LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 255 – Abgabe zu Aufgabe A403
Sommersemester 2022

Dinh Cuong Pham Dinh Cuong Luc Anton Do

1 Einleitung

Die Informationstheorie ist ein Teilbereich, die aus der Wahrscheinlichkeitstheorie und Statistik der mathematischen Theorie hervorgeht. Sie befasst sich vornehmlich mit technisch-statistischen Problemen, wie beispielsweise der Entropie, die Übertragung, die Kompression/Dekompression oder die Kodierung von Information basierend auf deren Gegebenheiten.

Diese Ausarbeitung setzt sich mit dem Teilbereich der Datenkompression beziehungsweise der Datendekomprimierung aus. Die Datenkomprimierung ist ein Vorgang bei der die Information einer Menge von Daten verdichtet oder reduziert wird. Die daraus resultierenden komprimierten Daten benötigen so deutlich weniger Speicher, sodass sich die Speicherkosten immens reduzieren lassen. Aufgrund des reduzierten Speicherplatzes können einerseits Speicherkosten verringert werden und andererseits lässt sich mehr Speicher für zusätzlichen Daten freigeben. Ein weiterer Vorteil von komprimierten Daten, ist die schnellere Datenübertragung, da die Daten weniger Bandbreite verbrauchen. Dies führt zu einer erhöhten Effizienz im Bereich der Informationsverarbeitung in sowohl der Industrie, als auch im Privatgebrauch.

Die Aufgaben dieses Projektes befassen sich mit den Bereichen Theorie und Implementierung.

Die Problemstellung der Theorie können in die Bereiche Darstellung von Bilddateien (Kapt. 2 Lösungsansatz), Zwei Algorithmen zur Bilddateikomprimierung (Kapt. 2 Lösungsansatz), Analyse der Kompressionsrate (siehe Kapitel 3. Korrektheit) und Analyse der Performanz (siehe Kapitel 4. Performanzanalyse) aufgeteilt werden.

Als Datenbasis eines Bildes wird hier ein Matrix von positiven ganzzahligen Werten - ein Raster von Pixeln - benutzt. Für jeden Pixel werden drei Werte, die die Intensität einer Farbe (Rot, Grün, Blau) dementsprechend darstellt. Eine direkte Speicherung solcher Pixelraster ist das PPM-Dateiformat.

Im folgenden Kapitel wird das PPM Format weiter beschrieben. Zusätzlich wird die Hauptimplementierung, die Lauflängenkodierung, vorgestellt und mögliche Alternativen werden diskutiert und ggf. näher erläutert.

2 Lösungsansatz

Die Portable Pixmap (kurz PPM) lässt sich in verschiedenen Varianten einordnen. In diesem Projekt wird allerdings nur die Variante für 24 Bit Farbtiefe betrachtet, durch

die sich rund 16,7 Millionen Farben abspeichern lassen. Eine PPM Datei besteht aus zwei wesentlichen Teilen: Header und Bilddaten. Der erste Eintrag im Header kennzeichnet das PPM Format. Die beiden möglichen Formate können als P3 für ASCII- bzw. P6 für binär-dargestellte Zahlenwerte spezifiziert werden. Die folgenden zwei Nummern deuten die Breite und Höhe von der Graphik an, während der nächste Wert den maximalen Farbwert anzeigt. Kommentare werden mit einem # gekennzeichnet. Die nachfolgenden Werte speichern die Farben der Pixel im RGB Format. Das RGB bezeichnet die drei Grundfarben: Rot, Grün und Blau, die jeweils als vorzeichenloses 8-Bit Integer dargestellt, also insgesamt 24 Bit je Pixel.

