Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Lehrstuhl für Theoretische Informatik II
Prof. Dr. Joachim Giesen

Entwurf und Implementierung von Sparse-Einsum mittels Sparse-Batch-Matrixmultiplikation

– Bachelorarbeit –

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)
im Studiengang Informatik

Verfasser:
Tim Lippert
geb. am 18.01.2004 in
07743 Jena

Gutachter:
Universitätsprofessor
Dr. Joachim Giesen
Betreuer:
Dr. Mark Blacher

Jena, 9. September 2025

I Zusammenfassung

Die `einsum`-Funktion ist eine bekannte Funktion zur Berechnung von Tensorkontraktionen. Diese wurde vielmals implementiert, jedoch gibt es kaum auf spärliche Tensoren zugeschnittene Implementierungen. Das Ziel dieser Arbeit ist der Entwurf und die Implementierung einer speziell für dünnbesetzte Tensoren konzipierten `einsum`-Funktion, die diese effizient mittels Abbildung auf eine sparse Batch-Matrixmultiplikation (BMM) durch Umordnung und Gruppierung der Indizes kontrahieren kann. Dabei entwickeln wir eine optimierte Speicher- und Indexstruktur zur Verarbeitung von Sparse-Tensoren und zeigen, wie sich paarweise Einsum-Ausdrücke auf eine sparse BMM abbilden lassen[1].

Wir evaluieren die vorgeschlagene Funktion auf synthetischen und aus dem `einsum_benchmark`-Datensatz [1] ausgewählten realen Problemen und untersuchen den Einfluss der Sparsity auf die Effektivität, sowie die Konkurrenzfähigkeit gegenüber anderen (dichten) Implementierungen. Wir demonstrieren ein Speedup bis zu einer Dichte von 0.0032 auf einer synthetischen BMM Berechnung, bis zu einer Dichte von 0.0128 bei der Kontraktion von zufällig generierten Tensor-Hypernetzwerken, sowie eine anderen Implementierungen überlegene Performanz auf ausgewählten realen Problemen aus dem `einsum_benchmark`-Datensatz mit durchschnittlicher Dichte unter 0.005.

[1] Der Code dieses Projekts ist auf einem öffentlich zugänglichen Repository unter `https://github.com/tim-lippert/bachelorthesis` zu finden

Inhaltsverzeichnis

II Abbildungsverzeichnis

III Tabellenverzeichnis

1 Einleitung

Tensorkontraktion ist eine wichtige Operation, die in vielen Forschungs- und Anwendungsgebieten wie der Simulation von Quantenschaltkreisen [2], Graph Neural Networks [3], Model Counting [4] und dem maschinellen Lernen [5] Anwendung findet.
Eine verbreitete Funktion, die solche Kontraktionen berechnen kann, ist die Funktion `einsum`. Diese wurde vielmals implementiert und gehört zu verschiedenen bekannten Bibliotheken wie *NumPy*, *PyTorch* und *TensorFlow*. Diese Implementierungen sind jedoch darauf ausgelegt, dichtbesetzte Tensoren zu kontrahieren. Da in realen Problemen Tensoren oft sehr dünnbesetzt auftreten, wie z.B. Graph-Adjazenz- oder Inzidenztensoren, bleibt bei dichten Implementierungen ein großes Einsparungspotential bei solchen Problemen ungenutzt. Zwar existiert eine spärliche Variante in der `sparse` Bibliothek, `sparse.einsum`, diese weist jedoch große Performanzschwächen bei komplexen Kontraktionen auf und basiert auf einem anderen algorithmischen Ansatz als der von uns verfolgten Berechnung mittels einer Sparse-BMM. Zu Beginn werden in Kapitel 3 die theoretischen Grundlagen des Themas umrandet. Im darauffolgenden Kapitel wird die Berechnung eines `einsum`-Ausdrucks auf BMM-Basis mathematisch beschrieben und die Implementierung dieses Konzepts vorgestellt und erklärt. Diese Implementierung wird im 5. Kapitel sowohl auf synthtischen als auch realen Problemen getestet und die Ergebnisse ausgewertet. Im Anschluss diskutieren wir einzelne Ergebnisse vertiefend und zeichnen Möglichkeiten, wie unsere Implementierung weiterhin optimiert werden kann.

2 Verwandte Arbeiten

Tensor Algebra Compiler für dünnbesetzte Tensoren. Spezialisierte Compiler für dünnbesetzte Tensoren wie TACO (Tensor Algebra Compiler) [6], SparseTIR [7] oder Sparse MLIR [8] haben automatische Generierung von performantem Code für Sparse-Berechnungen ermöglicht. Diese Compiler analysieren sowohl die Formate der Eingabe-Tensoren als auch die Struktur des mathematischen Ausdrucks und generieren daraus maßgeschneiderten, optimierten Code.

Während diese Compiler bei der statischen Code-Generierung sehr leistungsfähig sind, dienen diese primär als Optimierungs-Backends: Sie nehmen einen fest definierten Ausdruck entgegen und generieren eine hocheffiziente, statische Implementierung dafür. Im Gegensatz dazu zielt diese Arbeit hingegen auf eine dynamische Schnittstelle ab, die zur Laufzeit unterschiedliche Einsum-Ausdrücke direkt auswerten kann, ohne für jeden neuen Ausdruck einen expliziten Code-Generierungs- und Kompilierungsschritt zu erfordern.

Bibliotheken für dünnbesetzte lineare Algebra. Neben spezialisierten Compilern existieren einige Bibliotheken für dünnbesetzte lineare Algebra, die hochoptimierte, aber niederschwellige Routinen bereitstellen, darunter Intel MKL [9], SuiteSparse [10], sowie sparse Module von SciPy [11], PyTorch [12] und JAX [13]. Diese Bibliotheken bieten große Performance für spezifische, vordefinierte Operationen. Jedoch müssten komplexe Operationen manuell in eine Abfolge dieser Basisoperationen zerlegt werden.

Einsum-Implementierungen in bestehenden Frameworks. Alle wichtigsten numerischen Computing-Bibliotheken implementieren einsum-Funktionalität, darunter NumPy [14], PyTorch[12] und TensorFlow [15]. Diese Implementierungen arbeiten jedoch ausschließlich auf dichten Tensoren, was erhebliches Einsparungspotential bei dünnbesetzten Strukturen durch effizientere Speicher- und Zugriffsverwaltung ungenutzt lässt.

Bestehende Einsum-Implementierung für dünnbesetzte Tensoren. Eine Implementierung für `einsum`-Operationen spezifisch auf dünnbesetzten Tensoren findet sich in der `sparse`-Bibliothek des PyData-Projekts [16]. Die Auswertung eines Einsum-Ausdrucks erfolgt dabei nicht mit paarweisen Kontraktionen über einem optimierten Pfad, sondern einer globalen Reduktion über alle Tensoren. Zudem wird im Gegensatz zu dem hier vorgestellten Ansatz nicht auf eine BMM gemappt.

Obwohl die `sparse-Bibliothek` das gleiche Problem löst, unterscheidet sich ihr algorithmischer Ansatz fundamental von dem in dieser Arbeit verfolgten und hat eine massive Schwäche. Bei komplexen, langen Einsum-Ausdrücken auf einer großen Anzahl von Tensoren ist die paarweise Kontraktion nach einem optimierten Pfad deutlich effizienter, da sich die Problemgröße bei globaler Berechnung extrem aufblähen kann.

3 Theoretische Grundlagen

3.1 Tensoren

Ein Tensor ist eine verallgemeinerte mathematische Struktur, die als mehrdimensionale Anordnung von Zahlen betrachtet werden kann. Tensoren erweitern das Konzept von Skalaren (0-dimensional), Vektoren (1-dimensional) und Matrizen (2-dimensional) auf beliebig viele Dimensionen. Die Anzahl der Dimensionen wird als Rang n eines Tensors bezeichnet. Hier eine Beispielvisualisierung für einen Tensor mit Rang 3:

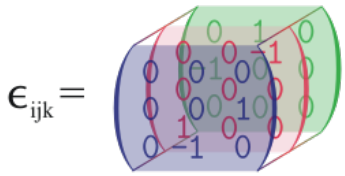

Abbildung 3.1: Ein Tensor vom Rang 3. Quelle: Kriesch [17]

3.1.1 Dichte

Die Dichte eines Tensors ist ein Maß dafür, wie viele seiner Elemente einen Wert ungleich Null haben. Sie wird als das Verhältnis der Anzahl der Nicht-Null-Elemente zur Gesamtzahl der Elemente im Tensor definiert.

3.1.2 Sparse-Formate

In der praktischen Anwendung treten Tensoren oft dünnbesetzt (sparse) auf, also mit sehr niedriger Dichte. Dabei haben die meisten Einträge des Tensors den Wert 0. Für solche Tensoren bietet es sich an, diese komprimiert zu speichern. Hierfür gibt es verschiedene Formate: COO (Coordinate), CSR (Compressed Sparse Row), CSF (Compressed Sparse Fiber), BSR (Block Compressed Sparse Row) und DOK (Dictionary of Keys) sind nur eine Auswahl davon. Jedes dieser Formate besitzt Vor- und Nachteile. Das COO-Format findet leicht abgewandelt in dieser Arbeit Anwendung.
Das COO-Format repräsentiert einen Sparse Tensor durch drei separate Arrays:

1. Koordinaten-Array: Speichert die Indizes aller Nicht-Null-Elemente
2. Werte-Array: Enthält die entsprechenden Nicht-Null-Werte
3. Shape-Array: Definiert die Dimensionen des ursprünglichen Tensors

Für einen n-dimensionalen Tensor mit k Nicht-Null-Elementen benötigt das COO-Format somit ein Koordinaten-Array der Größe (n × k), ein Werte-Array der Größe k und einen Shape-Vektor der Größe n. Vorteile dieses Formats sind beispielsweise eine intuitive Verständlichkeit, simple Implementierung und eine hohe Effizienz beim inkrementellen Aufbau des Tensors.

3.1.3 Fundamentale Tensor-Operationen

3.1.3.1 Transponierung von Tensoren

Die Transponierung ist eine fundamentale Operation in der linearen Algebra und im maschinellen Lernen, die die Dimensionen eines Tensors permutiert. Formal definiert die Transponierung eine Neuanordnung der Achsen eines Tensors gemäß einer gegebenen Permutation.

Mathematische Definition Sei $T \in \mathbb{R}^{d_1 \times d_2 \times \ldots \times d_n}$ ein n-dimensionaler Tensor und $\pi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ eine Permutation der Dimensionsindizes. Die Transponierung von T bezüglich π ist definiert als:

$$T^{\pi}[i_{\pi(1)}, i_{\pi(2)}, \ldots, i_{\pi(n)}] = T[i_1, i_2, \ldots, i_n]$$

Der resultierende Tensor T^{π} hat die Form $d_{\pi(1)} \times d_{\pi(2)} \times \ldots \times d_{\pi(n)}$.

3.1.3.2 Reshape von Tensoren

Die Reshape-Operation verändert die Form (Shape) eines Tensors, ohne die zugrunde liegenden Daten oder deren Reihenfolge zu ändern.

Mathematische Definition Sei $T \in \mathbb{R}^{d_1 \times d_2 \times \ldots \times d_n}$ ein Tensor mit $N = \prod_{i=1}^{n} d_i$ Elementen. Eine Reshape-Operation transformiert T in einen Tensor $T' \in \mathbb{R}^{d'_1 \times d'_2 \times \ldots \times d'_m}$ mit der Bedingung $\prod_{j=1}^{m} d'_j = N$.

Der lineare Index k für ein Element $T[i_1, i_2, \ldots, i_n]$ berechnet sich als:

$$k = \sum_{j=1}^{n} i_j \prod_{l=j+1}^{n} d_l = i_1 \cdot (d_2 \cdot d_3 \cdots d_n) + i_2 \cdot (d_3 \cdots d_n) + \ldots + i_{n-1} \cdot d_n + i_n$$

Die linearisierten Elemente werden dann in die neue Tensor-Form umgeordnet. Für ein Element am linearen Index k in der neuen Form berechnen sich die neuen Koordinaten $(i'_1, i'_2, \ldots, i'_m)$ durch:

$$\begin{aligned} i'_1 &= \lfloor k/(d'_2 \cdot d'_3 \cdots d'_m) \rfloor \\ i'_2 &= \lfloor (k \bmod (d'_2 \cdot d'_3 \cdots d'_m))/(d'_3 \cdots d'_m) \rfloor \\ &\vdots \\ i'_m &= k \bmod d'_m \end{aligned}$$

3.1.3.3 Batch-Matrixmultiplikation (BMM)

Die Batch-Matrixmultiplikation ist eine Erweiterung der herkömmlichen Matrixmultiplikation, die es ermöglicht, Stapel von Matrizen (Batches) effizient und parallel zu verarbeiten.

