NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Tópicos Avançados em Processamento Digital de Imagem

Trabalho Prático Final

Deteção de veículos na mesma faixa de rodagem

Trabalho realizado pelos alunos:

Nome: Francisco Santos	Número: 57901
Nome: Ricardo Monteiro	Número: 55541

1. Objetivo

Com o desenvolver deste trabalho pretende-se criar uma aplicação que seja capaz de analisar e interpretar sinal de vídeo adquirido por câmaras de veículos para a detecção de faixas de rodagem e viaturas na estrada. Para isso, será utilizado o recurso a Redes Neurais Convolucionais (CNNs) e o processamento paralelo com uso de placa gráfica (GPU) para o processamento de imagens.

2. Introdução

O esquema seguinte mostra os passos do processo proposto para a detecção de veículos em imagens de vídeo:

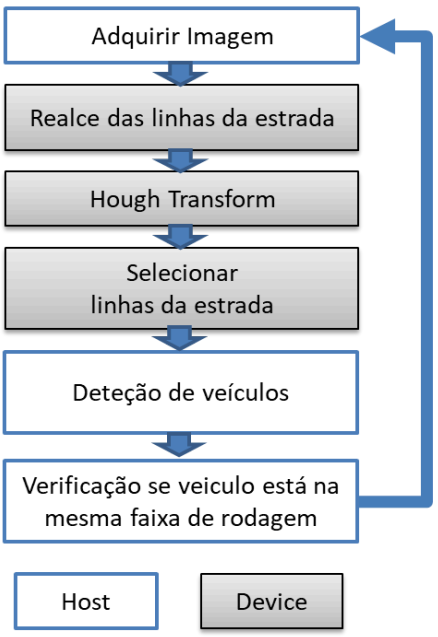

Figura 1 - Arquitetura proposta

O processo a uitlizar é dividido em duas partes principais: a parte de processamento da imagem de forma a detetar as linhas da estrada e a parte de deteção de veículos. Enquanto que a detecção das linhas da estrada será feito a nível da placa gráfica com o uso de computação paralela em opencl, a deteção de carros será feito com o uso de redes neuronais com o Tensorflow e bibliotecas python como o opencv.

O primeiro passo será adquirir as imagens de vídeo, dividindo este em frames e analisando cada uma das imagens utilizando a biblioteca opencv em python e de seguida enviar o frame para o kernel em opencl. Devido às características próprias do opencl e de acordo com o número de plataformas e dispositivos em cada uma destas, poder-se-á então criar um número N de threads do nosso kernel para executar a nossa aplicação. É nestas threads que o nosso programa deve identificar as linhas

que delimitam as faixas de tráfego na estrada, marcando-as visualmente com uma linha azul para facilitar a compreensão ao usuário. Para isso, será empregada um realce nas linhas da imagem, aplicando-se o filtro de sobel para que exista um maior contraste nas zonas de picos da imagem, e de seguida a transformada de Hough para a criação de um “hough space”.

O “hough space” é o resultado da técnica de transformada de Hough que serve para detecção de formas geométricas simples, como linhas e círculos. No exemplo que iremos utilizar, de detecção de linhas, cada ponto da imagem irá contribuir para um voto no “hough space”. Este espaço é representado por um sistema de coordenadas que difere da imagem original. Para linhas, o espaço de Hough é frequentemente expresso em termos de r e θ, onde r representa a distância da origem até a linha mais próxima, e θ é o ângulo de inclinação da linha.

Cada ponto na imagem que pertence a uma linha de certa forma poderá ser representado por uma curva dentro do “hough space”. Significa isto que um cruzamento de múltiplas curvas indica a presença de uma linha na imagem original. Ou explicado de outra forma, a acumulação de votos em pontos no espaço sinaliza que há uma linha com os parâmetros correspondentes na imagem.

Estando os pontos das linhas retirados poder-se-á passar à deteção de veículos, sendo que todos os que forem encontrados fora da região das linhas serão identificados a verde e os encontrados dentro da região serão identificados a vermelho.