(a) PPM Datei (b) PPM Bild

Abbildung 1: Beispiel von PPM Bild und seine Datei

Die Lauflängenkodierung ist ein simpler verlustfreier Algorithmus, der die Daten, durch Gruppierung von sequentiell identischen Symbolen, komprimiert. Die Lauflängenkodierung (kurz RLE, aus dem englisch "Run-Length Encoding") zählt die Anzahl von benachbarten Pixel, die gleiche RGB-Farbwerte haben und verkürzt diese Sequenz in einem vierer Tupel (Länge, Rot, Grün, Blau). Da die Länge, wie auch die Farbwerte, ein unsigned 8 Bit Integer ist, kann ein Tupel nur maximal 255 Pixel speichern. Besitzen mehrere Pixel den gleiche RGB-Werte, so werden sie in mehreren Tupel verteilt. Durch dieses Format wird so kompakt wie möglich gearbeitet, denn jeder Zahlenwert kann in Binärdarstellung durch genau ein Byte, also insgesamt vier Bytes oder 24 Bits, dargestellt werden. Dies entspricht genau dem Zahlenbereich zwischen 0 und 255 bzw. der 24 Bit Farbtiefe des PPM Format.

Die Lauflängenkodierung wird einmal als naive C-Implementierung und einmal als optimierte Implementierung mittels SIMD implementiert. Die naive Implementierung bearbeitet Pixel für Pixel. Im Vergleich wurden in der optimierten Implementierung Intel Intrinsics Instruktionen verwendet, die fünf Pixel gleichzeitig in eine 128-Bit Variable laden und vergleichen. Durch das gleichzeitige Verarbeitung der Pixel, wird die Effizienz des Programm erhöht.

Neben der Lauflängencodierung wird zusätzlich noch ein Vergleichsalgorithmus implementiert. Der Lempel-Ziv-Welch-Algorithmus ist ebenfalls ein verlustfreier Algorithmus zur Datenkomprimierung. Dieser nutzt ein dynamisches Wörterbuch, um wiederholte Informationen durch Abkürzungen zu ersetzen. Das Wörterbuch verwahrt Verbindungen zwischen 9-bis-16-bit-breite Abkürzungen ("Symbol") und Tupeln von mehrerer 8-bit Werte ("Word"). Aufgrund der Einschränkung auf 16 Bit Feldbreite, können maximal 0xFF00 bzw. 65280 Einträge gespeichert werden.

Unser Rahmenprogramm ist für die Ein- und Ausgabe von sowohl ASCII-kodierte Dateien (`P3, RLE_A, LZW_A`) als auch Binärdateien (`P6, RLE_B, LZW_B`) vorkonfiguriert. Außerdem ist die Erkennung von fehlerhaften Dateien wie z.B. invalide Werte oder ungenügende Datenwerten vorprogrammiert. Zusätzlich zu den drei Algorithmen zur

Bild-Kompimierung (V0 - SIMD-optimierte RLE, V1 - Lempel-Ziv-Welch, V2 - naive-C RLE), bietet das Programm auch die Wahl des Dateityp der Ausgabe-Datei: entweder ASCII zur besseren Lesbarkeit oder Binär um eine kompaktere Datei zu erhalten. Es gibt auch zwei optionale Features: `--scale`, in dem eine Bilddatei mehrfach unter sich "skaliert" wird (um z.B. Testeingaben zu generieren); `--noise`, in dem eine "Rainbow-Noise" Bilddatei beliebiger Höhe und Breite generiert wird (das den Worst-Case für Komprimierungsalgorithmen entspricht).

Nicht Teil des Rahmenprogramms, jedoch mit abgegeben, sind zwei Test-Suites, welche unter dem Verzeichnis `Implementierung/` mittels `make tests` aufrufbar sind. Im nachfolgenden Kapitel wird die Korrektheit/Genauigkeit der ausgewählten Algorithmen und dessen Implementierung mit verschiedenen Inputs durch die beigefügte Test-Suites bewertet.

3 Korrektheit

Dieses Kapitel befasst sich mit der Korrektheit und Effizienz der implementierten Algorithmen. Aufgrund der Eigenschaft der verlustfreien Datenverarbeitung, entfällt die Bewertung der Genauigkeit, da die vorgeführten Algorithmen die Daten ohne Informationsverlust verarbeiten. Stattdessen wird die Kompressionseffizienz bzw. die Kompressionsrate getestet, analysiert und letztendlich bewertet.

