LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 143 – Abgabe zu Aufgabe A208
Wintersemester 2023/24

Ar Pazari Edera Ndoj Demil Omerovic

1 Einleitung

Ein wichtiger Algorithmus im Bereich der Bildverarbeitung ist die Gamma-Korrektur, die die Anpassung von Helligkeit und Kontraste in digitalen Bildern ermöglicht[12]. Unsere Aufgabe umfasste die eingehende Untersuchung dieses Algorithmus, sowohl in der theoretischen als auch in der praktischen Betrachtung, um ihn in verschiedenen Implementierungen praktisch umzusetzen.

Abbildung 1: Visualisierung des Algorithmus

1.1 Funktionsweise des Algorithmus

Die Bestandteile der Funktionsweise unseres Projektes werden schrittweise erklärt: $P(x, y)$ repräsentiert vektorenweise ein Farbpixel an der Position (x, y) im Bild P, wobei die RGB-Werte zur Darstellung dienen. Der Bildbereich ist durch $D = \{0, \ldots, \text{Breite} - 1\} \times \{0, \ldots, \text{Höhe} - 1\}$ definiert, wobei jedes Pixel seine Dimensionen einnimmt.

1. Das erste Berechnungsschritt ist die Umwaldung in Graustufen[11]. Es erfolgt durch einen gewichteten pixelsweise Durchschnitt:

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

wobei R, G, B die drei Farbkanäle Rot, Grün und Blau repräsentieren. Für die Koeffizienten a, b, c werden üblicherweise die folgenden Werte verwendet:

$$a = 0.299, b = 0.587 \text{ und } c = 0.114.$$

Grund, warum die oben definierten Werte häufig verwendetet sind, liegt in der menschlichen Wahrnehmung von Farben und Helligkeit[10] und die Eigenschaften des Menschlichen Visuellen Systems (HVS)[7]. Diese Koeffizienten basieren auf der Empfindlichkeit

des menschlichen Auges gegenüber verschiedenen Farben, wobei Rot (R) etwa 30% zur Helligkeit beiträgt, Grün (G) etwa 59% und Blau (B) etwa 11%.

Methode / Standard	Rot (R)	Grün (G)	Blau (B)	Gamma
ITU-R BT.601-7 (SDTV)	0.299	0.587	0.114	2.2
ITU-R BT.709-6 (HDTV)	0.2126	0.7152	0.0722	2.2
ITU-R BT.2020 (UHDTV)	0.2627	0.6780	0.0593	2.4
Digitale Fotografie	0.3	0.59	0.11	2.2
Gleichmäßige Gewichtung (Einfach)	0.333	0.333	0.333	2.2

Tabelle 1: Koeffizienten und Gamma-Werte für die Graustufenkonvertierung[6]

2. Das Ergebnis des ersten Schrittes ist der einzelne Intensitätswert D, der die Helligkeit dieses Pixels darstellt:

$$Q_(x, y) = D \tag{2}$$

3. Anschließend unterzieht sich das Graustufenbild einer Gammakorrektur:

$$Q'_(x, y) = \left(\frac{Q_(x, y)}{255}\right)^{\gamma} \cdot 255 \tag{3}$$

wobei $\gamma \in [0, \infty)$ eine vom Benutzer festgelegte Fließkommazahl ist. Wenn $\gamma < 1$ erscheint die Ausgabedatei heller; bei $\gamma > 1$ wird sie dunkler. Wenn $\gamma = 1$ werden die ursprünglichen Pixelwerte unverändert[2]. Der am häufigsten verwendete γ-Wert ist 2.2. Es hilft, die Anzeige von Bildern auf Geräten wie z.B Monitoren zu korrigieren, die in der Regel eine nicht lineare Reaktion auf Intensitätsstufen aufweisen. Diese Wahl resultiert aus historischen Standard [13].

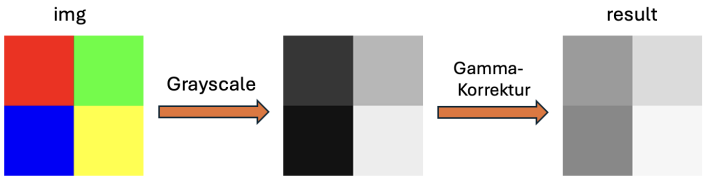


Abbildung 2: Visualisierung des Algorithmus

Nach den beiden Schritten besteht die Ausgabedatei aus Pixeln, deren Intensitätsinformationen von Schwarz bis Weiß abhängig von das Gammawert variieren.

