

LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME  
**Praktikum Rechnerarchitektur**

Gruppe 141 – Abgabe zu Aufgabe A208  
Sommersemester 2020

Aaron Tacke

Philip Haitzer

Thomas Sedlmeyr

## 1 Einleitung

Selbst über 100 Jahre nach der Entwicklung der Farbfotografie erfreuen sich Schwarzweißbilder hoher Beliebtheit. Denn ohne störende Farben lenken sie den Blick des Betrachters gezielt auf das Wesentliche und heben den Kontrast und die graphische Struktur der aufgenommenen Szenerie exzellent hervor.

Um möglichst realistische Schwarzweißbilder zu erzeugen, unterzieht man diese einer Gammakorrektur, bei der die Helligkeit der Pixel beeinflusst wird.



Möchte man mit heutigen Kameras digitale Schwarzweißaufnahmen machen, gelingt dies meist nur durch Umwandlung eines digitalen Farbbildes. Um von der höheren Auflösung moderner Kameras zu profitieren, benötigen wir ein Programm, welches eine große Anzahl an Pixeln mit sinnvoll spezifizierten Gewichtungen der Farben performant in Graustufen umwandelt und eine effiziente Gammakorrektur anwendet.

Als Eingabe soll jedes ASCII-Codierte Bild im PPM-Format und eine vom Benutzer gewählte positive Fließkommazahl für die Gammakorrektur fungieren. Dabei soll für jeden  $\gamma$ -Wert der mathematisch korrekt berechnete Pixelwert als Ganzzahl bestimmt und das umgewandelte Bild im PPM-Format abgespeichert werden.

## 2 Lösungsansatz

### 2.1 PPM-Format

Die erste Zeile eines PPM-Bildes kennzeichnet das PPM-Format. Da wir mit Bildern, welche in ASCII gespeichert sind, arbeiten, erwarten wir hier "P3". Ein optionaler Kommentar ist in der zweiten Zeile zu finden. In der dritten Zeile werden Breite und Höhe des Bildes, durch ein Leerzeichen getrennt, angegeben. Die vierte Zeile enthält den maximalen Wert eines Farbanteils. Jede folgende Zahl liegt also zwischen Null und dem hier spezifizierten Wert, welcher bei uns standardmäßig 255 sein muss. Danach folgen für jedes Pixel drei Zeilen, welche die Rot-, Grün- und Blau-Werte für das jeweilige Pixel beinhalten.

## 2.2 Berechnung des Graustufenfilters

Zu Beginn wird jedes Pixel in Graustufen umgewandelt. Dabei berechnen sich die neuen Werte eines Pixels mit folgender Formel.

$$p_{alt} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \quad D = \frac{a \cdot R + b \cdot G + c \cdot B}{a + b + c} \quad p_{neu} = \begin{pmatrix} D \\ D \\ D \end{pmatrix} \quad (1)$$

Wir entschieden uns dafür, dass für die Gewichtung

$$a + b + c = 256 \quad (2)$$

gelten soll. Dies birgt den Vorteil, dass für die weitere Berechnung die Division durch einen Shiftbefehl effizient berechnet werden kann.

## 2.3 Optimierung des Graustufenfilters mit SIMD

Die Grundidee für die optimierte Berechnung mit SIMD ist, mit Hilfe von drei XMM-Registern den Graustufenfilter auf fünf Pixel gleichzeitig anzuwenden. Hierfür werden jeweils der Rot-, Grün- und Blauanteil der Pixel in einem XMM-Register gespeichert, so dass sich vor jeder belegten Speicherzelle noch mindestens ein freies Byte befindet. Dann werden die XMM-Register jeweils mit zugehörigem Faktor ( $a$ ,  $b$  oder  $c$ ) multipliziert. Im nächsten Schritt werden die XMM-Register aufaddiert und mit einer entsprechenden Bytemaske die niederwertigen Bytes der Ergebnisse extrahiert. Eine explizite Division ist dabei nicht notwendig, da  $a$ ,  $b$  und  $c$  in Summe 256 ergeben und die Maskierung der höherwertigen Bytes einer Division mit 256 entspricht.