Hierfür werden verschiedene PPM Dateien mit verschiedener Komplexität verarbeitet, untersucht und miteinander verglichen. Bei den PPM Dateien werden es sich um P6 Formate handeln, da diese die Informationen in binärer Repräsentation speichern. Würde es sich bei diesen Dateien um das P3 Format handeln, also dem ASCII-Format, so können diese zwei- bis dreimal der eigentliche Größe entsprechen. Der Grund dafür ist, dass für jede Ziffer in ASCII ein Character benutzt werden muss. Eine vorzeichenlose Zahl bis 255 lässt sich in Binär mit dagegen nur einem Byte darstellen.

Unter dem Begriff "Komplexität" wurde hier eine besonders relevante Eigenschaft eines Bildes beschrieben. Sie entspricht die Vielschichtigkeit eines Bildes, d.h. eine hohe Anzahl an unterschiedliche Farben, sowie wenige sich wiederholende Sequenzen entsprechen einem Bild mit höherer Komplexität. In folgenden Graphiken soll dieser Unterschied dargestellt werden.

Diese Graphik Low2 entspricht der niedrigen Komplexität, da nur insgesamt 6 verschiedene Farben genutzt werden. Diese Farben bestehen aus zwei sich leicht unterscheidende Töne der Farben Grau, Gelb und Türkis. Die Farben sind kontinuierlich und decken eine Größe Fläche ab.

In der Graphik High1 wird die deutlich erhöhte Komplexität dargestellt. Zu erkennen ist eine Szenerie, die durch die Anzahl der verschiedenen Farben betont wird. Es sind keine kontinuierlichen Farben mehr zu erkennen, da diese in sich verschmelzen und ein Farbgradient erzeugen. Weder Muster noch offensichtlich identische Sequenzen sind hier sichtbar.

Um die Korrektheit und Effizienz der implementierten Algorithmen bewerten zu können werden drei unterschiedliche Graphiken mit verschiedenen Komplexitätsstu-

(a) niedrige Komplexität (Datei: Low2)

(b) hohe Komplexität (Datei: High1)

Abbildung 2: PPM Bilder mit verschiedenen Komplexitäten Stufen

fen verarbeitet. Diese drei Dateien werden einmal als Original und einmal als eine herunter skalierte Version getestet. Der Grund dafür ist die Größe des Wörterbuches des LZW-Algorithmus. Die genutzte Implementierung erlaubt maximal 65280 Einträge. Mit diesem Limit wird die Input Größe der Dateien erheblich eingegrenzt, sodass nur Graphiken mit rund 6500 Pixeln effizient komprimiert werden sollten.

Um die Komprimierungsraten zu testen wird das Compression Rate Test-Suite (aufrufbar durch `make test-comp-rate`) einen bereitgestellten Archiv verschiedener PPM Dateien entpacken und dann wird das Programm auf jeder Datei ausgeführt. So wird jede Graphik in ein Format der Lauflängenkodierung oder Lempel-Ziv-Welch verarbeitet. Nach diesem Schritt werden die Dateigrößen nach Kategorie PPM Datei, Lauflängen Datei und Lempel-Ziv-Welch Datei gemessen. Dazu wird vom Tester der Shell-Befehl `du` (Disk Usage) aufgerufen. In drei Log-Dateien unter dem Verzeichnis `Implementierung/test-results/compression-rate` werden sich alle Größen vor und nach der Verarbeitung auffinden lassen.

Die ausgesuchten Graphiken stellen den worst, best und average Case für die Komprimierung der Algorithmen dar. Die Größe wird vor und nach der Verarbeitung untersucht und die Ergebnisse werden anschließend bewertet und begründet.

Angefangen bei simplen Graphiken die den best Case simulieren sollen.

