Labwork_3

JACOB Mathieu

03/10/2025

1 Introduction

In this labwork, we are using the followed image :

Figure 1: Image used for the labwork

2 Implement greyscale using CPU

We are writing a simple code that doing the average of the three pixels RGB to turn the image into a grey one.

```
##### CPU #####

import matplotlib.pyplot as plt
from PIL import Image
from IPython.display import display
import numpy as np
from numba import cuda
import time

image = Image.open('/content/image.PNG')

start_time = time.time() # Timer

rgb_array = np.array(image)
grey_array = np.zeros((rgb_array.shape[0], rgb_array.shape[1]), dtype=np.uint8) # Allocate space for the result

width, height = grey_array.shape[0], grey_array.shape[1] # Shape of the image

for x in range(width):
    for y in range(height):
        r = rgb_array[x, y, 0]
        g = rgb_array[x, y, 1]
        b = rgb_array[x, y, 2]
        grey_array[x, y] = int((float(r) + float(g) + float(b)) / 3)  # Average of the 3 pixels -> grey

end_time = time.time() # Timer
print(f"Time taken : {end_time - start_time} sec")

grey_img = Image.fromarray(grey_array)
display(grey_img) # Showing the result
```

Figure 2: Code for greyscale with CPU

Figure 3: Result of greyscale using CPU

3 Implement greyscale using GPU

We are using functions present in the course presentation. The function `config.CUDA_ENABLE_PYNVJITLINK =` solves a problem with Google Colab. We started by allocate memory for the result. Then, we divide the image into blocks and blocks into threads. We use the function for greyscaling which work in the same way as the greyscaling for CPU. Finally, we send the result to the CPU and we display it.

```python
##### GPU #####

from PIL import Image
import time
from IPython.display import display
from numba import cuda
import numpy as np
from numba import config
config.CUDA_ENABLE_PYNVJITLINK = 1

# Function to convert in a grey image
@cuda.jit   # For GPU
def greyscale_gpu(src, dst):
    tidx = cuda.grid(1)
    if tidx < src.shape[0]: # Image limit
        r = src[tidx, 0]
        g = src[tidx, 1]
        b = src[tidx, 2]
        gy = np.uint8((r + g + b) / 3)  # Average of pixels
        dst[tidx, 0] = gy
        dst[tidx, 1] = gy
        dst[tidx, 2] = gy # The 3 pixels become grey

start_time = time.time() # Timer

img = Image.open("/content/image.PNG").convert("RGB") # Load image
rgb_array = np.array(img, dtype=np.uint8)
h, w, c = rgb_array.shape
pixelCount = h * w
flat_img = rgb_array.reshape(pixelCount, 3) # 1D array

grey_array = np.zeros_like(flat_img) # Allocate space for result

threads_per_block = 256
blocks = (pixelCount + threads_per_block - 1) // threads_per_block  # divide the image

d_src = cuda.to_device(flat_img)   # Input
d_dst = cuda.device_array_like(flat_img)  # Output
greyscale_gpu[blocks, threads_per_block](d_src, d_dst) # Function for greyscaling
cuda.synchronize()
hostDst = d_dst.copy_to_host() # GPU -> CPU
grey_img = hostDst.reshape((h,w,3))  # 1D -> 2D

end_time = time.time()  # Timer
print(f"Time taken : {end_time - start_time} sec")

display(grey_img) # Showing the result
```

Figure 4: Code for greyscale with GPU

```
Time taken : 0.07765340805053711 sec
ndarray (390, 724, 3) show data
```