Mathematische Definition Seien $X \in \mathbb{R}^{b \times n \times m}$ und $Y \in \mathbb{R}^{b \times m \times p}$ zwei Tensoren, die jeweils einen Stapel von b Matrizen repräsentieren. Die batched Matrixmultiplikation erzeugt einen Tensor $Z \in \mathbb{R}^{b \times n \times p}$. Jede Matrix Z_k im Ausgabe-Tensor ist das Ergebnis der Matrixmultiplikation der entsprechenden Matrizen X_k und Y_k aus den Eingabe-Tensoren.

Formal ist die Operation für jede Matrix im Stapel ($k = 1, \ldots, b$) definiert als:

$$(Z_k)_{ij} = \sum_{l=1}^{m} (X_k)_{il} (Y_k)_{lj}$$

wobei (X_k), (Y_k) und (Z_k) die k-ten Matrizen in den jeweiligen Tensoren sind.

3.2 Einsum

Die Darstellung der Konzepte der Notation, Tensor-Hypernetzwerken und Tensor-Kontraktion dieses Abschnitts orientiert sich an der Darstellung von Orgler und Blacher [18].

3.2.1 Geschichte

Die Einsteinsche Summenkonvention ist eine Notation, die von Albert Einstein Anfang des 20. Jahrhunderts entwickelt und 1916 eingeführt wurde, um Tensorberechnungen in der theoretischen Physik einfacher auszudrücken. Der Kern der Notation besteht darin, dass zwischen den Operanden wiederholte Indizes in einem Ausdruck eine implizite Summation über diese Indizes darstellen. Dieses Prinzip wurde später in maschinellen Lernbibliotheken aufgegriffen und zur modernen Einsum-Notation, kurz für Einstein Summation, weiterentwickelt, da es sich ideal eignet, um komplexe Operationen wie Matrixmultiplikationen, Transpositionen, Batch-Verarbeitung oder auch höherdimensionale Tensorprodukte effizient zu formulieren.

3.2.2 Notation

Betrachten wir eine einfache Tensor-Multiplikation $A \cdot B$:

$$(A \cdot B)_{i,j,l,n} = \sum_{k} A_{i,j,k} \cdot B_{k,l,n}$$

In der Einsteinschen Summenkonvention werden zwischen den Operanden wiederholte Indizes implizit aufsummiert. Folglich kann die obige Tensor-Multiplikation in dieser Notation wie folgt geschrieben werden:

$$(A \cdot B)_{i,j,l,n} = A_{i,j,k} \cdot B_{k,l,n}$$

In der modernen Notation werden die Bezeichner der Operanden nicht mehr benannt. Der gesamte Ausdruck wird in eine Zeichenfolge gebracht, in der die Indizes der Operanden durch ein Komma getrennt und mit einem Pfeil von den Indizes des Ergebnisses abgegrenzt werden. Der obige Ausdruck kann in der modernen Notation also wie folgt beschrieben werden:

$$ijk, kln \rightarrow ijln.$$

3.2.3 Tensor-Hypernetzwerke

Ein Einsum-Ausdruck lässt sich als Tensor-Hypernetzwerk betrachten. Dabei entsprechen Tensoren den Knoten und Indizes den Kanten des Netzwerks. Teilen zwei Tensoren einen Index, verbindet eine Kante die entsprechenden Knoten, wird ein Index von drei oder mehr Tensoren gemeinsam genutzt, entsteht eine Hyperkante. Offene Kanten repräsentieren die Ausgabedimensionen, während über interne Kanten (geteilte Indizes) summiert wird. Ein Beispielnetzwerk für die Tensoren A mit Indizes i, j, k, B mit Indizes k, l, m und C mit Indizes j, k, n ist in Abbildung 3.2 zu sehen.

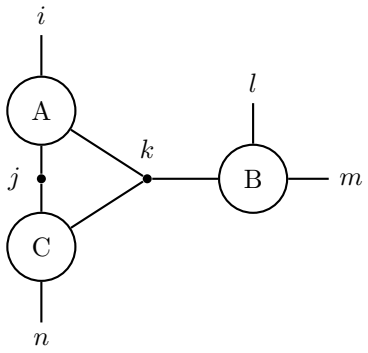

Abbildung 3.2: Tensor-Hypernetzwerk für den `einsum`-Ausdruck `ijk,klm,jkn→ilmn`

3.2.4 Tensor-Kontraktion

Das Kontrahieren eines Tensor-Hypernetzwerks entspricht dem Summieren über alle internen Indizes. Für das Beispiel

$$N \;=\; (A \cdot B \cdot C)_{ilmn} \;=\; \sum_{k} \sum_{j} A_{ijk}\, B_{klm}\, C_{jkn}$$

entfällt in der Einsteinschen Notation das Summenzeichen über den mehrfach auftretenden Indizes k und j, und in moderner Einsum-Notation lautet der Ausdruck:

$$\texttt{ijk,klm,jkn} \rightarrow \texttt{ilmn}.$$

Praktisch wird die Kontraktion meist paarweise ausgeführt, etwa zuerst $(A \cdot B)$ und anschließend $(AB) \cdot C$, oder alternativ $(B \cdot C)$ gefolgt von $A \cdot (BC)$. Obwohl alle Reihenfolgen dasselbe Resultat liefern, können sie sich erheblich in Speicherbedarf und Laufzeit unterscheiden. Unser Algorithmus wird einen bereits optimierten annotierten Kontraktionspfad als Eingabe verwenden, durch den er sich im Anschluss sequentiell durcharbeitet und die gegebenen Ausdrücke paarweise kontrahiert.

3.2.5 Mapping von Einsum zu Batch-Matrix-Multiplikation

Jeder paarweise Einsum-Ausdruck kann durch Vorverarbeitung, Permutierung und Reshape systematisch auf eine BMM abgebildet werden. Diese Reduktion erlaubt es, eine große Klasse von Einsum-Berechnungen auf einen hochoptimierten BMM-Kern zurückzuführen und bildet die Grundlage der auf spärliche Tensoren zugeschnittenen Implementierung dieser Arbeit. Diese Abbildung bzw. Implementierung wird im folgenden Kapitel sowohl mathematisch als auch algorithmisch beschrieben.

4 Implementierung

4.1 Mathematische Formulierung der paarweisen Einsum-Transformation

Die Implementierung der `einsum`-Operation basiert auf der Transformation der allgemeinen Tensor-Kontraktion in eine BMM. Dieser Prozess lässt sich in die folgenden Schritte unterteilen.

4.1.1 Problemstellung und Notation

Gegeben sei eine `einsum`-Operation, definiert durch eine Indexnotation $\Sigma_1, \Sigma_2 \to \Sigma_3$ und zwei Eingabe-Tensoren A und B. Sei Σ_1 die Sequenz der Indizes für Tensor A und Σ_2 für Tensor B. Die Operation berechnet einen Ergebnis-Tensor C mit Indexsequenz Σ_3.

4.1.2 Index-Partitionierung

Die Menge aller Eingabe-Indizes $\mathcal{X} = \text{set}(\Sigma_1) \cup \text{set}(\Sigma_2)$ wird im ersten Schritt in sechs disjunkte Teilmengen partitioniert, die die Rolle jedes Index in der Kontraktion definieren:

$$
\begin{aligned}
\mathcal{I}_{\text{batch}} &:= \{i \mid i \in \Sigma_1 \cap \Sigma_2 \cap \Sigma_3\} && \text{(Batch-Indizes, bleiben erhalten)} \\
\mathcal{I}_{\text{contract}} &:= \{i \mid i \in \Sigma_1 \cap \Sigma_2 \wedge i \notin \Sigma_3\} && \text{(Kontraktions-Indizes, werden summiert)} \\
\mathcal{I}_{\text{left-sum}} &:= \{i \mid i \in \Sigma_1 \wedge i \notin \Sigma_2 \wedge i \notin \Sigma_3\} && \text{(Summations-Indizes, nur in A)} \\
\mathcal{I}_{\text{right-sum}} &:= \{i \mid i \in \Sigma_2 \wedge i \notin \Sigma_1 \wedge i \notin \Sigma_3\} && \text{(Summations-Indizes, nur in B)} \\
\mathcal{I}_{\text{left-kept}} &:= \{i \mid i \in \Sigma_1 \wedge i \notin \Sigma_2 \wedge i \in \Sigma_3\} && \text{(Erhaltene Indizes, nur in A)} \\
\mathcal{I}_{\text{right-kept}} &:= \{i \mid i \in \Sigma_2 \wedge i \notin \Sigma_1 \wedge i \in \Sigma_3\} && \text{(Erhaltene Indizes, nur in B)}
\end{aligned}
$$

4.1.3 Vorverarbeitung der Tensoren

Im zweiten Schritt werden die Eingabe-Tensoren durch zwei Operationen vorbereitet:

Diagonalisierung (Entfernung doppelter Indizes) Falls ein Index in Σ_1 oder Σ_2 mehrfach vorkommt (z.B. `ijj`), werden die entsprechenden Tensoren auf ihre Diagonale reduziert. Dies ist eine Abbildung f_{diag}:

$$
\begin{aligned}
(A', \Sigma_1') &= f_{\text{diag}}(A, \Sigma_1) \\
(B', \Sigma_2') &= f_{\text{diag}}(B, \Sigma_2)
\end{aligned}
$$

Summation Es wird über die reinen Summations-Indizes, die in genau einem Eingabetensor auftreten summiert:

$$A_{\text{red}} = \sum_{i \in \mathcal{I}_{\text{left-sum}}} A' \quad \text{mit neuer Indexsequenz} \quad \Sigma_1'' = \Sigma_1' \setminus \mathcal{I}_{\text{left-sum}}$$
$$B_{\text{red}} = \sum_{i \in \mathcal{I}_{\text{right-sum}}} B' \quad \text{mit neuer Indexsequenz} \quad \Sigma_2'' = \Sigma_2' \setminus \mathcal{I}_{\text{right-sum}}$$

4.1.4 Transformation zur BMM-Form

Im dritten Schritt werden die vorbereiteten Tensoren nun in BMM-Form gebracht. Die Reihenfolge der Achsen ist dabei kernelabhängig.

Permutation der Achsen Zuerst werden die Achsen der Tensoren so umsortiert (permutiert), dass sie der BMM-Struktur entsprechen. Die Ziel-Ordnung für den `v1`-Kernel ist (Batch, Contract, Kept) für den `v2`-Kernel ist (Batch, Kept, Contract).

$$A_{\text{perm}} = \mathcal{P}_{\Sigma_1'' \to \Sigma_A}(A_{\text{red}}) \qquad \text{wobei} \quad \Sigma_A = \begin{cases} (\mathcal{I}_{\text{batch}}, \mathcal{I}_{\text{contract}}, \mathcal{I}_{\text{left-kept}}) & \text{bei Kernel v1} \\ (\mathcal{I}_{\text{batch}}, \mathcal{I}_{\text{left-kept}}, \mathcal{I}_{\text{contract}}) & \text{bei Kernel v2} \end{cases}$$
$$B_{\text{perm}} = \mathcal{P}_{\Sigma_2'' \to \Sigma_B}(B_{\text{red}}) \qquad \text{wobei} \quad \Sigma_B = \begin{cases} (\mathcal{I}_{\text{batch}}, \mathcal{I}_{\text{contract}}, \mathcal{I}_{\text{right-kept}}) & \text{bei Kernel v1} \\ (\mathcal{I}_{\text{batch}}, \mathcal{I}_{\text{right-kept}}, \mathcal{I}_{\text{contract}}) & \text{bei Kernel v2} \end{cases}$$

Hierbei ist $\mathcal{P}$ der Permutationsoperator.

Reshape Im Anschluss werden die permutierten Tensoren zu 3D-Tensoren umgeformt, indem die Dimensionen innerhalb jeder Indexgruppe (Batch, Kept, Contract) zusammengefasst werden. Seien $D_\Sigma = \{\prod_{i \in \mathcal{I}_X} \dim(i) \mid \mathcal{I}_X \in \Sigma\}$ die Dimensionsprodukte der einzelnen Indexgruppen. Damit gilt:

$$A_{\text{bmm}} = \text{reshape}(A_{\text{perm}}, D_{\Sigma_A})$$
$$B_{\text{bmm}} = \text{reshape}(B_{\text{perm}}, D_{\Sigma_B})$$

4.1.5 Kernoperation: Batch-Matrix-Multiplikation

Im vierten Schritt erfolgt die eigentliche Kontraktion nun als BMM. Die resultierende Matrix C_{bmm} hat die Shape $(D_{\text{batch}}, D_{\text{left-kept}}, D_{\text{right-kept}})$. Für die beiden Kernel ergibt sich

$$(C_{\text{bmm}})_{b,i,k} = \begin{cases} \sum_{j=1}^{D_{\text{contract}}} (A_{\text{bmm}})_{b,j,i} \cdot (B_{\text{bmm}})_{b,j,k} & \text{bei Kernel v1.} \\ \sum_{j=1}^{D_{\text{contract}}} (A_{\text{bmm}})_{b,i,j} \cdot (B_{\text{bmm}})_{b,k,j} & \text{bei Kernel v2.} \end{cases}$$

4.1.6 Rücktransformation

Im letzten Schritt wird der 3D-Ergebnis-Tensor in die geforderte Zielform zurücktransformiert.

