FH Aachen
Fachbereich 9
Studiengang Angewandte Mathematik und Informatik B.Sc.
Wintersemester 2023/2024
1. Betreuer: Prof. Dr. rer. nat. Philipp Rohde
2. Betreuer: Dr.-Ing. Julian Hofmann

RWTH Aachen
Institut für Wasserbau und Wasserwirtschaft
Mies-van-der-Rohe-Straße 17
52074 Aachen

Seminararbeit

Effiziente Kompression von hydrodynamischen 2D-Simulationsdaten

von Clemens Siebers
Matrikelnummer: 3527944

Aachen, den 14. Januar 2024

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema „Effiziente Kompression von hydrodynamischen 2D-Simulationsdaten“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Clemens Siebers

Aachen, den 14. Januar 2024

Zusammenfassung

Mit dieser Arbeit werden verschiedene Verfahren zur verlustlosen Datenkompression daraufhin getestet, ob sie sich für eine effiziente Speicherung von Trainingsdaten für ein KI-Modell eignen. Diese Trainingsdaten für das KI-Modell liegen als dreidimensionales *Array* vor und wurden durch Simulationen, mithilfe von hydromechanischen Modellen erstellt. Das *Array* speichert Hochwasserstände für unterschiedliche Gebiete in einem gewissen Zeitraum. Diese Datensätze liegen bisher in unkomprimierter Form vor und haben einen relativ großen Speicherbedarf. Beispielsweise hat der in dieser Arbeit verwendete Datensatz, welcher einen Staudammbruch in der Ukraine simuliert, einen Speicherplatzbedarf von 9,22 Gigabyte.

Bei den ausgewählten Verfahren handelt es sich um den LZMA, LZ4, Brotli-Algorithmus und den Zstandard. Zusätzlich wurde vermutet, dass viele Elemente des Arrays den Wert Null haben, daher wird die Eignung der Darstellungsarten CSR und CSC getestet, da diese *Sparse* Matrizen (Matrizen mit vielen Elementen, mit dem Wert Null) besonders speicherplatzsparend speichern können.

Die ausgewählten Verfahren wurden an drei, für die Trainingsdatensätze repräsentativen, Testdatensätzen getestet. Zusätzlich zu dem bereits erwähnten Ukraine Datensatz, wurden zwei Datensätze für die Stadt Aachen aus zwei unterschiedlichen Simulationen zugrunde gelegt. Die Tests wurden auf einem Windows 11 Betriebssystem mit einem AMD Ryzen 7 5700G Prozessor durchgeführt. Um die Verfahren zu testen, wurde eine Erweiterung für die 7-Zip Software genutzt. Damit ist es möglich, die Verfahren mit unterschiedlichen Parametern zu testen. Dazu gibt es unterschiedliche Level der einzelnen Verfahren, womit eine Variation der Ergebnisse zu erkennen ist. Diese Level erzielen in aufsteigender Reihenfolge jeweils eine größere Kompression der Daten. Diese größere Kompression geht allerdings tendenziell mit langsameren Laufzeiten einher. Aufgrund der Hauptanforderung, dass die Dekompressionszeit der Daten möglichst gering sein soll, soll das Verfahren ermittelt werden, welches bei der schnellsten Dekompressionszeit die höchste Speicherplatzeinsparung erzielt.

Die Ergebnisse des Ukraine-Datensatzes unterscheiden sich grundsätzlich von den Datensätzen der Stadt Aachen. Für den Ukraine Datensatz ist das Ergebnis ein Derivat des LZMA, der LZMA2. Bei den Datensätzen für die Stadt Aachen ist das Ergebnis zwar auch ein Derivat des LZMA, es handelt sich aber um den Fast-LZMA2. Zudem wurde deutlich, dass weitere Speicherplatzeinsparungen durch den Wechsel des ursprünglich verwendeten Datentyps möglich sind. Einen Wechsel von *float16* zu *integer16* hat bei den Aachener Datensätzen eine zusätzliche Einsparung vom Faktor vier ergeben, dafür jedoch eine langsamere Dekompressionszeit. Ein Wechsel von *float64* zu *float32* hat bei dem Ukraine Datensatz eine Einsparung vom Faktor zwei ermöglicht. Zudem wird dieser Datensatz um den Faktor 10 kleiner, wenn die CSR-Darstellungsart verwendet wird. Die Aachener Datensätze benötigen bei der Verwendung dieser Darstellungsform mehr Speicherplatz, als vorher. Als Kompromiss für eine allgemeine Lösung wurde das fünfte Level des LZMA2 gewählt. Für eine einheitlich optimale Lösung für alle Datensätze besteht weiterer Forschungsbedarf.

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren hat sich die Anzahl an Starkregenereignissen aufgrund des Klimawandels stark erhöht. Beispiele dafür sind die Überschwemmung des Ahrtals im Juli 2021 und bereits zwei Jahre später, zum Jahreswechsel 2023/2024, die außergewöhnlich ausgeprägten Hochwasserereignisse in Niedersachsen. Doch auch in Regionen ohne Flüsse kommt es häufiger zu Hochwasserereignissen, z.B. die Überschwemmungen aufgrund Starkregens in der Aachener Innenstadt im Mai 2018. Infolgedessen wurden Untersuchungen angestellt, wie frühzeitig Schutzmaßnahmen initiiert werden können. Dabei ist aufgefallen, dass vorhandene Frühwarnsysteme aufgrund von zu hohen benötigten Rechenleistungen zu lange gebraucht haben und somit hierfür nicht geeignet sind. Infolgedessen wurde ein KI-Modell erstellt, mit dem die Rechenzeiten erheblich verkürzt und die Vorhersagegenauigkeit zukünftig erhöht werden soll. Dieses Modell soll mit den Daten zu Hochwasserständen, welche mit den bisherigen verwendeten hydromechanischen Modellen simuliert wurden, trainiert werden. Diese Trainingsdaten wurden als 2D-Rasterdatensätze in einem dreidimensionalen Feld gespeichert. Dabei beschreiben die Rasterdatensätze die zweidimensionale Ausdehnung der Gebiete. Als dritte Dimension wurden die Zeiträume hinzugefügt, in denen die Simulationen durchgeführt wurden. Allerdings sind diese Daten unkomprimiert und daher relativ groß.

Das Ziel dieser Arbeit ist es Möglichkeiten zu finden, diese Trainingsdatensätze zu komprimieren und damit den Speicherbedarf zu reduzieren. Außerdem soll die bisherige Nutzbarkeit der Daten durch die Datenkompression nicht eingeschränkt werden. Das bedeutet, dass es möglich sein soll, das Ursprungsformat der Daten in möglichst kurzer Zeit innerhalb weniger Sekunden wiederherzustellen.

Diese Seminararbeit ist in sieben Kapitel unterteilt. Vorangegangen ist die Einleitung, in welcher der Ausgangszustand und die Zielsetzung vorgestellt wurden. Nun wird kurz die gewählte Vorgehensweise beschrieben. Anschließend wird der aktuelle Stand der Technik für verlustlose Kompressionsverfahren beschrieben. Dazu werden zunächst grundlegende Verfahren vorgestellt (2.1), die in den anschließend folgenden moderneren Algorithmen (2.2) teilweise verwendet werden. In dem dritten Kapitel wird zunächst das vorhandene Datenformat (3.1) und die Testdaten (3.2) vorgestellt. Darauf aufbauend werden die ermittelten Anforderungen (3.3) und die verwendete Testumgebung (3.4) vorgestellt. Im Anschluss wird das Einsparpotential durch eine Strukturanalyse der Daten (4.1) erläutert, sowie der Testvorgang beschrieben (4.2). Anschließend werden die Ergebnisse beschrieben, welche dann diskutiert und bewertet (6) werden. Zum Schluss wird eine allgemeingültige Lösung vorgestellt, sowie weiterführender Forschungsbedarf skizziert (7).

2. Stand der verlustlosen Kompressionstechniken

Es gibt verschiedene Techniken zur Datenreduktion. Das generelle Prinzip der Kompression beruht darauf, dass Redundanzen in Datensätzen reduziert werden. Die redundanten Informationen werden mittels verschiedener Ansätze möglichst effizient abgespeichert. Dabei wird zwischen verlustfreien und verlustbehafteten Techniken unterschieden. Im Hinblick auf die Nachnutzbarkeit der Daten durch die KI-Modelle, werden hier nur verlustfreie Methoden vorgestellt.

2.1 Historische Grundtechniken der Datenreduktion

In diesem Kapitel werden zunächst vier Grundtechniken zur Datenreduktion kurz erläutert. Auf diesen Grundtechniken bauen die anschließend vorgestellten moderneren Datenkompressionsverfahren auf.

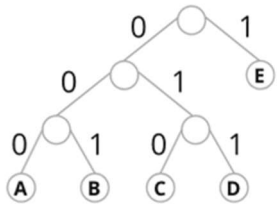
2.1.1 Lauflängencodierung

Die Lauflängencodierung ist ein einfaches Verfahren zur Datenreduktion. Dabei werden aufeinanderfolgende gleiche Zeichen gezählt. Die Anzahl steht nun anstelle der eigentlichen Zeichen codiert in der Datei. Für die Decodierung wird die Anzahl der Zeichen mit einem zusätzlichen Zeichen, welches selten, oder am besten gar nicht, im Text vorkommt codiert. Dieses Verfahren ist nur bei sehr vielen, gleichen und aufeinanderfolgenden Zeichen sinnvoll. Ansonsten führt das zusätzliche Codieren der Länge dazu, dass im Vergleich zum ursprünglichen Format sogar mehr Speicher benötigt wird. Diese Erläuterung ist entspricht der Darstellung im „Grundkurs Codierung“ von Wilfried Dankmeier 2017. [1]

2.1.2 Huffman-Codierung

Von David Huffman stammt ein anderer Ansatz. Anstatt die gleichen aufeinanderfolgenden Zeichen direkt zu codieren, wird hier mit der Häufigkeit der Zeichen in der gesamten Datei gearbeitet. Dazu wird von jedem Zeichen die absolute Häufigkeit in der zu codierenden Datei gezählt. Anschließend wird anhand der Häufigkeiten ein Binärbaum erzeugt. Die einzelnen Häufigkeiten werden zunächst alle als Blätter betrachtet. Dann werden die geringsten zwei Häufigkeiten als kleinerer Teilbaum zusammengefasst. Anschließend wird der Teilbaum mit der summierten Häufigkeit mit den anderen Blättern verglichen, und es werden erneut die zwei geringsten Häufigkeiten zusammengefasst. Dieser Ablauf wird so lange fortgesetzt, bis alle Zeichen im Binärbaum vorhanden sind. Aufgrund der Tiefe im Binärbaum bekommen die Zeichen eine entsprechend lange Bitcodierung. In Abbildung 1 ist das Ergebnis dieses Verfahrens zusammengefasst dargestellt. Hier wurden die einzelnen Buchstaben aufgrund ihrer

Häufigkeit zu Teilbäumen zusammengefasst und aufgrund ihrer Tiefe im Baum mit entsprechend vielen Bits codiert. Das komplette Beispiel und weiterführende Erläuterungen sind in den Ausführungen „Einführung in die Informatik“ von Sebastian Küppers zu finden. [2]



Daraus ergibt sich dann die folgende Codetabelle:

A	B	C	D	E
000	001	010	011	1

Abbildung 1 Beispiel Huffman Codierung [2, S.24]

2.1.3 Arithmetische Codierung

Ein alternatives Verfahren zur Huffman-Codierung ist die arithmetische Codierung. Diese wird auch als *Entropiecodierung* (Codieren der Zeichen in Abhängigkeit von ihrer Häufigkeit) bezeichnet. Die Vorgehensweise zum Codieren und Decodieren wird anhand von Wilfried Dankmeiers Ausführungen im „Grundkurs Codierung“ [1] vorgestellt.

