UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Filipe Caetano Oliveira de Resende

Multiplicação de Matrizes Comprimidas

Uberlândia, Brasil

2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Filipe Caetano Oliveira de Resende

Multiplicação de Matrizes Comprimidas

Trabalho de conclusão de curso apresentado à Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Felipe Alves da Louza

Universidade Federal de Uberlândia – UFU

Faculdade de Engenharia Elétrica

Bacharelado em Engenharia de Computação

Uberlândia, Brasil

2024

Filipe Caetano Oliveira de Resende

Multiplicação de Matrizes Comprimidas

Trabalho de conclusão de curso apresentado à Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Engenharia de Computação.

Trabalho aprovado. Uberlândia, Brasil, 09 de dezembro de 2024:

Prof. Dr. Felipe Alves da Louza
Orientador

Prof. Dr. Humberto Luiz Razente
Universidade Federal de Uberlândia

Prof. Dr. Daniel Saad Nogueira Nunes
Instituto Federal de Brasília

Uberlândia, Brasil
2024

Resumo

Este trabalho explora a área de *Compressed Linear Algebra*, que investiga métodos para comprimir matrizes, permitindo a realização de operações algébricas diretamente sobre as representações comprimidas de forma eficiente. Foi implementado um algoritmo de compressão de matrizes, juntamente com um método para multiplicar as matrizes comprimidas por um vetor à direita, ambos baseados na solução apresentada em Ferragina et al. (2022). O processo de compressão foi realizado de maneira particionada, dividindo as matrizes em blocos processados sequencialmente. Os experimentos avaliaram o impacto do número de blocos na redução do uso de memória RAM. Os resultados indicaram uma redução significativa no consumo de memória, tanto durante a compressão quanto na multiplicação, à medida que a quantidade de blocos utilizados na segmentação da matriz aumenta. O tempo de execução melhorou durante a compressão e permaneceu praticamente constante na multiplicação. Contudo, a taxa de compressão sofreu uma degradação moderada com o aumento do número de blocos utilizados na divisão de cada matriz. Concluímos que a abordagem proposta é promissora para cenários com recursos computacionais limitados, como dispositivos embarcados e IoT.

Palavras-chave: algoritmos de compressão, compressão de matrizes, compressed linear algebra, multiplicação de matrizes, compressão gramatical.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

No contexto atual da área de Aprendizado de Máquina, é cada vez mais comum lidar com matrizes de grandes dimensões. Vários algoritmos de aprendizado de máquina são iterativos, com acesso repetido para leitura dos dados. Esses algoritmos frequentemente utilizam multiplicações entre matrizes e vetores para alcançar um modelo ótimo. Essa operação requer que a matriz inteira seja varrida, com duas operações de ponto flutuante sendo realizadas por elemento (uma multiplicação e uma adição). Nessa situação, é essencial que a matriz caiba na memória RAM disponível para garantir um bom desempenho, o que torna a compressão dessas matrizes uma necessidade importante.

A largura de banda do disco é de 10 a 100 vezes mais lenta do que a da memória RAM, e a memória RAM, por sua vez, é de 10 a 40 vezes mais lenta do que o desempenho máximo de ponto flutuante. Portanto, mesmo quando a matriz inteira está armazenada na memória RAM, o acesso à memória ainda representa um gargalo para a multiplicação matriz-vetor (ELGOHARY et al., 2018).

Se conseguirmos comprimir a matriz de forma que seja possível realizar a multiplicação por vetor diretamente nos dados comprimidos, sem a necessidade de descompactá-los, poderemos não apenas armazenar a matriz inteira na memória RAM, mas também minimizar o número de acessos à memória durante a multiplicação, aumentando, assim, o desempenho temporal.

Frequentemente, a compressão utilizada no campo do Aprendizado de Máquina é do tipo com perdas, ou seja, resulta em alguma perda de informação. Sistemas como TensorFlow (ABADI et al., 2016) e PStore (BHATTACHERJEE; DESHPANDE; SUSSMAN, 2014), por exemplo, aplicam compressão com perdas, mais especificamente truncamento de mantissa, para melhorar a eficiência em tarefas específicas. Em cenários onde pequenas imprecisões são aceitáveis, como, por exemplo, cálculos iniciais de gradientes, a compressão com perdas pode ser muito útil (ELGOHARY et al., 2018).

Já no contexto de operações de álgebra linear, a compressão de matrizes sem perdas é desejada para garantir resultados precisos. Infelizmente, a maioria dos métodos de compressão sem perdas não consegue aproveitar as redundâncias das matrizes de forma eficiente, resultando em compressões menos eficazes. Além disso, esses métodos frequentemente exigem a descompressão total das matrizes para a execução de operações algébricas e, portanto, não economizam espaço no momento mais crítico (FERRAGINA et al., 2022).

Nos trabalhos de Elgohary et al. (2018) e Elgohary et al. (2019), foi introduzida uma nova linha de pesquisa chamada *Compressed Linear Algebra* (CLA). Nesses estudos, são apresentados métodos de compressão sem perdas que, além de oferecerem boas taxas

de compressão, também possibilitam a realização eficiente de operações algébricas, como a multiplicação de matrizes por vetores, diretamente na forma comprimida da matriz, o que permite economizar tempo e recursos durante o processamento.

Seguindo essa linha de pesquisa, Ferragina et al. (2022) propuseram um novo método de compressão sem perdas para matrizes, que permite a multiplicação de uma matriz comprimida por um vetor, tanto à direita quanto à esquerda, em tempo e espaço proporcionais ao tamanho da matriz comprimida.

1.1 Objetivos

Neste trabalho, investigamos o particionamento de matrizes em blocos com o objetivo de aplicar, de forma sequencial, o método de compressão proposto por Ferragina et al. (2022) a cada bloco e, em seguida, utilizar os blocos comprimidos para realizar multiplicações, também de forma sequencial. Essa abordagem visa reduzir o pico de uso de memória tanto na compressão quanto na multiplicação. No trabalho original, os autores também exploraram a segmentação de matrizes, mas utilizando processamento paralelo por meio de uma abordagem multithreading, que melhora os tempos de compressão e multiplicação, mas geralmente aumenta o pico de uso de memória.

Além de reduzir o pico de uso de memória, esperamos que nossa abordagem também melhore o desempenho em relação aos tempos de compressão das matrizes, embora de forma menos expressiva do que os ganhos obtidos com o uso de multithreading.

Para avaliar a eficácia da proposta, realizaremos experimentos com diferentes matrizes, analisando as taxas de compressão, os tempos de compressão, os tempos de execução da multiplicação e os picos de uso de memória para diferentes números de blocos. Em nossso trabalho, realizaremos multiplicações apenas por vetores à direita das matrizes.

1.2 Organização deste trabalho

O restante deste trabalho está estruturado da seguinte maneira. O **Capítulo 2** fornece as definições e conceitos essenciais para a compreensão do desenvolvimento do trabalho. O **Capítulo 3** revê os trabalhos relacionados, com ênfase nas pesquisas que fundamentam este estudo. Neste capítulo, o método de compressão proposto por Ferragina et al. (2022) será descrito detalhadamente, acompanhado de exemplos ilustrativos. O **Capítulo 4** apresenta a proposta do trabalho em detalhes, incluindo um exemplo passo a passo da aplicação do método de compressão em blocos proposto, seguido de uma explicação detalhada de como a representação compactada gerada nesse exemplo pode ser utilizada para calcular a multiplicação da matriz original por um vetor. Além disso, este capítulo fornece análises comparativas das abordagens com e sem particionamento

da matriz, considerando tanto os aspectos de compressão quanto de multiplicação. No **Capítulo 5**, são apresentados os experimentos realizados, seus resultados e respectivas análises. Por fim, no **Capítulo 6**, apresenta-se a conclusão do trabalho e sugestões para trabalhos futuros.

2 Referencial Teórico

2.1 Conceitos fundamentais

Nesta seção, apresentamos os conceitos fundamentais de alfabetos, cadeias e linguagens formais, em grande parte baseados em (SIPSER, 1997), exceto onde indicado.

Definição 2.1.1. Um alfabeto é um conjunto finito e não vazio. Seus elementos são chamados de símbolos .

Exemplos:

- $\Sigma_1 = \{a, b, c, d\}$
- $\Sigma_2 = \{x\}$
- $\Sigma_3 = \{0, 1\}$

Definição 2.1.2. Uma cadeia ω sobre um alfabeto Σ é uma sequência finita de símbolos de Σ, que pode, inclusive, ser vazia (neste caso, representada por ε).

Exemplos de cadeias sobre $\Sigma = \{a, b\}$:

- $\omega_1 = \varepsilon$
- $\omega_2 = a$
- $\omega_3 = aaa$
- $\omega_4 = baba$
- $\omega_5 = bbbbbb$
- $\omega_6 = bababbbaabba$

Definição 2.1.3. Uma linguagem L sobre um alfabeto Σ é um conjunto (finito ou infinito) de cadeias sobre Σ.

Exemplos de linguagens sobre $\Sigma = \{a, b\}$:

- $L_1 = \emptyset$
- $L_2 = \{ab, bb, aaba\}$

- $L_3 = \{\varepsilon, a, b, aa, ab, ba, bb, \cdots\}$
- $L_4 = \{baba\}$
- $L_5 = \{\varepsilon\}$

Definição 2.1.4. (HOPCROFT; MOTWANI; ULLMAN, 2007) A estrela de Kleene de uma linguagem L, denotada por L^*, é o conjunto de cadeias que podem ser formadas pela concatenação de cadeias de L, inclusive com repetições.

Considere o alfabeto $\Gamma = \{0, 1, 2, ..., 9\}$ e a linguagem $A = \{123, 456, 78\}$. Temos que

$$A^* = \{\varepsilon, 123, 456, 78, 123123, 123456, 12378, 456123, \cdots\}$$

A estrela de Kleene pode ser aplicada sobre alfabetos ao invés de linguagens, seguindo-se a mesma lógica:

$$\Gamma^* = \{\varepsilon, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 00, 01, \cdots\}$$

Neste caso, ela denota a linguagem formada por todas as cadeias sobre o alfabeto.

Lema 2.1.1. *Se L é uma linguagem sobre um alfabeto Σ, então $L \subseteq \Sigma^*$*

2.2 Gramáticas livres de contexto

Uma **gramática livre de contexto (GLC)** é um conjunto formal de regras de produção usado para definir linguagens. Essas gramáticas possuem um **símbolo inicial**, a partir do qual se pode gerar cadeias de símbolos terminais aplicando sucessivamente as regras de produção. Por exemplo, considere a gramática G a seguir:

$$\begin{aligned} S &\to XY \\ X &\to aX \mid a \\ Y &\to bY \mid b \end{aligned}$$

Nesta gramática:

- Os **símbolos terminais** são a e b, que formam as cadeias finais.
- Os **símbolos não-terminais** (ou variáveis) são S, X e Y.
- A **variável inicial** é S, ou seja, as derivações sempre começam por ela.

A **notação** | é usada para indicar múltiplas regras associadas à mesma variável. Na gramática G, por exemplo:

- A regra $X \to aX \mid a$ significa que a variável X pode ser substituída por aX ou apenas por a.
- Similarmente, $Y \to bY \mid b$ indica que a variável Y pode ser substituída por bY ou apenas por b.

