



SAPIENZA
UNIVERSITÀ DI ROMA

Implementation of the Einstein sum

Department of Computer Science
Applied Computer Science and artificial intelligence

Domenico Menale

Matricola 2049488

Relatore

Prof. Maria De Marsico

Correlatore

Dr. Diego Bellani

Anno Accademico 2025/2026

Tesi non ancora discussa

Implementation of the Einstein sum
Bachelor degree. Sapienza Università di Roma

© 2026 Domenico Menale. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: menale.2049488@studenti.uniroma1.it

*Da grandi poteri,
derivano grandi responsabilità.*

Sommario

This work explores the different possible implementation of the Einstein sum, a mathematical notation used in many fields of science, such as physics and engineering, to represent the summation of products of tensors. The Einstein sum is a powerful tool for simplifying complex mathematical expressions and can be used to derive important results in various fields. In this work, we will discuss the different ways to implement the Einstein sum, including its use in tensor calculus. We will also explore the advantages and disadvantages of each implementation.

Indice

1	Introduction to Einstein Summation and Tensor Calculus	1
1.1	The Nature of Tensors and the Einstein Convention	1
1.2	Einsum in the Architecture of Neural Networks	1
1.2.1	Linear Layers and Batch Processing	2
1.2.2	The Transformer and Multi-Head Attention	2
1.3	Broader Applications: From Fluids to Architecture	3
1.4	From Mathematical Notation to Low-Level Execution	3
1.4.1	The Compilation Pipeline	3
1.4.2	Execution Strategies and Algorithmic Diversity	3
1.5	Undecidability and the Necessity of Explicit Indices	4
1.5.1	The State Space of Contractions	4
1.5.2	Resolving Ambiguity	4
1.6	Conclusion	5
2	Implementations of the Einstein sum	7
2.1	Generalized Tensor Contraction via Odometer Iteration	7
2.1.1	The Technical Architecture	7
2.1.2	Execution and Performance Characteristics	8
2.1.3	Summary of the Generalized Engine	8
2.2	Just-In-Time (JIT) Code Generation and AOT Specialization	8
2.2.1	Optimization Mechanisms	9
2.2.2	Performance and Memory Trade-offs	9
2.2.3	Summary of the JIT Engine	9
2.3	Canonical Contraction through Permutation and Flattening	10
2.3.1	The Transformation Pipeline	10
2.3.2	Performance and Memory Trade-offs	11
2.3.3	Summary of the Permute Logic	11
2.4	BLAS-Accelerated Pattern Recognition and Greedy Dispatch	11
2.4.1	The Pattern Hierarchy	11
2.4.2	Intelligent Dispatch Logic	12
2.4.3	Future Work: Stride-Based Optimization	12
2.4.4	Summary of the BLAS Engine	12

Capitolo 1

Introduction to Einstein Summation and Tensor Calculus

1.1 The Nature of Tensors and the Einstein Convention

At the intersection of mathematics, physics, and computer science lies the concept of the **Tensor**. In its simplest form, a tensor is a multi-dimensional array of numerical values. While a scalar is a 0-order tensor, a vector is a 1st-order tensor, and a matrix is a 2nd-order tensor, the abstraction extends infinitely to N -th order tensors.

In the physical world—specifically in architectural engineering and structural analysis—tensors represent physical quantities that are independent of any specific coordinate system. For example, the stress tensor σ_{ij} describes the internal forces within a solid body. To calculate the interaction between these multi-dimensional objects, we require a robust notation: the **Einstein Summation Convention**.

Introduced by Albert Einstein in 1916, the convention simplifies tensor algebra by removing the explicit summation symbol (\sum). The rule is as follows: *whenever an index variable appears twice in a single term, it implies a summation over the entire range of that index.*

Consider the transformation of a vector v by a matrix M :

$$y_i = \sum_{j=1}^n M_{ij}v_j \quad \longrightarrow \quad y_i = M_{ij}v_j \quad (1.1)$$

This shorthand does not merely save space; it highlights the **contraction**—the process of reducing the dimensionality of the system by "contracting" shared indices.

1.2 Einsum in the Architecture of Neural Networks

In the modern computational landscape, tensors are the primary data structure for **Deep Learning**. Every operation in a neural network—from simple linear layers to complex attention mechanisms—is fundamentally a tensor contraction.

1.2.1 Linear Layers and Batch Processing

In modern deep learning, a Linear Layer is the most fundamental transformation. Mathematically, it represents an affine transformation of the input data. However, in a production environment, we rarely process a single data point at a time. Instead, we utilize **Batch Processing**, where multiple inputs are grouped into a single higher-order tensor to exploit the parallel processing power of GPUs.