Als erstes wird eine Häufigkeitstabelle der einzelnen Zeichen benötigt. Diese Häufigkeitstabelle wird nun erweitert. Zuerst werden die einzelnen prozentualen Anteile an der Gesamtmenge mithilfe der folgenden Formel berechnet:

$$p = \frac{Häufigkeit\ eines\ Zeichens}{Gesamtanzahl\ der\ Zeichen} \quad [1]$$

Diese prozentualen Anteile p werden nun in Teilintervalle gegliedert. Hierzu bildet das am häufigsten verwendete Zeichen das erste Teilintervall. Dieses beginnt bei Null und würde sich bis zum Anteil p erstrecken. Daran würde sich das nächste Zeichen mit dem zweit meisten Auftreten anschließen. Dieser Anteil wird auf den Ersten kumuliert, sodass das Intervall vom letzten Zeichen als obere Grenze Eins hat. An der Stelle ist es auch möglich, die Teilintervalle mit der absoluten Häufigkeit der Zeichen zu bilden. Infolgedessen würden die Teilintervalle bis zur Gesamtanzahl der Zeichen aufsummiert werden. Für diese Variante hat sich auch der Autor Wilfried Dankmeier entschieden. Daher beziehen sich die anschließenden Formeln auf diese Darstellung. Mithilfe der folgenden rekursiven Formel werden nun die einzelnen Zeichen codiert.

$$c_{u/o}(i+1) = c_u(i) + i_{u/o}(i+1) \cdot \frac{c_o(i) - c_u(i)}{n}$$

$$mit\ c_u(0) = 0, c_o(0) = 100, i = 0{,}1{,}2, \dots, n-1\ \text{S. } 324\ [1]\ 2$$

Das erste Zeichen vom Code $c_u(0+1)$ mit i=0 hat aufgrund der vordefinierten Werte für $c_u(0)$ und $c_o(0)$ für die untere und obere Grenze dieselben Grenzen, wie sie in der Tabelle stehen. Das Prinzip für die anschließende Codierung der folgenden Zeichen besteht darin, dass die Intervallgrenzen der einzelnen Zeichen auf die des Codes angepasst werden. Für das zweite Zeichen werden daher die abgelesenen Grenzen des ersten verwendet, jedoch für das dritte Zeichen die errechneten des zweiten Zeichens. Daraus ergibt sich abhängig von der Codelänge eine entsprechend lange Dezimalzahl. Bei der Berechnung wird daher meist eine im Intervall liegende Kommazahl mit möglichst wenig Nachkommastellen genutzt. An dieser Stelle ist zu beachten, dass hierfür entsprechende Programme verwendet werden sollten, die eine solche Genauigkeit der Darstellung von Fließkommazahlen gewährleisten können.

Die folgende Formel zeigt die Decodierung der Arithmetischen Codierung:

$$code \coloneqq \Big(code - i_u\big(z(h)\big)\Big) \cdot \frac{n}{i_o\big(z(h)\big) - i_u\big(z(h)\big)}\ \text{S. } 325\ [1]\ 3$$

Dazu wird zunächst die Codierung einem Teilintervall aus der Tabelle zugeordnet. Dadurch ist das erste Zeichen decodiert. Bei den folgenden Zeichen wird jeweils die untere Grenze des Zeichenintervalls des vorherigen Zeichens, vom aktuellen Code abgezogen. Anschließend wird mit der Inversen des Anteils p, welcher für die Codierung verwendet wurde, multipliziert. Mit dem Resultat lässt sich das nächste Zeichen mithilfe des Zeichenintervalls bestimmen. Diese Prozedur wird nun so lange wiederholt, bis ein Zeichen decodiert wird, welches das Ende der Zeichenkette kennzeichnet.

2.1.4 LZ77-Verfahren

Das LZ77-Verfahren ist das Verfahren, aus dem die heutigen gängigen Zip-Formate entstanden sind. Dieses Verfahren wurde von den Herren Lempel und Ziv geprägt.

Olaf Manz beschreibt in seinem Buch „Gut gepackt – Kein Bit zu viel“ [3] das LZ77-Verfahren. Anhand dieser Quelle wird das Verfahren kurz vorgestellt. Die LZ77-Codierung zählt zu den Wörterbuchcodierungen. Ziel ist es, gleiche aufeinanderfolgende Zeichenketten durch eine Darstellung mit weniger Speicherbedarf darzustellen. Dazu wird ein Wörterbuch verwendet. Dieses Wörterbuch wird konstruiert, während der Text eingelesen wird. Das Ein-

lesen des Textes funktioniert mit dem *Sliding Window* (engl. „Schiebefenster“). Dieses Fenster hat eine zuvor definierte Größe und besteht aus zwei Teilen. Der erste Teil besteht aus dem Vorschaupuffer, der zweite Teil aus dem Wörterbuchpuffer. Im Vorschaupuffer stehen nun die ersten Zeichen der Datei, welche codiert werden sollen. Im Wörterbuchpuffer wird abgeglichen, ob diese Zeichen bzw. Zeichenketten schon einmal aufgetreten sind. Anschließend wird die Position im Wörterbuchpuffer, die Länge der gleichen aufeinanderfolgenden Zeichen und das erste Zeichen, welches nicht mehr übereinstimmt, abgespeichert. Die dadurch entstandenen Triple stehen dann nacheinander in der codierten Datei. Das *Sliding Window* bewegt sich um die Länge der übereingestimmten Zeichenkette plus eins (um das nächste Zeichen einzulesen) durch die Datei. Die Abbildung 2 zeigt ein Bespiel für die LZ77-Codierung. Hier wird der Vorschaupuffer mit fünf Zeichen und der Wörterbuchpuffer mit 12 Zeichen initialisiert. Die Abkürzung EOT in der vorletzten Zeile bedeutet übersetzt „Ende des Textes“ und ist ein im Vorhinein definiertes Zeichen.

Da bei kleinen Texten oder Texten mit wenig gleichen Zeichen bzw. Zeichenketten manchmal auch die vorherige Darstellung weniger Speicherplatz braucht, haben James Storer und Thomas Szymanski die LZSS-Codierung erfunden. Dabei wird die eigentliche Darstellung beibehalten und nur in dem Fall durch LZ77 Muster codiert, wenn eine Einsparung der Daten zu erwarten ist. Allerdings müssen bei diesem Verfahren die einzelnen Zeichen zusätzlich mit einer *Flag* (einem zusätzlichen Bit) versehen werden, um zwischen den Darstellungsformen unterscheiden zu können. Die Abbildung 2 zeigt neben der LZ77-Codierung auch eine LZSS-Codierung.

Ausgabe Text	Wörterbuch 12	11	10	9	8	7	6	5	4	3	2	1	Vorschau 5 Zeichen	Eingabe Text	Ausgabe LZ77	Ausgabe LZSS
													A N A N A	S B A N A N E N E I S	(0,0,A)	A
												A	N A N A S	B A N A N E N E I S	(0,0,N)	N
											A	N	A N A S B	A N A N E N E I S	(2,2,A)	(2,2,A)
								A	N	A	N	A	S B A N A	N E N E I S	(0,0,S)	S
							A	N	A	N	A	S	B A N A N	E N E I S	(0,0,B)	B
						A	N	A	N	A	S	B	A N A N E	N E I S	(7,4,E)	(7,4,E)
	A	N	A	N	A	S	B	A	N	A	N	E	N E I S		(2,2,I)	(2,2,I
A N A	N	A	S	B	A	N	A	N	E	N	E	I	S		(10,1,EOT)	S
A N A N	A	S	B	A	N	A	N	E	N	E	I	S			fertig	

Abbildung 2 Bespiel für LZ77- und LZSS-Codierung [3, S.32]

Die Decodierung des Textes funktioniert analog hierzu. Dabei wird das Wörterbuch nicht zusätzlich in der Datei gespeichert, sondern beim Decodieren der Datei angelegt. Eine Abbildung zur Decodierung sieht somit auch analog zu Abbildung 2 aus. Der Unterschied hierbei ist, dass die beiden Spalten für die Ausgabe beim Codieren die Eingabe für das Decodieren sind. Somit benötigt das Decodieren auch keinen Vorschaupuffer.

2.2 Moderne Kompressionsverfahren

Im Folgenden werden die vier modernen Kompressionsverfahren vorgestellt, die für die Testreihe dieser Arbeit ausgewählt worden sind. Es handelt sich dabei um die Verfahren Lempel-Ziv-Markow-Chaim-Algorithmus (LZMA), Lempel-Ziv-4-Algorithmus (LZ4), Brotli-Algorithmus und das Zstandard-Verfahren. Diese Verfahren bauen auf den historischen Grundtechniken zur Datenreduktion auf.

2.2.1 Lempel-Ziv-Markov-Chain-Algorithmus (LZMA)

Aufbauend auf der LZ77-Codierung (2.1.4) und den in Kapitel 2.1.2 und 2.1.3 vorgestellten *Entropiecodierungen* gab es viele Verbesserungen, Kombinationen und Weiterentwicklungen. Eines der bekanntesten Derivate der genannten Verfahren, ist der Deflate-Algorithmus. Dieser kombiniert die in Kapitel 2.1.2 vorgestellte Huffman-Codierung und die in Kapitel 2.1.4 vorgestellte LZSS-Codierung. Dieses Verfahren wird aufgrund der Ähnlichkeiten der Prinzipien nicht näher erläutert. Für genauere Beschreibung wird auf die „Deflate Compressed Data Format Specification“ [4] verwiesen. Im Folgenden werden vier neuere Derivate kurz vorgestellt.