Portanto, a gramática permite gerar cadeias ao substituir as variáveis de acordo com as regras, até que todas as variáveis sejam eliminadas e apenas símbolos terminais permaneçam. Aqui estão alguns exemplos de cadeias que podem ser geradas pela gramática G:

- $S \Rightarrow XY \Rightarrow aY \Rightarrow ab$
- $S \Rightarrow XY \Rightarrow aXY \Rightarrow aaXY \Rightarrow aaaY \Rightarrow aaabY \Rightarrow aaabb$
- $S \Rightarrow XY \Rightarrow aY \Rightarrow abY \Rightarrow abbY \Rightarrow abbbY \Rightarrow abbbb$

O conjunto de todas as cadeias que podem ser geradas por uma gramática é denominado linguagem da gramática. No caso da gramática G, a linguagem gerada consiste em todas as cadeias que começam com um ou mais símbolos a, seguidos por um ou mais símbolos b. Mais formalmente, $L(G) = \{a^n b^m \mid n > 0, m > 0\}$.

Formalmente, definimos uma gramática livre de contexto da seguinte forma:

Definição 2.2.1. (SIPSER, 1997) Uma gramática livre de contexto é uma tupla (V, Σ, R, S), onde:

- V é um conjunto finito de variáveis (também chamadas símbolos não-terminais),
- Σ é um conjunto finito de símbolos denominados símbolos terminais. Este conjunto é disjunto de V (i.e., $V \cap \Sigma = \emptyset$),
- R é um conjunto finito de regras. Cada regra tem a forma $A \to \omega$, com $A \in V$ e $\omega \in (V \cup \Sigma)^*$,
- $S \in V$ é a variável inicial.

Sendo u, $v \in (V \cup \Sigma)^*$ e $A \to \omega \in R$, dizemos que uAv produz $u\omega v$. Em notação matemática, $uAv \Rightarrow u\omega v$. O que está sendo feito é a substituição de uma variável (no caso A) por uma cadeia $\omega \in (V \cup \Sigma)^*$ produzida por ela.

Sendo $u, v \in (V \cup \Sigma)^*$, dizemos que u deriva v ($u \Rightarrow^* v$) se $u = v$ ou se for possível gerar v a partir de u por meio da aplicação de sucessivas substituições associadas às regras de produção.

A linguagem de uma gramática G é definida por:

$$L(G) = \{\omega \in \Sigma^* \mid S \Rightarrow^* \omega\}$$

Em outras palavras, a linguagem $L(G)$ de uma gramática G é o conjunto de todas as cadeias de símbolos terminais que podem ser derivadas a partir da variável inicial S.

2.2.1 Gramáticas straight-line

Uma **gramática straight-line** (*straight-line grammar*, SLG) é uma gramática livre de contexto que gera exatamente uma única cadeia ω (BENZ; KÖTZING, 2013). Isso implica que existe apenas uma substituição possível para cada símbolo não-terminal, e, consequentemente, que o número de regras da gramática é igual ao número de símbolos não-terminais.

Exemplo

A gramática $G = (\{S, A, B, C, D\}, \{a, b\}, R, S)$, onde o conjunto de regras R é

$$\begin{aligned} S &\to AA \\ A &\to BB \\ B &\to CC \\ C &\to DD \\ D &\to ab \end{aligned}$$

é um exemplo de gramática straight-line que gera a cadeia

$$\omega = \ abababababababababababababababab$$

2.3 Algoritmos de compressão

Comprimir dados significa representá-los usando uma quantidade menor de bits. Em geral, um método de compressão envolve dois algoritmos distintos (SAYOOD, 2006): um deles é o algoritmo que comprime o arquivo original χ e produz uma representação χ_c, possivelmente com menos bits. O outro é o algoritmo de descompressão, que recebe a representação comprimida χ_c como entrada e produz uma reconstrução γ. Além disso, existem dois tipos de métodos para compressão:

- **Compressão sem perdas (*lossless*)**: Na compressão sem perdas, os dados podem ser completamente recuperados após a descompressão, ou seja, não há perda de dados. Neste tipo de compressão, temos que $\gamma = \chi$.

- **Compressão com perdas (*lossy*)**: Esse tipo de compressão reduz o tamanho dos dados ao descartar parte da informação, normalmente informações redundantes ou menos perceptíveis. Após a descompressão, os dados não podem ser recuperados exatamente como eram antes. Portanto, neste caso, temos que $\gamma \neq \chi$.

2.3.1 Compressão baseada em gramática

Dada uma cadeia ω, podemos derivar um conjunto de regras R cujo símbolo incial S gera ω exclusivamente, isto é, uma SLG que gera ω. O princípio central é substituir padrões repetidos na cadeia por regras reutilizáveis. Se a cadeia ω possuir muitas subcadeias repetidas, sua representação na forma de gramática livre de contexto pode economizar muito espaço. Esta forma de compressão é bastante utilizada para comprimir textos e denomina-se compressão gramatical (NAVARRO, 2016).

Exemplo:

A cadeia $\omega = abcabcabcabcabcabcabcabc$ pode ser representada pela SLG $G = (\{S, A, B\}, \{a, b, c\}, R, S)$, na qual R é

$$S \to AA$$
$$A \to BBBB$$
$$B \to abc$$

O tamanho de uma GLC é comumente definido como o número de símbolos à direita das regras de produção, e portanto é 9 neste caso. O problema de encontrar a menor SLG que gera uma dada cadeia é chamado problema da menor gramática. Charikar et al. (2005) provaram que este problema pertence à classe NP-difícil. Contudo, existem excelentes heurísticas para compressão gramatical, entre elas o algoritmo Re-Pair, proposto por Larsson e Moffat (1999), o qual detalharemos no próximo capítulo.

3 Trabalhos correlatos

3.1 Recursive Pairing

O método Re-Pair (*Recursive Pairing*) é um algoritmo de compressão gramatical que substitui recursivamente pares de símbolos adjacentes por um único símbolo em um texto, reduzindo, assim, o tamanho do texto original (LARSSON; MOFFAT, 1999). A cada iteração, o par de símbolos adjacentes com maior número de ocorrências no texto, digamos ab, é substituído por um novo símbolo não-terminal A, e uma nova regra é adicionada à gramática: $A \rightarrow ab$. Quando nenhum par aparecer duas ou mais vezes, o algoritmo termina. No fim tem-se uma sequência C de símbolos com as substituições apropriadas e um conjunto de regras R que, juntos, podem ser utilizados para reconstruir o texto original.

Exemplo:

Consideremos a cadeia $\omega =$ aabbaababababb. O par de símbolos adjacentes que mais se repete é ab. Assim, criamos a primeira regra $N_1 \rightarrow ab$ e realizamos as substituições necessárias. O resultado dessa substituição é a cadeia $aN_1baN_1N_1N_1b$.

Na nova cadeia, observa-se um empate entre as ocorrências dos pares aN_1, N_1b e N_1N_1, com cada um aparecendo duas vezes. Para resolver esse empate, podemos adotar o critério de desempate baseado na ordem de aparecimento dos pares, isto é, escolhemos o par que surge primeiro da esquerda para a direita. Com isso, criamos a regra $N_2 \rightarrow aN_1$. Após essa substituição, obtemos a cadeia comprimida $N_2bN_2N_1N_1b$.

O algoritmo é interrompido nesse ponto, pois não há mais pares de símbolos adjacentes que se repetem. Portanto, a forma comprimida final da cadeia ω é:

$$C = N_2bN_2N_1N_1b$$

E o conjunto de regras gerado é:

$$R = \{N_1 \rightarrow ab, N_2 \rightarrow aN_1\}$$

A quantidade de regras no conjunto R corresponde ao número de símbolos não-terminais gerados durante o processo de compressão. Além disso, se um símbolo não-terminal N_k aparece no lado direito de uma regra associada a um símbolo N_i, então $k < i$, já que N_k foi criado antes de N_i.

A forma comprimida final da cadeia ω também pode ser visualizada como uma SLG $G = (\{C, R\}, \{a, b\}, R', C)$, onde R' é

$$
\begin{aligned}
C &\rightarrow N_2 b N_2 N_1 N_1 b \\
N_1 &\rightarrow ab \\
N_2 &\rightarrow N_1 b
\end{aligned}
$$

3.2 Compressed Linear Algebra

Em Elgohary et al. (2018) e Elgohary et al. (2019) os autores propõe métodos de compressão sem perda para matrizes que não só oferecem boa taxa de compressão, mas também permitem a execução eficiente de operações de álgebra linear na forma comprimida.

Em Ferragina et al. (2022), é apresentado um novo método de compressão sem perdas, que torna possível a multiplicação de uma matriz comprimida por um vetor, tanto à direita quanto à esquerda, em tempo e espaço proporcionais ao tamanho da matriz comprimida. Os autores introduzem o formato CSRV (*compressed sparse row/value*), uma adaptação do conhecido formato CSR (*compressed sparse row*) (SAAD, 2003), e aplicam o algoritmo Re-Pair a matrizes no formato CSRV.

3.2.1 Representação CSRV

A representação CSRV de uma matriz é composta por duas estruturas: um vetor V, que guarda os valores não nulos distintos da matriz, e uma sequência de símbolos S, que indica as posições da matriz onde os valores de V aparecem. Esta sequência delimita as linhas da matriz por meio do símbolo especial \$. Se, em uma linha, aparecer o símbolo $\langle i, j \rangle$, significa que o valor $V[i]$ aparece na coluna j desta linha.

Exemplo

Considere a seguinte matriz de entrada:

$$
A = \begin{bmatrix}
5.3 & 8.1 & 6.0 & 2.7 & 6.0 & 5.3 \\
2.7 & 0 & 8.1 & 0 & 6.0 & 5.3 \\
2.7 & 0 & 8.1 & 0 & 6.0 & 5.3 \\
5.3 & 8.1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 6.0 & 5.3 \\
5.3 & 8.1 & 6.0 & 2.7 & 0 & 0 \\
5.3 & 8.1 & 0 & 0 & 0 & 0 \\
0 & 0 & 6.0 & 2.7 & 0 & 0
\end{bmatrix}
$$

Sua representação no formato CSRV é dada por:

$V = [5.3, 8.1, 6.0, 2.7]$

$S = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle\langle 2,4\rangle\langle 0,5\rangle\$$
$\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$$
$\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$$
$\langle 0,0\rangle\langle 1,1\rangle\$$
$\langle 2,4\rangle\langle 0,5\rangle\$$
$\langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle\$$
$\langle 0,0\rangle\langle 1,1\rangle\$$
$\langle 2,2\rangle\langle 3,3\rangle\$$

Em uma implementação, a sequência S pode ser armazenada como uma sequência de inteiros (por exemplo, inteiros de 32 bits sem sinal). O símbolo \$ pode ser representado pelo inteiro 0 e um par $\langle i, j\rangle$ pode ser codificado como $1 + im + j$, onde m é o número de colunas da matriz. Como $0 \leq j < m$, é possivel extrair i e j através da divisão euclidiana de $im + j$ por m: i é o quociente e j, o resto.

3.2.2 Compressão da sequência S com Re-Pair

O método de compressão proposto no artigo baseia-se na aplicação do algoritmo Re-Pair sobre o formato CSRV da matriz. Dada uma matriz M na representação (S, V), o Re-Pair é aplicado diretamente à sequência S. O resultado é uma representação compactada (C, R, V), onde C é a sequência final de símbolos, e R é o conjunto de regras que descrevem a expansão dos símbolos não-terminais em C.

