LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

# Grundlagenpraktikum: Rechnerarchitektur

Gruppe 121 – Abgabe zu Aufgabe A203

Wintersemester 2021/22

Adam Karamelo Philipp Czernitzki

## 1 Einleitung

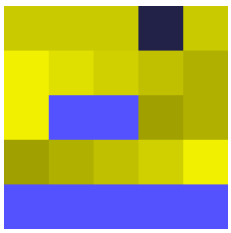

Abb. 1: Rastergrafik 5x5

Für die Darstellung von Grafiken auf digitalen Bildschirmen existieren eine Vielzahl von Bildformaten. Das Bitmap Format von Microsoft, kurz **BMP Format**, war ein beliebtes Format um Rastergrafiken darzustellen. Diese Grafiken können jedoch unkomprimiert Unmengen an Speicherplatz verbrauchen, weshalb es hilfreich ist diese zu komprimieren. Über die **Lauflängenkodierung** können Pixel gleicher Farbwerte zusammengefasst werden und so die Dateigröße minimiert werden. Formate wie PNG und JPEG, die standardmäßig komprimiert sind, erfreuen sich heute großer Beliebtheit. Diese Arbeit beschäftigt sich mit der **Komprimierung von Bilddateien** im Bitmap Format durch die Lauflängenkodierung. Anhand eines Beispielbildes wird das Bitmap Format und die Komprimierung über die Lauflängenkodierung erläutert. Es wird ein Algorithmus für die Kompression erarbeitet und seine Funktionsweise, Performanz und Korrektheit analysiert und geprüft.

## 2 Lösungsansatz

### 2.1 Bitmap Format

8 Bits per Pixel Bitmaps [2] bestehen aus vier Komponenten, einem **BitmapFileHeader** [5], einem **Bitmap Information Header** [3], einer **Color Palette**, und einem oder mehreren **Pixelindizes**. Die Color Palette enthält alle darstellbaren Farben der Grafik im RGB-Format, d.h. in **blau**, **grün** und **rot**. Jeder Pixelindex zeigt auf einen RGB-Eintrag in der Color Palette wodurch der Farbwert des Pixels erhalten wird. Im folgenden wird von 8 Bits per Pixel Bitmaps ausgegangen, für andere Werte kann sich das Format ändern.

#### 2.1.1 Bitmap File Header

Der BitmapFileHeader [5] gibt den Typ, die Dateigröße und das Offset zu den Pixelindizes der Bitmap an. Er ist 14 Bytes groß.

```
00000000  42 4d 7e 00 00 00 00 00  00 00 56 00 00 00        |BM~.....V...|
0000000e
```

Abb. 2: BitmapFileHeader für Rastergrafik

| **Feld** | **Bytes** | **Beschreibung** |
|---|---|---|
| Type | 2 | Zeigt, dass es sich um eine Bitmap handelt.<br>Muss auf **0x4d42** gesetzt sein. |
| Size | 4 | Die Dateigröße in Bytes |
| Reserved1 | 2 | Verwendet von Bildverarbeitungsprogrammen,<br>normalerweise mit **'0'** initialisiert. |
| Reserved2 | 2 | Verwendet von Bildverarbeitungsprogrammen,<br>normalerweise mit **'0'** initialisiert. |
| OffBits | 4 | Offset in Bytes von Anfang der Datei bis zu den Pixelindizes. |

Tabelle 1: Felder des BitmapFileHeader

### 2.1.2 Bitmap Information Header

Im Information Header stehen Informationen über das Bild der Bitmap.
Unter anderem ist die Breite, die Höhe, die verwendete Kompression und die Größe der Pixelindizes in Bytes enthalten.

| **Feld** | **Bytes** | **Beschreibung** |
|---|---|---|
| Size | 4 | Die Anzahl an Bytes benötigt |
| Breite | 4 | Die Breite in Pixel |
| Höhe | 4 | Die Höhe in Pixel |
| Planes | 2 | Anzahl an Flächen, muss **'1'** betragen. |
| BitCount | 2 | Anzahl an Bits die einen Pixel definieren |
| Compression | 4 | **'0'** - keine Komprimierung<br>**'1'** - 8bpp Lauflängenkodierung |
| SizeImage | 4 | Größe der komprimierten Pixeldaten in Bytes.<br>Wenn **'0'** dann hat keine Komprimierung stattgefunden. |
| XPelsPerMeter | 4 | Horizontale Auflösung in Pixel pro Meter des Zielgerätes.<br>Wenn **'0'** dann keine Präferenz |
| YPelsPerMeter | 4 | Vertikale Auflösung in Pixel pro Meter des Zielgerätes.<br>Wenn **'0'** dann keine Präferenz |
| ClrUsed | 4 | Anzahl an Farbindizes. Wenn **'0'** dann nutzt die Bitmap **2^bitCount** Einträge aus der Color Palette |
| ClrImportant | 4 | Anzahl an Farbindizes erforderlich um die Bitmap anzuzeigen. Wenn **'0'** dann sind alle Farbindizes erforderlich. |