Der Lempel-Ziv-Markov-Chain-Algorithmus wurde zusätzlich zu den Herren Lempel und Ziv, nach Herrn Markov (Entwickler der Markov-Kette) benannt und im Wesentlichen von Igor Pavlov entwickelt. Der Nachfolger LZMA2 wird im gängigen 7-Zip Format verwendet. Der LZMA und im Anschluss der LZMA2 wird nun anhand der Erläuterung aus dem Buch „Handbook of data compression“ von Salomon David [5] kurz vorgestellt.

Zusätzlich zu der in Kapitel 2.1.4 vorgestellten LZ77-Codierung, welche durch ein Tripel realisiert wurde, benutzt der LZMA weitere sechs Codierungsformen. Die erste Codierungsform besteht in der direkten Codierung des Zeichens, wie bei der LZSS-Codierung (Kapitel 2.1.4). Die anderen fünf Codierungsformen entstehen aus der jeweiligen Distanz zum letzten übereinstimmenden Muster. Diese Codierungen werden mit unterschiedlichen *Bitpräfixen* (vorangestelltes Bit zur Unterscheidung der Bytefolgen) gekennzeichnet, da sie unterschiedlich lang sind. Die erste der fünf Codierung ist eine „Short-Repetition“. Dabei kommt das Muster erneut direkt vor. Hier muss nur der entsprechende *Bitpräfix* gespeichert werden. Bei den anderen vier Codierungen werden zusätzlich zu den *Bitpräfixen* die Distanzen zu den letzten vier Mustern abgespeichert. Diese zusätzlichen Codierungen werden gebraucht, da das Wörterbuch etwa 1000mal größer ist als bei der LZ77-Codierung. Die Größe des Wörterbuches führt zusätzlich dazu, dass je nach Implementierung verkettete Listen oder Binärbäume für bessere Zugriffszeiten genutzt werden. Bei dem Durchlauf durch die einzelnen Zeichen

wird zunächst die Bytefolge *gehasht.* Das bedeutet, dass die Bytefolge einen Index für die Position im Feld bekommt, die durch eine Hashfunktion bestimmt wurde. Dort wird die Bytefolge in einem Feld abgespeichert. Die Größe dieses Feldes stimmt mit dem Wertebereich der Hashfunktion überein. In diesem Feld sind nun entweder verkettete Listen oder Binärbäume gespeichert. Bei der verketteten Liste werden die neu gefundenen Muster vorne eingefügt und bei den Binärbäumen an die Stelle der Wurzel gepackt. Dies hat bei den Binärbäumen eine Rotation zur Folge. Gesetzt den Fall, dass die Liste oder der Baum nicht leer sind, wird die jeweilige Datenstruktur durchlaufen und bis zu einer bestimmten Tiefe nach einer Teilüberstimmung des Musters gesucht. Dabei wird die Bytefolge entweder passend codiert oder abgespeichert. Ein weiterer Aspekt des Algorithmus ist die Bereichscodierung. Diese lässt sich mit der in Kapitel 2.1.3 vorgestellten Arithmetischen Codierung vergleichen, basiert allerdings auf dem Datentyp *integer*. Die Bereichscodierung wird auf die oben genannten Bytefolgen angewandt, um hier zusätzlich Speicherplatz einzusparen.

Die Verbesserungen, die Pavlov 2009 in seinem LZMA2 veröffentlichte, belaufen sich im Wesentlichen darauf, dass die zu codierende Datei in Blöcke aufgeteilt wird [6]. Diese einzelnen Blöcke werden nun mit dem LZMA codiert. Anschließend wird geprüft, ob der Block kleiner ist als der nicht codierte Block, und es wird der kleinere von beiden abgespeichert. Die Blöcke ermöglichen es zudem, dass der Algorithmus auf mehreren *Threads* (Ausführungssträngen) parallel laufen kann. Das wiederum führt zu einer schnelleren Performanz.

2.2.2 Lempel-Ziv-4 (LZ4)

Ein weiteres Derivat der in Kapitel 2.1.4 vorgestellten LZ77-Codierung ist der LZ4 Algorithmus. Dieser wurde von Yann Collet im Jahre 2011 veröffentlicht und wird anhand einer Dokumentation vorgestellt [7].

Das Format des LZ4 Algorithmus, in welchem die *Literale* (das ist die direkte Darstellung eines Wertes) abgespeichert werden, besteht aus fünf Blöcken. Der erste Block ist der *Token*, der aus zwei Teile besteht. Im ersten Teil wird die Länge für den zweiten Block gespeichert und im zweiten Teil wird die Länge für den letzten Block gespeichert. Der *Token* ist genau ein Byte groß. Das bedeutet, dass für die Codierung der Längen jeweils vier Bit zur Verfügung stehen. Dies würde bedeuteten, dass die maximale Länge des zweiten und fünften Blocks jeweils 15 wäre, aufgrund der Tatsache, dass mit vier Bit maximal 16 Zeichen codiert werden können und die 0 auch eine mögliche Länge ist. Um größere Längen zu realisieren ist demnach die Regel, dass wenn das erste Byte des jeweiligen Blocks 255 beträgt, ein weiteres Byte dem entsprechenden Block zugeordnet wird. Die 255 ist hier die maximale

Anzahl, da mit einem Byte maximal 256 Zeichen codiert werden können ($2^8 = 256$) und die 0 wieder eine mögliche Zahl ist. Diese Prozedur würde sich beim zweiten Byte fortsetzen, wenn dieses ebenfalls den Wert 255 enthält. Damit ergibt sich eine variable Länge des zweiten und fünften Blocks.

In dem ersten dieser variablen Blöcke (dem insgesamt zweiten Block) steht die Länge des Bytecodes für das im dritten Block abgespeicherten *Literal*. Der dritte Block ist somit ebenfalls variabel aufgebaut. Die *Literale* werden jeweils in ihrer Byte Darstellung kopiert, somit ist die Länge ein Vielfaches von Bytes. Der darauffolgende vierte Block ist der *Offset*. Hier wird die Distanz des aktuell betrachteten *Literals*, (in der Datei, welche codiert wird) zum übereinstimmenden *Literal* (welches bereits in der codierten Datei abgespeichert ist) gespeichert. Dieser Block ist auf eine Größe von zwei Bytes festgelegt. Das bedeutet, dass auf maximal $2^{16} - 1 = 65.535$ Byte vorher zugegriffen werden kann. Zudem wird der Offset in der *little-endian* Schreibweise abgespeichert. Dies bedeutet, dass das Byte, welches eine kleinere Wertigkeit hat, zuerst abgespeichert wird. Der letzte Block ist der variable Block, welcher die Länge des übereinstimmenden *Literals* beinhaltet. Dieses wird auch Match genannt.
Abhängig von der jeweiligen Implementierung werden nun verschiedenste Datenstrukturen genutzt, um die zu codierenden Dateien möglichst schnell in das oben beschriebene Format, bzw. in das Ursprungsformat zu bringen. Die Version, welche auf Google Codes zur Verfügung steht, benutzt wie der in Kapitel 2.1.4 vorgestellte LZMA, Hashtabellen.

Im Allgemeinen gibt es zwei unterschiedliche Formen des LZ4 Algorithmus. Zum einen die Form, welche die schnelle Kompression als Ziel hat und zum anderen die Form, welche eine große Kompressionsrate anstrebt, also eine große Speicherplatzeinsparung. Der Unterschied liegt in der Auswahl des jeweiligen Matches. Bei der schnelleren Variante wird das Match genommen, welches zuerst in der entsprechenden Hashtabelle steht. Bei der Variante, welche eine größere Kompressionsrate zum Ziel hat, werden sich alle (bzw. bis zu einem gesetzten Limit) Eintragungen angeschaut, um eventuell ein besseres Match zu finden.

2.2.3 Brotli

Der Broli Algorithmus ist eine Weiterentwicklung des Deflate-Algorithmus, die von Google am 22.09.2015 [8] angekündigt wurde. Anders als die Vorgängerversion Zopfli ist dessen Dateiformat nicht mit dem Deflate Dateiformat kompatibel. Die Struktur des Dateiformats und die Kompression der Daten werden anhand der „Brotli Compressed Data Format“ [9] Spezifikation kurz vorgestellt.

Das Dateiformat beginnt mit einem *Header*, gefolgt von verschieden vielen und verschieden langen *Metablöcken*. Informationen über die Größe des genutzten gleitenden Fensters werden im Header abgespeichert. Die Metablöcke haben eine Größe von 0-16 MiB und unterteilen sich jeweils in einen Header und in komprimierte Daten. In dem Header der Metablöcke sind Informationen über die jeweilige Repräsentation der folgenden Daten enthalten.
Die komprimierten Daten werden als Kommandos abgespeichert. Diese Kommandos gliedern sich in die Darstellung der Zeichen und in Zeiger. Diese Zeiger zeigen auf Zeichen aus dem Wörterbuch. Für die Kompression der Daten wird eine Kombination aus der in Kapitel 2.1.4 vorgestellten LZ77-Codierung und der in Kapitel 2.1.2 vorgestellten Huffman-Codierung genutzt. Die Huffman-Codierung wird auf jeden Metablock separat angewendet. Das ist bei der hier angewendeten Form der LZ77-Codierung anders. Diese wird nicht auf die einzelnen Metablöcke angewandt, sondern auf ein gleitendes Fenster. Das gleitende Fenster ist 1000mal größer, als bei der ursprünglichen Codierung. Dadurch können duplizierte Daten häufiger durch eine Codierung ersetzt werden. Zusätzlich gibt es ein vordefiniertes statisches Wörterbuch, welches über 13.000 häufig verwendete Strings beinhaltet. Aus diesem String lassen sich Teilsätze, Wörter und Phrasen zusammenfassen. [10]

2.2.4 Zstandard

Zum Schluss wird nun das Zstandard Verfahren anhand der Dokumentation von Facebook kurz vorgestellt [11]. Der Zstandard wurde von Yann Collet entwickelt, während er bei Facebook angestellt war. Dieser Algorithmus hat sich zum Ziel gesetzt unabhängig von der gegeben Hardware gute Kompressionen zu erzielen.