1.1.1 Die Eingabedatei

Die Eingabedatei ist keine Textdatei, sondern ein **24bpp PPM (P6) Bilder**[9]. Die Benutzung von P6 PPM Dateien ermöglicht eine einfache und klare Darstellung von Farbinformationen. Die Bestandteile einer P6 Eingabedatei sind:
-**Magischer Wert**: Identifiziert das Format der PPM Bilder (hier: "P6", codiert in ASCII)

-**Breite und Höhe des Bilds** (codiert in ASCII)
-**Maximalwert des Farbwertes** (codiert in ASCII)
-**Die Werte der Pixel**, die binär codiert werden, erhalten jeweils drei Bytes für die Farkanäle R, G, B. Mit 24 Bits pro Pixel beträgt die maximale Anzahl von darstellbaren Farben 2^{24}, was $16.777.216$ Farben entspricht. Diese umfassende Farbpalette ermöglicht die Darstellung von detailreichen Bildern.

1.1.2 Die Ausgabedatei

Die Ausgabedatei sollte in ein effizientere **Netpbm-Format** repräsentiert sein. Der Wechsel von (P6) Input Format zielt darauf ab, die Dateigröße zu reduzieren und die Repräsentation der Ausgabedatein zu vereinfachen. Da die Resultät ein Graustufenbild ist und eine geringere Dateigröße hat, wurde die Nutzung von **P5 PGM (Portable Graymap Format)**[8] Dateien die Interpretation des Pixels erleichtern. Im Gegensatz zu P6 Dateien, es wird nur ein Byte benötigt, um die Intensitätsinformationen jedes Pixels zu speichern. Deswegen wurde die Dateigröße um zwei Drittel reduziert, was zu einer effizienteren Speicherung führt. Als Binärdateiformat ist P5 auch nicht lesbar für Menschen, aber das Lesen und Schreiben Operationen sind im allgemeinen schneller. Eine andere PGM Format ist **P2**[4], welches nur eine begrenzte Anzahl von Graustufen ermöglicht und somit nicht die notwendige Genauigkeit für die Gamma-Korrektur bietet. Deswegen wurde **P5** gewählt, um eine präzisere Repräsentation der Bildinformation sicherzustellen.

Die Bestandteile einer P5 Ausgabedatei sind:
-**Magischer Wert**: Identifiziert das Format der PGM Bilder (hier: "P5", codiert in ASCII)
-**Breite und Höhe des Bilds** (codiert in ASCII decimal)
-**Maximalwert des Farbwertes** (codiert in ASCII decimal: Hier 255, da die initiale Pixels als $uint8$ dargestellt sind. Daher reicht der Bereich 0-255, um die Helligkeitsstufen der korrigierten Pixel effizient zu repräsentieren.)
-**Die Werte der Pixel**, die binär codiert werden, erhalten jeweils ein Byte für deren Intensitätsinformationen.

1.1.3 Exponentialfunktion mittels grundlegender mathematischer Operationen

Eine weitere Herausforderung ist eine Möglichkeit zu finden, die die in dritter Formel angewendete Exponentialfunktion (3) nur mittels grundlegender mathematischer Operationen zu berechnen. **Unter Verwendung der babylonischen Methode** [1] bzw. Herons Methode kann man die Exponentialfunktion ersetzen wie folgt:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right), \quad n \in \mathbb{N} \tag{4}$$

wobei x_n die Anfangsvermutung (z.B die Zahl oder die Hälfte der Zahl, deren Quadratwurzel gesucht wird). a ist die Zahl, deren Quadratwurzel bestimmt werden soll.

Die sukzessiven Näherungen $x_n + 1$ werden gemäß der oben definierten Iterationsschritte berechnet. Diese Formel wird so lange wiederholt, bis der Prozess konvergiert, also bis ein konsistentes sowie klar definiertes Ergebnis erreicht wird. Eine zehnmalige Wiederholung der Iterationsschritt ergibt eine gute Annäherung an die Quadratwurzel.

Eine andere Möglichkeit ist **die Taylor Reihe** [3]. Für die Berechnung von a^b verwenden wir die Taylor Reihe für die Exponentionalfunktion:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \tag{5}$$

Hier wurde a^b als $e^{b \cdot \ln(a)}$ approximiert, wobei $\ln(1+x) \approx x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$ den natürlichen Logarithmus approximiert. Diese Methode ermöglicht eine ungefähre Berechnung von a^b unter Verwendung einer endlichen Anzahl von Termen, was die Komplexität reduziert.

2 Lösungsansatz

Implementiert wurde der Gamma-Korrektur Algorithmus auf 5 verschiedene Weisen, die sich alle in den unten detaillierten Ansätze widerspiegeln werden. Sie unterscheiden sich hinsichtlich Optimierungen, mathematischer Vorgehensweise und Reihenfolge der Datenverarbeitung und Zwischentransformationen voneinander.