Abbildung 3: niedrige Komplexität (Datei: Low1)

Diese Graphik Low1 hat, wie vorher beschrieben, eine niedrige Komplexität. Es sind oft identische Sequenzen, sowie wenige unterschiedliche Farben zu erkennen. Aufgrund der sich oft und langen wiederholenden Sequenzen sollten die Algorithmen diese Wiederholungen ausnutzen können, um so die Größe der Datei um einiges reduzieren.

- Vor der Komprimierung hat die Datei Low1 einen benötigten Speicherbedarf von 1.26 MB bzw. 1.260.016 Bytes.
- Nach Anwendung der Lauflängenkodierung hat diese Datei eine Größe von 0.39 MB bzw. 395.562 Bytes.

- Nach Anwendung des LZW Algorithmus hat diese Datei eine Größe von 2.37 MB bzw. 2.371.830 Bytes.

Wie erwartet hat die Lauflängenkodierung die Datei komprimiert, sodass sie nun etwa 31,3% der ursprünglichen Größe ist. Der LZW Algorithmus hingegen hat die Datei um 188% vergrößert und somit ein nicht gewünschtes Ergebnis geliefert. Diese Speicherbedarf Erhöhung nach der Verarbeitung mit dem LZW Algorithmus wird später erklärt.

Es wird als nächstes eine skalierte Version getestet.

- Vor der Komprimierung hat die Datei Low1Scaled eine Größe von 19.8 KB bzw. 19.889 Bytes.
- Nach Anwendung der Lauflängenkodierung hat diese Datei eine Größe von 26.256 KB bzw. 26.256 Bytes.
- Nach Anwendung des LZW Algorithmus hat diese Datei eine Größe von 37.25 KB bzw. 37.252 Bytes.

Dieser Testinput betont die Schwäche der Lauflängenkodierung. Die Datei hat sich um 32% vergrößert, da der Algorithmus mit kürzeren Sequenzen arbeiten muss und durch die kürzeren Sequenzen, sind auch weniger identische Sequenzen möglich. Dies hat die Folge, dass die Lauflängenkodierung nicht effizient arbeiten kann. Für den LZW Algorithmus veränderte sich die Effizienz nicht.

Als nächstes werden Graphiken der Kategorie mittlere Komplexität betrachtet. Dazu wird folgend Graphik zur Datenverarbeitung genutzt.

Abbildung 4: mittlere Komplexität (Datei: Mid1)

Diese Datei soll den average Case darstellen. Daten, die einige verschiedene Farben und Muster nutzen, jedoch in Maßen und nicht zu komplex. In einer erhöhten Komplexität wird erwartet, dass beide Algorithmen erneut die identischen Sequenzen ausnutzen und die Daten komprimieren, jedoch mit geringerer Effizienz, als für niedrige Komplexität. Nach dem Ergebnis der niedrigen Komplexität, könnte man vermuten, dass der LZW Algorithmus ebenfalls einen erhöhten Speicherbedarf nach der Verarbeitung verursacht.

- Vor der Komprimierung hat die Datei Mid1 eine Speichergröße von 0.36 MB bzw. 360.015 Bytes.
- Nach Anwendung der Lauflängenkodierung hat diese Datei eine Größe von 0.11 MB bzw. 118.176 Bytes.
- Nach Anwendung des LZW Algorithmus hat diese Datei eine Größe von 0.68 MB bzw. 683.206 Bytes.

Wie vermutet, wurde die Datei nach der Anwendung der Lauflängenkodierung komprimiert, aber in abgenommener Effizienz. Die Lauflängenkodierung hat die Datei auf das 32% der Größe komprimiert. Ebenfalls hat der LZW Algorithmus sich wie erwartet verhalten, in dem er die Datei auf das 189% der Größe erweitert. Im Vergleich zu der niedrigen Komplexität hat die Ineffizienz der Verarbeitung, jedoch auch nicht verschlechtert und ist in etwa gleich geblieben.

