Documentação Trabalho Prático 0

Avelar Ribeiro Hostalácio

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

avelarrh@ufmg.br

1. Introdução

O problema a ser tratado consiste na implementação de um programa para conversão de uma imagem colorida para tons de cinza. A imagem de entrada é um arquivo no formato .ppm, no qual há uma representação de 24 bits para cada pixel (8 para cada componente R,G,B) e terá que ser convertida em .pgm e escrita em um arquivo de saída.

Esta documentação divide-se nos seguintes módulos:

2. Método: Descrição da implementação, detalhando as estruturas de dados, tipos abstratos de dados e métodos implementados.
3. Análise de Complexidade: Contém a análise da complexidade de tempo e espaço dos procedimentos implementados, formalizada pela notação assintótica.
4. Estratégias de Robustez: Contém a descrição, justificativa e implementação dos mecanismos de programação defensiva e tolerância a falhas implementados.
5. Análise Experimental: Apresenta os experimentos realizados em termos de desempenho computacional e localidade de referência, assim como as análises dos resultados.
6. Conclusões

2.Método

2.1 Configurações da máquina e ambiente

- Windows 10 Home x64

- WSL 2 com Ubuntu 20.04.

- Compilador g++ versão 6.3.0

- Programa desenvolvido em C++

2.2 Organização do código

O código segue os princípios da programação orientada a objetos, utilizando classes para implementação das estruturas de dados. Essas classes são controladas pelo arquivo *Main* do programa.

2.3 Estruturas de dados

O armazenamento das imagens foi feito com a utilização de matrizes, contudo, como o formato .pgm representa cada pixel com valores diferentes para cada componente RGB, foi criada uma classe para armazenar tais valores.

Para a realização da tarefa, foram criadas as classes *ColoredImg* e *GrayImg* que contém matrizes destinadas ao armazenamento de imagens coloridas e imagens em tons de cinza, respectivamente. Ademais, também foi criada a classe Pixel, que armazena os valores RGB de cada pixel presente na imagem colorida.

2.4 Funcionamento

O funcionamento do programa começa com a leitura da imagem no formato .ppm com a classe *ifstream* do C++, que permite a leitura de arquivos. A seguir é criado um objeto da classe *ColoredImg* que armazena o conteúdo do arquivo. Então é chamado o método *toGrayscale* que cria uma instância da classe *GrayImg* que contém valores correspondentes à conversão de cada pixel da imagem colorida para tons de cinza de acordo com a seguinte fórmula:

$$Y = \frac{49}{255}(0.30R + 0.59G + 0.11B)$$

Após a conversão, a escrita da imagem em um arquivo .pgm é feita com a utilização da classe *ofstream* do C++.

3.Análise de complexidade

3.1 Leitura de imagem colorida

O método *readImage* realiza operações em tempo constante O(1). Para preencher a matriz de altura n e largura m, são utilizados loops aninhados que percorrem cada elemento da matriz. Logo, a complexidade de tempo da função é O(mn).

Como não é alocado espaço extra, a complexidade de espaço é O(1).

3.2 Conversão de imagem colorida para tons de cinza

O método *toGrayscale* realiza suas operações em tempo constante O(1). O método utiliza laços aninhados para percorrer cada elemento matriz de pixels e realizar a conversão para cinza. Como a matriz possui tamanho m x n (largura x altura), a complexidade de tempo é O(mn).

Como não é alocado espaço extra, a complexidade de espaço é O(1).

3.3 Escrita de imagem em tons de cinza

O método *writeImage* realiza operações em tempo constante O(1). Para ler a matriz de altura m e largura n, são utilizados loops aninhados que percorrem cada elemento da matriz. Logo, a complexidade de tempo da função é O(mn).

Como não é alocado espaço extra, a complexidade de espaço é O(1).

3.4 Construtores

Para a construção das matrizes m x n (largura x altura) que armazenam o conteúdo dos arquivos .ppm e .pgm é utilizado um laço que aloca memória correspondente a uma linha da matriz. Como essas alocações são realizadas n vezes, a complexidade de tempo é O(n).

As matrizes armazenam imagens de m pixels de largura e n pixels de altura, logo a complexidade de espaço dos construtores é O(mn).