Para permitir que as operações de álgebra linear possam ser realizadas sem a necessidade de descompressão da matriz, Ferragina et al. (2022) modificaram o algoritmo Re-Pair para que nenhuma regra de R contenha o símbolo especial \$. Dessa forma, a sequência C resultante assume a seguinte forma:

$$C = \omega_1\$\omega_2\$\cdots\omega_n\$,$$

onde $\omega_1, \omega_2, \cdots, \omega_n \in (\Sigma \cup N)^*$, com Σ representando o conjunto de símbolos terminais da sequência original S (cada par $\langle i, j\rangle$ está sendo tratado como um único símbolo) e N o conjunto de símbolos não-terminais gerados pelo Re-Pair.

Exemplo

Considere a sequência S da representação CSRV da matriz A. Descreveremos a execução do Re-Pair passo a passo, mostrando as regras criadas e as versões de C em cada etapa:

1. $N_1 \rightarrow \langle 0,0\rangle\langle 1,1\rangle$

   $C = N_1\langle 2,2\rangle\langle 3,3\rangle\langle 2,4\rangle\langle 0,5\rangle\$\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$$
   $\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$N_1\$$
   $\langle 2,4\rangle\langle 0,5\rangle\$N_1\langle 2,2\rangle\langle 3,3\rangle\$$
   $N_1\$\langle 2,2\rangle\langle 3,3\rangle\$$

2. $N_2 \rightarrow \langle 2,4\rangle\langle 0,5\rangle$

   $C = N_1\langle 2,2\rangle\langle 3,3\rangle N_2\$\langle 3,0\rangle\langle 1,2\rangle N_2\$$
   $\langle 3,0\rangle\langle 1,2\rangle N_2\$N_1\$$
   $N_2\$N_1\langle 2,2\rangle\langle 3,3\rangle\$$
   $N_1\$\langle 2,2\rangle\langle 3,3\rangle\$$

3. $N_3 \rightarrow \langle 2,2\rangle\langle 3,3\rangle$

   $C = N_1N_3N_2\$\langle 3,0\rangle\langle 1,2\rangle N_2\$$
   $\langle 3,0\rangle\langle 1,2\rangle N_2\$N_1\$$
   $N_2\$N_1N_3\$$
   $N_1\$N_3\$$

4. $N_4 \rightarrow N_1N_3$

   $C = N_4N_2\$\langle 3,0\rangle\langle 1,2\rangle N_2\$$
   $\langle 3,0\rangle\langle 1,2\rangle N_2\$N_1\$$
   $N_2\$N_4\$$
   $N_1\$N_3\$$

5. $N_5 \rightarrow \langle 3,0\rangle\langle 1,2\rangle$

   $C = N_4N_2\$N_5N_2\$$
   $N_5N_2\$N_1\$$
   $N_2\$N_4\$$
   $N_1\$N_3\$$

6. $N_6 \rightarrow N_5N_2$

   $C = N_4N_2\$N_6\$$
   $N_6\$N_1\$$
   $N_2\$N_4\$$
   $N_1\$N_3\$$

Após a criação de N_6, não há mais pares de símbolos adjacentes ocorrendo mais de uma vez, e, com isso, o algoritmo para. Portanto, a representação (C, R, V) de A é:

$$C = N_4N_2\$N_6\$N_6\$N_1\$N_2\$N_4\$N_1\$N_3\$$$

$$\begin{aligned} R = \{&N_1 \to \langle 0,0\rangle\langle 1,1\rangle, N_2 \to \langle 2,4\rangle\langle 0,5\rangle,\\ &N_3 \to \langle 2,2\rangle\langle 3,3\rangle, N_4 \to N_1N_3,\\ &N_5 \to \langle 3,0\rangle\langle 1,2\rangle, N_6 \to N_5N_2\} \end{aligned}$$

$$V = [5.3, 8.1, 6.0, 2.7]$$

3.2.3 Multiplicação da matriz comprimida por vetor à direita

Considerando uma matriz $M \in \mathbb{R}^{n\times m}$ na representação (S, V) e um vetor $x \in \mathbb{R}^m$, Ferragina et al. (2022) definem:

Definição 3.2.1. Dado um par $\langle l, j\rangle \in S$,

$$eval_x(\langle l, j\rangle) = V[l] \cdot x[j]$$

Agora, considerando a matriz na representação final (C, R, V), é definido:

Definição 3.2.2. Dado um símbolo não-terminal $N_i \in C$ cuja a expansão é $\langle l_1, j_1\rangle, \langle l_2, j_2\rangle, \cdots \langle l_k, j_k\rangle$,

$$eval_x(N_i) = \sum_{i=1}^{k} eval_x(\langle l_i, j_i\rangle) = \sum_{i=1}^{k} V[l_i] \cdot x[j_i]$$

É fácil observar que, se a regra de produção associada a N_i é $N_i \to AB$, com $A, B \in \Sigma \cup N$, então $eval_x(N_i) = eval_x(A) + eval_x(B)$.

Para fins didáticos e simplificação nas explicações subsequentes, vamos assumir que C segue a forma:

$$C = N_{i_1}\$N_{i_2}\$\cdots N_{i_n}\$,$$

onde $N_{i_1}, N_{i_2}, \cdots, N_{i_n} \in N$. Embora sempre seja possível converter C para esse formato com a inclusão de regras adicionais ao final da execução do Re-Pair, isso não traz benefícios em termos de eficiência ou compressão. Portanto, essa forma será utilizada apenas para facilitar a compreensão da operação de multiplicação.

Note que, na nomenclatura que estamos adotando, N_1 representa o primeiro símbolo não-terminal gerado durante a execução do Re-Pair, enquanto N_{i_1} denota o símbolo não-terminal correspondente à primeira linha da matriz.

Lema 3.2.1. *Considerando uma matriz $M \in \mathbb{R}^{n\times m}$ na representação (C, R, V), com $C = N_{i_1}\$N_{i_2}\$\cdots N_{i_n}\$$ e um vetor $x \in \mathbb{R}^m$, se $y = M \cdot x$, então $y[r] = eval_x(N_{i_r})$, para $r = 1, 2, \cdots, n$.*

Demonstração. Temos que $y[r] = \sum_{j=1}^{m} M[r][j] \cdot x[j]$ para $r = 1, 2, \cdots, n$. Seja A o conjunto de índices de colunas com valor não nulo na linha r de M, temos $y[r] = \sum_{u\in A} M[r][u] \cdot x[u]$. Considere $\langle l_1, j_1\rangle, \langle l_2, j_2\rangle, \cdots \langle l_k, j_k\rangle$ a expansão de N_{i_r}. Como cada entrada não nula da linha r de M corresponde a um único símbolo $\langle l_i, j_i\rangle$ da expansão de N_{i_r}, e vice-versa, temos que $y[r] = \sum_{i=1}^{k} V[l_i] \cdot x[j_i] = \sum_{i=1}^{k} eval_x(\langle l_i, j_i\rangle) = eval_x(N_{i_r})$.

□

Para calcular $y = M \cdot x$ sem descomprimir a matriz, precisamos primeiramente preencher um vetor auxiliar $W[1, q]$, onde q é o número de regras de R. O vetor W deve ser preenchido de forma sequencial, do índice 1 ao índice q, de modo que $W[i] = eval_x(N_i)$.

Se $N_i \to AB$, então $W[i]$ recebe $\text{eval}_x(A) + \text{eval}_x(B)$. Se A ou B forem um símbolo terminal $\langle l, j\rangle$, então basta fazer $eval_x(\langle l, j\rangle) = V[l] \cdot x[j]$. Se A ou B forem um símbolo não-terminal N_k, então $eval_x(N_k)$ já estará presente em $W[k]$, pois $k < i$. Como cada posição de W é preenchida em $\mathcal{O}(1)$, temos que o preenchimento completo de W ocorre em $\mathcal{O}(q) = \mathcal{O}(|R|)$.

A etapa final consiste em iterar sobre $C = N_{i_1}\$N_{i_2}\$\cdots N_{i_n}\$$ à medida que preenchemos o vetor y. Se $N_{i_r} = N_j$, então $y[r]$ é preenchido com $W[j]$. Esta etapa ocorre em $\mathcal{O}(|C|)$.

Caso geral: cadeias com múltiplos símbolos

No caso geral, onde cada linha da sequência S está associada a uma cadeia que pode conter vários símbolos terminais e não-terminais, a etapa de iteração sobre C é ligeiramente diferente.

Considere $\omega_r = s_1 s_2 \cdots s_k$ a cadeia correspondente a linha r de M, onde cada s_i é um símbolo (terminal ou não) de ω_r. O cálculo de $y[r]$ é ajustado para

$$y[r] = \sum_{i=1}^{k} eval_x(s_i)$$

Cada $eval_x(s_i)$ ocorre em $\mathcal{O}(1)$. Portanto, assim como no caso simplificado, o tempo gasto na etapa de iteração sobre C é $\mathcal{O}(|C|)$.

Ao todo, a multiplicação da matriz comprimida (C, R, V) pelo vetor x é executada em tempo $\mathcal{O}(|C| + |R|)$ com uso de $O(|R|)$ palavras de memória auxiliar (devido ao vetor W).

Exemplo

Considere a representação (C, R, V) da matriz A utilizada anteriormente. Considere também o vetor $x \in \mathbb{R}^6$ tal que $x = [1.0, 3.2, 2.5, 3.2, 1.7, 8.0]^T$. Vamos descrever, passo a passo, como calcular $y = A \cdot x$ a partir da representação (C, R, V) de A.

- **Primeiro passo**: preenchimento do vetor auxiliar $W[0, 5]$

  1. $W[0] = eval_x(N_1) = eval_x(\langle 0, 0\rangle) + eval_x(\langle 1, 1\rangle) = V[0] \cdot x[0] + V[1] \cdot x[1] = 5.3 \cdot 1.0 + 8.1 \cdot 3.2 = 31.22$
  2. $W[1] = eval_x(N_2) = eval_x(\langle 2, 4\rangle) + eval_x(\langle 0, 5\rangle) = V[2] \cdot x[4] + V[0] \cdot x[5] = 6.0 \cdot 1.7 + 5.3 \cdot 8.0 = 52.6$
  3. $W[2] = eval_x(N_3) = eval_x(\langle 2, 2\rangle) + eval_x(\langle 3, 3\rangle) = V[2] \cdot x[2] + V[3] \cdot x[3] = 6.0 \cdot 2.5 + 2.7 \cdot 3.2 = 23.64$
  4. $W[3] = eval_x(N_4) = eval_x(N_1) + eval_x(N_3) = W[0] + W[2] = 31.22 + 23.64 = 54.86$
  5. $W[4] = eval_x(N_5) = eval_x(\langle 3, 0\rangle) + eval_x(\langle 1, 2\rangle) = V[3] \cdot x[0] + V[1] \cdot x[2] = 2.7 \cdot 1.0 + 8.1 \cdot 2.5 = 22.95$
  6. $W[5] = eval_x(N_6) = eval_x(N_5) + eval_x(N_2) = W[4] + W[1] = 22.95 + 52.6 = 75.55$

- **Segundo passo**: iteração sobre $C = N_4N_2\$N_6\$N_6\$N_1\$N_2\$N_4\$N_1\$N_3\$$