Rück-Reshape Zuerst wird C_{bmm} zurück in einen höherdimensionalen Tensor geformt, dessen Achsen den Ziel-Indizes entsprechen. Die resultierende Indexordnung ist $\Sigma_C = (\mathcal{I}_{\text{batch}}, \mathcal{I}_{\text{left-kept}}, \mathcal{I}_{\text{right-kept}})$. Es gilt:

$$C_{\text{shaped}} = \text{reshape}(C_{\text{bmm}}, (\dim(\mathcal{I}_{\text{batch}}), \dim(\mathcal{I}_{\text{left-kept}}), \dim(\mathcal{I}_{\text{right-kept}})))$$

Finale Permutation Im Anschluss wird der Tensor final in die Ziel-Indexordnung Σ_3 permutiert. Es gilt:

$$C = \mathcal{P}_{\Sigma_C \to \Sigma_3}(C_{\text{shaped}})$$

Dieser Prozess ermöglicht es, eine breite Klasse von `einsum`-Operationen auf einen einzigen, hochoptimierten und parallelisierten BMM-Kernel abzubilden.

4.2 Technische Implementierung

4.2.1 Datenstruktur

Das Herzstück der Implementierung ist eine spezialisierte Datenstruktur zur Repräsentation dünn besetzter Tensoren: `SparseTensor`. Der grundlegende Unterschied zu herkömmlichen COO-Formaten, die Koordinaten typischerweise als mehrdimensionale Arrays speichern, liegt im **Koordinaten-Packing**, das in Abbildung 4.1 dargestellt ist.

Grundidee. Anstatt die Koordinaten eines Nicht-Null-Elements in einem Vektor oder Tupel zu speichern, werden sie bitweise in einen einzigen 128-Bit-Integer gepackt. Dies transformiert eine Liste von Koordinatenvektoren in eine einzige Liste von Integern und ist in Abbildung 4.1 verdeutlicht.

Gewählt wurde dieses Design aufgrund von zwei signifikanten Vorteilen: Einerseits entfällt der Pointer-Overhead für separate Koordinatenvektoren und sorgt so für Speichereffizienz. Zudem wird der Vergleich zweier Koordinaten zu einem einzigen, extrem schnellen Integer-Vergleich. Dies ist besonders vorteilhaft für Sortieralgorithmen, die in der Permutations-Operation genutzt werden um effiziente BMM-Kernel zu ermöglichen. Die lexikographische Sortierung der Koordinaten wird zu einer simplen numerischen Sortierung der gepackten Integer. Der Tradeoff für diese Vorteile ist die Annahme, dass die Summe der für die Darstellung der jeweiligen Dimensionen benötigten Bits 128 Bits nicht überschreiten darf.

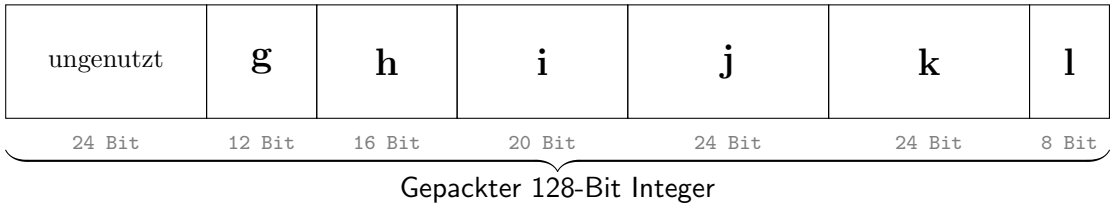

Abbildung 4.1: Gepackte Koordinaten eines 6-Dimensionalen Tensors unterschiedlich großer Dimensionen in einem 128-Bit Integer

Pack- und Entpack-Operationen. Zwei zentrale Methoden ermöglichen die Umwandlung zwischen Koordinatenvektoren und gepackten Integern:

- `pack_coords`: Diese Funktion nimmt einen Koordinatenvektor entgegen, iteriert durch die Koordinaten, verschiebt den bereits gepackten Wert um die für die aktuelle Dimension benötigte Bit-Anzahl nach links und fügt die neue Koordinate hinzu. Das Ergebnis ist ein einzelner `uint128_t`.
- `unpack_coords`: Diese Funktion kehrt den Prozess um. Sie extrahiert die einzelnen Koordinaten aus einem `uint128_t`, indem sie durch wiederholtes Anwenden von Bit-Masken und Rechts-Verschiebungen die einzelnen Bit-Segmente isoliert und in einen Integer umwandelt, und diese als Vektor zurückgibt.

Das Konzept dieser Struktur ist für intuitiveres Verständnis in Abbildung 4.2 visualisiert.

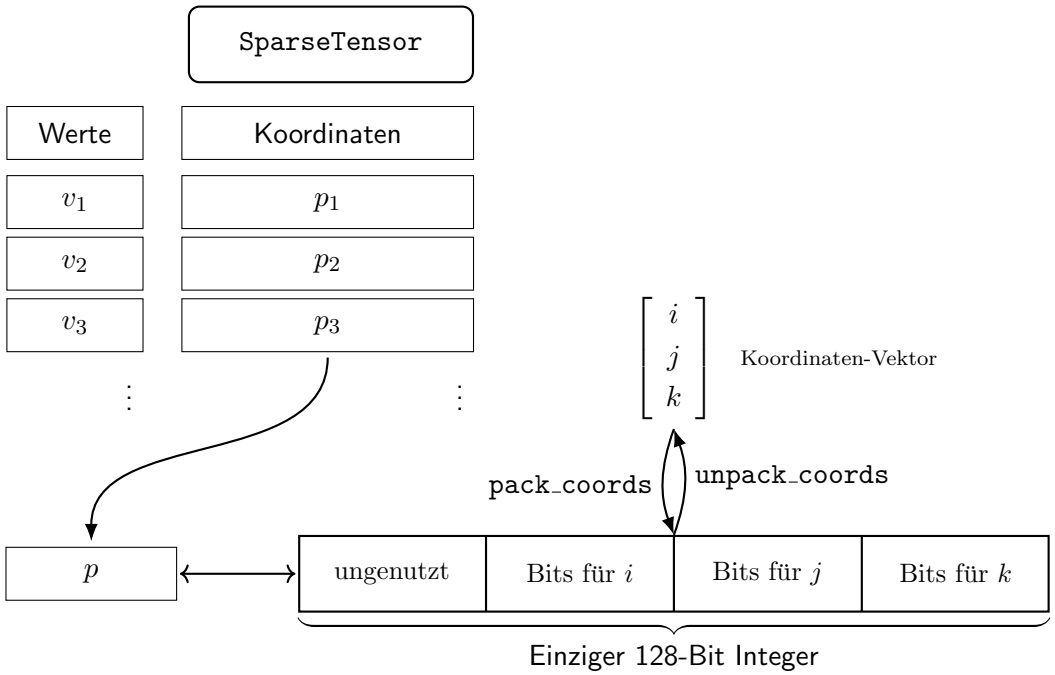

Abbildung 4.2: Funktionsweise der `SparseTensor`-Struktur für einen Beispieltensor mit drei gleich großen Dimensionen

4.2.2 Diagonalisierung

Die Funktion `clear_repeated_indices` ist für die Behandlung von `einsum`-Notationen mit wiederholten Indizes in einem der Eingabe-Operanden zuständig und eliminiert doppelte Vorkommen (z.B. `ijjik->ijk`). Der Algorithmus identifiziert zunächst, ob und welche Indizes in der Index-Sequenz eines Tensors mehrfach vorkommen. Wenn wiederholte Indizes gefunden werden, muss ein neuer, kleinerer Tensor erstellt werden, der nur die Elemente enthält, die auf den entsprechenden Diagonalen liegen. Das wird erreicht, indem die Indizes analysiert werden, und im Anschluss für jedes Element im Tensor geprüft wird, ob die Koordinaten auf solchen mehrfach vorkommenden Achsen gleich sind. Für besseres Verständnis ist diese Diagonalbedingung für Beispielindizes “ijjik” in Abbildung 4.3

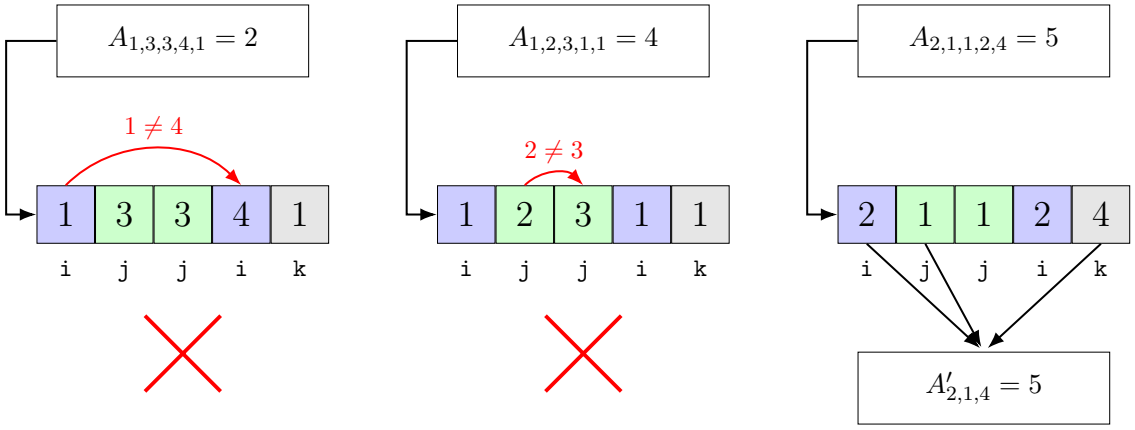

Abbildung 4.3: Prüfung der Elemente bei Diagonalisierung eines Tensors mit Indizes “ijjik” zu einem Tensor mit Indizes “ijk”

dargestellt. Nur die Elemente die diese Diagonalbedingung erfüllen werden in den Ergebnistensor übertragen. Zum Schluss wird die Indexsequenz an den neu erstellten Tensor angepasst.

4.2.3 Reduktion durch Summation

Die Funktion `eliminate_summed_indices` führt Summation über die Summationsindizes aus, also den Indizes, die ausschließlich in genau einem Eingabetensor auftreten und gemäß `einsum`-Regeln durch Summation eliminiert werden müssen. Das können wir erreichen, indem wir alle Nicht-Null-Elemente des Tensors basierend auf den Koordinaten der beizubehaltenden Achsen gruppieren und die gesammelten Werte aufsummieren. Für die einzelnen Elemente wird die zugehörige Gruppe ermittelt und der Wert aufsummiert, was auf der nächsten Seite in Abbildung 4.4 visualisiert wurde. Zur Umsetzung dieser Logik wird eine `unordered_map` verwendet, bei der die gepackten Zielkoordinaten jedes Elements als Schlüssel dienen. Jede Gruppe stellt nach Abarbeitung der einzelnen Elemente einen Eintrag im Ergebnistensor dar.

4.2.4 Transponierung und Sortierung

Die Funktion `transpose_and_sort` ist ein entscheidender Schritt zur Vorbereitung der Tensoren auf die BMM. Ihre Aufgabe ist es, die Achsen eines Tensors in eine definierte Zielreihenfolge zu permutieren und die Nicht-Null-Elemente entsprechend der neuen Koordinatenordnung zu sortieren. Die nachfolgenden BMM-Algorithmen sind auf diese Sortierung angewiesen, um eine hohe Effizienz zu erreichen. Der Prozess ist in zwei Hauptphasen unterteilt: eine parallelisierte Permutation und eine parallelisierte Sortierung. Die Sortierung kann durch das Setzen einer Flag weggelassen werden, da bei dem Transponieren nach der durchgeführten BMM auf die Sortierung verzichtet werden kann.

1. **Permutations-Mapping erstellen:** Zuerst wird ein Permutationsvektor berechnet, der angibt, wie die alten Achsen auf die neuen Positionen abgebildet werden.

