Trabalho Prático 0
Conversão de uma imagem colorida para tons de cinza

Wallace Eduardo Pereira

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

wallacepereira@dcc.ufmg.br

1. Introdução

O problema proposto foi implementar um algoritmo que, dada como entrada um arquivo de imagem colorida com extensão .ppm, seja realizado uma conversão e tenha como saída um arquivo em tons de cinza com extensão .pgm. Ademais, foi feita a análise de sua implementação em questão de complexidade e desempenho.

2. Método

O programa foi desenvolvido na linguagem C, compilado pelo compilador gcc da GNU Compiler Collection. Além disso, foi desenvolvido com um processador AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx 2.30 GHz, com 12,0 GB de RAM instalável (utilizável: 9,94 GB), e Windows 11 Home Single Language, porém utilizando o WSL (Windows Subsystem for Linux) e na IDE Visual Studio Code.

2.1. Estrutura de Dados

Para a implementação do programa, foi utilizado como estrutura de dados matrizes, que armazenam os pixels das imagens após a leitura do arquivo. Foi usada essa estrutura porque o formato PPM dos arquivos das imagens já apresenta, em seu cabeçalho, as dimensões nas quais os pixels estão armazenados, dessa forma, foi mais fácil criar as dimensões das matrizes para o armazenamento das informações.

Primeiro, foi aberto o arquivo da imagem com extensão .ppm, e a partir da leitura do seu cabeçalho, foi definido as dimensões das matrizes que iriam receber as informações. Após isso, foi chamada a função de alocar dinamicamente as matrizes, para que não tenham desperdício ou vazamento de memória. Passando como parâmetro o número de linhas e de colunas, foi feita a alocação e, em seguida, a matriz original salvou os dados do arquivo original, enquanto a matriz PeB armazenou os dados já convertidos para a escala de cinza.

Ademais, as matrizes PeB e original foram criadas utilizando ponteiro para ponteiro, para passagem das informações por referência, assim evitando a cópia e modificação das estruturas de dados ao serem chamadas por funções.

2.2. Structs

Para modularizar a implementação, foram utilizadas Structs que armazenam informações sobre as imagens e seus pixels. A struct Imagem contém os dados obtidos a partir do cabeçalho das imagens, sejam esses número de linhas, número de colunas, valor máximo dos

pixels, e o tipo da imagem. Na struct Pixel, são guardados as primitivas r, g e b, que representam os valores vermelho, verde e azul, e que representa um pixel.

2.3. Conversão e Escrita

Para que fosse possível gerar um documento em escala de cinza, foi necessário ler o arquivo com extensão .ppm e salvar seus dados, depois chamada a função que realiza o cálculo dos pixels e os transforma em escala de cinza, armazenando em uma matriz. Depois foi criado um arquivo em modo de escrita e transposto esses pixels em escala de cinza para esse arquivo, de forma a gerar a imagem de saída com extensão .pgm. Não seria necessário o armazenamento das informações da imagem original, visto que eles já foram lidos e transformados em escala de cinza, mas foi feito por cumprimento das especificações do TP.

3. Análise de complexidade

3.1. Tempo

Considerando as funções do programa, para que seja realizada a alocação das imagens, o algoritmo apresenta um custo de $O(n) + O(1)$ em que n é o número de linhas da matriz que irá ser alocada, sendo assim:

$$O(n) + 1 = O(n)$$

O mesmo também é válido para liberar memória.

Para ler e escrever uma imagem, as funções contém dois laços for aninhados, que juntos representam o custo o custo $O(n^2)$. Logo, mesmo com as declarações e atribuições que apresentam custo de $O(1)$, a soma das operações é resultada em:

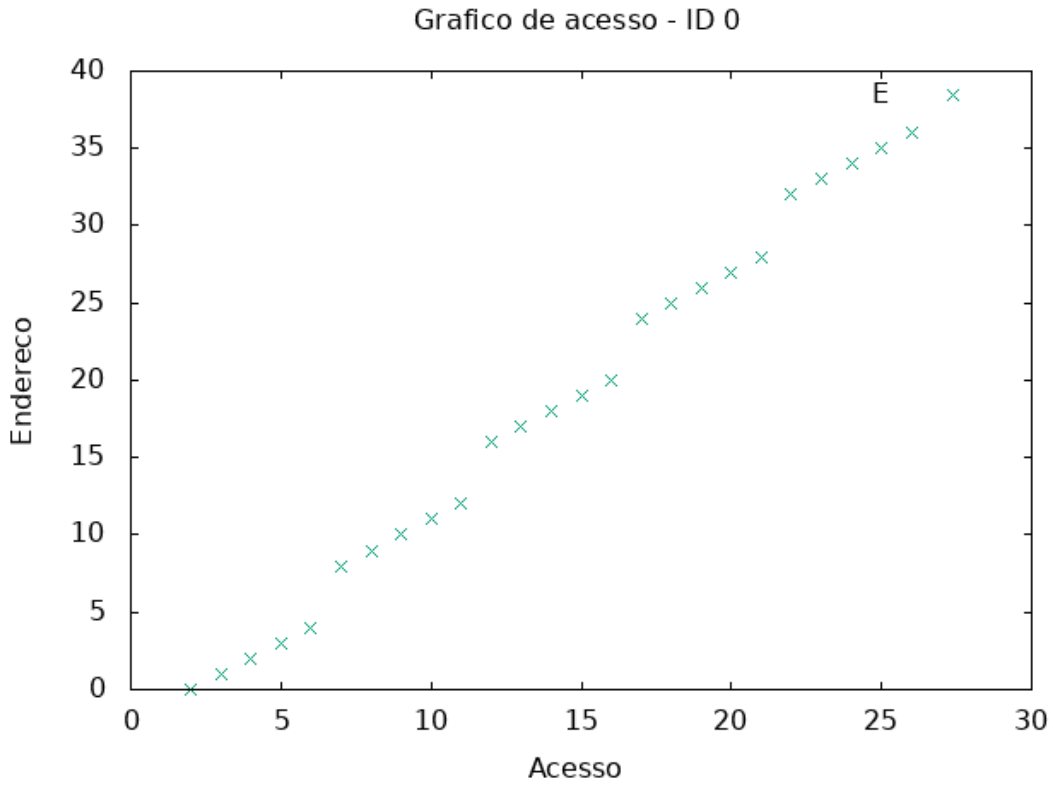
$$O(1) + O(n^2) = O(n^2)$$

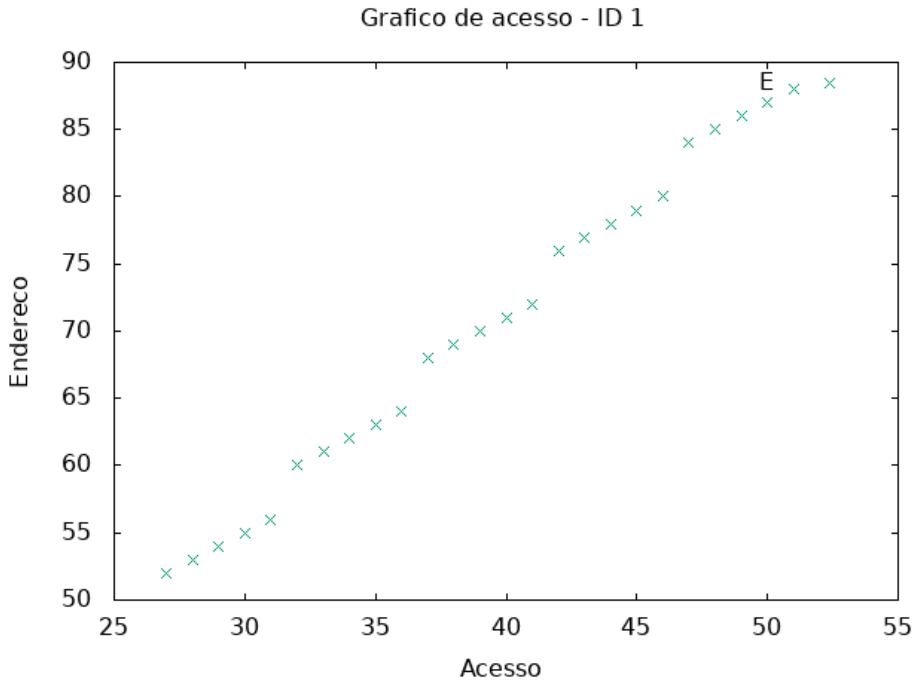
No processo de conversão é consumido um custo $O(1)$, visto que são realizadas operações simples e atribuição.

