Informação e Codificação

Relatório do Lab work nº 3

Eduardo Alves: nºmec 104179
Bernardo Marujo: nºmec 107322
Simão Almeida: nºmec 113085

eduardoalves@ua.pt
bernardomarujo@ua.pt
spsa@ua.pt

Link Repositório GitHub

Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2025

Contents

1 Introdução - Análise do Problema **2**
- 1.1 Estrutura do Formato Safetensors . . . 2
- 1.2 Distribuição de Tipos de Dados . . . 2
- 1.3 Análise de Entropia . . . 2
  - 1.3.1 Distribuição de Entropia em Redes Neuronais . . . 3
  - 1.3.2 Implicações para Compressão . . . 3

2 Solução Implementada **3**
- 2.1 Arquitetura da Solução . . . 3
- 2.2 Pré-Processamento: Bit-Shuffling . . . 4
  - 2.2.1 Algoritmo de Shuffling . . . 4
  - 2.2.2 Reversibilidade . . . 4
- 2.3 Algoritmo de Compressão: Zstandard . . . 5
  - 2.3.1 Escolha do Algoritmo . . . 5
  - 2.3.2 Configuração . . . 5
- 2.4 Gestão de Memória: Processamento por Blocos . . . 6
  - 2.4.1 Problema . . . 6
  - 2.4.2 Solução: Streaming por Blocos . . . 6
- 2.5 Implementações Disponíveis . . . 7
  - 2.5.1 Implementação Sequencial (`main.cpp`) . . . 7
  - 2.5.2 Implementação Paralela com OpenMP (`bf16_omp.cpp`) . . . 7
- 2.6 Formato do Ficheiro Comprimido . . . 8
  - 2.6.1 Estrutura Global . . . 8
  - 2.6.2 Cabeçalho . . . 8
  - 2.6.3 Metadados de Cada Bloco . . . 9
- 2.7 Implementação das Operações . . . 9
  - 2.7.1 Operação de Compressão . . . 9
  - 2.7.2 Operação de Descompressão . . . 10

3 Utilização **11**
- 3.1 Compilação . . . 11
- 3.2 Interface de Linha de Comandos . . . 11
  - 3.2.1 Compressão . . . 11
  - 3.2.2 Descompressão . . . 11
- 3.3 Teste Automatizado . . . 11

4 Resultados e Análise de Desempenho **12**
- 4.1 Desempenho da solução sequencial . . . 12
- 4.2 Resultados (OpenMP) . . . 12
  - 4.2.1 Desempenho da implementação paralela (OpenMP) . . . 12
- 4.3 Comparação entre a versão básica e a versão OpenMP . . . 13
- 4.4 Teste de compressores existentes . . . 13

1 Introdução - Análise do Problema

O objetivo principal deste projeto é desenvolver e implementar uma estratégia eficaz para a compressão do ficheiro `model.safetensors`. O armazenamento eficiente de Large Language Models (LLMs) representa um desafio atual significativo, uma vez que estes ficheiros são constituídos maioritariamente por parâmetros numéricos e não por texto convencional.

1.1 Estrutura do Formato Safetensors

Com o intuito de delinear a estratégia de compressão mais eficiente, procedeu-se à análise detalhada da estrutura e do conteúdo do ficheiro `model.safetensors`, para compreender a natureza dos dados e identificar redundâncias que pudessem ser exploradas. Foi desenvolvido um *script* Python (`analyze_safetensors.py`) para decompor e inspecionar a estrutura binária do formato. O ficheiro `model.safetensors` organiza-se em três componentes principais:

1. **Tamanho do cabeçalho** (8 bytes): Um inteiro sem sinal de 64 bits (`uint64_t`) em formato *little-endian* que identifica o tamanho da secção seguinte.
2. **Cabeçalho JSON**: Uma estrutura de metadados em formato JSON que contém informações sobre os tensores armazenados, incluindo:
   - Nome de cada tensor
   - Tipo de dados (`dtype`)
   - Dimensões (`shape`)
   - *Offset* e tamanho no *buffer* de dados
3. **Buffer de dados**: A sequência contígua de bytes que representa os valores dos tensores da rede neuronal (o restante conteúdo do ficheiro).

