bild verarbeitet. Diese Entscheidung basiert darauf, dass in gängigen Formaten wie PNG oder JPEG üblicherweise nur ein Bild gespeichert wird. Zudem vermeidet dies zusätzliche Komplexität im Programm, da der Schwerpunkt der Aufgabe auf der Bildbearbeitung liegt.

Das Ausgabeformat des Programms ist das PGM-Format (Portable Graymap Format), da dieses speziell für Graustufenbilder konzipiert ist. Jedes Pixel in einem PGM-Bild wird durch ein einzelnes Byte dargestellt, was eine direkte Korrespondenz zu den Graustufenwerten ermöglicht und die Verarbeitung der Bilder erleichtert.

2.2 Auswahl der Koeffizienten zur Graukonvertierung

Die Graustufenkonvertierung eines RGB-Bildes beruht auf der Wahl der Koeffizienten a, b und c. Um die Koeffizienten zu bestimmen, wurden mehrere Optionen verglichen. Der naive Ansatz der Durchschnittsmethode berechnet den Graustufenwert durch Mittelung der RGB-Komponenten: $D = \frac{1}{3}R + \frac{1}{3}G + \frac{1}{3}B$. Dieser Ansatz vernachlässigt jedoch die unterschiedliche Sensitivität des Menschlichen Visuellen Systems (HVS) für verschiedene Farben. Eine fortgeschrittenere Methode, die Luminanzmethode [2], berücksichtigt die physiologischen Aspekte des menschlichen Sehens, indem sie den Grünanteil stärker gewichtet, gefolgt vom Rot- und Blauanteil. Das Graustufenbild wird bei der Luminanzmethode mit den Koeffizienten $a = 0.21$, $b = 0.71$ und $c = 0.07$ berechnet.

(a) Original

(b) Durchschnitt

(c) Luminanz

Photo by Hugo Kruip on Unsplash

Der direkte Vergleich der Methoden zeigt, dass das durch die Durchschnittsmethode entstandene Bild dunkler als erwartet ist. Im Vergleich dazu liefert die Luminanzmethode ein ausgewogeneres Bild, das die Helligkeitsempfindlichkeit des menschlichen Auges besser abbildet, indem es den Grünanteil stärker gewichtet. Diesen Erkenntnissen zufolge wird die Luminanzmethode verwendet.

2.3 Implementierung der Wurzelfunktion

Als Teil der Aufgabenstellung wurde eine eigene Wurzelfunktion implementiert, welche für die Bestimmung des Faktors $k/\sigma = k/\sqrt{\text{Var}[Q']}$ in Formel (2) benötigt wird und dabei nur grundlegende mathematische Operationen verwendet. Hierfür wurden zwei Verfahren genauer betrachtet. Das *Heron-Verfahren* und das *Fast-Inverse-Squareroot-Verfahren*.

Das *Heron-Verfahren* [3], auch bekannt als die babylonische Methode, ist eine iterative Methode zur Berechnung der Quadratwurzel einer Zahl a. Ausgehend von einer initialen Schätzung x_0 wird eine Folge von Näherungswerten x_{n+1} durch die rekursive Formel

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

erzeugt. Dieses Verfahren konvergiert unter normalen Bedingungen schnell gegen den tatsächlichen Wert der Quadratwurzel von a.

Das *Fast-Inverse-Squareroot-Verfahren* [4] ist ein Algorithmus zur schnellen Berechnung von $\frac{1}{\sqrt{x}}$. Es wurde berühmt durch seine Verwendung in der Grafikprogrammierung, insbesondere in der Quake III Arena Engine. Der Algorithmus basiert auf einer geschickten Annäherung mittels Bit-Manipulationen. Die grundlegende Idee des Algorithmus ist die folgende Annäherung:

$$y = \text{MagicNumber} - (\text{bit\_shift\_right}(x, 1))$$

wobei MagicNumber eine Konstante ist, die empirisch ermittelt wurde. Da wir für unsere Implementierung ohnehin das Inverse der Wurzel benötigen, bietet sich dieser Algorithmus aufgrund seiner Einfachheit und Schnelligkeit an.