3. Metodologia

Começando pela função VideoAquisition, esta vai localizar e guardar o sinal de vídeo que queremos analisar através da função VideoCapture da biblioteca PythonCV. Adicionalmente servirá para retirar a altura, a largura e a hipotenusa do tamanho das imagens do vídeo, informações que serão úteis para funções mais à frente.

```
19  def VideoAquisition():
20      global cap
21      global image_size, MAX_DISTANCE
22
23      pathname = r"C:/Users/santo/Desktop/TAPDI/aula100/"
24      filename = "video3.mp4"
25      cap = cv2.VideoCapture(pathname + filename)
26
27      width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
28      height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
29      MAX_DISTANCE = int(np.sqrt((width ** 2) + (height ** 2)))
30      image_size = (width, height)
```

Figura 2 - Recolha do video

Mas é apenas na função main que iremos dividir o vídeo em diferentes imagens, sendo que iremos analisar cada uma das imagens individualmente.

```
    while (1):
        ret, frame = cap.read()
        if not ret:
            print('No frames grabbed!')
            break
```

Figura 3 - Recolha da Imagem

Sendo que precisaremos de utilizar o sistema de threads no gpu para identificar as linhas, precisaremos primeiro de construir o Programa Kernel, sendo que iremos pesquisar primeiro quais as plataformas presentes no computador, podendo estas ser drives da nvidia, intel, amd,...

Posteriormente iremos buscar todos os dispositivos dentro das plataformas e iremos associar a um contexto criado por nós. A esse contexto vai-se associar o ficheiro “final.cl” e construir o programa. Tal como está apresentado na figura 4.

```
1 usage
def BuildKernel():
    try:
        global platforms, device, ctx, commQ, prog

        platforms = cl.get_platforms()
        platform = platforms[0]

        devices = platform.get_devices()
        device = devices[0]

        ctx = cl.Context(devices)
        commQ = cl.CommandQueue(ctx, device)

        file = open("final.cl", "r")

        prog = cl.Program(ctx, file.read())
        prog.build()
    except Exception as e:
        print(e)
        return False
    return True

```

Figura 4 - Construção do programa Kernel

Já no processo de imagem a imagem, um processo necessário de realizar será passar a imagem de BGR (blue, green, red) para BGRA (blue, green, red, alpha), sendo que alpha é a acentuação das cores e esta é necessária para que a imagem tenha o formato ideal para ser passado para o Kernel.

Começando com o processo de criação de filtro de Sobel na imagem, esta primeira função apenas precisará de dois buffer como argumentos, um para a imagem original e um para a imagem modificada:

```
    img_dst = np.empty_like(imageBGRA)

    memImageIn = cl.Image(ctx, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR,
                          cl.ImageFormat(cl.channel_order.BGRA, cl.channel_type.UNSIGNED_INT8),
                          shape=(img.shape[1], img.shape[0]),  # image width, height
                          pitches=(img.strides[0], img.strides[1]),
                          hostbuf=img.data)
    memImageOut = cl.Image(ctx, cl.mem_flags.WRITE_ONLY,
                           cl.ImageFormat(cl.channel_order.BGRA, cl.channel_type.UNSIGNED_INT8),
                           shape=(img.shape[1], img.shape[0]))

    kernelName1 = prog.sobel_implementation
    kernelName1.set_arg(0, memImageIn)
    kernelName1.set_arg(1, memImageOut)
```

Figura 5 - Seleção de argumentos para o Sobel

Depois de se selecionar os argumentos, será necessário definir o número de threads a trabalhar ao mesmo tempo, decidiu-se assim definir o número de threads correspondente ao número de pixels na imagem, sendo que estarão a trabalhar em grupos de 16*16 pixeis.

```
globalWorkSize = image_size
workGroupSize = (16, 16)
kernelEvent = cl.enqueue_nd_range_kernel(commQ, kernelName1,
                                         global_work_size=globalWorkSize, local_work_size=workGroupSize)

kernelEvent.wait()

cl.enqueue_copy(commQ, img_dst, memImageOut, origin=(0, 0, 0), region=(img.shape[1], img.shape[0], 1))

memImageIn.release()
memImageOut.release()
```

Figura 6 - Conexão entre Devices e Host

Metendo o Kernel a correr, basta esperar que todas as threads acabem para retirar o buffer da imagem modificada e de seguida libertar ambos os buffers para que não ocupem espaço na memória.