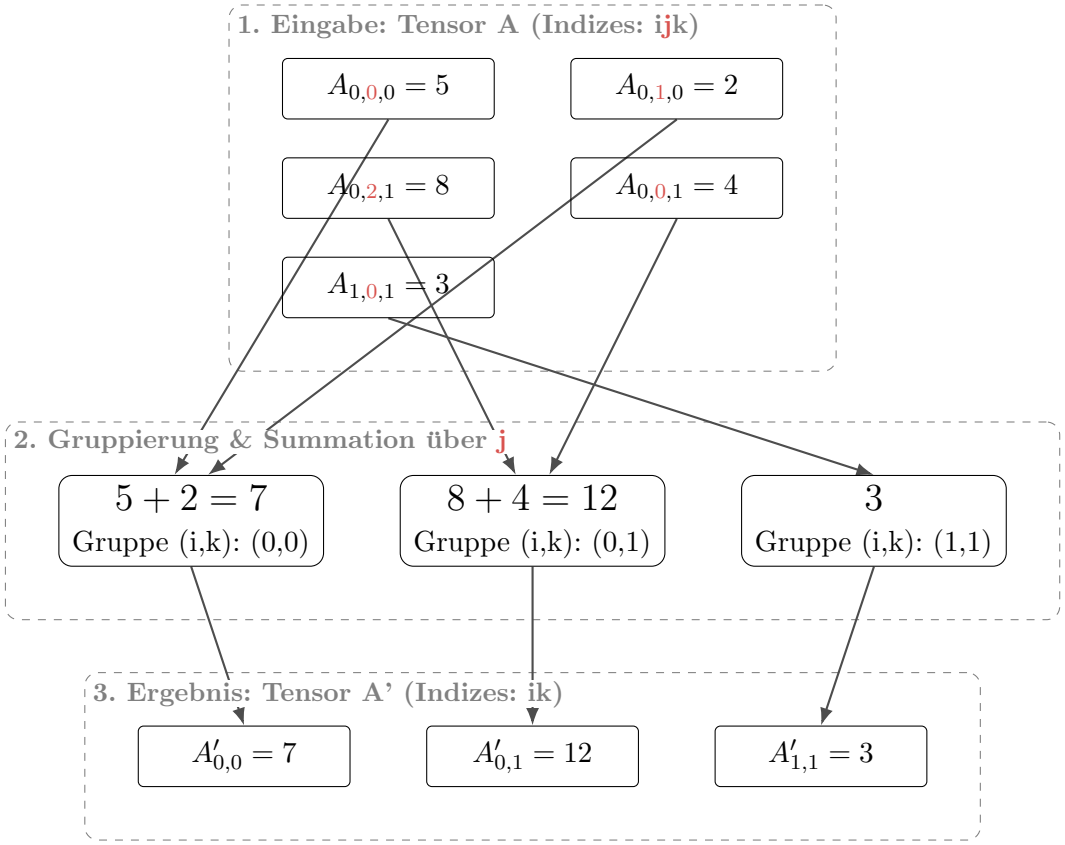

Abbildung 4.4: Summierung über die zweite Achse eines 3D-Beispieltensors mit fünf Elementen

2. **Parallele Permutation der Koordinaten:** In einer mit OpenMP [19] parallelisierten Schleife wird jedes Element des Eingabe-Tensors verarbeitet: Die ursprüngliche Koordinate wird aus dem 128-Bit-Integer entpackt, und durch Anwenden des Permutations-Mappings wird ein neuer Koordinatenvektor erstellt. Diese neue Koordinate wird wieder gepackt und zusammen mit dem zugehörigen Wert des Elements in einer temporären Liste von `CoordValue`-Strukturen gespeichert.

3. **Parallele Sortierung:** Nach der Permutation enthält die temporäre Liste alle Elemente des neuen Tensors, jedoch in einer durch die Permutation bedingten unsortierten Reihenfolge. Die hochperformante, parallelisierte Sortierbibliothek `ips4o` [20] wird verwendet, um diese Liste zu sortieren. Da die Sortierung direkt auf den gepackten 128-Bit-Integer-Koordinaten operiert, ist dieser Schritt extrem effizient. Das Ergebnis ist eine nach den neuen Koordinaten lexikographisch sortierte Liste.

4. **Erstellung des Ergebnis-Tensors:** Abschließend wird ein neuer `SparseTensor` erstellt und seine Koordinaten- und Werte-Vektoren werden aus der nun sortierten temporären Liste befüllt.

4.2.5 Umformung von Tensoren

Die Funktion `reshape` dient dazu, die Dimensionen eines Tensors zu verändern, während die Gesamtzahl und die Daten der Nicht-Null-Elemente sowie deren Sortierung erhalten bleiben. Diese Operation ist notwendig, um die permutierten Tensoren in das für die beiden BMM-Kernel erforderliche 3D-Format zu bringen, indem mehrere Batch-, Kontraktions- und Behaltene-Indizes zu einer einzigen Dimension zusammengefasst werden, bzw. um das 3D-Format des BMM-Ergebnistensors auf die Ziel-Achsen umzuformen. Analog zur theoretischen Beschreibung der `reshape`-Operation in Abschnitt 3.1.3.2 erfolgt dieses Reshape mittels Transformation über den linearen Index, wie für ein Beispiel in Abbildung 4.5 abgebildet. Zuerst werden die Strides der alten und neuen Tensorform berechnet. Im Anschluss erfolgen die Transformationen der Koordinaten: Jeder Eintrag wird mittels der alten Strides auf einen linearen Index abgebildet, aus dem die Koordinaten des neuen Eintrags mit den neuen Strides berechnet wird. Da die Einträge unabhängig voneinander sind, konnte diese Berechnung mittels OpenMP trivial parallelisiert werden

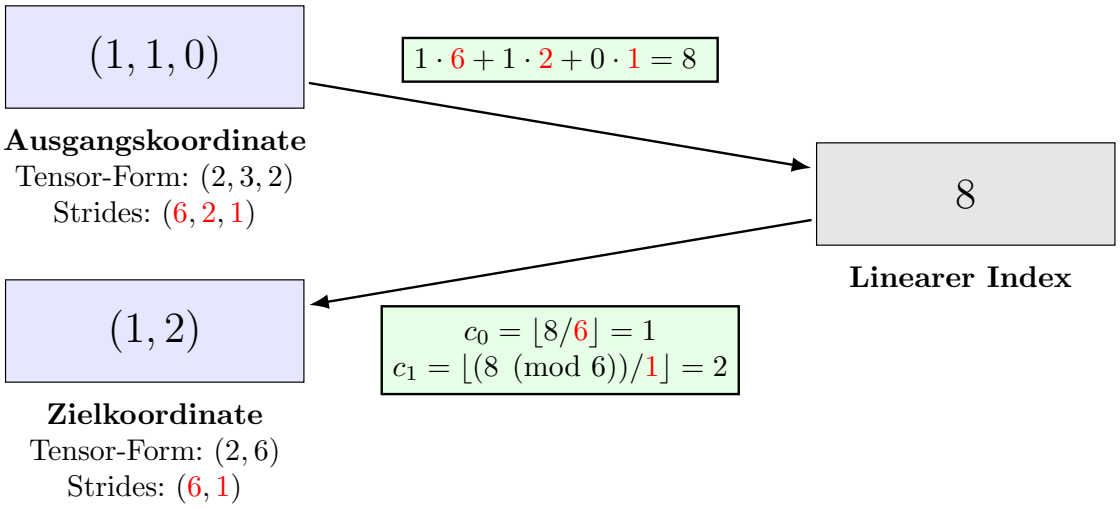

Abbildung 4.5: Beispieltransformation einer Koordinate

4.2.6 BMM-Rechenkern

Die Funktionen `sparse_bmm_bji_bjk` (v1) und `sparse_bmm_bij_bkj` (v2) stellen parallelisierte Rechenkerne für eine BMM dar. Sie sind spezifisch für die Kontraktion von Tensoren mit der Index-Anordnung `bji,bjk->bik` bzw. `bij,bkj->bik` entworfen. Die Algorithmen nutzen die Sortiertheit der Einträge in den Tensoren aus, um die Tensoren effizient zu verarbeiten.

4.2.6.1 Motivation

Die Motivation hinter der Entwicklung von zwei verschiedenen Rechenkernen ist der Tradeoff zwischen Parallelisierung und Kosten der Zusammenführung der Ergebnisse

Rechenkern v1: Durch die Anordnung `bji,bjk->bik` ist eine sehr feine Parallelisierung nach (b,j)-Blöcken möglich. Tradeoff dafür ist das Aufblähen des Problems: Jeder Block berechnet ein äußeres Produkt über $i \times k$, was am Ende sequenziell zwischen den Blöcken aggregiert werden muss. Je dichter diese Tensoren besetzt sind, desto größer wird dieses äußere Produkt. Es soll untersucht werden, ob dieser signifikante Nachteil im algorithmischen Vorgehen durch die bessere Parallelisierung speziell im spärlichen Anwendungsfall ausgeglichen wird.

Rechenkern v2: Durch die Anordnung `bij,bkj->bik` ist ein anderes Vorgehen möglich: Eine Parallelisierung nur auf Batch-Ebene, wobei für jeden Batch direkte Endergebnisse effizient berechnet werden können. Am Ende müssen nur noch die fertigen Werte jedes Batches gesammelt werden. Es soll geprüft werden, ob sich eine effizientere Zusammenführung im Tausch für eine schlechtere Parallelisierung lohnt.

4.2.6.2 Aufgabengenerierung

Rechenkern v1: Die beiden Eingabe-Tensoren A und B werden gleichzeitig durchlaufen, um übereinstimmenden Batch-Indizes b und Kontraktionsindizes j in den Tensoren zu bestimmen. Jeder übereinstimmende Batch definiert eine unabhängige Berechnungsaufgabe (`BJTask`), die den Indexbereich der Nicht-Null-Elemente für diese (b,j) in beiden Tensoren festhält.

Rechenkern v2: Die beiden Eingabe-Tensoren A und B weden gleichzeitig durchlaufen, um übereinstimmenden Batch-Indizes b in den Tensoren zu bestimmen. Jeder übereinstimmende Batch definiert eine unabhängige Berechnungsaufgabe (`BatchTask`), die den Indexbereich der Nicht-Null-Elemente für diesen Batch in beiden Tensoren festhält.

4.2.6.3 Parallele Verarbeitung

Rechenkern v1: Die ermittelten Aufgaben werden parallel berechnet. Für jede Aufgabe wird für jedes i und für jedes k der Eintrag des Zieltensors $C_{b,i,k} = A_{b,j,i} \cdot B_{b,j,k}$ berechnet. Jeder Thread sammelt seine Ergebnisse in einem eigenen, thread-privaten `SparseTensor`. Zu diesem Zeitpunkt können zwei verschiedene lokale Tensoren gleiche Koordinaten besitzen, da die Summation über `j` noch nicht stattgefunden hat, was das Problem ziemlich aufbläht.

Rechenkern v2: Die emittelten Batches werden parallelisiert berechnet. Für jeden Batch wird ein eigener `SparseTensor` zur Sammlung der fertigen Werte erstellt. Innerhalb eines Batches (also innerhalb eines Threads) wird über alle Zeilen i von Tensor A und alle Zeilen k von Tensor B iteriert. Für jedes Paar (i, k) wird ein einzelner Wert des Ergebnis-Tensors $(C)_{b,i,k}$ berechnet, indem über gleiche j multipliziert und aufsummiert wird. Dieser wird anschließend in dem Thread-spezifischen `SparseTensor` gespeichert.

4.2.6.4 Aggregation

Rechenkern v1: Nach der parallelen Berechnung der Tasks durchläuft der Haupt-Thread die Ergebnislisten aller Threads. Er verwendet eine `unordered_map`, um die Teilprodukte effizient zu summieren. Dabei dient der gepackte 128-Bit-Koordinaten-Integer als Schlüssel. Für jedes Teilprodukt wird dessen Wert zum des entsprechenden Schlüssels zugehörigen Wert addiert. Am Ende dieses Schrittes enthält die Hash-Map für jede Koordinate des Ergebnis-Tensors genau einen Eintrag mit dem finalen, summierten Wert. Der saubere Ergebnis-Tensor wird direkt aus den Schlüssel-Wert-Paaren der nun vollständig aggregierten Hash-Map aufgebaut.

Rechenkern v2: Nach der parallelen Berechnung der Tasks werden die Ergebnisse aus den thread-privaten Tensoren in einem letzten, sequentiellen Schritt in einen finalen Ergebnis-Tensor zusammengeführt.

4.2.7 Die `einsum`-Pipeline

Die beiden Funktionen `perform_einsum_v1` & `perform_einsum_v2` vereinen die gesamte Logik der oben beschriebenen Funktionen analog zur mathematischen Beschreibung, um einen paarweisen Einsum-Ausdruck zu berechnen. Die beiden Varianten unterscheiden sich lediglich in der Anordnung der Tensor-Achsen und dem verwendeten Rechenkern. Die einzelnen Schritte der Pipeline sind im Flussdiagramm 4.6 der nächsten Seite dargestellt.

