Final Report: Kuwahara filter

Nguyen Dang Minh - M23.ICT.008

November 11, 2024

1 Tasks:

Apply Kuwahara filter to this image.

Figure 1: A simple image

Information:

- Height: 720.
- Width: 1280.
- Channel: 3.

2 Kuwahara filter

2.1 Introduction

The Kuwahara filter is a smoothing method that reduces noise while preserving edges, making it effective for image processing. It works by examining small regions around each pixel, calculating variations, and then selecting the region with the lowest variance to soften the pixel's texture without blurring important details. It can be computationally intensive on a CPU, but the filter's simplicity and the independent calculations for each pixel make it applicable for GPU acceleration. The Kuwahara filter is also popular in artistic effects, where its selective smoothing creates a unique, brush-stroke look in images. Each Kuwahara filter have a parameter ω as the window size.

2.2 Logic steps

I applied Kuwahara filter in five steps:

1. Pad the image to handle the edge pixel.
2. Calculate the brightness of each pixel.
3. For each pixel, extract its corresponding four windows.
4. Calculate the standard deviation of each window based on the brightness channels
5. Assign R, G, B values for each pixel based on the mean value of the windows with the smallest standard deviation.

3 CPU Implementation

3.1 Implementation

3.1.1 Pad the image to handle the edge pixel

We use the numpy.pad with mode reflect to keep the integrity of the image. We pad ω pixels around the input image.

```
padded_image = np.pad(image, ((w, w), (w, w), (0, 0)), mode='
    reflect')
```

3.1.2 Calculate the brightness of each pixel

Adapted from RGB to HSV

```
def rgb_to_v(image):
    (H,W,_) = image.shape
    output = np.zeros((H,W), np.float32)
    for i in range(H):
```

```
        for j in range(W):
            r_v = image[i, j, 0]/255
            g_v = image[i, j, 1]/255
            b_v = image[i, j, 2]/255
            output[i, j] = max(r_v, g_v, b_v)
    return output
```

3.1.3 Extract four windows from each pixel

```
def get_windows(image, i, j, w):
    i,j = i+w,j+w
    windows = np.zeros((4, w+1, w+1, 3), np.uint8)
    windows[0] = image[i-w:i+1, j-w:j+1, :]
    windows[1] = image[i:i+w+1, j-w:j+1, :]
    windows[2] = image[i-w:i+1, j:j+w+1, :]
    windows[3] = image[i:i+w+1, j:j+w+1, :]
    return windows
```

3.1.4 Calculate the standard deviation of each window

```
def get_std_v(v_v,i,j,w):
    i,j = i+w,j+w
    v_list = np.zeros((4), np.float32)
    v_list[0] = np.std(v_v[i-w:i+1, j-w:j+1])
    v_list[1] = np.std(v_v[i:i+w+1, j-w:j+1])
    v_list[2] = np.std(v_v[i-w:i+1, j:j+w+1])
    v_list[3] = np.std(v_v[i:i+w+1, j:j+w+1])
    return v_list
```

3.1.5 Assign R, G, B values for each pixel

```
def get_mean(window):
    r_v = int(np.mean(window[:,:,0]))
    g_v = int(np.mean(window[:,:,1]))
    b_v = int(np.mean(window[:,:,2]))
    return r_v,g_v,b_v
```

3.2 Results

Figure 2: Output images with different window sizes (3, 5, 7, 9, 11)

4 GPU Implementation

For the GPU implementation, I modified the order of steps slightly. Unlike the CPU version, which first extracts the window, then calculates the standard deviation, and selects the window with the smallest value, here I calculate the standard deviation of the four windows using only the V channel first. I then find the minimum to select the appropriate window. The initial padding step remains the same as in the CPU version.

Blocksize is (16,16)

4.1 Without Shared Memory

4.1.1 Calculate the brightness of each pixel

```
@cuda.jit
def rgb_to_v_kernel(input_image, v_channel):
    x, y = cuda.grid(2)
    (H,W,_) = input_image.shape
    if y < H and x < W:
        r = input_image[y, x, 0] / 255.0
        g = input_image[y, x, 1] / 255.0
        b = input_image[y, x, 2] / 255.0
        v_channel[y, x] = max(r, g, b)
```

4.1.2 Calculate the standard deviation of 4 windows

```
@cuda.jit(device=True)
def calculate_std(v_channel, start_x, start_y, w):
    sum_v, sum_sq_v, count = 0.0, 0.0, 0
    for i in range(0, w + 1):
        for j in range(0, w + 1):
            nx, ny = start_x + i, start_y + j
            sum_v += v_channel[ny, nx]
            count += 1
    mean = sum_v / count
    for i in range(0, w + 1):
        for j in range(0, w + 1):
            nx, ny = start_x + i, start_y + j
            v = v_channel[ny, nx]
            diff = v - mean
            sum_sq_v += diff * diff
    std = math.sqrt( sum_sq_v/ count)
    return std
```

4.1.3 Select the smallest window

```
if w <= x < W-w and w < y < H-w:
        stds = cuda.local.array((4), numba.float32)
        stds[0] = calculate_std(v_channel, x-w, y-w, w)
        stds[1] = calculate_std(v_channel, x, y-w, w)
        stds[2] = calculate_std(v_channel, x-w, y, w)
        stds[3] = calculate_std(v_channel, x, y, w)
        cuda.syncthreads()
        min_index = 0
        for i in range(1, 4):
            if stds[i] < stds[min_index]:
                min_index = i
        window = cuda.local.array((12, 12, 3), numba.uint8)
        start_x = x-w if min_index in [0,2] else x
        start_y = y-w if min_index in [0,1] else y
        for i in range(w + 1):
            for j in range(w + 1):
                nx = start_x + i
                ny = start_y + j
                window[i, j, 0] = input_image[ny, nx, 0]
                window[i, j, 1] = input_image[ny, nx, 1]
                window[i, j, 2] = input_image[ny, nx, 2]
```