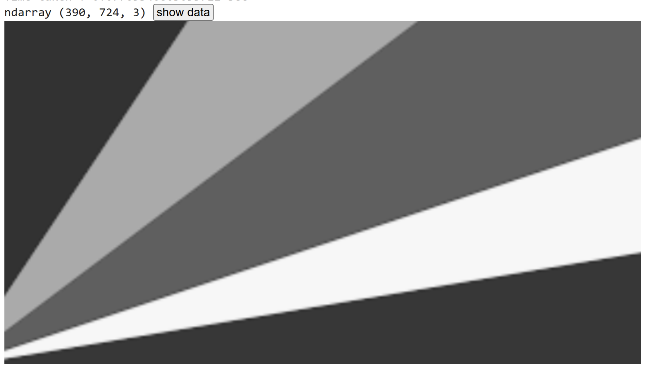

Figure 5: Result of greyscale using GPU

4 Test with different blocks & Graph

For this part, we just take the code from the GPU test and we add a loop in order to check the result with different block size values.

```
#### GPU Blocks & Graph ####

from PIL import Image
import time
import numpy as np
from numba import cuda, config
import matplotlib.pyplot as plt
config.CUDA_ENABLE_PYNVJITLINK = 1

# Function to convert in a grey image
@cuda.jit   # For GPU
def greyscale_gpu(src, dst):
    tidx = cuda.grid(1)
    if tidx < src.shape[0]: # Image limit
        r = src[tidx, 0]
        g = src[tidx, 1]
        b = src[tidx, 2]
        gy = np.uint8((r + g + b) / 3)  # Average of pixels
        dst[tidx, 0] = gy
        dst[tidx, 1] = gy
        dst[tidx, 2] = gy # The 3 pixels become grey

img = Image.open("/content/image.PNG").convert("RGB") # Load image
rgb_array = np.array(img, dtype=np.uint8)
h, w, c = rgb_array.shape
pixelCount = h * w
flat_img = rgb_array.reshape(pixelCount, 3) # 1D array

block_sizes = [1, 4, 16, 32, 64, 128, 256, 512, 1024] # Different block size values
times = [] # Timer

d_src = cuda.to_device(flat_img)   # Input
d_dst = cuda.device_array_like(flat_img)  # Output

# The Loop
for threads_per_block in block_sizes:
    blocks = (pixelCount + threads_per_block - 1) // threads_per_block

    start_time = time.time() # Timer
    greyscale_gpu[blocks, threads_per_block](d_src, d_dst) # Function for greyscaling
    cuda.synchronize()
    end_time = time.time()

    time_ = end_time - start_time # Timer
    times.append(time_)
    print(f"Block size = {threads_per_block}  ->  Time taken = {time_:.6f} sec")

# Graph
plt.figure(figsize=(7,4))
plt.plot(block_sizes, times, marker='o', linestyle='-', linewidth=2)
plt.xlabel('Block size')
plt.ylabel('Execution time (sec)')
plt.title('Block size vs Time')
plt.grid(True)
plt.show()
```

Figure 6: Code to test different block size and draw a graph

```
Block size = 1  ->  Time taken = 0.072486 sec
Block size = 4  ->  Time taken = 0.000708 sec
Block size = 16  ->  Time taken = 0.000238 sec
Block size = 32  ->  Time taken = 0.000137 sec
Block size = 64  ->  Time taken = 0.000126 sec
Block size = 128  ->  Time taken = 0.000123 sec
Block size = 256  ->  Time taken = 0.000134 sec
Block size = 512  ->  Time taken = 0.000134 sec
Block size = 1024  ->  Time taken = 0.000130 sec
```

Figure 7: Result of different block size values

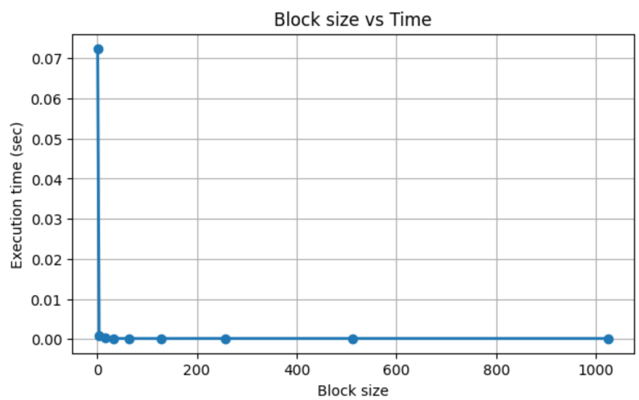

Figure 8: Graph block size vs time

Thus, according to the graph, we observe a decrease in execution time with the increase in block size. This phenomenon can be explained by the fact that for small blocks, there are way too many blocks to execute. This therefore slows down the execution time. Normally, the execution time should also increase when we have too large blocks but this phenomenon is not visible here.