HPC Programming: Labwork 3 Report

Le Chi Thanh

October 19, 2024

1 Introduction

The objective of this project is to convert an RGB image into its grayscale equivalent. This conversion process will be executed using both CPU and GPU methodologies, allowing for a comparative analysis of performance between the two approaches. By evaluating the efficiency and speed of each method, we aim to highlight the benefits and potential trade-offs associated with CPU versus GPU image processing.

2 Implementation

2.1 Image Processing

The first step in the image processing workflow involves acquiring and preparing the image for later processing. We utilize the Python Imaging Library (PIL) to open the specified image. We extract the image width, image height, total number of pixels, and convert the image into a (N,3) array (using NumPy). Each row of this array is a pixel represented in 3 color channels.

$$\begin{bmatrix} 253 & 237 & 247 \\ 253 & 237 & 247 \\ 253 & 237 & 247 \\ \vdots & & \\ 241 & 203 & 224 \\ 241 & 203 & 224 \\ 241 & 203 & 224 \end{bmatrix}$$

2.2 CPU Processing

We implement the RGB-to-Gray process on the CPU. This process occurs within a loop that iterates over each pixel's RGB values. Each RGB value is extracted and converted to an integer to prevent any potential overflow during calculations. The grayscale value is computed using the average method of all 3 RGB values of each pixel.

2.3 GPU Processing

We implement the RGB-to-Gray process on the GPU. The process is broken down into multiple steps in terms of explanation:

2.3.1 CPU feeds GPU with data:

In GPU computing, the first steps is to transfer data from the CPU (host) to the GPU (device). This ensure GPU can perform computations on this data without needing to access the slower system memory.

```
## 1. CPU feeds data to GPU: Allocate memory
dev_input = cuda.to_device(pixels)
dev_output = cuda.device_array((pixel_count, 3), dtype=np.uint8)
```

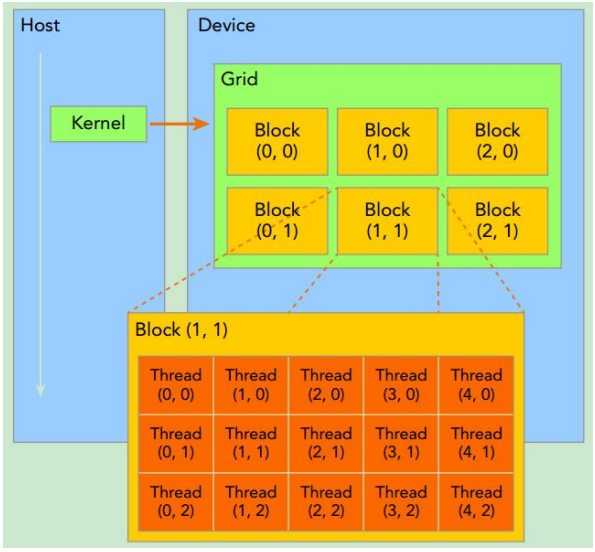

Figure 1: Thread, block, grid in CUDA programming. Source: nanxiao.me

- **cuda.to_device()**: This function allocates memory on the GPU and copies the data from CPU as input. The data (pixels point of image) is transferred from CPU memory to GPU memory.
- **cuda.device_array()**: This function creates an empty array on the GPU to store the output. It has dimensions (total number of pixel, 3 color channels). Each item in the array is represented by an 8-bit unsigned integer (because with 8 bits, we have $2\hat{8} = 256$ possible values which is equal to RGB color range).

2.3.2 CPU asks GPU to process data:

When the CPU ask the GPU to process data, it needs to define the execution configuration which determines how the GPU will organize its parallel processing. This is done through the concepts of blocks and grids in CUDA programming (Figure 1). We review some concepts:

- **Threads**: The smallest unit of parallel execution in CUDA. Each thread runs the same kernel function but operates on different data.
- **Blocks**: A group of threads. Threads within a block can *share memory* and *synchronize* their execution. CUDA architecture limits the numbers of threads per block (1024 threads per block limit).
- **Grid**: A collection of blocks. The entire kernel execution is organized as a grid of blocks.

```
## 2. CPU asks GPU to process
block_size = 64
grid_size = pixel_count // block_size
```

- **block_size = 64**: Number of threads in each block. Each thread will handle caculation of gray color for each pixel point.
- **grid_size = pixel_count/block_size**: Calculates the number of blocks needed or size of one grid.
- Example of GPU processing on 300x300 images. The total number pixel is 90000. We assign 90000 threads with same kernel function to handle every pixel of this image. With the block size equal to 1024. We estimate the grid size to be nearly 87.88 (normally we apply caculating technique to round up to 88) block.