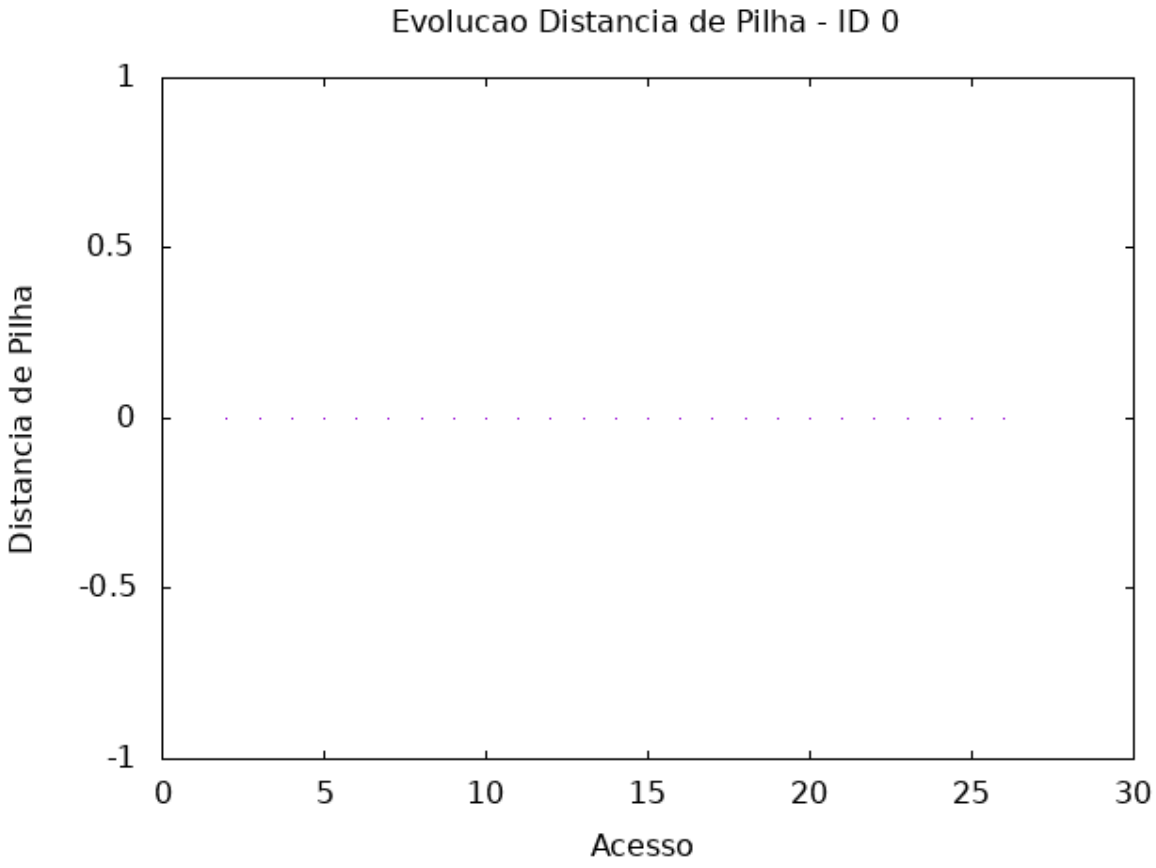
3.2. Espaço

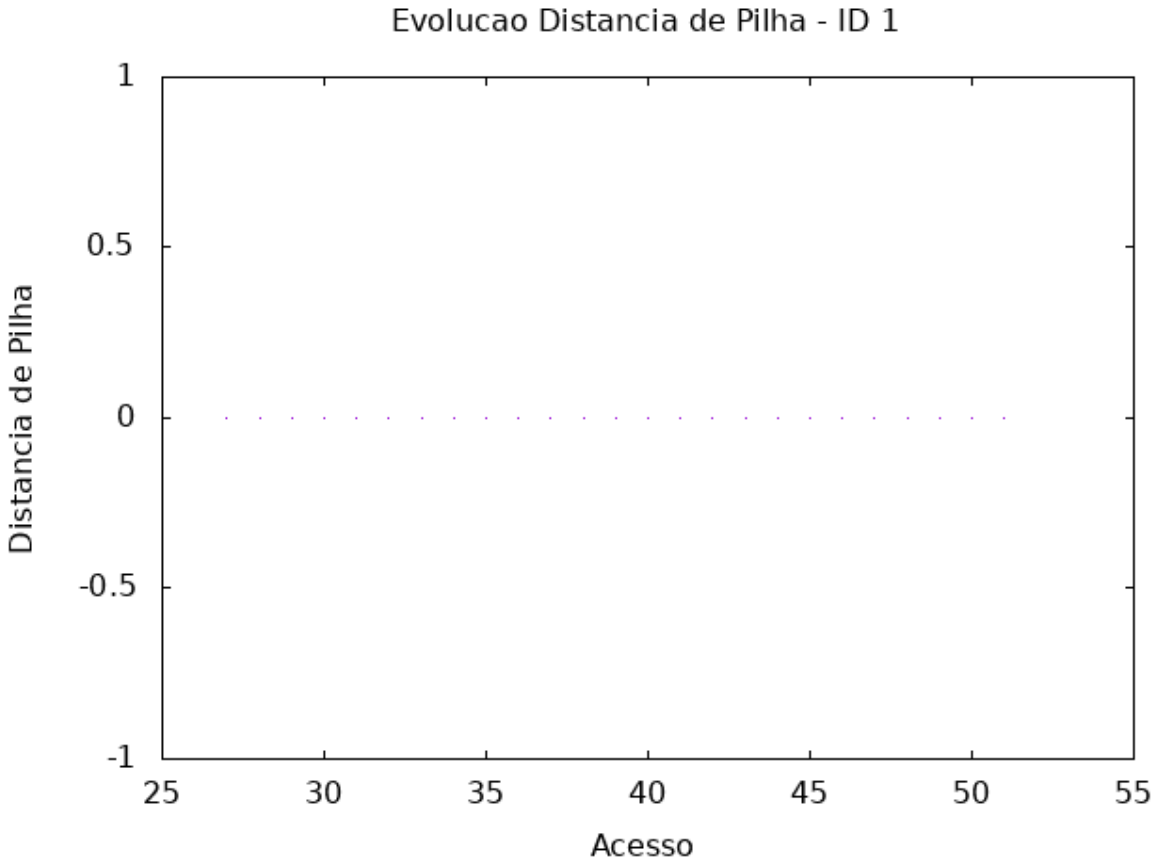
Considerando que cada uma das imagens são armazenadas em matrizes e ocupam um espaço mxn (em que m representa a quantidade de linhas e n a quantidade de colunas), é conclusivo que a execução do programa ocupe um espaço de $2*O(n^2)$, visto que são armazenadas as matrizes de entrada e de saída do programa.