- Vor der Komprimierung hat die Datei Mid1Scaled (Mid1 mit niedriger Resolution) eine Speichergröße von
  18.3 KB bzw. 18.373 Bytes.
- Nach Anwendung der Lauflängenkodierung hat diese Datei eine Größe von
  14.9 KB bzw. 14.904 Bytes.
- Nach Anwendung des LZW Algorithmus hat diese Datei eine Größe von
  34.288 KB bzw. 34.288 Bytes.

Die Lauflängenkodierung konnte die Graphik auf etwa 81% reduzieren, während der LZW Algorithmus erneut die Datei erweitert auf das 186% des Originals. Entgegen der ersten Beobachtung in der niedrigen Komplexität, ist die Ineffizienz bei einer Größeren Datei nicht gesunken und bleibt bei etwa 185%. Die reduzierte Komprimierung der Lauflängenkodierung lässt sich erneut durch die kürzeren Sequenzen erklären.

Es wird nun ein worst Case betrachtet. Eine Graphik, die zu einer sehr hohen Komplexität kategorisiert werden kann. Sie entspricht einem Bild mit einer hohen Auflösung und soll die Verarbeitung der Algorithmen in einem Extremfall testen.

- Vor der Komprimierung hat die Datei High1 eine Speichergröße von
  53.7 MB bzw. 53.747.729 Bytes.
- Nach Anwendung der Lauflängenkodierung hat diese Datei eine Größe von
  70.41 MB bzw. 70.412.216 Bytes.
- Nach Anwendung des LZW Algorithmus hat diese Datei eine Größe von
  107.39 MB bzw. 107.397.264 Bytes.

Entsprechend der hohen Komplexität und der hohen Resolution der Graphik, haben in diesem Fall beide Algorithmen ihren worst Case. Die Lauflängenkodierung hat diese Datei auf eine 131% Größe Datei verarbeitet und der LZW Algorithmus hat die Datei auf das 199%. Diese extreme Prozentzahl folgt aus die schnell voll gefülltem Wörterbuch - ein später erklärte Sonderfall.

Es wird nun ein letzter Sonderfall betrachtet. Im folgenden wird ein Schachbrettmuster, der Größe 1000 * 1000 Pixel, mit den Farbtönen schwarz und weiß getestet. Erwartet wird der absolute worst Case für die Lauflängenkodierung, da jeder Pixel sich vom Vorgänger unterscheidet. Für den LZW Algorithmus wird erwartet, dass bei diesem Testfall die Effizienz des Algorithmus zeigen lässt.

Abbildung 5: Schachbrett Graphik BlackWhiteAlt

- Vor der Komprimierung hat die Datei BlackWhiteAlt eine Speichergröße von 9 MB bzw. 9.000.016 Bytes.
- Nach Anwendung der Lauflängenkodierung hat diese Datei eine Größe von 3.9 MB bzw. 3.996.100 Bytes.
- Nach Anwendung des LZW Algorithmus hat diese Datei eine Größe von 4.9 MB bzw. 4.999.002 Bytes.

Entgegen der Erwartungen konnten, bei diesem Testfall, beide Algorithmen die Datei komprimieren. Die Lauflängenkodierung konnte diese Datei auf 44% der Größe reduzieren, während der LZW Algorithmus sie auf 55% des Größe komprimierte.

Wie bereits erwähnt, nutzen beide Algorithmen die Wiederholung von Informationen, um Daten zu komprimieren. Jedoch im Worst Case "komprimiert" die Lauflängenkodierung die Größe einer Datei auf das 133% vom Original. Dies passiert in dem Fall, das nahezu eine 1:1 Umsetzung von Pixel nach Tupel sich ergibt - d.h. kaum wiederholende Pixeln.