Tabelle 2: Felder des BitmapInfoHeader

Das Bitmap Format spezifiziert verschiedene Information Header in unterschiedlichen

Größen. Die umgesetzte Implementierung beschränkt sich explizit nur auf die von Microsoft dokumentierten Information Header: **BitmapCoreHeader** [4] (12 Bytes), **BitmapInfoHeader** [1] (40 Bytes), **BitmapV4Header** [6] (108 Bytes) und **BitmapV5Header** [7] (124 Bytes). Andere Versionen des Information Header sind nicht dokumentiert oder wurden von Dritten entwickelt. BitmapV4Header und BitmapV5Header erweitern die Felder des in Tabelle 2 gezeigten BitmapInfoHeader.

```
00000000  28 00 00 00 05 00 00 00  05 00 00 00 01 00 08 00  |(...............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  08 00 00 00 00 00 00 00                           |........|
00000028
```

Abb. 3: BitmapInfoHeader für Rastergrafik

### 2.1.3 Sonderfall: BitmapCoreHeader

Der BitmapCoreHeader [4] unterscheidet sich vom BitmapInfoHeader, BitmapV4Header und BitmapV5Header. Die *Höhe und Breite des Headers werden **in 2-Byte** statt den üblichen 4-Byte* angegeben. Desweiteren spezifiziert er *kein Feld für die Komprimierung*, daher kann eine Bitmap mit einem BitmapCoreHeader nicht komprimiert werden. Ebenfalls sind *Einträge in der Color Palette **in 3-Byte** statt 4-Byte* angegeben.
Die Implementierung wandelt einen BitmapCoreHeader dementsprechend in einen BitmapInfoHeader um, welcher die Komprimierung durch die Lauflängenkodierung unterstüzt.

| **Feld** | **Bytes** | **Beschreibung** |
|---|---|---|
| Size | 4 | Zeigt, dass es sich um eine Bitmap handelt.<br>Muss auf **0x4d42** gesetzt sein. |
| Width | 2 | Breite der Bitmap in Pixel |
| Height | 2 | Höhe der Bitmap in Pixel |
| Planes | 2 | Anzahl an Flächen, muss **'1'** betragen. |
| BitCount | 2 | Anzahl an Bits per Pixel |

Tabelle 3: Felder des BitmapCoreHeader

```
00000000  0c 00 00 00 05 00 05 00  01 00 08 00              |............|
0000000c
```

Abb. 4: BitmapCoreHeader für Rastergrafik

### 2.1.4 Color Palette

Die Farbpalette der Bitmap besteht aus mehreren **RGBQuad** [8] (4-Tupel) oder **RGBTriple** [9] (3-Tupel) Einträgen. Alle Farbwerte liegen im Intervall von [0, 255]. Es ist zu

beachten, dass nur der BitmapCoreHeader RGBTriple nutzt, alle anderen unterstützten Header nutzen RGBQuad.

- **RGBQuad**:
  (**blau**, **grün**, **rot**, **reserviert** = 0)
  Das reservierte Byte soll 0 sein
- **RGBTriple**: (**blau**, **grün**, **rot**)

Die maximale Größe der Color Palette ergibt sich aus dem **bitCount** Feld des Information Headers über **2^bitCount**. Eine 8 Bit per Pixel Bitmap kann maximal 256 Einträge besitzen.

```
00000000  ff 54 52 00 00 a0 a0 00  00 b0 b0 00 00 c0 c0 00  |.TR.............|
00000010  00 d0 d0 00 00 f0 f0 00  00 ca ca 00 49 22 22 00  |............I"".|
00000020
```

