Image Denoising

Aslem Slama

1 Einleitung

Heutzutage ist es immer eine gute Idee, die Speichergröße von Bildern zu reduzieren, um Speicherplatz zu sparen. In unserem Projekt konzentrierten wir uns auf die Bildkompression, ein Teilgebiet der Bildverarbeitung. Wir fokussierten uns auf die Lauflängencodierung (Run-Length-Encoding) Methode zur Verringerung der Größe von BMP (Bitmap)-Bildern. Wir entwickelten zwei Kompressionsmethoden, die beide in C implementiert sind. Eine der Methoden ist optimiert und verbessert die Kompressionsleistung ohne Nebenwirkungen.

Darüber hinaus haben wir ein Rahmenprogramm entwickelt, das zum Lesen der Eingabe und zum Schreiben der Ausgabe dient. Unser Rahmenprogramm enthält auch viele andere Flags mit nützlichen Funktionen.

1.1 BMP Format

BMP, kurz für Bitmap, ist ein unkomprimiertes Dateiformat für Bilder, das vorwiegend im Windows-Betriebssystem verwendet wird und ein zweidimensionales Rastergrafikformat darstellt. Dieses Format, entwickelt von Microsoft, speichert Farbinformationen in einem geräteunabhängigen Format.

Ein BMP besteht aus einem Bitmap File Header, einer Bitmap Info-Struktur, die sich aus einem Bitmap Info Header und einer Farbtabelle zusammensetzt, sowie den eigentlichen Bilddaten. Der File Header umfasst 14 Bytes und enthält unter anderem Informationen zum Dateityp und zur Dateigröße.

Der Info-Header beinhaltet detaillierte Informationen über die Eigenschaften der BMP-Datei. Die genauen Inhalte dieser Informationen können Tabelle 1 entnommen werden .

Offset	Größe	Hex Wert	Wert	Beschreibung
0h	2 Bytes	42 2D	"BM"	ID field (42h, 4Dh)
2h	4 Bytes	46 00 00 00	70 Bytes (54+16)	Größe der BMP-Datei
6h	2 Bytes	00 00	Unbenutzt	0
8h	2 Bytes	00 00	Unbenutzt	0
Ah	4 Bytes	36 00 00 00	54 Bytes (14+40)	Offset, in dem das Pixel-Array zu finden ist

Table 1: BMP Header

Der BMP-Info-Header enthält wichtige Informationen über eine BMP-Datei, wie Bildgröße, Farbtiefe und Komprimierung. Er ermöglicht die korrekte Interpretation der Bilddatei. Programme können auf die Header-Daten zugreifen, um die Eigenschaften der BMP-Datei zu analysieren. Die genauen Informationen können Tabelle 2 entnommen werden.

Offset (hex)	Offset (dez)	Größe (Bytes)	Windows BITMAPINFOHEADER
0E	14	4	Größe des Headers (40 Bytes)
12	18	4	Breite des Bildes (signed int)
16	22	4	Höhe des Bildes (signed int)
1A	26	2	Anzahl der Farbebenen (=1)
1C	28	2	Anzahl der Bits pro Pixel
1E	30	4	Verwendetes Komprimierungsverfahren
22	34	4	Größe des Bildes (in Bytes)
26	38	4	Horizontale Auflösung des Bildes (Pixel pro Meter)
2A	42	4	Vertikale Auflösung des Bildes (Pixel pro Meter)
2E	46	4	Anzahl der Farben in der Farbpalette
32	50	4	Anzahl der wichtigsten verwendeten Farben

Table 2: BMP Info Header

1.2 RLE oder Lauflängencodierung

Wie allgemein bekannt ist, besteht ein Bild aus vielen kleinen Pixeln. Jeder Pixel hat bestimmte Eigenschaften, die Farbe und Helligkeit definieren. RLE .*Run Length Encoding* ist eine einfache und weit verbreitete Komprimierungsmethode, die in der Datenkompression eingesetzt wird. Sie basiert auf dem Prinzip der wiederholten Zeichenfolgen in einer Datenquelle.