2.1 Implementierung 1: Exponentialfunktion durch math.h

Die erste Implementierung, dass `gamma_V0.c` heißt, wurde mit den Methoden aus der Mathematikbibliothek (`math.h`) zur Berechnung der Exponentialfunktion umgesetzt. Das Pixelarray eines Eingabebildes wurde durchläuft und für jedes Pixel den Gamma Korrekturalgorithmus angewendet wurde. Zuerst erfolgt eine Umwandlung in Graustufen, indem die RGB-Farbwerte jedes Pixels entsprechend den gegebenen Koeffizienten a, b und c gewichtet werden. Anschließend wird die Gamma Korrektur durchgeführt, wobei die Exponentialfunktion aus der `math.h`-Bibliothek verwendet wird.

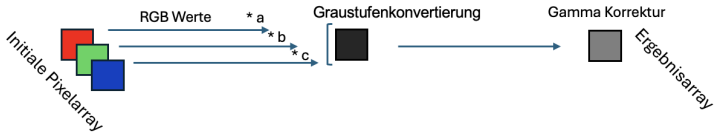


Abbildung 3: Visualisierung einer Beispielverwendung der `gamma_V0.c`

Die ausgewählte Funktion `pow()` minimiert mögliche Rundungsfehler, die bei Fließkommazahlen auftreten können. Das Ergebnis wird auf den Bereich von 0 bis 255 begrenzt, um gültige Pixelwerte zu erhalten. Die resultierenden Pixelwerte werden im Ergebnisarray gespeichert, um die bearbeitete Ausgabedatei zu rekonstruieren. Die oben genannte Operationen wurden in die Funktion `gamma_V0` implementiert.

2.2 Implementierung 2: Exponentialfunktion durch grundlegender mathematischer Operationen

In der zweiten Implementierung, namens `gamma_V1.c`, wurde der Algorithmus mithilfe der grundlegenden mathematischen Operatoren und zwei Hilfsfunktionen realisiert. Die erste Hilfsfunktion, `double integerPower(double base, int exponent)`, multipliziert die Basis mit sich selbst entsprechend dem ganzzahligen Exponenten (Vorkommastellen), während die zweite, `double fractionalPower(double base, double fractionalPart)`, die lineare Interpolation (6) und die Babylonische Methode (4) für nicht-ganzzahlige Exponenten (Nachkommastellen) anwendet. Die Wahl zwischen Interpolation und Babylonische Methode für nicht-ganzzahlige Exponenten hängt davon ab, ob der `fractionalPart` kleiner oder größer 0.5 ist.

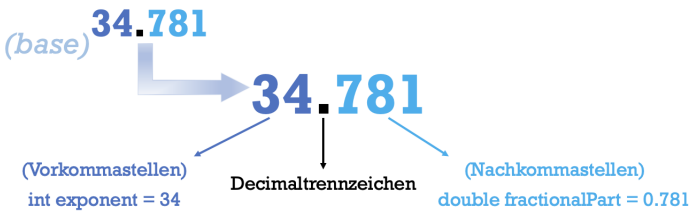


Abbildung 4: Die Trennung der γ-Wert in zwei Argumente für die Hilfsfunktionen

Für die `fractionalPart` Werte kleiner als 0.5, erfolgt die Interpolation nach der Formel:

$$1 + fractionalPart * (base - 1). \tag{6}$$

Die Formel(6) entspricht einer linearen Interpolation zwischen 1 (dem Ergebnis für eine Potenz von 0) und der Basis selbst (dem Ergebnis für eine Potenz von 1). Das ermöglicht eine glatte Übergangsberechnung wenn `fractionalPart` nahe bei 0 liegt. Wenn es jedoch 0.5 oder größer ist, wird die Babylonische Methode[1] zur Berechnung der Quadratwurzel verwendet. Schließlich wurde eine andere Funktion namens `power` implementiert, die die Hilfsfunktionen aufruft, um das Ergebnis der Exponentialsfunktion zu berechnen, ohne die in `math.h` inklusierte `pow` Funktionen zu verwenden. Hier wurde das Exponent in 2 Teilen getrennt: `int exponent` und `double fractionalPart` und genutzt als Parametern in beide Hilfsfunktionen. Die Iteration durch das Pixelarray und die Transformationen erfolgen ähnlich wie in der ersten Implementierung in einer Funktion namens `gamma_V1`. Insgesamt bietet diese Implementierung eine präzise Anwendung der Gamma-Korrektur und Flexibilität, die Berechnung der Exponentialfunktion ohne externe Bibliotheken durchzuführen.