Abb. 5: Color Palette der Rastergrafik

### 2.1.5 Pixelindizes / Pixeldaten

Pixeldaten werden über Indizes auf die Color Palette spezifiert. Ein Pixelindex ist **bitCount** groß. Gemäß Aufgabenstellung werden **8 Bits per Pixel** betrachtet. Entsprechend ist $bitCount = 8$. Ein Pixelindex kann Werte im Intervall von [0, 255] annehmen. Pixelindizes stehen zeilenweise nebeneinander. Jede Zeile, auch **Scan Line** genannt, ist 4-Byte aligned und wird am Ende falls nötig mit **Padding Bytes (0x00)** gefüllt. Die Beispiel Rastergrafik (5x5 Pixel) enthält 3 Padding Bytes (0x00) am Ende jeder Scan Line. Der erste Pixelindex beschreibt den Pixel links unten im Bild, da Bitmaps von unten nach oben konstruiert werden. [10]

```
00000000  00 00 00 00 00 00 00 00  01 02 03 04 05 00 00 00  |................|
00000010  05 00 00 01 02 00 00 00  05 04 03 02 01 00 00 00  |................|
00000020  06 06 06 07 06 00 00 00                           |........|
00000028
```

Abb. 6: Pixelindizes der Rastergrafik

In der Abbildung ist zu erkennen, dass 5 gleiche (blaue) Pixel geschrieben werden und 3 Padding Bytes folgen.

## 2.2 Implementierung

Ziel der Implementierung ist es eine Bitmap über die Lauflängenkodierung (run-length-encoding) zu komprimieren.

### 2.2.1 Annahmen

- Das Programm muss gemäß Aufgabenstellung explizit nur 8 Bit per Pixel Bitmaps unterstützen.
- Color Profiles wie sie im BitmapV5Header existieren können, werden nicht unterstützt.
- Die Komprimierung unterstützt Bilder bis zur einer Auflösung von 7680x7680. Das heißt Bilder mit einer Auflösung bis zu 8K werden unterstützt.

  *Die theoretisch maximalste Auflösung beträgt 2.147.483.648 Pixel. Da die komprimierten Pixelindizes in einen Buffer geschrieben werden, der mit **malloc(size_t)** alloziiert wird, hängt die maximal darstellbare Auflösung von der maximalen Größe der alloziierbaren Bytes ab. Auf einem 32 Bit System können bis zu 4.294.967.296 Bytes alloziiert werden. Im schlimmsten Fall sind alle nebeneinanderliegende Pixelindizes unterschiedlich und wir schreiben nur im Encoded Modus. Dann werden für jeden Pixelindex immer 2-Byte geschrieben.*

### 2.2.2 Validierung

Bevor die Bitmap komprimiert wird, prüft die Implementierung die Header Felder aus der Spezifikation, ob diese valide und richtig für eine unkomprimierte Bitmap gesetzt sind.

### 2.2.3 Komprimierung

Es wird das **Compression** Feld im Bitmap Information Header auf **'1'** gesetzt. Das Bitmap Format bietet 2 Modi für das schreiben von komprimierten Pixelindizes.

**Encoded Modus** Im Encoded Modus werden gleiche nebeneinander stehende Pixelindizes zusammengefasst. Man schreibt die Wiederholungen und den Pixelindex. Das heißt es werden 2-Byte in folgendem Format geschrieben: ***<Wiederholungen> <Pixelindex>*** In der Beispielrastergrafik befinden sich in der ersten Scan Line **5 gleiche Pixel**, dargestellt als ***0x00 0x00 0x00 0x00 0x00***, diese werden entsprechend zu ***0x05 0x00*** zusammengefasst.

**Absolute Modus** Im Absolute Modus werden unterschiedliche nebeneinander stehende Pixelindizes zusammengefasst. Man schreibt ein Nullbyte, die Anzahl der unterschiedlichen Pixelindizes und dann die Pixelindizes. Das Format sieht wie folgt aus:
***<Nullbyte> <Anzahl> <1.Pixelindex> <2. Pixelindex> <3.PixelIndex> ...***
Dieser Modus ist 2-Byte aligned, das heißt es muss gegebenfalls noch ein Nullbyte zum Schluss geschrieben werden, sodass die Menge der geschriebenen Bytes gerade bleibt. Ebenfalls muss beachtet werden, dass die Anzahl mindestends drei beträgt, damit dieser Modus verwendet werden kann.

**Scan Line Ende** Ist eine Scan Line durch den Absolute und/oder Encoded Modus komprimiert, werden zwei weitere Bytes geschrieben. Im Normalfall schreibt man am Ende einer Zeile ***0x00 0x00***, um das Zeilenende zu kennzeichnen. Falls man sich in der letzten Scan Line befindet wird jedoch ***0x00 0x01*** geschrieben, um das Ende der Bitmap zu kennzeichnen.

