Desenvolvimento de Aplicações em Assembly MIPS

Dep. Ciência da Computação - Universidade de Brasília (UnB)
Organização e Arquitetura de Computadores - Turma B -2º/2018
Douglas Mauricio Bispo de Souza - 14/0136720
Estanislau Jácome Dantas - 14/0137874
Cristovao Bartholo - 14/0135081
Data de realização: 07/10/2018

douglas.1507@hotmail.com estanislaudantas@mecajun.com cristovao@live.com

Abstract

O entendimento das estruturas básicas de representação de dados é fundamental nos dias de hoje em que muitas informações são dados escritos de uma forma particular. Uma dessas formas é agrupar dados e interpretá-los como uma matriz computacional, fazendo com que esses dados representem uma imagem. Logo, é de fundamental importância que projetistas saibam como manipular esses dados e como salvá-los afim de modoficar essas imagens. Para isso, necessitam seguir alguma conveção de representação de dados que faça com que qualquer sistema possa ler essa sequência de dados em memória e intepretá-lo como a devida imagem que representa. Assim, coompreender os formatos de codificação desses bits são fundamentais para trabalhar com imagens.

1. Objetivos

Implementar, testar e validar uma aplicação em Assembly MIPS 32 bits no ambiente MARS que receba uma imagem de entrada codificada em Bitmap de 24 bits e seja capaz de ler e escrever imagens coloridas nesse formato, apresentando-as com o auxílio da ferramenta Bitmap Display (disponível na versão 4.5 do MARS). A mesma aplicação deve ainda permitir que um usuário forneça uma imagem de entrada no formato descrito e selecione efeitos de borramento, extração de bordas ou binarização por limiar, no qual o mesmo seleciona os parâmetros de máscara e obtém o arquivo de saída desejado. A aplicação tem o intuito de familiarização com a linguagem de programação usada e de metodologias de aplicações eficientes e otimizadas.

2. Introdução

As imagens podem ser representadas por grandes matrizes nos computadores. Cada elemento de uma matriz será chamado de pixel, e quanto mais pixels, maior e melhor (em termos de resolução) a imagem será. A cada ponto da imagem exibida na tela teremos um pixel, de forma que a maioria das imagens requer um número muito grande de pixels para ser representada completamente. Pensemos em uma imagem comum de 800 pixels de largura por 600 de altura que necessita de 3 bytes para representar cada pixel (um para cada cor primária RGB) e mais 54 bytes de cabeçalho. Isso totaliza 1.440.054 bytes. Assim, fica evidente que devemos organizar todos esses bytes e identificá-los, assim como procurar técnicas de compressão para os mesmos se possível.

Foi pensando assim que se desenvolveu o formato Bitmap de representação de imagens. Um bitmap é uma matriz de bits que especifica a cor de cada pixel em uma matriz MxN de pixels. O número de bits dedicados a um pixel individual determina o número de cores que podem ser atribuídos a esse pixel. Por exemplo, se cada pixel for representado por 4 bits, um determinado pixel pode ter até 16 cores diferentes ($2\hat{4} = 16$). A Figura 1 apresenta uma ideia básica sobre a representação de imagens por matrizes computacionais de dados. Cada pixel é um conjunto de bytes que contém informações sobre aquele elemento da matriz.

Figura 1. Representação de imagem como matriz de dados

2.1. O padr[a]o de imagem Bitmap

Arquivos de disco que armazenam os bitmaps geralmente contêm um ou mais blocos de dados que armazenam informações como o número de bits por pixel, o número de pixels em cada linha e o número de linhas na matriz. O BMP é um formato padrão usado pelo Windows para armazenar imagens independente de dispositivos e aplicações.

Imagens manipuladas em memória para apresentação em MS Windows são internamente armazenadas em formato DIB (Device-Independent Bitmap) o que permite ao Windows apresentar a imagem em qualquer tipo de dispositivo suportado pelo sistema operacional (telas, impressoras, projetores, etc). O termo "device independent"embute o conceito de especificação de cores da imagem de uma maneira independente das limitações do dispositivo padrão de saída.

Um arquivo de imagem bitmap precisa de um cabeçalho (*header*) onde se encontram informações básicas acerca do mesmo. No padrão bitmap, os primeiros 54 bytes constituem o cabeçalho. Todo arquivo bitmap está dividido em 3 ou 4 partes, que são:

1. Cabeçalho de arquivo (14 bytes): Contém a assinatura característica do Bitmap e informações sobre o tamanho e lay-out do arquivo BMP (disposição dos dados dentro do arquivo) conforme Figura 2.

