Project

First, we have an image with width=850 and height=567:

Figure 1: A simple image

1 CPU implementation

1.1 Logic steps

There were five steps that I did in this project:

- Pad the image based on the window's size.

- Convert the image from RGB to HSV.
- For each pixel, extract its corresponding four windows.
- Calculate the standard deviation of each window based on the V channels using numpy.std.
- Assign R, G, B values for each pixel based on the mean value of the windows (numpy.mean) with the smallest standard deviation.

1.2 Implementation steps

1.2.1 Pad the image based on the window's size

```
h_pad = args.window_size // 2
w_pad = args.window_size // 2
image = np.pad(
    image,
    pad_width=(
        (h_pad, h_pad),
        (w_pad, w_pad),
        (0, 0),
    ),
    mode="constant",
    constant_values=0,
)
```

1.2.2 Convert the image from RGB to HSV

```
def rgb_to_hsv(image):
    y, x = image.shape[0], image.shape[1]
    output = np.zeros((image.shape), np.float32)
    for i in range(y):
        for j in range(x):
            max_value = max(image[i, j, 0], image[i, j, 1], image[i, j,
    2])
            min_value = min(image[i, j, 0], image[i, j, 1], image[i, j,
    2])
            delta = max_value - min_value
```

```python
            if delta == 0:
                h_value = 0
            elif max_value == image[i, j, 0]:
                h_value = 60 * (((image[i, j, 1] - image[i, j, 2]) / delta) % 6)
            elif max_value == image[i, j, 1]:
                h_value = 60 * (((image[i, j, 2] - image[i, j, 0]) / delta) + 2)
            elif max_value == image[i, j, 2]:
                h_value = 60 * (((image[i, j, 0] - image[i, j, 1]) / delta) + 4)

            if max_value == 0:
                s_value = 0
            else:
                s_value = delta / max_value
            v_value = max_value
            output[i, j, 0] = h_value
            output[i, j, 1] = s_value
            output[i, j, 2] = v_value
    return output
```

1.2.3 Extract four corresponding windows

```python
def extract_window(img, top, height, left, width):
    return img[top : top + height, left : left + width]

window_coordinates = [
[
    h - small_window_height + 1,
    small_window_height,
    w - small_window_width + 1,
    small_window_width,
],
[
    h - small_window_height + 1,
    small_window_height,
    w,
    small_window_width,
```

```
],
[
    h,
    small_window_height,
    w - small_window_width + 1,
    small_window_width,
],
[h, small_window_height, w, small_window_width],
]

window1 = extract_window(image_v, *window_coordinates[0])
window2 = extract_window(image_v, *window_coordinates[1])
window3 = extract_window(image_v, *window_coordinates[2])
window4 = extract_window(image_v, *window_coordinates[3])
```

1.2.4 Calculate the standard deviation of each window

```
std_dev1 = np.std(window1)
std_dev2 = np.std(window2)
std_dev3 = np.std(window3)
std_dev4 = np.std(window4)
min_std = min(std_dev1, std_dev2, std_dev3, std_dev4)
```

1.2.5 Assign the corresponding $R, G, and B$ values for each pixel

```
if std_dev1 == min_std:
    mean_r = np.mean(extract_window(image_r, *window_coordinates[0]))
    mean_g = np.mean(extract_window(image_g, *window_coordinates[0]))
    mean_b = np.mean(extract_window(image_b, *window_coordinates[0]))
elif std_dev2 == min_std:
    mean_r = np.mean(extract_window(image_r, *window_coordinates[1]))
    mean_g = np.mean(extract_window(image_g, *window_coordinates[1]))
    mean_b = np.mean(extract_window(image_b, *window_coordinates[1]))
elif std_dev3 == min_std:
    mean_r = np.mean(extract_window(image_r, *window_coordinates[2]))
    mean_g = np.mean(extract_window(image_g, *window_coordinates[2]))
    mean_b = np.mean(extract_window(image_b, *window_coordinates[2]))
elif std_dev4 == min_std:
    mean_r = np.mean(extract_window(image_r, *window_coordinates[3]))
```

```
    mean_g = np.mean(extract_window(image_g, *window_coordinates[3]))
    mean_b = np.mean(extract_window(image_b, *window_coordinates[3]))
image_output[
    h - small_window_height + 1, w - small_window_height + 1, 0
] = mean_r
image_output[
    h - small_window_height + 1, w - small_window_height + 1, 1
] = mean_g
image_output[
    h - small_window_height + 1, w - small_window_height + 1, 2
] = mean_b
```