  1. $y[0] = eval_x(N_4) + eval_x(N_2) = W[3] + W[1] = 54.86 + 52.6 = 107.46$
  2. $y[1] = eval_x(N_6) = W[5] = 75.55$
  3. $y[2] = eval_x(N_6) = W[5] = 75.55$
  4. $y[3] = eval_x(N_1) = W[0] = 31.22$
  5. $y[4] = eval_x(N_2) = W[1] = 52.6$
  6. $y[5] = eval_x(N_4) = W[3] = 54.86$
  7. $y[6] = eval_x(N_1) = W[0] = 31.22$
  8. $y[7] = eval_x(N_3) = W[2] = 23.64$

Por fim, temos que o vetor resultante da multiplicação $A \cdot x$ é

$$y = [107.46, 75.55, 75.55, 31.22, 52.6, 54.86, 31.22, 23.64]^T$$

4 Proposta

Em Ferragina et al. (2022), os autores apresentam versões multithread dos métodos de compressão e multiplicação propostos. O particionamento utilizado por eles funciona da seguinte forma: primeiramente, a matriz inteira é convertida para a representação (S, V) (formato CSRV). A sequência S é, então, dividida em b blocos $S_1, S_2, \cdots, S_b$, de modo que cada bloco contenha a mesma quantidade de linhas, exceto, possivelmente, o último bloco S_b, que pode conter menos linhas. Em seguida, cada bloco S_i é comprimido em paralelo com o Re-Pair, gerando uma sequência de símbolos C_i e um conjunto de regras R_i. Ao final, obtêm-se b grupos (C_i, R_i) e um único vetor V de valores não nulos distintos, compartilhado por todos os blocos. Para calcular a multiplicação da matriz inteira por um vetor (tanto à direita quanto à esquerda), cada bloco é multiplicado em paralelo. Os autores avaliaram os tempos de execução e os picos de memória tanto durante a compressão quanto durante a multiplicação nesta abordagem multithread. Apesar da redução significativa dos tempos de compressão e multiplicação, houve, em geral, aumento das memórias de pico.

Neste trabalho, propomos uma abordagem em que os blocos são comprimidos e multiplicados de forma sequencial, em vez de utilizarmos multithreading. O objetivo principal é reduzir o pico de memória. Avaliaremos três variáveis à medida que aumentamos a quantidade de blocos: taxas de compressão, tempos de compressão, tempos de execução da multiplicação por vetor à direita e picos de uso de memória. Como será detalhado em 4.1.1.3, também esperamos que essa abordagem reduza o tempo de compressão em comparação com a compressão global, embora não tanto quanto a abordagem multithreading.

Nossa implementação funciona da seguinte forma: dividimos a matriz $A_{n\times m}$ em b blocos, cada um com $\frac{n}{b}$ linhas e m colunas . Caso n não seja divisível por b, particionamos a matriz de forma que alguns blocos tenham $\left\lfloor \frac{n}{b} \right\rfloor$ linhas e outros $\left\lceil \frac{n}{b} \right\rceil$ linhas. Diferentemente de Ferragina et al. (2022), particionamos a matriz antes de convertê-la para o formato CSRV. Em decorrência disso, temos b grupos (C_i, R_i, V_i) ao fim da compressão. Observe que, como os valores não nulos distintos de cada bloco podem variar, os vetores V_i resultantes podem ser diferentes entre si. Isso contrasta com a abordagem de Ferragina et al. (2022), na qual todos os blocos compartilham um único vetor V, que contém todos os valores não nulos distintos da matriz como um todo.

Por fim, multiplicamos cada submatriz A_i na forma (C_i, R_i, V_i) por um vetor à direita $x \in \mathbb{R}^m$, obtendo uma sequência de subvetores $y_1, y_2, \cdots, y_b$, cada um com número de elementos igual ao número de linhas do respectivo bloco. Estes subvetores, quando

considerados em conjunto, formam o vetor $y = A \cdot x$.

4.1 Exemplo

Para exemplificar o que está sendo proposto, mostraremos como funcionaria a compressão, utilizando 3 blocos, da matriz A previamente apresentada. Em seguida, mostraremos a multiplicação da matriz comprimida em blocos pelo vetor $x = [1.0, 3.2, 2.5, 3.2, 1.7, 8.0]^T$.

4.1.1 Compressão em blocos

Para $b = 3$, teremos dois blocos com três linhas e um bloco com duas linhas. Mostraremos a aplicação do compressor sobre cada submatriz.

4.1.1.1 Representações CSRV dos blocos

Primeiramente, colocamos cada submatriz na representação CSRV:

- $A_1 = \begin{bmatrix} 5.3 & 8.1 & 6.0 & 2.7 & 6.0 & 5.3 \\ 2.7 & 0 & 8.1 & 0 & 6.0 & 5.3 \\ 2.7 & 0 & 8.1 & 0 & 6.0 & 5.3 \end{bmatrix}$

  $V_1 = [5.3, 8.1, 6.0, 2.7]$

  $S_1 = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle\langle 2,4\rangle\langle 0,5\rangle\$\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$$

- $A_2 = \begin{bmatrix} 5.3 & 8.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6.0 & 5.3 \\ 5.3 & 8.1 & 6.0 & 2.7 & 0 & 0 \end{bmatrix}$

  $V_2 = [5.3, 8.1, 6.0, 2.7]$

  $S_2 = \langle 0,0\rangle\langle 1,1\rangle\$\langle 2,4\rangle\langle 0,5\rangle\$\langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle\$$

- $A_3 = \begin{bmatrix} 5.3 & 8.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 2.7 & 0 & 0 \end{bmatrix}$

  $V_3 = [5.3, 8.1, 6.0, 2.7]$

  $S_3 = \langle 0,0\rangle\langle 1,1\rangle\$\langle 2,2\rangle\langle 3,3\rangle\$$

4.1.1.2 Compressões com Re-Pair

Em seguida, aplicamos o Re-Pair a cada sequência S_i:

- $S_1 = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle\langle 2,4\rangle\langle 0,5\rangle\$\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$\langle 3,0\rangle\langle 1,2\rangle\langle 2,4\rangle\langle 0,5\rangle\$$

1. $N_1 \rightarrow \langle 2,4\rangle\langle 0,5\rangle$
   $C_1 = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle N_1\$\langle 3,0\rangle\langle 1,2\rangle N_1\$\langle 3,0\rangle\langle 1,2\rangle N_1\$$
2. $N_2 \rightarrow \langle 3,0\rangle\langle 1,2\rangle$
   $C_1 = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle N_1\$N_2N_1\$N_2N_1\$$
3. $N_3 \rightarrow N_2N_1$
   $C_1 = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle N_1\$N_3\$N_3\$$

- $S_2 = \langle 0,0\rangle\langle 1,1\rangle\$\langle 2,4\rangle\langle 0,5\rangle\$\langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle\$$

1. $N_1 \rightarrow \langle 0,0\rangle\langle 1,1\rangle$
   $C_2 = N_1\$\langle 2,4\rangle\langle 0,5\rangle\$N_1\langle 2,2\rangle\langle 3,3\rangle\$$

- $S_3 = \langle 0,0\rangle\langle 1,1\rangle\$\langle 2,2\rangle\langle 3,3\rangle\$$

  Não é possível comprimir S_3 com o Re-Pair, uma vez que não há pares de símbolos adjacentes se repetindo. Portanto, ficamos com $C_3 = \langle 0,0\rangle\langle 1,1\rangle\$\langle 2,2\rangle\langle 3,3\rangle\$$

Por fim temos os seguintes blocos na representação comprimida:

- (C_1, R_1, V_1)

$$C_1 = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle N_1\$N_3\$N_3\$$$

$$R_1 = \{N_1 \rightarrow \langle 2,4\rangle\langle 0,5\rangle, N_2 \rightarrow \langle 3,0\rangle\langle 1,2\rangle, N_3 \rightarrow N_2N_1\}$$

$$V_1 = [5.3, 8.1, 6.0, 2.7]$$

- (C_2, R_2, V_2)

$$C_2 = N_1\$\langle 2,4\rangle\langle 0,5\rangle\$N_1\langle 2,2\rangle\langle 3,3\rangle\$$$

$$R_2 = \{N_1 \rightarrow \langle 0,0\rangle\langle 1,1\rangle\}$$

$$V_2 = [5.3, 8.1, 6.0, 2.7]$$

- (C_3, R_3, V_3)

$$C_3 = \langle 0,0\rangle\langle 1,1\rangle\$\langle 2,2\rangle\langle 3,3\rangle\$$$

$$R_3 = \{\}$$

$$V_3 = [5.3, 8.1, 6.0, 2.7]$$

Os vetores de valores não nulos distintos V_1, V_2 e V_3 são iguais neste caso, mas poderiam ser diferentes se os valores não nulos distintos que aparecem em cada bloco (ou sua ordem de aparição) fossem diferentes.

4.1.1.3 Análise da compressão em blocos

Ao comprimir uma matriz em blocos, é comum que pares de símbolos adjacentes que apresentem alta frequência ao longo de toda a sequência $S_1, S_2, \ldots, S_b$, sejam parcial ou completamente ignorados em favor de pares que são mais frequentes localmente, em blocos individuais.

Considere as duas abordagens utilizadas para comprimir a matriz A: como um todo ou em blocos. Na compressão global da matriz, o par mais frequente na sequência S da representação CSRV de A é $\langle 0,0\rangle\langle 1,1\rangle$. Ao criar a regra $N_1 \rightarrow \langle 0,0\rangle\langle 1,1\rangle$, podemos realizar 4 substituições ao longo de S. Já na abordagem em blocos, essa redundância é explorada apenas no segundo bloco, pois nos blocos primeiro e terceiro, o par $\langle 0,0\rangle\langle 1,1\rangle$ aparece apenas uma única vez em cada. Como resultado, são feitas apenas 2 substituições em vez de 4.

O mesmo ocorre com o par $\langle 2,4\rangle\langle 0,5\rangle$, que na compressão da matriz completa é substituído 4 vezes, mas na abordagem em blocos é substituído apenas 3 vezes, já que sua quarta ocorrência está isolada no segundo bloco.

Além disso, o par $\langle 2,2\rangle\langle 3,3\rangle$, que aparece 3 vezes em A e é substituído por N_3 na compressão global, nem sequer é substituído na abordagem em blocos, pois cada uma das 3 ocorrências está localizada em um bloco distinto.

Esses exemplos ilustram uma das principais razões pelas quais a compressão em blocos tende a ser menos eficiente em termos de taxa de compressão: ela aproveita menos as redundâncias, resultando em um número reduzido de substituições.

Por outro lado, essa redução no número de substituições realizadas durante as aplicações do Re-Pair na abordagem em blocos resulta em um tempo de compressão menor em comparação com a abordagem global, mesmo sem o uso de multithreading.

Outra limitação da taxa de compressão é que, em nossa implementação, cada bloco possui seu próprio vetor de valores não nulos distintos V_i, exigindo o armazenamento repetido de valores que, na compressão global, seriam representados por uma única entrada no vetor compartilhado V. No caso da matriz A, por exemplo, a compressão global utiliza um único vetor V, enquanto a compressão em blocos resulta na duplicação desse vetor em V_1, V_2 e V_3, já que cada um dos três blocos contém todos valores não nulos distintos presentes na matriz como um todo.

