LAB ASSIGNMENT # 1

Name: Muhammad Tayyab

Roll No: SP23-BCS-099

Section: C
Course: PDC

Teacher: Sir Akhzar Nazir

Part 1:
Write a simple CUDA kernel that prints:
Hello from thread X
* *Understand how GPU threads, blocks, and grids work by experimenting with different*
launch configurations.

Solution:

```
test_code = r"""
#include <stdio.h>
__global__ void helloWorld(){
  printf("Hello from the other side %d\n",threadIdx.x);
}
int main(){
  helloWorld<<<1,5>>>();
  cudaDeviceSynchronize();
  return 0;
}
"""

with open("hello.cu","w") as f:
  f.write(test_code)
!nvcc -arch=sm_75 -o hello hello.cu
!./hello
```

Part 2:

Implement vector addition of two large arrays (e.g., 10 million elements):

o First on CPU (normal C++ loop).

o Then on GPU (CUDA kernel).

* Measure the execution time of both.*

* Calculate the speedup ratio:*

Solution:

```
cuda_code = r"""
// Paste your complete CUDA C++ code here
#include <iostream>
#include <chrono>
#include <cmath>
#include <cuda_runtime.h>

__global__ void vectorAdd(const float* A, const float* B, float* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        C[i] = A[i] + B[i];
    }
}
```

```
void vectorAddCPU(const float* A, const float* B, float* C, int N) {
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    const int N = 10 * 1000 * 1000;
    size_t size = N * sizeof(float);
    float* h_A = new float[N];
    float* h_B = new float[N];
    float* h_C_cpu = new float[N];
    float* h_C_gpu = new float[N];

    for (int i = 0; i < N; i++) {
        h_A[i] = i * 1.0f;
        h_B[i] = (N - i) * 1.0f;
    }

    auto start_cpu = std::chrono::high_resolution_clock::now();
    vectorAddCPU(h_A, h_B, h_C_cpu, N);
    auto end_cpu = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float, std::milli> cpu_duration = end_cpu - start_cpu;
    std::cout << "CPU Vector addition time: " << cpu_duration.count() << "
ms\n";

    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    cudaEventRecord(start);
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    float gpu_milliseconds = 0;
    cudaEventElapsedTime(&gpu_milliseconds, start, stop);
    std::cout << "GPU Vector addition time: " << gpu_milliseconds << " ms\n";

    cudaMemcpy(h_C_gpu, d_C, size, cudaMemcpyDeviceToHost);

    bool success = true;
    for (int i = 0; i < N; i++) {
        if (fabs(h_C_cpu[i] - h_C_gpu[i]) > 1e-5) {
            std::cout << "Mismatch at index " << i << ": CPU " << h_C_cpu[i] << ",
GPU " << h_C_gpu[i] << "\n";
            success = false;
            break;
        }
    }
    if (success) std::cout << "Results match!\n";
    else std::cout << "Results do not match!\n";

    float speedup = cpu_duration.count() / gpu_milliseconds;
    std::cout << "Speedup (CPU time / GPU time): " << speedup << "x\n";

    delete[] h_A;
    delete[] h_B;
    delete[] h_C_cpu;
    delete[] h_C_gpu;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
"""

with open("vector_add.cu", "w") as f:
    f.write(cuda_code)
!nvcc -o vector_add vector_add.cu

!./vector_add
```

Output:

```
CPU Vector addition time: 48.3885 ms
GPU Vector addition time: 7.8135 ms
```

```
Mismatch at index 0: CPU 1e+07, GPU 0
Results do not match!
Speedup (CPU time / GPU time): 6.19294x
```

Part 3:

Load an image (e.g., PNG or JPG).
□ Implement pixel inversion:
new_pixel=255−old_pixel\text{new_pixel} = 255 - \text{old_pixel}new_pixel=255−old_pixel
□ Do it once using a CPU loop, and again using a CUDA kernel.
□ Compare performance and verify that the output images are identical.