Trotz der potenziell längeren Laufzeit wurde sich für das Heron-Verfahren entschieden. Der ausschlaggebende Faktor war die Genauigkeit des Verfahrens. Da die Wurzel in unserem Anwendungsfall nur einmal berechnet werden muss, fällt die längere Laufzeit nicht signifikant in das Gewicht.

2.4 Implementierung

Das Projekt umfasst drei Implementierungen:

- Eine Hauptimplementierung mittels SIMD (V0)
- Eine optimierte Vergleichsimplementierung ohne SIMD (V1)
- Eine naive Vergleichsimplementierung ohne SIMD (V2)

Die Implementierungen werden von den Funktionen *brightness_contrast_*{Version}*()* umgesetzt, wobei zu beachten ist, dass die Funktionssignatur leicht angepasst wurde und nun ein int zurückgibt, der den Erfolg der Bildbearbeitung angibt. Alle Optionen die von dem Programm unterstützt werden können mittels `'-h' / '--help'` angezeigt werden. Die Implementierungen werden im Folgenden aufsteigend nach ihrer Optimierungsstufe erläutert, da diese jeweils aufeinander aufbauen.

2.4.1 Naive Vergleichsimplementierung (V2)

Die naive Vergleichsimplementierung dient als Referenz für die optimierten Implementierungen. Diese Version implementiert eine einfache und direkte Umsetzung der Graustufenkonvertierung, Helligkeits- und Kontrastanpassung ohne den Einsatz von SIMD-Instruktionen.

Zunächst wird jedes Pixel des Bildes mithilfe der Formel (1) in einen Graustufenwert konvertiert. Die Helligkeitsanpassung erfolgt, indem der Helligkeitswert zu jedem Graustufenwert addiert wird. Hierbei wird sichergestellt, dass die resultierenden Werte im gültigen Bereich von 0 bis 255 bleiben. Falls der Kontrast angepasst werden soll, berechnet die Implementierung zunächst den Mittelwert und die Varianz der Graustufenwerte. Unter Verwendung der Kontrastanpassungsformel (2) wird jeder Graustufenwert dann entsprechend modifiziert.

2.5 Optimierte Vergleichsimplementierung (V1)

Die optimierte Vergleichsimplementierung (V1) stellt eine Verbesserung zu V2 dar, indem unnötige Schleifen, Speicherzugriffe und Rechenoperationen vermieden werden. Zuerst werden die Koeffizienten a, b, c normalisiert:

$$a' = \frac{a}{a+b+c}, b' = \frac{b}{a+b+c}, c' = \frac{c}{a+b+c},$$

Der Graustufenwert kann jetzt durch $D = a' \cdot R + b' \cdot G + c' \cdot B$ berechnet werden. Danach werden vier Fälle unterschieden, je nachdem welche der zwei Optionen –brightness und –contrast vom Nutzer gesetzt wurden. Das erlaubt es, gewisse Schleifen zusammenzufassen und somit die Anzahl der Speicherzugriffe zu minimieren. So wird z.B. die Helligkeitsanpassung parallel zur Graustufenkonvertierung durgeführt. Sollte der Nutzer den Kontrast anpassen wollen, wird die erste Schleife ebenfalls für die Berechnung

des Mittelwerts μ genutzt. Für Kontrastanpassung selbst empfiehlt sich eine Lookup-Tabelle, da sich der Input sowie Output der Kontrastanpassung (Formel 2) auf den Bereich $[0, 255]$ beschränken. Die Kontrastanpassung kann somit nach Erstellung einer Lookup-Tabelle elegant und schnell realisiert werden.

2.6 Hauptimplementierung (V0)

Die Hauptimplementierung hat einen ähnlichen Kontrollfluss wie V1, jedoch mithilfe von SSE-Instruktionen. Um die Effizienz des Verfahrens zu erhöhen, werden die Koeffizienten zuerst wie in V1 normalisiert und dann auf ganzzahlige Koeffizienten $a_s, b_s, c_s \in [0, 256]$ abgebildet, sodass $a_s + b_s + c_s = 256$ gilt. Im Idealfall gilt folgende Äquivalenz:

$$a_s = 256 \cdot a', \quad b_s = 256 \cdot b', \quad c_s = 256 \cdot c'$$

Die Formel zur Graustufenkonvertierung kann nun folgendermaßen umgewendelt werden:

$$\begin{aligned} D &= a' \cdot R + b' \cdot G + c' \cdot B \\ &= \frac{256 \cdot a' \cdot R + 256 \cdot b' \cdot G + 256 \cdot c' \cdot B}{256} \\ &= \frac{a_s \cdot R + b_s \cdot G + c_s \cdot B}{256} \end{aligned}$$

