Report.3.cuda

Ta Quang Minh

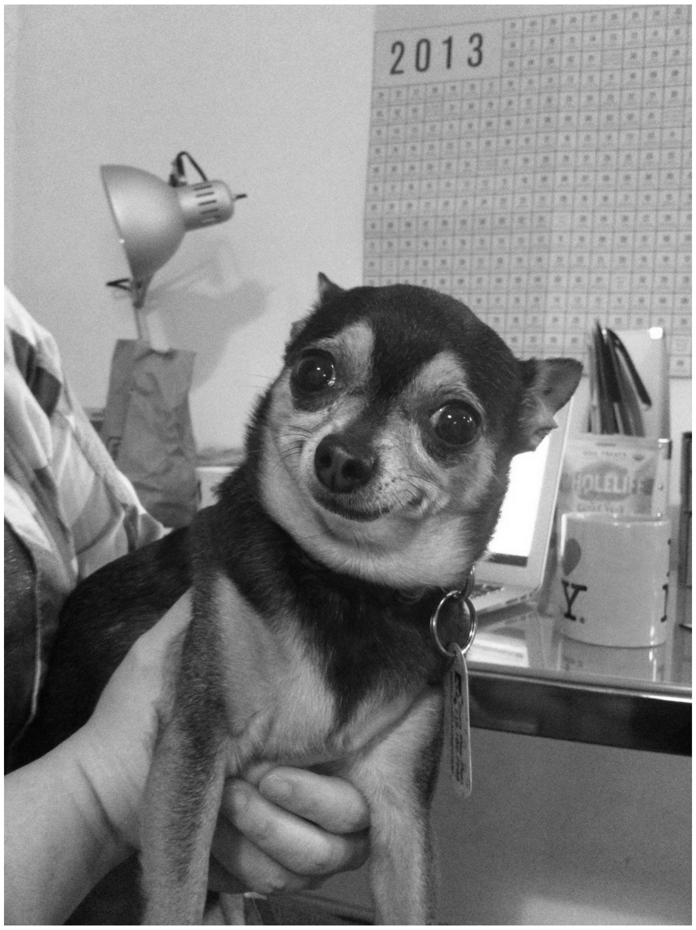
October 2024

1 CPU version

```
    from PIL import Image
import time
from IPython.display import display

# Start the timer
start_time = time.time()

# Load the image
img = Image.open('/content/tumblr_mhwsoaQpXT1rttifoo1_1280.jpg')

rgb_array = np.array(img)
# Create an empty array for the grayscale image
gray_array = np.zeros((rgb_array.shape[0], rgb_array.shape[1]), dtype=np.uint8)

# # Convert the image to grayscale manually using average formula
width, height = gray_array.shape[0], gray_array.shape[1]

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]
        gray_array[x, y] = int((float(r) + float(g) + float(b)) / 3)


# End the timer
end_time = time.time()
print(f"Time taken: {end_time - start_time} seconds")

# Display the grayscale image using IPython
display(gray_img)
```

In short, my code load an color image, initialized a zero array corresponding to its grey version.

Then sequentially, i iterate through all the pixel to average the 3 rgb value to get the grey scale value. I measure the time it took for the operation and display the grey image



It took 2.02103328704834 seconds

2 GPU version

```
    from PIL import Image
import time
from IPython.display import display
from numba import cuda
import numpy as np
```

```
# Function to convert the image to grayscale on the GPU
'''
3. Device processes data in parallel (Parallel Execution)
'''
@cuda.jit
def rgb_to_gray_gpu(rgb_array, gray_array):
    x, y = cuda.grid(2)
    if x < rgb_array.shape[0] and y < rgb_array.shape[1]:
        r = rgb_array[x, y, 0]
        g = rgb_array[x, y, 1]
        b = rgb_array[x, y, 2]
        # Average formula for grayscale
        gray_value = (r + g + b) // 3
        gray_array[x, y] = gray_value

# Start the timer
start_time = time.time()

# Load the image and convert to NumPy array
img = Image.open('/content/tumblr_mhwsoaQpXT1rttifoo1_1280.jpg')
rgb_array = np.array(img)

# Create an empty array for the grayscale image
gray_array = np.zeros((rgb_array.shape[0], rgb_array.shape[1]), dtype=np.uint8)

# Define the block and grid size for GPU execution
threads_per_block = (16, 16)
blocks_per_grid_x = int(np.ceil(rgb_array.shape[0] / threads_per_block[0]))
blocks_per_grid_y = int(np.ceil(rgb_array.shape[1] / threads_per_block[1]))
blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

'''
1. Host feeds device with data (Data Transfer from Host to Device)
'''
# Copy data to the GPU
rgb_array_device = cuda.to_device(rgb_array)
gray_array_device = cuda.to_device(gray_array)

# Run the kernel on the GPU
'''
2. Host asks device to process data (Kernel Launch)
'''
rgb_to_gray_gpu[blocks_per_grid, threads_per_block](rgb_array_device, gray_array_device)

'''
4. Device returns result (Data Transfer from Device to Host)
```

```
'''
# Copy the result back to the host
gray_array_device.copy_to_host(gray_array)

# End the timer
end_time = time.time()
print(f"Time taken: {end_time - start_time} seconds")

# Convert the grayscale array back to an image
gray_img = Image.fromarray(gray_array)

# Display the grayscale image using IPython
display(gray_img)
```

In this code a load the image. I then define a CUDA kernel function, which computes the grayscale values for each pixel by averaging the red, green, and blue components of the pixel colors. The script begins by loading an image and converting it to a NumPy array. It then allocates an empty array for the grayscale image and specifies the grid and block sizes for GPU execution. Data is transferred from the host (CPU) to the device (GPU) before launching the kernel for processing. After the grayscale conversion is complete, the result is copied back to the host, and the time taken for the entire operation is printed.