4.1.4 Assign R, G, B values for each pixel

```
@cuda.jit(device=True)
def calculate_mean_color(window, w):
    r_sum, g_sum, b_sum = 0, 0, 0
    count = (w + 1) ** 2
    for i in range(w + 1):
        for j in range(w + 1):
            r_sum += window[i, j, 0]
            g_sum += window[i, j, 1]
            b_sum += window[i, j, 2]
    return r_sum // count, g_sum // count, b_sum // count
```

4.2 With Shared Memory

To load the shared memory, we need to bring in the image data and the brightness values, along with padding of size ω for each block. This task is challenging due to the combined size of ω and the number of threads in each block. If ω is too large, some image data may not load correctly. Although loops could be used so that each pixel loads all four of its windows, this approach would ruins the benefits of parallel computing and lead to significant redundancy and wasted computation.

To simplify, each thread loads its own pixel along with the 8 pixels surrounding it within a radius of ω. With this loading scheme, the filter can handle any $\omega \leq$ block size. The image below illustrates this loading scheme with ω set to 15 or 17 and a block size of $(16, 16)$. A white pixel indicates a loaded pixel, while a black pixel represents a missing pixel.

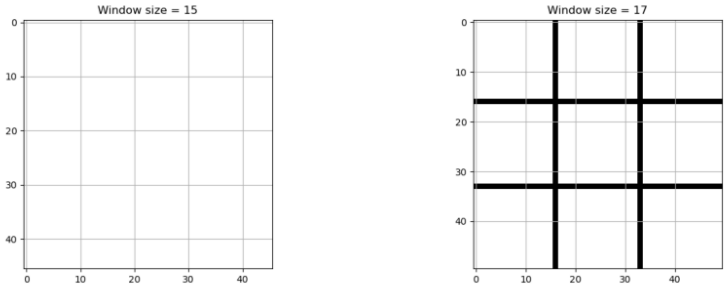

Figure 3: Loading scheme with ω set to 15 or 17 and a block size of $(16, 16)$.

```
@cuda.jit
@cuda.jit
def load_share_mem(shared_image,shared_v,image,v,w):
    x, y = cuda.grid(2)
    x, y = x+w,y+w
    local_x, local_y = cuda.threadIdx.x, cuda.threadIdx.y
    H, W, _ = image.shape
```

```
    if x < W-w and y< H-w:
        shared_v[local_y, local_x] = v[y, x]
        shared_v[local_y-w, local_x-w] = v[y-w, x-w]
        shared_v[local_y+w, local_x-w] = v[y+w, x-w]
        shared_v[local_y-w, local_x+w] = v[y-w, x+w]
        shared_v[local_y+w, local_x+w] = v[y+w, x+w]
        shared_v[local_y-w, local_x] = v[y-w, x]
        shared_v[local_y+w, local_x] = v[y+w, x]
        shared_v[local_y, local_x-w] = v[y, x-w]
        shared_v[local_y, local_x+w] = v[y, x+w]
        for i in range(3):
            shared_image[local_y, local_x,i] = image[y, x,i]
            shared_image[local_y-w, local_x-w,i] = image[y-w, x-w,i
                ]
            shared_image[local_y+w, local_x-w,i] = image[y+w, x-w,i
                ]
            shared_image[local_y-w, local_x+w,i] = image[y-w, x+w,i
                ]
            shared_image[local_y+w, local_x+w,i] = image[y+w, x+w,i
                ]
            shared_image[local_y-w, local_x,i] = image[y-w, x,i]
            shared_image[local_y+w, local_x,i] = image[y+w, x,i]
            shared_image[local_y, local_x-w,i] = image[y, x-w,i]
            shared_image[local_y, local_x+w,i] = image[y, x+w,i]
    cuda.syncthreads()
```

4.3 Results

Figure 4: Kuwahara filter with $\omega = 11$ using GPU

5 Run times comparison

The following table presents a run time comparison between different implementations on the CPU and GPU.

Specifications:

- **CPU**: Intel Core i7-13620H
- **GPU**: NVIDIA GeForce RTX 4050 Laptop GPU

Time (sec)	Window sizes				
	3	5	7	9	11
CPU	60.137	59.560	59.531	60.014	62.423
GPU no SM	0.016	0.021	0.022	0.033	0.043
GPU with SM	0.009	0.018	0.022	0.036	0.036

Table 1: Runtime comparison between 3 different methods on five different window sizes

	Speed up
GPU no SM	2224.350
GPU with SM	2494.573

Table 2: Speed up compare with CPU version

The results show a significant speed-up using the GPU over the CPU for all window sizes, with GPU processing times in milliseconds compared to roughly 60 seconds on the CPU. Utilizing shared memory on the GPU further optimizes performance, especially for smaller window sizes. Compared to the CPU, the GPU achieves a speed-up factor of 2224x, while the GPU with shared memory reaches a factor of 2495x.

6 Conclusions Future works

- GPU performance is much better than CPU.
- Shared memory optimizes performance.

SFuture works

- Improve the shared memory loading scheme to handle bigger window size.