1. Einlesen aus dem Speicher																
Stelle im Xmmer-Register:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Byte Werte im Speicher	12	144	222	44	66	124	185	129	244	255	9	37	131	181	179	0
2. Nach Farbwerten in Xmmer Register aufteilen																
erstes Xmmer-Register			12			44		185			255			131		
zweites Xmmer-Register			144			66		129			9			181		
drittes Xmmer-Register			222			124		244			37			179		
Xmm Register mit den Koeffizienten:																
Xmm-Register für a				77				77			77			77		
Xmm-Register für b				151				151			151			151		
Xmm-Register für c				28				28			28			28		
3. Multiplikation mit den Koeffizienten:																
Multiplikation mit a		3	156		13	60	55	165			76	179		39	103	
Multiplikation mit b		84	240		38	238	76	23			5	79		106	195	
Multiplikation mit c		24	72		13	144	26	176			4	12		19	148	
4. Addition der Xmmer-Register																
Ergebnis der Addition	112	212		65	186	158	108			86	14		165	190		
5. Extrahieren und Kopieren der Ergebnisse in Xmmer-Register																
Ergebnis Xmmer-Register	112	112	112	65	65	65	158	158	158	86	86	86	165	165	165	0

Abbildung 1: SIMD Optimierung

Dabei stellten wir fest, dass die mit SIMD optimierte Ausführung weniger effizient verlief als die Vergleichsimplementierung in C. Der Grund hierfür war, dass die Pixel nicht in 16 Byte Blöcken aligned waren, was die Speicherzugriffe sehr teuer gemacht hat. Außerdem benötigten wir deshalb (beispielsweise zum Speichern des überflüssigen 16. Wertes) zu viele Shift-Befehle, welche ca. 35% der gesamten Berechnungsdauer in Anspruch nahmen. Aus diesem Grund sorgten wir für ein Alignment des Speichers, indem wir nach jedem 15. Byte ein 0 Byte einfügten und die Assemblerimplementierung dementsprechend abänderten. Dabei gelang es uns, mit der SIMD optimierten Version die Laufzeit (auf einem 2000x2000-Pixel Bild) um 66% gegenüber des in C implementierten Graustufenfilters zu verbessern.

## 2.4 Untersuchung der Gammakorrektur

Unsere oberste Priorität war es, für alle positiven Werte für  $\gamma$ , die mit dem Datentyp float darstellbar sind, die mathematisch korrekte Gammakorrektur durchführen zu können.

$$p_{neu} = \left( \frac{p_{alt}}{255} \right)^\gamma \cdot 255 \quad (3)$$

Diese Funktion nennen wir im Folgenden Gammafunktion.

Da es sich bei der Potenz mit  $\gamma$  um eine Gleitkommazahl im Exponenten handelt, ist der naive Lösungsansatz,  $\gamma$  mit einem Bruch zu approximieren und die Potenz durch eine ganzzahlige Potenz und das Ziehen einer n-ten Wurzel zu ersetzen. Vor allem das Berechnen der n-ten Wurzel stellte sich allerdings als Problem heraus, weil hierfür eine Vielzahl an mathematischen Operationen ausgeführt werden müssen. Deshalb nutzen wir den eingeschränkten Definitions- und Wertebereich  $\{0, \dots, 255\}$  und den Fakt, dass die Funktion monoton, steigt für weitere Optimierungen.

## 2.5 Berechnung aller Gammafunktionen

Die Erkenntnisse über den geringen Definitions- und Wertebereich unterschiedlicher Gammafunktionen und über das Monotonieverhalten nutzten wir, um mit Hilfe eines C-Programms Intervalle für  $\gamma$  zu berechnen, in denen die entsprechenden Gammafunktionen für jeden Eingabewert von 0 – 255 die gleiche Ausgabe liefern. Hierfür verwendeten wir Bisektion, indem mit zwei aufeinander folgenden float Werten gestartet und die Größe des Intervalls zunächst so lange nach rechts verdoppelt wird, bis sich die zugehörigen Funktionen an den Intervallgrenzen unterscheiden. Von diesem neuen Intervall wird die Mitte und die Funktion an der mittleren Stelle berechnet. Ist diese gleich der Funktion der rechten Intervallgrenze, wird der rechte Rand des Intervalls, andernfalls der Linke, auf die Mitte gesetzt. Dies wird solange fortgeführt, bis die Stelle gefunden ist, an der sich für zwei aufeinander folgende Float-Werte die Funktionen unterscheiden. Nach Ausführung des Programms speicherten wir die 64484 resultierenden Werte in einer Headerdatei ab, um bei späteren Tests einfach darauf zugreifen zu können.