This is the resulting image from the code. It took 0.15901541709899902 seconds to run

3 Compare different block size

We use the following code to compare. Here we varies different threads per block, which result in corresponding bigger or smaller block to execute.

```
    from PIL import Image
import time
from IPython.display import display
from numba import cuda
import numpy as np

# Function to convert the image to grayscale on the GPU
```

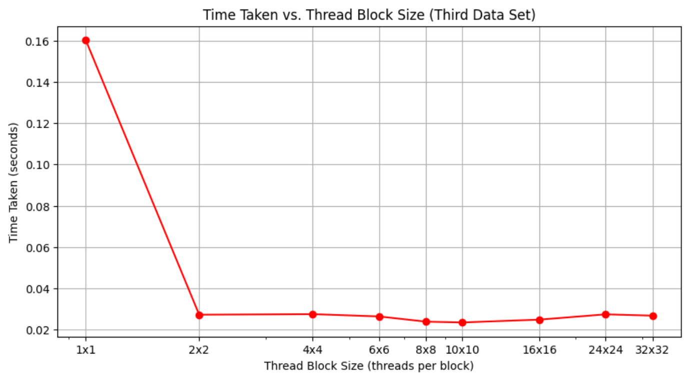
```
'''
3. Device processes data in parallel (Parallel Execution)
'''
@cuda.jit
def rgb_to_gray_gpu(rgb_array, gray_array):
    x, y = cuda.grid(2)
    if x < rgb_array.shape[0] and y < rgb_array.shape[1]:
        r = rgb_array[x, y, 0]
        g = rgb_array[x, y, 1]
        b = rgb_array[x, y, 2]
        # Average formula for grayscale
        gray_value = (r + g + b) // 3
        gray_array[x, y] = gray_value

for i in [(1,1),(2,2),(4,4),(6,6),(8,8),(10,10),(16, 16), (24, 24), (32, 32)]:
  # Start the timer
  start_time = time.time()

  # Load the image and convert to NumPy array
  img = Image.open('/content/tumblr_mhwsoaQpXT1rttifoo1_1280.jpg')
  rgb_array = np.array(img)

  # Create an empty array for the grayscale image
  gray_array = np.zeros((rgb_array.shape[0], rgb_array.shape[1]), dtype=np.uint8)

  # Define the block and grid size for GPU execution
  threads_per_block = i
  blocks_per_grid_x = int(np.ceil(rgb_array.shape[0] / threads_per_block[0]))
  blocks_per_grid_y = int(np.ceil(rgb_array.shape[1] / threads_per_block[1]))
  blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

  '''
  1. Host feeds device with data (Data Transfer from Host to Device)
  '''
  # Copy data to the GPU
  rgb_array_device = cuda.to_device(rgb_array)
  gray_array_device = cuda.to_device(gray_array)

  # Run the kernel on the GPU
  '''
  2. Host asks device to process data (Kernel Launch)
  '''
  rgb_to_gray_gpu[blocks_per_grid, threads_per_block](rgb_array_device, gray_array_device)

  '''
  4. Device returns result (Data Transfer from Device to Host)
```

```
'''
# Copy the result back to the host
gray_array_device.copy_to_host(gray_array)

# End the timer
end_time = time.time()
print(f"Time taken for {threads_per_block} thread per block: {end_time - start_time} secon
```

With the thread_block_sizes in the set of [(1, 1), (2, 2), (4, 4), (6, 6), (8, 8), (10, 10), (16, 16), (24, 24), (32, 32)], we get the following time of execution:

```
    Time taken for (1, 1) thread per block: 0.16016292572021484 seconds
Time taken for (2, 2) thread per block: 0.027263641357421875 seconds
Time taken for (4, 4) thread per block: 0.027539730072021484 seconds
Time taken for (6, 6) thread per block: 0.02643442153930664 seconds
Time taken for (8, 8) thread per block: 0.02390313148498535 seconds
Time taken for (10, 10) thread per block: 0.02353048324584961 seconds
Time taken for (16, 16) thread per block: 0.024882078170776367 seconds
Time taken for (24, 24) thread per block: 0.027444124221801758 seconds
Time taken for (32, 32) thread per block: 0.026804208755493164 seconds
```

This correspond to this graph:



We see that having more than 1 threads does result in improvement in performance.

We can see that for this particular small image with the dimension 968x1296, increasing the number of threads (reducing block size) passing a certain threshold will not result in improvement in performance.