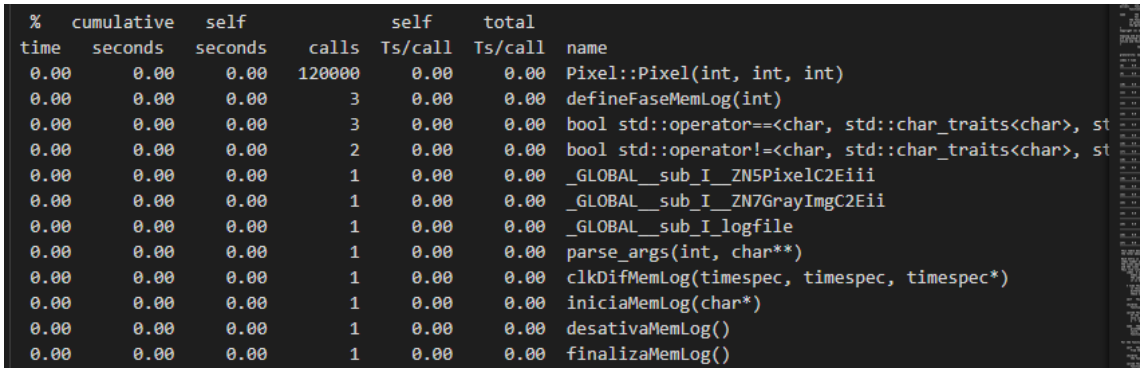
4.Estratégias de robustez

Para garantir que o código tenha o funcionamento esperado, a biblioteca *msgassert* para a implementação de mecanismos de programação defensiva e tolerância a falhas. A biblioteca define macros de asserções para o controle de operações inválidas.

5.Análise experimental

A análise de desempenho computacional foi feita com as ferramentas *gprof* e *memlog*. O *gprof* foi útil para informar quais métodos impactaram mais o desempenho e como o tamanho da imagem influencia no tempo de execução. O *memlog* foi utilizado para avaliar o tempo de execução total do programa.

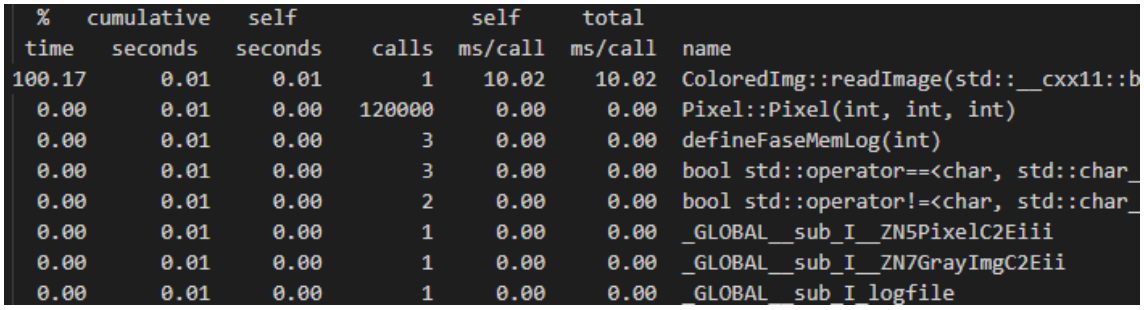
Para a imagem 5x5 o tempo de execução é praticamente instantâneo. De acordo com o *gprof*, o construtor da classe Pixel foi a função mais chamada. Já o *memlog* avaliou que o tempo decorrido foi de 0.024 segundos.



% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	120000	0.00	0.00	Pixel::Pixel(int, int, int)
0.00	0.00	0.00	3	0.00	0.00	defineFaseMemLog(int)
0.00	0.00	0.00	3	0.00	0.00	bool std::operator==<char, std::char_traits<char>, st
0.00	0.00	0.00	2	0.00	0.00	bool std::operator!=<char, std::char_traits<char>, st
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5PixelC2Eiii
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN7GrayImgC2Eii
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_logfile
0.00	0.00	0.00	1	0.00	0.00	parse_args(int, char**)
0.00	0.00	0.00	1	0.00	0.00	clkDifMemLog(timespec, timespec, timespec*)
0.00	0.00	0.00	1	0.00	0.00	iniciaMemLog(char*)
0.00	0.00	0.00	1	0.00	0.00	desativaMemLog()
0.00	0.00	0.00	1	0.00	0.00	finalizaMemLog()

```
I 1 3218.266130700
F 2 3218.290381200 0.024250500
```

Para a imagem 400 x 300, o *gprof* não nota diferença significativa no tempo de execução de cada método, e o construtor de Pixel continua tendo o maior número de chamadas. O *memlog* registrou um tempo de execução de 0.1 segundos, aproximadamente 4 vezes maior do que o anterior.



% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.17	0.01	0.01	1	10.02	10.02	ColoredImg::readImage(std::__cxx11::b
0.00	0.01	0.00	120000	0.00	0.00	Pixel::Pixel(int, int, int)
0.00	0.01	0.00	3	0.00	0.00	defineFaseMemLog(int)
0.00	0.01	0.00	3	0.00	0.00	bool std::operator==<char, std::char_
0.00	0.01	0.00	2	0.00	0.00	bool std::operator!=<char, std::char_
0.00	0.01	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5PixelC2Eiii
0.00	0.01	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN7GrayImgC2Eii
0.00	0.01	0.00	1	0.00	0.00	_GLOBAL__sub_I_logfile

```
I 1 3490.537267500
F 2 3490.646839800 0.109572300
```

Para a imagem 1600 x 1200, nota-se que o maior impacto no tempo de execução são os métodos de ler imagem, converter imagem e escrever imagem. O *memlog* registra um tempo de execução de 0.8 segundos, aproximadamente 8 vezes maior que a imagem 400x300 e 32 vezes maior que a 5x5.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.39	0.02	0.02	1920000	0.00	0.00	Pixel::Pixel(int, int,
33.39	0.04	0.02	1	20.03	40.07	ColoredImg::readImage(
16.70	0.05	0.01	1	10.02	10.02	ColoredImg::toGrayscal
16.70	0.06	0.01	1	10.02	10.02	GrayImg::writeImage(st
0.00	0.06	0.00	3	0.00	0.00	defineFaseMemLog(int)
0.00	0.06	0.00	3	0.00	0.00	bool std::operator==<c
0.00	0.06	0.00	2	0.00	0.00	bool std::operator!=<c

```
I 1 3922.636371200
F 2 3923.458167800 0.821796600
```

Portanto, conclui-se que o desempenho computacional é fortemente impactado pelo aumento das dimensões da matriz de entrada.

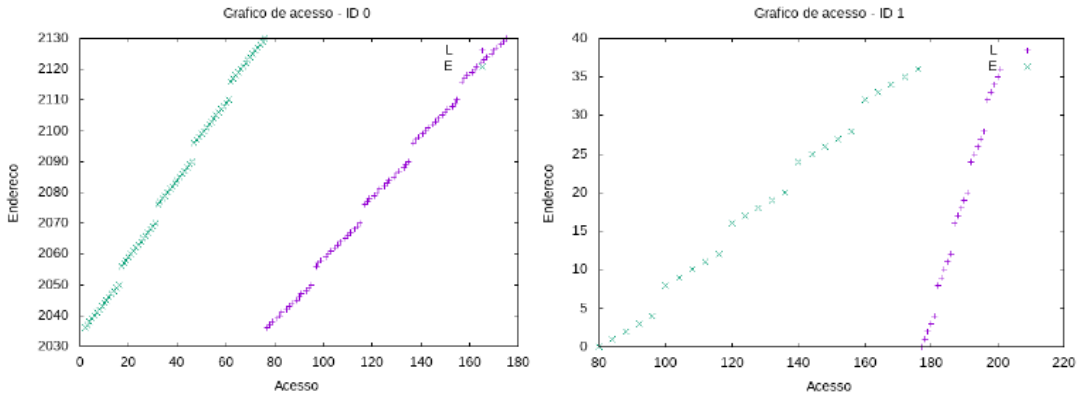
5.Localidade de referência

A análise da localidade de referência foi feita com a utilização da aplicação *analisamem* e gráficos plotados com o *gnuplot*. Essa análise foi feita utilizando uma imagem 5 x 5 para teste, pois dessa forma é mais fácil visualizar cada endereço acessado.

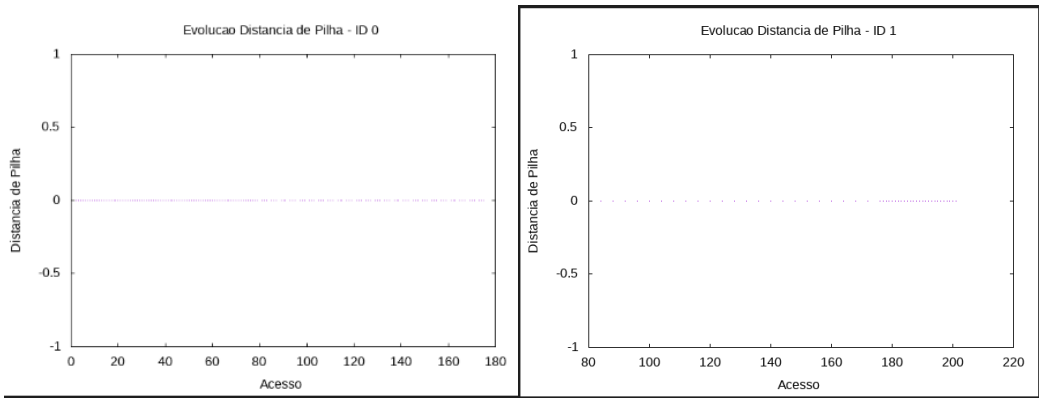
A matriz da imagem colorida possui ID 0 e a imagem cinza possui ID 1.