## 2.6 Effiziente Speicherung von vorberechneten Gammafunktionen

Da eine Compile-Zeit von ca. 18 Sekunden zu lang und die Größe der ausführbaren Dateien mit ungefähr 17MB zu groß ist, versuchten wir, die vorberechneten Funktionen effizienter zu speichern.

Da die Gammafunktion monoton steigend ist und stets  $f(0) = 0$  gilt, reicht es aus abzuspeichern, um wie viel der nächste y-Wert größer ist als der Vorherige. Dies bedeutet, dass sich jede Gammafunktion unär repräsentieren lässt, indem die Gesamtanzahl der Nullen vor der n-ten Eins den n-ten Funktionswert liefert. Abbildung 2 zeigt ein Beispiel für die ersten sechs Werte einer Gammafunktion, welche als Binärzahl 001100101001001<sub>2</sub> oder dezimal mit 6473<sub>10</sub> beschrieben werden kann.

Da die Gammafunktionen jeweils nur 256 Funktionswerten aus dem Wertebereich zwischen 0 und 255 besitzen, benötigt man 256 Einsen und 256 Nullen, um eine Gammafunktion abzuspeichern. Wir speicherten die codierte Funktion mit acht Dezimalzahlen des Datentyps *long long* ab. Somit gelang es uns, den Speicherverbrauch von 70MB auf ein Siebtel zu reduzieren und die Compile-Zeit von zuvor 18 Sekunden auf zwei Sekunden zu senken.

x	1	2	3	4	5	6
f(x)	2	2	4	5	7	9
$\Delta f(x)$	2	0	2	1	2	2
Unär	00		00	0	00	00
Binär	00 1	1 00 1	0 1 00 1	00		

Abbildung 2: Beispielcodierung

## 2.7 Decodierung der komprimierten Funktionen

Für das Decodieren einer so komprimierten Funktion werden für den gegebenen Gammawert die acht zugehörigen Werte aus dem Speicher der Reihe nach geladen und bitweise nach links geshiftet. Nach jedem Shiftbefehl wird mit Hilfe des Carry-Flags überprüft, ob eine Null oder Eins aus dem Register geshiftet wurde und der jeweilige Zähler wird erhöht. Handelt es sich um eine Eins, bedeutet dies, dass der Funktionswert für  $x = \text{Zähler}$  der Einsen, der Anzahl der bisher gezählten Nullen entspricht. Dieser Wert wird abgespeichert und der Algorithmus solange wiederholt, bis die gesamten 512 Bit durchgeshiftet wurden. Mit diesem Verfahren werden für einen Gammawert die Funktionswerte aller möglichen  $x \in 0, \dots, 255$  vorberechnet.

Effizientes Decodieren während der Gammakorrektur könnte das Nutzen des Hauptspeichers überflüssig machen und potenziell schneller sein als unsere Lösung, weshalb wir dies im Ausblick am Ende der Arbeit kurz aufgreifen.

## 2.8 Berechnung der Wurzel

Das Berechnen einer Potenz mit nicht ganzzahligem Exponent ist schwierig, weshalb wir  $\gamma$  als Bruch aus den natürlichen Zahlen  $z$  und  $n$  schreiben.

Ein gängiges Verfahren um nun  $\sqrt[n]{\left(\frac{p_{alt}}{255}\right)^z}$  zu berechnen, ist es diese als Nullstelle des Polynoms

$$f(x) = x^n - \left(\frac{p_{alt}}{255}\right)^z \quad (4)$$

auszudrücken und diese mit verschiedenen Möglichkeiten zur Nullstellenapproximation, wie zum Beispiel dem Newtonverfahren, anzunähern. Weil aber selbst für das schnell konvergierende Newtonverfahren mehrmals verschiedene Funktionswerte von  $f(x)$  berechnet werden müssen, ist diese Methode nicht effizient genug.

Für hohe Effizienz sollten  $n$  und  $z$  entsprechend klein sein, bei kleinem  $n$  und  $z$  lässt sich  $\gamma$  allerdings nur ungenau beschreiben, weshalb wir nicht immer mathematisch korrekt berechnete Ergebnisse erhalten.