1.2 Distribuição de Tipos de Dados

A execução do *script* de análise no ficheiro `model.safetensors` revelou que a totalidade dos dados é armazenada no formato **bfloat16** (*Brain Floating Point*). Este formato de 16 bits distribui-se da seguinte forma:

- **1 bit** de sinal
- **8 bits** de expoente
- **7 bits** de mantissa (fração)

Esta observação é crucial para a estratégia de compressão, pois todos os dados partilham as mesmas características estruturais.

1.3 Análise de Entropia

A eficácia da compressão está intimamente relacionada com a entropia da informação. No caso de números em vírgula flutuante como o `bfloat16`, a entropia não é uniforme ao longo dos bits que compõem o número. Uma análise da representação binária dos números em vírgula flutuante revela uma divisão clara na entropia de informação.

1.3.1 Distribuição de Entropia em Redes Neuronais

Os pesos de redes neuronais apresentam propriedades estatísticas particulares:

- **Bytes mais significativos (MSB) – Sinal e Expoente:** Contêm o bit de sinal e os 8 bits de expoente. Em redes neuronais treinadas:
  - Os valores tendem a concentrar-se em gamas de valores semelhantes após normalização
  - O sinal pode ser consistente em grupos de pesos relacionados
  - Os expoentes repetem-se frequentemente, criando padrões
  - Isto resulta numa baixa entropia nos *bytes* mais significativos, criando sequências repetitivas
  - **Resultado:** Baixa entropia, alta compressibilidade
- **Bytes menos significativos (LSB) – Mantissa:** Representam a mantissa (parte fracionária, 7 bits):
  - Contêm detalhes de precisão dos valores
  - A parte fracionária comporta-se praticamente como ruído aleatório
  - Pouca correlação entre valores consecutivos
  - **Resultado:** Alta entropia, baixa compressibilidade

1.3.2 Implicações para Compressão

A aplicação direta de algoritmos de compressão genéricos a dados `bfloat16` é ineficiente porque:

1. Os bits de alta e baixa entropia estão *intercalados* na representação em memória
2. Compressores genéricos operam sobre sequências de bytes
3. A intercalação dificulta a identificação de padrões repetitivos

2 Solução Implementada

Com base na análise anterior, desenvolveu-se uma solução que combina técnicas de reorganização de dados com compressão entrópica adaptativa.

2.1 Arquitetura da Solução

A solução divide-se em três componentes principais:

1. **Pré-processamento:** Reorganização estrutural dos dados (*bit-shuffling*)
2. **Compressão:** Aplicação do algoritmo Zstandard
3. **Gestão de memória:** Processamento por blocos para escalabilidade

2.2 Pré-Processamento: Bit-Shuffling

Visto que a entropia dos dados em vírgula flutuante não é uniforme, a aplicação direta de compressores genéricos é ineficaz. Como estabelecido na análise de entropia, a eficiência da compressão aumenta significativamente quando os dados com características semelhantes estão espacialmente próximos. Para contornar este problema, implementou-se uma função de *Bit-Shuffling* que reorganiza a disposição dos *bytes* em memória antes da compressão.