Já dentro do “__kernel void sobel_implementation”, este terá uma simples função, agarrar em todos os pixels à volta do pixel que queremos agarrar e utilizar as matrizes de gradiente de sobel que são as melhores para intensificar linhas horizontais e verticais:

1 2 1	-1 0 1
0 0 0	-2 0 2
-1 -2 -1	-1 0 1

Isto será feito para cada uma das cores (verde, azul e vermelho) sendo que se fará depois uma média e uma diferença entre as diferentes cores para verificar se a intensificação dessa zona se verifica em todos os gradientes e se esse registo deverá ficar guardado na imagem destino ou não, sendo que se sim esta ficará a branco e se não, ficará a preto.

Voltando ao main, já temos a imagem com destaque nas linhas da estrada, podemos então passar ao Hough transform. Sendo que o programa já está construído e conseguirá ler a nossa função “ __kernel void hough_implementation” não será preciso construir o kernel outra vez, podendo nós passar diretamente para a fase de passar os argumentos.

```
92      try:
93          lefthough_spaces = np.zeros( shape: (MAX_DISTANCE, MAX_ANGLE), dtype=np.uint32)
94          righthough_spaces = np.zeros( shape: (MAX_DISTANCE, MAX_ANGLE), dtype=np.uint32)
95          region_yUp = 130  # linha de baixo
96          region_yDown = 100  # linha de cima
97
98          memImageIn = cl.Image(ctx, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR,
99                                cl.ImageFormat(cl.channel_order.BGRA, cl.channel_type.UNSIGNED_INT8),
100                               shape=(img.shape[1], img.shape[0]),  # image width, height
101                               pitches=(img.strides[0], img.strides[1]),
102                               hostbuf=img.data)
103
104         LeftmemBufferHS = cl.Buffer(ctx, flags=cl.mem_flags.COPY_HOST_PTR | cl.mem_flags.READ_WRITE,
105                                     hostbuf=lefthough_spaces)
106         RightmemBufferHS = cl.Buffer(ctx, flags=cl.mem_flags.COPY_HOST_PTR | cl.mem_flags.READ_WRITE,
107                                      hostbuf=righthough_spaces)
108
109         kernelName2 = prog.hough_implementation
110         kernelName2.set_arg(0, memImageIn)
111         kernelName2.set_arg(1, LeftmemBufferHS)
112         kernelName2.set_arg(2, RightmemBufferHS)
113         kernelName2.set_arg(3, np.int32(image_size[0]))
114         kernelName2.set_arg(4, np.int32(image_size[1]))
115         kernelName2.set_arg(5, np.int32(region_yDown))
116         kernelName2.set_arg(6, np.int32(region_yUp))
```

Figura 7 - Argumentos para Hough transform

Para esta função, é necessário um número de argumentos muito superior visto que para além da imagem transformada que temos, pretendemos criar dois hough spaces, um para criar a linha da esquerda e outro para criar a linha da direita,

adicionalmente teremos argumentos para o tamanho da imagem e para a zona de região que queremos selecionar. Ambas estas servirão para que possamos criar uma seleção dos pontos que queremos para que exista uma maior probabilidade do ponto com mais votos seja o pretendido, ou seja, um ponto da linha.