#### 2.2.4 Optimierung der Kompressionsrate

Die Kompressionsrate kann erhöht werden, wenn man den Encoded Modus nur verwendet wenn nötig. Sind mindestens drei nebeneinanderliegende Pixel gleich oder der Absolute Modus kann nicht verwendet werden weil die Anzahl kleiner gleich zwei beträgt, soll der Encoded Modus verwendet werden. In allen anderen Fällen soll der Absolute Modus verwendet werden. Sind zwei nebeneinanderliegende Pixel gleich soll der Absolute Modus verwendet werden.

**Pixelindizes**

```
0x00 0x01 0x02 0x03 0x03 0x00 0x01 0x02
```

**Komprimiert ohne Optimierung** - 2 gleiche Pixel in Encoded Modus

```
0x00 0x03 0x00 0x01 0x02 0x00 0x02 0x03 0x00 0x03 0x00 0x01 0x02 0x00
```

**Komprimiert mit Optimierung**

```
0x00 0x08 0x00 0x01 0x02 0x03 0x03 0x00 0x01 0x02
```

Die Implementierungen von Version 0 bis 2 nutzen diese Optimierung.

### 2.3 Implementierte Versionen

#### 2.3.1 RLE Encoded Modus V3 (*bmp_rle_encode_V3.c*)

Die Version 3 ist ein naiver Ansatz der Lauflängekodierung. Sie nutzt nur den Encoded Modus der Bitmap Komprimierung. Der Algorithmus geht durch die Pixeldaten, vergleicht jeweils zwei Pixelindizes und zählt in einem Zähler wie oft der Pixelindex unverändert bleibt. Ist der nächste Pixelindex nicht mehr gleich, wird der Zähler und Pixelindex im Encoded Modus geschrieben.

#### 2.3.2 RLE Encoded und Absolute Modus V2 (*bmp_rle_V2.c*)

Die Version 2 arbeitet mit 2 Zählern gleichzeitig: **reps** und **diff**. Pixel werden paarweise verglichen. Wenn sie unterschiedlich sind wird **diff** inkrementiert und **reps** zurückgesetzt. Wenn sie gleich sind, werden **reps** und **diff** beide inkrementiert. Diff wird auch inkrementiert, um den Fall zu vermeiden, dass 2 gleiche Pixel im Encoded Modus geschrieben werden, siehe hier. Nur wenn ein unterschiedliches Pixel nach einer Reihe von mindestens 3 gleichen Pixeln entdeckt wird, werden die Pixeldaten geschrieben. Zuerst wird die **diff** Anzahl an Pixeln geschrieben. Danach werden die **reps** von Pixeln geschrieben.

### 2.3.3 RLE Encoded und Absolute Modus V1 (*bmp_rle_V1.c*)

Die Version 1 vergleicht jeden Pixel mit den nächsten zwei, um drei gleiche zu finden. So lange keine drei gleichen Pixeln gefunden werden wird der Zähler **diff** inkrementiert. Bei drei gleichen Pixel, werden die bereits gezählten **diff** geschrieben. Dann wird gezählt ob es weitere gleiche Pixel in einem Zähler **rep** gibt. Gibt es keine gleichen Pixel mehr, werden die gleichen Pixel **rep** geschrieben und es werden wieder **diff** gezählt.

### 2.3.4 RLE Encoded und Absolute Modus mit SIMD Intrinsics V0 (*bmp_rle.c*)

Bei Version 0 werden SIMD Intrinsics benutzt. SIMD ermöglicht durch Vektorisierung, Pixel in 16 Byte Blöcken zu vergleichen und zu speichern. Durch die Intrinsics Funktion $\_mm\_cmpeq\_epi8$ werden jeweils zwei nebeneinanderstehende Pixel verglichen. Eine Maske gibt an ob zwei Pixel gleich waren (0xff) oder ob diese unterschiedlich waren (0x00). Daraufhin werden unterschiedliche Pixel in einem Zähler **diff** und gleiche Pixel in einem Zähler **rep** gezählt. Je nachdem wird entschieden ob der Absolute Modus oder Encoded Modus zum Schreiben verwendet wird. Wenn wir im Absolute Modus schreiben berechnen wir wie oft wir mit SIMD 16 Byte Blöcke schreiben können und schreiben die unaligned Pixeldaten mit *memcpy*.

