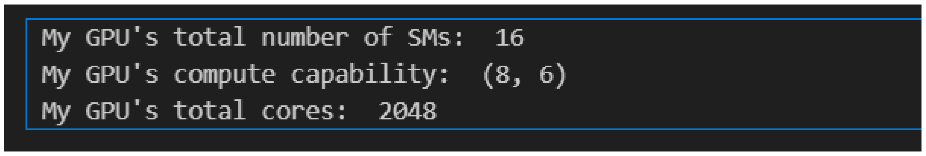
HPC labwork 2, 3, 4 report

Nguyen Huy Hoang

October 2025

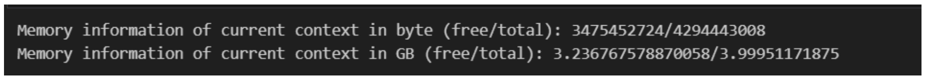
1 Labwork 2

From the figure below, we can see that:



```
My GPU's total number of SMs:  16
My GPU's compute capability:  (8, 6)
My GPU's total cores:  2048
```

Figure 1: Number of SMs and cores



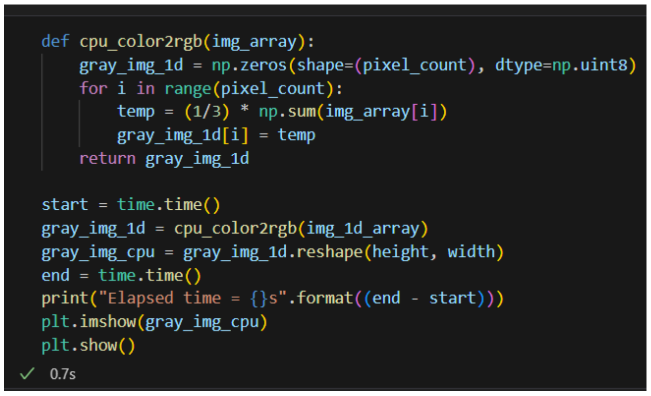
```
Memory information of current context in byte (free/total): 3475452724/4294443008
Memory information of current context in GB (free/total): 3.236767578870058/3.99951171875
```

Figure 2: Number of memory

My GPU's compute capability is (8, 6), which means major revision number is 8, using NVIDIA Ampere GPU Architecture. The figure above shows all the details of SM number, number of cores and the number of memory.

2 Labwork 3

Figure 3: Original Image



```
def cpu_color2rgb(img_array):
    gray_img_1d = np.zeros(shape=(pixel_count), dtype=np.uint8)
    for i in range(pixel_count):
        temp = (1/3) * np.sum(img_array[i])
        gray_img_1d[i] = temp
    return gray_img_1d

start = time.time()
gray_img_1d = cpu_color2rgb(img_1d_array)
gray_img_cpu = gray_img_1d.reshape(height, width)
end = time.time()
print("Elapsed time = {}s".format((end - start)))
plt.imshow(gray_img_cpu)
plt.show()
```

Figure 4: CPU convert to gray image implementation

From the implementation using CPU, the elapsed time is 0.69 s.