2.2.1 Algoritmo de Shuffling

Dado um bloco de dados `bfloat16` (cada valor ocupa 2 bytes, ou 16 bits), o algoritmo separa-os em dois vetores contíguos:

```
void shuffle_bf16(const uint8_t* src, uint8_t* dst, size_t size) {
    if (size % 2 != 0)
        throw std::runtime_error("Data size must be even for BF16 shuffle");

    size_t half = size / 2;

    // Separa bytes altos (MSB) e baixos (LSB)
    for (size_t i = 0; i < half; ++i) {
        dst[i] = src[2 * i + 1];        // Byte alto (sinal + expoente)
        dst[half + i] = src[2 * i];     // Byte baixo (mantissa)
    }
}
```

Estrutura resultante:

- **Vetor MSB (*Most Significant Bytes*) – Primeira metade do buffer:** Agrupa todos os bytes de sinal e expoente consecutivos
  - Este vetor apresenta repetibilidade de valores alta e baixa entropia
  - Alta correlação entre valores adjacentes
  - Ideal para compressão
- **Vetor LSB (*Least Significant Bytes*) – Segunda metade do buffer:** Agrupa todas as mantissas consecutivamente
  - Este vetor mantém uma entropia elevada, o que fará com que seja menos compressível que o vetor anterior
  - Comportamento pseudo-aleatório
  - Mantém a precisão dos valores originais

2.2.2 Reversibilidade

Esta transformação é totalmente reversível através da função inversa, utilizando a função `unshuffle_bf16`, garantindo que o processo é totalmente sem perdas (*lossless*):

```
void unshuffle_bf16(const uint8_t* src, uint8_t* dst, size_t size
    ) {
    if (size % 2 != 0)
        throw std::runtime_error("Data size must be even for BF16
    unshuffle");

    size_t half = size / 2;

    // Reconstroi os valores bfloat16 originais
    for (size_t i = 0; i < half; ++i) {
        dst[2 * i + 1] = src[i];        // Restaura byte alto
        dst[2 * i] = src[half + i];     // Restaura byte baixo
    }
}
```

Esta propriedade garante que a compressão preserva cada bit do modelo original, mantendo a integridade dos dados necessária para a utilização do modelo LLM.

2.3 Algoritmo de Compressão: Zstandard

2.3.1 Escolha do Algoritmo

Após o pré-processamento, os dados são submetidos ao algoritmo **Zstandard** (zstd). Decidiu-se usar o Zstandard devido à sua capacidade de atingir rácios de compressão competitivos sem penalizar excessivamente o tempo de descompressão, oferecendo ainda granularidade no ajuste de desempenho através dos seus diversos níveis de operação.

Zstandard é um algoritmo moderno desenvolvido pela Meta (Facebook) que oferece:

- **Rácios de compressão elevados:** Competitivo com algoritmos como LZMA e bzip2
- **Velocidade de descompressão:** Significativamente mais rápido que alternativas com rácios semelhantes
- **Níveis configuráveis:** De 1 (mais velocidade) a 22 (mais compressão)
- **Streaming support:** Permite processamento por blocos
- **Biblioteca estável:** API bem documentada e amplamente utilizada

2.3.2 Configuração

A implementação utiliza o nível de compressão **3** como padrão, oferecendo um equilíbrio entre:

- Rácio de compressão satisfatório
- Tempo de processamento aceitável
- Uso moderado de memória durante a compressão

O nível pode ser ajustado via linha de comandos (1-22) para privilegiar velocidade ou compressão conforme necessário.

2.4 Gestão de Memória: Processamento por Blocos

2.4.1 Problema

O carregamento integral de um ficheiro de grande dimensão (aproximadamente 1 GB ou 988.1 MB) para a memória pode sobrecarregar o sistema, especialmente em máquinas com recursos limitados, comprometendo a escalabilidade da solução ao lidar com ficheiros maiores. Este cenário apresenta desafios:

- **Limitações de RAM:** Sistemas com memória limitada podem não suportar o ficheiro completo
- **Escalabilidade:** Modelos maiores (10GB+) tornam-se inviáveis
- **Eficiência:** Alocações massivas podem fragmentar a memória e degradar o desempenho