2. Cabeçalho de mapa de bits (40 bytes): Contém as informações da imagem contida no arquivo. Define as dimensões, tipo de compressão (se houver) e informações sobre as cores da imagem conforme Figura 3.

3. Paleta ou mapa de cores (opcional): Somente estará presente em arquivos de imagens que usam 56 cores ou menos (4 e 8 bits/pixel, conforme Figura 4). Para os demais tipos de BMP, vem em seu lugar a área de dados. No BMP a cor é representada de forma diferenciada dos demais formatos de arquivos de imagens. A paleta é um vetor de bytes da estrutura RGBA, representando a intensidade de cada cor, através de 1 byte. Imagens de 8 bits por pixel, com no máximo 256 cores terão 256 posições na paleta, da mesma forma que imagens de 4 bits por pixel terão 16 posições e imagens de 1 bit por pixel terão 2 posições na paleta.

4. Área de dados da imagem contida no arquivo: Dados que permitem a exibição da imagem propriamente dita, o dados dos pixels a serem exibidos. Podem ser com ou sem compressão, contendo informações do método de compressão no caso da última alternativa citada.

Campo	Bytes	Descrição
BfType	2	Assinatura do arquivo: os caracteres ASCII "BM" ou $(42\ 4D)_h$.É a identificação de ser realmente BMP.
BfSize	4	Tamanho do arquivo em Bytes
BfReser1	2	Campo reservado 1; deve ser ZERO
BfReser2	2	Campo reservado 2; deve ser ZERO
BfOffSetBits	4	Especifica o deslocamento, em bytes, para o início da área de dados da imagem, a partir do início deste cabeçalho. - Se a imagem usa paleta, este campo tem tamanho=14+40+(4 x NumeroDeCores) - Se a imagem for true color, este campo tem tamanho=14+40=54

Figura 2. Estrutura detalhada do cabeçalho de arquivo Bitmap

Campo	Bytes	Descrição
BiSize	4	Tamanho deste cabeçalho (40 bytes). Sempre $(28)_h$.
BiWidth	4	Largura da imagem em pixels
BiHeight	4	Altura da imagem em pixels
BiPlanes	2	Número de planos de imagem. Sempre 1
BiBitCount	2	Quantidade de Bits por pixel (1,4,8,24,32) Este campo indica, indiretamente, ainda o número máximo de cores, que é $2^{\text{Bits por pixel}}$
BiCompress	4	Compressão usada. Pode ser: 0 = BI_RGB _ sem compressão 1 = BI_RLE8 – compressão RLE 8 bits 2 = BI_RLE4 – compressão RLE 4 bits
BiSizeImag	4	Tamanho da imagem (dados) em byte - Se arquivo sem compressão, este campo pode ser ZERO. - Se imagem em true color, será Tamanho do arquivo (Bfsize) menos deslocamento (BfOffSetBits)
BiXPPMeter	4	Resolução horizontal em pixels por metro
BiYPPMeter	4	Resolução vertical em pixels por metro
BiClrUsed	4	Número de cores usadas na imagem. Quando ZERO indica o uso do máximo de cores possível pela quantidade de Bits por pixel , que é $2^{\text{Bits por pixel}}$
BiClrImpor	4	Número de cores importantes (realmente usadas) na imagem. Por exemplo das 256 cores, apenas 200 são efetivamente usadas. Se todas são importantes pode ser ZERO. É útil quando for exibir uma imagem em 1 dispositivo que suporte menos cores que a imagem possui.

Figura 3. Estrutura detalhada do cabeçalho de mapa de bits

Campo	Bytes	Descrição
Blue	1	Intensidade de Azul. De 0 a 255
Green	1	Intensidade de Verde. De 0 a 255
Red	1	Intensidade de Vermelho. De 0 a 255
Reservado	1	Campo reservado deve ser ZERO sempre

Figura 4. Paleta/mapa de cores - definição tabela de cores

2.2. Filtros

Como destacado, as imagens são muito utilizadas nos dias de hoje e em muitas vezes, precisamos manipular ou realizar operações nessas imagens a fim de atender alguma necessidade. Para isso, fazemos algumas operações com a matriz de pixels.