Consider a weight matrix W with dimensions (I, O) , where I is the number of input features and O is the number of output features. If we process a single vector x of size I , the operation is a standard matrix-vector product. However, with a batch size B , our input becomes a 2D tensor $X \in \mathbb{R}^{B \times I}$.

Using the `einsum` notation, the operation is expressed as:

$$Y_{bo} = X_{bi}W_{io} \quad (1.2)$$

This specific notation provides several computational advantages:

- **Feature Mapping:** The index i is the "contraction dimension." It represents the internal connection where each input feature is weighted and summed to contribute to an output feature.
- **Batch Preservation:** The index b is a "broadcast index." It remains untouched by the summation, meaning the same weights W are applied identically to every sample in the batch.
- **Parallelism:** By representing the batch as a tensor dimension, the underlying `einsum` engine can dispatch the entire operation as a single **GEMM** (General Matrix Multiply) call.

Without the batch dimension, the hardware would have to launch B separate, smaller kernels, leading to significant overhead and under-utilization of the high-speed memory bus. Thus, the Einstein notation does not just describe the math; it defines the **memory access pattern** that makes large-scale neural network training feasible.

1.2.2 The Transformer and Multi-Head Attention

One of the most significant uses of `einsum` today is in the Transformer architecture. These models utilize 4-dimensional tensors to represent batches, sequence lengths, attention heads, and feature dimensions. A typical attention score calculation involves:

$$\text{Score}_{btsh} = Q_{bthd}K_{bshd} \quad (1.3)$$

Expressing this as a sequence of standard matrix multiplications would require multiple permutations and reshapes. `einsum` provides a declarative interface that allows the underlying engine to optimize the memory layout automatically.

1.3 Broader Applications: From Fluids to Architecture

The utility of `einsum` extends far beyond artificial intelligence. In **Continuum Mechanics**, the convention is used to describe the deformation of materials. In an architectural project the Einstein sum can be used to model elasticity tensors that describe how the building's concrete perimeter reacts to thermal expansion. In **Fluid Dynamics**, it can model wind pressure around the central courtyard public spaces using the Navier-Stokes equations in tensor form.

1.4 From Mathematical Notation to Low-Level Execution

The translation of a symbolic string like "`ij,jk->ik`" into machine code is a complex engineering task. The implementation must bridge the gap between high-level logic and **Low-Level Stride Arithmetic**.

1.4.1 The Compilation Pipeline

When a user calls an `einsum` function, the system undergoes several stages of translation:

1. **Parsing:** The string is analyzed to create bitmasks representing index occupancy.
2. **Contraction Mapping:** The engine identifies which indices are shared (summation) and which are unique (free).
3. **Memory Stride Calculation:** To access data in a flat memory buffer, the engine calculates the *stride*—the number of elements to skip to reach the next index in a given dimension.

1.4.2 Execution Strategies and Algorithmic Diversity

The implementation of a universal `einsum` engine requires a multi-tiered execution strategy. Because the Einstein notation is so expressive, no single algorithm can handle every case with peak efficiency. Instead, we present different implementations that fit different use cases, strarting with a general unoptimized algorithm and moving towards better performing options.

1. **Generalized Odometer Interpretation:** The most flexible approach is the Odometer-based iteration. This strategy treats the N -dimensional space as a single flat counter. It identifies every unique index in the notation (both free and summed) and builds a virtual "odometer" where each gear corresponds to a dimension size.

While this method is capable of handling arbitrary rank- N tensors without pre-compilation, it is limited by the overhead of coordinate-to-offset calculations performed inside the innermost loop. It serves as a study for the genuine implementation of the algorithm that is capable of handling any tensor.

2. **JIT-Code Generation and Specialization:** Another possible strategy is **Just-In-Time (JIT)** compilation. Instead of calling a library, the engine generates a raw C or C++ source file containing the exact nested loops required for the specific tensor shapes at hand.

By hardcoding the loop bounds and strides, the system allows the compiler to perform aggressive optimizations like **SIMD vectorization** and **Loop Unrolling**. This eliminates the "index lookup" overhead entirely, creating a zero-overhead kernel specialized for a single mathematical task.

3. **Canonical Permutation and Flattening:** By using a **Permute-Flatten** strategy we can treat the operation as a **matrix multiplication**. This involves reordering the axes of the input tensors so that the summation indices are grouped contiguously.

Once the data is rearranged, the multidimensional tensor is "flattened" into a 2D matrix view. This allows the system to treat a 4D or 6D contraction as a simple 2D GEMM operation, drastically reducing the complexity from $O(d^n)$ to $O(M \cdot K \cdot N)$, where M, K, N are the products of the respective dimension groups.