## 2.9 Binäre Suche zur Berechnung der Gammafunktion

Da  $p_{neu}$  ganzzahlig sein soll, und das Berechnen einer Wurzel relativ kompliziert ist, haben wir die Gammafunktion umgeformt zu:

$$p_{neu} = \max\{x \in \mathbb{N} : \left(\frac{x}{255}\right)^n \leq \left(\frac{p_{alt}}{255}\right)^z\} \quad (5)$$

Zur Berechnung der ganzzahligen Potenzen implementierten wir zudem eine Methode, die es ermöglicht mit Hilfe von binärer Exponentiation die Anzahl der für die Potenzierung erforderlichen Multiplikationen auf ein Minimum zu reduzieren.

Für hohe Effizienz sollten  $n$  und  $z$  entsprechend klein sein, bei kleinem  $n$  und  $z$  lässt sich  $\gamma$  allerdings nur ungenau beschreiben, weshalb wir nicht immer mathematisch korrekt berechnete Ergebnisse erhalten. Da dies unserer Vorgabe der Exaktheit unseres Programms widersprach, versuchten wir unser Verfahren weiter zu verbessern, indem wir auf den Exponenten rechneten.

## 2.10 Berechnung auf den Exponenten

Da wir wissen, dass  $p_{alt} = 0 \implies p_{neu} = 0$  stets gilt, können wir die restliche Gammafunktion schreiben als:

$$p_{neu} = \max\{x \in \mathbb{N} : l(x) \leq l(p_{alt}) \cdot \gamma\} \quad \text{mit} \quad l(x) = \ln\left(\frac{x}{255}\right) \quad (6)$$

Speichert man die Werte von  $l(x)$  im Speicher, muss man nur noch den passenden Wert laden, mit  $\gamma$  multiplizieren und mittels binärer Suche den zugehörigen x-Wert des nächstkleineren Funktionswertes von  $l(x)$  bestimmen, welcher das gesuchte  $p_{neu}$  ist.

Da dieses Verfahren (auf einem 2000x2000-Pixel Bild) 58% langsamer als die Vergleichsimplementierung war, mussten wir strukturelle Änderungen an dem Programm vornehmen. Das Berechnen der Funktion am Anfang und das Speichern der 256 Funktionswerte im Hauptspeicher sollte dank des Caches eindeutig bessere Ergebnisse liefern.

Um die gesamte Funktion noch effizienter zu berechnen, nutzten wir erneut das Monotonieverhalten und bestimmten den deshalb stets positiven Wert  $d$ , welcher zwischen zwei Funktionswerten liegt.

Sei  $g(x)$  hier die Gammafunktion, also  $p_{neu} = g(x) \iff x = p_{alt}$

$$d = |\{n \in \mathbb{N} : n > g(x - 1) \wedge l(n) \leq l(x) \cdot \gamma\}| \quad (7)$$

Als Algorithmus lässt sich diese Beobachtung folgendermaßen umsetzen: Man beobachtet die beiden Sequenzen, die sich aus  $l(x)$  und  $l(x) * \gamma$  im Definitionsbereich  $\{1, \dots, 255\}$  ergeben, führt für beide einen Zeiger  $i_l = 0$  und  $i_\gamma = 0$  sowie einen Zähler  $c = 0$  ein. Ist  $l(i_l) \leq l(i_\gamma) * \gamma$ , so erhöht man  $i_l$  und  $c$ . Andernfalls speichert man  $c$  als den Funktionswert von  $i_\gamma + 1$  ab.

Mit diesem Verfahren kann man die gesamte Funktion durch 510 Lesezugriffe und Vergleiche sowie 255 Schreibzugriffe und Multiplikationen bestimmen.

## 2.11 Vergleichsimplementierung

Die Vergleichsimplementierung nutzt die Funktionen der C-Standardbibliothek, um die Pixel nacheinander mit den angegebenen mathematischen Funktionen zu berechnen. Trotz Kompilierung mit -O3 haben wir die Laufzeit dieser Implementierung so extrem unterboten, dass wir eine zweite Vergleichsimplementierung erstellt haben, welche auf die Funktionswerte aller möglicher zuvor berechneter Funktionen im Speicher zugreifen kann, und statt des langsamen Potenzierens lediglich die passende Funktion binär sucht. Wir konnten mit dieser Methode kein falsches Ergebnis feststellen, allerdings ist die vollständige Korrektheit mathematisch schwer zu beweisen. Zudem führten die vorberechneten Funktionen zu langsamem Compile-Zeiten und großen ausführbaren Dateien, weshalb sie lediglich zum Benchmarking, nicht jedoch in der finalen Version des Programmes eingesetzt werden.