Bei RLE wird eine Zeichenkette analysiert, und anstatt jede einzelne Instanz desselben Zeichens oder einer Zeichenfolge zu speichern, wird die Information über die Anzahl der Wiederholungen und das wiederholte Zeichen selbst gespeichert. Dadurch wird der Speicherbedarf reduziert. Beispielsweise wird statt "AABBBBBBCCC" nur "2A6B3C" gespeichert. Hierbei wird die Sequenz von zwei 'A'-Zeichen, sechs 'B'-Zeichen und drei 'C'-Zeichen durch ihre Wiederholungszahl und das entsprechende Zeichen kompakt dargestellt.

Im unteren bekommt man also:

RLE=((1,S),(1,R),(2,G),(2,B),(1,W),(2,S),(2,W),(1,B),(2,R),(1,B),(1,R))

1.3 RLE-komprimierte BMPs

Um ein Bitmap mit indizierten 8 Bit pro Pixel zu komprimieren, kann man die Lauflängenkodierung nutzen. Die Komprimierung besteht grundsätzlich aus zwei Modi: dem absoluten und dem kodierten Modus.

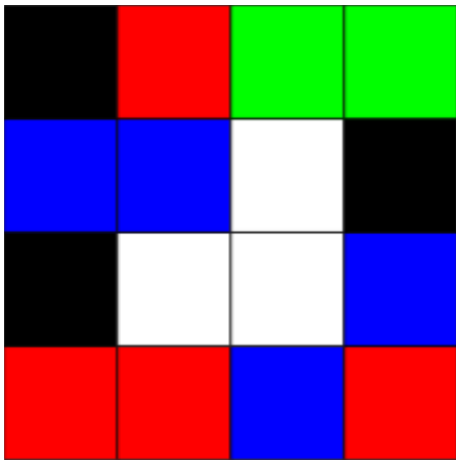

Figure 1: 16 Pixel Bild

1.3.1 Der codierte Modus

Dieser Modus repräsentiert das aktuelle Prinzip von RLE, indem er aus zwei Bytes besteht; das erste gibt die Wiederholungen des zweiten Bytes an, das den wiederholten Pixel repräsentiert.Dieser Modus repräsentiert das aktuelle Prinzip von RLE, indem er aus zwei Bytes besteht; das erste gibt die Wiederholungen des zweiten Bytes an, das den wiederholten Pixel repräsentiert.
Beispiel: [03 04] entspricht [04 04 04]

1.3.2 Der absolute Modus

Dieser Modus wird verwendet, wenn der codierte Modus nicht so speichereffizient ist, wie erwünscht, beispielsweise wenn jeweils zwei aufeinanderfolgende Pixel unterschiedliche Werte haben. In diesem Modus ist das erste Byte ein Null, dass zweite ein Wert im Bereich 0x03-0xff.Dieser stellt die Anzahl der folgenden Bytes dar, die jeweils den Farbindex eines einzelnen Pixels enthalten. Hat man eine ungerade Anzahl an Elemente im 2. Byte, dann ist ein 0 am Ende hinzuzufügen,um an einer 16-Bit-Wortgrenze zu enden.
Beispiel: [00 03 45 56 67 00] entspricht [45 56 67]

1.3.3 Escape-Zeichen

Hat man in einem ersten Byte eine Null und im zweiten einen Wert zwischen 0 und 2, dann entspricht dies einem Escape-Zeichen

1. *0x00 0x00* : Ende einer Linie.
2. *0x00 0x01* : Ende der Bitmap.

3. *0x00 0x02* : Delta: Die 2 Bytes nach dem Escape enthalten vorzeichenlose Werte, die den Versatz nach rechts und oben des nächsten Pixels von der aktuellen Position angeben.