Dentre as operações necessárias nesse experimento, podemos destacar o efeito de borramento que se baseia em filtros lineares para suavização de detalhes da imagem. Assim, um filtro de borramento/suavização calcula uma média dos pixels contidos na vizinhança da máscara de filtragem, na qual o tamanho da vizinhança da máscara determina o grau de perda dos detalhes da imagem e o grau de suavização. Logo, ao substituir um valor de cada pixel de uma imagem pela média dos níveis de intensidade da vizinhança definida pela máscara, o processo resulta em uma imagem com perda da nitidez, ou seja, com redução das transições repentinas nas intensidades. É importante destacar que quando falamos de realizar a média entre os pixels vizinhos, não nos limitamos apenas a média aritmética, mas dependendo da aplicação e da necessidade, podemos ter um uso de média geométrica (distribuindo pesos de acordo com a distância dos pixels para a região central, por exemplo), medianas e outras operações matemáticas.

O efeito de extração de bordas é uma técnica de detecção de bordas que pode incluir uma variedade de métodos matemáticos que visam identificar pontos com padrões específicos em uma imagem na qual o brilho da mesma muda

drasticamente, ou seja, com descontinuidades. Os pontos nos quais o brilho da imagem muda drasticamente são tipicamente organizados em um conjunto de segmentos de linhas curvas chamados de arestas. O objetivo de detectar alguma dessas mudanças bruscas no brilho da imagem é capturar algum padrão que possa representar, por exemplo, mudanças nas propriedades da imagem. Assim, devemos ter em mente o que seriam essas bordas. Uma borda, independente do ponto de vista, geralmente reflete as propriedades inerentes de algum padrão específico em uma região da imagem, como as marcas de superfície ou a forma da superfície. Uma aresta reflete a geometria da cena, como padrões/objetos diferenciando-se um do outro.

Uma aresta típica pode, por exemplo, ser a fronteira entre um bloco de cor vermelha e um bloco de amarelo. Em contraste, uma linha (como pode ser extraída por um detector de diferenças RGB) pode ser um pequeno número de pixels de uma cor diferente em um fundo de outra maneira inalterável. Para uma linha, normalmente pode haver uma borda em cada lado da linha.

Temos também a binarização por limiar (*Thresholding*) que é um processo de segmentação de imagens que é baseada na análise do histograma de alguma propriedade (em geral, o nível da cor cinza na imagem). Dado o número de pixels numa imagem ser geralmente muito elevado, pode-se considerar o histograma como uma boa aproximação à densidade de probabilidade da propriedade que ele representa. Logo, a partir de um limiar que deve ser estabelecido de acordo com as características das partes que se quer isolar, a imagem pode ser segmentada em dois grupos: o grupo de pixels com níveis de cinza abaixo do limiar e o grupo de pixels com níveis de cinza acima do limiar (considerando aqui, o nível de cinza como propriedade a ser analisada). Assim, uma imagem qualquer g(x, y) limiarizada pode ser definida por (considerando f(x,y) como a imagem de entrada):

$$g(x,y) = \begin{cases} 1 & \text{se } f(x,y) > \text{T, onde T é o limiar} \\ 0 & \text{se } f(x,y) \leq \text{T, onde T é o limiar} \end{cases}$$

Vale ressaltar que a melhor segmentação da imagem será obtida de acordo com a escolha de um limiar ótimo que faça a melhor separação entre as partes desejadas.

3. Materiais e Métodos

Para implementar o algoritmo de leitura e escrita de imagens coloridas no formato Bitmap de 24 bits, utilizamos a plataforma MARS MIPS Simulator, que permite desenvolver aplicações em Assembly MIPS, assim como utilizamos a ferramenta Bitmap Display para permitir que as imagens sejam visualizadas pelo usuário afim de conferir os efeitos selecionados. Vale lembrar que quando falamos escrita de imagem, nossa aplicação deve disponibilizar opções ao nosso usuário para que o mesmo escolha uma delas. Essas opções são filtros de imagens que serão aplicados na imagem de entrada, sendo elas: borramento, detecção de bordas ou binarização por limiar.

O projeto consiste em receber do usuário um arquivo no formato .bmp e fazer a leitura de seus 54 bytes iniciais, que são as informações sobre a imagem inseridas nos cabeçalhos padrões de um arquivo bitmap e que nos fornecerá parâmetros importantes, como as dimensões da imagem (dimensão da matriz de pixels), paleta de cores e outros. Essas informações serão muito imporatnes porque teremos que separar parte da memória disponível no MARS para salvar a imagem, já que a mesma é composta por pixels, que para nossa aplicação, são bytes da memória. Então, vamos salvar esses bytes e para cada operação que o usuário desejar realizar sobre a imagem, vamos realizar operações com esses bytes.