Do ponto de vista do uso de memória RAM, a compressão em blocos apresenta uma vantagem: enquanto na compressão global a sequência S, o vetor V e o conjunto de regras R são armazenados integralmente na memória, potencialmente ocupando um espaço significativo, na compressão em blocos os dados são processados de forma incremental. Nesse caso, armazenam-se S_1, R_1 e V_1, seguidos por S_2, R_2 e V_2, e assim sucessivamente. Essa abordagem reduz o pico de uso de memória, pois cada S_i é menor que S, e cada R_i e V_i têm tamanhos iguais ou menores que R e V, respectivamente. Na implementação do Re-Pair que estamos utilizando, a sequência C é gerada diretamente a partir de S, reutilizando o espaço de memória já alocado para S. Por esta razão não consideramos C nesta análise do pico de uso de memória.

4.1.2 Multiplicação

A partir dos três blocos comprimidos anteriormente, mostraremos como calcular o vetor $y = A \cdot x$.

4.1.2.1 Preenchimento dos vetores auxiliares

Inicialmente, preenchemos os vetores auxiliares W_i para cada bloco, com exceção de W_3, que é vazio, pois C_3 não contém símbolos não-terminais.

- Preenchimento do vetor auxiliar W_1:

  1. $W_1[0] = eval_x(N_1) = eval_x(\langle 2, 4\rangle) + eval_x(\langle 0, 5\rangle) = V_1[2] \cdot x[4] + V_1[0] \cdot x[5] = 6.0 \cdot 1.7 + 5.3 \cdot 8.0 = 52.6$
  2. $W_1[1] = eval_x(N_2) = eval_x(\langle 3, 0\rangle) + eval_x(\langle 1, 2\rangle) = V_1[3] \cdot x[0] + V_1[1] \cdot x[2] = 2.7 \cdot 1.0 + 8.1 \cdot 2.5 = 22.95$
  3. $W_1[2] = eval_x(N_3) = eval_x(N_2) + eval_x(N_1) = W_1[1] + W_1[0] = 22.95 + 52.6 = 75.55$

- Preenchimento do vetor auxiliar W_2:

$$\begin{aligned} W_2[0] &= eval_x(N_1) = eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle) \\ &= V_2[0] \cdot x[0] + V_2[1] \cdot x[1] = 5.3 \cdot 1.0 + 8.1 \cdot 3.2 = 31.22 \end{aligned}$$

4.1.2.2 Cálculo do vetor resultante

Com os vetores auxiliares W_i preenchidos, iteramos sobre cada sequência C_i para calcular os elementos do vetor resultante y.

- Iteração sobre $C_1 = \langle 0,0\rangle\langle 1,1\rangle\langle 2,2\rangle\langle 3,3\rangle N_1\$N_3\$N_3\$$:

  1. $y[0] = eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle) + eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle) + eval_x(N_1) = V_1[0] \cdot x[0] + V_1[1] \cdot x[1] + V_1[2] \cdot x[2] + V_1[3] \cdot x[3] + W_1[0] = 5.3 \cdot 1.0 + 8.1 \cdot 3.2 + 6.0 \cdot 2.5 + 2.7 \cdot 3.2 + 52.6 = 107.46$
  2. $y[1] = eval_x(N_3) = W_1[2] = 75.55$
  3. $y[2] = eval_x(N_3) = W_1[2] = 75.55$

- Iteração sobre $C_2 = N_1\$\langle 2,4\rangle\langle 0,5\rangle\$N_1\langle 2,2\rangle\langle 3,3\rangle\$$:

  1. $y[3] = eval_x(N_1) = W_2[0] = 31.22$
  2. $y[4] = eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle) = V_2[2] \cdot x[4] + V_2[0] \cdot x[5] = 6.0 \cdot 1.7 + 5.3 \cdot 8.0 = 52.6$
  3. $y[5] = eval_x(N_1) + eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle) = W_2[0] + V_2[2] \cdot x[2] + V_2[3] \cdot x[3] = 31.22 + 6.0 \cdot 2.5 + 2.7 \cdot 3.2 = 54.86$

- Iteração sobre $C_3 = \langle 0,0\rangle\langle 1,1\rangle\$\langle 2,2\rangle\langle 3,3\rangle\$$:

  1. $y[6] = eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle) = V_3[0] \cdot x[0] + V_3[1] \cdot x[1] = 5.3 \cdot 1.0 + 8.1 \cdot 3.2 = 31.22$
  2. $y[7] = eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle) = V_3[2] \cdot x[2] + V_3[3] \cdot x[3] = 6.0 \cdot 2.5 + 2.7 \cdot 3.2 = 23.64$

  O resultado final obtido é o seguinte vetor:

$$y = [107.46, 75.55, 75.55, 31.22, 52.6, 54.86, 31.22, 23.64]^T$$

4.1.2.3 Análise da multiplicação

- **Multiplicação com compressão global**

  Considere a representação (S, V) da matriz A comprimida como um todo. Mesmo antes de aplicar o algoritmo Re-Pair, já é possível realizar a multiplicação da matriz neste formato com o vetor $x = [1.0, 3.2, 2.5, 3.2, 1.7, 8.0]^T$:

  1. $y[0] = eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle) + eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle) + eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle) = 107.46$
  2. $y[1] = eval_x(\langle 3,0\rangle) + eval_x(\langle 1,2\rangle) + eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle) = 75.55$
  3. $y[2] = eval_x(\langle 3,0\rangle) + eval_x(\langle 1,2\rangle) + eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle) = 75.55$
  4. $y[3] = eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle) = 31.22$
  5. $y[4] = eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle) = 52.6$
  6. $y[5] = eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle) + eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle) = 54.86$
  7. $y[6] = eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle) = 31.22$
  8. $y[7] = eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle) = 23.64$

  Nesta abordagem, diversos cálculos foram realizados repetidamente. Por exemplo, a operação $eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle)$, que envolve acessos à memória, duas multiplicações e uma soma, foi repetida quatro vezes: nos cálculos de $y[0]$, $y[3]$, $y[5]$ e $y[6]$. Da mesma forma, $eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle)$ foi calculado quatro vezes, e $eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle)$, três vezes.

  Após aplicar o algoritmo Re-Pair, além de comprimir ainda mais a representação da matriz, o processo de multiplicação torna-se mais eficiente devido à utilização dos resultados parciais armazenados no vetor W. Por exemplo:

  - $W[0]$ armazena $eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle)$, evitando que essa operação seja recalculada em cada uso posterior; basta acessar $W[0]$.

  - Da mesma forma, $eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle)$ é armazenado em $W[1]$, e $eval_x(\langle 2,2\rangle) + eval_x(\langle 3,3\rangle)$ é armazenado em $W[2]$.

  A cada entrada adicionada ao vetor W, diversos cálculos repetidos são evitados, tornando a multiplicação mais rápida e eficiente.

- **Multiplicação com compressão em blocos**

  Na compressão em blocos, a matriz A é dividida em partições, e os cálculos não podem ser reaproveitados entre blocos. Isso resulta na necessidade de repetir operações que poderiam ser otimizadas na compressão global.

  Por exemplo, ao realizar a multiplicação da matriz A comprimida em três blocos:

- O cálculo $eval_x(\langle 0,0\rangle) + eval_x(\langle 1,1\rangle)$ foi executado três vezes: uma no bloco 1 (durante a iteração sobre C_1), uma no bloco 2 (no preenchimento de W_2) e outra no bloco 3 (na iteração sobre C_3). A única repetição evitada ocorreu dentro do bloco 2, onde $\langle 0,0\rangle\langle 1,1\rangle$ aparece duas vezes.

- O cálculo $eval_x(\langle 2,4\rangle) + eval_x(\langle 0,5\rangle)$ foi realizado duas vezes: uma no bloco 1 (durante o preenchimento de W_1) e outra no bloco 2 (durante a iteração sobre C_2). No bloco 1, onde $\langle 2,4\rangle\langle 0,5\rangle$ aparece três vezes, as repetições foram eliminadas, mas esse reaproveitamento não ocorreu no bloco 2.

Essa repetição de cálculos entre blocos prejudica a eficiência da multiplicação em comparação com a compressão global. No entanto, dentro de cada bloco, ainda é possível evitar redundâncias locais por meio do uso dos vetores auxiliares W_i.

Portanto, embora a compressão em blocos permita o reaproveitamento de cálculos dentro de cada bloco durante a multiplicação, ela não possibilita a reutilização de resultados de cálculos entre blocos distintos, o que a torna menos eficiente em termos de tempo de multiplicação em comparação com a compressão global.

5 Experimentos

Executamos todos os experimentos em uma máquina equipada com um processador AMD Ryzen 9 7900 de 12 núcleos, com frequência máxima de 5.48 GHz. A máquina possui 24 processadores (12 núcleos com 2 threads por núcleo) e está configurada com 128 GB de RAM.

Implementamos tanto o algoritmo responsável pela conversão de matrizes para a representação CSRV quanto o método Re-Pair. Para isso utilizamos a linguagem de programação C++. Todos os códigos estão disponíveis para download em <https://github.com/filipecoresende/compressed-matrix-multiplication>. Para medir os tempos de execução utilizamos a biblioteca `<time.h>`. Já os picos de uso de memória foram medidos por meio da biblioteca *malloc_count*, disponível em <https://github.com/bingmann/malloc_count>.

As taxas de compressão são dadas por:

$$\text{Taxa de Compressão} = \frac{\text{Tamanho Comprimido}}{\text{Tamanho Original}}$$

5.1 Datasets

Para os experimentos, utilizamos matrizes obtidas em <https://www.kaggle.com/datasets/giovannimanzini/some-machine-learning-matrices>. Estas são as mesmas utilizadas por Ferragina et al. (2022), com exceção da matriz ImageNet. Na Tabela 1, cada uma das matrizes são detalhadas. O tamanho corresponde à representação da matriz em memória utilizando 8 bytes por valor.

Tabela 1 – Matrizes utilizadas nos experimentos

dataset	tamanho (MB)	linhas	colunas	não nulos	não nulos distintos
Covtype	239.37	581,012	54	22.00%	6,682
Census	1275.36	2,458,285	68	43.03%	45
Optical	432.55	325,834	174	97.50%	897,176
Airline78	3199.96	14,462,943	29	72.66%	7,794
Susy	686.65	5,000,000	18	98.82%	20,352,142
ImageNet	8666.17	1,262,102	900	30.99%	824
Mnist2m	11962.89	2,000,000	784	25.25%	255
Higgs	2349.85	11,000,000	28	92.11%	8,083,943

Dividimos cada matriz em diferentes números de blocos (1, 2, 4, 8, 16 e 32) e as comprimimos. Em seguida, para cada matriz e número de blocos, executamos a multipli-

cação da matriz por um vetor à direita. Para cada matriz $M \in \mathbb{R}^{n \times m}$, o vetor $x \in \mathbb{R}^m$ escolhido para a multiplicação possui todos os elementos iguais 1, i.e. $x = (1, 1, \cdots, 1)^T$.