Die komprimierte Datei ist durch sogenannte *Frames* gegliedert. Diese sind unabhängig voneinander und ermöglichen eine parallele Ausführung der Dekomprimierung. Ein *Frame* ist jeweils in vier unterschiedliche Arten von Blöcken unterteilt, wobei der letzte optional ist.
Der erste Block in einem *Frame* ist die *Magic-Number*, dies ist die jeweilige Kennung eines *Frames*. Diese ist wichtig, um beispielsweise bei einer parallelen Ausführung, die einzelnen Blöcke der richtigen Position zuzuordnen. Für diesen Block sind vier Byte an Speicherplatz vorgesehen. Zudem wird der Block im *little-Endian* Format abgespeichert. Daran anschließend im zweiten Block, dem *Frame-Header*, werden die Metadaten für die daran anschließenden Datenblöcke abgespeichert. Dieser Block hat eine variable Größe von 2 bis 14 Bytes. Die Größe ist abhängig von den nun folgenden Datenblöcken. Die Größe dieser Datenblöcke ist in dem vorherigen Block abgespeichert. Hier ist es zudem möglich, dass auch mehrere Datenblöcke abgespeichert werden. Der letzte Block ist die *Content-Checksum*. Dieser Block ist dazu da, um die Integrität der Daten (nach vollendeter Dekomprimierung) zu

überprüfen. Wie bereits erwähnt, ist dieser letzte Block optional und kann durch einen zusätzlichen Parameter bei der Kompression gesetzt werden. In diesem Fall hat der letzte Block eine Größe von vier Byte und wird ebenfalls im *little-Endian* Format abgespeichert.

Die zu kodierende Datei wird ähnlich wie bei der LZ77-Codierung (Kapitel 2.1.4) durchgegangen. Allerdings sind die hier verwendeten gleitenden Fenster um ein Tausendfaches größer als bei der LZ77-Codierung. Zudem werden die Literale aus der Datei mittels der in Kapitel 2.1.2 vorgestellten Huffman-Codierung zusätzlich codiert. Alle anderen Metadaten werden mithilfe der Methode der asymmetrischen Zahlensysteme (ANS) [12] codiert. Diese Methode nutzt die Kompressionsrate der in Kapitel 2.1.3 vorgestellten Arithmetischen Codierung und die Geschwindigkeit der Huffman-Codierung, um eine schnellere Entropiecodierung zu schaffen.

2.3 Darstellungsarten von Matrizen

Anschließend werden zwei verschiedene Darstellungsarten von Matrizen vorgestellt. Die Erläuterungen sind analog zu denen, aus dem Werk „Templates for the Solution of Linear Systems“ [13]. Diese Verfahren sind dafür geeignet dünnbesiedelte Matrizen, auch *Sparse* Matrizen genannt, effizienter abzuspeichern. Außerdem soll damit auch der Speicherbedarf der Matrix reduziert werden. Die Idee besteht darin, anstelle der gesamten Matrix nur die Positionen der wenigen *non-zero-values* („Nicht-Null-Einträge“) abzuspeichern.

2.3.1 Compressed Sparse Row (CSR)

Bei dem CSR oder auch *Compressed Row Storage* (CRS) genanntem Verfahren wird die Position der *non-zero-values* in zwei Feldern abgespeichert. In einem dritten Feld werden die von Null abweichenden Werte gespeichert. Das Feld für die Position gibt den Index der Spalte (in der ursprünglichen Matrix) an, in dem sich der Wert befindet. In dem anderen Feld werden Zeiger für die Reihen gespeichert. Hierbei wird jeweils der erste Index aus dem Feld für die Spalten abgespeichert, welcher in einer neuen Zeile steht.
Beispielsweise sind drei, von Null abweichende, Werte in der ersten Reihe einer *Sparse* Matrix abgespeichert. Diese drei Werte werden zunächst im Feld für die Werte abgespeichert. Dann werden die jeweiligen Indizes für die Spalten abgespeichert. Abschließend wird in dem Zeigerfeld für die Reihen zunächst die Eins bzw. die Null (je nachdem mit welchem Index begonnen wird zu zählen) für den ersten Wert abgespeichert. Anschließend wird die Vier bzw. die Drei abgespeichert. Dies wäre der Wert, bei dem eine neue Reihe beginnt.

2.3.2 Compressed Sparse Column (CSC)

Das CSC-Verfahren oder auch *Compressed Column Storage* (CCS) Verfahren genannt, funktioniert analog zum CSR-Verfahren. Der Unterschied besteht darin, dass sich die Felder für Spalten und Reihen tauschen. Somit werden die Indizes für die Reihen gespeichert. In dem zweiten Feld für die Position werden die Zeiger auf den Index gespeichert, mit dem eine neue Zeile beginnt.

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}$$

val	10	3	3	9	7	8	4	8	8 ··· 9	2	3	13	-1
row_ind	1	2	4	2	3	5	6	3	4 ··· 5	6	2	5	6

col_ptr	1	4	8	10	13	17	20

Abbildung 3 Compressed Column Storage (CCS) [13, S.57-58]

Die Abbildung 3 zeigt links eine dünnbesiedelte Matrix und rechts die Darstellung der Matrix im CSC- bzw. CCS-Format. Die Werte, die nicht N ull sind, werden im Feld *val* (für engl. Value, der Wert) gespeichert. Die Indizes für die Reihe und die Zeiger für die Reihe werden in *row_ind* (für engl. Row Indice, Reihenindize) bzw. *col_ptr* (für engl. Column Pointer, Zeiger auf Spalte) gespeichert. Da hier drei Werte in der ersten Spalte vorhanden sind, ist der zweite Wert in *col_ptr* vier, da beim vierten Wert die nächste Spalte beginnt.

3. Ausgangslage

Nachdem in Kapitel 2 die verschiedenen Verfahren zu Datenkompression vorgestellt wurden, wird nun das Material beschrieben, welches komprimiert werden soll. Das Material besteht aus Datensätzen, welche für ein KI-Modell als Trainingsdatensätze genutzt werden sollen. Dieses KI-Modell wurde aufgrund unzureichender Ergebnisse in der Hochwasservorhersage entwickelt, um diese Ereignisse zukünftig zuverlässiger und präziser vorhersagen zu können. Für das Training des KI-Modells wird eine große Anzahl von Trainingsdatensätzen benötigt, damit die KI Muster erkennen kann. Die Trainingsdatensätze wurden mit Hilfe von bestehenden hydromechanischen Modellen für unterschiedliche Standorte generiert. Dies wurde mit der Software Mike21 durchgeführt. Anschließend wurden diese Simulationsdaten in das NPY-Dateiformat überführt. Dieses wird nun kurz beschrieben.

3.1 Datenformat

Die Trainingsdatensätze liegen im NPY-Dateiformat vor. Dieses Dateiformat wird von der Python Bibliothek *NumPy* genutzt, was ein Akronym für *numeric Python* ist. Das NPY-Dateiformat speichert ein *NumPy-Array* (andere Darstellung eines Feldes). Die Daten werden in ihrer binären Darstellung gespeichert. Die Informationen über die Form und den enthaltenen Datentyp werden auch gespeichert. Somit kann das *Array* auch auf anderen System Architekturen genutzt werden. Dabei werden die Speicherarten *little-endian* und *big-endian* (Byte mit höherer Wertigkeit, wird zuerst abgespeichert) unterstützt. Der Speicherbedarf der einzelnen Datentypen wird gespeichert und nicht der Datentyp, welcher auf unterschiedlichen Systemen unterschiedliche Größen haben könnte. Die Unterklassen eines *NumPy-Arrays* werden akzeptiert, jedoch als *Array* Daten ausgeschrieben. Die Dateiendung des Formates ist *npy* [14].

Diese NPY-Dateien enthalten ein mehrdimensionales *Array*. In diesem *Array* werden Zeitreihen von zweidimensionalen Höhenrasterkoordinaten abgespeichert. Die erste Dimension des *Arrays* repräsentiert einen Zeitschritt der Zeitreihe. Die beiden folgenden Dimensionen beschreiben die x-, y-Koordinaten von Wasserständen. Das gesamte *Array* hat den Datentyp *float*. Die Einheit für die Wasserstände sind Meter.

3.2 Testdaten

Der erste Testdatensatz beschreibt das Hochwasser Ereignis in der Aachener Innenstadt im Mai 2018. Die Wasserstände wurden auf einem ungefähr 7 x 8 km^2 großen Gebiet simuliert. Dabei wurden insgesamt 24 Zeitschritte in einem Abstand von fünf Minuten generiert. Die

Messung hat somit zwei Stunden gedauert. Diese Wasserstände sind in dem in 3.1 beschriebene Dateiformat gespeichert. Der verwendete Datentyp ist hierbei ein *float16*. Jeder Quadratmeter benötigt ein Feld im *Array*. Dieser Quadratmeter braucht aufgrund des gewählten Datentyps zwei Byte an Speicher. Somit ergibt sich für den Datensatz ein Speicherplatzbedarf von ungefähr 2,6 Gigabyte.

Der zweite Datensatz beschreibt einen Staudammbruch in der Ukraine. Das diesem Datensatz zugehörige Gebiet beläuft sich auf ungefähr 3,6 x 7 km^2. Anders als beim Datensatz für die Stadt Aachen gibt es bei diesem Datensatz Positionen, für die kein Wert angegeben wurde. Hier wird dann der Wert -9.999 als ein so genannter *no-data-value* genutzt (Platzhalter, wenn keine Daten vorhanden sind). Die Simulationslänge variiert ebenfalls. Hier werden 48 Zeitschritte generiert. Diese haben einen Abstand von jeweils einer Stunde, somit werden insgesamt zwei Tage simuliert. Der Datentyp, welcher hier genutzt wird, ist ein *float64*. Dementsprechend ist die Datei, trotz kleinerem Gebiet, um ungefähr das Vierfache größer (9,22 GB).

Der dritte Datensatz betrifft ebenfalls die Stadt Aachen. Somit verändert sich das Gebiet nicht. Der Datensatz stellt die weitere Simulation eines Starkregenereignisses dar. Auch die Aufzeichnungsdauer, der Speicherbedarf und der Datentyp bleiben gleich.

3.3 Anforderungen

Im Folgenden werden die Anforderungen an die komprimierte Darstellung der Trainingsdatensätze beschrieben.