4.2.8 Der Pfad-Executor

Die Funktionen `einsum_v1` und `einsum_v2` sind die Schnittstelle, die von Python aufgerufen werden kann. Ihre Aufgabe ist es, eine Sequenz von binären `einsum`-Operationen, die durch den übergebenen Pfad definiert ist, schrittweise abzuarbeiten. Dafür werden zu Beginn die Eingabe-Tensoren aus dem `torch.sparse_coo_tensor`-Format in die interne `SparseTensor`-Struktur umgewandelt. Für jeden Schritt des Pfades wird die Kontraktion von der zugehörigen `perform_einsum` Variante berechnet und der zurückgegebene Ergebnistensor der Tensorliste hinzugefügt. Nachdem alle Schritte des Pfades abgearbeitet wurden, wird der letzte erzeugte Tensor zurück in ein PyTorch-kompatibles COO-Format (Koordinaten, Werte, Shape) konvertiert und zurückgegeben. Mit einem Wrapper wird daraus ein neuer `torch.sparse_coo_tensor` erzeugt.

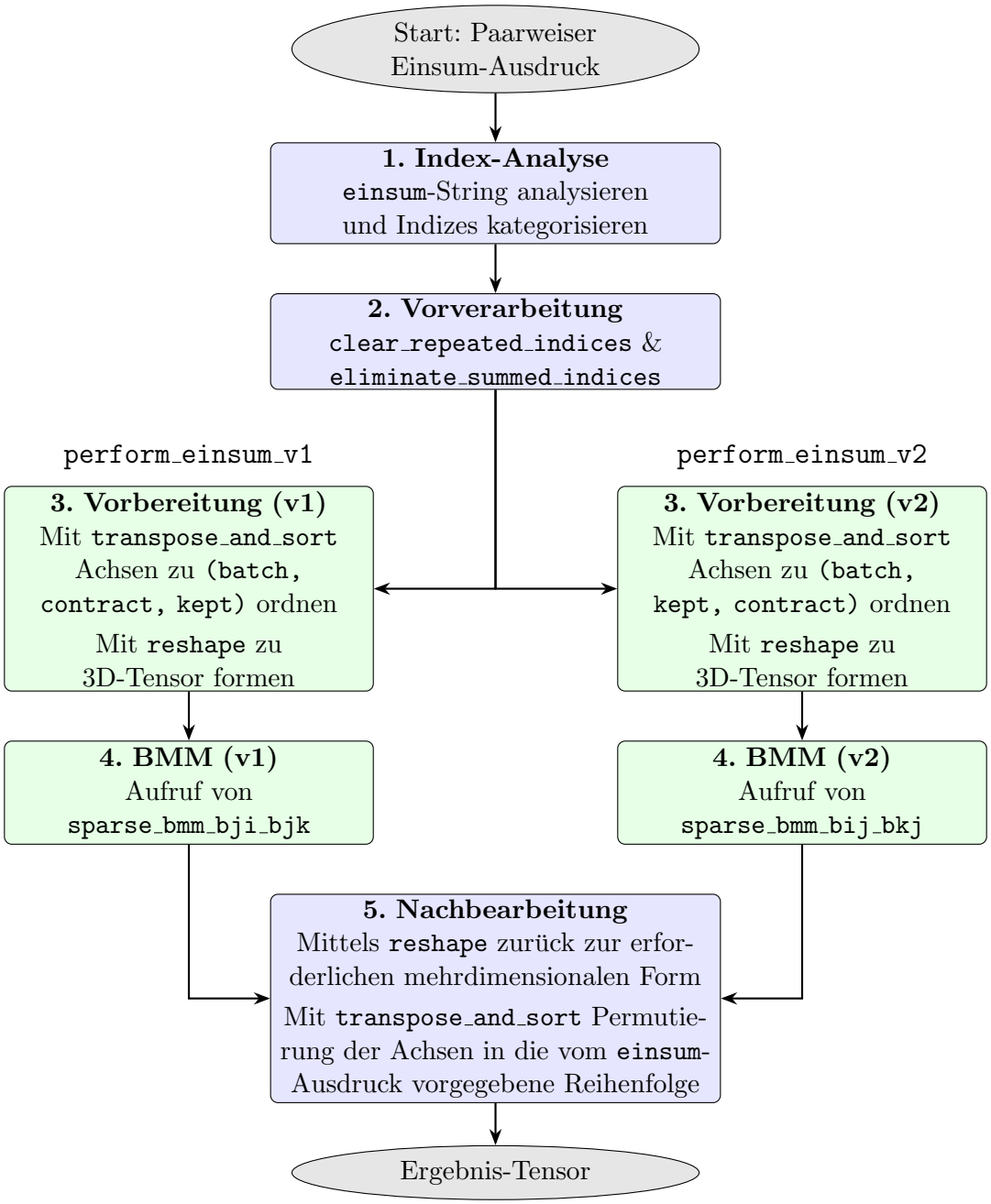

Abbildung 4.6: Die Pipeline der beiden `perform_einsum` Varianten

5 Experimente

In diesem Kapitel werden die beiden Versionen der Implementierung als v1 (Version mit Kernel $bji, bjk \to bik$) und v2 (Version mit Kernel $bij, bkj \to bik$) bezeichnet.

5.1 Versuchsaufbau

Um den entwickelten Algorithmus zu testen, führen wir zwei verschiedene Arten von Experimenten durch. In dem ersten Abschnitt wird verglichen, bis zu welcher Dichte unser Programm unter simplen Bedingungen mit dichten Implementierungen aus der State-Of-The-Art Bibliothek PyTorch konkurrieren kann. Dafür vergleichen wir im ersten Experiment dieses Abschnitts einen simplen BMM `einsum`-Ausdruck auf beiden Sparse-Implementierungen mit der torch.bmm Funktion. Dabei wird zudem analysiert, wie viel Laufzeit die einzelnen Schritte in der Verarbeitung unserer Implementierung benötigen, um eventuelle Flaschenhälse aufzuzeigen. Im zweiten Experiment dieses Abschnitts werden randomisiert generierte paarweise Einsum-Ausdrücke jeweils mit torch.einsum sowie beiden Versionen der Sparse-Implementierung berechnet, wobei wir auch hier untersuchen, bis zu welcher Dichte unsere Implementierung die dichte Version von PyTorch übertrifft. Im zweiten Abschnitt der Experimente werden ausgewählte Probleme des einsum_benchmark-Datensatzes berechnet. Das Ziel hierbei ist zu prüfen, ob der hier entwickelte Algorithmus auch auf realen Problemen eine Zeitersparnis im Vergleich zu PyTorch hervorrufen kann, sowie der direkte Vergleich zu der bereits existierenden Implementierung für dünnbesetzte Tensoren, `sparse.einsum`. Alle Experimente werden auf einem HP ProBook 640 G4 mit Intel Core i5–8250U (4 Kerne) durchgeführt. Das C++-Modul wird mit GCC 11 (`g++-11`, C++17) mit maximalen Optimierungsflags kompiliert, die Auswertung erfolgt in Python 3.10.12. Beachtet werden sollte, dass die bei der Kompilierung gesetzten Flags `-ffast-math` und `-Ofast` zu numerischer Instabilität führen und minimale Rundungsfehler verursachen können. Zufallszahlengeneratoren werden mit einem festen Seed initialisiert, um Reproduzierbarkeit der Eingabedaten sicherzustellen.

5.2 Ergebnisse

5.2.1 Synthetische Probleme

5.2.1.1 BMM

In diesem Experiment wird eine BMM mit zwei großen 3D-Tensoren mit shape (512, 512, 512) ausgeführt.

Dieses erste Experiment dient als Messung unter idealisierten Bedingungen. Die Tensoren besitzen gleichmäßige Dimensionen und die vorhandene Batch-Dimension stellt die Voraussetzung für eine effiziente Parallelisierung in den BMM-Kernen. Der einzig variierende Faktor ist die Dichte der Eingabetensoren. Ziel ist es, zu ermitteln, ab welchem Grad an Sparity sich die Nutzung einer spezialisierten Sparse-Verarbeitung gegenüber einer hochoptimierten dichten Implementierung wie `torch.bmm` lohnt.

Der annotierte SSA-Pfad für die BMM-Berechnung mittels `einsum` lautet [(0,1,“bij,bjk-¿bik”)]. Es werden Dichten zwischen 1e-4 und 6.4e-3 untersucht. Jede Messung sind gemittelte Werte über 10 Ausführungen pro Instanz bei 10 Instanzen pro Dichte. Die Ergebnisse sind in Abbildung 5.1 dargestellt.

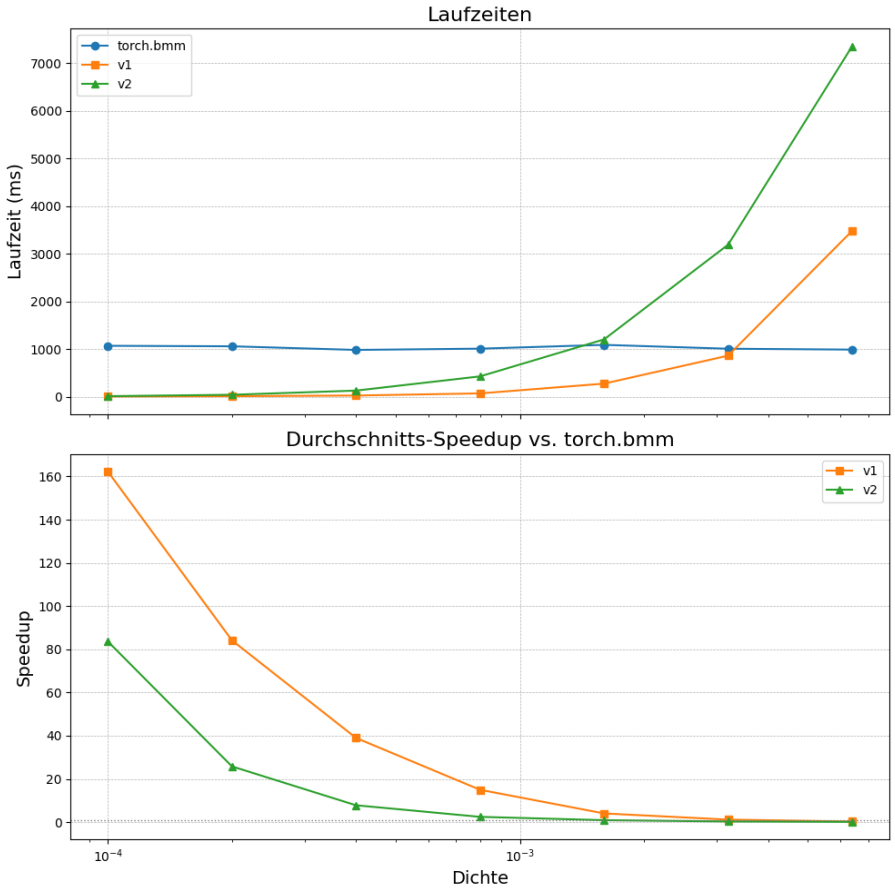

Abbildung 5.1: Durchschnittliche Laufzeiten und Speedups von v1 und v2 gegenüber torch.bmm

Wir beobachten sehr große Speedups bei niedrigen Dichten. Bei der geringsten getesteten Dichte von 1e-4 (0.01% Nicht-Null-Elemente) ist die `v1`-Variante unserer Implementierung im Durchschnitt 162.2-mal schneller als die dichte `torch.bmm`-Variante.

Dieser Vorteil nimmt mit steigender Dichte erwartungsgemäß ab, da die Anzahl der Nicht-Null-Elemente und somit der Rechenaufwand für die Sparse-Algorithmen wächst. Die Laufzeit von `torch.bmm` bleibt hingegen weitgehend konstant, da sie unabhängig von der Dichte immer die volle Matrix verarbeitet. Ab einer Dichte von etwas über 3.2e-3 (0,32% Nicht-Null-Elemente) kehrt sich das Verhältnis um, und die dichte Implementierung von

PyTorch wird schneller. Dieses Experiment zeigt deutlich das Potenzial des entwickelten Algorithmus für rechenintensive Operationen auf hochgradig dünn besetzten Tensoren in einem idealen Szenario.