2 Lösungsansatz

2.1 Rahmenprogramm

Dank des Rahmenprogramms hat der Nutzer die Möglichkeit, eine oder mehrere Optionen gleichzeitig zu wählen. Diese Wahl gibt dem Nutzer Flexibilität bei der Ausführung unseres Programms.

Option	Name	Beschreibung
-V.`<Zahl>`	Version	-V 0 : RLE mit SIMD *'Hauptimplementierung'* -V 1 : RLE ohne SIMD
-B`<Zahl>`	Benchmark	Laufzeitmessung in ms .
-o `<Dateiname>`	Output	Ausgabedatei (Comprssed BMP Bild).
-h `oder` –help	Help page	Eine Auflistung aller Optionen des Programms.
`<Dateiname>`	Input	Eingabedatei (BMP Bild).
-t	Tests	Korrektheit des Ausgabearray.

Table 3: Optionen des Rahmenprogramms

Anmerkungen :

1. Wenn die Versionsoption nicht gesetzt ist, sollte ebenfalls die Hauptimplementierung ausgeführt werden.

2. Benchmark hat ein optionales Argument. Diese Option gibt die Anzahl an Wiederholungen des Funktionsaufrufs an. Falls diese Option nicht gesetzt ist, wird die Anzahl standardmäßig auf 3 festgelegt.

2.1.1 Eingabemethode

Die Eingabemethode erhält den Dateinamen einer BMP-Bilddatei und eine Datenstruktur `ImageData` als Eingabe. Falls keine Fehlermeldungen auftreten, extrahiert diese Methode die relevanten Informationen, liest die Pixeldaten aus der Inputdatei und speichert sie im zuvor reservierten Speicherbereich

2.1.2 Ausgabemethode

Die Ausgabemethode liefert dem Nutzer ein komprimiertes BMP-Bild. Sie kopiert die unveränderten Informationen aus der Input-Datei und ändert die Dateigröße, den Kompressionstyp, die Bildgröße und füllt den neuen Pixelbereich (das Ergebnis der RLE-Kompression) an den entsprechenden Positionen ein

2.2 Implementierter Algorithmus

Unsere Methode, die bedienen wird um ein BMP Bild mittels RLE zu komprimieren wird gemäß folgende Signatur funktionieren:

size_t bmp_rle(const **uint8_t*** img_in, **size_t** width, **size_t** height, **uint8_t*** rle_data);

Also wir bekommen ein Zeiger auf einem Array, wo die zu komprimierenden Pixel sich befinden, seine Bereite, um zu wissen wo eine Zeile zu Ende kommt und dementsprechend *(0x00 0x00)*, ein Escape-Zeichen, das das Ende einer Zeile kennzeichnet, und seine Höhe, damit können wir bestimmen, wenn die Kompression zu Ende kommt und *(0x00 0x01)* hinzufügen. Das komprimierte Bild wird dank des 2. Zeigers in einem in dem Rahmenprogramm allozierten Speicherbereich gespeichert Die Method würde Folgende Ansatz verwenden: Ein Zeiger, der zum Schreiben der rle_data wird hier benutzt und wird jedes Mal erhöht, wenn etwas darin geschrieben ist. Bei jeder Zeile beginned vom 1. Pixel prüfen wir auf Gleichheit. Das heißt, wir rechnen, wie viele Pixel beginnend von dem Aktullen den gleichen Wert haben. Das wäre hilfreich umzu bestimmen welcher Modus zu wählen. Hätten wir mindestens 3 aufeinanderfolgende Pixel mit dem gleichen Wert, dann ist codierte Modus *'Encoded Mode'* zu verwenden, weil er besser für die räumliche Effizienz ist (Hier würden wir nur zwei Werten Schreiben (welches Pixel zu schreiben ist und wie oft es sich wiederholt) statt mindestens 5. Werten (zwei werden die Absolute Modus kennzeichen und noch mind. drei Pixel dazu)). Andernfalls, sollte geprüft werden, ob der Absolute Modus die bessere Variante darstellt. Beginnend mit dem Aktuellen Pixel suchen wir das erste das im absoluten Modus geschrieben werden sollte. Dies entspricht einem Pixel, dass sich mindestens 3 mal widerholt und eventuell das Ende der Zeile markiert. Alle davor sollten im Absoluten Modus gespeichert werden. Ein Feld in diesem Fall in dem rle_data-Array wird mit 0 gefüllt, das zweite gibt an, wie viele Pixel im absoluten Modus gespeichert werden sollten, gefolgt von den aktuellen Pixeln, die unverändert aus dem unkomprimierten Bild übernommen werden sollten.