1.2.6 Result

We have the resulting image:

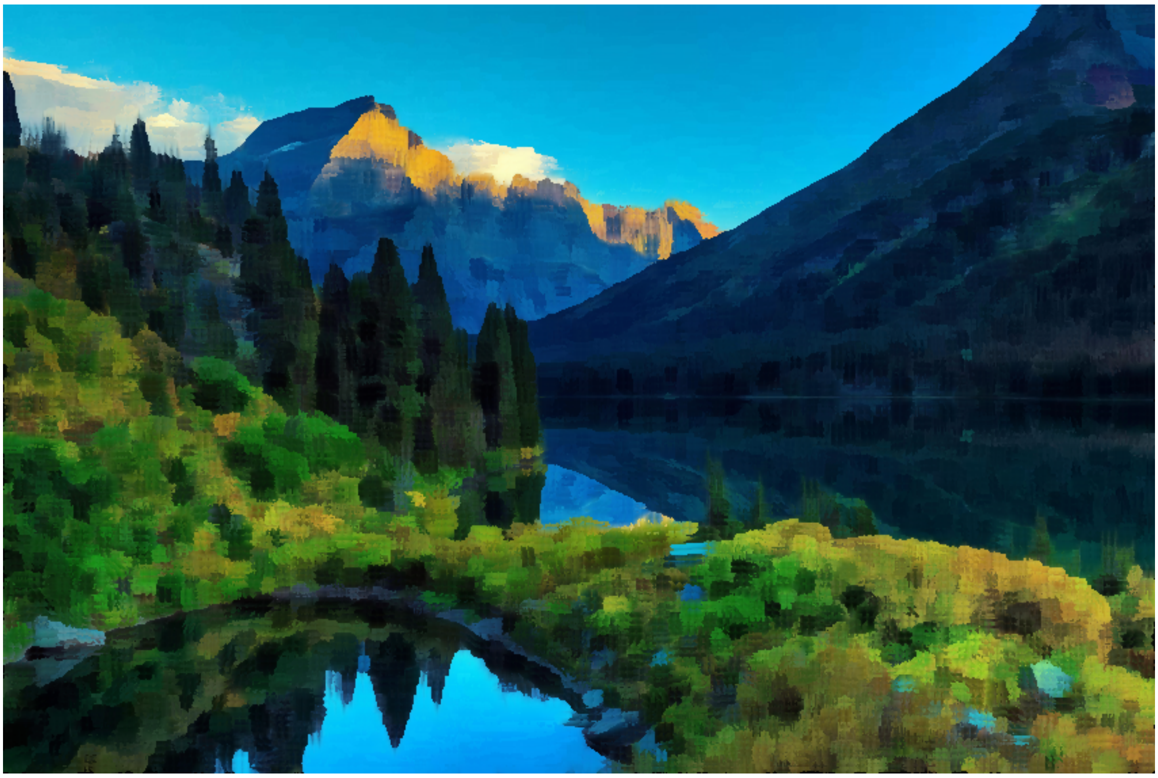

Figure 2: Applying Kuwahara filter on the image using CPU

2 GPU implementation (without shared memory)

2.1 Logic steps

There were five steps that I did in this project:

- Pad the image based on the window's size.
- Convert the image from RGB to HSV.
- For each pixel, extract its corresponding four windows.
- Calculate the standard deviation of each window based on the V channels (from scratch).
- Assign R, G, and B values for each pixel based on the mean value of the windows (from scratch) that has the smallest standard deviation.