Der Lempel-Ziv-Welch-Algorithmus hingegen kann eine Datei im worst-Case um das doppelte vergrößern. Bei gelesene Werteserien wobei keine geeignete Abkürzung aus dem Wörterbuch gefunden konnte, mussten jedes Wert im Serie "verbreitet", d.h. zusätzlich zu den origial 8-bits wurden Bits der Wert Null hinzugefügt, um die Breite des Wertes gegen die Breite eines Symbols auszugleichen. In dem Fall, dass das Wörterbuch voll ist, passiert das Verfahren besonders oft. Insbesondere hat die Breite dann ihr Maximumwert von 16-bits. Diese Faktoren ergeben sich zusammen eine Verdoppelung des Dateigrößes.

Nach betrachten dieser Inputs lässt sich schlussfolgern, dass die Lauflängenkodierung in drei von vier Fällen, das erwartete Ergebnis geliefert hat. Für diesen Algorithmus kann angenommen werden , dass in vielen Fällen, besonders bei großen Daten, sich eine effiziente und sinnvolle Komprimierung sehen lässt. Obwohl die Lauflängenkodierung zu den simplen Algorithmen der Datenkomprimierung zählt, stellt sich heraus, dass sie für simple bis komplexe Graphiken gut eignet.

Der Lempel-Ziv-Welch-Algorithmus hingegen stellte sich als sehr Space-ineffizient heraus. Es mangelt das Zurücksetzen des dynamischen Wörterbuches, um neue Muster zu erkennen und zum Wörterbuch hinzu zu fügen. Das Feature wurde aufgrund der fehlenden Zeit und Ressourcen des Teams außer Betracht gelassen. Der Algorithmus wurde zwar korrekt als verlustfreier Prozess implementiert, d.h. eine Datei nach Komprimierung und Entkompimierung verlustlos beibehalten ist.

Schlussendlich kann zu dieser Kapitel gesagt werden: Die Lauflängenkodierung wurde, als Hauptimplementierung, korrekt und effizient implementiert und funktioniert in den jeweiligen Fällen wie erwartet. Die Vergleichsimplementierung kann im Gegensatz dazu nur als mangelhaft bezeichnet werden. Im folgenden Kapitel wird ein anderes Merkmal einer Performanz analysiert und bewertet.

4 Performanzanalyse

In diesem Abschnitt wird die Laufzeit der Algorithmen analysiert. Zuerst wird die Leistung der Algorithmen bei verschiedenen Dateikomplexitäten gemessen. Im nächsten Teil analysieren wir die Laufzeit des Programmes unter Eingabe von einer Reihe skalierten Dateien, deren Größe von 1 bis zu 100 MB variiert. Die Zeit für die Verarbeitung der I/O wird nicht berücksichtigt, d.h. lediglich der Zeitraum vor und nach dem Aufruf der Komprimierungsfunktionen wird untersucht.

Die Zeitmessungen wurden auf Fedora Release 35 x86_64 mit Intel i7-1165G7 (8) der 11. Generation bei 4,700 GHz CPU durchgeführt. Das Programm wurde mit -O3 kompiliert, somit sind alle Kompiler-Optimierungen enthalten. Für die Messung haben wir ein Test-Suite (mittels `make test-performance` aufrufbar) implementiert, dass das Bild scale-orig mit der Größe 1 MB mehrmals hochskaliert, um die Testdateien zu erzeugen. Damit ein korrektes Ergebnis erzielt werden kann, wird das Skript mit einer passenden Anzahl an Iterationen ausgeführt, sodass die gesamte Laufzeit mindestens eine Sekunde beträgt. Das Programm verwendet das Systemcall clock_gettime mit CLOCK_MONOTONIC zur Zeitmessung.