2.3 Implementierung 3: Trennung der zweiten Haupttransformationen

Um die Anforderungen der Aufgabestellung vollständig zu erfüllen, ist es notwendig, eine andere Implementierung zu realisieren, welcher ein Bild erst in Graustu-

fen total konvertiert und anschließend eine Gammakorrektur auf allen Pixeln ausführt. In `gamma_V2.c` sind zwei Funktionen vorhanden, die die Transformationen der Farbpixel getrennt voneinander durchgeführen. Die Graustufenkonvertierung erfolgt durch die Funktion `convertToGrayscale()`, während die Gamma-Korrektur durch `applyGammaCorrection()` durchgeführt wird. Im Unterschied zu den vorherigen Implementierungen wird in dieser Version die Exponentialfunktion durch die Funktionen `exp` und `log` aus der `math.h`-Bibliothek ersetzt. Die Formel(3) wurde dann so aussehen:

$$Q'_(x,y) = \exp\left(\log\left(\frac{Q_{(x,y)}}{255}\right) \cdot \gamma\right) \cdot 255 \tag{7}$$

Abgesehen von diesen gibt es noch eine weitere Methode namens `gamma_V2()`, die nur die beiden Hilfsmethoden nacheinander anruft. Der Parameter `img` von `gamma_V2()` ist auch der Eingabeparameter für `convertToGrayscale()`, das eine PPM-Datei nimmt und die Grauskalierungstransformation durchführt, um `first_img` zu erstellen. Anschließend wird `first_img` von der Funktion `applyGammaCorrection` verwendet, um für jedes Pixel eine Gamma-Korrektur durchzuführen, was schließlich zur endgültigen Ausgabe `result` führt. Eine detallierte Darstellung (siehe Abbildung 5) der Funktionsweise der `gamma_V2.c` ist hilfreich, um zu verstehen, in welche Aspekten `gamma_V2.c` sich von den ersten Ansätzen unterscheidet.

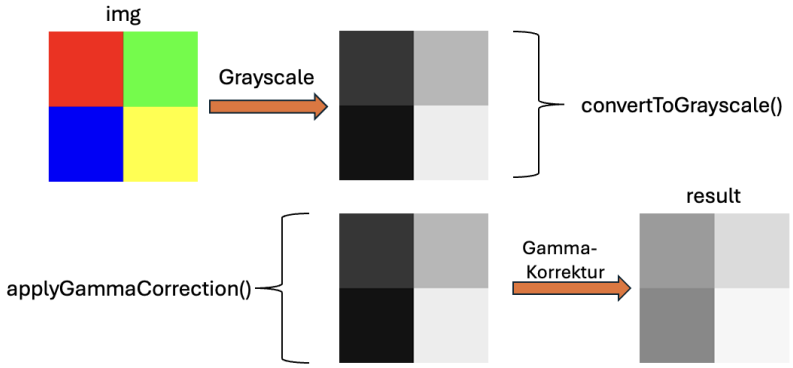


Abbildung 5: Visualisierung der Funktion in `gamma_V2.c`

In dieser Implementierung sind die Transformationsschritte klar voneinander getrennt, und die Verwendung von `exp` und `log` stellt eine andere mathematische Herangehensweise dar.

2.4 Implementierung 4: Hybrider Ansatz - Kombination von SIMD und skalaren Operationen

Die vierte Implementierung zielt darauf ab, die Verarbeitung von RGB-Bilddaten durch die Nutzung der SIMD-Befehlssätze weiter zu optimieren, wobei ein spezieller Fokus

auf die effiziente Anordnung von Pixelwerten und die Kombination von SIMD- und Skalarberechnungen liegt. Im ersten Schritt werden die RGB-Pixelwerte als Integervariablen eingelesen und dann so neu ausgerichtet, dass in einer SIMD-Variablen mit 128 Bits vier rote, vier grüne und vier blaue Werte sequenziell gespeichert werden (siehe Abbildung 6). Die restliche 1.3 Byte werden zwar geladen und mit berechnet aber sie werden im letzten Schritt nicht gespeichert. Sie werden in der darauffolgenden Iteration behandelt.