Profiling der einzelnen Schritte Um die Performance der beiden Versionen des entwickelten Algorithmus besser zu verstehen und potenzielle Flaschenhälse zu identifizieren, ist eine detaillierte Analyse der einzelnen Verarbeitungsschritte unerlässlich. Das hier durchgeführte BMM-Experiment eignet sich dafür hervorragend, da es ein kontrolliertes und idealisiertes Szenario darstellt. Die Ergebnisse dieser Analyse sind in Abbildung 5.2 veranschaulicht. Der Teil "Rest" umfasst die summierte Laufzeit der Konvertierungen der Eingabetensoren in die `SparseTensor`-Struktur und der Konvertierung des Ergebnis-Tensors in das `torch.sparse_coo_tensor`-Format, sowie die Indexpartitionierung.

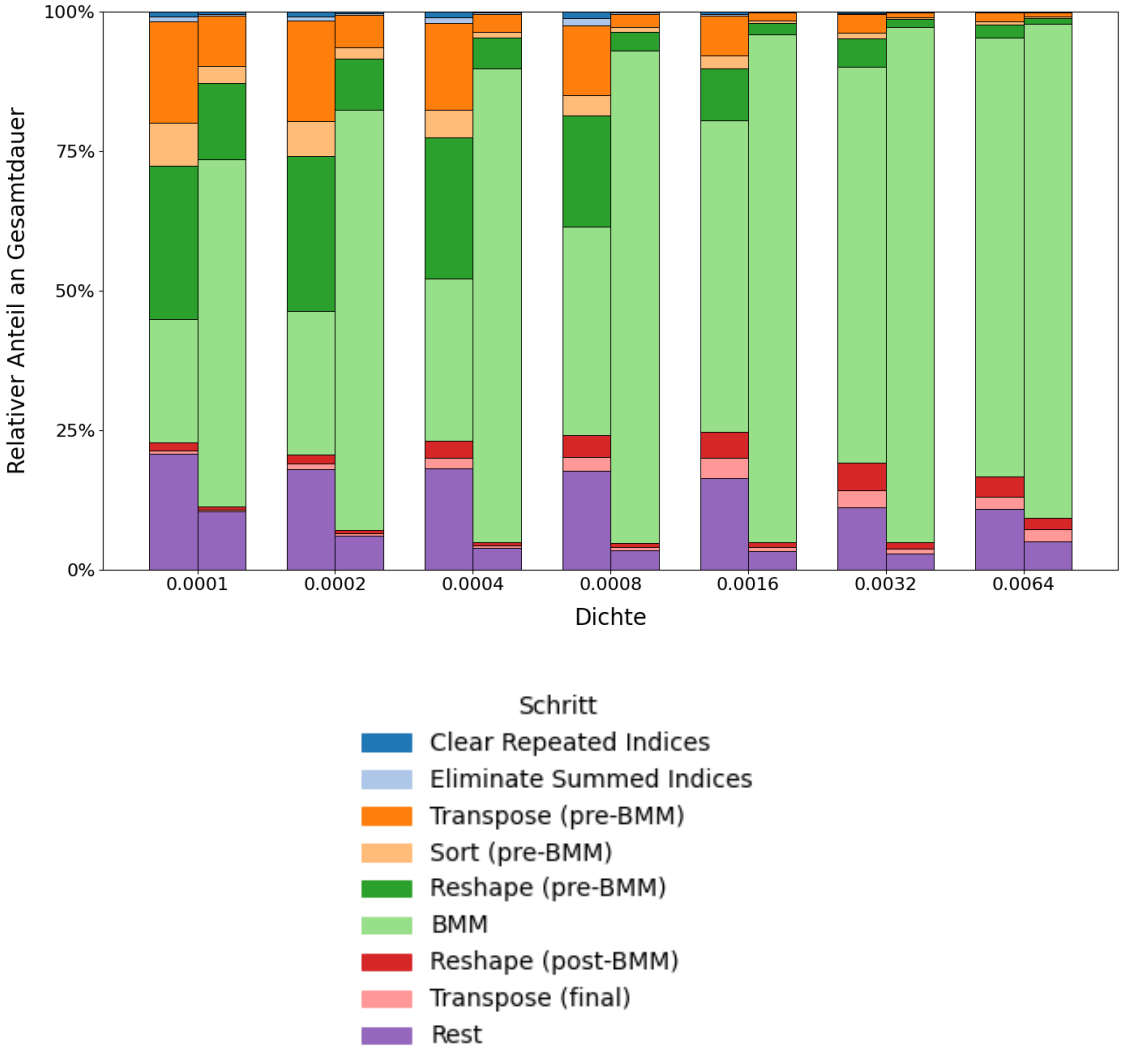

Abbildung 5.2: Relative Zusammensetzung der einzelnen Berechnungszeiten bei steigender Dichte

Da es in der vorliegenden Operation keine zu eliminierenden Diagonalen oder Summationsachsen gibt, verursachen die Funktionen `clear_repeated_indices` und `eliminate_summed_indices` kaum Kosten. Wir beobachten bei zunehmender Dichte, dass sich die BMM zum Flaschenhals in der Berechnung entwickelt. Nimmt diese bei einer Dichte von 0.0001 bei Implementierung `v1` nur 22% der Laufzeit ein, sind es bei der Dichte von 0.0064 schon 78%. Zudem können wir etwas Interessantes beobachten: Während die BMM in `v1` bei niedrigen Dichten deutlich schneller ist, skaliert diese bei höheren Dichten deutlich schlechter als v2 (Bei Verdopplung der Dichte von 0.0032 auf 0.0064 steigt die Laufzeit der v1-BMM ungefähr um den Faktor 4.5, die der v2-BMM jedoch nur ungefähr um den Faktor 2.2). Dieser Unterschied in Skalierung kann anhand der Implementierung erklärt werden: Der `v2` Kernel skaliert mit zunehmender Dichte besser, da er pro (b, i, k) lediglich einen Merge-basierten Skalarproduktschritt über die gemeinsame j-Dimension ausführt. Es werden keine großen Zwischenmengen berechnet. Demgegenüber erzeugt der `v1`-Kernel für jeden (b, j)-Block ein äußeres Produkt über $(i \times k)$ mit $O(nnz(A_{b,j,:}) \cdot nnz(B_{b,j,:}))$ intermediären Produkten, die anschließend per `unordered_map` aggregiert werden. Dieser Aufwand wächst mit der Dichte quadratisch je j-Block und lässt Zeit- und Speicherverbrauch bei höheren Dichten für diesen BMM-Kernel vergleichsweise explodieren. Bei niedrigen Dichten ist die BMM der v1-Implementierung jedoch deutlich schneller als die von v2.

5.2.1.2 Zufällige Einsum-Ausdrücke

Nachdem im ersten Experiment ein idealisiertes Szenario betrachtet wurde, zielt das zweite Experiment auf synthetischen Daten darauf ab, die Robustheit und Generalisierungsfähigkeit der entwickelten Sparse-Implementierungen zu überprüfen. Anstelle einer einzelnen, festen Operation wird nun anhand von 10 mittels `einsum_benchmark` zufällig generierten Hypernetzwerken die Laufzeit gemessen. Hierfür wurde die Methode `einsum_benchmark.generators.random.connected_hypernetwork` genutzt. Die Parameter wurden wie in Tabelle 5.1 gesetzt:

Tabelle 5.1: Gesetzte Parameterwerte der Funktion `connected_hypernetwork`.

Parameter	Wert
`number_of_tensors`	10
`regularity`	2
`max_tensor_order`	3
`max_edge_order`	2
`diagonals_in_hyper_edges`	False
`number_of_output_indices`	`random.randint(1,2)`
`max_output_index_order`	1
`diagonals_in_output_indices`	False
`number_of_self_edges`	0
`number_of_single_summation_indices`	`random.randint(0,2)`
`min_axis_size`	8
`max_axis_size`	64
`seed`	1-10
`global_dim`	False

Der Versuchsaufbau vergleicht die Laufzeiten von `torch.einsum` mit den beiden Sparse-Varianten (`sparse_einsum` und `sparse_einsum_v2`) über einen Dichtebereich von `1.0e-4` bis `1.28e-2`. Die gemittelten Laufzeiten und Speedups je Dichte sind in Abbildung 5.3 auf der nächsten Seite dargestellt.

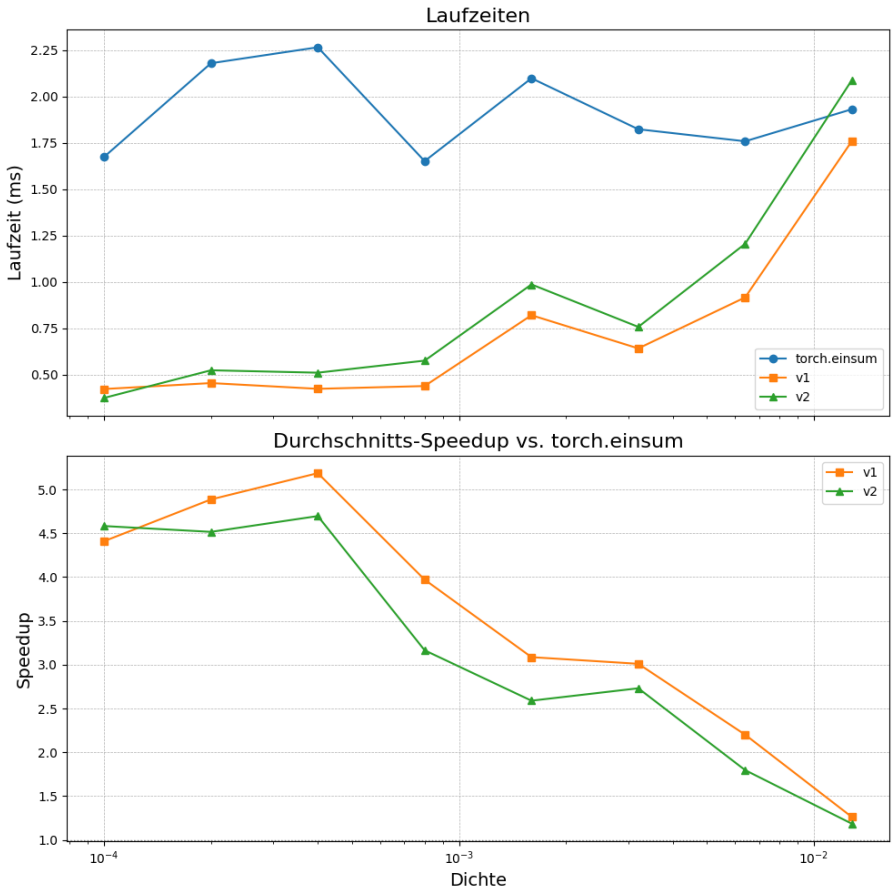

Abbildung 5.3: Durchschnittliche Laufzeit und Speedup von v1 und v2 gegenüber `torch.einsum`

Die Ergebnisse bestätigen die Beobachtungen aus dem ersten Experiment und verallgemeinern sie für eine breite Klasse von Problemen.

- **Variante 1:** Diese Implementierung erweist sich als äußerst leistungsfähig. Bei einer Dichte von `1.0e-4` beträgt die durchschnittliche Laufzeit nur rund 0.422 ms, womit sie etwa 4.4× schneller ist als `torch.einsum` (1.674 ms). Dieser Geschwindigkeitsvorteil bleibt über mehrere Dichtestufen hinweg bestehen, nimmt jedoch mit zunehmender Dichte deutlich ab. So liegt der Speedup bei `3.2e-3` noch bei rund 3.0x (0.641 ms vs. 1.823 ms). Der Kipppunkt, an dem der Overhead der Sparse-Verarbeitung die Vorteile überwiegt, wird knapp über der Dichte von `1.28e-2` erreicht, wo sich die Laufzeiten von v1 (1.760 ms) und `torch.einsum` (1.931 ms) sehr stark annähern.
- **Variante 2:** Die zweite Variante zeigt hier bis auf die erste Dichte von 1e-4 eine leicht schlechtere Performanz als v1, weist aber von dem Verlauf der Messwerte ein vergleichbares Verhalten zu v1 auf. Bei sehr geringer Dichte von `1.0e-4` liegt

die Laufzeit bei 0.374 ms (ca. 4.6x schneller als `torch.einsum`), während sie bei mittleren Dichten etwas hinter v1 zurückfällt (z. B. 0.986 ms vs. 0.820 ms bei `1.6e-3`). Auch die Skalierung bei höheren Dichten ist in diesem Experiment ähnlich: Während die Laufzeit von v1 bei der Verdopplung der Dichte von `6.4e-3` auf `1.28e-2` etwa 1.9x ansteigt (0.916 ms zu 1.760 ms), sind es bei v2 nur ungefähr 1.7x (1.205 ms zu 2.089 ms).