### 2.3.5 Coding Style

Die durch die Aufgabenstellung erhaltene Methodensignatur wurde geringfügig verändert um die Naming Konvention von Variablen in camelCase zu bewahren.

## 3 Korrektheit

Zunächst vergleichen wir die Rastergrafik rechts. Es ist zu erkennen, dass die Bitmap Komprimierung zwei Byte mehr verwendet wenn die Rastergrafik mit der Lauflängenkodierung komprimiert ist. Das liegt daran, dass die Rastergrafik wenig gleiche Pixel besitzt und diese nicht zusammengefassen werden. Im Absolute Modus werden mindestens sechs Bytes geschrieben und im Encoded Modus mindestens zwei Byte.

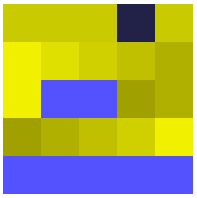

Abb. 7: Rastergrafik 5x5

```
00000000  42 4d 7e 00 00 00 00 00  00 00 56 00 00 00 28 00  |BM~.......V...(.|
00000010  00 00 05 00 00 00 05 00  00 00 01 00 08 00 00 00  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 08 00  |................|
00000030  00 00 00 00 00 00 ff 54  52 00 00 a0 a0 00 00 b0  |.......TR.......|
00000040  b0 00 00 c0 c0 00 00 d0  d0 00 00 f0 f0 00 00 ca  |................|
00000050  ca 00 49 22 22 00 00 00  00 00 00 00 00 00 01 02  |..I""...........|
00000060  03 04 05 00 00 00 05 00  00 01 02 00 00 00 05 04  |................|
00000070  03 02 01 00 00 00 06 06  06 07 06 00 00 00        |..............|
0000007e
```

Abb. 8: Unkomprimierte Rastergrafik

Kann der Algorithmus nun die Pixel nicht zusammenfassen, entspricht das einem erheblichen Mehraufwand/Verbrauch für einen 1-Byte Pixel.

```
00000000  42 4d 80 00 00 00 00 00  00 00 56 00 00 00 28 00  |BM........V...(.|
00000010  00 00 05 00 00 00 05 00  00 00 01 00 08 00 01 00  |................|
00000020  00 00 2a 00 00 00 00 00  00 00 00 00 00 00 08 00  |..*.............|
00000030  00 00 00 00 00 00 ff 54  52 00 00 a0 a0 00 00 b0  |.......TR.......|
00000040  b0 00 00 c0 c0 00 00 d0  d0 00 00 f0 f0 00 00 ca  |................|
00000050  ca 00 49 22 22 00 05 00  00 00 00 05 01 02 03 04  |..I""...........|
00000060  05 00 00 00 00 05 05 00  00 01 02 00 00 00 00 05  |................|
00000070  05 04 03 02 01 00 00 00  03 06 01 07 01 06 00 01  |................|
00000080
```

Abb. 9: Komprimierte Rastergrafik 5x5 - Version 0

```
00000000  42 4d 80 00 00 00 00 00  00 00 56 00 00 00 28 00  |BM........V...(.|
00000010  00 00 05 00 00 00 05 00  00 00 01 00 08 00 01 00  |................|
00000020  00 00 2a 00 00 00 00 00  00 00 00 00 00 00 08 00  |..*.............|
00000030  00 00 00 00 00 00 ff 54  52 00 00 a0 a0 00 00 b0  |.......TR.......|
00000040  b0 00 00 c0 c0 00 00 d0  d0 00 00 f0 f0 00 00 ca  |................|
00000050  ca 00 49 22 22 00 05 00  00 00 00 05 01 02 03 04  |..I""...........|
00000060  05 00 00 00 00 05 05 00  00 01 02 00 00 00 00 05  |................|
00000070  05 04 03 02 01 00 00 00  03 06 01 07 01 06 00 01  |................|
00000080
```

Abb. 10: Komprimierte Rastergrafik 5x5 - Version 1

Abb. 11: Einfarbige Bitmap 150x10

Eine einfarbige Bitmap hingegen in der Größe 150x10 wird mit einer Kompressionsrate von 16:1 komprimiert, dies entspricht einer um 93% reduzierten Bitmapgröße. In diesem Fall werden bis zu 150 Pixel in einer Zeile zusammengefasst und als 2-Byte im Encoded Modus geschrieben.

```
00000000  42 4d 62 00 00 00 00 00  00 00 3a 00 00 00 28 00  |BMb.......:...(.|
00000010  00 00 96 00 00 00 0a 00  00 00 01 00 08 00 01 00  |................|
00000020  00 00 28 00 00 00 00 00  00 00 00 00 00 00 01 00  |..(.............|
00000030  00 00 00 00 00 00 b7 00  ff 00 96 00 00 00 96 00  |................|
00000040  00 00 96 00 00 00 96 00  00 00 96 00 00 00 96 00  |................|
*
00000060  00 01                                            |..|
00000062
```