Trotz der Einschränkung, dass nur 5,3 RGB-Pixel in eine 128-Bit-Variable passen, ermöglicht diese Konfiguration eine effiziente Verarbeitung. Dabei werden die RGB-Werte zunächst in ein `__m128i`-Variable geladen. Anschließend erfolgt eine Konvertierung von `__m128i` zu `__m128` (float), um die Berechnungen für die Graustufenkonvertierung in SIMD durchzuführen. Da die vorhandenen SIMD-Bibliotheken wie Intels SSE (Streaming SIMD Extensions)[5] keinen spezifischen Befehl *pow*-Funktion bieten, erfolgt die Berechnung der Gammakorrektur skalar. Ein wichtiger Aspekt ist das Alignment der Daten. Die Entscheidung, die Daten auf 4 Bytes auszurichten und nicht auf 5, resultiert aus der Überlegung, dass bei einer Breite, die nicht durch 5 teilbar ist, in jeder Zeile vier skalare Berechnungen notwendig wären. Durch die Anordnung in 4-Byte-Blöcken reduziert sich die Anzahl der erforderlichen Skalarberechnungen auf drei, was zu einer effizienteren Verarbeitung führt.

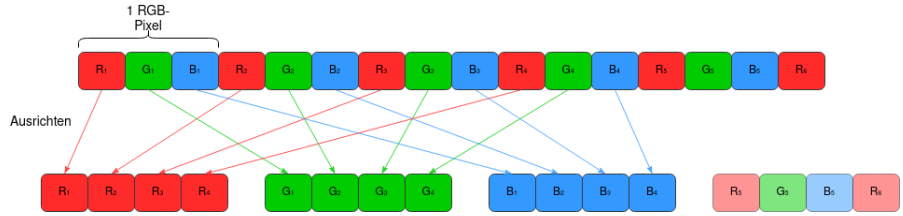


Abbildung 6: Neuausrichtung der einzelnen RBG-Pixel

2.5 Implementierung 5: Optimierung - SIMD mit Approximationsfunktionen

Dieser Ansatz ist aufbauend auf dem vorherigen, nur mit der Unterscheidung, dass anstatt teilsakalarer nun vollständige SIMD Berechnungen durchgeführt werden. Ein wesentlicher Unterschied ist, das dies hier mit Hilfe von Approximationsfunktionen innerhalb des SIMD-Frameworks erreicht wurde. Durch diesen Ansatz ergibt sich ein signifikanter Unterschied zu dem bereits beschriebenen Ansatz (2.4), um die Gamma-Korrektur x^γ, effizienter durchzuführen. Die Potenzfunktion kann mit Exponentialfunktionen und Logarithmen als $a^b = e^{b \cdot \ln(a)}$ ausgedrückt werden. Da es ebenfalls keine Intel SEE Intrinsics für die ln- weder noch für die e-Funktion gibt, wurden dies durch SIMD-Berechnnung durch Approximation mit der Taylor-Reihe erreicht (5). Die Implementierung mittels SIMD und Taylor-Reihen-Approximation für die Gamma-Korrektur stellt zwei Haupttherausforderungen dar: Erhaltung der Berechnungspräzision bei gleichzeitiger Optimierung der Leistung. Die Präzision der Approximation hängt von der

Anzahl der berücksichtigten Taylor-Reihen-Terme ab, was in einem SIMD-Kontext zu einem Trade-off zwischen Genauigkeit und Rechenzeit führt. Zudem erfordert die effiziente Nutzung von SIMD eine sorgfältige Organisation der Berechnungen, um die parallele Datenverarbeitung zu maximieren und die Overheads zu minimieren.

Herangehensweise:

- *Optimierung der Termzahl:* Eine ausgeglichene Anzahl der Taylor-Reihen-Termen sollte bestimmt werden, um eine akzeptable Genauigkeit bei minimalem Leistungsverlust zu erreichen.
- *Effiziente Vektorisierung:* Berechnungen werden für die SIMD-Architektur optimiert, um Parallelität und Effizienz zu verbessern.
- *Wertebereichsanpassung:* Normalisierung der Eingabewerte, um die Effektivität der Taylor-Reihe zu steigern und die Konvergenz zu beschleunigen.

Durch diese Ansätze wird die SIMD-basierte Gamma-Korrektur effizienter, ohne dass signifikante Präzisionsverluste hingenommen werden müssen.

