Departamento de Ciência da Computação – Universidade de Brasília (UnB)
Brasília – DF – Brasil
Organização e Arquitetura de Computadores - Laboratório 1

David Potolski Lafetá
Matricula 15/0122969
davidpotolskilafeta@gmail.com

Guilherme Fleury Franco
Matricula 18/0121472
guilherme.fleury26@gmail.com

Manoel Vieira C Neto
Matrícula 180137816
vieiranetoc@gmail.com

Abstract

This paper aims to demonstrate the group's understanding of the Assembly MIPS language for the R2000 / R3000 family processors. For this purpose,we implemented a computational vision algorithm that has functions that apply predetermined effects by specifications of the project to a loaded image into the program.

1. Objetivos

Utilizando o ambiente MARS 4.5, o grupo desenvolveu uma aplicação que lê e escreve imagens codificadas em BMP e é capaz de aplicar os seguintes efeitos:borramento, extração de bordas e binarização por limiar. Para operar esse programa, o usuário fará uso de um menu, que permite selecionar qual a operação desejada, seus parâmetros e, ao final do programa, salvar a imagem com os efeitos aplicados.

2. Introdução

Uma imagem BITMAP (.bmp)[2], é um arquivo de formato binário, onde 54*bytes* iniciais consitem o cabeçalho com suas propriedades a serem lidas por programas de visualização de imagens ou para operações que necessitem alterar as propriedades da imagem como tamanho do pixel, tabela de cor (caso de pixel de 24*bits*), recorte, entre outras. No nosso caso faremos operações morfológicas na imagem, portanto nenhuma alteração é necessária no cabeçalho da imagem. O programa desenvolvido opera com imagens de tamanho 512x512, o formato .bmp guarda as componentes de cor RGB em 3*bytes*, portanto o tamanho ocupado pelo vetor de cores (cada 3 correspondendo a um pixel) é dado por: $512x512x3 = 786432bytes$. Somando este valor ao cabeçalho temos o tamanho total do arquivo (em bytes): $786432 + 54 = 786486bytes$. A literatura acerca do assunto é bem extensa e de fácil acesso, para o desenvolvimento do software foram consultadas as teorias descritas na documentação da biblioteca OpenCV, do livro *Applied pattern recognition: algorithms and implementation in C++* [3], bem como de sítios auxiliares.

3. Materiais

Para escrever e compilar a aplicação foi utilizado o programa MARS 4.5 que simula o *MIPS*. Para facilitar a integração dos fragmentos de código escritos por cada integrante foi utilizada a ferramenta Git Kraken, com o GitHub como repositório remoto.

4. Métodos

No trabalho foram utilizando basicamente 4 métodos para realizar as tarefas necessárias. Um método chamado Operações com arquivos (nele foram englobadas as ações de ler a imagem e escrever a imagem final em um novo arquivo; As operações intermediárias a essas ações serão detalhadas na subseção Operações com arquivo), um método chamado Binarização por Limiar, um chamado Efeito de Borramento e um chamado Efeito de Extração de Bordas. Todos esses métodos serão descritos em suas respectivas subseções.

4.1. Binarização por Limiar

Thresholding é o método mais simples de segmentação de imagem. A partir de uma imagem em tons de cinza o *Thresholding* pode ser usado parar criar imagens binárias.

Os métodos de limiar mais simples substituem cada pixel de uma imagem por um pixel preto se a intensidade do pixel for menor que alguma constante fixa T (essa constante é passada pelo usuário durante a execução do programa), ou um pixel branco se a intensidade da imagem for maior que essa constante.

Esse método trabalha com duas modificações na imagem para atingir o resultado desejado. Inicialmente percorremos a imagem *pixel* a *pixel* transformando a cor de RGB para

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

Figura 1: Equação matemática da Binarização [1]

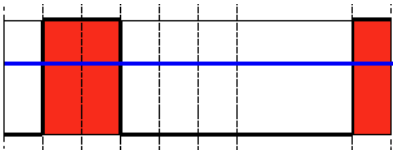

Figura 2: Gráfico de funcionamento do Threshold [1]

tons de cinza. Atingimos isso a partir da seguinte equação:

$$\textbf{Grayscale} = \frac{(R+G+B)}{3}$$

Fazemos uma média entre os valores de cor nos *bytes* predeterminados como RGB e salvamos essa média em todos os *bytes* do *pixel*, deixando assim ele em uma escala de cinza que será necessária para atingirmos o efeito final de binarização por limiar.