5.2 Compressão de matrizes

5.2.1 Taxas de compressão

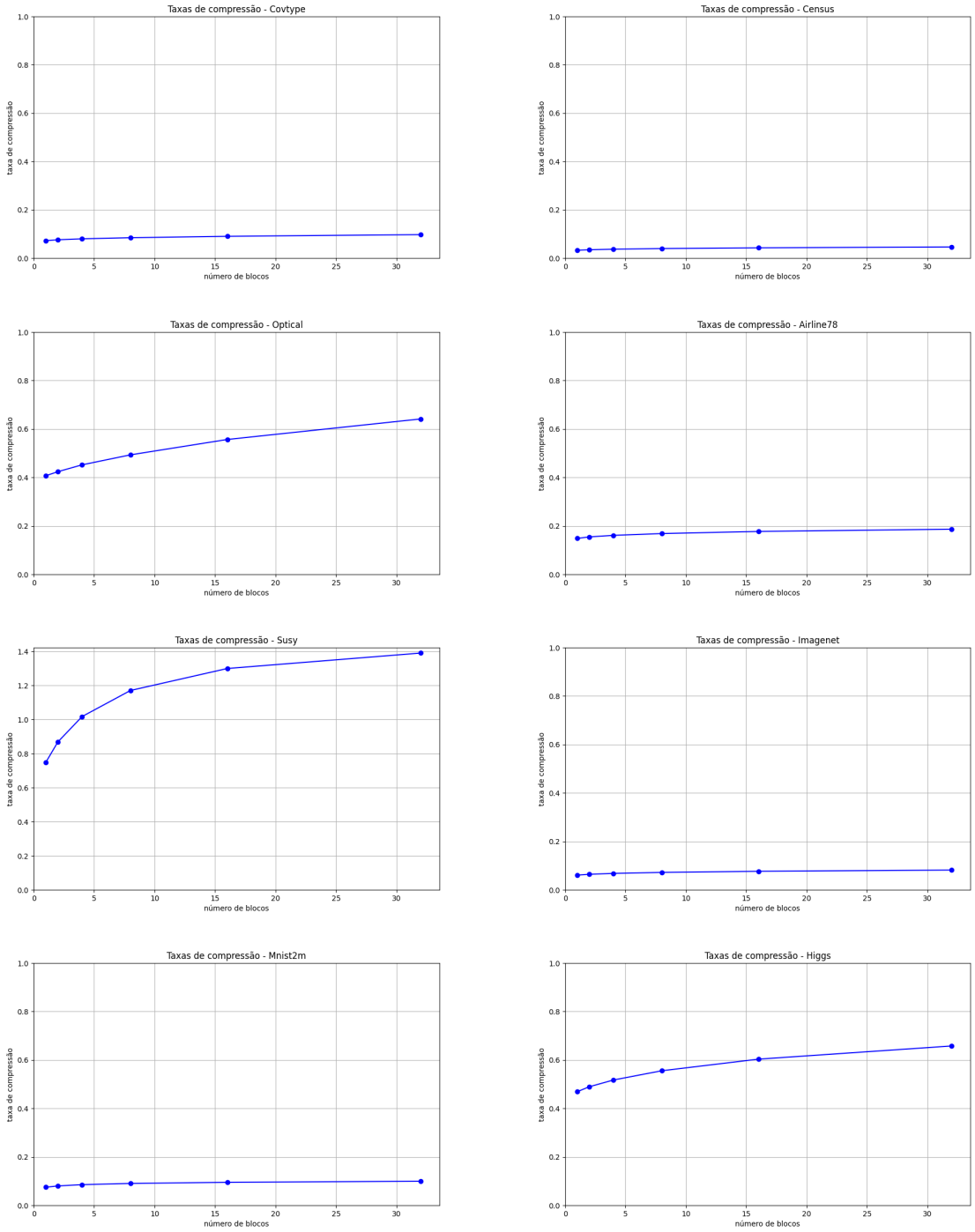

Figura 1 – Taxa de compressão × Número de blocos

5.2.2 Tempos de compressão

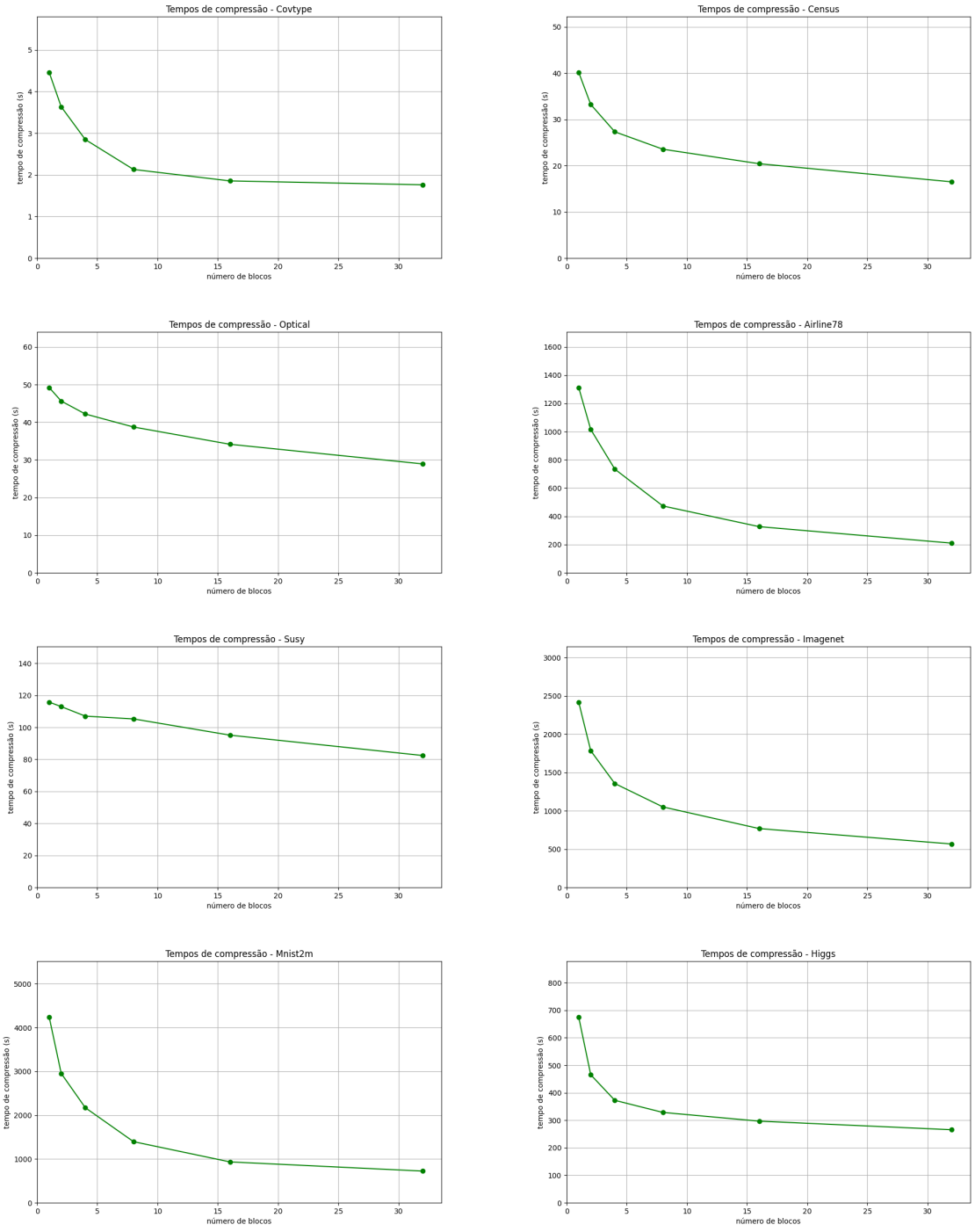

Figura 2 – Tempo de compressão × Número de blocos

5.2.3 Picos de memória da compressão

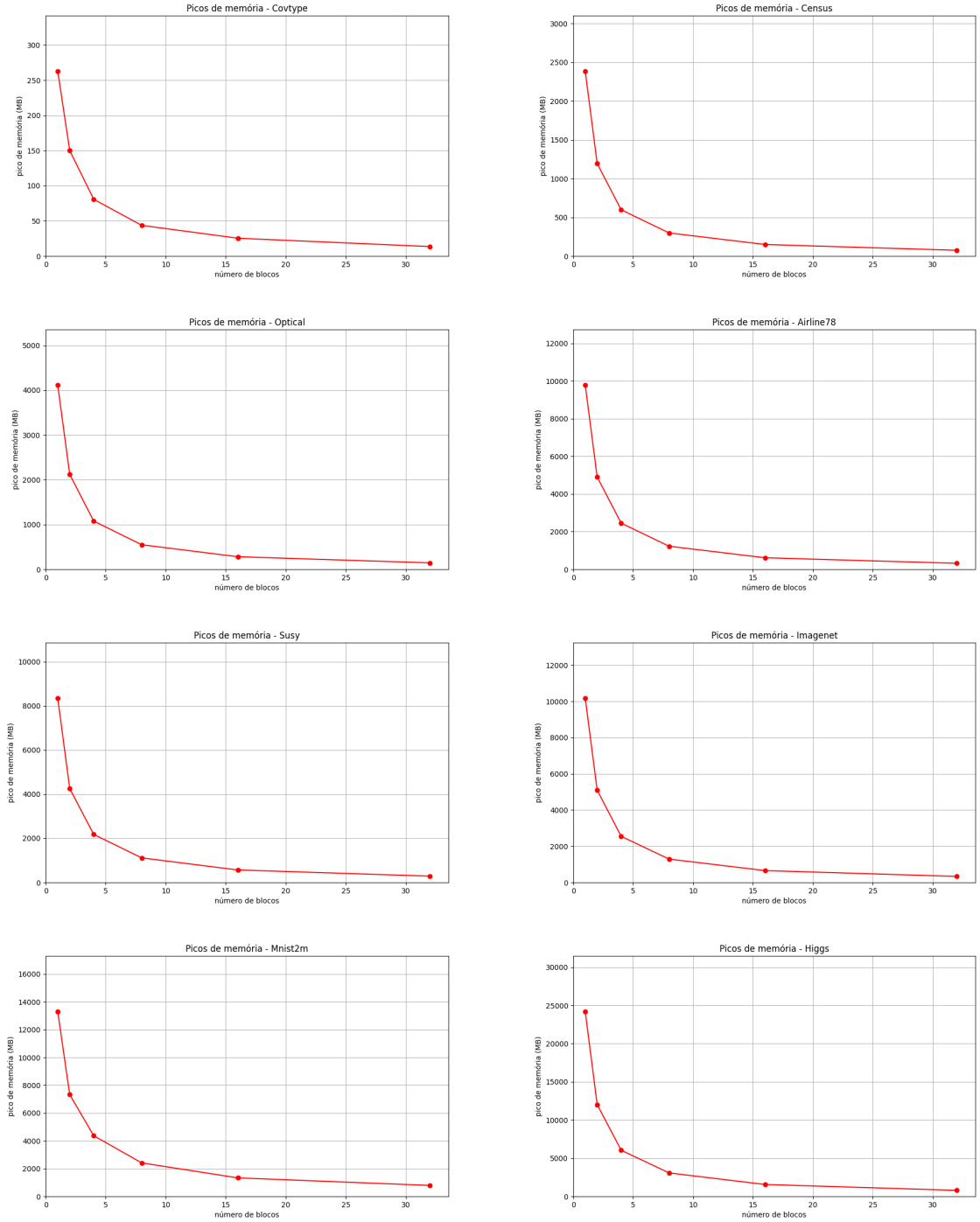

Figura 3 – Pico de memória da compressão × Número de blocos

A seguir, as Tabelas 2 e 3 comparam os resultados obtidos para cada matriz com a abordagem de compressão global e compressão com 32 blocos.