5.2.2 Reale Probleme aus dem `einsum_benchmark`-Datensatz

5.2.2.1 Vergleich von torch.einsum, sparse.einsum, sparse_einsum_v1 & sparse_einsum_v2

In diesem Experiment ist unser Ziel zu prüfen, ob der entwickelte Algorithmus auch auf reale Probleme angewendet werden kann. Hierfür wurde eine Auswahl aus spärlichen Problemen mit durchschnittlicher Dichte < 0.005 und Datentyp float64 aus dem einsum_benchmark Datensatz gewählt. Verglichen wird die Laufzeit zwischen der bereits existierenden dünnbesetzten Einsum-Implementierung aus der `sparse`-Bibliothek (sparse) mit den beiden Versionen der Implementierung (v1, v2) sowie, wenn möglich, mit `torch.einsum` (torch). Aufgrund der Größe und Komplexität der ausgewählten Instanzen aus dem einsum_benchmark Datensatz ist es in manchen Fällen für `torch.einsum` und `sparse.einsum` auf unserem Setup nicht möglich, diese Ausdrücke zu berechnen. Für Instanzen, bei denen die Berechnung aufgrund von Speicherüberschreitung abgebrochen wurde, wird dies mit "-" gekennzeichnet. Aufgrund der extrem ineffizienten Implementierung von sparse.einsum für komplexe Kontraktionen wird diese zur paarweisen Abarbeitung des im Voraus ermittelten Pfades genutzt, was eine optimierte Vorgehensweise ähnlich unser Implementierung für einen fairen Vergleich darstellt. Die Pfade wurden dem `einsum_benchmark`-Datensatz entnommen und nach Größe der Tensoren optimiert. Für jede Instanz wurde der Ausdruck 5x berechnet und die Mittelwerte notiert. Die Ergebnisse sind in Tabelle 5.2 zu sehen.

Tabelle 5.2: Vergleich der Laufzeiten verschiedener Einsum-Implementierungen in Sekunden

			Laufzeit (s)			
Instanz	**Tensoren**	**Avg. Sparsity**	`torch`	`sparse`	`v1`	`v2`
`mc_2022_087`	7345	0.11%	63.200	15.910	4.343	**4.324**
`mc_2021_036`	9553	0.05%	-	52.720	12.095	**11.431**
`mc_2022_025`	2900	0.45%	15.152	12.770	4.450	**2.802**
`wmc_2022_038`	4275	0.49%	12.644	13.277	5.157	**2.973**
`wmc_2023_036`	10 145	0.46%	129.049	137.708	45.500	**24.128**
`mc_rw_log-1`	3785	0.07%	-	55.076	23.467	**18.257**
`wmc_2021_145`	12 086	0.38%	-	-	151.840	**124.425**

Die Ergebnisse zeigen, dass die beiden Versionen unserer Implementierung durchgängig die niedrigste Laufzeit besitzen und auch Probleme berechnen können, die mit `torch.einsum` oder `sparse.einsum` auf unserem Setup aufgrund von Speicherüberläufen nicht berechenbar sind. In allen Instanzen erreicht diesmal v2 die beste Laufzeit. Gegenüber

`torch.einsum`, das in 3 von 7 Instanzen nicht in der Lage ist, das Problem zu berechnen, erreicht unsere Implementierung dort, wo `torch.einsum` terminiert, deutliche Beschleunigungen: Von etwa 5.34x (wmc_2023_036) bis zu etwa 14.62x (mc_2022_087). Auch gegenüber der optimierten `sparse.einsum`-Variante sind die Gewinne konsistent: Von einer durchschnittlichen Beschleunigung von etwa 3.02x bei der Instanz mc_rw_log-1, bis zu 5.7x bei der Instanz wmc_2023_036. Insgesamt kann bei diesem Experiment eine deutliche Verbesserung zu bisherigen Implementierungen nachgewiesen werden: Unsere Implementierung konnte ein Problem berechnen, bei dem sowohl die `torch`-, als auch die `sparse`-Einsum Implementierung scheiterte, sowie zwei weitere in denen ausschließlich `torch` scheiterte. In den Fällen in denen diese beiden Implementierungen Ergebnisse berechnen konnten, wurden deren Laufzeiten um einen Faktor von 3.02 bis 14.62 von unserer Implementierung unterboten.

6 Diskussion

Im Design der `SparseTensor`-Struktur wurde für die Vorteile des Koordinaten-Packings die Limitierung auf 128 Bits zur Speicherung aller Koordinaten eines Eintrags in Kauf genommen. Davon abgesehen, dass dieses Limit auch bei den komplexen Problemen im dritten Experiment nicht erreicht wurde, kann an diesem Punkt optimiert werden: In der aktuellen Implementierung wird für das Koordinaten-Packing immer ein 128-Bit-Integer verwendet. In vielen Fällen würden jedoch deutlich weniger Bits zur Darstellung reichen. In seltenen Fällen könnten möglicherweise mehr als 128 Bits nötig sein. Hier könnte man einen dynamischen Mechanismus implementieren, der den Datentyp zum Packen der Koordinaten bei der Initialisierung des `SparseTensors` in Abhängigkeit von der Shape wählt.

Im Profiling der einzelnen Schritte wurde ermittelt, dass bei hohen Dichten der Flaschenhals der Berechnung bei der BMM liegt. Somit ist der größte Hebel zur Verbesserung unserer Implementierung die Optimierung dieser beiden Rechenkerne. In der ersten Version der BMM mit `bji,bjk->bik`-Kernel kann das sequenzielle Zusammenführen der berechneten äußeren Produkte durch parallele Reduktion erheblich optimiert werden. Zudem gibt es in beiden Rechenkernen in der aktuellen Implementierung redundante Entpack-Operationen in inneren Schleifen, die durch temporäre Speicherung der entpackten Koordinaten-Vektoren eliminiert werden können. In naher Zukunft ist geplant, diese Optimierungsansätze zu implementieren und das bereits als Quellcode verfügbare Projekt in ein pip-installierbares Paket zu überführen. Auch eine kombinierte Version der Implementierungen mit heuristischer Kernelwahl ist eine Möglichkeit, das Projekt in Zukunft weiterzuentwickeln.

Im Profiling der einzelnen Schritte wurde sichtbar, dass das Post-BMM Reshape und die finale Transponierung bei beiden Versionen anfangs schneller als die pre-BMM Gegenstücke sind, was einerseits an Ausführung auf zwei Tensoren pre-BMM aber nur einem Tensor post-BMM, andererseits an der noch geringeren Dichte des Ergebnistensors bei niedrigen Dichten liegt. Bei einer Dichte über 0.0032 kehrt sich dieser Effekt um, ab dieser Schwelle scheint der Ergebnistensor mehr Elemente als die beiden Eingabetensoren zusammen zu umfassen, da Reshape und Transponierung nach der BMM länger dauert als davor.

In den synthetischen Experimenten war v1 außer bei der Dichte von 1e-4 im zweiten synthetischen Experimentin jeder Berechnung schneller als v2. Bei Experimenten auf dem `einsum_benchmark`-Datensatz konnte hingegen v2 in allen berechneten Instanzen die niedrigste Laufzeit erzielen. Wie in Kapitel 4.3 des `einsum_benchmark`-Papers beschrieben, starten die aus den Kategorien “Model counting” und “Weighed Model Counting” gewählten Probleme mit initial dichten Ausgangstensoren und können während der Auswertung hochgradig Sparse werden. Ein Erklärungsansatz für die beobachteten Ergebnisse ist, dass die dichten Berechnungen von dem v2-Kernel deutlich effizienter berechnet wurden, da dieser bei steigender Dichte besser skaliert. Die bessere Performanz der v1-Version auf

sehr dünnbesetzten Berechnungen wird vermutlich durch die deutlich schlechtere Laufzeit auf den dichten Berechnungen dieser Probleme ausgeglichen. Aufgrund der fehlenden Optimierung von v1 bei der Aggregation der Zwischenergebnisse kann nicht abschließend bewertet werden, ob die bessere Parallelisierung in der v1 BMM die komplexere Zusammenführung der Zwischenergebnisse wert ist. Bei Berechnungen auf ausschließlich dünnbesetzten Tensoren ist diese Variante aber zu bevorzugen.

7 Fazit

Diese Arbeit hat gezeigt, dass sich Einsum-Operationen auf dünnbesetzten Tensoren effizient auf eine sparse Batch-Matrixmultiplikation abbilden lassen, wenn die Eingabetensoren mit Vorverarbeitungsschritten, Permutation und Reshape in die für den Rechenkern benötigte Form transformiert werden. Zentrales Bauteil ist dabei die entwickelte SparseTensor-Struktur mit gepackten 128-Bit-Koordinaten, die Pointer-Overhead vermeidet und lexikographische Vergleiche von Koordinaten auf schnelle Integervergleiche reduziert. Die gesamte Pipeline skaliert in den Transformationsschritten über weite Bereiche nahezu linear, der dominierende Aufwand verlagert sich mit zunehmender Dichte erwartungsgemäß in den BMM-Schritt.

Mit den zwei spezialisierten BMM-Kernen v1 (`bji,bjk->bik`) und v2 (`bij,bkj->bik`) wurden komplementäre Stärken demonstriert: v1 ermöglicht eine besonders feine Parallelisierung über (b,j)-Blöcke und erzielt bei sehr geringer Dichte die höchsten Beschleunigungen, skaliert jedoch bei wachsender Dichte schlechter aufgrund des äußeren Produkts pro Block und der nachgelagerten Aggregation. v2 parallelisiert nur über die Batch-Dimension, vermeidet jedoch große Zwischenmengen und zeigt deshalb eine robustere Skalierung bei höheren Dichten. In synthetischen Experimenten übertrifft die Implementierung bei hoher Sparsity dichte Baselines deutlich. Für eine synthetische BMM können Beschleunigungen bis zu einer Dichte von 0.0032, für zufällig generierte paarweise `einsum`-Ausdrücke bis zu einer Dichte von 0.0064 erreicht werden, wobei v1 konstant die besten Laufzeiten liefert. Auf ausgewählten realen Instanzen aus dem `einsum_benchmark`-Datensatz wurden gegenüber `torch.einsum` und einer paarweise optimierten Variante von sparse.einsum konsistent Laufzeitgewinne erzielt. Hierbei liefert v2 die besten Laufzeiten, was an dichten Kontraktionen am Anfang der Berechnung liegt, welche die bessere Performanz auf sparsen Kontraktionen von v1 ausgleichen.

Die Grenzen der aktuellen Implementierung liegen vor allem im BMM-Kern und in der festen 128-Bit-Grenze des Koordinaten-Packings. Perspektivisch versprechen eine parallele Reduktion im v1-Kern, das Vermeiden redundanter Entpack-Operationen, eine dynamische Wahl der Bitbreite für das Koordinaten-Packing sowie Heuristiken zur automatischen Kernelwahl für die BMM weitere Beschleunigungen. Insgesamt belegen die Ergebnisse, dass die hier vorgestellte BMM-basierte Sparse-Einsum-Architektur in stark dünnbesetzten Anwendungsszenarien dichte Verfahren schlägt, wobei die bestehende Implementierung von `sparse.einsum` auch bei paarweiser Kontraktion nicht mit unserer Implementierung mithalten kann.

IV Literaturverzeichnis

1. Mark Blacher u. a. "Einsum Benchmark: Enabling the Development of Next-Generation Tensor Execution Engines". In: *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2024. URL: https://openreview.net/forum?id=tllpLtt14h.

2. Thorsten B. Wahl und Sergii Strelchuk. "Simulating Quantum Circuits Using Efficient Tensor Network Contraction Algorithms with Subexponential Upper Bound". In: *Physical Review Letters* 131.18 (2023). ISSN: 1079-7114. DOI: 10.1103/physrevlett.131.180601. URL: http://dx.doi.org/10.1103/PhysRevLett.131.180601.

3. Hy Truong Son und Chris Jones. *Graph neural networks with efficient tensor operations in CUDA/GPU and Graphflow deep learning framework in C++ for quantum chemistry*. 2019.

4. Jeffrey M. Dudek, Leonardo Dueñas-Osorio und Moshe Y. Vardi. *Efficient Contraction of Large Tensor Networks for Weighted Model Counting through Graph Decompositions*. 2020. arXiv: 1908.04381 [cs.DS]. URL: https://arxiv.org/abs/1908.04381.

5. Jean Kossaifi u. a. "Tensor contraction layers for parsimonious deep nets". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2017, S. 26–32.

6. Fredrik Berg Kjølstad. "Sparse tensor algebra compilation". Diss. Massachusetts Institute of Technology, 2020.

7. Zihao Ye u. a. "Sparsetir: Composable abstractions for sparse compilation in deep learning". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023, S. 660–678.

8. Ruiqin Tian u. a. "A High Performance Sparse Tensor Algebra Compiler in MLIR". In: *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2021, S. 27–38. DOI: 10.1109/LLVMHPC54804.2021.00009.