2.2 Implementation steps

2.2.1 Pad the image based on the window's size

This step is the same as in the CPU

2.2.2 Convert the image from RGB to HSV

```
@cuda.jit
def rgb_to_hsv(src, dst):
    x, y = cuda.grid(2)
    if y < dst.shape[0] and x < dst.shape[1]:
        max_value = max(src[y, x, 0], src[y, x, 1], src[y, x, 2])
        min_value = min(src[y, x, 0], src[y, x, 1], src[y, x, 2])
        delta = max_value - min_value
        if delta == 0:
            h_value = 0
        elif max_value == src[y, x, 0]:
            h_value = 60 * (((src[y, x, 1] - src[y, x, 2]) / delta) % 6)
        elif max_value == src[y, x, 1]:
            h_value = 60 * (((src[y, x, 2] - src[y, x, 0]) / delta) + 2)
```

```python
        elif max_value == src[y, x, 2]:
            h_value = 60 * (((src[y, x, 0] - src[y, x, 1]) / delta) + 4)

        if max_value == 0:
            s_value = 0
        else:
            s_value = delta / max_value
        v_value = max_value
        dst[y, x, 0] = h_value
        dst[y, x, 1] = s_value
        dst[y, x, 2] = v_value
```

2.2.3 Extract four corresponding windows

This step is a bit different from the CPU version. Instead of having four windows with specific sizes, I just calculate their coordinates(top, left, width, and height)

```python
tops = cuda.local.array(4, numba.int64)
heights = cuda.local.array(4, numba.int64)
lefts = cuda.local.array(4, numba.int64)
widths = cuda.local.array(4, numba.int64)

tops[0] = y - small_window_height + 1
tops[1] = y - small_window_height + 1
tops[2] = y
tops[3] = y

heights[0] = small_window_height
heights[1] = small_window_height
heights[2] = small_window_height
heights[3] = small_window_height

lefts[0] = x - small_window_width + 1
lefts[1] = x
lefts[2] = x - small_window_width + 1
lefts[3] = x

widths[0] = small_window_width
widths[1] = small_window_width
widths[2] = small_window_width
```

```
widths[3] = small_window_width
```

2.2.4 Calculate the standard deviation of each window and get the coordinate of the window that has the smallest deviation

```
smallest_std_window = np.inf
smallest_window_idx = -1
for window in range(4):
    total_sum_window = 0
    top = tops[window]
    left = lefts[window]
    height = heights[window]
    width = widths[window]
    for i in range(top, top + height):
        for j in range(left, left + width):
            total_sum_window += image_v[i, j]

    mean_window = total_sum_window / (width * height)
    sum_of_squared_diff_window = 0

    for i in range(top, top + height):
        for j in range(left, left + width):
            diff = image_v[i, j] - mean_window
            sum_of_squared_diff_window += diff * diff

    std_window = math.sqrt(sum_of_squared_diff_window / (width * height)
)
    if std_window < smallest_std_window:
        smallest_std_window = std_window
        smallest_window_idx = window

top = tops[smallest_window_idx]
left = lefts[smallest_window_idx]
height = heights[smallest_window_idx]
width = widths[smallest_window_idx]
```

2.2.5 Assign the corresponding R, G, and B values for each pixel

```python
total_sum_window_r = 0.0
total_sum_window_g = 0.0
total_sum_window_b = 0.0

for i in range(top, top + 4):
    for j in range(left, left + 4):
        total_sum_window_r += src_rgb[i, j, 0]
        total_sum_window_g += src_rgb[i, j, 1]
        total_sum_window_b += src_rgb[i, j, 2]

dst[y, x, 0] = total_sum_window_r / (width * height)
dst[y, x, 1] = total_sum_window_g / (width * height)
dst[y, x, 2] = total_sum_window_b / (width * height)
```

We have the resulting image:

Figure 3: Applying Kuwahara filter on the image using GPU (without shared memory)

3 GPU implementation (with shared memory)

3.1 Logic steps

There were five steps that I did in this project:

- Pad the image based on the window's size.
- Convert the image from RGB to HSV.
- For each pixel, extract its corresponding four windows.
- Calculate the standard deviation of each window based on the V channels (from scratch).
- Assign R, G, and B values for each pixel based on the mean value of the windows (from scratch) with the smallest standard deviation.