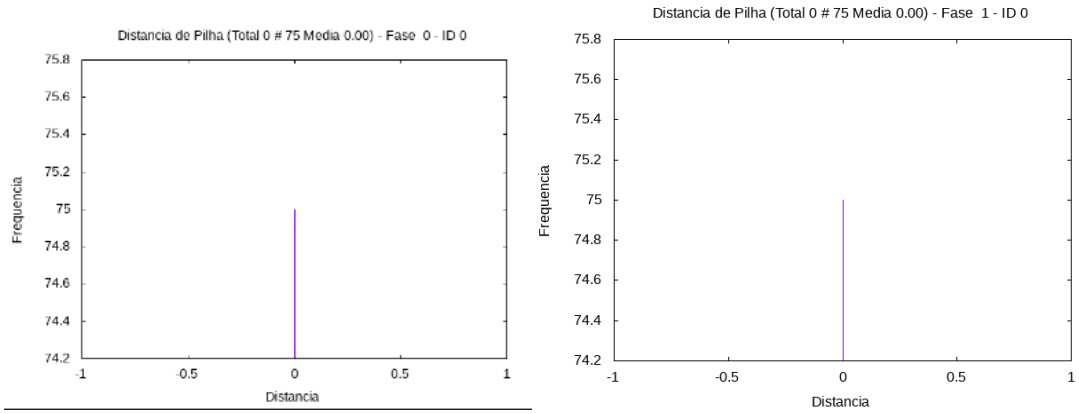
Durante a execução do código os dados da imagem são escritos na matriz colorida e em seguida lidos e, após conversão, escritos na matriz cinza, a qual é lida para a escrita no arquivo pgm.

Os saltos existentes no gráfico são provenientes da utilizacão de matrizes para armazenamento de dados, já que elas são acessaadas linha a linha.



As distâncias de pilha são sempre 0 pois em cada fase, cada matriz é acessada sequencialmente uma única vez. Logo, nenhum endereço de memória é acessado múltiplas vezen em uma mesma fase, levando à regularidade dos gráficos.





6.Conclusões

Após desenvolver, analisar a complexidade e tempo de desempenho, assim como observar os padrões de acesso à memória, conclui-se que as dimensões possuem grande influência no custo computacional da conversão de imagens. Ademais, a realização do trabalho possibilitou que fossem colocados em prática os conhecimentos relativos a análise de complexidade e localidade de referência recém adquiridos na disciplina de Estrutura de Dados, além de reforçar conceitos de programação defensiva e boas práticas de programação, que são necessários para o desenvolvimento de software eficiente e fácil de manter.

Bibliografia

NETPBM. Wikipedia, The Free Encyclopedia, 26 jun. 2022. Disponível em: https://en.wikipedia.org/wiki/Netpbm. Acesso em: 08 set. 2022.

Instruções para compilação e execução

A compilação é feita por meio do Make.

Ao executar o comando make all, todos os arquivos são compilados e um executável é gerado na pasta bin.

Para a execução do programa, existem as seguintes opções de linha de comando:

-i “nome do arquivo de entrada” : entrada em formato .ppm

-o “nome do arquivo de saída” : saída em formato .pgm (caso não seja fornecida saída, será criado um arquivo “output.pgm” na raiz do projeto)

-p “arquivo de registro de desempenho” : nome do arquivo que será gerado com o desempenho do programa

-l : opcional, indica se devem ser registrados os acessos à memória

Exemplo de mensagem de uso:

```
./bin/run.out -i ./Sample/bolao.ppm -o output.pgm -p ./log.out -l
```

Após a execução, os comandos make gprof e make plot podem ser utilizados, respectivamente, para fazer a análise de tempo de execução com o gprof e plotar os gráficos contendo os acessos a memória.