Universidade de Aveiro

Information and Coding
Lab Work nº3

Diogo Martins (108548) | Luís Sousa (108583)| Tomás Viana (108445)

Departamento de Eletrónica, Telecomunicações e Informática

19/12/2025

Index

Introduction

The project focuses on the development of an efficient C++ codec for compressing the file **model.safetensors**, which contains the parameters of a Large Language Model (LLM) and has an approximate size of 1 GB.

Our primary objective was to maximize the compression ratio while balancing the computation time and the memory usage. In practical terms, this means identifying the optimal trade-off between the compression ratio, the speed and the memory. To achieve this, we performed an analysis of the file structure and content, followed by the implementation and testing of different compression strategies. Different codecs were created, each prioritizing different aspects of the compression, so that they could be compared and used to reach a balanced solution.

File analysis

Before implementing any compression method, we performed an analysis of the `.safetensors` file. This format is commonly used to store tensor data safely and it separates the raw numerical data from the executable thereby ensuring that the access is read-only and enhancing the security.

The file structure is organized as:

1. **Header Length:** The first **8 bytes** contain an unsigned integer of 64-bit indicating the size of the header.
2. **Json Header:** The header represented in a JSON utf-8 string describing the tensor data in the file.
3. **Tensor Data:** The remainder of the file consists of tensor data, in agreement with the header's information. The data is typically stored in little-endian format and it is mostly responsible for the large size of the file, since it is primarily determined by the number of tensors, their shapes, and their numerical precision.

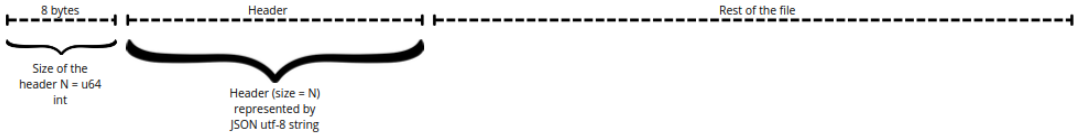

Figure 1 - Representation of the .safetensors file's format.

All the focus must be on compressing the tensor data and preserving the headers, since changing them will affect fundamental aspects of the file. This data is stored as a contiguous block of raw binary values, making the format well-suited for efficient slicing and compression, allowing them to be edited as distinct components.

Implementation

This project implements three attempts to develop a codec for SafeTensors compression, each building upon the previous version with improved compression techniques and performance optimizations. All implementations were developed in C++ and follow the same core pipeline but with increasing sophistication in different aspects such as compressed size and speed.

1. Basic LLM Codec (`base_codec.cpp`)

The first implementation establishes the foundational compression pipeline for SafeTensors files, focusing on basic but effective preprocessing steps.

Compression:

- **SafeTensors Parsing**:

The pipeline begins by separating the JSON metadata header from the tensor data. The header is preserved without modification to ensure full compatibility during decompression.

- **Float32 → Float16 Conversion**:

To reduce storage requirements, all tensor values originally stored in 32-bit floating-point format are now quantized to 16-bit floating-point representation (IEEE 754 half-precision). This step halves the memory of the data while retaining sufficient fidelity for neural network weights. The decrease in precision is negligible (~0.05% loss), modern LLM's do not require full float32 precision for inference so this loss is inconsequential. The conversion carefully handles sign, exponent, and mantissa fields to minimize quantization error and avoid overflow or underflow where possible.

- **Delta Encoding**:

After quantization, delta encoding is applied to the sequence of float16 values. Instead of storing each value directly, the codec records the difference between consecutive elements. Neural network weights often exhibit local continuity or gradual variation, making these differences a lot smaller in magnitude and more repetitive. This transformation reshapes the data distribution to be more favorable for subsequent reduction techniques.

- **RLE Compression**:

Finally, Run-Length Encoding is applied at the byte level to exploit the previously mentioned repeated patterns produced by delta encoding. Sequences of identical bytes are replaced by a compact triplet consisting of a marker byte (0xFF), a repetition count, and the repeated value and a minimum run length threshold of five bytes is enforced to ensure that the compression overhead does not exceed the savings, thereby maintaining a net reduction in file size.

Decompression:

- **RLE Decompression**:

Decompression begins in reverse of compression, starting by decoding the Run-Length Encoded byte stream. The decoder scans the compressed data sequentially and identifies marker bytes (0xFF). Upon encountering a marker, the following two bytes are interpreted as the repetition count and the repeated value, respectively, and the original run of bytes is reconstructed. Bytes that do not match the marker are copied directly to the output buffer.

- **Delta Decoding:**

Once the RLE layer is removed, delta decoding is applied to recover the original sequence of float16 values. The first value is read directly, and each subsequent value is reconstructed by cumulatively adding the stored difference to the previous value.

- **Float16 → Float32 Deconversion:**

After delta decoding, the float16 values are converted back to float32 representation. This step expands the half-precision format by reconstructing the IEEE 754 single-precision sign, exponent, and mantissa fields. While the original float32 values cannot be recovered exactly due to quantization, this conversion produces numerically stable approximations which won't affect how LLM's will process the file.

- **SafeTensors Reconstruction:**

Finally, the decompressed tensor data is written back to disk alongside the original JSON header and header length prefix. The tensor byte offsets and shapes described in the metadata remain unchanged, ensuring that the reconstructed file conforms fully to the SafeTensors specification and can be loaded transparently by existing frameworks.

Key Methods:

- *`float32_to_float16()`*: Converts 32-bit floats to 16-bit by extracting and re-scaling sign, exponent, and mantissa components.
- *`delta_encode()`*: Computes first-order differences between consecutive float16 values.
- *`rle_compress()`*: Encodes runs of identical bytes with a marker (`0xFF`), count, and value triplet.