## 2.12 Intuitve Bestimmung des Graustufenfilters

Die einfachste Form des Graustufenfilters, bei der nur der Durchschnitt der einzelnen Farbwerte mit Hilfe von (1) berechnet wird, nutzt  $a = b = c = 1$  als Parameter. Bei dieser einfachen Umwandlung wird jedoch fälschlicherweise angenommen, dass alle Farben vom menschlichen Auge gleich hell wahrgenommen werden. In der Realität unterscheidet sich die Farbwahrnehmung jedoch von dieser Vorstellung, da beispielsweise die Farbe grün heller wahrgenommen wird als blau. Um diese Informationen bei der Umwandlung in Graustufen nicht zu verlieren, müssen die Parameter  $a$ ,  $b$  und  $c$  so angepasst werden, dass die Helligkeit in dem Graustufenbild die Farben auf dem ursprünglichen Bild möglichst genau repräsentiert.

Im Vergleich der beiden Bilder ist zu erkennen, dass die Farben mit einer sinnvollen Gewichtung des Graustufenfilters viel realistischer in Graustufen umgewandelt werden. So sind die als heller wahrgenommenen Farben gelb und grün auch in dem Graustufenbild deutlich heller dargestellt und lassen sich besser von den anders farbigen Objekten



Abbildung 3: Ohne Gewichtung



Abbildung 4: Mit Gewichtung

unterscheiden. Im ersten Bild hingegen lässt sich der Farbunterschied zwischen den einzelnen Früchten weniger erkennen. Es ist somit deutlich, dass die Wahl der Parameter eine entscheidende Rolle für die realistische Umwandlung eines farbigen Bildes in Graustufen spielt. Wie ein möglichst idealer Wert, der nicht nur auf Intuition basiert, sondern an die menschliche Farbwahrnehmung angepasst ist, bestimmt werden kann, wird im Folgenden erläutert.

### 2.13 Anpassung des Graustufenfilters an das menschliche Auge

Um den Einfall von Licht wahrzunehmen, befinden sich in der Retina des Auges sogenannte Stab- und Zapfenzellen, beides eine Art modifizierte Nervenzelle, welche Photonen (Licht) absorbieren können und ein entsprechendes elektrisches Signal Richtung Gehirn weiterleiten. Die Farbwahrnehmung dieser Zellen lässt sich durch eine sogenannte Absorptionskurve beschreiben, die angibt, welche Wellenlängen (Farben) am stärksten absorbiert, in elektrische Signale umgewandelt und somit wahrgenommen werden. Zapfenzellen erzeugen bei guter Beleuchtung scharfe Bilder, wohingegen Stabzellen für das Sehen bei geringem Licht und in peripheren Blickwinkeln verantwortlich sind. Somit müssen wir uns für die Wahl der Gewichtung von Farben für den Graustufenfilter lediglich an den Absorptionskurven von Zapfenzellen orientieren.

Verrechnet man die Anzahl der unterschiedlichen Zapfenzellen mit dessen Absorptionskurven, erhält man für unterschiedliche Wellenlängen (Farben) des Lichts die Intensitäten, mit der diese elektrischen Signale im Gehirn auslösen, welche genau unseren Faktoren  $a = 0,3$ ,  $b = 0.59$  und  $c = 0.11$  entsprechen. [2]

### 2.14 Resultate bei unterschiedlichen Gammawerten

Die Wahrnehmung von Helligkeit des menschlichen Auges folgt, wie viele andere neurologische Prozesse, dem Weber-Fechner-Gesetz [3] und ist somit logarithmisch. In dunklen Bereichen steigt die Helligitätssensitivität stärker an [4]. Durch die richtige Wahl von  $\gamma$  können die Grauwerte eines Bildes gezielt so verändert werden, dass sie für das menschliche Auge realistischer wahrgenommen werden.