9. Intel Corporation. *Intel oneAPI Math Kernel Library (oneMKL)*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Version 2025.0, abgerufen am 2025-08-16. 2025.

10. Timothy A. Davis. *SuiteSparse: A Suite of Sparse Matrix Software*. https://github.com/DrTimothyAldenDavis/SuiteSparse. Abgerufen am 16.08.2025.

11. Pauli Virtanen u. a. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17.3 (2020), S. 261–272. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0686-2. URL: http://dx.doi.org/10.1038/s41592-019-0686-2.

12. Adam Paszke u. a. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703.

13. James Bradbury u. a. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

14. Charles R. Harris u. a. “Array programming with NumPy”. In: *Nature* 585 (2020), S. 357–362. DOI: 10.1038/s41586-020-2649-2.

15. Martín Abadi u. a. “TensorFlow: A System for Large-Scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016, S. 265–283.

16. PyData Development Team. *sparse: COO and DOK sparse arrays*. Version v0.13.0. 2021. DOI: 10.5281/zenodo.5136938. URL: https://doi.org/10.5281/zenodo.5136938.

17. Arian Kriesch. *Grafik zur dreidimensionalen Darstellung eines Levi-Civita-Symbols (auch Epsilon-Tensor)*. Wikimedia Commons. Ursprüngliche Version. Abgerufen am 21. August 2025. 2006. URL: https://commons.wikimedia.org/wiki/File:Epsilontensor.svg.

18. Sheela Orgler und Mark Blacher. *Optimizing Tensor Contraction Paths: A Greedy Algorithm Approach With Improved Cost Functions*. 2024. arXiv: 2405.09644 [quant-ph]. URL: https://arxiv.org/abs/2405.09644.

19. Leonardo Dagum und Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), S. 46–55.

20. Michael Axtmann u. a. *In-place Parallel Super Scalar Samplesort (IPS^{4}o)*. 2017. arXiv: 1705.02257 [cs.DC]. URL: https://arxiv.org/abs/1705.02257.

V KI-Quellenverzeichnis

Dokumentation von KI-Nutzung auf Basis der Freigabeerklärung

KI wurde in der vorliegenden Arbeit bei folgenden Aufgaben als Hilfestellung genutzt:

- Programmierung
- Schreiben
- Umformulieren
- Erstellung von Tikz-Bildern
- Übertragen von Messwerten

Verwendet wurden via GitHub Copilot die Modelle “GPT-4.1”, “GPT-5”, “Claude Sonnet 3.7 Thinking” sowie “Gemini 2.5 Pro”.

VI Anlagen

Tabelle 1: Aggregierte Laufzeiten und Speedups pro Dichte. Zeiten in Millisekunden. (Experiment 1a)

Dichte	torch.bmm	v1	v2	Speedup v1	Speedup v2
0.00010	1069.568	**6.935**	13.242	162.245	83.607
0.00020	1059.073	**13.014**	42.604	84.129	25.802
0.00040	983.326	**26.586**	129.989	38.981	7.815
0.00080	1009.844	**71.177**	428.071	14.982	2.452
0.00160	1088.924	**276.226**	1203.639	4.034	0.913
0.00320	1008.401	**862.881**	3190.264	1.173	0.319
0.00640	**989.859**	3480.577	7352.165	0.285	0.136

Tabelle 2: Durchschnittliche Schrittzeiten pro Dichte für sparse_einsum (v1) und sparse_einsum_v2 (v2). Werte in Millisekunden. (Profiling Experiment 1a)

Schritt	0.00010		0.00020		0.00040		0.00080		0.00160		0.00320		0.00640	
	v1	v2	v1	v2	v1	v2	v1	v2	v1	v2	v1	v2	v1	v2
Clear Repeated Indices	0.060	0.052	0.109	0.114	0.272	0.264	0.786	0.563	1.069	0.969	1.893	1.923	3.739	3.671
Eliminate Summed Indices	0.059	0.052	0.107	0.104	0.252	0.246	0.924	1.191	1.129	0.942	1.933	1.938	3.649	3.674
Transpose (pre-BMM)	1.265	1.182	2.331	2.512	4.141	4.267	8.979	10.383	19.661	18.105	28.302	28.542	53.931	55.701
Sort (pre-BMM)	0.531	0.403	0.828	0.885	1.329	1.353	2.609	3.568	6.177	4.793	9.706	9.578	18.254	18.761
Reshape (pre-BMM)	1.908	1.819	3.613	3.876	6.716	7.188	14.177	14.072	25.888	24.658	44.006	48.138	81.060	83.296
BMM	1.536	8.237	3.341	32.116	7.735	110.288	26.557	378.269	154.103	1095.004	611.459	2943.816	2739.683	6510.825
Reshape (post-BMM)	0.101	0.073	0.209	0.237	0.822	0.776	2.777	3.019	12.758	10.329	43.257	38.386	128.851	151.081
Transpose (final)	0.042	0.043	0.143	0.174	0.484	0.509	1.763	2.001	10.272	9.147	25.739	26.688	74.133	150.127
Rest	1.434	1.381	2.333	2.586	4.836	5.098	12.606	15.006	45.168	39.693	96.586	91.255	377.276	375.029
Gesamt	6.935	13.242	13.014	42.604	26.586	129.989	71.177	428.071	276.226	1203.639	862.881	3190.264	3480.577	7352.165

Tabelle 3: Gemittelte Laufzeiten und Speedups pro Dichte. Zeiten in Millisekunden. (Experiment 1b))

Dichte	torch.einsum	v1	v2	Speedup v1	Speedup v2
1.0e-04	1.674	**0.422**	0.374	4.41	4.58
2.0e-04	2.179	**0.454**	0.523	4.89	4.52
4.0e-04	2.265	**0.423**	0.510	5.19	4.70
8.0e-04	1.651	**0.438**	0.575	3.97	3.16
1.6e-03	2.098	**0.820**	0.986	3.09	2.59
3.2e-03	1.823	**0.641**	0.757	3.01	2.73
6.4e-03	1.758	**0.916**	1.205	2.20	1.80
1.28e-02	1.931	**1.760**	2.089	1.26	1.18

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Stand: Juli 2024

Freigabeerklärungen

Von der prüfenden Person konkret festzulegender Geltungsbereich
(*z.B. Thema, Veranstaltung, Zeitraum*):

Bachelorarbeit von Tim Lippert: Entwurf und Implementierung von Sparse-Einsum mittels Sparse-Batch-Matrixmultiplikation

Die folgenden Tools müssen nicht als Hilfsmittel deklariert werden und dürfen zur Erstellung von Seminar- und Abschlussarbeiten in ihrem standardmäßigem Leistungsumfang genutzt werden, auch wenn sie KI-gestützt sind:

- Textverarbeitungsprogramme, z. B. Word oder OpenOffice Writer
- Tabellenkalkulation, z. B. Excel oder LibreOffice Calc
- Rechtschreib- und Grammatikprüfung sowie -korrektur inkl. Werkzeugen in Textverarbeitungsprogrammen, z. B. DeepKomma
- Suchmaschinen
- Digitale Wörterbücher und Thesaurus
- Mindmap-Tools
- Recherchetools, z. B. wissenschaftliche Literatursuche via PubMed
- Recherchetools, die keine Ideen generieren, z. B. wissenschaftliche Literatursuche via Google Scholar
- *eigene Erweiterungen...*

Seminar- und Abschlussarbeiten sollen zuvorderst die Gedanken, Ideen und Erkenntnisse der Verfasserin bzw. des Verfassers beinhalten. Es muss klar erkennbar sein, ob und an welchen Stellen diese durch generierende KI ergänzt wurden. Im Folgenden finden Sie eine Auflistung der für die betroffene Prüfung erlaubten KI-Werkzeuge und wie deren Einsatz zu kennzeichnen ist. Die Dokumentation soll in einem separaten KI-Quellenverzeichnis erfolgen.

Erlaubte Werkzeuge:

- ☒ **Textgenerierende** KI-Werkzeuge: Die wörtliche oder inhaltliche Übernahme aus KI-generierten Textquellen (einschließlich Quellcodes, mathematischen Ausdrücken etc.) ist erlaubt.
- ☐ **Bildgenerierende** KI-Werkzeuge: Die direkte Übernahme aus KI-generierten Bildquellen ist erlaubt.
- ☐ **Bildverarbeitende** KI-Werkzeuge: Die direkte Übernahme aus Bildquellen, die mittels KI-Werkzeug weiterverarbeitet werden, ist erlaubt.
- ☐ **Übersetzung** durch KI-Werkzeuge: Die wörtliche Übernahme aus KI-generierten Übersetzungen ist erlaubt.
- ☐ *eigene Erweiterungen...*

Zur Dokumentation des KI-Einsatzes ist mindestens die Bezeichnung des verwendeten Werkzeugs und die Version erforderlich. Darüber hinaus sind die folgenden Angaben zu machen:

- ☐ Angabe der direkt übernommenen Generate des KI-Werkzeugs.
- ☐ Zeit und Datum der Nutzung des Werkzeugs.
- ☐ Die vollständige Eingabe in das Werkzeug, z. B. durch Prompts.
- ☐ Die vollständige Ausgabe des Werkzeugs.
- ☐ Falls vorhanden: Die Internetadresse, unter der das Werkzeug aufgerufen wurde.
- ☒ *eigene Erweiterungen...* Im Anhang der Arbeit angeben, welche Aufgaben mit KI-Textgenerierung bearbeitet wurden (z. B. Schreiben, Umformulieren, Übersetzen, Programmieren, Formatieren).

Bei der mehrstufigen Nutzung von KI-Werkzeugen in Form einer Sequenz von Überarbeitungen eines Inhalts sind alle Zwischenschritte einzeln zu dokumentieren. Dies, sowie Ein- und Ausgabe, können ggf. mit Links zu unveränderlichen Chatprotokollen sichergestellt werden.

20.08.2025

Datum — Unterschrift der prüfenden Person

Eigenständigkeitserklärung:

1. Hiermit versichere ich, dass ich die vorliegende Arbeit - bei einer Gruppenarbeit die von mir zu verantwortenden und entsprechend gekennzeichneten Teile - selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich trage die Verantwortung für die Qualität des Textes sowie die Auswahl aller Inhalte und habe sichergestellt, dass Informationen und Argumente mit geeigneten wissenschaftlichen Quellen belegt bzw. gestützt werden. Die aus fremden oder auch eigenen, älteren Quellen wörtlich oder sinngemäß übernommenen Textstellen, Gedankengänge, Konzepte, Grafiken etc. in meinen Ausführungen habe ich als solche eindeutig gekennzeichnet und mit vollständigen Verweisen auf die jeweilige Quelle versehen. Alle weiteren Inhalte dieser Arbeit ohne entsprechende Verweise stammen im urheberrechtlichen Sinn von mir.

2. Ich weiß, dass meine Eigenständigkeitserklärung sich auch auf nicht zitierfähige, generierende KI-Anwendungen (nachfolgend „generierende KI“) bezieht. Mir ist bewusst, dass die Verwendung von generierender KI unzulässig ist, sofern nicht deren Nutzung von der prüfenden Person ausdrücklich freigegeben wurde (Freigabeerklärung). Sofern eine Zulassung als Hilfsmittel erfolgt ist, versichere ich, dass ich mich generierender KI lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss deutlich überwiegt. Ich verantworte die Übernahme der von mir verwendeten maschinell generierten Passagen in meiner Arbeit vollumfänglich selbst. Für den Fall der Freigabe der Verwendung von generierender KI für die Erstellung der vorliegenden Arbeit wird eine Verwendung in einem gesonderten Anhang meiner Arbeit kenntlich gemacht. Dieser Anhang enthält eine Angabe oder eine detaillierte Dokumentation über die Verwendung generierender KI gemäß den Vorgaben in der Freigabeerklärung der prüfenden Person. Die Details zum Gebrauch generierender KI bei der Erstellung der vorliegenden Arbeit inklusive Art, Ziel und Umfang der Verwendung sowie die Art der Nachweispflicht habe ich der Freigabeerklärung der prüfenden Person entnommen.

3. Ich versichere des Weiteren, dass die vorliegende Arbeit bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt wurde oder in deutscher oder einer anderen Sprache als Veröffentlichung erschienen ist.

4. Mir ist bekannt, dass ein Verstoß gegen die vorbenannten Punkte prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass meine Prüfungsleistung als Täuschung und damit als mit „nicht bestanden“ bewertet werden kann. Bei mehrfachem oder schwerwiegendem Täuschungsversuch kann ich befristet oder sogar dauerhaft von der Erbringung weiterer Prüfungsleistungen in meinem Studiengang ausgeschlossen werden.

_____ _____

Ort, Datum Unterschrift