Um Überläufe zu vermeiden, schrieben wir maximal 255 in Absolute Modus auf einmal, was das Maximum von uint8_t entspricht. Wenn ein Pixel, das zum Beispiel sich mehr als 255 widerholt, werden nur die 1. 255 betrachtet und ab den 256. Mal wird die Komprimierung wieder von diesem durchgeführt. Zudem wird die Suche nach dem 1. Pixel, das im codierten Modus zu speichern ist, sich auf das vorvorletzte Pixel in einer Zeile limitieren, weil der Ansatz der Suche sich auf die Überprüfung von zwei nächsten Pixeln basiert. Die zwei letzten Pixel werden in diesem Fall im kodierten Modus einfach geschrieben, oder falls maximal 253 (um Überläufe zu vermeiden) Pixel im absoluten Modus zu schreiben sind, könnten wir diese ebenfalls hinzufügen.

Die Einzige Ausnahme hier, ist wenn die Anzahl der Pixel, die im Absoluten

Modus zu schreiben, nur 1 oder 2 beträgt, weil *(0x00 0x01)* wird als Ende der Datei und *(0x00 0x02)* als Delta verstanden. Um diese Verwirrung zu vermeiden, nutzen wir in diesem Fall einfach den codierten Modus. Danach betrachten wir mit der gleichen Logik das nächste Pixel in der Zeile, bis die zum Ende kommt.*(0x00 0x00)* ist hier zu schreiben, um das Ende der Zeile abzugeben und dann wiederholen wir die Verfahrung mit der nächsten bis zum Erreichen der letzte. In diesem Fügen wir *(0x00 0x01)* hinzu, um abzugeben, dass die Komprimierung beendet ist. Der Wert des Zeigers, der am Anfang der Methode initialisiert ist und sich jedes Mal erhöht, wann ein Wert in rle_data gespeichert ist, entspricht jetzt der Größe der Komprimierten Daten und wird von der Methode zurückgegeben.

2.3 Optimierte Implementierung mit SIMD

Um die Methode zu optimieren, wird hier SIMD (auch bekannt als Single Instruction Multiple Data) an drei Stellen eingesetzt, an denen das Programm verlangsamt werden könnte. Nämlich:

1. die Bestimmung der Häufigkeit des wiederholten Pixels
2. die Anzahl der Pixel, die im absoluten Modus geschrieben werden sollten
3. das direkte Kopieren der Pixel aus der unkomprimierten Datei in den absoluten Modus.

An diesen drei Stellen wird die Berechnung durchgeführt, indem das Eingabe-Array Pixel für Pixel betrachtet wird. Dies kann mit SIMD insbesondere bei Arrays mit einer Breite von mindestens 16 effizient durchgeführt werden. Das ergibt sich daraus, dass hier SSE SIMD verwendet wird, wo die Vektoren 16 Werte von `uint8_t` enthalten können (16 = 128/8).

2.3.1 Bestimmung der Häufigkeit des Wiederholten Pixels

Anstatt ab dem nächsten Pixel Pixel für Pixel auf Gleichheit mit dem Aktuellen zu prüfen, kann man, falls möglich, die nächsten 16 Pixel auf einmal überprüfen. Dies kann geschehen, indem man einen Vektor mit `_mm_set1_epi8` erstellt, der 16 Mal den aktuellen Pixel enthält, und diesen dann mithilfe von `_mm_cmpeq_epi8` mit den nächsten 16 Pixeln vergleicht, die selbst in einem Vektor mit `_mm_loadu_si8` gespeichert sind.