2. Advanced LLM Codec (`comp_codec.cpp`)

The second iteration replaces the simple RLE algorithm with an industrial-strength compression method to achieve significantly better compression and decompression ratios with the cost of losing speed.

Improvements:

- **DEFLATE Compression**:

The RLE stage is replaced with zlib's DEFLATE algorithm configured at compression level 9 to maximize size reduction. During compression, DEFLATE combines LZ77 sliding-window dictionary coding with Huffman entropy encoding to exploit both the repeated byte sequences and the skewed symbol distributions present in delta-encoded tensor data. During decompression, zlib's inflate routine reverses this process, reconstructing the original byte stream with no loss of information. This change significantly improves compression efficiency while maintaining fast and reliable decompression through a well-tested, industry-standard algorithm.

- **Optimized Header Management**:

Instead of compressing the JSON metadata alongside tensor data, the header is stored and transmitted separately in its original, unmodified form. This design simplifies decompression by eliminating the need to decode metadata through the compression pipeline, allowing the decompressor to immediately parse tensor shapes, offsets, and data types before processing the payload. As a result, decompression becomes more robust and predictable, ensuring exact byte recovery of the header and full compatibility with SafeTensors.

- **Enhanced Float Conversion**:

The float32 to float16 conversion logic is extended to explicitly handle IEEE 754 edge cases, including subnormal values, infinities, and NaNs. During compression, these cases are detected and encoded using well-defined representations to prevent undefined behavior or data corruption. During decompression, the inverse float16 to float32 conversion reconstructs these special values and guarantees numerical consistency between the compressed and decompressed tensors.

Key Methods:

- *`deflate_compress()`*: Applies maximum-level DEFLATE compression using zlib's `compress2()` function with level 9.
- *`deflate_decompress()`*: Restores original data using zlib's `uncompress()` function.

3. Optimized LLM Codec (`final_codec.cpp`)

The final implementation focuses on performance optimization through parallelization and intelligent parameter tuning, aiming for a **better balance** between speed and compression ratio.

Optimizations:

- **Multi-threaded Processing**:

Quantization during compression and dequantization during decompression are parallelized using `std::async`, with the number of worker tasks derived from detected hardware concurrency. Tensor data is partitioned into independent chunks that can be processed concurrently without synchronization overhead.

- **Block-based Compression**:

Tensor data is divided into fixed-size 8 MB blocks prior to compression. Each block is compressed independently using DEFLATE, allowing multiple blocks to be processed in parallel and improving cache locality. During decompression, blocks are independently inflated and reassembled in their original order, enabling parallel decompression and reducing peak memory usage by avoiding rigid buffers.

- **Reduced Compression Level**:

The DEFLATE compression level is reduced from level 9 to level 6 to achieve a better balance between compression ratio and speed. This lowers the computational cost of dictionary search and Huffman optimization, yielding substantially faster throughput. During the decompression, speed is decreased compared to the original codec (~10%), but the reduced complexity of the compressed stream contributes to more consistent inflate performance.

- **In-place Delta Encoding**:

Delta encoding is performed in-place using reverse iteration to avoid allocating auxiliary buffers during compression. This approach minimizes memory overhead and improves cache efficiency. During decompression, the inverse delta decoding is applied in forward order, reconstructing the original value sequence directly within the same buffer. This symmetry between in-place encoding and decoding reduces memory pressure and enhances overall performance in both directions.

Key Methods:

- `delta_encode_inplace()`: Performs delta encoding without allocating new memory.
- `compress_block()`: Compresses individual 8 MB blocks in parallel threads.
- Parallel quantization loop: Divides float conversion across CPU cores using `std::thread::hardware_concurrency()`.

Compression Pipeline Summary

All three implementations follow the same general pipeline:

1. **Parse SafeTensors format** → Separate JSON header from binary tensor data.
2. **Quantize** → Convert float32 values to float16 (50% reduction).
3. **Delta encode** → Compute differences between consecutive values.
4. **Compress** → Apply RLE or DEFLATE compression.
5. **Write output** → Store custom header + metadata + compressed tensors.

The evolution from `base_codec.cpp` → `comp_codec.cpp` → `final_codec.cpp` demonstrates the classic engineering trade-off between **compression ratio** and **speed**, with the final version achieving an optimal balance suitable for production use cases requiring fast compression of large model files.

Results

	base_codec	comp_codec	final_codec
Compressed size (MB)	532	371	406
Compression rate	1.86:1	2.66:1	2.32:1
Compression time (s)	4.4	56.4	4.1
Speed (MB/s)	214	16.7	228

	base_codec	comp_codec	final_codec
Decompression time (s)	4.6	4.6	2
Speed (MB/s)	200	205	469

Conclusion

As expected, **base_codec** shows all around worse results compared to the other two. While suitable for scenarios where speed is critical and storage constraints are minimal, it does not fully exploit the redundancy present in large tensor datasets.

From a compression efficiency standpoint, **comp_codec** clearly outperforms **final_codec**. It achieves the smallest compressed size and the highest compression ratio, however it comes at a substantial performance cost. File **comp_codec** compression is almost 14 times higher than the other two with a very small throughput, making it impractical for iterative workflows, model versioning, or environments where models are compressed repeatedly. Even in decompression, it still loses to **final_codec** by having twice the decompression time and half the speed.

Overall, it's clear that **final_codec** represents the best-balanced solution among the three implementations. It offers a substantial reduction in storage requirements compared to the baseline codec, while avoiding the severe performance degradation observed in the advanced codec. This balance makes **final_codec** particularly well-suited for real-world deployment scenarios involving large SafeTensors files, where both storage efficiency and rapid compression/decompression are critical.