Abb. 12: Komprimierte Einfarbige Rastergrafik 150x10 - Version 0

```
00000000  42 4d 62 00 00 00 00 00  00 00 3a 00 00 00 28 00  |BMb.....:...(.|
00000010  00 00 96 00 00 00 0a 00  00 00 01 00 08 00 01 00  |................|
00000020  00 00 28 00 00 00 00 00  00 00 00 00 00 00 01 00  |..(.............|
00000030  00 00 00 00 00 00 b7 00  ff 00 96 00 00 00 96 00  |................|
00000040  00 00 96 00 00 00 96 00  00 00 96 00 00 00 96 00  |................|
*
00000060  00 01                                            |..|
00000062
```

Abb. 13: Komprimierte Einfarbige Rastergrafik 150x10 - Version 1

# 4 Performanzanalyse

In diesem Abschnitt wird auf die Laufzeitanalyse und Kompressionsrate der verschiedenen Ansätze des Algorithmus eingegangen. Getestet wurde auf einem System mit einem Intel Core i5-8300H CPU, 2.30GHz, 8GB Arbeitsspeicher, WSL Ubuntu 18.04, 64 Bit, Kernel 4.4.0-19041-Microsoft. Kompiliert wurde mit GCC 7.5.0 mit der Option -O2. Wichtig zu erwähnen ist, dass die vier Implementierungsvarianten sich unterschiedlich im Bezug auf die Eingabedatei verhalten. Hier ist zwischen drei Fällen zu unterscheiden:

- **Worst Case Image** Bilder, die keine drei nacheinander gleiche Pixel haben.
  getestet mit: lena_gray_6C_512x512.bmp , lena_7C_3x10.bmp, random_0C_10x10.bmp
- **Average Case Image:** Bilder die eine vernünftige Menge an gleichen und abwechselnden Pixelindizes beinhalten.
  getestet mit: lena_7C_512x512.bmp, deer_7C_397x706.bmp, nasa_7C_523x549.bmp, nature_7C_380x570.bmp, people_7C_476x317.bmp, planet_7C_499x499.bmp, tank_7C_277x182.bmp
- **Best Case Image:** Bilder die nur aus einer Farbe, einem Pixelindex bestehen.
  getest mit: pink_7C_512x512.bmp

## 4.1 Laufzeitanalyse

In Abbildung 14 wird die Laufzeit der vier verschiedenen Implementierungen verglichen. Es wurden jeweils drei Bilder der Größe 512x512 (aus jeder Kategorie) getestet.
Die Version 0 (V0) ist mit SIMD Optimierungen in allen Fällen um einen Faktor 1.82 bis 4 performanter, abhängig vom Typ der Eingabedatei.
V0, V1 und V2 kodieren die Pixel sowohl in Encoded als auch in Absolute Modus.
Dank SIMD, werden in V0 die Pixel in 16 Byte Blöcken geladen und gespeichert. Deshalb gewinnen wir stark an Performanz.
Die V1 inkrementiert abwechselnd **reps** und **diff** gemäß des Resultats der zwei Pixelvergleiche, was einen Overhead erzeugt.
Die V2 macht nur einen paarweisen Vergleich, das aktuellste mit dem nächsten. Allerdings arbeitet V2 mit zwei Zählern die oft zusammen inkrementiert werden und auch

öfters überprüft werden müssen, was sich stark auf die Laufzeit einwirkt. Bei häufig wechselnden Pixelwerten ist V2 performanter, sonst V1. Die V3 ist in diesem Fall sehr ineffizient, weil nach unterschiedlichen Pixeln immer direkt geschrieben wird.

Interessant zu verstehen ist warum Worst Case Images eine bessere Laufzeit als Average Case Images haben, bei den Implementierungsversionen, die beide Modi nutzen. Denn im Worst Case wird nur ein Zähler **diff** inkrementiert, da alle Pixel unterschiedlich sind. Somit werden weniger Bedingungen überprüft.