Wenn es keinen Unterschied gibt, kann man die Variable, die die Häufigkeit des aktuellen Pixels darstellt, einfach um 16 erhöhen und das Verfahren dann wiederholen. Falls ein Pixel unterschiedlich ist, wird das entsprechende 8-Bit-Element auf 0 gesetzt. Die Position dieses Elements kann gefunden werden, indem man das höchstwertige Bit jedes 8-Bit-Elements mit `_mm_movemask_epi8` in einen Integer umwandelt, XOR verwendet und dann `_bit_scan_forward` nutzt, um die Position der ersten 1 (die vor dem XOR 0 war, was eine Ungleichheit

bedeutet) in diesem Integer zu finden. Schließlich erhöht man die Häufigkeit um die gefundene Position, und somit ist die Anzahl gefunden. Hier wird die Häufigkeit auch auf maximal 255 begrenzt, und das Verfahren wird beendet, wenn diese Anzahl erreicht oder überschritten werden könnte. Um Überläufe und Segmentation Faults zu vermeiden, wird der naivere Ansatz genutzt, falls in der Zeile nicht mehr genügend Felder für SIMD verfügbar sind.

2.3.2 Bestimmung der Anzahl der zu schreibenden Pixel im absoluten Modus

Anstatt ab dem nächsten Pixel jedes Mal die beiden folgenden Pixel auf Gleichheit mit dem aktuellen Pixel zu prüfen, kann man, falls möglich, drei Vektoren erstellen: Der erste Vektor beinhaltet 16 Werte ab dem aktuellen Pixel, der zweite Vektor beinhaltet 16 Werte ab dem nächsten Pixel und der dritte Vektor beinhaltet 16 Werte ab dem übernächsten Pixel. Mit `_mm_cmpeq_epi8` kann man je zwei Vektoren vergleichen und mit `_mm_and_si128` kann man die Vergleichsergebnisse in nur einem Vektor zusammenfassen. Dieser Vektor enthält dann an jeder Stelle den gewünschten 3-Pixel-Vergleich. Wie auch schon in der ersten Optimierung, wird ein `int` mit `_mm_movemask_epi8` erstellt, das das höchstwertige Bit jedes 8-Bit-Elements in diesem Vektor enthält. Die Stelle der ersten 1, die den drei aufeinanderfolgenden gleichen Pixeln entspricht, wird auch mit `_bit_scan_forward` gefunden und somit ist die Anzahl an Pixeln ermittelt, die im absoluten Modus zu speichern sind. Falls der Vektor nur Nullen enthält, wird die Anzahl der Pixel auch um 16 erhöht, bis das Ende der Zeile oder 255 erreicht ist oder eine Gleichheit gefunden wird. Dazu wird auch nochmal, wie in der ersten Optimierung, der naive Ansatz genutzt, falls es nicht mehr 16 Felder für SIMD in der Zeile gibt.

2.3.3 Schreiben von Pixeln im absoluten Modus

Das naive Schreiben von Pixeln im absoluten Modus funktioniert, indem jedes Pixel einzeln behandelt wird. Dieser Vorgang kann weiter optimiert werden, indem 16 Pixel auf einmal aus dem Eingabe-Array mit `_mm_loadu_si128` in einen Vektor geschrieben werden, der dann mit `_mm_storeu_si128` in das `rle_data`-Array geschrieben wird. Auch hier wird der naive Ansatz verwendet, falls weniger als 16 Pixel zum Kopieren vorhanden sind.