An den Abbildungen 5 bis 8 (siehe unten) ist zu erkennen, dass für  $\gamma > 1$  die Bilder dunkler werden und sich in hellen Bereichen der Kontrast erhöht. Dieser Effekt lässt sich am besten verstehen, wenn man die zugehörigen Graphen der unterschiedlichen Gammafunktionen betrachtet. So verlaufen alle Graphen mit  $\gamma > 1$  unter dem Graphen von  $f(x) = x$ . Dies bedeutet, der Grauwert wird kleiner und das Bild wird dunkler dargestellt. Alle Funktionen mit  $\gamma' = \frac{1}{\gamma}$  stellen die Umkehrfunktion zur Funktion mit  $\gamma$  dar und sind deshalb symmetrisch zur Winkelhalbierenden im ersten Quadranten. Daher verlaufen diese Funktionen oberhalb des Graphen von  $f(x)$  und liefern größere Grauwerte und das bearbeitete Bild wird heller dargestellt. Wenn man für  $\gamma$  den Wert 1,25 wählt, erhält man ein Bild, dessen Kontrast bei diesem Beispiel am realistischsten wirkt (siehe Abbildung 6).

### 3 Korrektheit/Genauigkeit

#### 3.1 Beispielanwendungen

Abbildung 5 wurde von unserem Programm mit verschiedenen  $\gamma$ -Werten in ein Schwarz-weißbild umgewandelt:



Abbildung 5: Originalbild [1]



Abbildung 6: Korrektur mit  $\gamma = 1,25$



Abbildung 7: Korrektur mit  $\gamma = 0,5$



Abbildung 8: Korrektur mit  $\gamma = 2,0$

### 3.2 Berechnung aller möglichen Gammafunktionen

Unsere oberste Priorität war es, für jedes positive  $\gamma$  den korrekten Wert zu berechnen und somit die Funktionalität nicht einschränken zu müssen. Auch kennen wir alle Gammafunktionen, die mit dem Datentyp float darstellbar sind und können daher für unser Programm garantieren, dass jeder Wert für  $\gamma$  korrekt berechnet wird.

### 3.3 Test-Funktionalitäten

Das Programm bietet die Möglichkeit, die optimierte Implementierung zu testen, indem entweder ein vorgegebenes oder selbst gewähltes Test-Bild umgewandelt wird, danach das Ergebnis mit der Umwandlung der Vergleichsimplementierung Pixel für Pixel verglichen und das Ergebnis auf der Konsole ausgegeben wird. Dafür kann der  $\gamma$ -Wert entweder frei gewählt werden oder der Test wird für 10 Werte zwischen 0,2 und 2 durchgeführt.

Wir testeten unsere Implementierung mit einem Bild mit allen möglichen  $2^{24}$  Pixeln und unterschiedlichsten  $\gamma$ -Werten.

## 4 Performanceanalyse

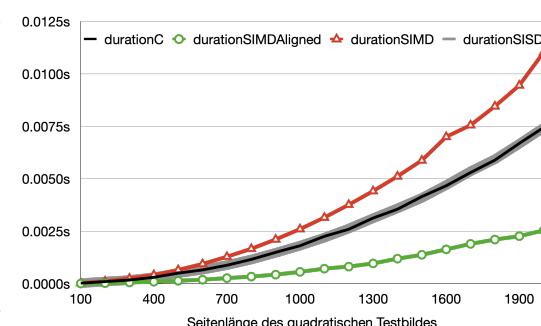
### 4.1 Systemdokumentation

Getestet wurde auf einem System mit einem Intel Core i7-4650U Prozessor, 1.70 GHz, 8GB Arbeitsspeicher (1600 MHz DDR3), macOS Catalina 10.15.4, 64 Bit. Kompiliert wurde mit GCC 4.2.1 mit der Option -O3.

Jeder ermittelte Wert wurde mit 30 Iterationen und 10 unterschiedlichen  $\gamma$ -Werten (soweit  $\gamma$  nicht Teil des Diagramms war) berechnet. Die Wahl der Bildgrößen (200x200 bis 2000x2000 Pixel) lässt sich dadurch begründen, dass Bilder mit ähnlichen Anzahlen an Pixeln besonders häufig vorkommen.

### 4.2 Analyse des Graustufenfilters

Bei dem Vergleich der Geschwindigkeit der C-Implementierung mit einer einfachen SISD-Implementierung in Assembler fällt auf, dass diese annähernd identisch sind. Die nahe liegende Vermutung, dass die Disassembly der mit -O3 kompilierten C-Methode fast unserer SISD-Implementierung entspricht, bestätigt sich beim Vergleich der Befehle und Geschwindigkeiten, die das Programm "Perf" ausgibt. Der erste Versuch einer SIMD-Implementierung, welche die Graustufen-Werte von fünf Pixeln gleichzeitig berechnet, war trotz geringer Anzahl an Multiplika-