Figura 3: Comparação entre a imagem original e a imagem em *grayscale* (lembrando que isso é apenas uma representação, durante a execução do código a imagem não ficará assim em nenhum momento).

O próximo passo é comparar o tom de cinza do *pixel* com o *threshold* e modificar a cor do *pixel* para branco caso ele possua um tom abaixo do tom de cor do *threshold* e preto caso possua um tom acima. O valor do *threshold* é 0x007f7f7f, escolhido por ser o tom médio entre as cores preto e branco, sendo possível ser alterado a partir do menu. Para isso selecione um número entre 1 e 10 (sendo 1 o *threshold* mais próximo da cor branca,5 o *threshold* médio, sendo mantido o valor original 0x007f7f7f para comparação e 10 o *threshold* mais próximo do preto).

Figura 4: Comparação entre a imagem original e as imagens após a execução do programa com os filtros 1,5 e 10 respectivamente.

4.2. Efeito de Borramento

No processamento de imagens, um desfoque gaussiano (também conhecido como suavização gaussiana) é o resultado do desfoque de uma imagem por uma função gaussiana. É um efeito amplamente utilizado no *software* gráfico, normalmente para reduzir o ruído da imagem e reduzir os detalhes. O efeito visual dessa técnica de desfoque é um desfoque suave que lembra a visualização da imagem através de uma tela translúcida. O alisamento gaussiano também é usado como um estágio de pré-processamento em algoritmos de visão computacional para melhorar as estruturas de imagem em diferentes escalas.

Matematicamente, aplicar um desfoque gaussiano a uma imagem é o mesmo que convolver a imagem com uma função gaussiana. Isso também é conhecido como uma transformação *Weierstrass* bidimensional. Como a transformada de *Fourier* de um Gauss é outro Gaussiano, a aplicação de um desfoque gaussiano tem o efeito de reduzir os componentes de alta frequência da imagem; um borrão gaussiano é, portanto, um filtro de baixa passagem.

Um efeito de desfoque Gaussiano é normalmente gerado pela convolução de uma imagem com um *kernel* de valores Gaussianos. Na prática, é melhor aproveitar a propriedade separável do borrão de Gauss, dividindo o processo em duas passagens. Na primeira passagem, um *kernel* unidimensional é usado para desfocar a imagem apenas na direção horizontal ou vertical. Na segunda passagem, o mesmo *kernel* unidimensional é usado para desfocar na direção restante. O efeito resultante é o mesmo que convolver com um *kernel* bidimensional em uma única passagem, mas requer menos cálculos (A implementação realizada nesse trabalho foi com um *kernel* bidimensional) .

Esse método trabalha com matrizes 3x3, 5x5 e 7x7 (opção escolhida pelo usuário do programa) para aplicar o efeito de *Blur* na imagem. Ele funciona percorrendo a imagem *pixel* a *pixel* colocando cada *pixel* no centro da matriz selecionada. Após colocá-lo no centro da matriz um registrador percorre a matriz recebendo cada *byte* das cores R,G e B de cada posição e somando cada valor de cada *byte* de cor a um registrador próprio para cada cor.

Ao final do processo de percorrer a matriz é realizada uma média do valor de cada registrador (cada cor). Esses

valores serão armazenados no pixel central da matriz e o centro da matriz é deslocado para o próximo pixel da imagem. Processo que é repetido até passar por toda a imagem.

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad K = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figura 5: Exemplos de kernel 3x3 e 5x5 com seus respectivos pesos e médias

Observação: Para evitar que fosse alocada mais memória, ou que a matriz pegasse lixo de memória, foram pulados 1 *pixel* das bordas da imagem no *loop* relativo ao *kernel* 3x3, 2 *pixels* para o *kernel* 5x5 e 3 *pixels* para o *kernel* 7x7.

Figura 6: Comparação entre a imagem original e as imagens após o filtro de *blur* com os tamanho de matriz 3x3,5x5 e 7x7 respectivamente.

4.3. Efeito de Extração de Bordas

4.3.1 Operador de Primeira Ordem

Para extrair as bordas de uma imagem, uma propriedade matemática é bem muito útil: a derivada. A implementação de um operador de segunda ordem facilita o processo de extração de bordas, uma vez que podemos explorar a propriedade de cruzamento de zeros [3]. Então se com um operador derivativo de primeira ordem (tomemos Sobel como exemplo), podemos identificar os pontos de máximo como bordas da imagem (figura 7).