Abbildung 6: Datei: scale-orig

Zuerst messen wir die Laufzeit beider Version von RLE (naiv und optimiert) und LZW bei verschiedenen Eingaben. Durch die Datei High1 ergeben sich folgende Laufzeitwerte:

- Naives RLE: 13.145 Sekunden (bzw. 0.013145 Sekunden per Iteration)
- Optimiertes RLE: 10.329 Sekunden (bzw. 0.010329 Sekunden per Iteration)
- LZW: 279.866 Sekunden (bzw. 0.279866 Sekunden per Iteration)

Wenn wir das naive RLE als Basis festlegen, läuft die optimierte Version 21,42% schneller, während das LZW etwa 2029% langsamer ist. Diese Ausgabe kann damit erklärt werden, dass die optimierte Implementierung aufgrund der Verwendung der Intrinsics-Anweisungen schneller arbeitet. Die 128 Bit Variablen werden verwendet, um bis zu fünf Pixel gleichzeitig zu verarbeiten. Im Vergleich, kann die naive Version nur Pixel für Pixel zählen. Der LZW hat eine schlechte Laufzeit, weil es ein dynamisches Wörterbuch bilden muss. Das heißt, es muss für jedes eingelesen Wort geprüft werden, ob das Wort bereits im Wörterbuch enthalten ist. Wenn nicht, wird das Wort dort gespeichert. Dieser Suchvorgang kostet viel Zeit, da für jedes Symbol/ jeden Wert muss durch das ganze Wörterbuch durchgesucht werden. Das ergibt eine Laufzeitkomplexität von $\mathcal{O}(n^2)$. Aus Zeitgründen wurde nur eine einfache Version von LZW implementiert und nicht mit SIMD oder “smart dictionary” optimiert.

Nun gehen wir zur Datei mit geringerer Komplexität. Die Datei Mid1 wird mit 10000 Iterationen durchgeführt, damit das Ergebnis präziser ist.

- Naives RLE: 3.72 Sekunden (bzw. 0.000372 Sekunden per Iteration)

- Optimiertes RLE: 3.95 Sekunden (bzw. 0.000395 Sekunden per Iteration)
- LZW: 60.98 Sekunden (bzw. 0.006098 Sekunden per Iteration)

In diesem Fall läuft die optimierte Version 6,18% langsamer als die naive Version. Dieses Ergebnis wurde nicht erwartet, lässt sich aber erklärt werden. Der Grund ist die geringe Dateigröße, da die Einrichtungszeit für SIMD Variablen in der optimierten Implementierung mehr Zeit in Anspruch nimmt. Das LZW braucht, gegenüber der Basis, etwa 1639% mehr Laufzeit aufgrund der ineffizienten Wörterbuchssuche.

(a) RLE: naiv vs optimiert (b) optimiertes RLE vs LZW

Abbildung 7: Vergleich von der Laufzeit der Algorithmen

Im nächsten Fall untersuchen wir die Laufzeit der wenig komplexen Datei Low1:

- Naives RLE: 1.546 Sekunden (bzw. 0.001546 Sekunden per Iteration)
- Optimiertes RLE: 1.483 Sekunden (bzw. 0.001483 Sekunden per Iteration)
- LZW: 14.69 Sekunden (bzw. 0.01469 Sekunden per Iteration)

Es wird erwartet, dass die optimierte Version in diesem Fall am effizientesten ist. In der Praxis, läuft sie nur 4,25% schneller als die naive Implementierung. Das Eingabebild nutzt wenige Farben, das bedeutet jedoch nicht, dass es der Best Case ist. Ein Faktor ist die Länge der Sequenz identischer Pixel. Je länger die Sequenz ist, desto effizienter arbeitet die optimierte Implementierung. In diesem Fall sind die Sequenzen von identischen Pixeln nicht lang genug, um den Unterschied zwischen zwei Versionen deutlich hervor zu heben. Das LZW läuft überraschenderweise 850% langsamer als das naive RLE. Aufgrund der Inflexibilität des Wörterbuches wird es schnell voll, und weitere Patterns im Eingabebild konnte nicht erkannt werden.

Im letzten Fall untersuchen wir die bereitgestellte Datei tick.ppm. Ähnlich wie bei der Datei Mid1 ist die Dateigröße zu klein und wir müssen sie 10000 Mal ausführen.