Os speedups são dados por:

$$\text{Speedup} = \frac{\text{Tempo de Compressão com 1 Bloco}}{\text{Tempo de Compressão com 32 Blocos}}$$

Tabela 2 – Taxa de compressão: comparação entre 1 bloco e 32 blocos

Dataset	**Taxa de Compressão (1 bloco)**	**Taxa de Compressão (32 blocos)**
Covtype	7.21%	9.71%
Census	3.24%	4.60%
Optical	40.70%	64.15%
Airline78	14.84%	18.64%
Susy	74.80%	138.95%
ImageNet	6.08%	8.17%
Mnist2m	7.47%	9.95%
Higgs	46.91%	65.78%

Tabela 3 – Speedup e redução de pico de memória na compressão: comparação entre 1 bloco e 32 blocos

Dataset	**Speedup**	**Redução do pico de memória**
Covtype	2.53	94.87%
Census	2.44	96.87%
Optical	1.70	96.49%
Airline78	6.25	96.74%
Susy	1.41	96.57%
ImageNet	4.27	96.72%
Mnist2m	5.87	94.07%
Higgs	2.54	96.73%

Como era de se esperar, a compressão em blocos aumentou o tamanho dos dados comprimidos em relação à compressão global em todos os casos. Certas matrizes apresentam uma taxa de compressão mais sensível ao particionamento do que outras, seja pelo maior espalhamento de padrões repetitivos ao longo de seus dados, seja pela redundância de um grande número de valores não nulos distintos que se repetem entre os vetores V_i de cada bloco. Por exemplo, a compressão da matriz Susy em 32 blocos resulta em um aumento de 85.76% no tamanho da representação comprimida em comparação com a compressão global. Para essa matriz, o particionamento, a partir de 4 blocos, gerou representações com tamanhos maiores que o da matriz original (calculado como o número de entradas multiplicado por 8 bytes). Considerando o grande número de valores não nulos distintos desta matriz (20,352,142), é mais provável que a degradação na compressão seja resultante da repetição de valores entre os vetores V_i, e não de uma grande distribuição de padrões repetitivos entre os blocos.

Por outro lado, conjuntos de dados como Airline78 e Mnist2m apresentam aumentos mais modestos de 25.57% e 33.14%, respectivamente. Além de apresentarem um número reduzido de valores não nulos distintos (7,794 e 255, respectivamente), é provável que os padrões repetitivos nessas matrizes sejam mais concentrados, o que permite que a compressão em blocos aproveite de forma eficiente as repetições.

De modo geral, percebe-se que compressão em blocos, mesmo que executada de forma sequencial, resulta em uma redução significativa no tempo de compressão. Os conjuntos Airline78, Mnist2m e ImageNet destacam-se com speedups impressionantes de 6.25, 5.87 e 4.27, respectivamente.

Conforme já havíamos previsto, a compressão em blocos reduz substancialmente o consumo de memória em comparação com a compressão global. Em todos os casos apresentados, a redução do pico de memória é extremamente alta, variando entre 94.07% e 96.87%. Isso significa que a abordagem de blocos é altamente eficiente para sistemas com limitações de memória.

Com base nos resultados apresentados nas tabelas, podemos concluir que, apesar de comprometer a taxa de compressão, a compressão em blocos oferece uma grande vantagem em termos de redução de tempo de compressão e consumo de memória.

5.3 Multiplicação de matrizes comprimidas por vetores à direita

5.3.1 Tempos de execução da multiplicação

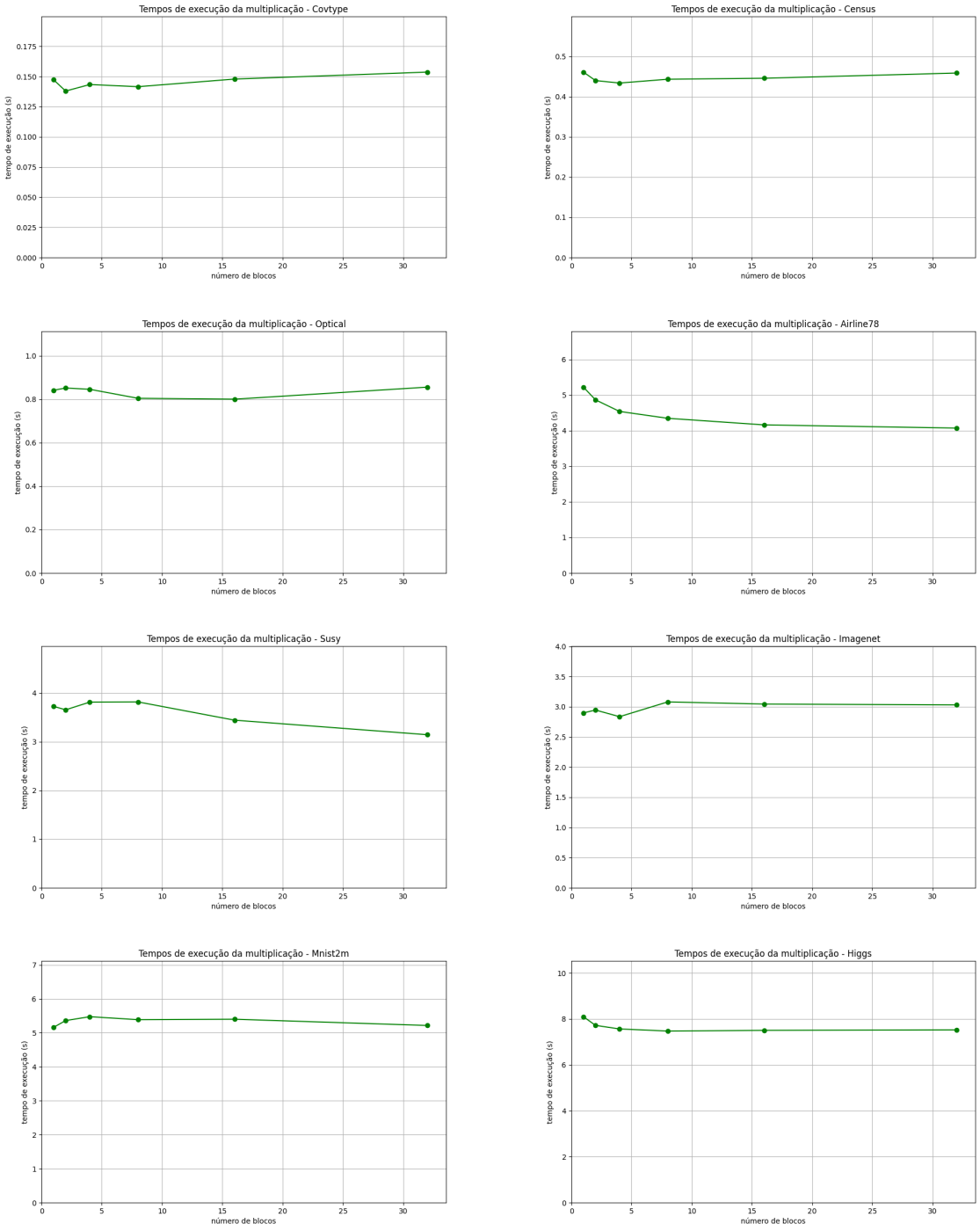

Figura 4 – Tempo de execução da multiplicação × Número de blocos

5.3.2 Picos de memória da multiplicação

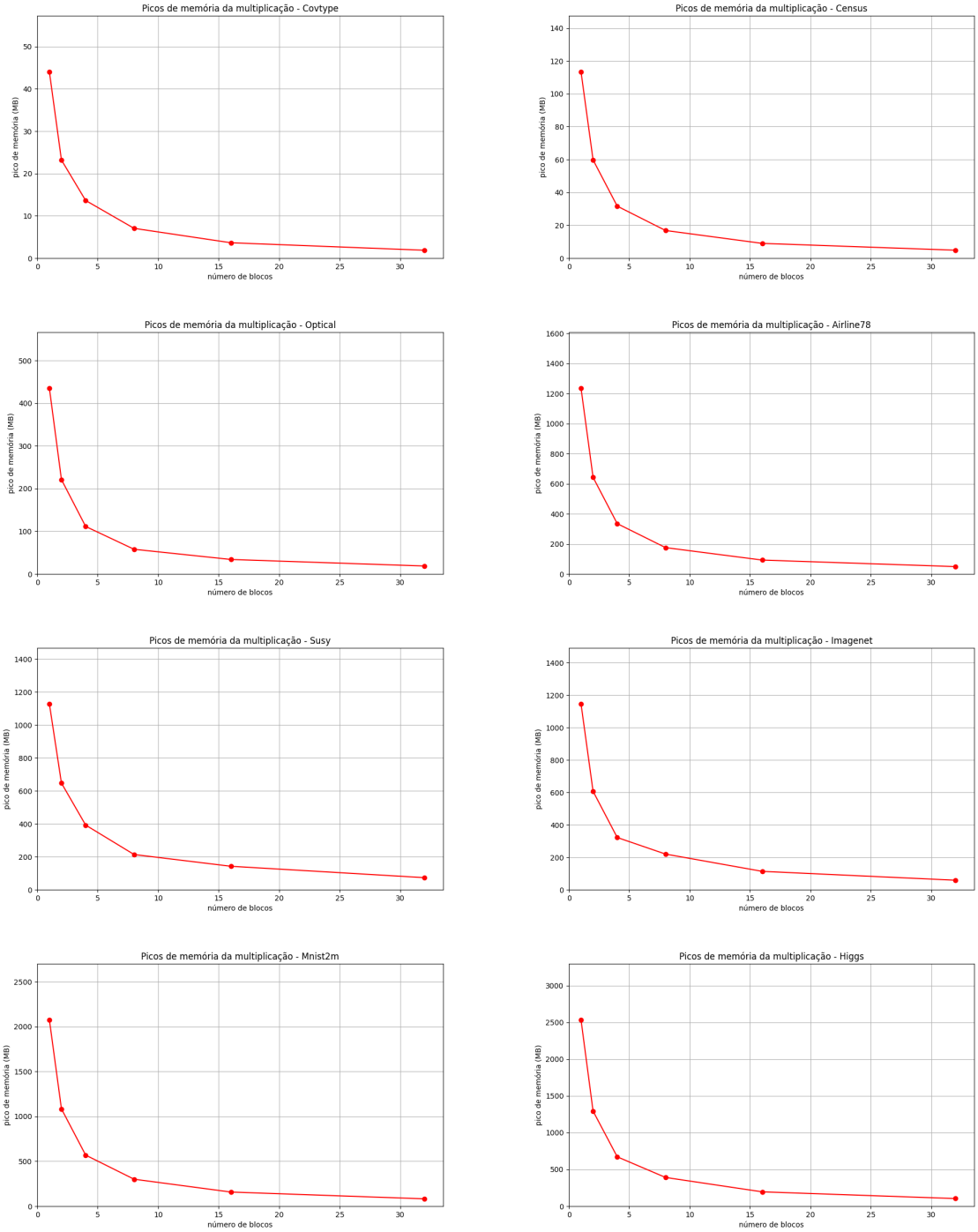

Figura 5 – Pico de memória da multiplicação × Número de blocos

Podemos observar que, na prática, o tempo de multiplicação nem sempre aumentou com o aumento no número de blocos utilizados na compressão. Acreditamos que isso se deve à ineficiência no uso da memória cache durante a multiplicação de matrizes grandes comprimidas em um único bloco.

A **memória cache** é uma memória de pequeno tamanho, extremamente rápida, posicionada próxima ou integrada ao processador. Sua principal função é armazenar dados

e instruções frequentemente acessados ou com alta probabilidade de serem reutilizados. Isso permite que o processador acesse essas informações muito mais rapidamente do que se precisasse buscá-las na RAM ou em outros níveis de armazenamento.