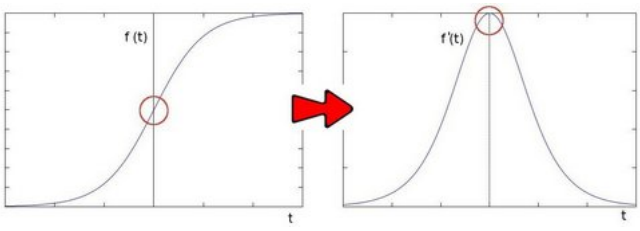

Figura 7: Operador de Sobel

Porém há alguns problemas com os operadores de primeira ordem, uma delas é que é necessário extrair a magnitude entre a derivada vertical e horizontal, tais operações são definidas por uma convolução da imagem com a Matriz A, seguida de uma convolução com a Matriz B e por último calculando o módulo de ambas as operações. A formula para a extração de bordas para um operador de primeira ordem é dado por 1

$$I_{bordas} = \sqrt{A \circledast I + B \circledast I} \tag{1}$$

Onde, $A = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ e $B = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

É perceptível como tal operação é computacionalmente custosa, e temos como objetivo assegurar o melhor desempenho possível, usar um operador linear como o de Sobel não só envolve duas convoluções como também uma raiz quadrada que deveria ser implementada para ser resolvida numericamente, usando operações de ponto flutuante. Todas as operações são demasiadamente custosas com um baixo custo benefício computacional, uma vez que um operador de segunda ordem pode nos fornecer a mesma informação usando apenas um *kernel*, para tal escolhemos o operador laplaciano.

4.3.2 Operador de Segunda Ordem

Foi implementado um operador laplaciano de primeira ordem, para diminuir a quantidade de iterações necessárias para a obtenção do resultado. No caso do operador de segunda ordem podemos nosso pontos de interesse são os pontos $max(f'(x))$, ou $f''(x) = 0$ (ponto de cruzamento dos zeros), como ilustrado abaixo.

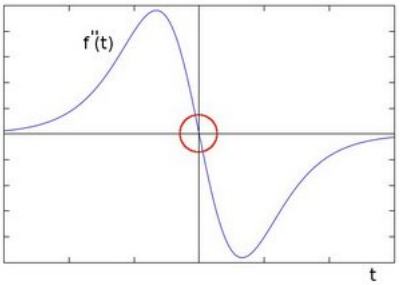

Figura 8: Operador de Laplace

O kernel do laplaciano, também conhecido como operador de Laplace, é dado a seguir:

$$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{2}$$

E podemos fazer uma operação de convolução entre a imagem (I) e L para obtermos a imagem de interesse. Assim,

$$I_{bordas} = I \circledast L \tag{3}$$

Diminuindo a quantidade de instruções necessárias para a extração de bordas. Porém, o operador de segunda ordem é altamente sensível ao ruído, então precisamos fazer um *blur* gaussiano antes a fim de diminuir o ruído.

4.3.3 Filtro Gaussiano

O filtro gaussiano é um filtro passa-baixa (apenas valores com baixa frequência passam)

O filtro é baseado na função Gaussiana. Um filtro gaussiano de duas dimensões pode ser obtido a partir da multiplicação de duas funções gaussianas de uma dimensão[4]:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-x^2}{2\sigma^2}} \tag{4}$$

Uma função gaussiana com uma distribuição de 2σ *é equivalente ao produto de duas gaussianas de distribuição* σ Tal propriedade permite-nos separar a matriz de kernel com coeficientes gaussianos em uma única linha/coluna, permitindo otimizar nosso código, uma vez que - por ser simétrica e separável - o kernel do filtro gaussiano 3x3 pode ser descrito como:

$$G_{3x3} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \tag{5}$$

Como desejamos uma matriz normalizada com os coeficientes da linha de interesse do triângulo de pascal. Então:

$$K = \frac{1}{16} * \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \tag{6}$$

A definição de uma imagem (I) filtrada por uma função gaussiana (de kernel 3x3) é dada por:

$$I_f = K_{3x3} \circledast I \tag{7}$$

Substituindo a eq. 6 em eq. 7

$$I_f = \frac{1}{16} * \left(\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \circledast I \right) * \left(\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \circledast I \right) \tag{8}$$

Como a linha e a coluna a ser convoluída são iguais, economizamos espaço de memória ao guardar apenas uma linha do kernel, e o número de iterações uma vez que o índice será o mesmo para a convolução horizontal e vertical.