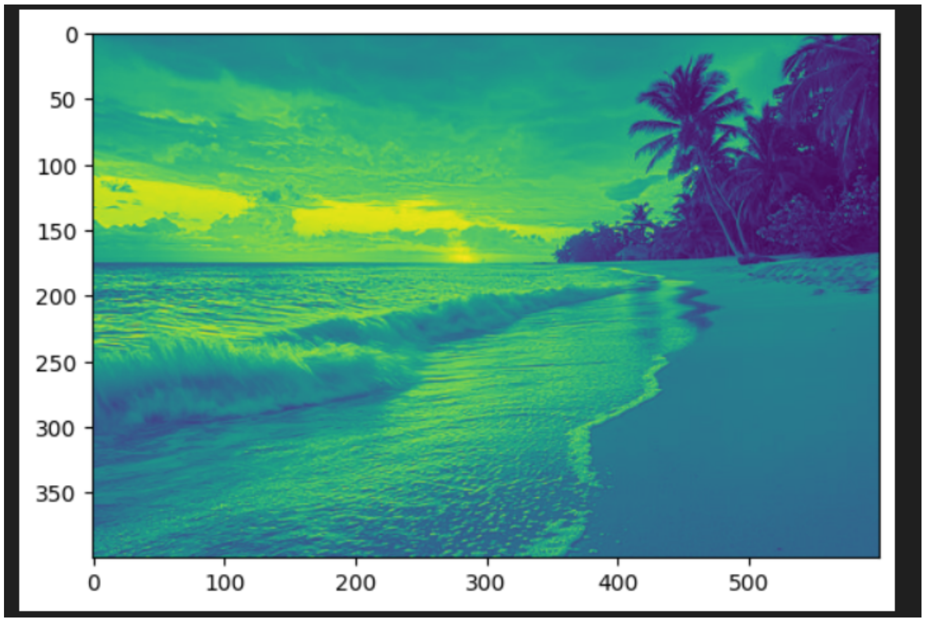
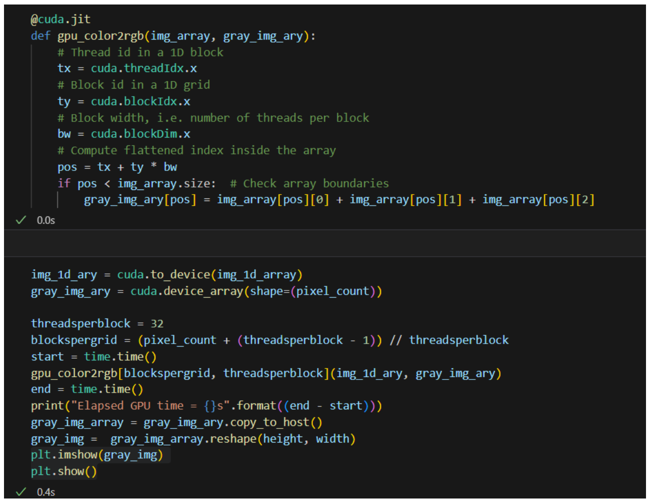


Figure 5: The result of gray image using CPU implementation



```
@cuda.jit
def gpu_color2rgb(img_array, gray_img_ary):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < img_array.size:  # Check array boundaries
        gray_img_ary[pos] = img_array[pos][0] + img_array[pos][1] + img_array[pos][2]
✓ 0.0s
```

```
img_1d_ary = cuda.to_device(img_1d_array)
gray_img_ary = cuda.device_array(shape=(pixel_count))

threadsperblock = 32
blockspergrid = (pixel_count + (threadsperblock - 1)) // threadsperblock
start = time.time()
gpu_color2rgb[blockspergrid, threadsperblock](img_1d_ary, gray_img_ary)
end = time.time()
print("Elapsed GPU time = {}s".format((end - start)))
gray_img_array = gray_img_ary.copy_to_host()
gray_img =  gray_img_array.reshape(height, width)
plt.imshow(gray_img)
plt.show()
✓ 0.4s
```

Figure 6: GPU Implementation

With the implementation of GPU, the elapsed time now becomes 0.26s, which is faster than CPU implementation very much.