2.4.2 Solução: Streaming por Blocos

A solução implementada adota uma estratégia de processamento por blocos de 32 MB. A solução implementa processamento incremental com blocos de tamanho fixo:

```
constexpr size_t CHUNK_SIZE = 32 * 1024 * 1024; // 32 MB
```

Algoritmo de processamento:

1. O ficheiro é lido sequencialmente em blocos de 32 MB (`CHUNK_SIZE` bytes)
2. Cada bloco é processado de forma independente: aplicar *shuffling* ao bloco
3. Comprimir o bloco com Zstandard
4. Escrever metadados e dados comprimidos no ficheiro de saída
5. O resultado é escrito no disco imediatamente, libertando a memória para o bloco seguinte
6. Repetir até processar todo o ficheiro

Esta abordagem garante que o uso máximo de memória permanece constante e reduzido, independentemente do tamanho total do ficheiro de entrada.

Vantagens:

- **Uso de memória constante:** Aproximadamente 96 MB independentemente do tamanho do ficheiro
  - 32 MB para buffer de leitura
  - 32 MB para buffer após *shuffling*
  - ∼32 MB para buffer de compressão (varia com nível de compressão)
- **Escalabilidade:** Suporta ficheiros arbitrariamente grandes
- **Paralelização:** Blocos independentes permitem processamento paralelo (implementado na versão OpenMP)

2.5 Implementações Disponíveis

O projeto oferece **duas implementações** da solução de compressão, ambas partilhando o mesmo algoritmo de *bit-shuffling* e formato de ficheiro, mas diferindo na estratégia de processamento:

2.5.1 Implementação Sequencial (`main.cpp`)

A implementação base processa os blocos de forma sequencial:

```
void compress(const string& input_path, const string& output_path, int level) {
    // ... processar cabecalho ...

    vector<uint8_t> raw_buf(CHUNK_SIZE);
    vector<uint8_t> shuffled_buf(CHUNK_SIZE);

    while (input.read((char*)raw_buf.data(), CHUNK_SIZE)) {
        size_t bytes_read = input.gcount();

        // Processar um bloco de cada vez
        shuffle_bf16(raw_buf.data(), shuffled_buf.data(), bytes_read);
        size_t c_size = ZSTD_compress(...);
        write_chunk_metadata_and_data(output, bytes_read, c_size, ...);
    }
}
```

2.5.2 Implementação Paralela com OpenMP (`bf16_omp.cpp`)

A implementação paralela utiliza OpenMP para processar múltiplos blocos simultaneamente, explorando sistemas multi-core:

```
constexpr int BATCH_SIZE = 8; // Processa 8 blocos em paralelo

void compress(const string& input_path, const string& output_path, int level) {
    int num_threads = omp_get_max_threads();
    cout << "Compressing with " << num_threads << " threads (Batch size: "
         << BATCH_SIZE << ")..." << endl;

    // ... processar cabecalho ...

    vector<Chunk> batch(BATCH_SIZE); // Pre-alocar buffers para 8 blocos

    while (!done) {
        // A. Ler batch (Serial)
        for (int i = 0; i < BATCH_SIZE; ++i) {
            input.read(..., CHUNK_SIZE);
```

```
            // ...
        }

        // B. Processar batch (Paralelo)
        #pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < chunks_in_batch; ++i) {
            // Cada thread processa um bloco independentemente
            shuffle_bf16(...);
            ZSTD_CCtx* cctx = ZSTD_createCCtx(); // Thread-local
            ZSTD_compressCCtx(cctx, ...);
            ZSTD_freeCCtx(cctx);
        }

        // C. Escrever batch (Serial)
        for (int i = 0; i < chunks_in_batch; ++i) {
            write_chunk_metadata_and_data(...);
        }
    }
}
```

2.6 Formato do Ficheiro Comprimido

Para suportar a descompressão por fluxo (*streaming*), a validação da integridade e a descompressão eficiente, foi definido um formato binário personalizado estruturado para o ficheiro comprimido:

2.6.1 Estrutura Global

```
+-------------------------+
| Header Size (8 bytes)   | <- uint64_t
+-------------------------+
| Header JSON Data        | <- Header Size bytes (não comprimido)
+-------------------------+
| Chunk 1 Metadata        | <- 16 bytes
+-------------------------+
| Chunk 1 Compressed Data | <- chunk_comp_size bytes
+-------------------------+
| Chunk 2 Metadata        | <- 16 bytes
+-------------------------+
| Chunk 2 Compressed Data | <- chunk_comp_size bytes
+-------------------------+
| ...                     |
+-------------------------+
```

2.6.2 Cabeçalho

O cabeçalho JSON do formato Safetensors (tamanho e conteúdo do cabeçalho JSON original) é armazenado **sem compressão** no ficheiro de saída. Esta decisão permite:

- Inspeção rápida dos metadados sem descompressão

- Validação da estrutura do ficheiro antes de processar os dados
- Compatibilidade com ferramentas que esperam cabeçalhos legíveis

2.6.3 Metadados de Cada Bloco

Uma sequência de blocos de dados, onde cada bloco comprimido é precedido por 16 bytes de metadados (dois inteiros de 64 bits):

- `chunk_raw_size` (8 bytes, `uint64_t`): O tamanho original dos dados (após *unshuffle*), antes da compressão
- `chunk_comp_size` (8 bytes, `uint64_t`): O tamanho do bloco após compressão

Esta estrutura permite que o descompressor reconstrua o ficheiro original, mantendo a integridade dos dados necessária para a utilização do modelo LLM.

Estes metadados permitem ao descompressor:

1. Alocar memória exata para os buffers
2. Validar a integridade de cada bloco
3. Processar o ficheiro sequencialmente sem necessidade de índice global

2.7 Implementação das Operações

2.7.1 Operação de Compressão

```
void compress(const string& input_path, const string& output_path
    , int level) {
    ifstream input(input_path, ios::binary);
    ofstream output(output_path, ios::binary);

    // 1. Processar e escrever cabecalho (nao comprimido)
    uint64_t header_size;
    input.read((char*)&header_size, 8);
    vector<uint8_t> header(header_size);
    input.read((char*)header.data(), header_size);

    output.write((char*)&header_size, 8);
    output.write((char*)header.data(), header_size);

    // 2. Processar dados em blocos
    vector<uint8_t> raw_buf(CHUNK_SIZE);
    vector<uint8_t> shuffled_buf(CHUNK_SIZE);

    while (input.read((char*)raw_buf.data(), CHUNK_SIZE)) {
        size_t bytes_read = input.gcount();

        // Shuffle
        shuffle_bf16(raw_buf.data(), shuffled_buf.data(),
    bytes_read);
```

```
23
24          // Comprimir
25          size_t c_size = ZSTD_compress(comp_buf.data(), bound,
26                                        shuffled_buf.data(),
    bytes_read, level);
27
28          // Escrever metadados e dados
29          write_uint64(output, bytes_read);      // chunk_raw_size
30          write_uint64(output, c_size);          // chunk_comp_size
31          output.write((char*)comp_buf.data(), c_size);
32      }
33  }
```