4. **BLAS Pattern Dispatching:** When treating 1D or 2D patterns (such as Dot, GER, or GEMV), the best solution is to route the data to **Basic Linear Algebra Subprograms (BLAS)**. These are hardware-vendor-tuned kernels (like Intel MKL or Apple Accelerate) that use sophisticated cache-blocking and tiling techniques.

By utilizing BLAS, we ensure that standard operations achieve the peak theoretical FLOPS (Floating Point Operations Per Second) of the host CPU or GPU.

1.5 Undecidability and the Necessity of Explicit Indices

A core theoretical problem in tensor computing is the ambiguity that arises without **Explicit Indexing**. In a coordinate-free notation, an expression like $A \otimes B$ is ambiguous.

1.5.1 The State Space of Contractions

Given two 3rd-order tensors, there are dozens of ways they could interact. Without explicit indices (e.g., " ijk,klm "), a computer cannot decide which dimensions should be summed and what the final dimension should be.

1.5.2 Resolving Ambiguity

Explicit indices serve as a **Schema**. They resolve the "Undecidability Problem" by providing a strict map of the data topology. Without this metadata, the engine would have to guess the user's intent, leading to non-deterministic results.

1.6 Conclusion

This work seeks to demonstrate that while the Einstein sum was invented as a simple way to express tensor contractions, its efficient implementation is the cornerstone of modern engineering. Whether we are optimizing a neural network or designing the structural perimeter of an architectural complex, the ability to rapidly contract tensors is what enables us to simplify complex computations in vast contexts.

By exploring these implementations, this work aims to provide a clear roadmap for building efficient tensor libraries and to contribute to the broader field of computational tensor calculus.

Capitolo 2

Implementations of the Einstein sum

This chapter presents the different implementations of the Einstein sum.

2.1 Generalized Tensor Contraction via Odometer Iteration

This foundational implementation provides a generalized approach to tensor contraction by treating the dimensions of the tensors as units of measure for an **Odometer**. This object allows the system to systematically iterate through all possible index combinations required by the Einstein summation notation. This method is built upon the principle of explicit summation, where every calculation is performed by manually traversing each coordinate of the tensors involved.

While this approach offers high clarity for educational purposes and absolute flexibility across tensor ranks, its computational complexity grows exponentially. For a tensor of rank n and dimension d , the total iterations required is d^n , making it a robust “one-size-fits-all” structure primarily suitable for small-to-medium datasets.

2.1.1 The Technical Architecture

The implementation relies on three core components to manage the complexity of multi-dimensional spaces: **Lexicographic Stepping**, **Bitwise Labeling**, and **Linear Offsetting**.

1. **Lexicographic Stepping (next_indices):** The core of the iteration logic is the `next_indices` function. This simulates a multi-dimensional counter that increments in row-major order.

- It identifies the least-significant dimension (the fastest-changing index).
- It increments the index until the maximum bound is reached, at which point it “carries” the increment to the next more-significant dimension.

This logic allows a single flat loop to simulate an arbitrary number of nested loops, enabling the code to handle any tensor rank dynamically.

2. **Bitwise Labeling (IndexBitmap):** To manage Einstein notation labels (e.g., 'i', 'j', 'k'), the system utilizes an `IndexBitmap`. This converts lowercase letters into a 26-bit integer.

$$\text{Bitmap} = \sum \text{bit}(c - \text{'a'})$$

This allows for nearly instantaneous membership tests. Finding common indices between two tensors is reduced to a simple bitwise AND operation, eliminating expensive string comparisons during the inner loops of the summation.

3. **Linear Offset Calculation:** Data is stored in a contiguous memory buffer following a row-major model. The `compute_offset` function translates a multi-dimensional coordinate into a single memory address using a Horner-like scheme:

$$\text{Offset} = \sum_{i=0}^{n-1} \left(\text{index}_i \cdot \prod_{j=i+1}^{n-1} \text{shape}_j \right)$$

While this provides $O(n)$ access time, the dynamic recalculation of these offsets during each iteration step accounts for the primary logic overhead of this approach.

2.1.2 Execution and Performance Characteristics

In the final execution stage, the `einsum` function creates a “combined iteration shape” encompassing every unique index in the notation. The odometer sweeps across this high-dimensional space, multiplying values and accumulating them into the result.

Feature	Odometer Implementation
Computational Efficiency	Moderate (significant logic overhead per iteration)
Memory Overhead	Very Low (no temporary tensor copies required)
Flexibility	Maximum (handles any rank/notation out-of-the-box)
Complexity Scaling	$O(d^n)$ iterations

Tabella 2.1. Performance characteristics of the Odometer-Based Generalized approach.

