HPC-Assignment-18

April 25, 2024

1 HPC Assignment 18

1.1 Image Processing CUDA

1.2 Implement the following Image Processing operations in sequential and parallel using CUDA Programming.

1.2.1 Gaussian blur

Describe Gaussian Blur in brief. Gaussian blur is a popular image processing technique used to minimize visual noise and detail, resulting in a smoother look. Gaussian blur convolves each pixel in the image with a Gaussian kernel, a 2D matrix that represents a bell-shaped curve. The size and standard deviation (sigma) of the Gaussian kernel control the amount of blurring applied to the image. A larger kernel size and a higher sigma value cause more noticeable blurring. During convolution, each pixel in the image is replaced by a weighted average of its neighbors, the weights of which are defined by the Gaussian function. This method efficiently decreases the image's high-frequency components, smoothing out sharp transitions and noise.

Where parallelism can be inserted? Parallelism can be inserted in Gaussian blur at key stages:

1. image Copy and Memory Allocation: Transfer input picture data to GPU while allocating memory for the output image in parallel.
2. Convolution: Perform the convolution process with CUDA kernels, with each thread computing the weighted sum for a given output pixel.
3. Border Handling: Process border regions concurrently, allocating various threads to handle different parts of the border.
4. image Copy Back: Transfer processed picture data from the GPU to the host RAM in parallel.

Analyze the performance in serial and parallel model.

Serial

[28]:
```python
import os
import cv2
import numpy as np
import time
```

[29]:

```
def gaussian_kernel(size, sigma=1):
    kernel = np.fromfunction(lambda x, y: (1/(2*np.pi*sigma**2)) * np.
 ↪exp(-((x-(size-1)/2)**2 + (y-(size-1)/2)**2)/(2*sigma**2)), (size, size))
    return kernel / np.sum(kernel)
```

[30]:
```
def gaussian_blur_sequential(image, kernel_size):
    kernel = gaussian_kernel(kernel_size)
    blurred_image = cv2.filter2D(image, -1, kernel)  # Using OpenCV's filter2D␣
 ↪for convolution
    return blurred_image
```

[31]:
```
def process_images_in_folder(folder_path, kernel_size):
    times = {}
    for filename in os.listdir(folder_path):
        if filename.endswith(".jpg") or filename.endswith(".png"):
            image_path = os.path.join(folder_path, filename)
            image = cv2.imread(image_path, cv2.IMREAD_COLOR)
            start_time = time.time()
            blurred_image = gaussian_blur_sequential(image, kernel_size)
            end_time = time.time()
            times[filename] = end_time - start_time
    return times
```

[32]:
```
folder_path = "/content/drive/MyDrive/data_set/minion"
kernel_size = 3
```

[33]:
```
start_time_total = time.time()
times_per_image = process_images_in_folder(folder_path, kernel_size)
end_time_total = time.time()
total_time = end_time_total - start_time_total
```

[34]:
```
print("Time taken for Gaussian blur on each image:")
for filename, time_taken in times_per_image.items():
    print(f"{filename}: {time_taken:.4f} seconds")
```

```
Time taken for Gaussian blur on each image:
1.jpg: 0.0348 seconds
2.jpg: 0.0024 seconds
3.jpg: 0.0618 seconds
4.jpg: 0.0038 seconds
5.jpg: 0.0173 seconds
6.jpg: 0.0064 seconds
7.jpg: 0.0085 seconds
8.jpg: 0.0044 seconds
9.jpg: 0.0034 seconds
10.jpg: 0.0033 seconds
11.jpg: 0.0027 seconds
```

```
[35]: print(f"\nTotal time taken for all images: {total_time:.4f} seconds")
```

```
Total time taken for all images: 0.6478 seconds
```

Parallel

```
[36]: import cupy as cp
from PIL import Image
import os
import time

def process_image(image_array):
    def gaussian_kernel(size, sigma=1):
        kernel_1D = cp.linspace(-(size // 2), size // 2, size)
        for i in range(size):
            kernel_1D[i] = cp.exp(-0.5 * (kernel_1D[i] / sigma) ** 2)
        kernel_2D = cp.outer(kernel_1D, kernel_1D)
        kernel_2D /= kernel_2D.sum()
        return kernel_2D

    kernel_size = 5
    gaussian_kernel_array = gaussian_kernel(kernel_size)
    blurred_image = cp.asarray(Image.fromarray(cp.asnumpy(image_array)).
 ↪convert("L"))

    return blurred_image

directory = "/content/drive/MyDrive/data_set/minion"
num_images = 0
start_time = time.time()

image_paths = [os.path.join(directory, filename) for filename in os.
 ↪listdir(directory) if filename.endswith(".jpg")]
image_arrays = [cp.array(Image.open(image_path).convert("L")) for image_path in␣
 ↪image_paths]
processed_images = [process_image(image_array) for image_array in image_arrays]

cp.cuda.Device().synchronize()

total_time_parallel = time.time() - start_time
num_images = len(image_paths)

print("Number of images processed in parallel:", num_images)
print("Time taken for parallel processing:", total_time_parallel, "seconds")
```

```
Number of images processed in parallel: 11
Time taken for parallel processing: 0.41269755363464355 seconds
```