3 Korrektheit

Es ist sinnvoll, die Korrektheit des Algorithmus zu überprüfen, um sicherzustellen, dass die neu generierten Pixel der Ausgabedatei korrekt sind. Um die Korrektheit zu testen, können eigene PPM-Dateien manuell erstellt und als Parameter an die zu implementierende Funktion zu übergeben. Die resultierten Werte lassen sich dann einzeln mit den Ergebnissen der mathematischen Berechnungen anhand gegebenen Formeln vergleichen. Die Korrektheitskontrolle wurde erfolgreich für kleine Bilder durchgeführt, wobei die korrekten Werte berechnet wurden. Es ist jedoch unpraktisch, für große Bilder sehr viele Werte einzeln zu berechnen. Um die bestmöglichen Ergebnisse zu erzielen, ist eine sorgfältige Berücksichtigung sowie umfangreiche Experimente und Tests mit verschiedenen Einstellungen erforderlich. Es wurden die Ausgaben der verschiedenen Ansätze verglichen, um die Korrektheit zu testen(siehe Abbildung (8)). Es ist leicht erkennbar, dass die Aufgabestellung erfüllt wurde: Die Eingabedatei wurde erstmals in Graustufenform bereitgestellt und schließlich dann eine Gamma Korrektur angewandt. Nach umfangreichen Tests war es deutlich, dass die Ausgabedateien in den Implementierungen, in denen keine approximierte Berechnung für die Potenz des γ-Wertes verwendet wurde, ähnlich und genauer sind. Die Verwendung einer approximierten Potenzberechnung führt im Vergleich zu den präziseren Potenzberechnungen zu einer gewissen Ungenauigkeit.
Für kleine γ-Werte ist `gamma_V4.c` die richtige Wahl, aber ansonsten nimmt die Genauigkeit langsam ab und es kommt zu Fehlern im Ausgabebild. Durch die Erhöhung der Terme in den beiden Approximationsfunktionen `log` und `exp` können diese Fehler behoben werden. Daher kann `gamma_V4.c` mit Einschränkungen verwendet werden. Das ist halt das perfekte Beispiel zwischen den Tradeoff Performance und Korrektheit.

Kleine Abweichungen können durch Approximation verursacht werden, insbesondere bei sehr feinen Farb- oder Helligkeitsunterschieden. Es ist auch wichtig zu betonen, dass aufgrund von Rundungsfehlern die Resultate von Berechnungen mit Fließkommazahlen eine begrenzte Korrektheit haben. Dadurch können sowohl Approximationsverluste als auch Pixelinformationen verloren gehen.

Wir haben den Mittelwert der Pixel berechnet, um dies zu beweisen und sicherzustellen, dass der Unterschied gering und unproblematisch ist.

Implementation	Mittelwert der Pixel
V0	160
V1	160
V2	159
V3	160
V4	163

Abbildung 7: Berechnung der Mittelwerte mit $\gamma = 0.5$ und standarte Graustufekoeffizienten

Es ist zu erwarten, dass solche Ungenauigkeiten bei Fließkommazahlenberechnungen gibt. Aber jeder Ansatz zeigt die Ausgabedatei klar. Folglich werden zusätzliche Tests zur Zusammenführung verschiedener Parameter beschrieben (mehr dazu in der Performanzanalyse).

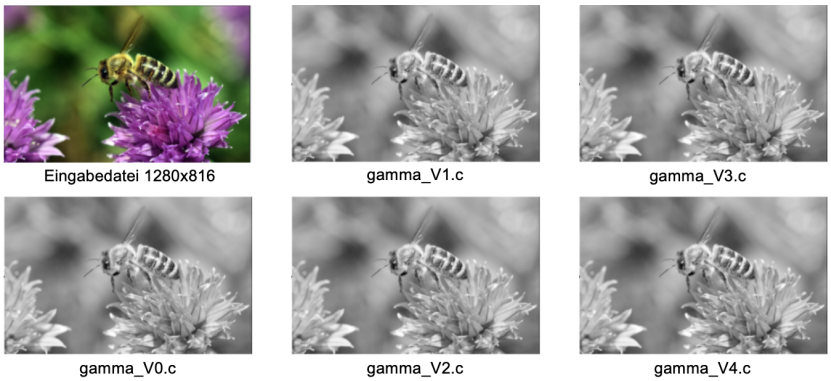


Abbildung 8: Die Vergleichung der Ausgabedateien generiert von verschiedenen Implementierungen mit $\gamma = 0.5$

4 Performanzanalyse

Um eine Beurteilung der unterschiedlichen Implementierung der Gamma-Korrektur zu treffen, wurde eine Performanzanalyse durchgeführt. Die Analyse umfasst drei Experimente, welche auf unterschiedlichen Bildgrößen sowie Iterationszahlen durchgeführt wurden, um eine umfassende Bewertung der Leistungsfähigkeit der implementierten Version zu ermöglichen.

Das dafür verwendete System mit Intel Core i7 mit 3,6 GHz, 24 GB DDR4 und Ubuntu 22.04.3 LTS, durchgeführt. Die Softwareumgebung bestand aus C17 und GCC 11.4.0.