O funcionamento da memória cache é fundamentado em dois princípios de acesso a dados amplamente observados na computação:

- **Localidade temporal**: Dados ou instruções acessados recentemente têm alta probabilidade de serem acessados novamente em um curto intervalo de tempo. A cache retém esses dados para que o processador os encontre de forma eficiente em futuros acessos.
- **Localidade espacial**: Dados armazenados próximos entre si (em termos de endereço de memória) têm alta probabilidade de serem acessados sequencialmente. Por essa razão, a cache frequentemente armazena blocos de dados, antecipando que itens dentro do mesmo bloco serão requisitados.

Relembrando, a multiplicação de matrizes no formato comprimido consiste em primeiro preencher o vetor auxiliar W e, em seguida, calcular o vetor resultante y à medida que iteramos sobre a cadeia de símbolos C. No preenchimento de W (ou de cada W_i, no caso de matrizes comprimidas em blocos), é comum que o acesso envolva posições distantes do vetor. Por exemplo, na compressão global de A, temos a regra $N_6 \to N_5 N_2$. Isso significa que, para preencher $W[5]$ (última posição de W), é necessário acessar $W[1]$ (segunda posição de W).

Esse padrão de acesso disperso no vetor W também se manifesta durante a etapa de iteração sobre C (ou sobre cada C_i, no caso da compressão em blocos). Na compressão global de A, por exemplo, observa-se que $y[2] = W[5]$ e, logo em seguida, $y[3] = W[0]$.

Quando a compressão em blocos é utilizada, os vetores auxiliares W_i são menores e geralmente cabem bem na memória cache, podendo até ser armazenados por completo. Por outro lado, na compressão global de matrizes grandes, o vetor auxiliar W pode alcançar dimensões muito maiores, excedendo a capacidade da cache e gerando problemas de desempenho. Nos experimentos realizados, as matrizes comprimidas globalmente produziram vetores W com milhões de regras. Essa dispersão nos acessos ao vetor aumenta significativamente a frequência de falhas de cache (*cache misses*), impactando negativamente a eficiência do processamento.

Nas operações de multiplicação, durante a etapa de iteração sobre C, eventualmente encontramos um símbolo terminal $\langle i, j \rangle$ e devemos calcular $eval_x(\langle i, j \rangle) = V[i] \cdot x[j]$, onde V é o vetor de valores não nulos distintos da matriz e x é o vetor pelo qual estamos multiplicando a matriz. O índice i pode variar entre posições distantes de V, o que pode gerar o mesmo problema de eficiência de uso de memória cache do caso do vetor W. Em

alguns casos, o vetor V de valores não nulos distintos da matriz como um todo é consideravelmente maior que qualquer um dos vetores V_i dos blocos na abordagem particionada, de forma a gerar falhas de cache que não ocorreriam na multiplicação da matriz comprimida em blocos.

Embora a multiplicação de matrizes comprimidas sem particionamento evite recálculos, a ineficiência no uso da cache nesse caso pode comprometer o desempenho. Como resultado, o tempo de multiplicação apresenta variações irregulares à medida que o número de blocos aumenta, alternando entre pioras e melhorias no tempo de execução.

Nos experimentos, o tempo de execução da multiplicação apresentou pouca variação com o aumento do número de blocos. Em contrapartida, o pico de uso de memória foi reduzido de forma significativa. A Tabela 3 mostra a redução percentual no pico de uso de memória durante a multiplicação ao compararmos o cenário com 1 bloco e o com 32 blocos.

Tabela 4 – Redução de pico de memória na multiplicação: comparação entre 1 bloco e 32 blocos

Matriz	**Redução de pico de memória**
Covtype	95.75%
Census	95.73%
Optical	95.72%
Airline78	96.03%
Susy	93.53%
ImageNet	94.91%
Mnist2m	96.05%
Higgs	95.91%

Portanto, a divisão da matriz em blocos se mostrou vantajosa para a multiplicação de forma geral, pois alterou minimamente o tempo de execução mas reduziu muito o uso de memória RAM durante a operação.

6 Conclusão

Neste trabalho, exploramos possibilidades dentro de uma área recente chamada de *Compressed Linear Algebra*, a qual estuda a compressão de matrizes de forma a permitir a realização de operações algébricas sobre elas diretamente na forma comprimida, e de forma mais rápida.

Implementamos o algoritmo de compressão de matrizes proposto por Ferragina et al. (2022), juntamente com o algoritmo que possibilita a multiplicação da matriz comprimida por um vetor à direita. Utilizamos um conjunto de matrizes e aplicamos o algoritmo de compressão de forma particionada, processando as matrizes em blocos de maneira sequencial. Avaliamos a redução do pico de uso de memória à medida que o número de blocos de divisão da matriz aumenta, tanto durante a compressão quanto na multiplicação das matrizes comprimidas com vetores à direita. Os resultados indicaram uma redução significativa no consumo de memória.

No caso da compressão das matrizes, além da redução no uso de memória RAM, observou-se uma diminuição significativa no tempo de execução à medida que o número de blocos aumentava. Já na execução da multiplicação, o tempo de execução variou de forma irregular: o aumento no número de blocos, em alguns casos, diminuiu o tempo gasto na multiplicação, enquanto em outros, aumentou esse tempo. No entanto, de modo geral, o tempo de execução da multiplicação não variou significativamente com o aumento do número de blocos. A única variável que ficou relativamente comprometida foi a taxa de compressão, que apresentou uma degradação à medida que o número de blocos aumentava.

Nossa abordagem pode ser particularmente benéfica em cenários onde a disponibilidade de recursos computacionais é limitada. Por exemplo, em sistemas embarcados ou dispositivos com memória RAM restrita.

Trabalhos futuros

Possibilidades de trabalhos futuros incluem:

- Implementação da multiplicação em CUDA

  CUDA (*Compute Unified Device Architecture*) é uma plataforma de computação paralela desenvolvida pela NVIDIA, que permite utilizar a GPU para processamento de propósito geral. Seria interessante empregá-la para executar a multiplicação das matrizes comprimidas de forma paralela em GPUs, comparando o desempenho com a execução utilizando multithreading em uma CPU convencional, como apresentado em Ferragina et al. (2022).

- Exploração de outros métodos de compressão baseada em gramática

  Seria interessante experimentar outros compressores baseados em gramática ao invés do Re-Pair. Uma ideia seria experimentar o algoritmo GCIS (*Grammar Compression by Induced Suffix Sorting*) (NUNES et al., 2022).

Referências

ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G. S.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; GOODFELLOW, I. J.; HARP, A.; IRVING, G.; ISARD, M.; JIA, Y.; JÓZEFOWICZ, R.; KAISER, L.; KUDLUR, M.; LEVENBERG, J.; MANÉ, D.; MONGA, R.; MOORE, S.; MURRAY, D. G.; OLAH, C.; SCHUSTER, M.; SHLENS, J.; STEINER, B.; SUTSKEVER, I.; TALWAR, K.; TUCKER, P. A.; VANHOUCKE, V.; VASUDEVAN, V.; VIÉGAS, F. B.; VINYALS, O.; WARDEN, P.; WATTENBERG, M.; WICKE, M.; YU, Y.; ZHENG, X. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. **CoRR**, abs/1603.04467, 2016. Disponível em: <http://arxiv.org/abs/1603.04467>. Citado na página 8.

BENZ, F.; KÖTZING, T. An effective heuristic for the smallest grammar problem. In: BLUM, C.; ALBA, E. (Ed.). **Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013**. ACM, 2013. p. 487–494. Disponível em: <https://doi.org/10.1145/2463372.2463441>. Citado na página 14.

BHATTACHERJEE, S.; DESHPANDE, A.; SUSSMAN, A. Pstore: an efficient storage framework for managing scientific data. In: JENSEN, C. S.; LU, H.; PEDERSEN, T. B.; THOMSEN, C.; TORP, K. (Ed.). **Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014**. ACM, 2014. p. 25:1–25:12. Disponível em: <https://doi.org/10.1145/2618243.2618268>. Citado na página 8.

CHARIKAR, M.; LEHMAN, E.; LIU, D.; PANIGRAHY, R.; PRABHAKARAN, M.; SAHAI, A.; SHELAT, A. The smallest grammar problem. **IEEE Trans. Inf. Theory**, v. 51, n. 7, p. 2554–2576, 2005. Disponível em: <https://doi.org/10.1109/TIT.2005.850116>. Citado na página 15.

ELGOHARY, A.; BOEHM, M.; HAAS, P. J.; REISS, F. R.; REINWALD, B. Compressed linear algebra for large-scale machine learning. **VLDB J.**, v. 27, n. 5, p. 719–744, 2018. Disponível em: <https://doi.org/10.1007/s00778-017-0478-1>. Citado 2 vezes nas páginas 8 e 17.

_____. Compressed linear algebra for declarative large-scale machine learning. **Commun. ACM**, v. 62, n. 5, p. 83–91, 2019. Disponível em: <https://doi.org/10.1145/3318221>. Citado 2 vezes nas páginas 8 e 17.

FERRAGINA, P.; MANZINI, G.; GAGIE, T.; KÖPPL, D.; NAVARRO, G.; STRIANI, M.; TOSONI, F. Improving matrix-vector multiplication via lossless grammar-compressed matrices. **Proc. VLDB Endow.**, v. 15, n. 10, p. 2175–2187, 2022. Disponível em: <https://www.vldb.org/pvldb/vol15/p2175-tosoni.pdf>. Citado 9 vezes nas páginas 3, 8, 9, 17, 18, 20, 23, 31 e 41.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to automata theory, languages, and computation, 3rd Edition**. [S.l.]: Addison-Wesley, 2007. (Pearson international edition). ISBN 978-0-321-47617-3. Citado na página 12.

LARSSON, N. J.; MOFFAT, A. Offline dictionary-based compression. In: **Data Compression Conference, DCC 1999, Snowbird, Utah, USA, March 29-31, 1999**. IEEE Computer Society, 1999. p. 296–305. Disponível em: <https://doi.org/10.1109/DCC.1999.755679>. Citado 2 vezes nas páginas 15 e 16.

NAVARRO, G. **Compact Data Structures - A Practical Approach**. Cambridge University Press, 2016. ISBN 978-1-10-715238-0. Disponível em: <http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB>. Citado na página 15.

NUNES, D. S. N.; LOUZA, F. A.; GOG, S.; AYALA-RINCóN, M.; NAVARRO, G. Grammar compression by induced suffix sorting. **ACM J. Exp. Algorithmics**, Association for Computing Machinery, New York, NY, USA, v. 27, ago. 2022. ISSN 1084-6654. Disponível em: <https://doi.org/10.1145/3549992>. Citado na página 42.

SAAD, Y. **Iterative methods for sparse linear systems**. SIAM, 2003. ISBN 978-0-89871-534-7. Disponível em: <https://doi.org/10.1137/1.9780898718003>. Citado na página 17.

SAYOOD, K. **Introduction to Data Compression, Third Edition**. [S.l.]: Elsevier Morgan Kaufmann, 2006. (The Morgan Kaufmann series in multimedia information and systems). ISBN 978-0-12-620862-7. Citado na página 14.

SIPSER, M. **Introduction to the theory of computation**. [S.l.]: PWS Publishing Company, 1997. ISBN 978-0-534-94728-6. Citado 2 vezes nas páginas 11 e 13.