Die Datensätze sollen durch eine verlustfreie Kompression abgespeichert werden. Hier sollen keine Informationen verloren gehen, sodass das KI-Modell mit ungefilterten Daten rechnen kann. Aufgrund dessen dürfen auch keine Zeitschritte weggelassen werden. Zudem sollte die Zugriffgeschwindigkeit auf die Daten nicht mehr als wenige Sekunden in Anspruch nehmen, somit ist eine schnelle Dekompression essentiell. Die Genauigkeit der in den Datensätzen abgespeicherten Wasserstände, soll bis auf einen Millimeter genau sein. Ein Maximalwert von 10 Metern ist hierbei ausreichend. Um die *Interoperabilität* (Zusammenspiel verschiedener Systeme) zu gewährleisten, sollte die komprimierte Form eine Über- und eine Rückführung in das in Kapitel 3.1 vorgestellte NPY-Dateiformat beinhalten. Zudem sollten die Datensätze nach der Dekomprimierung in einen Fließkommadatentyp vorliegen.

3.4 Testumgebung

Anschließend wird die vorhandene Testumgebung beschrieben, welche genutzt wird, um die in Kapitel 2.2 vorgestellten moderneren Kompressionsverfahren zu vergleichen. Dazu wird als Erstes das verwendete System vorgestellt und anschließend die Software, welche zur Kompression genutzt wurde.

Die Tests wurden auf einem Computer mit dem Betriebssystem Windows 11 (in der 64-Bit Version) durchgeführt. Zudem wurde der Prozessor AMD Ryzen 7 5700G verwendet. Dieser Prozessor hat eine Basisgeschwindigkeit von 3,80 GHz und besitzt acht Kerne. Diese acht Kerne werden unter Windows auf 16 logische Prozessoren erweitert. Die Basisgeschwindigkeit kann bis auf eine maximale boost Geschwindigkeit von 4,60 GHz ansteigen. Des Weiteren ist ein 32 GB großer Arbeitsspeicher vorhanden. Dieser ist in zwei 16 GB Module aufgeteilt und gehört zu DDR4 Generation. Zudem hat dieser eine 3600 MHz Taktung. Außerdem ist eine SSD-Festplatte verbaut, welche auf dem M.2 PCIe-4.0 Steckplatz angebracht ist. Hiermit sollen Lesetransferraten von 5.6 GB/s möglich sein.

Als Softwarepaket wird eine Erweiterung für das 7-Zip Programm genutzt. Diese Erweiterung ist auf Github zu finden [17]. Zusätzlich zu den in 7-Zip vorhandenen Algorithmen zur verlustlosen Datenkompression wurden durch die Erweiterung die Algorithmen Brotli, Fast-LZMA2, Lizard, LZ4, LZ5 und Zstandard hinzugefügt. Anstelle des in Kapitel 2.2.1 vorgestellten LZMA, werden die Derivate LZMA2 und Fast-LZMA2 genutzt. Der Fast-LZMA2 ist laut des Github Repository eine schnellere Version des LZMA2. Allerdings wird die Kompressionsrate durch die gewonnene Schnelligkeit schlechter. Der Unterschied zum LZMA2 ist die Verwendung eines *radix matchfinder* (nutzt einen Speicher optimierten Baum zur Mustersuche) und es wurden einige Optimierungen des Zstandard hinzugefügt.

4. Methodik

Nun werden die Trainingsdatensätze hinsichtlich ihres Einsparpotentials untersucht. Darauf aufbauend wird untersucht, inwiefern eine geänderte Darstellungsform (2.3) eine Kompression bewirken kann. Anschließend wird der Testvorgang für die in Kapitel 2.2 vorgestellten Kompressionsverfahren LZMA2, LZ4, Brotli und Zstandard, sowie der Derivate Fast-LZMA2 und LZ5 im Hinblick auf ihre Kompressionsrate und der anschließend benötigten Dekompressionszeit beschrieben.

4.1 Strukturanalyse der Daten

Um das Einsparpotential der Daten zu ermitteln, wird zunächst die Struktur der Daten analysiert. Dazu werden zuerst die Datentypen in 4.1.1 der jeweiligen Testdatensätze bestimmt und auf das Einsparpotential hin untersucht. Anschließend wird die Anzahl der *non-zero-values* (von Null abweichende Werte) in 4.1.2 herausgefunden und ebenfalls auf ihr Einsparpotential hin analysiert.

4.1.1 Datentypen

Die Aachener Testdatensätze werden, wie in Kapitel 3.2 beschrieben, in einem dreidimensionalen Feld mit dem *float16* Datentyp abgespeichert. Die Abfolge von Einsen und Nullen ist bei Fließkommazahlen durch die Berechnung der Mantisse, des Exponenten und dem Vorzeichenbit nicht so linear ansteigend, wie beim *integer* Datentyp. Aufgrund dessen sind mit einem *integer* Datentyp bessere Kompressionen zu erwarten. Diese Überlegungen beruhen auf den Aussagen von F.Ghido in seinem Paper „An efficient algorithm for lossless compression of IEEE float audio“ [15]. Er erhielt bei seinem Anwendungsfall eine um sieben Prozent bessere Kompression.

Nun wird geprüft, ob eine Umstellung des Datentyps von einem *float16* auf einen *integer* den in Kapitel 3.3 erläuterten Anforderung entspricht. Durch den Wechsel gäbe es einen Informationsverlust im Hinblick auf die Fließkommadarstellung, jedoch kann mit einer Skalierung durch den Faktor 1000 die geforderte Genauigkeit von einem Millimeter gewährleistet werden. Eine Gefahr für einen Überlauf ist nicht abzusehen, da als obere Grenze 10 Meter genannt worden sind. Dies entspräche mit dem Skalierungsfaktor einer Grenze von 10 Tausend. Diese Grenze wird nicht erreicht, da ein *integer* Datentyp ohne Vorzeichen mit 16 Bit $2^{16} = 65.536$ Ganzzahlen darstellen kann. Somit könnten also Wasserstände von ca. 65 Metern erfasst werden. Negative Wasserstände werden ebenfalls nicht gespeichert, daher kann hier ein *integer* ohne Vorzeichen genutzt werden. Der entsprechende Datentyp wäre bei der Bibliothek *NumPy* ein *unsigned short* [16].

Anschließend wird geprüft, ob die zusätzlich benötigte Zeit zur Skalierung des Datentyps und zur Rückführung in den ursprünglichen Datentype *float16*, den Anforderungen an die Dekompressionszeit entspricht.

Es folgt die Analyse des Datentyps des Testdatensatzes von dem Staudammbruch in der Ukraine. Die Werte bei diesem Datensatz sind mit dem *float64* Datentyp abgespeichert. Das bietet ein großes Einsparpotential aufgrund des (im vorherigen Absatz erläuterten) geforderten Wertebereichs. Allerdings ist eine Skalierung, um den Datentyp auf einen *integer* Datentyp umzuwandeln, wesentlich komplexer. An dieser Stelle müsste eine Lösung für den Umgang mit dem *no-data-value* konstruiert werden. Ein Wechsel auf den Datentyp *float16* hat aufgrund der Genauigkeit einen Datenverlust zur Folge. Hier wird nur die dritte Nachkommastelle garantiert, der *no-data-value* hat allerdings vier Nachkommastellen. Eine Umstellung auf den *float32* Datentyp ist möglich. Dabei wird eine Genauigkeit bis auf die siebte Nachkommastelle gewährleistet. Somit würde in dem Wertebereich kein Verlust stattfinden und der Speicherbedarf für den Datensatz würde sich halbieren.

Hierfür muss kein Test durchgeführt werden, da hier keine Rückführung nötig ist. Es wird allerdings getestet, ob eine Umwandlung des Datentyps Performance Vorteile hat.

4.1.2 Non-zero-values

Da in den Trainingsdatensätzen Hochwasserstände gespeichert werden ist anzunehmen, dass viele der Einträge der Matrix den Wert Null haben. Dies wird repräsentativ an den drei vorliegenden Testdatensätzen geprüft.

Zunächst wird der Testdatensatz von dem Staudammbruch in der Ukraine untersucht. Von diesem Testdatensatz sind ungefähr 3,1 Prozent der gesamten Einträge der Matrix *non-zero-values* (von Null abweichende Werte). Dabei ist zu berücksichtigen, dass ungefähr 21,5 Prozent dieser Werte *no-data-values* sind. Damit gibt es viele gleiche Werte in diesem Datensatz. Dieser Trainingsdatensatz hat somit ein großes Einsparpotential von benötigtem Speicherplatz.

In den Datensätzen der Stadt Aachen haben ungefähr ein Drittel der Elemente den Wert Null. Anfangs variiert der Anteil noch abhängig vom Datentyp. Dies wird auf der nächsten Seite in Abbildung 4 veranschaulicht. Diese Abbildung zeigt die *non-zero-Values* zum einen für den Datentyp *float* und zum anderen für den Datentyp *integer* bzw. *unsigned short*. In der Grafik werden alle 24 Zeitschritte von einem der Aachen Datensätze abgebildet. Bei Beginn

der Aufzeichnung ist zu erkennen, dass sehr niedrige Wasserstände bei der *integer* Darstellung nicht mehr abgespeichert werden. Hier findet, aufgrund von Informationsverlust, eine Datenreduktion statt. Dieser ist tolerierbar, da dieser im geforderten Wertebereich geschieht.

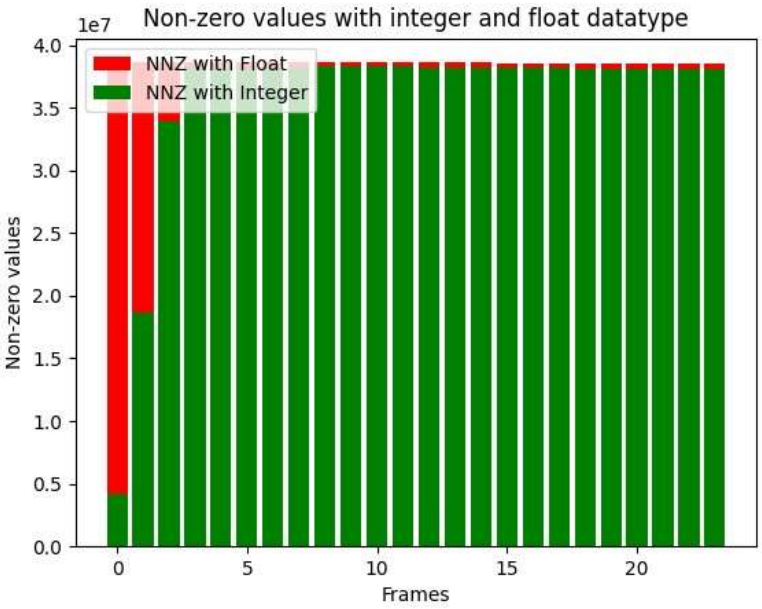


Abbildung 4 Von Null abweichende Werte mit integer und float Datentyp

4.1.3 Darstellungsformen

Das Einsparpotential der vielen non-zero-values des Ukraine Datensatz, kann beispielsweise durch die Änderung der Darstellungsform in eine der in Kapitel 2.3 vorgestellten Darstellungsformen genutzt werden. Der Speicherbedarf für die Darstellung eines Zeitschrittes beträgt in der derzeitigen Darstellung $O(n \cdot m)$. Dabei steht n für die Zeilen und m für die Spalten. Bei der *O-Notation* (hier als Maß für den Speicherbedarf) wird hier das Multiplizieren mit der Größe des Datentyps zum einfachen Vergleich mit den anderen Darstellungen weggelassen. Der Speicherbedarf für das in Kapitel 2.3.1 vorgestellte CSR-Format lässt sich mit $O(2nnz + n + 1)$ [13] beschreiben. Dabei steht *nnz* für die *non-zero-values* der Matrix. Hier benötigt es zweimal den Speicher für die *non-zero-values*, da diese zum einen im Feld für die Werte stehen und zum anderen im Feld für die Spaltenindizes abgespeichert werden. Zusätzlich benötigt es noch Speicher für das Feld, welches den Zeiger auf die Zeilen speichert. Dieser Speicher wird mit n + 1 angegeben.