3.2 Implementation steps

3.2.1 Pad the image based on the window's size

I pad the image so that the image's width and height are divisible by 8.

```python
def pad_image_to_divisible_by_8(image, n):
    padded_height = image.shape[0] + 2 * n
    padded_width = image.shape[1] + 2 * n

    left_pad = top_pad = n
    right_pad = bottom_pad = n

    if padded_width % 8 != 0:
        additional_width = 8 - (padded_width % 8)
        left_pad += additional_width // 2
        right_pad += additional_width - (additional_width // 2)

    if padded_height % 8 != 0:
        additional_height = 8 - (padded_height % 8)
        top_pad += additional_height // 2
        bottom_pad += additional_height - (additional_height // 2)
```

```python

    padded_image = np.pad(
        image, ((top_pad, bottom_pad), (left_pad, right_pad), (0, 0)), mode="constant"
    )

    return padded_image, left_pad, right_pad, top_pad, bottom_pad
image, left_pad, right_pad, top_pad, bottom_pad = pad_image_to_divisible_by_8(
    image, h_pad
)
```

3.2.2 Convert the image from RGB to HSV

```python
x, y = cuda.grid(2)
tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
shared_hsv = cuda.shared.array(shape=(8, 8, 3), dtype=numba.float32)

if y < dst.shape[0] and x < dst.shape[1]:
    for i in range(3):
        shared_hsv[ty, tx, i] = src[y, x, i]
cuda.syncthreads()
if ty < 8 and tx < 8:
    max_value = max(
        shared_hsv[ty, tx, 0], shared_hsv[ty, tx, 1], shared_hsv[ty, tx, 2]
    )
    min_value = min(
        shared_hsv[ty, tx, 0], shared_hsv[ty, tx, 1], shared_hsv[ty, tx, 2]
    )
    delta = max_value - min_value
    if delta == 0:
        h_value = 0
    elif max_value == shared_hsv[ty, tx, 0]:
        h_value = 60 * (
            ((shared_hsv[ty, tx, 1] - shared_hsv[ty, tx, 2]) / delta) % 6
```

```
            )
        elif max_value == shared_hsv[ty, tx, 1]:
            h_value = 60 * (
                ((shared_hsv[ty, tx, 2] - shared_hsv[ty, tx, 0]) / delta) +
    2
            )
        elif max_value == shared_hsv[ty, tx, 2]:
            h_value = 60 * (
                ((shared_hsv[ty, tx, 0] - shared_hsv[ty, tx, 1]) / delta) +
    4
            )

        if max_value == 0:
            s_value = 0
        else:
            s_value = delta / max_value

        v_value = max_value
        dst[y, x, 0] = h_value
        dst[y, x, 1] = s_value
        dst[y, x, 2] = v_value
cuda.syncthreads()
```

3.2.3 Extract four corresponding windows

This step is a bit different from the CPU version. Instead of having four windows with specific sizes, I just calculate the coordinates of them (top, left, width, and height)

```
tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
shared_hsv = cuda.shared.array(shape=(26, 26, 1), dtype=numba.float32)
shared_rgb = cuda.shared.array(shape=(26, 26, 3), dtype=numba.float32)
if (
    x < src_hsv.shape[1]
    and y < src_hsv.shape[0]
):
    shared_hsv[ty, tx] = src_hsv[
        y - 3,
        x - 3,
    ]
```

```
    shared_hsv[ty + 6, tx] = src_hsv[
        y + 3,
        x - 3,
    ]
    shared_hsv[ty, tx + 6] = src_hsv[
        y - 3,
        x + 3,
    ]
    shared_hsv[ty + 6, tx + 6] = src_hsv[
        y + 3,
        x + 3,
    ]

if (
    x < src_rgb.shape[1]
    and y < src_rgb.shape[0]
):
    for i in range(3):
        shared_rgb[ty, tx, i] = src_rgb[y - 3, x - 3, i]
        shared_rgb[ty + 6, tx, i] = src_rgb[y + 3, x - 3, i]
        shared_rgb[ty, tx + 6, i] = src_rgb[y - 3, x + 3, i]
        shared_rgb[ty + 6, tx + 6, i] = src_rgb[y + 3, x + 3, i]

cuda.syncthreads()
if (
    x < src_hsv.shape[1]
    and y < src_hsv.shape[0]
):
    tops = cuda.local.array(4, numba.int64)
    heights = cuda.local.array(4, numba.int64)
    lefts = cuda.local.array(4, numba.int64)
    widths = cuda.local.array(4, numba.int64)

    tops[0] = ty
    tops[1] = ty
    tops[2] = ty + small_window_height - 1
    tops[3] = ty + small_window_height - 1

    heights[0] = small_window_height
    heights[1] = small_window_height
```

```
    heights[2] = small_window_height
    heights[3] = small_window_height

    lefts[0] = tx
    lefts[1] = tx + small_window_width - 1
    lefts[2] = tx
    lefts[3] = tx + small_window_width - 1

    widths[0] = small_window_width
    widths[1] = small_window_width
    widths[2] = small_window_width
    widths[3] = small_window_width
```