2.7.2 Operação de Descompressão

```
1   void decompress(const string& input_path, const string&
    output_path) {
2       ifstream input(input_path, ios::binary);
3       ofstream output(output_path, ios::binary);
4
5       // 1. Recuperar cabecalho
6       uint64_t header_size;
7       input.read((char*)&header_size, 8);
8       vector<uint8_t> header(header_size);
9       input.read((char*)header.data(), header_size);
10
11      output.write((char*)&header_size, 8);
12      output.write((char*)header.data(), header_size);
13
14      // 2. Descomprimir blocos
15      ZSTD_DCtx* dctx = ZSTD_createDCtx();
16      uint64_t chunk_raw_size, chunk_comp_size;
17
18      while (read_uint64(input, chunk_raw_size)) {
19          read_uint64(input, chunk_comp_size);
20
21          // Ler dados comprimidos
22          input.read((char*)comp_buf.data(), chunk_comp_size);
23
24          // Descomprimir
25          ZSTD_decompressDCtx(dctx, shuffled_buf.data(),
    chunk_raw_size,
26                              comp_buf.data(), chunk_comp_size);
27
28          // Unshuffle e escrever
29          unshuffle_bf16(shuffled_buf.data(), final_buf.data(),
    chunk_raw_size);
30          output.write((char*)final_buf.data(), chunk_raw_size);
31      }
32
```

```
33      ZSTD_freeDCtx(dctx);
34  }
```

3 Utilização

3.1 Compilação

O projeto inclui um `Makefile` para facilitar a compilação de ambas as implementações:

```
$ make all
g++ -O3 -Wall -Wextra -std=c++17 main.cpp -o compressor -lzstd
g++ -O3 -Wall -Wextra -std=c++17 -fopenmp bf16_omp.cpp -o bf16_omp -lzstd
```

3.2 Interface de Linha de Comandos

3.2.1 Compressão

Versão Sequencial:

```
$ ./compressor compress <input> <output> [level]
```

Versão Paralela:

```
$ ./bf16_omp compress <input> <output> [level]
```

3.2.2 Descompressão

Versão Sequencial:

```
$ ./compressor decompress <input> <output>
```

Versão Paralela:

```
$ ./bf16_omp decompress <input> <output>
```

Controlo de threads (versão OpenMP): É possível controlar o número de threads através da variável de ambiente `OMP_NUM_THREADS`:

```
$ OMP_NUM_THREADS=4 ./bf16_omp compress model.safetensors output.zst 5
Compressing with 4 threads (Batch size: 8)...
```

3.3 Teste Automatizado

O projeto inclui um *script* de teste (`run_compression.sh`) que valida ambas as implementações automaticamente:

1. Compila ambos os programas (`make all`)
2. Executa compressão com a versão sequencial
3. Descomprime e verifica integridade (versão sequencial)

4. Executa compressão com a versão paralela OpenMP
5. Descomprime e verifica integridade (versão paralela)
6. Compara ambos os ficheiros restaurados com o original usando `diff`
7. Reporta sucesso ou falha

Utilização:

```
$ bash run_compression.sh [input_file] [compression_level] [omp_threads]
```

4 Resultados e Análise de Desempenho

4.1 Desempenho da solução sequencial

A solução desenvolvida foi testada com diferentes niveis de compressão do Zstd para avaliar o compromisso entre tempo e taxa de compressão.

Nivel	Tamanho Final (MB)	Rácio (%)	Compressão	Descompressão
1	674.7	31.7	0m03s	0m02s
4	700.6	29.1	0m05s	0m02s
7	689.7	30.2	0m12s	0m02s
11	687.0	30.5	0m37s	0m02s
15	684.2	30.8	4m28s	0m02s
19	661.4	33.1	8m43s	0m02s
22	661.7	33.0	9m39s	0m02s

Table 1: Resultados de compressão da solução apresentada usando diferentes niveis de compressão

4.2 Resultados (OpenMP)

Este capítulo apresenta o desempenho da implementação paralela (OpenMP) e a sua comparação com a versão básica (sequencial). Os testes foram realizados sobre o ficheiro `model.safetensors` (988.1 MB), mantendo o mesmo pré-processamento BF16 (shuffle por bytes) e o mesmo formato de saída (cabeçalho não comprimido + blocos comprimidos por chunk). Para a versão OpenMP foi definido `OMP_NUM_THREADS=16`.

4.2.1 Desempenho da implementação paralela (OpenMP)

A Tabela 2 mostra os resultados para diferentes níveis do Zstd. Observa-se que os tamanhos finais são idênticos aos obtidos pela versão básica (o paralelismo altera apenas o tempo de execução, não o formato nem o conteúdo comprimido).