Já dentro do kernel, a ideia será pegar em cada pixel da imagem e verificar se está dentro da região pretendida, neste caso escolheu-se variar apenas o y, agarrando apenas numa pequena região da estrada, a mais próxima do carro nas imagens devido a ter menos variações de picos e ser mais confiável.

```
70      int2 coord = (int2)(get_global_id(0), get_global_id(1));
71
72      if (coord.y > region_Up && coord.y < (height - region_Down)){
73          uint4 pixel = read_imageui(image, sampler, coord);
74
75          // If pixel is white
76          if (pixel.x == 255 && pixel.y == 255 && pixel.z == 255) {
77              double theta;
78              int rho;
79
80              // Left line
81              for (int i = 0; i < 55; i++) {
82                  theta = i * (M_PI / 180);  // theta in rads
83                  rho = coord.x * cos(theta) + coord.y * sin(theta);
84                  atomic_add(&HS1[(rho * 180 + i)], 1);
85              }
86
87              // Right line
88              for (int i = 125; i < 180; i++) {
89                  theta = i * (M_PI / 180);
90                  rho = (width - coord.x) * -cos(theta) + coord.y * sin(theta);
91                  atomic_add(&HS2[(rho * 180 + i)], 1);
92              }
93          }
94      }
```

Figura 8 - Hough transform

Sabendo que nós queremos apenas guardar os brancos / pontos intensificados no hough space iremos primeiro verificar a cor do pixel e se for branco poderemos guardar os diversos ângulos e a distância deste ao ponto inicial. Será a função “atomic_add(&HS1[(rho * 180 + i)], 1);” que irá nos dar votos neste espaço, sendo que iremos acrescentar ao indice que equivale ao valor que nos deu. Adicionalmente, a razão por se ter escolhido o ângulo de 55 tanto para a esquerda como para a direita é porque as retas estão na diagonal e estes valores apresentam ser os mais confiáveis dos testados.

Estando os hough spaces definidos poderemos então desenhar as linhas:

```
def DrawLines(img, hough_space, hough_space2):
    max_index_left = np.unravel_index(hough_space[:, :85].argmax(), hough_space[:, :85].shape)
    rho = max_index_left[0]
    theta = np.deg2rad(max_index_left[1])
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a * rho
    y0 = b * rho
    x1 = int(x0 + 1000 * (-b))
    y1 = int(y0 + 1000 * a)
    x2 = int(x0 - 1000 * (-b))
    y2 = int(y0 - 1000 * a)
    pt1_left = (x1, y1)
    pt2_left = (x2, y2)
    cv2.line(img, pt1: (x1, y1), pt2: (x2, y2), COLOR, THICKNESS, cv2.LINE_AA)
```

Figura 9 - Desenhar as linhas na imagem

É com esta função que iremos transformar os hough space criados nas linhas que pretendemos. A ideia será analisar no hough space qual o ponto com maior votos, este em princípio fará parte da nossa linha. Pegando na posição desse valor podemos então calcular a distância e o ângulo em relação ao ponto superior esquerdo da imagem tal como demonstra a figura 10.

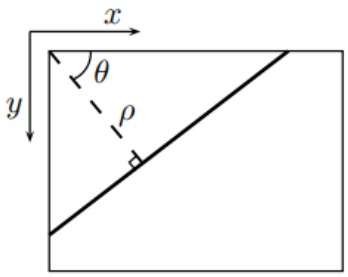

Figura 10 - Representação Polar de uma linha

Como também será possível verificar a linha fica a 90º da linha da distância do ponto selecionado ao ponto inicial. Sabendo isto facilmente podemos criar dois pontos que fazem parte dessa reta e assim criar a linha que pretendemos. Isto é feito para os dois hough spaces sendo que para o do lado direito em vez de se considerar o centro de origem o ponto esquerdo superior, deverá se considerar o ponto direito superior.

Passando para a detecção de carros, este será feito utilizando um modelo yolo já pré feito. Como será possível observar na figura 11, incialmente coloca-se o frame como input do teste e de seguida retirar-se-á todos os objetos que o modelo conseguiu identificar.

```
# Frame preprocessing for deep learning
blob = cv2.dnn.blobFromImage(img, 1 / 255.0,  size: (416, 416),
                             swapRB=True, crop=False)
yolo_net.setInput(blob)

# Getting only output layer names that we need from YOLO
ln = net.getLayerNames()
ln = [ln[i - 1] for i in net.getUnconnectedOutLayers()]
outputs = yolo_net.forward(ln)
```