Para um uso mais abrangente i interativo do programa e para facilitar os testes necessários, fizemos um programa que começa pedindo uma imagem ao usuário, e após a validação da mesma (verificar se ela existe, se é possível abrí-la. Se sim nos dois casos, já carregamos os bytes da imagem para a memória) abre-se um menu inicial e pede uma opção ao usuário. O mesmo pode escolher entre aplicar algum dos filtros citados - borramento (opção 1), detecção de bordas (opção 2), limiarização (opção 3) - trocar a imagem que selecionou por outra (opção 4), aplicar o efeito já utilizado anteriormente com os mesmos parâmetros (opção 5), exportar imagem (opção 6), resetar configurações (opção 7) ou sair do programa (opção 0). Logo, para um correto uso da aplicação, basta salvar a imagem desejada na mesma pasta que está salvo o MARS MIPS Simulator e após compilar o programa, inserir o nome da imagem no arquvio padrão esperado (Bitmap 24 bits). Com isso, podemos fazer um estudo sobre a análise de desempenho do algoritmo implementado, utilizando ferramentas do MARS de contagem e tipo de instrução usada (*Instruction Statistics*).

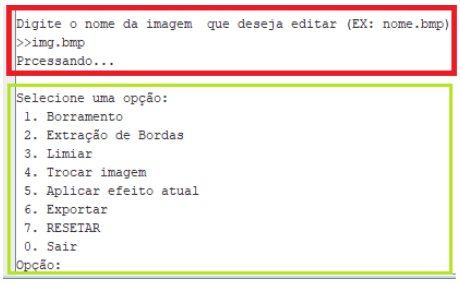

Figura 5. Menu do programa. Na parte inicial vemos a região em que o usuário informa o nome do arquvivo .bmp para trabalhar e logo baixo o menu com as operações realizáveis pela aplicação

3.1. Procedimentos e algoritmo

Como dito, a primeira coisa que o usuário deve fazer é informar o nome do arquivo BMP a ser trabalhado. Com essa informação, o programa pesquisa por esse arquivo e após achá-lo, lê os primeiros 54 bytes do arquivo que é o cabeçalho padrão Bitmap e de lá, recolhe algumas informações importantes. É importante lembrar que assim que a aplicação faz a leitura do arquivo, ela separa inicialmente 3 espaços na memória do tamanho do arquivo. O primeiro espaço é preenchido com o arquivo lido, o arquivo de entrada do usuário. O segundo espaço em memória reservado será o espaço para conter a imagem com as alterações realizadas pelos filtros selecionados pelo usuário (isso quer dizer que o programa preserva a imagem inicial e as alterações são realizadas e salvas em um outro bloco da memória). O terceiro espaço é, por fim, o bloco que recebe a imagem com as alterações já realizadas e a guarda em uma região específica de memória para que a ferramenta Bitmap Display do MARS possa acessá-la.

Sobre a leitura da imagem, após a leitura dos 54 bytes do cabeçalho, o progama começa a ler os dados relativos a imagem em si. Como as informações de um pixel geralmente estão salvas em 3 bytes, optamos por fazer uma lógica que lê esses 3 bytes e os concatenam com um quarto byte de zeros, formando uma word de 4 bytes.

Quando a imagem é lida como uma entrada, ela é salva na memória inversamente na vertical, "de cabeça para baixo", o que pode complicar um pouco sua interpretação para o usuário e o programador. Para isso, decidimos realizar uma operação de swap entre os bytes (um flip na vertical) para deixar a imagem mais fácil de ser manipulada. Para realizar esse flip, devemos observar o tamanho da imagem e preparar um swap entre os pixels superiores e inferiores.

Como teremos linhas com 512 words (para uma imagem de entrada de 512x512), multiplicamos esse valor por 4*número de linhas da imagem - 1 a fim de achar o endereço do início da última linha da imagem. Tendo a primeira linha identificada e a última também, podemos realizar um swap entre elas para colocar a imagem na sua orientação correta. Fazemos a troca de cada um dos elementos de uma linha e em seguida, trocamos as linhas até passar por todas. Devemos observar a operação de swap para que a mesma não entre em ciclo de trocas infinitas. Como estamos trocando a posição dos bytes, vamos trocar os bytes da linha final com os da inicial, da segunda linha com a penúltima e assim por diante. Como as imagens são sempre de tamahos pares, não ocorre o risco de que uma sequência de trocas não tenha condição de parada (quando o ponteiro que marca o limite superior está na mesma linha que o que marca o limite inferior).