Die letzte Zeile kann mittels Integer -Multiplikation/-Addition und einem 8-Bit-Shift für die Division durch 256 umgesetzt werden. Dadurch kann auf Fließkommaarithmetik verzichtet werden. Zudem wird weniger Platz pro Pixelkonvertierung benötigt, was die Parallelverarbeitung von mehreren Pixeln ermöglicht. Für die gewählten Standardparameter gilt die oben gezeigte Äquivalenz, welche aber für benutzerdefinierte Parameter nicht gelten muss. Deshalb werden die Parameter von der Funktion *convert_coeffs_to_max256()* bestmöglich angenähert, indem folgende Verlust-Funktion minimiert wird:

$$f(a_s, b_s, c_s) = |256a' - a_s| + |256b' - b_s| + |256c' - c_s|$$

Die neuen Koeffizienten werden daraufhin in den __m128i variablen *a_coeff*, *b_coeff*, *c_coeff* 8-mal hintereinander als 16-Bit-Integer gespeichert.

In jedem Schleifendurchlauf werden 16 Pixel (16 x 3 Bytes) geladen. Die Farbwerte werden daraufhin durch Suffle-Operationen als 16-Bit-Integer nacheinander in __m128 Variablen gespeichert (zwei Variablen pro Farbe). Als nächstes werden die Farbwerte, welche sich in dem Bereich $[0, 255]$ befinden mit den jeweiligen angepassten Koeffizienten a_s, b_s, c_s multipliziert. Daraufhin werden die Ergebnisse zusammenaddiert. Die Division lässt sich im nächsten Schritt durch einen 8-Bit-Shift realisieren. Die Graustufenwerte liegen nun in den unteren 8 Bit und sind somit wieder auf einen Bereich von $[0, 255]$ beschränkt. Ein Beweis, dass dieses Verfahren kein Overflow erzeugt findet sich im Abschnitt 3. Die soeben beschriebene Funktionalität übernimmt die Funktion *load_and_convert_to_grey16()*.

Folgende Grafik verdeutlicht den Kern des Verfahrens erneut:

Graustufenkonvertierung mittels SIMD:

a_coeff	= [	a_s	a_s	a_s	a_s	a_s	a_s	a_s	a_s	]
		·	·	·	·	·	·	·	·	
RED	= [	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	]
		+	+	+	+	+	+	+		
b_coeff	= [	b_s	b_s	b_s	b_s	b_s	b_s	b_s	b_s	]
		·	·	·	·	·	·	·	·	
GREEN	= [	G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7	]
		+	+	+	+	+	+	+	+	
c_coeff	= [	c_s	c_s	c_s	c_s	c_s	c_s	c_s	c_s	]
		·	·	·	·	·	·	·	·	
BLUE	= [	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	]
		=	=	=	=	=	=	=	=	
	[	$\_\_D_0$	$\_\_D_1$	$\_\_D_2$	$\_\_D_3$	$\_\_D_4$	$\_\_D_5$	$\_\_D_6$	$\_\_D_7$	]
		« 8	« 8	« 8	« 8	« 8	« 8	« 8	« 8	
GREY	= [	$D_0\_\_$	$D_1\_\_$	$D_2\_\_$	$D_3\_\_$	$D_4\_\_$	$D_5\_\_$	$D_6\_\_$	$D_7\_\_$	]

Wünscht der Nutzer zudem eine Helligkeitsanpassung, wird diese direkt nach der Graukonvertierung durchgeführt, bevor die Pixel in den allozierten Speicherbereich geschrieben werden. Hierfür wird die Variable *brightness_vector* verwendet, in die der Wert l zur Helligkeitsanpassung als 16-Bit-Integer 8-mal hintereinander geschrieben wird. Die Helligkeitsanpassung kann nun durch eine Addition und einer anschließenden Baschränkung auf den Wertebereich $[0, 255]$ durchgeführt werden.