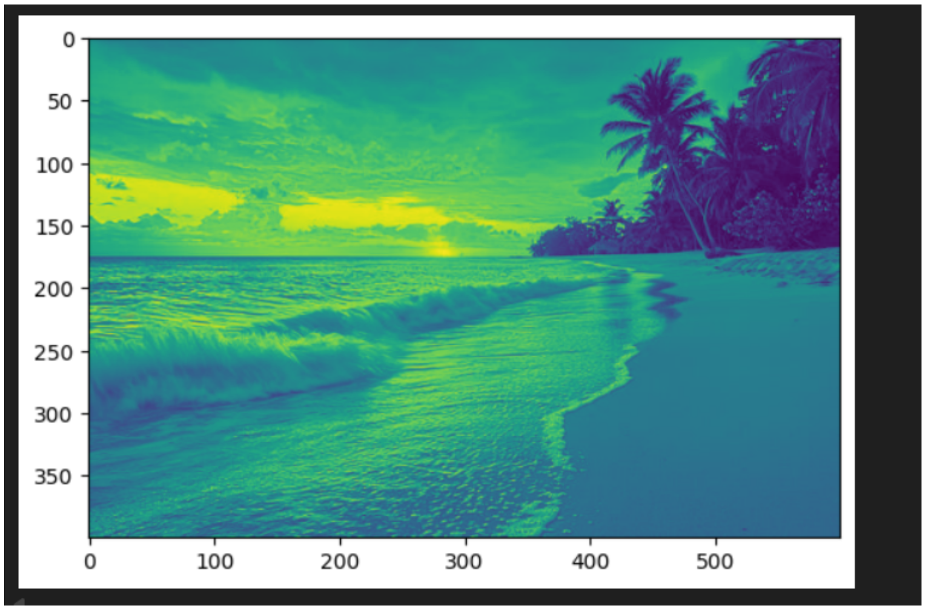


Figure 7: The result of gray image using GPU implementation

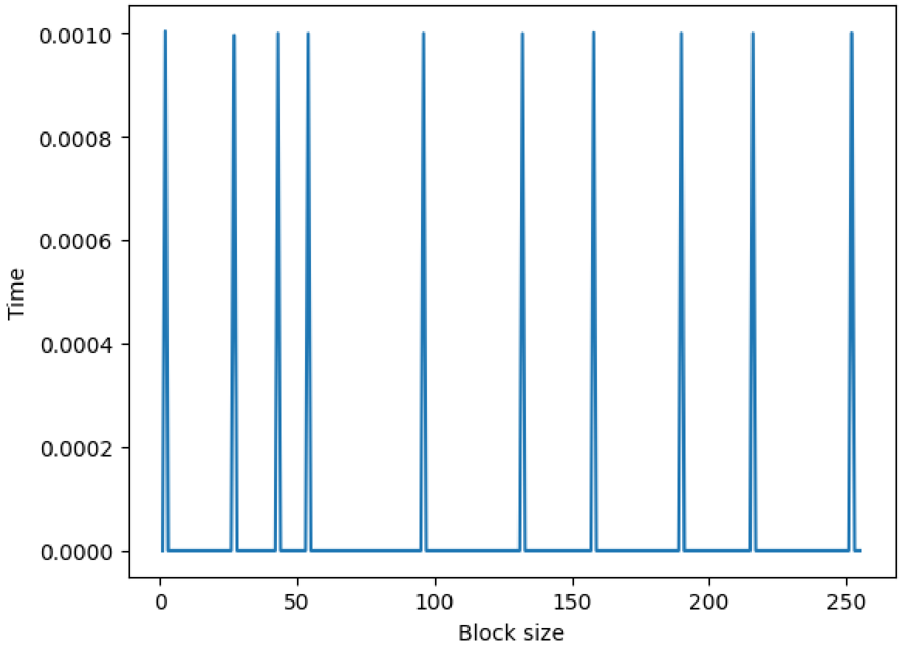
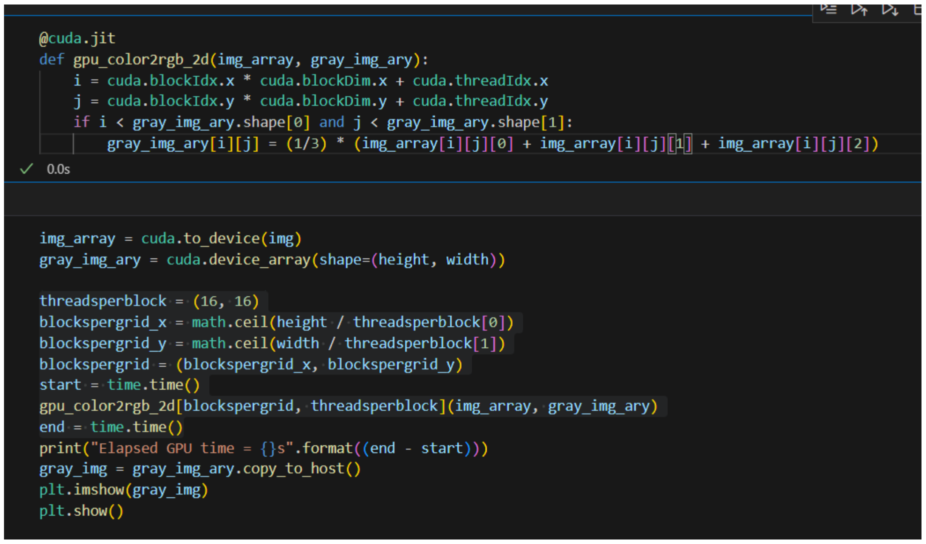


Figure 8: Block size vs time

3 Labwork 4



```
@cuda.jit
def gpu_color2rgb_2d(img_array, gray_img_ary):
    i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    j = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y
    if i < gray_img_ary.shape[0] and j < gray_img_ary.shape[1]:
        gray_img_ary[i][j] = (1/3) * (img_array[i][j][0] + img_array[i][j][1] + img_array[i][j][2])
```

✓ 0.0s

```
img_array = cuda.to_device(img)
gray_img_ary = cuda.device_array(shape=(height, width))

threadsperblock = (16, 16)
blockspergrid_x = math.ceil(height / threadsperblock[0])
blockspergrid_y = math.ceil(width / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
start = time.time()
gpu_color2rgb_2d[blockspergrid, threadsperblock](img_array, gray_img_ary)
end = time.time()
print("Elapsed GPU time = {}s".format((end - start)))
gray_img = gray_img_ary.copy_to_host()
plt.imshow(gray_img)
plt.show()
```

Figure 9: The implementation of convert gray image using 2d blocks

The elapsed time result is similar to 1D block implementation, which is approximately 0.25s.