3 Korrektheit

Nachdem wir das Bild komprimiert haben, möchten wir testen, ob die Komprimierungsmethode korrekt ist. Das Ziel ist es, den Eingabe- und Ausgabe-Array zu vergleichen und sicherzustellen, dass der Ausgabe-Array von BMP-Readern lesbar ist. Dafür dient die Methode *'Tests'*. Diese Methode nimmt als Eingabe den Eingabe- und Ausgabe-Array der RLE-Kompressionsmethode, die Breite des Bildes und die Länge. Sie dekodiert den Array und gibt das nicht komprimierte Bild zurück. Gleichzeitig überprüft sie die Breite und Länge mittels eines

spezifischen Verfahrens. Der benutzte Algorithmus unterteilt sich in zwei Teile: Dekodierung und Vergleich.

1. Dekodierung: Es gibt eine Schleife, die überprüft, ob das Ende des Arrays erreicht wurde oder ob der Anzahl der dekodierten Pixel mehr als der Anzahl der Eingabepixel ist. Wenn das Ende erreicht wurde, bedeutet das, dass das Bild ohne Probleme dekodiert wurde, und es wird der Boolean-Variablen "endoffile" der Wert "true" zugewiesen. Um zu überprüfen, ob das Ende niemals erreicht wurde oder die maximale erreichbare Größe überschritten wurde, gibt es die Variable "counter", die die Anzahl der dekodierten Bytes zählt. Die Variable "max" nimmt den maximalen Wert an Bytes auf, der durch die Formel [(Breite * Höhe)] berechnet wird. In jeder Schleife wird überprüft, ob "endoffile" gleich "true" ist oder ob "counter" größer als "max" ist. Wenn ja, geht das Programm zum zweiten Teil über. Wenn nicht, wird überprüft, ob das aktuelle Byte ungleich 0 ist.
   Wenn nicht, bedeutet das, dass es sich um den kodierten Modus handelt. Wenn ja, muss das Programm das nächste Byte kennen. Wenn das nächste Byte gleich 1 ist, handelt es sich um das Ende des Arrays *(0 1)*. Wenn nicht, wird dieses Byte mit 0 verglichen *(0 0)*. Wenn sie gleich sind, ist das Ende der Zeile erreicht. Dazu wird mit Hilfe einer Variable , die die Anzahl der dekodierten Pixel zählt, überprüft, ob die Breite der Zeile korrekt ist oder nicht, und eine zweite Variable , die die Länge zählt, wird inkrementiert. Wenn der Wert des Bytes weder 1 noch 0 ist, handelt es sich um den absoluten Modus. In diesem Fall wird der Wert dieses Bytes gespeichert, da er die Anzahl der folgenden zu dekodierenden Bytes repräsentiert. Es ist wichtig zu beachten, dass nach diesen Pixeln immer eine Null (0) steht, wenn ihre Anzahl ungerade ist, was für Padding dient. Wenn nach dem Beginn der Schleife das zu überprüfende Byte einen nicht-null-Wert enthält, wird dies als kodierter Modus erkannt. Der Wert wird gespeichert, da er die Anzahl der zu speichernden Pixel repräsentiert. Außerdem sind die zu schreibenden Pixel identisch mit dem nächsten Byte. Die Anzahl der dekodierten Pixel wird sowohl im kodierten als auch im absoluten Modus immer zum Zähler der Breite addiert, sodass sie am Ende der Zeile immer mit der Breite verglichen wird. Am Ende der Schleifen wird überprüft, ob die Anzahl der Zeilen, die die Länge repräsentiert, der wahren Länge entspricht oder nicht.

2. Vergleich der Pixel: Zum Schluss wird der dekodierte Array mit dem Eingabe-Array verglichen. Wenn das Programm irgendwann eine falsche Breite, falsche Länge oder ungleiche Pixel feststellt, wird eine Fehlermeldung auf der Konsole angezeigt.