1.2.2 FFT- Fast Fourier Transform

Describe FFT in brief The FFT algorithm decomposes the DFT computation into smaller subproblems, exploiting the periodicity and symmetry properties of complex exponential functions. By iteratively applying these subproblems and integrating the answers, the FFT computes the DFT in much less operations than direct computation.

The FFT output represents the input signal's frequency spectrum, including the magnitude and phase of each frequency component. This information is useful for filtering, noise reduction, spectrum analysis, and feature extraction.

Where parallelism can be inserted? Parallelism in the Fast Fourier Transform (FFT) can be introduced at numerous stages:

1. Data Splitting: Split incoming data into chunks that can be processed independently by several cores or thread
2. Butterfly Operations: Parallelize butterfly operations, particularly on SIMD architectures such as GUs.
3. Twiddle Factor Calculation: Calculate twiddle factors simultaneously across several processor nits.
4. Memory Access: Improve memory access patterns for parallel processing, such as coalesced memory access and preftching.
5. Inverse FFT (IFFT): Use parallelization techniques to improve inverse FFT computation permance.

Analyze the performance in serial and parallel model.

serial

[37]:
```python
def fft_sequential(image):
    fft_image = np.fft.fft2(image)
    return fft_image
```

[38]:
```python
def process_images_in_folder_fft(folder_path):
    times = {}
    for filename in os.listdir(folder_path):
        if filename.endswith(".jpg") or filename.endswith(".png"):
            image_path = os.path.join(folder_path, filename)
            image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
            start_time = time.time()
            fft_image = fft_sequential(image)
            end_time = time.time()
            times[filename] = end_time - start_time
    return times
```

[39]:
```python
folder_path = "/content/drive/MyDrive/data_set/minion"
```

```
[40]: start_time_total = time.time()
      times_per_image_fft = process_images_in_folder_fft(folder_path)
      end_time_total = time.time()
      total_time_fft = end_time_total - start_time_total
```

```
[41]: print("Time taken for FFT on each image:")
      for filename, time_taken in times_per_image_fft.items():
          print(f"{filename}: {time_taken:.4f} seconds")
```

```
Time taken for FFT on each image:
1.jpg: 0.0262 seconds
2.jpg: 0.0302 seconds
3.jpg: 1.3454 seconds
4.jpg: 0.0321 seconds
5.jpg: 0.2315 seconds
6.jpg: 0.0642 seconds
7.jpg: 0.0505 seconds
8.jpg: 0.0690 seconds
9.jpg: 0.0231 seconds
10.jpg: 0.0405 seconds
11.jpg: 0.0352 seconds
```

```
[42]: print(f"\nTotal time taken for FFT on all images: {total_time_fft:.4f} seconds")
```

```

Total time taken for FFT on all images: 2.2286 seconds
```

Parallel

```
[43]: import os
      import time
      import cupy as cp
      from PIL import Image

      directory = "/content/drive/MyDrive/data_set/minion"

      num_images = 0
      start_time = time.time()

      def process_image(image_array):
          global num_images
          num_images += 1

          fft_image = cp.fft.fft2(image_array)
          fft_image_shifted = cp.fft.fftshift(fft_image)

          rows, cols = image_array.shape
          center_row, center_col = rows // 2, cols // 2
          radius = 20
```

```
    high_pass_filter = cp.ones((rows, cols))
    mask = cp.zeros((rows, cols))
    mask[center_row - radius:center_row + radius, center_col - radius:
 ↪center_col + radius] = 1
    high_pass_filter -= mask

    filtered_image_fft = fft_image_shifted * high_pass_filter

    filtered_image = cp.abs(cp.fft.ifft2(cp.fft.ifftshift(filtered_image_fft)))
    return filtered_image

image_paths = [os.path.join(directory, filename) for filename in os.
 ↪listdir(directory) if filename.endswith(".jpg")]
image_arrays = [cp.array(Image.open(image_path).convert("L")) for image_path in␣
 ↪image_paths]
processed_images = [process_image(image_array) for image_array in image_arrays]

cp.cuda.Device().synchronize()

total_time_parallel = time.time() - start_time

print("Number of images processed in parallel:", num_images)
print("Time taken for parallel processing:", total_time_parallel, "seconds")
```

```
Number of images processed in parallel: 11
Time taken for parallel processing: 0.7189762592315674 seconds
```