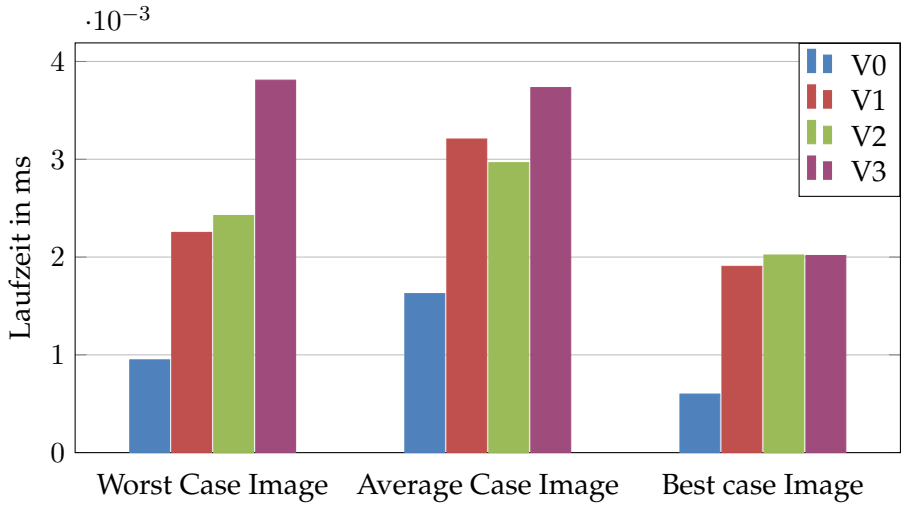


Abb. 14: Laufzeit von Implementierungsversionen für verschiedene Bild Typen

## 4.2 Kompressionsrate

Die Kompressionsrate hängt stark vom benutzen Modus der Komprimierung ab. Die ersten drei Versionen, nämlich V0, V1, V2 benutzen beide Modi optimal bezüglich der Komprimierung. Sie sind gleich mächtig in der Kompressionsrate. V3 hingegen ist mit Encoded Modus viel ineffizienter und nur dann zu verwenden wenn die Bilddatei eine sehr große Menge an gleichfarbigen Pixeln enthält.

In Abbildung 15 ist die Kompressionsrate im Sinne von Speicherplatz Zugewinn dargestellt. Beide Implementierungen V0 und V3 werden mit verschiedenen Bitmaps getestet und ihre Kompressionsraten verglichen. Wie oben argumentiert, ist deutlich zu sehen, dass V0 strikt besser als V3 ist. Nur bei einfarbigen Bilder ist die Kompressionsrate gleich.