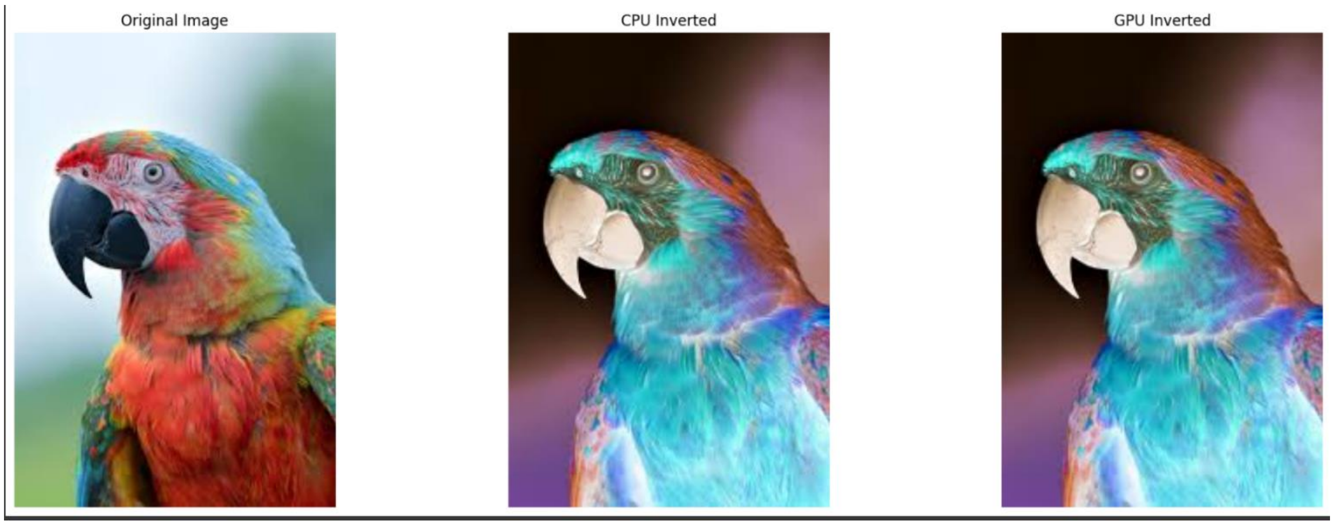
Solution:

```
from PIL import Image
import numpy as np
import time
import cupy as cp
import matplotlib.pyplot as plt
import os
try:
    from google.colab import files
    uploaded = files.upload()  # Let user upload any image
    input_image_path = list(uploaded.keys())[0]  # Automatically get uploaded
filename
except ImportError:
    input_image_path = 'parrot_image.jfif'  # Change this to your file name if
running locally

try:
    pil_image = Image.open(input_image_path).convert('RGB')
    image_data = np.asarray(pil_image)
    assert image_data.ndim == 3 and image_data.shape[2] == 3, "Expected
RGB image"
    print(f"Image '{input_image_path}' loaded successfully.")
    print(f"Image shape: {image_data.shape}, dtype: {image_data.dtype}")
except FileNotFoundError:
    print(f"Error: File '{input_image_path}' not found.")
    exit()
except Exception as e:
    print(f"Error loading image: {e}")
    exit()

print("\nRunning CPU inversion using NumPy...")
start_time_cpu = time.perf_counter()
inverted_cpu = 255 - image_data
end_time_cpu = time.perf_counter()
time_cpu = end_time_cpu - start_time_cpu
print(f"CPU inversion time: {time_cpu:.6f} seconds")

print("\nRunning GPU inversion using CuPy...")
cp.asarray(np.zeros((10, 10)))  # Warm up GPU

image_data_gpu = cp.asarray(image_data)
start_time_gpu = time.perf_counter()
inverted_gpu = 255 - image_data_gpu
```

```
cp.cuda.Stream.null.synchronize()
end_time_gpu = time.perf_counter()
time_gpu = end_time_gpu - start_time_gpu
print(f"GPU inversion time: {time_gpu:.6f} seconds")

inverted_gpu_host = cp.asnumpy(inverted_gpu)

print("\n--- Result Comparison ---")
is_identical = np.allclose(inverted_cpu, inverted_gpu_host, atol=1)
print(f"CPU and GPU results identical: {is_identical}")
speedup = time_cpu / time_gpu if time_gpu > 0 else float('inf')
print(f"GPU Speedup over CPU: {speedup:.2f}x")

from PIL import Image
Image.fromarray(inverted_cpu).save("inverted_cpu.png")
Image.fromarray(inverted_gpu_host).save("inverted_gpu.png")
print("Saved inverted images.")

fig, axes = plt.subplots(1, 3, figsize=(18, 6))
axes[0].imshow(image_data)
axes[0].set_title("Original Image")
axes[0].axis('off')

axes[1].imshow(inverted_cpu)
axes[1].set_title("CPU Inverted")
axes[1].axis('off')

axes[2].imshow(inverted_gpu_host)
axes[2].set_title("GPU Inverted")
axes[2].axis('off')
```

```
plt.tight_layout()
plt.show()
```

Output:


Original Image
CPU Inverted
GPU Inverted