Arquitectura de Sistemas e Computadores I
Licenciatura em Engenharia Informática

Miguel Barão

Ano Lectivo 2018-2019

1 Objectivo

Pretende-se com este trabalho desenvolver um conjunto de funções em assembly MIPS para remover ruído em imagens em tons de cinzento. Dado um ficheiro com uma imagem, o programa deverá abrir a imagem, correr um algoritmo de remoção de ruído e guardar o resultado noutro ficheiro de imagem com o mesmo formato e dimensão.

2 Teoria

2.1 Cores RGB

Embora fisicamente exista uma infinidade de cores diferentes, a visão humana é genericamente sensível a apenas três cores: vermelho, verde e azul.

Qualquer cor "visível" é uma combinação das três cores primárias. Assim, quando o objectivo é gerar uma imagem para ser observada pelo olho humano, basta reproduzir estas três cores com intensidades apropriadas. Chama-se a esta representação RGB (Red, Green, Blue).

Num computador digital, cada uma das três intensidades RGB é representada por um número inteiro de N bits a que se chama *profundidade de cor* (*color depth* ou *bit depth*).

Frequentemente são usados 8 bits (1 byte) para cada componente, com níveis de intensidade que vão de 0 a 255. Deste modo temos 24 bits (3×8) para especificar uma cor RGB. Como existem $2^{24} = 16777216$ combinações diferentes, temos mais de 16 milhões de cores diferentes.

2.2 Tons de cinzento

Quando as três componentes RGB são $(0, 0, 0)$, correspondente à ausência de luz nas 3 cores primárias, dizemos que a cor é preta. Quando as três componentes têm a intensidade máxima $(255, 255, 255)$ obtemos a cor branca. Nas situações intermédias (x, x, x), com $0 < x < 255$, dizemos que a cor é cinzenta, aproximando-se do preto ou branco nas extremidades do intervalo.

Uma imagem em tons de cinzento é uma imagem em que as três componentes RGB têm a mesma intensidade. Neste caso, basta conhecer um único número, a intensidade x, para que o tom de cinzento fique definido.

2.3 Conversão RGB para tons de cinzento

A conversão de uma cor RGB para tons de cinzento consiste em converter as três componentes de cor para apenas uma componente que codifica a intensidade luminosa total, tal como é percepcionada pelo olho humano.

Sabe-se que a retina existente no olho humano tem sensibilidades diferentes para cada uma das componentes RGB, sendo mais sensível ao verde e menos ao azul. Portanto, se pretendermos que as imagens em tons de cinzento reflictam estas diferenças de sensibilidade, então as componentes de cor devem contribuir com pesos diferentes na intensidade luminosa de um pixel.

A fórmula seguinte é um *standard* neste tipo de conversão:

$$I = 0.299R + 0.587G + 0.114B.$$

2.4 Representação de imagens monocromáticas

Uma imagem em tons de cinzento consiste numa matriz de pixeis onde cada elemento da matriz contém a intensidade do respectivo pixel:

$$\begin{bmatrix} I_{11} & I_{12} & \cdots & I_{1n} \\ I_{21} & I_{22} & \cdots & I_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ I_{m1} & I_{m2} & \cdots & I_{mn} \end{bmatrix}.$$

Para guardar a imagem em memória, ou num ficheiro, é necessário guardar os elementos "linearmente" em posições consecutivas. Uma forma de fazer isso é organizá-los sequencialmente em linhas, isto é:

$$[\underbrace{I_{11} \cdots I_{1n}}_{1^{\text{a}}\text{ linha}} \underbrace{I_{21} \cdots I_{2n}}_{2^{\text{a}}\text{ linha}} \cdots \underbrace{I_{m1} \cdots I_{mn}}_{m\text{-ésima linha}}].$$

Esta forma de organização chama-se *row major*. Outra organização possível é a *column major* onde a arrumação dos elementos segue ao longo de colunas[1].

2.5 Representação de imagens RGB

Uma imagem a cores RGB consiste numa matriz de pixels, onde a cor de cada pixel é representado pela tupla (R, G, B).

Uma imagem de dimensão $m \times n$ corresponde a uma matriz

$$\begin{bmatrix} (R_{11}G_{11}B_{11}) & (R_{12}G_{12}B_{12}) & \cdots & (R_{1n}G_{1n}B_{1n}) \\ (R_{21}G_{21}B_{21}) & (R_{22}G_{22}B_{22}) & \cdots & (R_{2n}G_{2n}B_{2n}) \\ \vdots & \vdots & & \vdots \\ (R_{m1}G_{m1}B_{m1}) & (R_{m2}G_{m2}B_{m2}) & \cdots & (R_{mn}G_{mn}B_{mn}) \end{bmatrix}.$$

Para guardar a imagem em memória, ou num ficheiro, é necessário guardar os pixeis em posições consecutivas. Uma forma de fazer isso é usar as organizações *row major* ou *column major*:

$$[\underbrace{R_{11}G_{11}B_{11} \cdots R_{1n}G_{1n}B_{1n}}_{1^{\text{a}}\text{ linha}} \underbrace{R_{21}G_{21}B_{21} \cdots R_{2n}G_{2n}B_{2n}}_{2^{\text{a}}\text{ linha}} \cdots \underbrace{R_{m1}G_{m1}B_{m1} \cdots R_{mn}G_{mn}B_{mn}}_{m\text{-ésima linha}}].$$

Alternativamente, a imagem RGB pode ser dividida em três imagens monocromáticas: Uma imagem só com a componente vermelha, outra verde e outra azul. Neste caso, a imagem RGB é representada por três matrizes, R, G, B, uma para cada componente de cor.