Soll der Kontrast angepasst werden, wird die erste Schleife ebenfalls dazu genutzt, den Durchschnitt μ der Graustufenwerte zu berechnen, um die Zugriffe auf den Speicher zu minimieren. Die Varainz wird im darauffolgenden Schritt ebenfalls mittels SIMD-Instruktionen berechnet. Da sich jetzt auf Gleitkomma-Arithmetik nicht verzichten lässt, können nur vier Pixel gleichzeitig bearbeitet werden. Die Kontrastanpassung selbst wird wie in V1 mithilfe einer Lookup Tabelle realisiert.

2.6.1 Alternativen

Zur Berechnung der Kontrastanpassung wurden mehrere Alternativen getestet. Die Implementierung mithilfe einer Lookup-Tabelle hat sich jedoch auch gegenüber einer SIMD-Implementierung behauptet und wird deshalb in V0 und V1 verwendet.

Eine Alternative zu dem SIMD-Verfahren (V0) ist das Laden von nur 16 Byte pro Bearbeitungsschritt, wodurch jedoch nur fünf Pixel gleichzeitig bearbeitet werden können. Dieses Verfahren ist deutlich ineffizienter, da nicht nur mehr Speicherzugriffe entstehen, sondern diese auch größtenteils unaligned sind.

Eine Implementierung der SSE-SIMD-Implementierung (V0) mittels AVX Instruktionen wirkt auf den ersten Blick vielversprechend. Jedoch beschränkt die 128-Bit-Lane-Beschränkung innerhalb der *__m256* Variablen von AVX das Optimierungspotential. Aus diesem Grund wurde sich vorerst gegen eine Implementierung mittels AVX entschieden.

3 Korrektheit

Die Entscheidung, die Analyse auf die Korrektheit und nicht auf die Genauigkeit zu fokussieren beruht darauf, dass das menschliche visuelle System (HVS), nicht in der Lage ist, minimalen Abweichungen der Graustufen zu erkennen. Die Korrektheit unseres Programms begründen wir im Folgenden sowohl assertiv als auch nicht-assertiv.

3.1 Korrektheit der SIMD-Hauptimplementierung

Der kritische Punkt der Hauptimplementierung besteht darin, dass die Koeffizienten zur Graukonvertierung auf Integer in dem Bereich $[0, 256]$ abbildet werden. Wie bereits erwähnt kann das menschliche Auge diesen Informationsverlust nicht wahrnehmen, jedoch muss gezeigt werden, dass die Konvertierung korrekt funktioniert und es nicht zu einem Overflow innerhalb der *__m128i* Variablen kommen kann. Dass die Summe der angepassten Koeffizienten 256 ist, wird durch einen Check der Funktion *convert_coeffs_to_max256()* sichergestellt.

Für die Konvertierung zum Graustufenwert stehen pro Pixel jeweils 16 Bit zur innerhalb der *__m128i* Variablen zur Verfügung. Es ist also zu zeigen, dass das Ergebnis der Konvertierung $a_s \cdot A + b_s \cdot G + c_s \cdot B$ immer in 16 repräsentierter ist:

$$
\begin{aligned}
& a_s \cdot A + b_s \cdot G + c_s \cdot B \\
& \leq a_s \cdot 255 + b_s \cdot 255 + c_s \cdot 255 \\
& = 255 \cdot (a_s + b_s + c_s) \\
& = 255 \cdot 256 \\
& = 65280 \\
& < 2^{16} - 1
\end{aligned}
$$

Da für die Graustufen nur 8 Bit relevant sind, ist ein Overflow auf 16 Bit durch die Helligkeitsanpassung, welche sich auf den Bereich [-255, 255] beschränkt ebenfalls ausgeschlossen.