1. *Experiment: ITU-R BT.601-7 (SDTV) mit 680x408 Bild*

Im ersten Experiment wurden die Implementierungen mit einem Standard-Definition-Bild (SDTV) gemäß ITU-R BT.601-7 getestet, und einer Bildgröße von 680x408 Pixel. Die Iterationszahlen variierten von 100 bis 10.000, um die Skalierbarkeit der Implementierungen zu bewerten. Wie in der ersten Grafik ersichtlich, weisen alle Implementierungen zeigen eine lineare Steigerung der Ausführungszeit mit zunehmender Anzahl von Iterationen. Implementierung v1 hat offensichtlich die schlechteste Performance, während v4 die niedrigste Ausführungszeit bei der höchsten Anzahl von Iterationen aufweist. Die vierte Version (v4) eine deutliche Leistungssteigerung gegenüber der Implementierungen v0, v2 und v1, mit einer Reduzierung der Ausführungszeit um 68% als v0 und v2 und um 51,46% als v3, bei 10.000 Iterationen.

2. *Experiment: ITU-R BT.601-7 (SDTV) mit 1920x1224 Bild*

Das zweite Experiment untersuchte die Performanz auf einem größeren Bild von 1920x1224 Pixeln, um die Effekte der Bildgröße auf die Verarbeitungszeit zu analysieren. Die Ergebnisse, dargestellt in der zweiten Grafik, verdeutlichen die Skalierbarkeitsprobleme der Basisimplementierung (v0), insbesondere im Vergleich zu den optimierten Versionen. Version 4 (v4) ist bei größeren Bildern noch überlegener zu den anderen Implementierungen. Im Vergleich ist v4 um das 4.3-fache schneller gegenüber der nächst effizienteren Variante v3, welche ebenfalls eine SIMD Implementation ist.

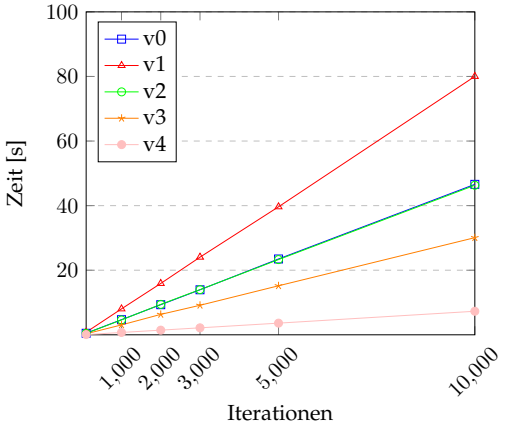
3. *Experiment: Durchschnittlicher Pixelwert vs. Gamma-Wert*

Zuletzt wurde der Einfluss verschiedener Gamma-Werte auf den durchschnittlichen Pixelwert untersucht, um die Auswirkungen der Gamma-Korrektur auf die Bildhelligkeit zu quantifizieren. Die Ergebnisse, veranschaulicht in der dritten Grafik, zeigen eine deutliche Abnahme des durchschnittlichen Pixelwerts mit zunehmendem Gamma, was die Effektivität der Gamma-Korrektur bei der Anpassung der Bildhelligkeit bestätigt.

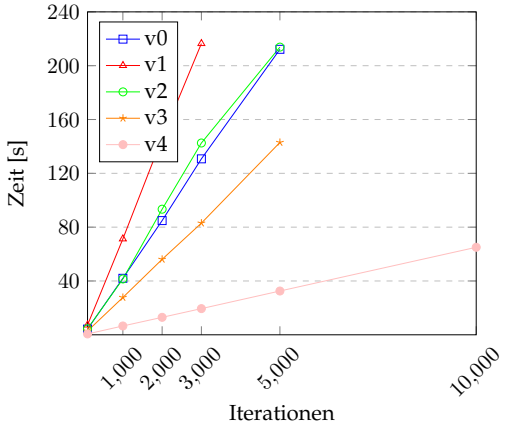
Zusammenfassung der Experimente

Durch die durchgeführten Experimente konnten wertvolle Erkenntnisse hinsichtlich Leistungsfähigkeit der verschiedenen Implementierungen der Gamma-Korrektur gezeigt werden. Während alle Versionen unter verschiedenen Bedingungen variierende Leistungsniveaus zeigten, hebt sich *v4* als besonders effizient hervor, insbesondere in Bezug auf die Ausführungszeit und Skalierbarkeit. Auch konnte im zweiten Experiment verdeutlicht werden, dass SIMD bei größeren Bildern deutlich effizienter ist als die Basisimplementationen v0 - v2.