3.1.1 Efeito de borramento

Para o filtro de borramento, pedimos um limiar como valor de 0 a 5 para o usuário usar de entrada. Esse limiar será a base para o cálculo do tamanho da matriz que será utilizada para fazer a interpolação. Assim, com uma entrada de valor N (para $1 \leq N \leq 5$), a matriz de interpolação terá dimensões PxP, de onde P = $2N + 1$.

Figura 6. Resultado esperado após aplicação de filtro de borramento. A esquerda temos a imagem original e a direita a imagem manipulada.

3.1.2 Efeito de detecção de bordas

Para o filtro de detecção de bordas (*Edge detectors*) devemos localizar padrões específicos de nível de cinza, como bordas em orientações específicas, cantos, pixels isolados ou formas particulares. Assim, para o processameto dessa imagem devemos realizar cálculos que podem exigir um maior esforço computacional. Visando uma melhora de rendimento, devemos usar algum kernel de convolução. Assim, aplicar esse filtro passa a ser projetar e aplicar um kernel de correspondência de padrões que realize as convoluções desejadas e com essas convoluções podemos fazer aproximações de derivadas direcionais que nos indicarão as bordas.

Logo, fez-se necessário escolher quais kernels iríamos utilizar. O processo de detecção de linhas é realizado através da convolução de dois kernels sobre a imagem, um deles representa a derivada parcial na direção X (horizontal) e o outro na direção Y (vertical). Nesta aplicação foram utilizados três métodos análogos de detecção de bordas, porém com pesos diferentes na matriz de convolução, que são apresentados na Figura 7.

O algoritmo implementado funciona da seguinte forma: no ponto da imagem em que se quer operar se obtém o valor RGB de verde (que é uma aproximação da imagem em preto e branco) dos nove pixels que contém a região envolvendo o pixel e para cada um deles se multiplica pelo peso presente na mesma posição do kernel a ser aplicado. Estes valores obtidos são adicionados a um acumulador que ao final do procedimento com os nove pixels faz a média com o valor de normalização de cada matriz ($\frac{1}{3}$ para Prewitt, $\frac{1}{4}$ para sobel e 1 para Detector de Linhas).

O valor obtido, se for positivo, é então atribuído a uma matriz de mesma dimensão da imagem original. Para o

Prewitt Edge Detectors (gradients): (horizontal and vertical)

$$\frac{1}{3}\begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix}, \frac{1}{3}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix}$$

Sobel Edge Detectors (gradients): (horizontal and vertical)

$$\frac{1}{4}\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}, \frac{1}{4}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix}$$

Line Detectors: (horizontal and vertical)

$$\begin{bmatrix} -1 & +2 & -1 \\ -1 & +2 & -1 \\ -1 & +2 & -1 \end{bmatrix}, \begin{bmatrix} -1 & -1 & -1 \\ +2 & +2 & +2 \\ -1 & -1 & -1 \end{bmatrix}$$

Figura 7. Kernels selecionados para a aplicação

caso da derivada parcial na direção X se atribuiu ao valor de verde e no caso da direção Y se atribuiu o valor para a coloração RGB em vermelho. Outros processamentos poderiam ser feitos para obtenção de um único valor para cada ponto, mas optamos por manter dessa forma para não perdermos a informação da derivada parcial no ponto. Vale observar que o procedimento só é realizado para pontos em que a matriz de convolução se posicione sem exceder o tamanho da imagem.

Figura 8. Resultado esperado após aplicação de filtro de detecção de bordas com kernel sobel. A esquerda temos a imagem original e a direita a imagem manipulada. Vale ressaltar que o aparecimento ou não de determinadas cores na imagem de saída vai depender dos parâmetros de projeto.

3.1.3 Efeito de binarização por limiar

Para a binarização por limiar, após o usuário informar um valor de limiar que varie de 0 a 255, o programa percorre cada pixel verificando se dentro da escala RGB do pixel, o valor referente ao verde (*green*) é maior ou menor que o limiar estabelecido pelo usuário. Se o valor de luminosidade do pixel verde for maior, o pixel fica branco (configuramos seus parâmetros de RGB a fim de que o mesmo fique branco) e se for menor, o pixel fica preto (por processo semelhante ao descrito para o pixel branco). Deve-se observar que se o usuário entrar com o valor máximo de 250, a imagem tenderá a ficará toda preta, assim como se entrar com o valor de 0, a imagem tenderá a ficar toda branca.