[1] Nas linguagens de programação que suportam arrays multidimensionais é importante perceber qual a organização usada. Na linguagem C/C++ os arrays estão organizados em row major. Algumas linguagens comuns em computação científica usam column major, como é o caso de Fortran, MATLAB, Julia, etc.

2.6 Formatos de imagem

Quando uma imagem é guardada em memória, basta-nos guardar os pixeis consecutivamente num array. Além dos pixeis temos também de manter informação sobre a dimensão da imagem (horizontal e vertical). Toda a informação relevante pode estar dispersa em memória e registos consoante se trata de variáveis locais, globais ou dados dinâmicos.

Quando se guarda num ficheiro, todos os dados têm obrigatoriamente de ser guardados consecutivamente de modo a que seja fácil reconstruir toda a imagem quando o ficheiro for lido. Por exemplo, não se deve começar directamente com os pixeis da imagem, guardando as dimensões e profundidade de cor só no final do ficheiro, pois isso tornaria a leitura da imagem muito mais difícil.

A forma como a informação é codificada num ficheiro designa-se genericamente por formato do ficheiro. Existem inúmeros formatos para ficheiros de imagem: JPEG, PNG, BMP, GIF, TIFF, PGM, PBM, etc. Neste trabalho vamos usar os formatos RGB e GRAY que consistem simplesmente em escrever as linhas da imagem consecutivamente. Os ficheiros RGB e GRAY não contêm informação acerca do tamanho da imagem nem da profundidade de cor, essa informação é perdida na conversão. Para descodificar uma imagem neste formato é necessário conhecer *a priori* as características da imagem original.

Para converter uma imagem para este formato, pode ser usado o programa `convert` disponibilizado pelo pacote de software `ImageMagick` em Linux[2]. Por exemplo, a conversão entre os formatos JPEG, RGB e GRAY pode ser realizada com os comandos:

```
convert imagem.jpg imagem.rgb                              # JPG  -> RGB
convert -size 512x512 -depth 8 imagem.rgb imagem.jpg       # RGB  -> JPG
convert -size 512x512 -depth 8 imagem.gray imagem.jpg      # GRAY -> JPG
```

Nos dois últimos casos, correspondentes à conversão de RGB e de GRAY para JPEG, é necessário passar como opções a dimensão `-size 512x512` e a profundidade de cor `-depth 8` uma vez que o ficheiro RGB não contém essa informação.

2.7 Processamento de imagem

O processamento de imagem consiste essencialmente em transformar uma imagem A numa imagem B processada. Neste caso pretende-se transformar imagens com ruído em imagens com o mínimo de ruído possível.

Consideremos uma imagem ruidosa A, em tons de cinzento. O ruído na imagem consiste essencialmente em pixeis que foram corrompidos e que apresentam intensidades adulteradas. Uma técnica de rudimentar de remoção de ruído, chamada *filtro de média*, consiste em substituir cada pixel pela média dos pixeis numa vizinhança à sua volta. Como pixeis vizinhos têm tipicamente intensidades parecidas, a média tem o efeito de substituir os pixel de ruído por valores semelhantes aos vizinhos.

Matematicamente, o filtro de média consiste em definir uma *máscara* (ou *kernel*), que não é mais do que a matriz 3×3

$$M = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix},$$

e aplicar esta máscara do seguinte modo: Para calcular o pixel $B(i, j)$ na posição (i, j) da imagem filtrada B, sobrepõe-se a matriz M à imagem original A centrada na posição (i, j) e multiplicam-se os pesos da matriz pelos pixeis que ficam "por baixo". Somando os 9 resultados obtém-se a média, que é então guardada na imagem filtrada. O processo repete-se para cada pixel da imagem.

Esta operação chama-se *convolução 2D* e representa-se por um asterisco (não é um produto matricial):

$$B = M * A.$$

[2]Instalação em ubuntu/mint/debian: sudo apt install imagemagick

Figura 1: Imagem “Lena” corrompida por ruído aleatório. Tamanho 512×512.

(a) Resultado obtido com o filtro de média. (b) Resultado obtido com o filtro de mediana.

Figura 2: Images filtradas.

Cada elemento B_{ij} da matriz B é obtido pelo *somatório de convolução*

$$B(i,j) = \sum_{p\in\{-1,0,1\}} \sum_{q\in\{-1,0,1\}} A(i+p, j+q)\ M(2-p, 2-q).$$

As imagens 1 e 2(a) mostram o antes e depois da aplicação do filtro de média. Esta técnica esbate um pouco o ruído, mas tem a desvantagem de esbater também a própria imagem (tente perceber porquê?).