Nível	Tamanho Final (MB)	Rácio (%)	Compressão	Descompressão
1	674.7	31.7	0m01s	0m01s
4	700.6	29.1	0m03s	0m02s
7	689.7	30.2	0m07s	0m01s
11	687.0	30.5	0m22s	0m01s
15	684.2	30.8	1m29s	0m01s
19	661.4	33.0	2m25s	0m01s
22	661.7	33.0	2m29s	0m01s

Table 2: Resultados da implementação paralela (OpenMP) para diferentes níveis do Zstd (`OMP_NUM_THREADS=16`).

4.3 Comparação entre a versão básica e a versão OpenMP

A Tabela 3 resume o ganho de desempenho (*speedup*) obtido pela paralelização na compressão e descompressão. O *speedup* foi calculado como $Tserial/Tomp$.

Nível	Comp. Serial	Comp. OMP	Speedup	Desc. Serial	Desc. OMP
1	1.96s	0.98s	2.00x	0.88s	0.79s
4	4.36s	2.74s	1.59x	1.01s	1.78s
7	10.20s	6.70s	1.52x	1.00s	1.03s
11	34.07s	22.10s	1.54x	1.02s	0.92s
15	342.68s	89.28s	3.84x	0.95s	1.05s
19	569.73s	144.54s	3.94x	0.76s	0.83s
22	530.50s	148.86s	3.56x	0.79s	0.84s

Table 3: Comparação de desempenho entre a versão básica e a versão OpenMP (`OMP_NUM_THREADS=16`).

Analisando os resultados, verifica-se que a paralelização traz ganhos moderados nos níveis mais baixos (aproximadamente 1.5x–2x), onde a compressão é muito rápida e o tempo total é influenciado por overheads inevitáveis (leitura do ficheiro, escrita do output e gestão de batches/threads). Por outro lado, nos níveis altos (15–22), onde o custo computacional do Zstd domina, o ganho torna-se significativo, atingindo cerca de 3.5x–3.9x.

No caso da descompressão, os ganhos são pouco expressivos (e por vezes inexistentes), uma vez que esta fase é fortemente limitada por I/O e pela criação/gestão de contextos de descompressão por chunk, reduzindo o potencial de escalabilidade. Assim, a versão OpenMP é especialmente vantajosa quando o objetivo é reduzir o tempo de compressão em níveis elevados, mantendo exatamente o mesmo rácio de compressão.

4.4 Teste de compressores existentes

Para estabelecer uma *baseline* justa, o ficheiro `model.safetensors` (988.1 MB) foi comprimido com compressores genéricos amplamente utilizados, sem qualquer pré-processamento. Nesta experiência, o ficheiro foi comprimido e depois descomprimido, validando-se a integridade com uma comparação binária entre o ficheiro restaurado e o original.

Foram usados modos de compressão elevados (`gzip -9` e `xz -9e`) e medidos os tempos.

Compressor	**Tamanho (MB)**	**Redução (%)**	**Comp**	**Descomp**
gzip -9	781.9	20.9	88.61s	8.34s
xz -9e	692.0	30.0	445.04s	10.91s
Zstd + byte-shuffling	674.7	31.7	3s	2s

Table 4: Comparação com os resultados medidos usando compressores de uso geral (sem pré-processamento)

Conclusão: O `xz -9e` obtém uma redução superior ao `gzip -9`, mas a compressão é significativamente mais lenta. Estes resultados são expectáveis, pois os compressores genéricos operam diretamente sobre a sequência de bytes do formato BF16 (onde bytes de alta e baixa entropia estão intercalados), o que dificulta a exploração de padrões repetitivos. Ao reorganizar os bytes antes da compressão, é possível obter melhor compressão com tempos muito reduzidos.