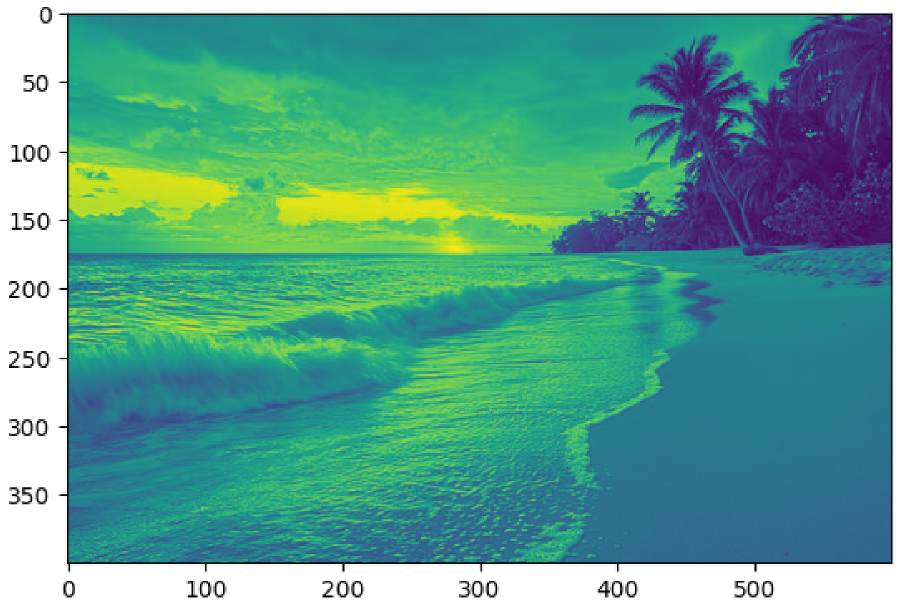


Figure 10: The result of gray image using 2D block GPU implementation

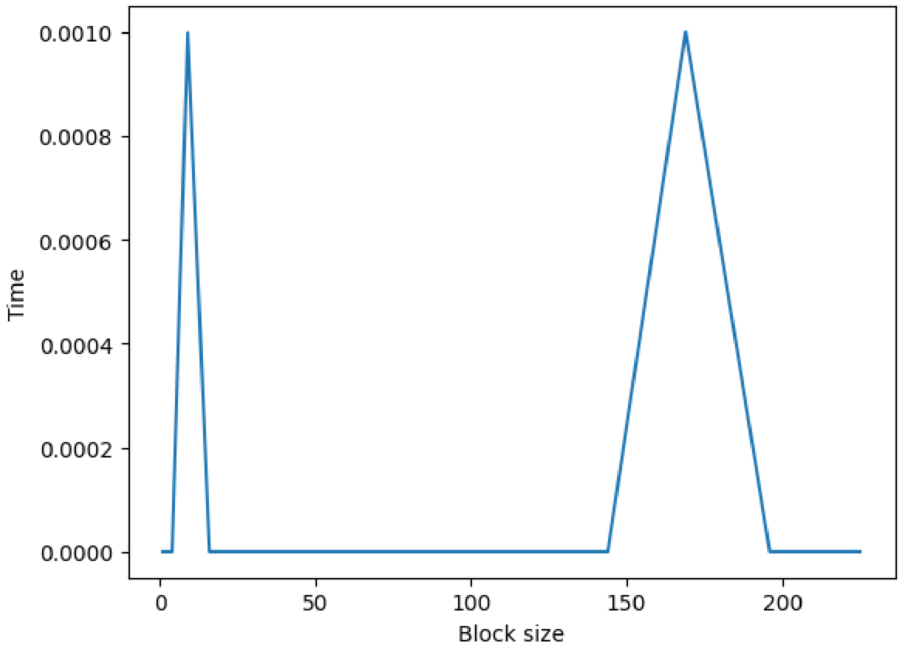


Figure 11: 2D block size vs time

From the figure, we can see that the result of 2D block is much more stable than the 1D block; we only have 2 peak cycles. Meanwhile, in 1D block, it has a short cycle, which leads to the results of many peaks. The reason may come from the fact that the image structure is in 2 dimension (height, width), it is convenient for GPU to allocate the thread in 2 dimensions to perform the processing.