Outra técnica de remoção de ruído consiste em usar um *filtro de mediana*. O filtro de mediana é semelhante ao filtro de média, mas em vez de substituir cada pixel pela média da vizinhança, substitui pela mediana.

Dados 9 pixeis, a mediana é um valor x tal que existem tantos valores maiores como menores do que x. Um método para obter a mediana consiste em ordenar os valores e escolher o que fica a meio. Por exemplo, se numa vizinhança de um certo pixel os valores forem

$$\begin{bmatrix} 97 & 101 & 99 \\ 96 & 149 & 102 \\ 89 & 93 & 94 \end{bmatrix},$$

então ordenando os valores obtém-se $(89, 93, 94, 96, 97, 99, 101, 102, 149)$, e portanto a mediana é 97.

A imagem 2(b) mostra o resultado da aplicação do filtro de mediana à imagem 1. Como se pode observar, a imagem produzida não apresenta o esbatimanento do filtro de média e o resultato é consideravelmente melhor.

3 Implementação

A implementação dos filtros para remoção de ruído deve ser estruturada em várias funções independentes. Na estruturação do seu trabalho deverá ter as seguintes funções:

read_gray_image função que recebe um nome de um ficheiro, abre o ficheiro e lê a imagem para memória (dados dinâmicos). Retorna o endereço de memória onde está a imagem.

write_gray_image função que escreve uma imagem em formato GRAY num ficheiro. A função recebe como argumentos o nome de um ficheiro, um buffer com a imagem e o comprimento do buffer (por esta ordem).

mean_filter função que calcula a convolução de uma imagem A com uma máscara (matriz 3×3) e coloca o resultado numa matriz B. A função recebe como argumentos um buffer com a matriz A um buffer que vai conter a imagem filtrada B, e as dimensões da imagem.

median_filter função que aplica o filtro de mediana a uma imagem. Os argumentos são o buffer de entrada A e o buffer de saída B, e as dimensões da imagem.

3.1 I/O para Ficheiros

A leitura e escrita de ficheiros é efectuada da mesma maneira que em sistemas UNIX/LINUX com as funções `open`, `read`, `write` e `close`.

Para abrir um ficheiro para leitura pode usar-se o código seguinte:

```
la $a0, FILENAME     # string containing filename to open
li $a1, 0
li $a2, 0            # read only
li $v0, 13           # open
syscall              # v0 = file descriptor
```

No final, o registo `$v0` contém o descritor do ficheiro que será usado para as operações de leitura seguintes. A leitura do ficheiro é efectuada com um `read` da seguinte maneira:

```
# a0 = file descriptor
# a1 = buffer address
# a2 = length (number of bytes to read)
li $v0, 14          # read
syscall
```

Para abrir um ficheiro para escrita pode usar-se o código seguinte:

```
la $a0, FILENAME     # string containing filename to open
li $a1, 1            # 0 = read, 1 = write/create, 9 = write/append
li $a2, 0            # -- ignorado --
li $v0, 13           # open
syscall              # v0 = file descriptor (-1 on failure)
```

A escrita neste ficheiro é efectuada com um `write` da seguinte maneira:

```
# a0 = file descriptor
# a1 = buffer address
# a2 = length (number of bytes to write)
li $v0, 15          # write
syscall
```

No final todos os ficheiros abertos devem ser fechados com a função `close`:

```
# a0 = file descriptor
li $v0, 16          # close
syscall
```

Ver documentação do Mars no menu *"help"*.

4 Notas finais

- O trabalho deverá ser realizado em grupos de 2 alunos.
- Começe com imagens de muito baixa dimensão para testar o código. Por exemplo uma imagem com 5×5 pixeis. Quando tudo funcionar correctamente numa imgem pequena, experimente a imagem lena com dimensões progressivamente maiores.
- Poderá ser útil usar as instruções `madd`, `maddu`, e `lbu`.
- O código deve estar comentado do seguinte modo:

```
#########################################################
# xpto - Esta funcao calcula a area de um retangulo
#
# Argumentos:
#  a0 - largura
#  a1 - comprimento
#
# Retorna:
#  v0 - area do retangulo
#########################################################
xpto:
    mult $a0, $a1   # calcula area
    mflo $v0
    jr $ra
    nop
```

- Além do código devidamente comentado e bem organizado, será necessário entregar um relatório em PDF onde explica o problema e como organizou a sua solução, mas sem incluir código. (Sugestão: use LaTeXpara a escrita do relatório)
- Submeter o trabalho no moodle num único ficheiro comprimido (rar, zip, tar.gz, etc) com nome formado pelos números dos alunos, por exemplo `11111-22222.rar`.
- O trabalho é longo e deverá demorar várias semanas a realizar. Guarde a última semana exclusivamente para a escrita do relatório.

Bom trabalho!
Miguel Barão