Die Abbildung 4 aus Kapitel 4.1.2 zeigt, dass es für die Aachener Trainingsdatensätze durchschnittlich ungefähr $3{,}7 \cdot 10^7$ *non-zero-values* gibt. Dies macht ungefähr zwei Drittel der Gesamten Werte in der Matrix aus. Mit dem Speicherbedarf von $2 \cdot nnz$ wird somit durch

die vorgestellten Darstellungsformen aus Kapitel 2.3 noch mehr Speicher benötigt, als mit der aktuellen Darstellungsform.

Anders sieht das bei dem Trainingsdatensatz aus der Ukraine aus. Hier gibt es nur ungefähr 3,1 Prozent *non-zero-values*. Nun wird der Speicherbedarf grob mit $3 \cdot nnz$ überschlagen. Damit ergibt sich ein Speicherbedarf von knapp über 9 Prozent. Dieser Datensatz könnte somit durch ein Transferieren der Darstellungsform, seinen Speicherbedarf auf ein Zehntel reduzieren.

4.3 Testen der Verfahren

Die verschiedenen Verfahren werden unmittelbar auf die Testdatensätze angewandt. Dadurch soll getestet werden, ob eine allgemeine Empfehlung für die Trainingsdatensätze ausgesprochen werden kann. Dabei werden die verschiedenen Level der Verfahren getestet. Das Ergebnis dieser Level unterscheidet sich in der Laufzeit und in der Kompressionsrate. Die Veränderungen der Ergebnisse entstehen durch die Änderung verschiedener Parameter.

Die konfigurierbaren Parameter sind zum einen die Wörterbuchgröße, die Wortgröße und die Speichernutzung und zum anderen, für die parallele Ausführung, die Größe der soliden Blöcke und die Anzahl der *CPU-Threads*. Desto größer die ersten genannten Parameter gewählt werden, desto größer fällt die Kompressionsrate aus. Das hat eine höhere Speichernutzung und eine langsamere Ausführung zur Folge. Die zweiten genannten Parameter beschleunigen die Ausführung, je größer diese werden.

Um die Auswirkungen zu testen, welche die konfigurierbaren Parameter auf die Trainingsdatensätze haben, werden für die Testdatensätzen die unterschiedlichen Level durchlaufen. Dadurch soll eine breite Variation der Parameter getestet werden. Diese Parameter lassen sich auch individuell einstellen. Dabei gibt es allerdings zu viele verschiedene Einstellungen, um diese im Umfang dieser Arbeit testen zu können. Daher wird eine repräsentative Testung der unterschiedlichen Level der Algorithmen vorgenommen

Anschließend werden die Verfahren mit den Ergebnissen aus der Strukturanalyse kombiniert. Dazu wird der Datentyp der Aachener Datensätze in ein *integer* geändert und anschließend mit dem Vergleichssieger aus dem vorherigen Test komprimiert.
Bei dem Ukraine Datensatz wird zuerst der Datentyp geändert, dann die Darstellungsform und abschließend wird der Datensatz mit dem Vergleichssieger aus dem vorherigen Test komprimiert.

5. Ergebnisse

Die Abbildung 5 zeigt die Ergebnisse der Kompression am Testdatensatz der Ukraine Simulation. Aufgrund der weit außerhalb des gezeigten Wertebereichs liegenden Ergebnisse wurden die Verfahren LZ4 und LZ5 nicht in der Abbildung aufgeführt.

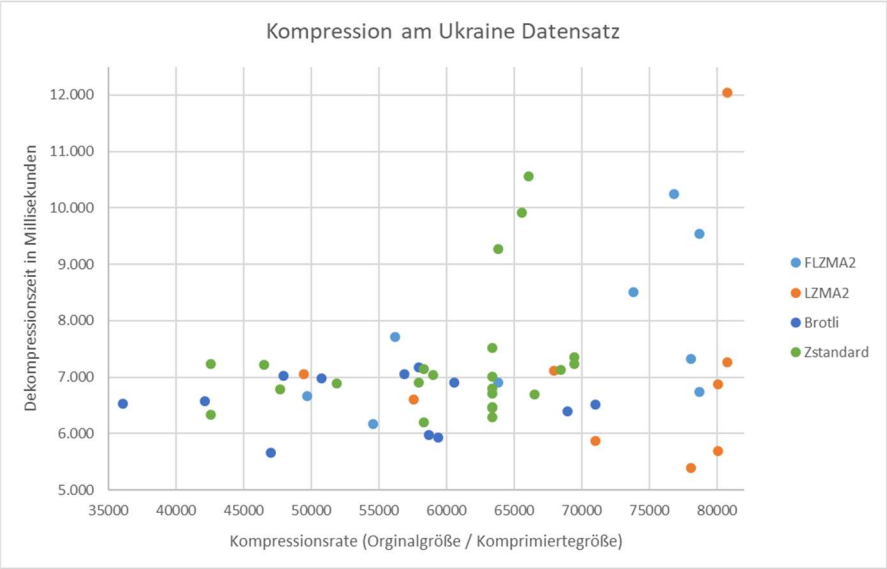


Abbildung 5 Kompression am Ukraine Datensatz

Dargestellt sind die Dekompressionszeit und die Kompressionsrate der vier Verfahren, welche aufgrund der Anforderungen am effizientesten sind. Die einzelnen Punkte sind die verschiedenen Level. Die Level sind von links nach rechts aufsteigend, mit ansteigender Kompressionsrate angeordnet. Bei diesem Datensatz ist das Level fünf des LZMA2 am effizientesten. Für die beiden Aachener Datensätze ist das Level vier des Fast-LZMA2 am effizientesten. Die Messergebnisse dazu sind im Anhang zu finden.

Beim anschließenden Test für den Ukraine Datensatz ist bei der Verwendung des CSR-Formats und der Umwandlung des Datentyps, sowie einer anschließenden Kompression mit dem LZMA2 eine um den Faktor vier größere Kompression erreicht worden. Dabei wurde die Dekompressionszeit für die Rückführung um weniger als zwei Sekunden erhöht.

Bei den Aachener Datensätzen ist mit der Umwandlung des Datentyps ebenfalls eine Verbesserung der Kompression um den Faktor vier erreicht worden. Allerdings kommt es durch die Umwandlung von *integer* zu *float* und die Skalierung der Daten zu einem Zuwachs der Dekompressionszeit von über vier Sekunden.

6. Diskussion

Bei der Anwendung der verschiedenen Kompressionsverfahren auf die Datensätze konnten die beiden Derivate des LZMA (LZMA2 und Fast-LZMA2) die schnellsten Dekompressionszeiten vorweisen und somit die Nutzbarkeit der Daten erhalten. Die Performance dieser Kompressionsverfahren zeigt sich allerdings unterschiedlich je nach untersuchtem Testdatensatz. Das bedeutet, dass keine allgemeingültige optimale Lösung für alle Testdatensätze gefunden werden konnte. Allerdings liefert das jeweils langsamere Verfahren ebenfalls gute Ergebnisse (siehe Anhang, Tabelle 1 und 3). Hier wäre die allgemeine Empfehlung Level fünf des LZMA2 zu wählen. Dieses Level hatte bei dem Ukraine Datensatz die schnellste Dekompressionszeit. Da der Ukraine Datensatz grundsätzlich langsamere Dekompressionszeiten vorweist, wurde hier das Level gewählt, mit dem die schnellsten Ergebnisse erzielt wurden. Zudem führt dieses Level bei den Aachener Datensätzen, bei geringfügig langsamerer Dekompressionszeit, zu einer größeren Kompressionsrate als das Level vier des Fast-LZMA2 (schnellste Dekompressionszeit für die Aachener Datensätze).

Des Weiteren wurde festgestellt, dass ein Datentypwechsel für die Aachener Datensätze (*float* zu *integer*) zu einer größeren Kompressionsrate (um den Faktor 4) führt. Allerdings führt dieser Datentypwechsel (mit anschließender Skalierung) zu einer Erhöhung der Dekompressionszeit (ungefähr 4,5 Sekunden). Das erfüllt die Anforderung in Bezug auf eine schnelle Wiederherstellungszeit nicht mehr. Hier wird die Dekompressionszeit um ungefähr 125 Prozent erhöht und somit wird dieser Wechsel nicht empfohlen.
Der Datentypwechsel für den Datensatz der Ukraine (*float64* zu *float32*) führt zu einer Halbierung des Speicherbedarfs. Die hierfür benötigte zusätzliche Zeit erhöht nur die Kompressionszeit. Eine Rückführung in den Datentyp *float64* kann nicht die gleichen Daten wie vor der Umwandlung garantieren. Hier gibt es einen Verlust der Genauigkeit der Simulationsergebnisse, da diese nur noch bis auf die siebte Nachkommastelle genau sind. Das liegt jedoch im geforderten Wertebereich und wird somit empfohlen.

Die Änderung der Darstellungsform der in den Daten gespeicherten Matrizen erzielte bei dem Ukraine Datensatz eine Reduktion des Speicherbedarfs um ungefähr 95 Prozent. In Kombination mit dem Datentypwechsel (*float64* zu *float32*) und einer anschließenden Kompression durch den LZMA2 werden bessere Laufzeiten bei einer höheren Kompressionsrate als durch die alleinige Anwendung des LZMA2 erreicht. Dieses Vorgehen ist somit zu empfehlen und wäre der einfachen Nutzung des LZMA2 vorzuziehen.
Anders sieht das bei der Änderung der Darstellungsform für die Aachener Datensätze aus. Hier steigt der Speicherbedarf an und es wird somit nicht empfohlen.