Figura 11 - Trabalhar com o modelo de redes neuronais

De seguida, tendo os objetos identificados iremos verificar o grau de precisão que este tem sobre a certeza de ser ou não um objeto. Se de facto comprovar-se que a precisão é elevada podermos guardar as coordenadas de um futuro quadrado à volta do objeto.

```
# At least one detection should exist
if len(indices) > 0:
    for i in indices.flatten():
        (x, y) = (boxes[i][0], boxes[i][1])
        (w, h) = (boxes[i][2], boxes[i][3])
        intersection_dentro = cv2.pointPolygonTest(region_selected_lane[0],  pt: (x + w // 2, y + h // 2),
        # intersection_fora = cv2.pointPolygonTest(region_selected_direction[0], (x + w // 2, y + h // 2

        if intersection_dentro >= 0:
            cv2.rectangle(img, (x, y), (x + w, y + h), (0, 0, 255), 2)
        # elif intersection_fora >= 0:
        #    cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 0), 2)
        else:
            cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 0), 2)

return img
```

Figura 12 - Desenhar o quadrado de identificação de objetos

Para a acabar, temos que identificar se o carro que identificamos estará à frente do nosso carro. Como se poderá observar na figura 12, para tal utilizamos as linhas já pré definidas. Utilizando os pontos das linhas podemos facilmente definir uma região e verificar se existe uma interseção entre o carro com as linhas. Caso exista uma interseção, o retângulo ficará vermelho significando que este estará à nossa frente. Caso contrário ficará a verde, sinalizando que está noutra via.