2.3.3 GPU process data in parallel with kernel function:

At this step, we define a *kernel* - which is a function executed on a seperated pixels.

```
## 3. GPU processing with Kernel:
@cuda.jit
def grayscale_kernel(src, dst):
    # Calculate the index for each thread
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if idx < src.shape[0]:
        # Compute grayscale value using average method
        g = (src[idx, 0] + src[idx, 1] + src[idx, 2]) // 3
        dst[idx, 0] = dst[idx, 1] = dst[idx, 2] = g

grayscale_kernel[grid_size, block_size](dev_input, dev_output)
```

- **@cuda.jit**: This is Kernel decorator - which tells Numba (library) to compile this function into a CUDA kernel that can run on the GPU.
- **grayscal_kernel(src, dst)**: This function has 2 parameters. *src*: Input array (source image); *dst*: Output array (destination for grayscale image)
- cuda.threadIdx.x: The index of the current thread within its block
- cuda.blockIdx.x: The index of the current block within the grid
- cuda.blockDim.x: The number of threads per block

We take a look at the first logic function:

```
idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
```

When calculating the grid size, we ensure it is larger than the actual bounds of the image (width and height in pixels). This approach results in some threads handling pixels outside the image bounds. We implement a condition to avoid kernel calculations for these outliers.

```
idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
```

This calculation returns the index of the image pixel corresponding to an active thread. src.shape[0] represents the total number of indexed pixels. We perform kernel calculations only for pixels within the image bounds:

```
g = (src[idx, 0] + src[idx, 1] + src[idx, 2]) // 3
dst[idx, 0] = dst[idx, 1] = dst[idx, 2] = g
```

This code calculates the new color as the average value of the three RGB color components for each pixel.

2.3.4 GPU copy results to CPU

We implement a simple function copy_to_host to copy the result allocated in dev_output on GPU to CPU.

```
host_output_gpu = dev_output.copy_to_host()
print(host_output_gpu)
```

3 Observation of block size and processing time relation

I investigated the relationship between CUDA block size and GPU processing time for the image processing task. By implementing the labwork with various block sizes (16 to 512), I observed that the relationship between block size and processing time is not linear or consistently predictable. At the first glance, this show that GPU performance depends on multiple factors beyond block size alone. Different block sizes can lead to varying performance, not always improving with larger or smaller sizes.

4 Appendix

Full code implementation:

```
import os
import time
import numpy as np
from numba import cuda
from PIL import Image  # Import PIL for saving as JPEG

# Get image
image_path = 'image2.jpg'
image = Image.open(image_path).convert('RGB')  # Read the image properly using PIL
print(f"Processing images:")
print(f"Image width:",image.width)
print(f"Image width:",image.height)
pixels = np.array(image)
pixel_count = image.width * image.height  # Total number of pixels
pixels = pixels.reshape(-1, 3)  # Reshape to (pixel_count, 3) for RGB

print(f"Processing pixels.")
print(pixels)


### RGB to Gray: CPU
print("==================CPU PROCESSING==================")
start_cpu = time.time()
host_output_cpu = np.zeros((pixel_count, 3), dtype=np.uint8)  # Adjust output to store RGB values
for i, (r, g, b) in enumerate(pixels):
    # Convert RGB values to int to prevent overflow
    r = int(r)
    g = int(g)
    b = int(b)
    # Using the average method for grayscale conversion
    g_value = (r + g + b) // 3
    host_output_cpu[i] = (g_value, g_value, g_value)  # Set all channels to the grayscale value
end_cpu = time.time()
print(host_output_cpu)


### RGB to Gray: GPU
print("==================GPU PROCESSING==================")
## 1. CPU feeds data to GPU: Allocate memory
dev_input = cuda.to_device(pixels) #Must be (Nx3) matrix, with N=number of pixel and 3 is channels
dev_output = cuda.device_array((pixel_count, 3), dtype=np.uint8)  # Adjust output to store RGB values

## 2. CPU asks GPU to process
block_size = 64
grid_size = pixel_count // block_size

## 3. GPU processing with Kernel:
@cuda.jit
def grayscale_kernel(src, dst):
    # Calculate the index for each thread
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if idx < src.shape[0]:
        # Compute grayscale value using average method
```

```
        g = (src[idx, 0] + src[idx, 1] + src[idx, 2]) // 3
        dst[idx, 0] = dst[idx, 1] = dst[idx, 2] = g

grayscale_kernel[grid_size, block_size](dev_input, dev_output)

# 4. GPU copy results to CPU
host_output_gpu = dev_output.copy_to_host()
print(host_output_gpu)

# Print processing times
print("==================COMPARISON==================")
cpu_time = end_cpu - start_cpu
gpu_time = time.time() - end_cpu
print(f'CPU Time: {cpu_time:.4f} seconds')
print(f'GPU Time: {gpu_time:.4f} seconds')
print(f'Speedup: {cpu_time / gpu_time:.2f}x')

# Saving resutls
print("==================SAVING==================")
def save_grayscale_image(output_array, output_path, width, height):
    ## Output from CPU and GPU are all (N,3) matrix, with each row a pixel
    gray_image = Image.new('RGB', (width, height))
    gray_image.putdata([tuple(row) for row in output_array])  # Use all channels for grayscale
    gray_image.save(output_path)

cpu_output_path = os.path.splitext(image_path)[0] + '_gray_cpu.jpg'
save_grayscale_image(host_output_cpu, cpu_output_path, image.width, image.height)

gpu_output_path = os.path.splitext(image_path)[0] + '_gray_gpu.jpg'
save_grayscale_image(host_output_gpu, gpu_output_path, image.width, image.height)

print(f"Saving results completed")
```