3.2.4 Calculate the standard deviation of each window and get the coordinate of the window that has the smallest deviation

```
smallest_std_window = np.inf
smallest_window_idx = -1

for window in range(4):
    total_sum_window = np.float32(0)
    top = tops[window]
    left = lefts[window]
    height = heights[window]
    width = widths[window]
    for i in range(top, top + height):
        for j in range(left, left + width):
            total_sum_window += shared_hsv[i, j, 0]
    mean_window = total_sum_window / (width * height)
    sum_of_squared_diff_window = 0

    for i in range(top, top + height):
        for j in range(left, left + width):
            diff = shared_hsv[i, j, 0] - mean_window
            sum_of_squared_diff_window += diff * diff

    std_window = math.sqrt(sum_of_squared_diff_window / (width * height))
    if std_window < smallest_std_window:
```

```
            smallest_std_window = std_window
            smallest_window_idx = window

top = tops[smallest_window_idx]
left = lefts[smallest_window_idx]
height = heights[smallest_window_idx]
width = widths[smallest_window_idx]
```

3.2.5 Assign the corresponding R, G, and B values for each pixel

```
total_sum_window_r = np.float32(0)
total_sum_window_g = np.float32(0)
total_sum_window_b = np.float32(0)

for i in range(top, top + 4):
    for j in range(left, left + 4):
        total_sum_window_r += shared_rgb[i, j, 0]
        total_sum_window_g += shared_rgb[i, j, 1]
        total_sum_window_b += shared_rgb[i, j, 2]
dst[y, x, 0] = total_sum_window_r / (width * height)
dst[y, x, 1] = total_sum_window_g / (width * height)
dst[y, x, 2] = total_sum_window_b / (width * height)
```

We have the resulting image:

Figure 4: Applying Kuwahara filter on the image using GPU (with shared memory)

4 Run time comparison

All experiments are run on Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz and GTX 1650.

	Ws 3	Ws 5	Ws 7	Ws 9
CPU	53.12	56.35	58.46	51.58
GPU (without shared memory)	0.6169	0.5848	0.6291	0.6349
GPU (with shared memory)	0.5844	0.6296	0.5837	0.6678

Table 1: Runtime comparison between 3 different methods on four different window sizes (Ws: window size).

I compare the run time on CPU, GPU (without shared memory), and GPU (with shared memory) with five different window sizes. For the sake of simplicity, I only compare the runtime of Kuwahara filter and not other functions. The execution of the filter on the CPU is very slow, around 51-60s for different window sizes. I noticed an interesting thing: Large windows size do not always run slower than small windows size. I decrease the run time by nearly 90-100 times when I run the code on GPU (without shared memory). Using shared memory results in nearly the same run times as not using shared memory (same block size = 8).

5 Conclusion and future work

Things that I have done in this project:

- Implementation of Kuwahara filter on CPU, GPU (without shared memory), and GPU (with shared memory).
- No hardcoded of most values (except block size and shared array's shape).

Things that I have not done in this project:

- Optimization of Kuwahara filter on GPU.