2.1.3 Summary of the Generalized Engine

The primary strength of this implementation is its **Source Anonymity**. Because it does not rely on hardcoded loops or specific patterns, it acts as a universal interpreter for Einstein notation. It serves as the essential fallback mechanism for the system when a contraction does not match optimized BLAS patterns or requires a level of dimensionality that exceeds the limits of specialized kernels.

2.2 Just-In-Time (JIT) Code Generation and AOT Specialization

This implementation represents a **Meta-Programming** approach to tensor operations. Instead of relying on a generalized function that interprets notation at

runtime, this system acts as a domain-specific compiler. It writes custom-tailored C source code for specific tensor shapes and notations, which is then compiled into a highly optimized binary kernel.

The primary shift here is the move from runtime interpretation to **Ahead-of-Time (AOT) Specialization**. By baking the geometry of the tensors directly into the instruction stream, we eliminate the CPU cycles typically wasted on managing loop metadata and index validation.

2.2.1 Optimization Mechanisms

The generator utilizes three primary optimization strategies: **Literal Stride Injection**, **Hierarchical Loop Ordering**, and **Compiler-Assisted Vectorization**.

1. **Literal Stride Injection:** In generalized functions, strides are looked up in arrays at runtime. This generator calculates these strides once and prints them as literal constants.

For an index “ij” with shape [7, 3], the generator outputs the direct arithmetic $(i * 3) + j$. This allows the C compiler to utilize **LEA (Load Effective Address)** instructions, calculating memory offsets in a single clock cycle.

2. **Hierarchical Loop Ordering:** The generator enforces a strict hierarchy to maximize **Data Locality**:

- **Outer Shell:** Iterates over the Free Indices (those present in the output), ensuring a contiguous and organized traversal of the destination memory.
- **Inner Core:** Iterates over Summation Indices.

This structure maximizes **Temporal Locality** by keeping the accumulation sum for a specific output element within a high-speed CPU register for the entire duration of the summation loop.

3. **Compiler-Driven Vectorization:** By outputting high-level C code rather than raw assembly, the implementation leverages the full power of modern optimizers (GCC/Clang).

Because loop bounds (e.g., $i < 7$) are known at compile time, the compiler can perform aggressive **Loop Unrolling** and automatically inject **SIMD (Single Instruction, Multiple Data)** instructions to process multiple floating-point operations in parallel.

2.2.2 Performance and Memory Trade-offs

2.2.3 Summary of the JIT Engine

This approach treats Einstein notation as a high-level language, translating it into the most efficient mathematical representation possible for the underlying hardware. By moving the computational burden from the runtime environment to the **Compilation Phase**, it minimizes logic overhead and memory latency, making it adapt for high-performance production environments where tensor shapes are known in advance.

Feature	JIT Code Generation
Computational Efficiency	Extreme (hardware-level optimization)
Memory Overhead	Minimal (no temporary tensors, only source code strings)
Flexibility	Low (requires a new function for every shape change)
Complexity Handling	Best for static, performance-critical production kernels

Tabella 2.2. Performance characteristics of the JIT-based specialization approach.

2.3 Canonical Contraction through Permutation and Flattening

This implementation introduces a mathematical transformation designed to bridge the gap between abstract tensor notation and high-performance linear algebra. Rather than iterating through a high-dimensional space with a generalized odometer, this approach transforms any arbitrary N -dimensional Einstein summation into a standard 2D matrix multiplication ($C = A \times B$).

By reducing the problem to a canonical matrix form, the system can leverage highly optimized hardware routines while maintaining the flexibility to handle complex tensor geometries.

2.3.1 The Transformation Pipeline

The core logic follows a three-stage pipeline: **Permute, Flatten, and Multiply**.

1. **Tensor Permutation and Grouping:** To treat a tensor as a matrix, the memory layout must be rearranged so that relevant dimensions are contiguous. The function `matrix_permute` reorders the axes of the source tensors:

- **Tensor A** indices are grouped into [Free Indices, Summation Indices].
- **Tensor B** indices are grouped into [Summation Indices, Free Indices].

This grouping ensures that the “right side” of Tensor A (the dimensions being summed over) aligns perfectly with the “left side” of Tensor B.

2. **Flattening into 2D Matrices:** Once dimensions are grouped, the implementation calculates the product of the shapes in each group to determine the effective number of rows and columns. Let I be the set of free indices in A , and K be the set of shared summation indices. Tensor A is reshaped into a matrix of size $M \times L$, where:

$$M = \prod_{i \in I} \text{shape}(i) \quad \text{and} \quad L = \prod_{k \in K} \text{shape}(k)$$

Similarly, Tensor B is reshaped into a matrix of size $L \times N$, where N is the product of the free indices in B .