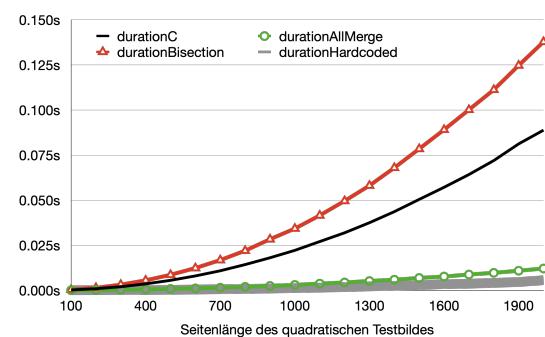
tionen langsamer als die Vergleichsimplementierung. Dies lag an der Häufigkeit der 128-Bit Shift-Operationen, deren Kosten wir unterschätzt hatten, und vor allem dem nicht-ausgerichteten Zugriff auf den Speicher. Durch Weglassen nicht notwendiger Shift-Befehle und 16-Byte Ausrichtung aller fünf aufeinander folgenden Pixel im Speicher, erreichten wir unabhängig von der Größe des Bildes eine deutlich bessere Laufzeit als mit vorherigen Ansätzen und der Vergleichsimplementierung.

### 4.3 Analyse der Gammakorrektur

Die Berechnung der Gammafunktion für jedes Pixel einzeln ist trotz Bisektion mit logarithmischer Laufzeit langsamer als die Vergleichsimplementierung, weshalb es sinnvoll war, die 256 möglichen Funktionswerte vorher zu berechnen.

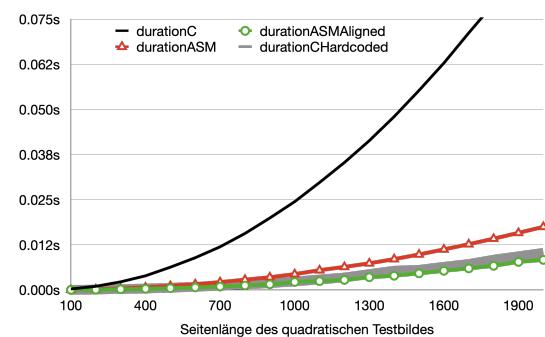
Dies war leider stets langsamer als die binäre Suche, welche unter allen 64484 möglichen hardcodeten Funktionen die richtige auswählt. Da diese Menge an Funktionen allerdings Nachteile wie sehr lange Compile-Zeit (ca. 18 Sekunden) und große Binaries (ca.

17MB) mit sich zog, wäre sie lediglich in abgewandelter Form in Frage gekommen. Auf 512-Bit codierte Funktionen hätten weniger Speicher verbraucht und könnten einmalig in Register geladen werden, weshalb es nicht mehr auf den Speicherzugriff als Flaschenhals ankäme, sondern lediglich auf die Effizienz der Decodierung (siehe Ausblick).



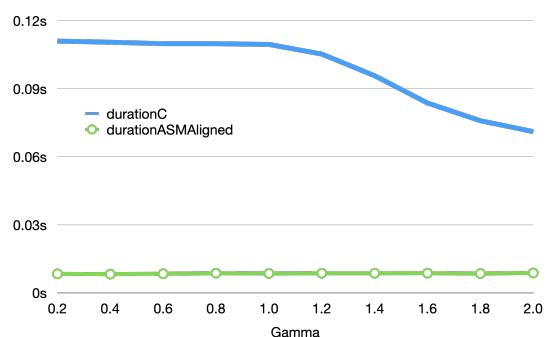
### 4.4 Analyse des fertigen Programms

An dieser Stelle ist der Vergleich mit der ursprünglichen Vergleichsimplementierung in C uninteressant, weshalb wir versuchten, die Laufzeit bei der Benutzung von hardcodeten Funktionen zu unterbieten. Da bei Ausführung des Graustufenfilters und der Gammafunktion zwischendurch kein passend ausgerichtetes Ergebnis benötigt wird, sondern die Ausgabe des Graustufenfilters unbearbeitet an die entsprechend angepasste Gammakorrektur gegeben werden kann, konnten weitere Shift-Operationen eingespart werden. Letztlich genügt die Verbesserung durch den SIMD-Graustufenfilter, um in Verbindung mit ausgerichteten Speicherzugriffen die Laufzeit des C-Programmes, welche hardcodete Funktionen für die Gammakorrektur verwendet, zu unterbieten.