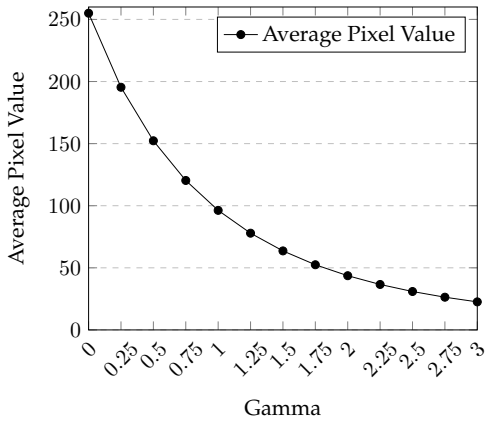
ITU-R BT.601-7 (SDTV) und 680 x 408 Bild



ITU-R BT.601-7 (SDTV) und 1920 x 1224 Bild



Average Pixel Value vs. Gamma 2beeM Bild und ITU-R BT.601-7 (SDTV)



5 Zusammenfassung und Ausblick

In dieser Arbeit wurde die Anwendung der Gamma-Korrektur auf Bilder im 24bpp PPM-Format in Graustufen umgewandelt und anschließend mittels Gamma-Korrektur in Helligkeit und Kontrast angepasst wurden. Die Analyse von vier Implementierungsansätzen in C hat gezeigt, dass trotz unterschiedlicher Methoden keine signifikanten Leistungsunterschiede zwischen den skalar basierten Versionen (v0 und v3) festgestellt wurden. Die Untersuchung hebt das Potenzial von SIMD-Parallelismus zur Leistungssteigerung hervor.

Für zukünftige Arbeiten könnte die Erkundung von SIMD-Optimierungen und adaptiven Gamma-Korrekturtechniken weitere Verbesserungen in Effizienz und Anpassungsfähigkeit an unterschiedliche Bildinhalte bieten. Dies könnte unter anderem durch eine Implementierung von Parallel Computing erreicht werden, vor allem in Kombination mit SIMD.

Literatur

[1] Franka Miriam Brückler. *Geschichte der Mathematik kompakt: das Wichtigste aus Analysis, Wahrscheinlichkeitstheorie, angewandter Mathematik, Topologie und Mengenlehre*. Springer Spektrum, Berlin, 2018. Seiten 89-90.

[2] Copyshop-tips. *Gammawert und Gradation*, 2024. `https://www.copyshop-tips.de/sam02.php`, Aufgerufen am 31.01.2024.

[3] Laurenz Göllmann. *Mathematik für Ingenieure: Verstehen - Rechnen - Anwenden, Band 1: Vorkurs, Analysis in einer Variablen, Lineare Algebra, Statistik*. Springer Vieweg Berlin, Heidelberg, 2017. Seite 182.

[4] INF-Schule. *P2 PGM Files*, Oktober 2023. `https://www.inf-schule.de/information/darstellunginformation/binaerdarstellungbilder/exkurs_pbmpgmppm`, Aufgerufen am 12.01.2024.

[5] Intel. *Intel Intrinsics Guide: Streaming SIMD Extensions*, Juli 2023. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, Aufgerufen am 10.01.2024.

[6] ITU INT. *Koeffizienten und Gamma-Werte für die Graustufenkonvertierung*, 2024. `https://www.itu.int/pub/R-REP-BT`, Aufgerufen am 02.02.2024.

[7] Herbert Kopp. *Bildverarbeitung interaktiv: eine Einführung mit multimedialem Lernsystem auf CD-ROM*. Teubner, Stuttgart, 1997. Seiten 14-19.

[8] Netpbm. *P5 PGM Files*, Oktober 2016. `https://netpbm.sourceforge.net/doc/pgm.html`, Aufgerufen am 23.01.2024.

[9] Netpbm. *P6 PPM Files*, August 2020. `https://netpbm.sourceforge.net/doc/ppm.html`, Aufgerufen am 28.01.2024.

[10] Uwe Schneider. *Imaging: Bildverarbeitung und Bildkommunikation*. Springer, Berlin, Heidelberg, 2013. Seites 45-46.

[11] Robert Sedgewick. *Einführung in die Programmierung mit Java*. München [u.a.], Pearson, 2011. Seite 348.

[12] Wilhelm Burger und Mark James Burge. *Digitale Bildverarbeitung: Eine algorithmische Einführung mit Java*. Springer Vieweg Berlin, Heidelberg, 2009. Seiten 78-80.

[13] Wilhelm Burger und Mark James Burge. *Digitale Bildverarbeitung: Eine algorithmische Einführung mit Java*. Springer Vieweg Berlin, Heidelberg, 2009. Seiten 81, 85.