4. Resultados

Figura 13 - Filtro de Sobel

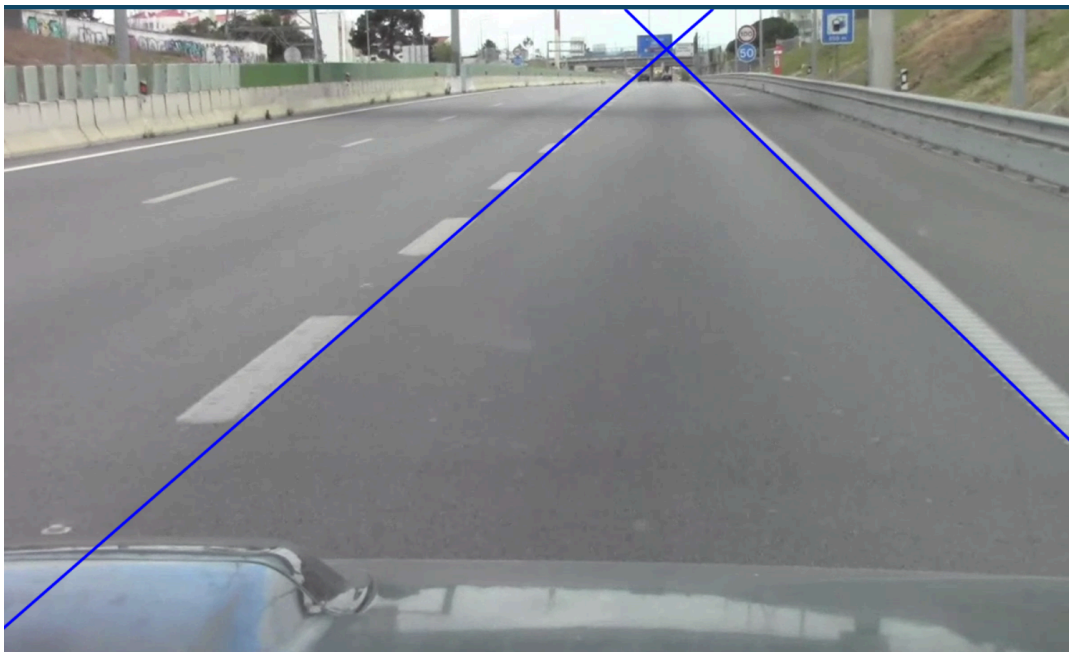

Figura 14 - Identificação das linhas

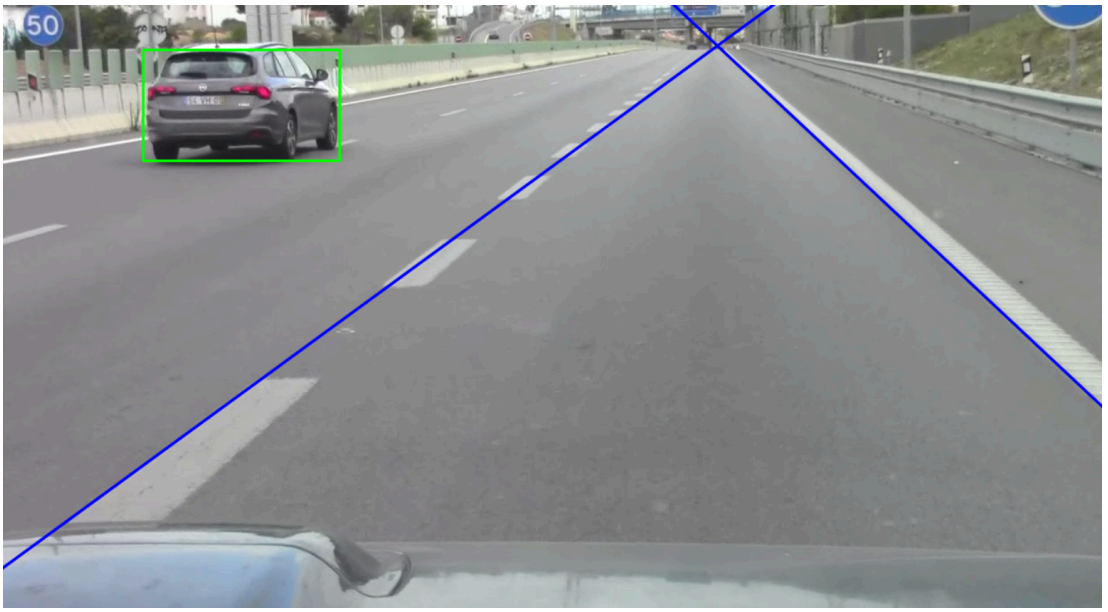

Figura 15 - Identificação de carros

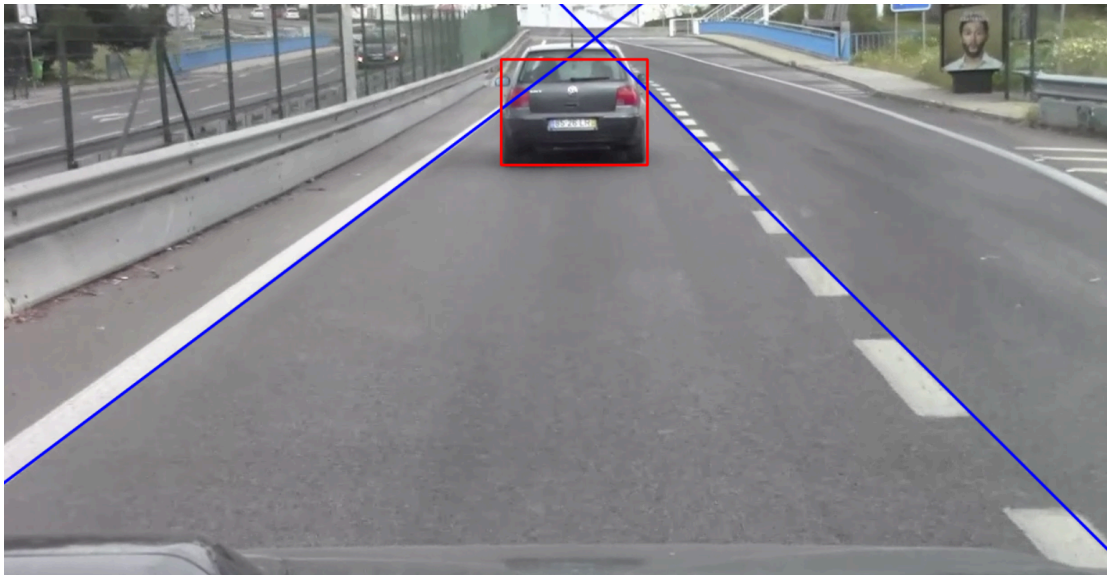

Figura 16 - Identificação de carros com junção das linhas paralelas

5. Anexos

```
__kernel void gray_implementation(
    __read_only image2d_t src,
    __write_only image2d_t dst)
{
    const sampler_t samp = CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

    // receber o pixel
    int2 coord = (int2)(get_global_id(0), get_global_id(1));

    uint4 pixel = read_imageui(src, samp, coord);

    // transformar o pixel em cinzentos
    float gray = 0.299f * pixel.x + 0.587f * pixel.y + 0.114f * pixel.z;
    uint4 gray_pixel = (uint4)(gray, gray, gray, 255);

    write_imageui(dst, coord, gray_pixel);
}



__kernel void sobel_implementation(
    __read_only image2d_t src,
    __write_only image2d_t dst)
{
    const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

    int2 coord = (int2)(get_global_id(0), get_global_id(1));

    uint4 Pixel00 = read_imageui(src, sampler, (int2)(coord.x - 1, coord.y -
1));
    uint4 Pixel01 = read_imageui(src, sampler, (int2)(coord.x, coord.y - 1));
    uint4 Pixel02 = read_imageui(src, sampler, (int2)(coord.x + 1, coord.y -
1));

    uint4 Pixel10 = read_imageui(src, sampler, (int2)(coord.x - 1, coord.y));
    uint4 Pixel12 = read_imageui(src, sampler, (int2)(coord.x + 1, coord.y));

    uint4 Pixel20 = read_imageui(src, sampler, (int2)(coord.x - 1, coord.y +
1));
    uint4 Pixel21 = read_imageui(src, sampler, (int2)(coord.x, coord.y + 1));
    uint4 Pixel22 = read_imageui(src, sampler, (int2)(coord.x + 1, coord.y +
1));

    uint4 Gx = Pixel00 + (2 * Pixel10) + Pixel20 - Pixel02 - (2 * Pixel12) -