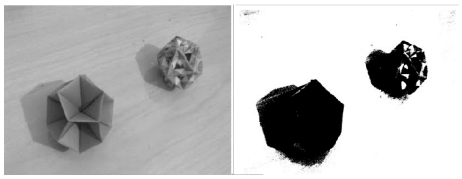

Figura 9. Resultado esperado após aplicação de filtro limiarização. A esquerda temos a imagem original e a direita a imagem manipulada

4. Resultados

Após a implementação do algoritmo, testou-se a aplicação com a imagem cedida para teste no ambiente Moodle da disciplina - Figura 10. Os resultados obtidos foram satisfatórios, pois o programa consegue ler corretamente o cabeçalho de arquivos BMP e passar as informações necessárias para o loops de manipulação das imagens. Com isso, percebemos que as imagens de saída foram sempre conforme o esperado. Em alguns casos, houve uma pequena espera por parte do usuário até que as imagens fossem totalmente processadas, o que se deve ao fato de os loops de manipulação de imagens estarem trabalhando com matrizes razoavelmente grandes com algumas operações matemáticas que demandam um certo poder de processamento. Além disso, temos o fato do programa estar buscando informações na memória, o que tem altíssimos custos computacionais.

Figura 10. Imagem base para teste disponibilizada no Moodle

Assim, com a Figura 10 sendo considerada como imagem base para os testes, variando em alguns momentos apenas suas dimenões, obtivemos os resultados expressos a seguir. O primeiro deles está exposto na Tabela 1 e refere-se a porcentagem de instruções (tipos de instruções) utilizadas apenas para carregar a imagem que o usuário pediu (nesse caso a Figura 10 no formato 512x512).

Instrução	Uso (%)
ALU	43%
Jump	9%
Branch	9%
Memory	22%
Others	17%
Total de intruções	7.082.653

Tabela 1. Percentual de uso de instruções para carregar uma imagem na memória

Podemos ver pela Tabela 1 que a maior parte do uso de instruções para carregar uma imagem na memória foi de operações na ULA (geralmente instruções do tipo R), o que já era esperado, já que para carregar essas imagens precisamos ficar comparando valores de bytes entre si e comparálos na operação de swap para organizar a imagem no bloco da memória. Também foi muito utilizada a memória pelo fato de estarmos lendo os bytes de um arquivo e mapeandoos no bloco de memória, o que requer uma constante operação de leitura de registradores específicos, concatenção de dados e salvá-los em memória.

4.1. Borramento

Para o efeito de borramento, o usuário deve entrar com um valor de 1 a 5 para o grau de borramento. Logo, quanto maior o valor de entrada, maior será a matriz utilizada para os borramentos e mais perceptível é o efeito. Vale destacar que caso o usuário entre com um valor maior que 5, a aplicação aceitará esse valor e irá operar normalmente com ele. Contudo, o tempo de execução pode ser grande, visto que esse efeito realiza muitos cálculos aritméticos de média entre termos da matriz de pixels, que influenciam bastante no tempo de execução, além, pricipalmente, dos acessos mais recorrentes a memória. Os resultados podem ser conferidos na Figuras 11, 12 e 13 abaixo.

Figura 11. Imagem de saída após uso do efeito de borramento com grau 1 - matriz de borramento 3x3

Figura 12. Imagem de saída após uso do efeito de borramento com grau 3 - matriz de borramento 7x7

Figura 13. Imagem de saída após uso do efeito de borramento com grau 5 - matriz de borramento 11x11

Também foi feita uma análise da porcentagem de instruções utilizadas no uso do filtro de borramento. Com isso, foi montada a Tabela 2 para uma análise do algoritmo.

Instrução	Grau 1 (%)	Grau 2 (%)	Grau 3 (%)
ALU	54	52	51
Jump	5	6	6
Branch	12	12	12
Memory	16	18	18
Others	13	13	13
Total de instruções	3.309.625	7.700.537	14.188.601

Tabela 2. Percentual de uso de instruções do filtro de borramento para diferentes níveis da aplicação.

Podemos perceber pela Tabela 2 que um maior uso de operações aritméticas já era esperado, visto que o filtro de borramento vai trabalhar fazendo média de pixels na região vizinha da máscara. Vale destacar também a considerável

porcentagem dos acessos a memória, que ocorrem pelo fato de o algoritmo ter que buscar constantemente valores dos pixels das regiões vizinhas, que estão salvos na memória (essa ida a memória é gerenciada por algumas instruções de comparação e condicionais no nosso código, o que explica a porcentagem de branch na Tabela 2 ser semelhante a de acessos a memória).