3.2 Validierung der Pixelkonvertierung

Zur Überprüfung der korrekten Implementierung der Pixelkonvertierung wurden ebenfalls Tests entwickelt. Diese sind durch `./testing/pixel_conversion_test.sh` aus dem Verzeichniss `/Implementierung` ausführbar. Die Tests umfassen das Ausführen des Programms mit verschiedenen Testbildern und Beispielargumenten. Anschließend erfolgt ein Abgleich der erzeugten Ergebnisse mit Referenzbildern. Hierbei wird eine Toleranz von einem Graustufenwert pro Pixel zugelassen, um geringfügige Abweichungen zu berücksichtigen. Dieser Ansatz stellt sicher, dass die Pixelkonvertierung präzise und gemäß den definierten Anforderungen funktioniert.

3.3 Tests für fehlerhafte Benutzereingaben

Um die Robustheit des Programms zu gewährleisten, wurden umfassende Tests implementiert, die das Verhalten des Programms bei fehlerhaften Nutzereingaben überprüfen. Diese Tests können vom Verzeichnis `/Implementierung` aufgerufen werden und gliedern sich in zwei Kategorien:

1. **Fehlerhafte Verwendung von Kommandozeilenargumenten:** Diese Tests überprüfen die korrekte Eingabe und Verarbeitung der Kommandozeilenargument. Sie können mit dem Befehl `./testing/arguments_test.sh` ausgeführt werden.
2. **Fehler im PPM P6 Format:** Hierbei konzentrieren wir uns auf die korrekte Handhabung des PPM P6 Bildformats. Diese Tests sind über `./testing/format_test.sh` zugänglich.

Zu den spezifischen Tests für Benutzereingaben zählen:

- Überprüfung der Einhaltung von Wertebereichen für Argumente wie *brightness* und *contrast*.
- Validierung des Vorhandenseins und der Lesbarkeit der Eingabedatei.
- Erkennung von fehlerhaft formatierten Argumenten, z.B. bei nicht interpretierbaren Zahlenwerten.

Für die Überprüfung des korrekten Einlesens von Bildern wurden mehrere Testbilder mit unterschiedlichen Formatverletzungen erstellt. Diese Tests umfassen:

- Erkennung und Validierung des korrekten Bildformats.
- Angemessener Umgang mit Kommentaren im Bildformat.
- Überprüfung der Parameter auf korrekte Anzahl und Gültigkeit.
- Einhaltung korrekter Abstände (Spacing) im Bildformat.

Durch diese Tests stellen wir sicher, dass unser Programm den Anforderungen und Richtlinien des PPM-Formats entspricht, wie sie in der PPM Spezifikation dargelegt sind.

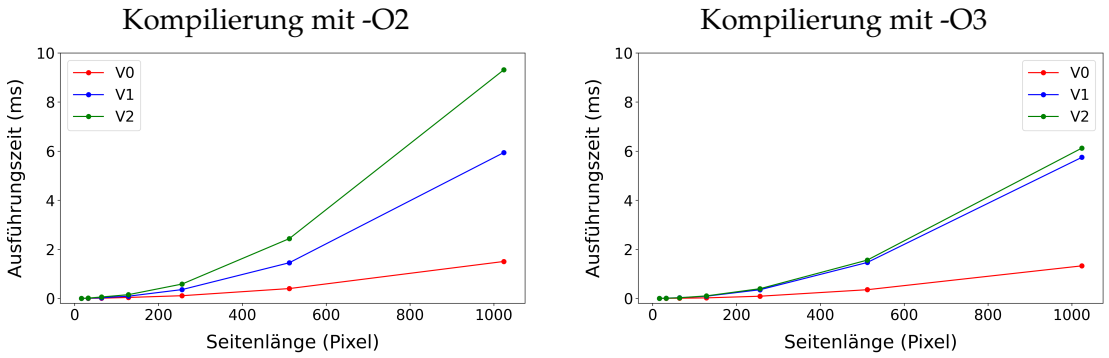
4 Performanzanalyse

Die Performanzanalyse unserer Implementierungen bietet Einblicke in die Leistungsunterschiede zwischen den verschiedenen Versionen und zeigt auf, wie Compiler-Optimierungsstufen die Ausführungszeiten beeinflussen. Folgende Implementierungen wurden getestet:

- Hauptimplementierung mit SIMD-Optimierung (V0)
- Optimierte Vergleichsimplementierung ohne SIMD (V1)
- Naive Vergleichsimplementierung ohne SIMD (V2)

Die Tests wurden auf einem System mit einem i5 10210U Prozessor und 16 GB Arbeitsspeicher unter Manjaro Linux x86_64 (Kernel 5.14-49-1-Manjaro) und mit der GCC-Version 12.1.0 durchgeführt. Als Maßstab dient die durchschnittliche Ausführungszeit unter Verwendung der Optionen '–brightness 10 –contrast 10 -B1000' auf quadratischen Bildern mit zufälligen Farbwerten. Bei kleineren Bildern wurde die Anzahl der Ausführungen erhöht um eine Ausführungszeit von mehr als einer Sekunde sicherzustellen.