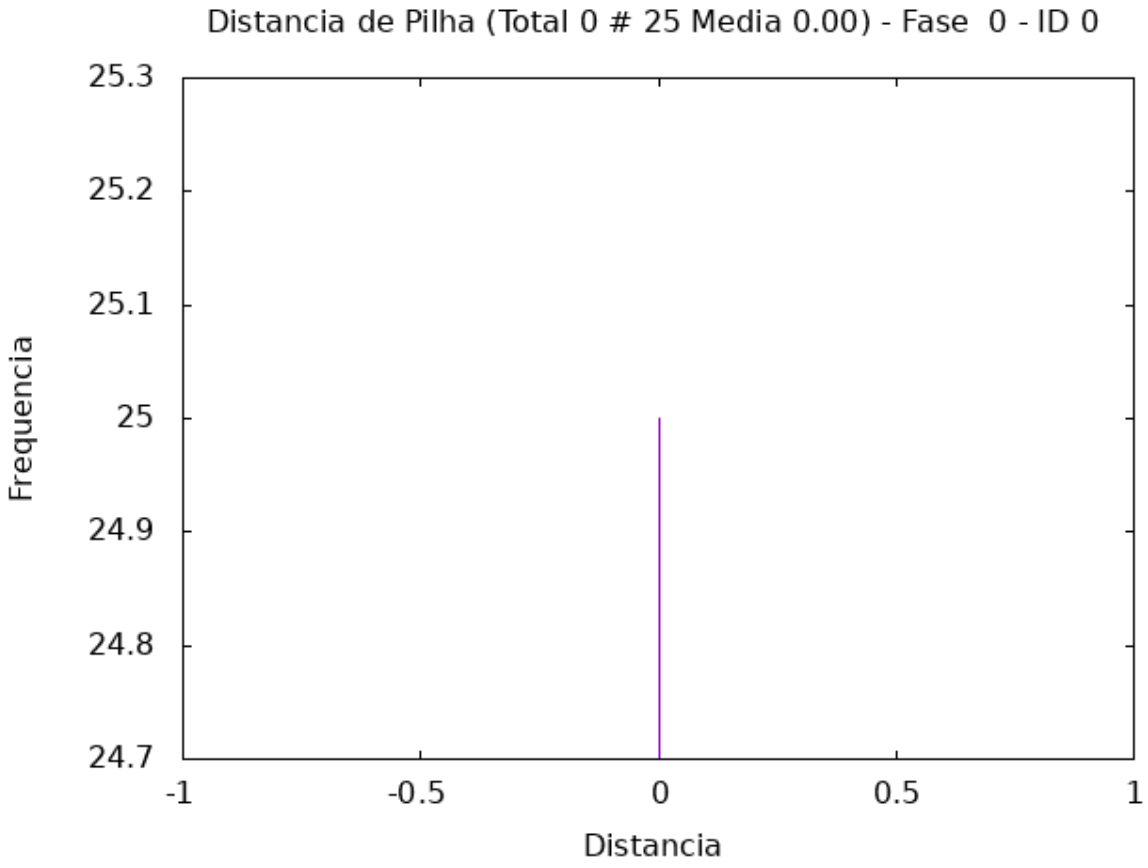
4. Análise Experimental

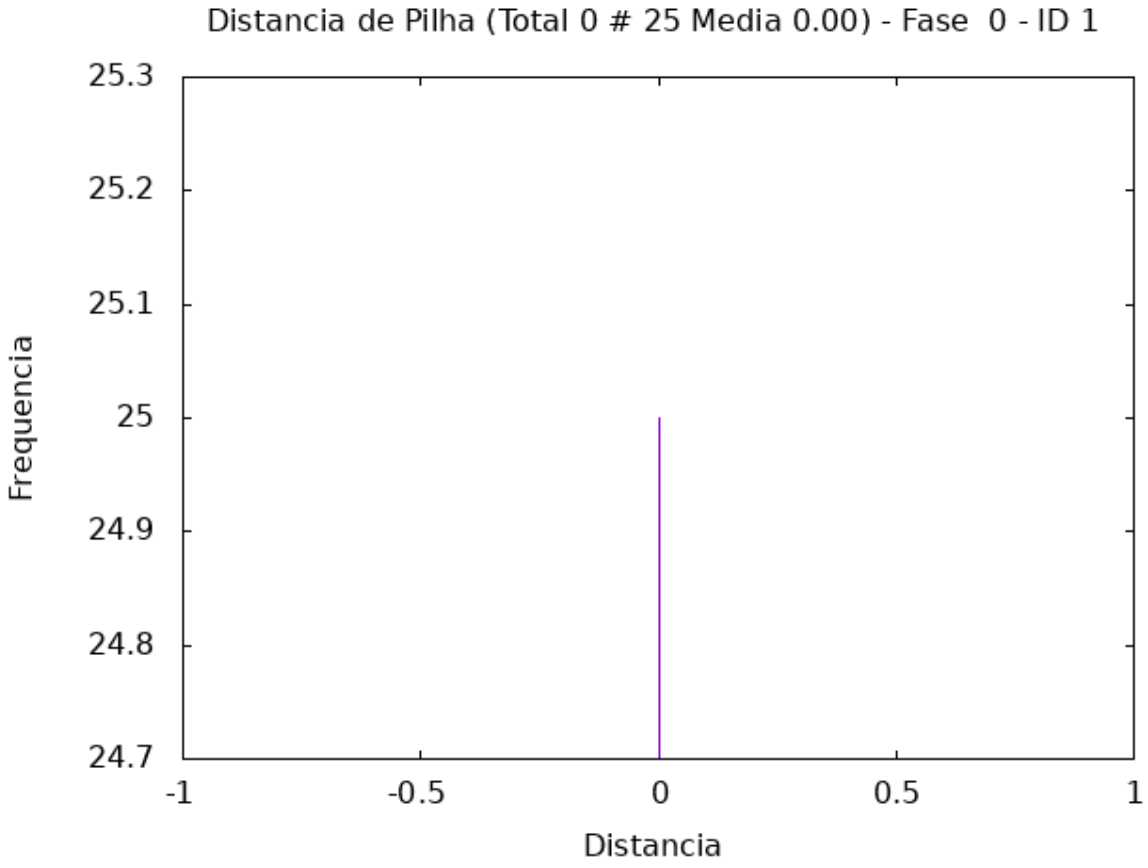


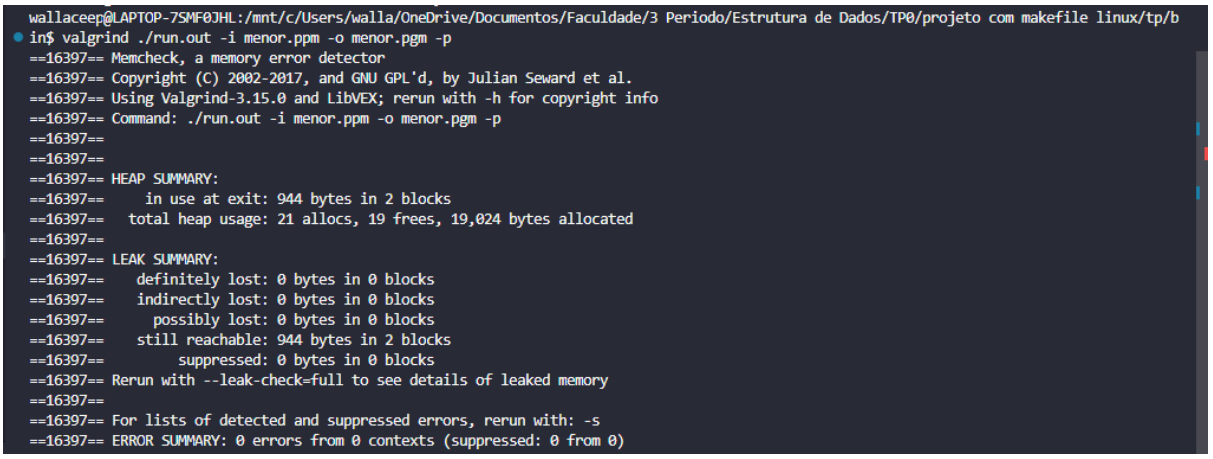



Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
1
0.5
0
-0.5
-1
0
5
10
15
20
25
30
Acesso


Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
1
0.5
0
-0.5
-1
25
30
35
40
45
50
55
Acesso


Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 0
Frequencia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1
Distancia


Distancia de Pilha (Total 0 # 25 Media 0.00) - Fase 0 - ID 1
Frequencia
25.3
25.2
25.1
25
24.9
24.8
24.7
-1
-0.5
0
0.5
1
Distancia



5. Conclusão

Após a implementação do programa, é notável que ele foi dividido em três partes distintas e importantes, sendo elas, a leitura e armazenamento da imagem original, a conversão e escrita da imagem em tons de cinza, e a análise de desempenho do projeto.

O principal desafio consistiu em analisar o desempenho do projeto, visto que tive dificuldades em implementar as ferramentas.

Referências

https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/

6. Instruções de compilação e execução

- Acesse o diretório em que se encontra o tp;
- Utilizando um terminal, compile os arquivos utilizando o seguinte comando: **make all;**
- Com esse comando, os arquivos serão compilados e um executável será gerado na pasta **bin;**
- Acesse o diretório em que se encontra a pasta bin e execute o arquivo **run.out** com o seguinte comando: **./run.out -i [imagem de entrada] -o [imagem de saída] -p;**
- Com esse comando, será executado o programa, e resultará em uma imagem de saída com o nome [imagem de saída] e um **log.out**, que contém informações sobre a leitura e escrita dos dados.