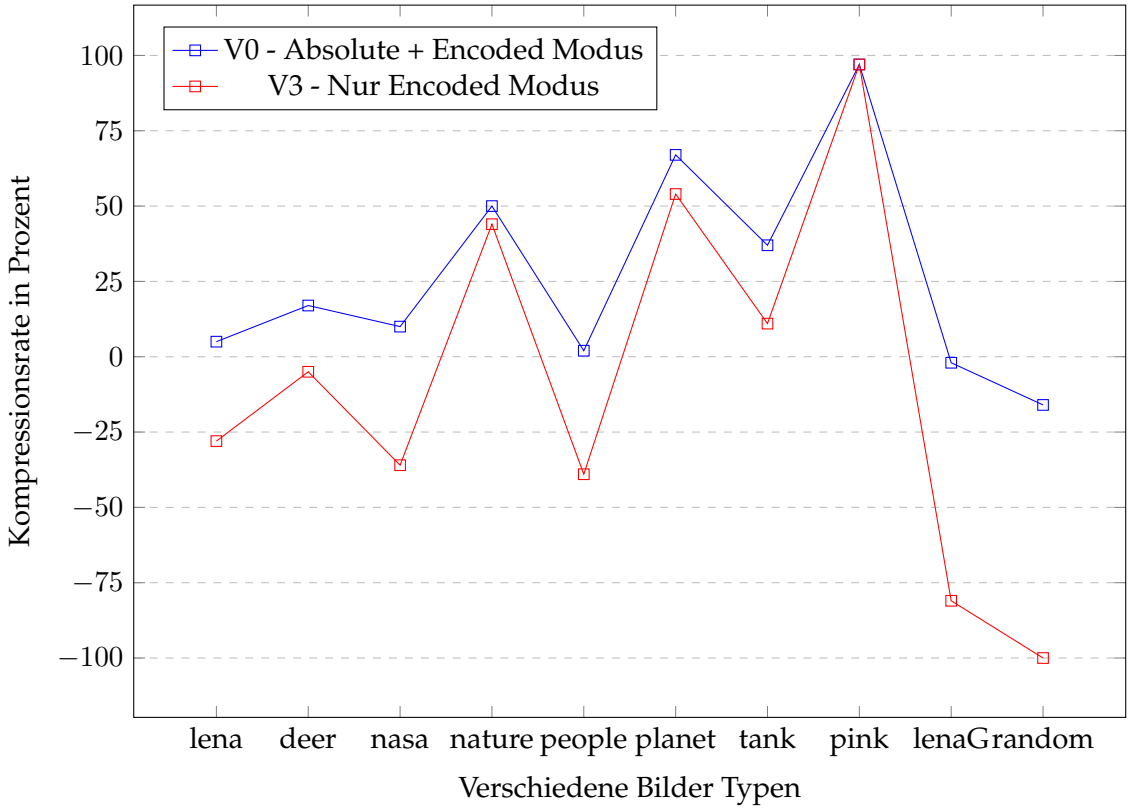


Abb. 15: Kompressionsrate bei verschiedenen Bilder

## 5 Zusammenfassung und Ausblick

Die Lauflängenkodierung ermöglicht eine starke Verbesserung des Speicherverbrauchs, indem gleichfarbige Pixel zusammengefasst werden. Allerdings ist die Lauflängenkodierung nicht immer das effizienteste Komprimierungsverfahren, denn sie verkleinert nicht oder nur gering die Dateigröße von Bitmaps, die wenige gleichfarbige Pixel beinhalten. Für Bitmaps mit vielen gleichfarbigen Pixeln ist die Lauflängenkodierung immer noch ein effizientes Verfahren um die Dateigröße zu minimieren, so können im Average Case gute Kompressionsraten erreicht werden und im Best Case noch viel stärkere Kompressionsraten bis zu 64:1 (beim Schreiben von 255-Byte Blöcken).

## Literatur

[1] Microsoft. *BITMAPINFOHEADER structure*. Microsoft, April 2018. `https://docs.microsoft.com/en-us/previous-versions/dd183376(v=vs.85)`, visited 2022-02-05.

[2] Microsoft. *About Bitmaps*. Microsoft, January 2021. `https://docs.microsoft.com/en-us/windows/win32/gdi/about-bitmaps`, visited 2022-02-05.

[3] Microsoft. *Bitmap Header Types*. Microsoft, July 2021. `https://docs.microsoft.com/en-us/windows/win32/gdi/bitmap-header-types`, visited 2022-02-05.

[4] Microsoft. *BITMAPCOREHEADER structure (wingdi.h)*. Microsoft, February 2021. `https://docs.microsoft.com/en-us/windows/win32/api/wingdi/ns-wingdi-bitmapcoreheader`, visited 2022-02-05.

[5] Microsoft. *BITMAPFILEHEADER structure (wingdi.h)*. Microsoft, April 2021. `https://docs.microsoft.com/de-de/windows/win32/api/wingdi/ns-wingdi-bitmapfileheader`, visited 2022-02-05.

[6] Microsoft. *BITMAPV4HEADER structure (wingdi.h)*. Microsoft, February 2021. `https://docs.microsoft.com/en-us/windows/win32/api/wingdi/ns-wingdi-bitmapv4header`, visited 2022-02-05.

[7] Microsoft. *BITMAPV5HEADER structure (wingdi.h)*. Microsoft, February 2021. `https://docs.microsoft.com/en-us/windows/win32/api/wingdi/ns-wingdi-bitmapv5header`, visited 2022-02-05.

[8] Microsoft. *RGBQUAD structure (wingdi.h)*. Microsoft, February 2021. `https://docs.microsoft.com/en-us/windows/win32/api/wingdi/ns-wingdi-rgbquad`, visited 2022-02-05.

[9] Microsoft. *RGBTRIPLE structure (wingdi.h)*. Microsoft, February 2021. `https://docs.microsoft.com/en-us/windows/win32/api/wingdi/ns-wingdi-rgbtriple`, visited 2022-02-05.

[10] Uday Hiwarale. *Bits to Bitmaps: A simple walkthrough of BMP Image Format*, October 2019. `https://medium.com/sysf/bits-to-bitmaps-a-simple-walkthrough-of-bmp-image-format-765dc6857393`, visited 2022-02-05.