O resultado do filtro é apresentado abaixo:

Figura 9: Imagem antes e depois do filtro de gauss

4.4. Operações com arquivo

Para que possamos operar com arquivos no Mars é necessário fazer uso da função *syscall*. É importante ressaltar também que o Mars deverá ser chamado a partir do mesmo diretório em que está o arquivo a ser operado.

4.4.1 Leitura de Arquivo

Para ler um arquivo e armazená-lo na memória têm que ser utilizados os *syscall* 13, 14 e 16 respectivamente. O *syscall* 13 é uma função para abrir o arquivo, e quando chamada, tem que receber uma *string* com o nome do arquivo e uma *flag* indicando se o arquivo está sendo aberto para leitura, escrita e *append*[1]. Para a leitura de arquivo, a *flag* deverá ser 0, equivalente à *flag* de leitura de arquivo. Ao final da função será retornado o descritor de arquivo, que será necessário para a utilização dos *syscalls* 14, 15 e 16. O *syscall* 14 é a função de leitura de arquivo. Ela escreve as informações lidas em algum lugar da memória escolhido pelo programador. Por fim, o *syscall* 16 fecha o arquivo. Ao final desse processo, as informações do arquivo de imagem estarão escritas no espaço de memória "*buffer*", que tem 786486[2] bytes de armazenamento.

4.4.2 Organização do Arquivo para operação

Como o MIPS tem uma organização que divide a memória em *words* de 32 *bits* e o .BMP tem 3 *bytes* de informação para cada *pixel*, a informação que foi armazenada na memória terá que ser adaptada para que possa ser operada. Para isso, foi utilizado um *loop* que armazena a imagem adaptada no espaço de memória "*image*"(endereço

[1] Caso o arquivo não exista, será criado um arquivo e escrito o conteúdo desejado. Caso o arquivo já exista, as informações a serem escritas vão para o final do arquivo.

[2] Esse número equivale a 3x512x512+54, ou seja,o produto entre a dimensão da imagem em bytes e o número de componentes(RGB) somado com o numero de bytes do cabeçalho do formato BMP.

0x10010000)[3].Esse *loop* funciona armazenando cada componente RGB de um determinado *pixel* em registradores diferentes, deslocando esses *bytes* para a posição correta e concatenando todos na mesma palavra. No término do *loop*, os *pixels* estarão armazenados na memória em palavras de 32 *bits* no formato 0RGB. É bom observar também que, como o .BMP armazena as informações do arquivo de trás para frente, foi necessário que a leitura fosse feita de trás para frente findando que a imagem fosse visualizada na orientação correta.

Embora a imagem tenha sido organizada quanto sua orientação vertical, ela ainda se apresenta espelhada. é necessário então, outro *loop* que corrija esse problema. Esse loop consiste em apontar para os *pixels* das pontas da primeira linha, trocá-los de lugar, e andar os ponteiros uma posição em direção ao meio da linha. Quando todos os *pixes* da linha forem trocados uma vez, o mesmo é feito na próxima linha, até que a imagem toda tenha sido percorrida.

4.4.3 Organização do Arquivo para escrita

A imagem manipulada está organizada em *words*, com 0RGB em cada uma delas. É preciso que a imagem volte para seu formato original de organização em RGB, e que ela volte para sua orientação original. Primeiro, o mesmo procedimento que foi realizado para espelhar a imagem é repetido novamente. Posteriormente, os dados armazenados em "*image*"serão transferidos para "*buffer*". Isso é feito por meio de um *loop* que copia os dados de "*image*"e escreve de trás para frente em "*buffer*". Ao final desse processo, os *pixels* da imagem manipulada estarão na posição correta e o cabeçalho não foi alterado, tornando a imagem,portanto, pronto para ser escrita em um arquivo.