4.1 Ergebnisse der Performance-Tests



Die Ergebnisse der Kompilierung mit der Optimierungsstufe -O2 zeigen, dass die Hauptimplementierung (V0) lediglich etwa ein Viertel der Zeit der Vergleichsimplementierung V1 und ein Sechstel der Zeit der naiven Vergleichsimplementierung V2 benötigt. Dies entspricht einem Speedup von Faktor 4 bzw. 6.

Bei der Kompilierung mit der Optimierungsstufe -O3 lässt sich keine signifikante Veränderung in der Performance von V0 und V1 feststellen. Die Ergebnisse der naiven Vergleichsimplementierung (V2) nähern sich jedoch der Leistung der optimierten Vergleichsimplementierung (V1) an. Die SIMD-Hauptimplementierung (V0) weist somit immer noch einen Speedup von ca. 4 im Vergleich zu den anderen beiden Versionen auf.

4.2 Interpretation der Ergebnisse

Die erhaltenen Ergebnisse zeigen deutlich die Vorteile der SIMD-Optimierung und des gewählten Ansatzes in der Hauptimplementierung (V0). Durch die parallele Datenverarbeitung wird eine erhebliche Zeitersparnis gegenüber den Vergleichsimplementierungen erreicht. Die Ergebnisse durch die Optimierungsstufe -O3 lassen vermuten, dass der Compiler das gleiche Optimierungspotential in der Parallelverarbeitung wie in V1 von verschiedenen Schritten entdeckt und umgesetzt hat. Diese Vermutung konnte mittels einer Analyse des disassemblierten Programms mit objdump bestätigt werden. Allerdings bleibt die SIMD-basierte Implementierung unangefochten führend in Bezug auf die Ausführungszeit.

5 Zusammenfassung und Ausblick

In diesem Projekt haben wir uns mit der Implementierung eines Bildverarbeitungsprogramms befasst, welches Graustufenkonvertierung, Helligkeits- und Kontrastanpassung unterstützt. Dabei wurden drei Versionen implementiert: Eine naive Vergleichsimplementierung, eine optimierte Vergleichsimplementierung und die SIMD-Hauptimplementierung. Bei der SIMD-Hauptimplementierung konnten wir durch einen kreativen Ansatz der Koeffizientenabbildung auf Integer und dem parallelen Verarbeiten von mehreren Pixeln einen erheblichen Geschwindigkeitsgewinn erreichen. Die Korrektheit des Programms wurde sowohl durch assertive als auch nicht-assertive Methoden überprüft. Als zukünftige Erweiterungen könnten Implementierungen unter Einsatz von AVX oder Multithreading in Betracht gezogen werden, um die Performance weiter zu steigern.

Literatur

[1] Wolfgang Reith. Computertomographie. *Diagnostische und interventionelle Radiologie*, pages 29–35, 2011. `https://link.springer.com/chapter/10.1007/978-3-540-87668-7_4`, visited 2024-01-26.

[2] David Connah, Graham D Finlayson, and Marina Bloj. Seeing beyond luminance: A psychophysical comparison of techniques for converting colour images to greyscale. In *Color Imaging Conference*, pages 336–341, 2007.

[3] Matthias Richter. Radizieren mit dem heron-verfahren. *A+ A*, 2:2an, 2011. `https://matthias.richter-edu.de/Skripte/RadizierenHeron.pdf`, visited 2024-01-25.

[4] Charles McEniry. The mathematics behind the fast inverse square root function code. *Tech. rep.*, 2007. `https://0x5f37642f.com/documents/McEniryMathematicsBehind.pdf`, visited 2024-01-25.