7. Fazit und Ausblick

Insgesamt wird in der Diskussion (6) deutlich, dass keines der geprüften Verfahren uneingeschränkt zur Kompression von Datensätzen unter den gegebenen Anforderungen empfohlen werden kann. Dies wird an den unterschiedlichen Ergebnissen für die drei Testdatensätze verdeutlicht. Das betrifft zum einen den Datentypwechsel. Dieser ist für die Aachener Datensätze und für den Ukraine Datensatz jeweils ein anderer. Zum anderen führt nur bei dem Ukraine Datensatz die Änderung der Darstellungsform zu einer Kompression der Daten. Bei den Aachener Datensätzen wird der Speicherbedarf sogar noch größer.
Um ein allgemeingültiges Ergebnis für alle Trainingsdatensätze zu finden, müssen Abstriche bei einer Art der Datensätze gemacht werden. Diese werden bei der Dekompressionszeit der Aachener Datensätze gemacht, da diese eine generell schnelle Dekomprimierung vorweisen. Das allgemeingültige empfohlene Kompressionsverfahren ist somit das fünfte Level des LZMA2. Dieses weist die schnellste Dekompressionszeit für den Ukraine Datensatz und eine vertretbare Dekompressionszeit für die Aachener Datensätzen auf (bei höherer Kompressionsrate, als bei dem Verfahren mit der schnellsten Dekompressionszeit für die Aachener Datensätze).

Für eine weitere Steigerung der Effizienz des Kompressionsverfahrens bietet es sich an eine effiziente Analyse des Datensatzes vor der Kompression einzuführen. Durch diese Analyse im Kompressionsschritt sollte es möglich sein, einen Wechsel des Datentyps oder eine Änderung der Darstellungsform entsprechend durchzuführen. Dies hätte zur Folge, dass der Speicherbedarf im Vorhinein stark reduziert und anschließend mit den in Kapitel 2.2 beschriebenen Verfahren komprimiert werden könnte. Die Herausforderung dabei ist es, die Dekompressionsgeschwindigkeit zu erhalten bzw. in einem vertretbaren Bereich zu erhöhen.

Abbildungsverzeichnis

Literaturverzeichnis

[1] W. Dankmeier, *Grundkurs Codierung*. Wiesbaden: Springer Fachmedien, 2017. doi: 10.1007/978-3-8348-2154-6.

[2] B. Küppers, „Codierung: Bits, Bytes und Bananen“, in *Einführung in die Informatik: Theoretische und praktische Grundlagen*, B. Küppers, Hrsg., in Studienbücher Informatik. , Wiesbaden: Springer Fachmedien, 2022, S. 3–30. doi: 10.1007/978-3-658-37838-7_1.

[3] O. Manz, *Gut gepackt – Kein Bit zu viel: Kompression digitaler Daten verständlich erklärt*. in essentials. Wiesbaden: Springer Fachmedien, 2020. doi: 10.1007/978-3-658-31216-9.

[4] L. P. Deutsch, „DEFLATE Compressed Data Format Specification version 1.3“, Internet Engineering Task Force, Request for Comments RFC 1951, Mai 1996. doi: 10.17487/RFC1951.

[5] D. Salomon, G. Motta, und D. Bryant, *Handbook of data compression*, 5. ed. London Heidelberg: Springer, 2010.

[6] A. Butting, „Der Lempel-Ziv-Markov Chain Algorithmus“.

[7] Cyan, „RealTime Data Compression: LZ4 explained“, RealTime Data Compression. Zugegriffen: 6. Januar 2024. [Online]. Verfügbar unter: https://fastcompression.blogspot.com/2011/05/lz4-explained.html

[8] „Introducing Brotli: a new compression algorithm for the internet“, Google Open Source Blog. Zugegriffen: 27. November 2023. [Online]. Verfügbar unter: https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html

[9] J. Alakuijala und Z. Szabadka, „Brotli Compressed Data Format“, Internet Engineering Task Force, Request for Comments RFC 7932, Juli 2016. doi: 10.17487/RFC7932.

[10] Z. Syed und T. Soomro, „Compression Algorithms: Brotli, Gzip and Zopfli Perspective“, *Indian J. Sci. Technol.*, Bd. 11, S. 1–4, Dez. 2018, doi: 10.17485/ijst/2018/v11i45/117921.

[11] „zstd/doc/zstd_compression_format.md at master · facebook/zstd“, GitHub. Zugegriffen: 7. Januar 2024. [Online]. Verfügbar unter: https://github.com/facebook/zstd/blob/master/doc/zstd_compression_format.md

[12] J. Duda, „Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding“. arXiv, 6. Januar 2014. doi: 10.48550/arXiv.1311.2540.

[13] R. Barrett *u. a.*, „Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods“.

[14] „numpy.lib.format — NumPy v2.0.dev0 Manual“. Zugegriffen: 16. November 2023. [Online]. Verfügbar unter: https://numpy.org/devdocs/reference/generated/numpy.lib.format.html

[15] F. Ghido, „An efficient algorithm for lossless compression of IEEE float audio“, in *Data Compression Conference, 2004. Proceedings. DCC 2004*, März 2004, S. 429–438. doi: 10.1109/DCC.2004.1281488.

[16] „Scalars — NumPy v1.26 Manual“. Zugegriffen: 9. Dezember 2023. [Online]. Verfügbar unter: https://numpy.org/doc/stable/reference/arrays.scalars.html#numpy.ushort

[17] T. Reichardt, „README“. 7. Januar 2024. Zugegriffen: 7. Januar 2024. [Online]. Verfügbar unter: https://github.com/mcmilk/7-Zip-zstd

Anhang

	schnellste D
	beste R
	schnellste K

Tabelle 1 für den ersten Aachener Datensatz:

Algo	Level	Kom_Z	Dekom_Z	Size_K_MiB	Real_S	Ratio
FLZMA2	1	12.493	1.952	267	2.652.246	9933,506
	2	10.725	1.588	228	2.652.246	11632,66
	3	13.475	1.588	216	2.652.246	12278,92
	4	12.538	1.566	198	2.652.246	13395,18
	5	20.990	1.723	188	2.652.246	14107,69
	6	26.230	1.886	186	2.652.246	14259,39
	7	30.564	2.243	184	2.652.246	14414,38
	8	36.845	2.265	182	2.652.246	14572,78
	9	43.108	2.248	181	2.652.246	14653,29
						0
LZMA2	1	8.703	1.820	433	2.652.246	6125,279
	2	8.618	2.482	243	2.652.246	10914,59
	3	10.135	1.591	228	2.652.246	11632,66
	4	10.921	1.727	225	2.652.246	11787,76
	5	34.555	1.716	191	2.652.246	13886,1
	6	39.691	1.932	190	2.652.246	13959,19
	7	53.228	1.933	189	2.652.246	14033,05
	8	65.737	2.351	189	2.652.246	14033,05
	9	65.971	2.413	189	2.652.246	14033,05
						0
LZ4	1	1.424	1.604	680	2.652.246	3900,362
	2	1.404	1.613	680	2.652.246	3900,362
	3	1.814	1.857	651	2.652.246	4074,111
	4	1.834	1.555	650	2.652.246	4080,378
	5	1.932	1.643	638	2.652.246	4157,125
	6	1.934	1.583	637	2.652.246	4163,651
	7	2.058	1.612	636	2.652.246	4170,198
	8	2.356	1.634	636	2.652.246	4170,198
	9	2.163	1.580	636	2.652.246	4170,198
	10	3.860	2.199	636	2.652.246	4170,198
	11	4.329	1.583	635	2.652.246	4176,765
	12	5.935	2.367	635	2.652.246	4176,765
						0
LZ5	1	1.422	1.611	782	2.652.246	3391,619
	2	1.400	1.634	674	2.652.246	3935,083
	3	1.483	2.258	657	2.652.246	4036,904
	4	1.519	1.617	627	2.652.246	4230,057
	5	1.536	1.591	626	2.652.246	4236,815
	6	2.129	1.582	616	2.652.246	4305,594
	7	2.526	1.594	615	2.652.246	4312,595
	8	6.007	1.618	615	2.652.246	4312,595

	9	7.714	1.565	613	2.652.246	4326,666
	10	7.834	1.609	613	2.652.246	4326,666
	11	8.638	1.987	613	2.652.246	4326,666
	12	12.357	2.228	606	2.652.246	4376,644
	13	14.281	1.583	610	2.652.246	4347,944
	14	14.843	2.288	604	2.652.246	4391,136
	15	57.741	1.685	611	2.652.246	4340,828
	16	59.693	1.678	611	2.652.246	4340,828
						0
Brotli	0	1.482	3.245	558	2.652.246	4753,129
	1	1.533	1.794	537	2.652.246	4939,006
	2	1.629	7.119	353	2.652.246	7513,445
	3	2.015	2.226	343	2.652.246	7732,496
	4	9.062	1.681	291	2.652.246	9114,247
	5	8.759	1.631	264	2.652.246	10046,39
	6	14.634	2.135	259	2.652.246	10240,33
	7	23.085	1.613	256	2.652.246	10360,34
	8	34.552	1.709	255	2.652.246	10400,96
	9	42.121	1.839	253	2.652.246	10483,19
	10	103.481	1.756	217	2.652.246	12222,33
	11	329.201	1.813	213	2.652.246	12451,86
						0
Zstandard	0	2.988	2.925	478	2.652.246	5548,632
	1	1.570	2.927	478	2.652.246	5548,632
	2	1.488	2.872	305	2.652.246	8695,889
	3	1.484	2.863	289	2.652.246	9177,322
	4	3.392	2.910	287	2.652.246	9241,275
	5	4.537	2.649	278	2.652.246	9540,453
	6	4.513	3.998	272	2.652.246	9750,904
	7	6.881	2.996	267	2.652.246	9933,506
	8	7.789	2.759	261	2.652.246	10161,86
	9	8.777	3.030	261	2.652.246	10161,86
	10	11.709	2.665	260	2.652.246	10200,95
	11	12.852	2.692	260	2.652.246	10200,95
	12	13.423	2.746	260	2.652.246	10200,95
	13	11.491	2.605	260	2.652.246	10200,95
	14	12.173	2.609	260	2.652.246	10200,95
	15	15.005	2.879	260	2.652.246	10200,95
	16	22.553	2.695	252	2.652.246	10524,79
	17	29.765	2.984	251	2.652.246	10566,72
	18	42.437	2.912	249	2.652.246	10651,59
	19	89.487	2.969	245	2.652.246	10825,49
	20	105.336	3.151	244	2.652.246	10869,86
	21	163.082	3.152	244	2.652.246	10869,86
	22	309.730	3.283	243	2.652.246	10914,59