- Naives RLE: 1.711 Sekunden (bzw. 0.0001711 Sekunden per Iteration)
- Optimiertes RLE: 1.313 Sekunden (bzw. 0.0001313 Sekunden per Iteration)
- LZW: 39.6 Sekunden (bzw. 0.00396 Sekunden per Iteration)

Die Datei enthält viele lange Folgen von redundanter Informationen. Das Programm liefert das erwartete Ergebnis: die optimierte RLE läuft 30,31% schneller als die naive, während das LZW 2214% langsamer läuft. Wie wir sehen, verbessern lange Sequenzen mit identischen Pixeln die Laufzeit von LZW nicht.

Im Folgenden messen wir die Laufzeit von Algorithmen auf der gleichen Datei `scale-orig.ppm`, jedoch in unterschiedlichen Größen. Die folgende Grafik zeigt das Ergebnis der Tests. Aufgrund der Kompiler-Optimierung -O3 sehen wir kaum einen Unterschied zwischen naiven und optimierten Versionen von RLE. Zwischen RLE und LZW ist aufgrund ihrer Effizienz noch ein großer Abstand.

(a) RLE: naiv vs optimiert

(b) optimiertes RLE vs LZW

Abbildung 8: Vergleich von der Laufzeit der Algorithmen auf verschiedenen Dateigrößen

Die optimierte Implementierung von RLE könnte weiter optimiert werden, da derzeit eine Vergleichsschleife verwendet wird. Um die Anzahl der Pixel mit identischen RGB-Werten zu berechnen, können wir `mask` und `popcnt` verwenden. Diese können die durchschnittliche Anzahl an Iterationen reduzieren.

5 Zusammenfassung und Ausblick

Das Ziel dieses Projektes war, sich mit einem kleinen Themenbereich der Informationstheorie, der Datenkomprimierung, zu beschäftigen. Es sollten verschiedene Algorithmen implementiert werden und diese sollten dann analysiert und bewertet werden.

Aus den Ergebnissen lässt sich schließen, dass obwohl die Lauflängenkodierung ein simpler Algorithmus zur Datenkomprimierung ist, diese sich als ein effizienter und relativ schneller Algorithmus nennen lässt. Wie jeder Algorithmus hat dieser seine Stark- und Schwachpunkt, deswegen sollte er in seinem jeweiligen Bereich genutzt werden.

Der Lempel-Ziv-Welch-Algorithmus, im Gegensatz dazu, lässt sich als ein eleganter, aber sehr schwer umsetzbarer Algorithmus kategorisieren. Ein großes Problem in dessen Implementierung war der Mangel eines “Wörterbuch Überwachers”, der das dynamische Wörterbuch ab und zu zurücksetzen, um eine Pattern-adaptive Komprimierung zu ermöglichen. Dies verletzt besonders die Komprimierungsrate und die Laufzeit des Algorithmus.

Rückblickend lässt sich sagen, dass eine andere Vergleichsimplementierung effizienter sein könnten, jedoch auch mit ihren Problemen kommen würde. Beispielsweise wurde während des Projektes auch der Huffman Algorithmus in Betracht gezogen,

der aufgrund seiner Implementierung, das Erstellen eines Baum-Datentypes benötigte. Die Speicherung bzw. Verarbeitung solcher Datentyp würden auf jedem Fall zu einem größerem Bedarf an Speicherplatz und Laufzeit führen.

Bei der Recherche wurden oftmals Kombinationen von einigen Algorithmen erwähnt. Diese wären im Rahmen des Projektes jedoch weitaus zu Komplex oder zu Zeitaufwendig. Schlussendlich lässt sich im Nachhinein sagen, dass in dem Bereich Datenkomprimierung viel komplexe Problemen sich versteckt, auch für “simpel” Algorithmen. Um eine gute Datei-Kompressor zu entwickeln, müssen viele Techniken und Algorithmen genutzt und angepasst werden.

Literatur