4.4.4 Escrita de Arquivo

Para escrever em um arquivo, utilizamos a função *syscall* de maneira análoga ao processo de leitura, porém, agora serão utilizados os *syscalls* 13, 15 e 16. A abertura do arquivo funciona da mesma maneira, apenas diferindo nos argumentos passados à função. No caso, foi utilizada a *flag* 1, pois a ideia é criar um novo arquivo com a imagem modificada. O *syscall* 15 é a função de escrita no arquivo. O espaço de memória "*buffer*"é onde está a imagem modificada, e o nome do arquivo foi pré-determinado como "saida.bmp". Após a função de escrita, é chamado o *syscall* 16 para fechar o arquivo.

[3]Para que a imagem possa ser lida na ferramenta *Bitmap Display*, ela precisa estar em um dos endereços estabelecidos nela. O endereço escolhido pelo grupo foi o começo dos dados estáticos.

5. Resultados

Utilizando a ferramenta *Instructions Statistics* do MARS, obtivemos os seguintes dados:

	BL	B(3x3)	B (5x5)	B (7x7)	EB
ALU	34%	47%	42%	40%	36%
JUMP	3%	1%	1%	1%	2%
BRANCH	14%	13%	11%	10%	14%
MEMORY	38%	19%	23%	25%	6%
OTHER	10%	19%	23%	24%	42%

	SA	Menu	Exit
ALU	54%	60%	57%
JUMP	4%	0%	0%
BRANCH	7%	12%	29%
MEMORY	35%	28%	0%
OTHER	0%	1%	14%

Tabela 1: Tabela de percentuais de utilização das instruções tipo ALU,Jump,Branch, Memory e Other. BL -Binarização Linear; B - Blur; EB - Extração de borda; SA - Salvar Arquivo

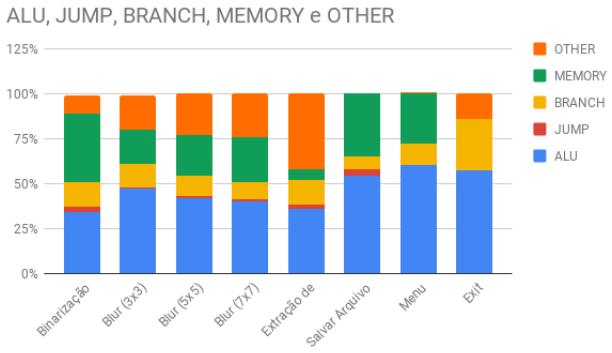

Figura 10: Gráfico de coluna de percentuais de utilização das instruções tipo ALU, Jump, Branch,Memory e Other.

Como pode-se notar, a menor parte das operações realizadas no código são operações de memória. Isso é bom para o desempenho, pois as operações com registradores são, de maneira geral, mais rápidas que as operações com memória. O tipo de instrução mais presente é o de ULA.

6. Discussão e Conclusões

Após a implementação do código foram levantados alguns pontos a serem considerados, tanto em relação a desempenho quanto a melhorias que podem ser implementadas no futuro. O primeiro deles é que o uso de *macros* me-

lhora o desempenho pois trabalha no princípio de agilizar o caso comum. Além disso, o código fica mais legível, o que facilita eventuais *debugs* ou desenvolvimentos em grupo.

Nas operações de convolução baseadas em *kernel*, para evitar que o programa utilizasse em seus cálculos lixos de memória, foi decidido que o efeito seria aplicado com um offset baseado no tamanho do *kernel*. Isso fez com que a imagem final possuísse partes em que o efeito não foi efetivado.

Na operação de filtro Gaussiano, há uma perda de energia na imagem. Isso se deve ao fato de que na equação 8 a operação de divisão foi feita com registradores de inteiro, resultando em uma perda de precisão

O método de extração de borda não foi finalizado, tendo sido implementados o filtro gaussiano e a conversão para a escala de cinza ambas pré requisitos para o filtro laplaciano (Método escolhido para implementar a extração de bordas Vide seção 4.3.2)

Durante o desenvolvimento desse trabalho o grupo atingiu os objetivos propostos, lapidando os conhecimentos e técnicas da implementação da linguagem *assembly MIPS* e dos conceitos propostos de visão computacional.

Referências

[1] Site opencv. `https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html`.

[2] A. Conci. O FORMATO BMP DE ARMAZENAMENTO DE IMAGENS. `http://www2.ic.uff.br/~aconci/curso/FormatoBMP.htm`.

[3] D. Paulus and J. Hornegger. *Applied pattern recognition: algorithms and implementation in C++*. Springer Science & Business Media, 2003.

[4] D. Rákos. Efficient gaussian blur with linear sampling. `http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/`, 2010.