Tabelle 2 für den Ukraine Datensatz:

Algo	Level	Kom_Z	Dekom_Z	Size_K_MiB	Real_S	Ratio
FLZMA2	1	20.857	6.658	190	9.446.144	49716,55
	2	19.551	6.176	173	9.446.144	54601,99
	3	20.363	7.716	168	9.446.144	56227,05
	4	19.660	6.903	148	9.446.144	63825,3
	5	19.106	8.511	128	9.446.144	73798
	6	19.311	10.238	123	9.446.144	76797,92
	7	19.964	7.316	121	9.446.144	78067,31
	8	21.958	9.544	120	9.446.144	78717,87
	9	58.588	6.734	120	9.446.144	78717,87
						0
LZMA2	1	7.989	7.051	191	9.446.144	49456,25
	2	9.427	6.602	164	9.446.144	57598,44
	3	11.635	7.121	139	9.446.144	67957,87
	4	12.439	5.874	133	9.446.144	71023,64
	5	25.173	5.396	121	9.446.144	78067,31
	6	25.797	5.693	118	9.446.144	80052,07
	7	27.093	6.875	118	9.446.144	80052,07
	8	27.803	7.270	117	9.446.144	80736,27
	9	28.094	12.049	117	9.446.144	80736,27
						0
LZ4	1	4.740	6.393	271	9.446.144	34856,62
	2	5.252	9.456	271	9.446.144	34856,62
	3	5.190	20.876	265	9.446.144	35645,83
	4	4.926	10.471	265	9.446.144	35645,83
	5	5.639	5.794	265	9.446.144	35645,83
	6	4.999	29.247	265	9.446.144	35645,83
	7	5.699	11.244	265	9.446.144	35645,83
	8	5.781	17.508	265	9.446.144	35645,83
	9	4.939	8.577	264	9.446.144	35780,85
	10	5.586	6.665	265	9.446.144	35645,83
	11	6.200	17.468	264	9.446.144	35780,85
	12	7.395	7.342	264	9.446.144	35780,85
						0
LZ5	1	4.996	5.510	274	9.446.144	34474,98
	2	4.765	6.488	273	9.446.144	34601,26
	3	4.828	6.376	271	9.446.144	34856,62
	4	4.838	7.606	270	9.446.144	34985,72
	5	5.060	7.242	269	9.446.144	35115,78
	6	6.984	5.500	269	9.446.144	35115,78
	7	5.797	6.578	269	9.446.144	35115,78
	8	6.403	6.829	269	9.446.144	35115,78
	9	8.506	6.557	269	9.446.144	35115,78
	10	8.314	10.221	269	9.446.144	35115,78

	11	8.521	11.788	268	9.446.144	35246,81
	12	7.624	20.332	268	9.446.144	35246,81
	13	8.055	8.542	269	9.446.144	35115,78
	14	7.757	17.143	269	9.446.144	35115,78
	15	50.664	7.582	269	9.446.144	35115,78
	16	50.227	9.822	269	9.446.144	35115,78
						0
Brotli	0	5.361	6.524	262	9.446.144	36053,98
	1	4.794	6.573	224	9.446.144	42170,29
	2	5.515	5.656	201	9.446.144	46995,74
	3	5.309	7.030	197	9.446.144	47949,97
	4	6.163	6.984	186	9.446.144	50785,72
	5	6.699	7.056	166	9.446.144	56904,48
	6	6.740	7.173	163	9.446.144	57951,8
	7	7.490	5.982	161	9.446.144	58671,7
	8	8.980	5.936	159	9.446.144	59409,71
	9	9.483	6.904	156	9.446.144	60552,21
	10	84.686	6.400	137	9.446.144	68949,96
	11	117.069	6.521	133	9.446.144	71023,64
						0
Zstandard	0	5.581	6.335	222	9.446.144	42550,2
	1	5.154	7.235	222	9.446.144	42550,2
	2	6.530	7.214	203	9.446.144	46532,73
	3	5.909	6.781	198	9.446.144	47707,8
	4	8.729	6.882	182	9.446.144	51901,89
	5	5.734	6.908	163	9.446.144	57951,8
	6	6.018	6.196	162	9.446.144	58309,53
	7	5.751	7.142	162	9.446.144	58309,53
	8	5.924	7.036	160	9.446.144	59038,4
	9	9.860	7.522	149	9.446.144	63396,94
	10	6.540	6.474	149	9.446.144	63396,94
	11	11.592	7.005	149	9.446.144	63396,94
	12	7.682	6.806	149	9.446.144	63396,94
	13	8.279	6.714	149	9.446.144	63396,94
	14	8.458	6.451	149	9.446.144	63396,94
	15	9.318	6.282	149	9.446.144	63396,94
	16	11.589	9.277	148	9.446.144	63825,3
	17	13.865	9.915	144	9.446.144	65598,22
	18	16.714	10.558	143	9.446.144	66056,95
	19	18.109	6.694	142	9.446.144	66522,14
	20	19.211	7.133	138	9.446.144	68450,32
	21	21.673	7.348	136	9.446.144	69456,94
	22	31.744	7.234	136	9.446.144	69456,94

Tabelle 3 für den zweiten Aachener Datensatz:

Algo	Level	Kom_Z	Dekom_Z	Size_K_MiB	Real_S	Ratio
FLZMA2	1	11.024	1.973	209	2.652.246	12690,17
	2	9.739	3.221	178	2.652.246	14900,26
	3	11.454	3.394	173	2.652.246	15330,9
	4	10.823	1.578	159	2.652.246	16680,79
	5	18.185	1.689	146	2.652.246	18166,07
	6	23.051	1.760	144	2.652.246	18418,38
	7	26.868	2.217	142	2.652.246	18677,79
	8	32.406	2.054	140	2.652.246	18944,61
	9	40.084	2.076	139	2.652.246	19080,91
						0
LZMA2	1	6.560	1.817	324	2.652.246	8185,944
	2	6.601	1.562	188	2.652.246	14107,69
	3	7.748	1.671	176	2.652.246	15069,58
	4	8.412	1.761	174	2.652.246	15242,79
	5	33.468	1.985	148	2.652.246	17920,58
	6	38.433	1.998	147	2.652.246	18042,49
	7	58.882	1.869	143	2.652.246	18547,17
	8	72.913	1.972	142	2.652.246	18677,79
	9	72.612	1.992	142	2.652.246	18677,79
						0
LZ4	1	1.351	1.527	528	2.652.246	5023,193
	2	1.357	1.540	528	2.652.246	5023,193
	3	1.602	1.568	494	2.652.246	5368,919
	4	1.658	1.586	492	2.652.246	5390,744
	5	1.699	1.733	487	2.652.246	5446,09
	6	1.804	1.704	485	2.652.246	5468,548
	7	1.940	1.622	484	2.652.246	5479,847
	8	2.222	1.665	484	2.652.246	5479,847
	9	2.175	2.687	484	2.652.246	5479,847
	10	3.692	1.758	483	2.652.246	5491,193
	11	4.537	1.585	483	2.652.246	5491,193
	12	6.035	1.589	483	2.652.246	5491,193
						0
LZ5	1	1.418	1.646	626	2.652.246	4236,815
	2	1.365	1.564	527	2.652.246	5032,725
	3	1.405	1.756	519	2.652.246	5110,301
	4	1.534	1.639	485	2.652.246	5468,548
	5	1.501	1.722	484	2.652.246	5479,847
	6	2.140	1.678	477	2.652.246	5560,264
	7	2.360	1.556	477	2.652.246	5560,264
	8	4.619	1.974	476	2.652.246	5571,945
	9	6.127	1.705	474	2.652.246	5595,456
	10	6.085	1.574	473	2.652.246	5607,285

	11	6.658	2.101	473	2.652.246	5607,285
	12	10.522	1.662	463	2.652.246	5728,393
	13	14.267	1.884	467	2.652.246	5679,328
	14	15.802	1.894	462	2.652.246	5740,792
	15	51.365	1.753	467	2.652.246	5679,328
	16	49.348	1.734	467	2.652.246	5679,328
						0
Brotli	0	1.417	1.611	432	2.652.246	6139,458
	1	1.466	1.655	404	2.652.246	6564,965
	2	2.256	2.764	266	2.652.246	9970,85
	3	2.472	1.834	258	2.652.246	10280,02
	4	7.431	1.583	226	2.652.246	11735,6
	5	8.722	1.666	202	2.652.246	13129,93
	6	12.517	2.189	199	2.652.246	13327,87
	7	16.133	1.615	196	2.652.246	13531,87
	8	21.756	1.902	194	2.652.246	13671,37
	9	27.042	1.818	193	2.652.246	13742,21
	10	92.355	1.915	166	2.652.246	15977,39
	11	327.449	1.948	162	2.652.246	16371,89
						0
Zstandard	0	1.564	2.804	337	2.652.246	7870,166
	1	1.538	2.702	337	2.652.246	7870,166
	2	1.525	2.500	236	2.652.246	11238,33
	3	1.597	2.546	226	2.652.246	11735,6
	4	2.704	2.573	225	2.652.246	11787,76
	5	4.380	2.547	218	2.652.246	12166,27
	6	3.945	2.750	211	2.652.246	12569,89
	7	5.313	2.555	210	2.652.246	12629,74
	8	5.676	2.479	204	2.652.246	13001,21
	9	7.032	2.451	203	2.652.246	13065,25
	10	9.116	2.751	202	2.652.246	13129,93
	11	10.695	2.680	202	2.652.246	13129,93
	12	11.214	2.822	202	2.652.246	13129,93
	13	10.113	2.481	202	2.652.246	13129,93
	14	11.268	2.447	202	2.652.246	13129,93
	15	14.557	2.727	201	2.652.246	13195,25
	16	22.052	2.519	194	2.652.246	13671,37
	17	29.135	2.619	192	2.652.246	13813,78
	18	45.408	2.663	189	2.652.246	14033,05
	19	91.912	2.678	184	2.652.246	14414,38
	20	110.737	2.877	182	2.652.246	14572,78
	21	173.227	2.977	181	2.652.246	14653,29
	22	319.874	2.921	180	2.652.246	14734,7