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

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. Ao contrário de dados textuais, onde a redundância é facilmente explorável, os dados em vírgula flutuante apresentam frequentemente elevada entropia na representação binária, dificultando a atuação de compressores genéricos.

A avaliação da solução proposta não se limita apenas ao rácio de compressão final, mas considera uma análise que inclui o tempo computacional de compressão e descompressão e o consumo de memória durante o processo.

2 Análise

2.1 Estrutura de dados

Com o intuito de delinear a estratégia de compressão mais eficiente, procedeu-se a uma análise da estrutura e do conteúdo do ficheiro `model.safetensors`. Esta etapa foi fundamental para compreender a natureza dos dados e identificar redundâncias intrínsecas passíveis de serem exploradas.

2.2 Estrutura do Formato Safetensors

Através do desenvolvimento do *script* `analyze_safetensors.py` em Python, foi possivel decompor a estrutura binária do ficheiro. O ficheiro `model.safetensors` organiza-se em três blocos distintos:

1. **Tamanho do cabeçalho:** Um inteiro de 64 bits que identifica o tamanho da secção seguinte.
2. **Cabeçalho JSON:** Uma string descritiva que contém metadados sobre os tensores.
3. **Buffer de dados:** O restante conteúdo do ficheiro representa os dados da LLM.

2.3 Distribuição do tipo de dados

A execução do *script* de análise revelou que os dados do ficheiro são compostos integralmente por dados no formato **bfloat16**.

2.4 Análise de Entropia

Uma análise da representação binária dos números em virgula flutuante (como o BF16) revela uma divisão na entropia de informação:

- **Bits de Sinal e Expoente:** Em redes neuronais, os valores tendem-se a agrupar em gamas de valores semelhantes. Isto resulta numa baixa entropia nos *bytes* mais significativos, crianod sequências repetitivas que são teoricamente mais compressiveis.

- **Bits da Mantissa:** A parte fracionária do número comporta-se práticamente como ruído aleatório, logo tem uma entropia alta.

Esta estrutura sugere que a compressão direta do ficheiro será ineficiente. A estratégia ideal deverá passar por um pré-processamento que separe os componentes de baixa entropia dos componentes de alta entopia. Esta conclusão orientou o desenvolvimento da solução apresentada no capitulo seguinte.

3 Implementação

Com base na análise anterior, foi desenvolvida uma solução de compressão que combina técnicas de pré-processamento estrutural com algoritmos de codificação entrópica. A implementação foi realizada em `C++` para garantir a máxima eficiência no acesso à memória e controlo sobre os recursos do sistema.

3.1 Pré-Processamento

Visto que a entropia dos dados em virgula flutuante não é uniforme, a aplicação direta de compressores genéricos é eficaz. Para contornar este problema, implantou-se uma função de *Bit-Shuffling*.

Esta função reorganiza a dispozição dos *bytes* em memória antes da compressão. Dado um bloco de números de 16 bits (2 bytes), o algoritmo separa-os em dois vetores:

- **Vetor MSB (Most Significant Bytes):** Agrupa todos os bytes de sinal e expoente. Devido à natureza das redes neuronais, este vetor apresenta uma repetibilidade de valores alta e baixa entropia.
- **Vetor LSB (Least Significant Bytes):** Agrupa as mantissas. Este vetor mantém uma entropia elevada, o que fará com que seja menos compressivel que o vetor anterior.

Esta transformação é reversivel usando a função `unshuffle_bf16`, garantindo que o processo é totalmente sem perdas.

3.2 Gestão de Memória

O carregamento integral de um ficheiro de quase 1 GB para a memória pode sobrecarregar o sistema, especialmente em máquinas com recursos limitados, comprometendo a escalabilidade da solução ao lidar com ficheiros maiores.

A solução implementada adota uma estratégia de processamento por blocos de 32 MB. O algoritmo opera da seguinte forma:

1. O ficheiro é lido sequencialmente em blocos de 32 MB.
2. Cada bloco é processado de forma independente.
3. O resultado é escrito no disco imediatamente, libertando a memória para o bloco seguinte.

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

3.3 Algoritmo de compressão

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.

3.4 Formato do Ficheiro de Saída

Para suportar a descompressão for fluxo, foi definido um formato binário personalizado para o ficheiro comprimido:

1. **Cabeçalho Global:** Tamanho e conteúdo do cabeçalho JSON original.
2. **Blocos de Dados:** Uma sequência de blocos, onde cada um é precedido por dois inteiros de 64 bits:
    - `chunk_size`: O tamanho original dos dados
    - `compressed_size`: O tamanho do bloco comprimido

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

4 Resultados

Este capitulo apresenta o desempenho da solução desenvolvida em comparação com ferramentas de compressão padrão.

4.1 Desempenho de compressores padrão

Para estabelecer uma *baseline*, o ficheiro `model.safetensors` (988.1 MB) foi comprimido utilizando ferramentas de uso geral sem qualquer pré-processamento.

Compressor	**Tamanho Final (MB)**	**Rácio (%)**	**Compressão**	**Descompressão**
gzip	781.9	20.9	1m16s	0m06s
XZ / LZMA	696.1	29.5	8m32s	0m26s

Table 1: Resultados de compressão com ferramentas de uso geral

4.2 Desempenho da solução proposta

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
-7	887.8	10.0	0m03s	0m02s
-3	848.2	14.2	0m03s	0m02s
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 2: Resultados de compressão da solução apresentada usando diferentes niveis de compressão

Analisando estes resultados, é possivel reparar que o Nivel 1 apresentou um desempenho superior aos niveis intermédios (4 a 15), atingindo uma redução de 31.7% em apenas 3 segundos. Isto justifica-se pela natureza dos dados: como os exponentes de baixa entropia estão agrupados, o algoritmo rápido do Zstd consegue captar estas repetições instantaneamente. Os niveis superiores tentam procurar padrões mais complexos nas mantissas, o que se revela improdutivo e computacionalmente dispendioso.

Comparando o Nivel 1 com o Nivel 19, observa-se um ganho marginal de apenas 1.4%, mas com um custo temporal de 3 segundos para quase 9 minutos.

4.2.1 Comparação com Compressores Padrão

A análise dos dados tambem permite retirar conclusões definitivas sobre a eficiência da solução em comparação aos compressores padrão. Nestas comparações foram utilizados os dados recolhidos no Nivel 1, pois foram os valores com melhor equilibrio entre velocidade e taxa de compressão:

- **Solução Proposta vs Gzip:** A solução desenvolvida não só comprime 10.8% a mais que o Gzip (31.7% vs 20.9%), como é 25 vezes mais rápida (3s vs 1m16s). Isto prova que o Gzip, sendo um algoritmo agnóstico à estrutura dos dados, não consegue explorar eficazmente a redundância dos floats.
- **Solução Proposta vs XZ(LZMA):** O XZ é tradicionalmente uma referência para uma taxa de compressão alta, mas neste caso foi derrotado. Além da grande diferença no tempo necessário para a compressão (3s vs 8m43s), a solução proposta tambem apresenta uma taxa de compressão maior (31.7% vs 29.5%). Isto demonstra que o pré-processamento da estrutura de dados consegui superar a "força bruta" do ZX, que gastou recursos computacionais massivos a tentar encontrar padrões no meio dos dados sem processamento, enquanto a nossa solução isolou os dados e cumprimiu o que era relevante.

A solução proposta é a vencedora inequivoca, oferecendo o melhor racio de compressão, o menor tempo de compressão e descompressão e o menor consumo de recursos entre todas as alternativas testadas.