4 Performanzanalyse

4.1 Entwicklung von Laufzeit in Abhängigkeit von Filegröße

In unserem Programm haben wir zwei Methoden implementiert. Eine davon ist mit SIMD optimiert. Im untenstehenden Diagramm haben wir die Laufzeit in Abhängigkeit von der Dateigröße gemessen. Wie es am Anfang aussieht, zeigt die Leistung beider Methoden einen Unterschied: die unoptimierte Methode benötigt durchschnittlich doppelt so viel Zeit wie die optimierte. Auch wenn wir große Dateien komprimieren möchten, zeigt die mit SIMD optimierte Methode eine bessere Leistung als die Methode ohne SIMD.

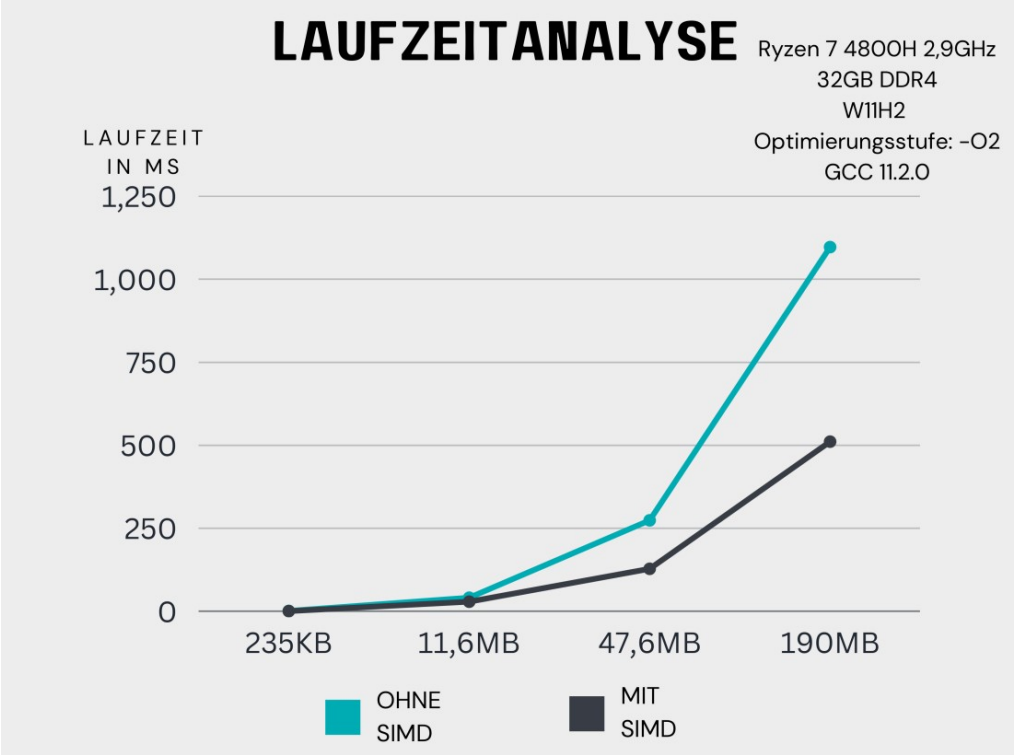

Figure 2: Performanz

4.2 Vergleich zwischen beide Methoden

Zum Vergleich beider Methoden haben wir ein Balkendiagramm erstellt. Im besten Fall sollte ein Bild die gleiche Farbe haben, wie es in unserem zweiten Beispiel (Blaues Bild) erscheint. Hier besteht ein großer Unterschied zwischen 0,62 ms und 0,09 ms (ungefähr siebenmal besser als die unoptimierte Methode).

Für das Projektbeispielbild benötigt die mit SIMD implementierte Methode nur halb so viel Zeit wie die ohne SIMD. Wir haben auch zwei Beispiele genannt: eines enthält viele verschiedene farbige Pixel und das andere zeigt unser MI-Gebäude.

Alle oben genannten Beispiele weisen immer eine deutlich bessere Leistung auf, wenn sie die SIMD-Methode verwenden.