## 4.5 Performance in Abhängigkeit von Gamma

Während die Vergleichsimplementierung durch eine längere Dauer der Berechnung der Potenz gerade bei häufig benutzten Werten von  $\gamma$  zwischen 0 und 1 länger dauert, unterbietet unsere Implementierung die Laufzeit mit  $\gamma$ -unabhängiger Geschwindigkeit. Dies ist dadurch begründet, dass die optimierte Berechnung der Gammafunktion stets die gleiche Anzahl an Befehlen ausführt und lediglich triviale mathematische Funktionen (Multiplikation und Addition) verwendet werden, da wir auf den Exponenten rechnen.



## 5 Zusammenfassung und Ausblick

Unser Programm ermöglicht das effiziente Anwenden eines Graustufen-Filters mit Gammakorrektur auf beliebige Farbbilder im ASCII-Codierten PPM-Format.

Das Ziel, dass unsere Optimierung für alle möglichen positiven  $\gamma$ -Werte die mathematisch korrekt berechneten Pixel liefert, gab uns zwar nicht die Möglichkeit Rundungen oder Abschätzungen zum Steigern der Performance zu verwenden, zwang uns dafür allerdings, die unterschiedlichen Probleme mehrfach mit einer gewissen Kreativität neu anzugehen.

Schließlich gelang es uns, unter Verwendung von SIMD-Befehlen, passendem Alignment, geschickt gewählten Shiftoperationen und auf Assembler abgestimmte mathematische Formeln einen Algorithmus zu implementieren, der (bei einem 2000x2000-Pixel Bild) zehn mal effizienter arbeitet als eine naive Vergleichsimplementierung und sogar schneller vorgeht als ein C-Programm, welches lediglich vorberechnete Funktionen aus dem Speicher liest.

Einen Performancevorteil gegenüber unserer Implementierung hätte lediglich ein Algorithmus, welcher eine Gammafunktion codiert in Registern ablegt, da der Speicherzugriff der einzige Teil unserer Implementierung ist, welcher noch merklich verbessert werden kann.

Damit der Vorteil des Registers gegenüber dem Cache die Gesamtaufzeit positiv beeinflusst, benötigt man eine Suche nach der Position des  $x$ -ten gesetzten Bits eines Registers. Diese Funktionalität lässt sich zwar durch binäre Suche mit Shifts umsetzen, aber schneller als unser Programm ist sie nur sobald ein Assembler-Befehl unterstützt würde, welcher diese Aufgabe effizient löst.

## Literatur

- [1] Bild: Obstkorb.  
<https://www.truebenecker.de/wp-content/uploads/2019/02/Obstkorb-f%C3%BCrs-B%C3%BCro-premium-medium.jpg>, visited 2020-07-10.
- [2] Sawakinome. Was ist der Unterschied zwischen Stabzellen und Kegelzellen.  
<https://www.sawakinome.com/articles/science/what-is-the-difference-between-rod-cells-and-cone-cells.html>, visited 2020-07-03.
- [3] Spektrum. Weber-Fechner-Gesetz.  
<https://www.spektrum.de/lexikon/biologie/weber-fechnersches-gesetz/70397>, visited 2020-07-20.
- [4] Stephan Thesmann. Einführung in das Design multimedialer Webanwendungen.  
[https://books.google.de/books?id=NHts2eHUlTMC&pg=PA13&lpg=PA13&dq=gammakorrektur+schwarzweiss+menschliches+sehen&source=bl&ots=J1siHt4w90&sig=ACfU3U16\\_ktR2B5nWvhMKcsYtveAIC\\_QqA&hl=de&sa=X&ved=2ahUKEwjA7\\_uF7bbqAhXV3oUKHY60DfoQ6AEwDXoECAsQAQ#v=onepage&q=gammakorrektur%20schwarzweiss%20%20menschliches%20sehen&f=false](https://books.google.de/books?id=NHts2eHUlTMC&pg=PA13&lpg=PA13&dq=gammakorrektur+schwarzweiss+menschliches+sehen&source=bl&ots=J1siHt4w90&sig=ACfU3U16_ktR2B5nWvhMKcsYtveAIC_QqA&hl=de&sa=X&ved=2ahUKEwjA7_uF7bbqAhXV3oUKHY60DfoQ6AEwDXoECAsQAQ#v=onepage&q=gammakorrektur%20schwarzweiss%20%20menschliches%20sehen&f=false), visited 2020-07-05.