Pixel22;
    uint4 Gy = Pixel00 + (2 * Pixel01) + Pixel02 - Pixel20 - (2 * Pixel21) -
Pixel22;

    uint4 G = (uint4)(0, 0, 0, Pixel00.w);
    G.x = sqrt((float)(Gx.x * Gx.x + Gy.x * Gy.x)); // B
    G.y = sqrt((float)(Gx.y * Gx.y + Gy.y * Gy.y)); // G
    G.z = sqrt((float)(Gx.z * Gx.z + Gy.z * Gy.z)); // R

    double diff = abs((int)(G.z-G.y)) + abs((int)(G.z-G.x)) +
abs((int)(G.y-G.x));
    double average = (G.x+G.y+G.z)/3;
```

```
    if(diff>10 && average>50){
        write_imageui( dst, (int2)(coord.x,coord.y) , (uint4)(255,255,255,0));
    }
    else
        write_imageui( dst, (int2)(coord.x,coord.y) , (uint4)(0,0,0,0));
}


__kernel void hough_implementation(
    __read_only image2d_t image,
    __global int* HS1,
    __global int* HS2,
    int width,
    int height,
    int region_Down,
    int region_Up)
{
    const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

    int2 coord = (int2)(get_global_id(0), get_global_id(1));

    if (coord.y > region_Up && coord.y < (height - region_Down)){
        uint4 pixel = read_imageui(image, sampler, coord);

        // If pixel is white
        if (pixel.x == 255 && pixel.y == 255 && pixel.z == 255) {
            double theta;
            int rho;

            // Left line
            for (int i = 0; i < 55; i++) {
                theta = i * (M_PI / 180);  // theta in rads
                rho = coord.x * cos(theta) + coord.y * sin(theta);
                atomic_add(&HS1[(rho * 180 + i)], 1);
            }

            // Right line
            for (int i = 125; i < 180; i++) {
                theta = i * (M_PI / 180);
                rho = (width - coord.x) * -cos(theta) + coord.y * sin(theta);
                atomic_add(&HS2[(rho * 180 + i)], 1);
            }
        }
    }
}
```