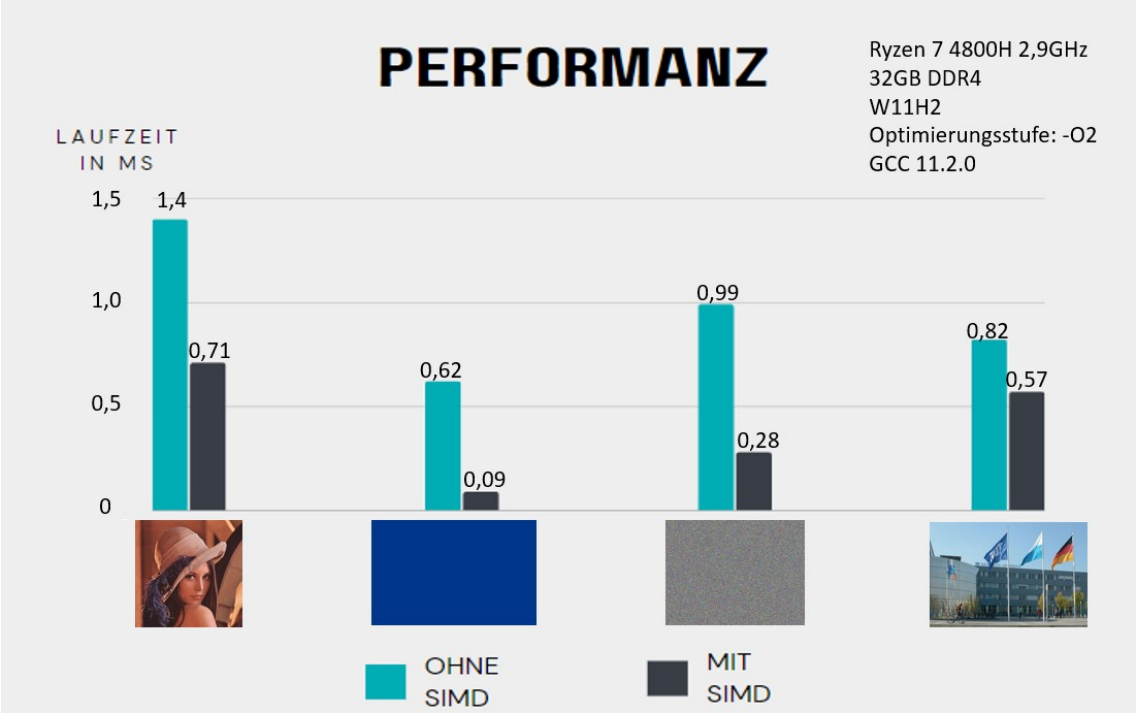

Figure 3: Performanz

5 Zusammenfassung und Ausblick

RLE ist eine verlustfreie Kompression, die im Best-Case-Szenario (alle Zeilen bestehen nur aus gleichen Pixeln) sehr effizient ist.Die SIMD-Implementierung kann die Leistung der Kompression ohne Nebenwirkungen verbessern. Es gibt jedoch auch weitere Optimierungsmöglichkeiten, wie beispielsweise die Verwendung von AVX oder MIMD (Multiple Instructions Multiple Data). Eine Möglichkeit wäre, dass eine bestimmte Anzahl von Threads parallel arbeitet und jeder Thread an einer Zeile arbeitet. Es ist jedoch zu beachten, dass Multithreading nicht in C-Standard vorhanden ist und nur durch externe Libraries wie pthread eingeführt werden kann. Im Worst-Case-Szenario ist das mit RLE komprimierte

Bild jedoch nicht kleiner als das Originalbild, was dem Prinzip der Kompression widerspricht. Dementsprechend sollte man beachten, wann es eine gute Idee ist, RLE zu verwenden, beispielsweise bei der Kompression von Wörterbüchern oder Bildern, die eine dominante Farbe haben.

6 Quellenverzeichnis

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
https://learn.microsoft.com/en-us/windows/win32/gdi/bitmap-compression?redirectedfrom=MSDN