Labwork 3: Introduction to CUDA

Do Thanh Dat - M23.ICT.002

October 7, 2025

Introduction

This article is about Make image RGB-to-gray converter using CPU and Numba CUDA
In this Labwork, the main environment i used is Google Colab with GPU T4 and Python 3

Using CPU

In this part, the grayscale conversion was implemented using a sequential CPU approach. Each pixel is processed one by one in a `for`-loop, where the red (R), green (G), and blue (B) components are averaged to compute the grayscale value:

$$Gray = \frac{R + G + B}{3}$$

The Python implementation is shown below:

```
t0 = time.time()
gray = np.zeros(pixelCount, dtype=np.float32)
for i in range(pixelCount):
    r, g, b = pixels[i]
    gray[i] = (r + g + b) / 3
cpu_time = time.time() - t0
print(f"CPU grayscale time: {cpu_time:.4f} seconds")
```

This implementation is easy to understand but runs slowly because it processes pixels sequentially. The total execution time increases linearly with the number of pixels. In later sections, this result will be compared to the GPU implementation to demonstrate the advantage of parallel execution.

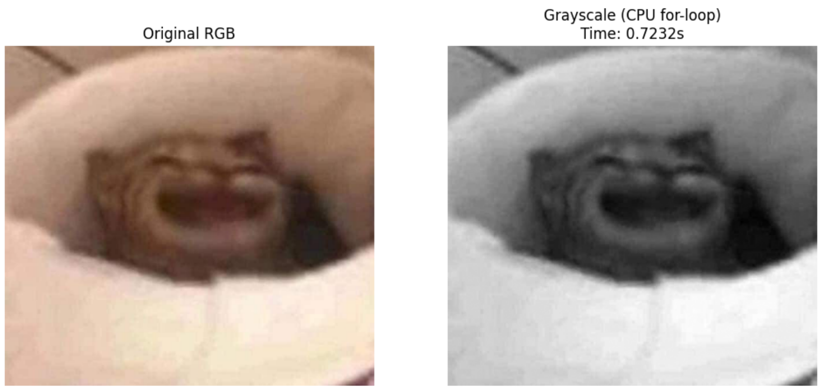

Figure 1: Grayscale image generated using CPU for-loop implementation.

GPU Implementation (Numba CUDA)

The grayscale conversion was also implemented using **Numba CUDA** to leverage GPU parallelism. Each CUDA thread processes one pixel based on its (x, y) position in a 2D grid. This allows thousands of pixels to be converted simultaneously instead of sequentially on the CPU.

$$Gray = \frac{R + G + B}{3}$$

The implementation uses a CUDA kernel defined with `@cuda.jit` as shown below:

```
@cuda.jit
def rgb2gray_kernel(rgb, gray, width, height):
    x, y = cuda.grid(2)
    if x < height and y < width:
        i = x * width + y
        r = rgb[i, 0]
        g = rgb[i, 1]
        b = rgb[i, 2]
        gray[i] = (r + g + b) / 3
```

The kernel is launched using 2D blocks and grids:

```
threads_per_block = (16, 16)
blocks_per_grid_x = (w + threads_per_block[0] - 1) // threads_per_block[0]
blocks_per_grid_y = (h + threads_per_block[1] - 1) // threads_per_block[1]
rgb2gray_kernel[(blocks_per_grid_x, blocks_per_grid_y), threads_per_block](...)
```

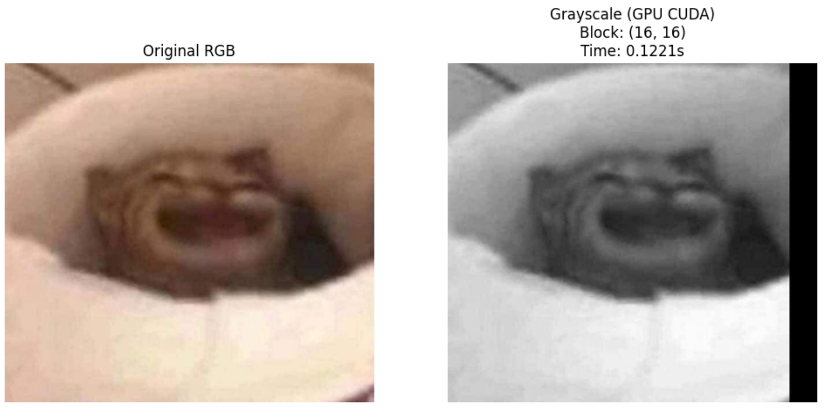

Figure 2: Grayscale image generated using GPU CUDA implementation.

The GPU implementation significantly reduces execution time compared to the CPU version, as each thread performs the computation for one pixel in parallel. Optimal performance was observed with a block size of $(16, 16)$.

Conclusion

In this labwork, an RGB-to-grayscale image conversion was implemented and tested on both CPU and GPU. The CPU version used a sequential `for`-loop, which is simple to implement but slow due to its single-threaded nature. In contrast, the GPU version utilized **Numba CUDA** to parallelize the computation, where each thread handled one pixel independently.
Experimental results showed a significant reduction in execution time on the GPU compared to the CPU, especially for large images. The best performance was achieved with a block size of $(16, 16)$, which provided a good balance between thread occupancy and memory usage.
Overall, this experiment demonstrates the power of GPU computing in accelerating data-parallel tasks such as image processing, highlighting how parallelism can drastically improve performance over traditional CPU-based approaches.