Observando os resultados do filtro de borramento, expostos nas Figuras 11, 12 e 13, podemos observar que o comportamento foi conforme o esperado, já que o mesmo deve calcular uma média de valores dos pixels contidos na vizinhança da máscara de filtragem, sendo que o tamanho da máscara de filtragem influencia diretamente no grau de borramento. Assim, podemos obervar que a Figura 13, que tem grau de borramento 5 e uma matriz de borramento (máscara de filtragem) de 11x11 foi a mais afetada pelo filtro, conforme o esperado teoricamente. Contudo, o algoritmo projetado teve um alto custo computacional, já que em média foram realizadas aproximadamente 8.400.000 instruções, aumentando consideravelmente esse número para um maior grau de borramento (influenciado pela maior matriz de borramento).

4.2. Detecç[a]o de bordas

Nesse caso, temos 3 possibilidades de saídas. O usuário pode escolher entre Prewitt, Sobel e Line Detector. Os resultados obtidos estão expressos nas Figuras 14, 15 e 16 logo abaixo.

Figura 14. Imagem de saída após uso do método de Prewitt para detecção de bordas

Figura 15. Imagem de saída após uso do método de Sobel para detecção de bordas

Figura 16. Imagem de saída após uso do método de Line Detector para detecção de bordas

Também foi feita uma análise de porcentagem de instruções utilizadas em cada um dos procedimentos de aplicação dos filtros de detecção de borda. Com isso, foi montada a Tabela 3 para uma análise de cada método.

Instrução	Prewitt	Sobel	Line Detec.
ALU	71	71	71
Jump	1	1	2
Branch	7	7	7
Memory	10	9	9
Others	11	11	11
Total de instruções	4.402.411	6.293.166	1.968.689

Tabela 3. Percentual de uso de instruções para cada um dos kernels do filtro de detecção de bordas

Para esse caso, as instruções também se comportaram, de forma geral, conforme esperado, já que para uma detecção de borda devemos obeservar a região da imagem que estamos passando e comparar seus valores de verde (RGB, para uma aproximação de preto) e multiplicar por pesos de acordo com o padrão do kernel selecionado, sendo essas operações todas operações matemáticas que precisam do poder de processamneto da ULA para serem executadas, o que faz a porcentagem de uso da ULA ser considavelmente maior que os outros.

Vale observar da Tabela 3 que os kernels utilizados têm porcentagem de uso de instruções muito semelhantes, o que já era esperado, visto que o algoritmo deles são semelhantes em seu grande escopo (como acessam a memória, sequência de comparação e outros). O que os diferencia são os cálculos para a realocação de pixels. Contudo, como todos realizam cálculos matemáticos na ULA, eles acabam se aproximando bastante na porcentagem de uso dessa unidade operativa. Também podemos observar que a média de instruções para esses 3 kernels foi de 4.221.422 intruções realizadas, o que nos permite dizer que o kernel de Line Detection que precisou de 1.968.689 intruções utilizou muito menos instruções para realizar a detecção, o que pode significar ganhos de desempenho significativos em comparação aos outros.

4.3. Binarizaç[a]o por limiar

Nesse caso, a imagem de saída irá variar de acordo com o limiar informado pelo usuário. Os resultados obtidos estão expressos nas Figuras 17, 18 e 19 logo abaixo.

Figura 17. Imagem de saída após uso do filtro de limiarização com valor do limiar em 51

Figura 18. Imagem de saída após uso do filtro de limiarização com valor do limiar em 153

Figura 19. Imagem de saída após uso do filtro de limiarização com valor do limiar em 204

Também foi feita uma análise de porcentagem de instruções utilizadas para a limiarização. Com isso, foi montada a Tabela 4 para uma análise desse filtro.

Instrução	Limiar=0	Limiar=170	Limiar=255
ALU	38	36	35
Jump	15	15	15
Branch	15	15	15
Memory	19	19	19
Others	15	15	16
Total de instruções	344.090	332.844	327.706

Tabela 4. Percentual de uso de instruções para aplicação do filtro de limiarização com diferentes valores de limiar

Da Tabela 4 podemos perceber que esse filtro também se usa bastante de operações aritméticas em seu algoritmo, devido ao alto índice de uso da ULA. Contudo, como as operações são simples (comparar se o limiar de entrada do usuário é maior ou menor que uma região de pixels - instrução de comparação de valores), o uso da ULA é menor se comparado aos outro métodos. Com isso, percebemos a maior porcentagem de acessos em memória para buscar os dados da comparação (alinhados de branch e jump que são as instruções usadas no escopo de nosso algoritmo para realizar os saltos para pegar dados no bloco de memória da região da imagem de entrada e levá-los a região do bloco de memória da imagem manipulada).