3. **Optimized Multiplication and Reshaping:** With the tensors transformed into 2D matrices, the code performs a standard accumulation:

$$C_{ij} = \sum_{k=1}^L A_{ik} \cdot B_{kj}$$

The result initially exists in a canonical order where all free indices from A appear before those from B . A final `matrix_permute` call is performed on the intermediate result to match the specific output notation requested by the user.

2.3.2 Performance and Memory Trade-offs

Feature	Permute-Flatten Implementation
Computational Efficiency	Very High (utilizes linear matrix multiplication)
Memory Overhead	Significant (requires temporary permuted copies)
Complexity Handling	Excellent for large contractions $O(M \cdot L \cdot N)$
Cache Behavior	Optimized via contiguous linear writes

Tabella 2.3. Performance characteristics of the Canonical Contraction approach.

2.3.3 Summary of the Permute Logic

The helper function `matrix_permute` acts as the engine of this implementation. It maps the linear index of the destination tensor back to the coordinates of the source tensor. By iterating through the **destination** linearly, it ensures that memory writes are contiguous, which is vital for utilizing the CPU's write-combine buffers and maximizing cache line efficiency.

2.4 BLAS-Accelerated Pattern Recognition and Greedy Dispatch

This implementation utilizes a **Greedy Pattern Matching** strategy to map Einstein notation to the highly optimized Basic Linear Algebra Subprograms (**BLAS**) library. Instead of treating every contraction as a generic tensor operation, this system attempts to identify standard linear algebra archetypes that can be executed using hardware-tuned Level-1, Level-2, or Level-3 BLAS routines.

By selecting the most specific applicable pattern, the engine maximizes performance through superior cache locality and reduced memory traffic, providing a significant speedup over generalized methods.

2.4.1 The Pattern Hierarchy

The implementation utilizes a hierarchy of functions, prioritizing specialized kernels over more general ones to minimize computational overhead.

1. Level-1 Operations (Vector-Vector):

- **DOT:** Detects patterns like " $i, i \rightarrow$ " to perform a scalar dot product.
- **AXPY:** Recognizes " $i, i \rightarrow i$ " for constant-multiplication and vector addition ($y := \alpha x + y$).

2. Level-2 Operations (Matrix-Vector):

- **GER:** Identifies " $i, j \rightarrow ij$ " to perform a rank-1 outer product.
- **GEMV:** Matches " $ij, j \rightarrow i$ " or " $i, ij \rightarrow j$ " to execute matrix-vector multiplication, supporting both standard and transposed variants.

3. Level-3 Operations (Matrix-Matrix):

- **GEMM:** The most critical kernel, matching " $ij, jk \rightarrow ik$ ". This represents the gold standard of optimization, where $O(n^3)$ operations are performed on $O(n^2)$ data, allowing for maximum data reuse.

2.4.2 Intelligent Dispatch Logic

The system analyzes the input notation using bitwise masks and dimension counts to fill a **BLASAnalysis** metadata structure. This structure guides the **Main Dispatch Function**, which routes the matrices to specific case handlers.

Pattern	BLAS Level	Mathematical Operation
DOT	Level 1	Scalar Product: $s = \sum x_i y_i$
AXPY	Level 1	Vector Accumulation: $y = \alpha x + y$
GER	Level 2	Outer Product: $A = xy^T$
GEMV	Level 2	Matrix-Vector: $y = Ax$
GEMM	Level 3	Matrix-Matrix: $C = AB$

Tabella 2.4. The greedy dispatch hierarchy used for BLAS acceleration.

2.4.3 Future Work: Stride-Based Optimization

A unique feature of this implementation is the theoretical groundwork for **Stride-Based Patterns**. While the current version handles unit strides, standard BLAS functions accept an `incx` parameter.

In future iterations, this would allow the system to handle complex patterns like " $ii \rightarrow$ " (Matrix Trace) or " $xii, xii \rightarrow$ " (Diagonal Dot Product) without moving data. By calculating a custom stride based on the tensor shape, the engine could traverse diagonal elements as if they were a contiguous vector, further reducing the need for temporary memory allocations or tensor permutations.

2.4.4 Summary of the BLAS Engine

This approach transforms Einstein notation from a mathematical abstraction into a sequence of high-performance library calls. By identifying the most specific “building block” for a given operation, the engine ensures that simple operations remain fast while complex matrix multiplications reach the peak theoretical performance of the hardware.