Observando os resultados do filtro de limiarização, expostos nas Figuras 17, 18 e 19, podemos observar que o comportamento foi conforme o esperado, já que o mesmo deve analisar o valor do limiar passado pelo usuário e observar a região de pixels em volta. Se a região tiver uma escala de luminosidade do RGB, em especial o verde, abaixo do limiar, esse pixel recebe luminosade 0 (fica preto), e caso contrário, fica branco. Observando as imagens citadas, observamos exatamente esse padrão. Com um valor de limiar baixo, como na Figura 17, muitos dos outros pixels da imagem original terão valores de luminosidade RGB maiores, fazendo com que a imagem fique mais branca. Analisando o inverso, no caso de um limiar muito alto, como na Figura 19, muitos dos pixels da região a ser analisada terão luminosidade RGB menor que o limiar, fazendo-as ficarem pretas.

5. Conclusão

Podemos observar e estudar como trabalhar com as imagens no formato Bitmap 24 bits, assim como interpretar o cabeçalho desse padrão e ler quais os bits de informações são fundamentais de serem interpretados e usados como parâmetros para podermos mapear os dados na memória, assim como quais os espaçamentos das máscaras que devemos utilizar e quais os bytes devemos usar para os cálculos matemáticos das manipulações dos filtros.

Pode-se concluir que as técnicas de manipulação em imagens apresentadas, são técnicas eficientes e tradicionais no cenário do processamento digital e produziram um resultado satisfatório para aplicações simples. Elas funcionaram de acordo com o esperado para o trabalho e são viáveis de implementação em ambiente MIPS.

Com o auxílio da ferramenta *Instruction Statistics* do MARS, observamos as porcentagens de instruções utilizadas em cada processo de nossa aplicação, o que nos permitiu confirmar que as instruções que envolviam algum tipo de aritmética eram predominantes nos algoritmos, já que as aplicação dos filtros se resumiam a operações matemáticas em elementos de matrizes. O que chamou a atenção do grupo foi o alto número de instruções para a execução dos efeitos. Já era de se esperar um número alto de instruções executadas devido ao fato de termos vários loops comparando elementos em matrizes grandes, pegando valores em registradores e e comparando com valores que foram carregados da memória. Contudo, analisando as médias de número total de instruções executadas para cada um dos filtros, que foram 8.400.00 para o borramento,4.221.422 para a detecção de bordas e 334.880 para a limiarização. Assim, podemos concluir que algum tipo de melhoria na escrita do algoritmo ou na eficiência dos processos nas comparações pode ser projetada, visando loops mais curtos que possam diminuir de alguma forma o número médio de instruções totais, principalmente nos efeitos de borramento e de detecção de bordas.

6. Referências

[1] D. A. Patterson and J. L. Hennessy, Computer Organization and Design, 3th ed. Morgan Kaufmann.

[2] GONZALEZ, Rafael C. WOODS, Richard C. Pro-

cessamento Digital de Imagens. 3ªed. São Paulo: Pearson Prentice Hall, 2010.

[3] Processamento de imagens (Acesso em 27/09/2018). [Online]. Disponível em: https://sites.google.com/site/imgprocgpu/

[4] De Oliveira, Marcelo Vasques. Formato de arquivo: BMP, 2000. (Acesso em 27/09/2018). [Online]. Disponível em: http://www2.ic.uff.br/ aconci/curso/bmp.pdf

[5] O formato BMP de armazenamento de imagens. (Acesso em 29/09/2018). [Online]. Disponível em: http://www2.ic.uff.br/ aconci/curso/FormatoBMP.htm

[6] Falcão, Alexandre Xavier e Menotti, David. Segmentação por Limiarização. (Acesso em 01/09/2018). [Online]. Disponível em: http://www.ic.unicamp.br/ afalcao/mo443/slides-aula25a.pdf

[7] Jr, Roger L. Easton. Fundaamentals of Digital Image Processing, 2010. (Acesso em 25/09/2018). [Online]. Disponível em: www.cis.rit.edu/class/simg361/Notes_11222010.pdf