

**LAB Assignment 1**

|  |  |
| --- | --- |
| **Name:** | **Noor Fatima** |
| **Registration No:** | **Sp23-bcs-109** |
| **Course:** | **Parallel and distributed computing** |
| **Teacher:** | **Sir Akhzar Nazir** |
| **Date:** | **25-9-25** |
| **Section :** | **C** |

**Part 1: Hello GPU with CUDA**

%%writefile hello.cu

#include <stdio.h>

\_\_global\_\_ void helloFromGPU() {

int threadId = threadIdx.x;

int blockId = blockIdx.x;

int globalId = blockIdx.x \* blockDim.x + threadIdx.x;

printf("Hello from block %d, thread %d (global %d)\n", blockId, threadId, globalId);

}

int main() {

// Launch kernel with 2 blocks, 4 threads per block

helloFromGPU<<<2, 4>>>();

cudaDeviceSynchronize();

return 0;

}

**Part 2: Vector Addition (CPU vs GPU):**

%%writefile vector\_add.cu

#include <stdio.h>

#include <cuda.h>

#include <chrono>

#include <iostream>

using namespace std;

\_\_global\_\_ void vectorAdd(float \*A, float \*B, float \*C, int n) {

int i = blockIdx.x \* blockDim.x + threadIdx.x;

if (i < n) {

C[i] = A[i] + B[i];

}

}

int main() {

int n = 10000000; // 10 million

size\_t size = n \* sizeof(float);

// Allocate host memory

float \*h\_A = new float[n];

float \*h\_B = new float[n];

float \*h\_C = new float[n];

for (int i = 0; i < n; i++) {

h\_A[i] = 1.0f;

h\_B[i] = 2.0f;

}

// ---------------- CPU Vector Add ----------------

auto start\_cpu = chrono::high\_resolution\_clock::now();

for (int i = 0; i < n; i++) {

h\_C[i] = h\_A[i] + h\_B[i];

}

auto end\_cpu = chrono::high\_resolution\_clock::now();

double cpu\_time = chrono::duration<double>(end\_cpu - start\_cpu).count();

// ---------------- GPU Vector Add ----------------

float \*d\_A, \*d\_B, \*d\_C;

cudaMalloc((void\*\*)&d\_A, size);

cudaMalloc((void\*\*)&d\_B, size);

cudaMalloc((void\*\*)&d\_C, size);

cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

int threadsPerBlock = 256;

int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

auto start\_gpu = chrono::high\_resolution\_clock::now();

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, n);

cudaDeviceSynchronize();

auto end\_gpu = chrono::high\_resolution\_clock::now();

double gpu\_time = chrono::duration<double>(end\_gpu - start\_gpu).count();

cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);

// ---------------- Results ----------------

printf("CPU time = %f sec\n", cpu\_time);

printf("GPU time = %f sec\n", gpu\_time);

printf("Speedup = %f\n", cpu\_time / gpu\_time);

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C);

delete[] h\_A; delete[] h\_B; delete[] h\_C;

return 0;

}A black rectangular object with white text

AI-generated content may be incorrect.

**Part 3: Image Inversion (CPU vs GPU)**

%%writefile image\_invert.cu

#include <opencv2/opencv.hpp>

#include <iostream>

#include <cuda\_runtime.h>

using namespace cv;

using namespace std;

\_\_global\_\_ void invertImage(unsigned char\* input, unsigned char\* output, int width, int height, int channels) {

int x = blockIdx.x \* blockDim.x + threadIdx.x;

int y = blockIdx.y \* blockDim.y + threadIdx.y;

if (x < width && y < height) {

int idx = (y \* width + x) \* channels;

for (int c = 0; c < channels; c++) {

output[idx + c] = 255 - input[idx + c];

}

}

}

int main() {

Mat input = imread("sample.jpg");

if (input.empty()) {

cout << "Error: Could not open image!" << endl;

return -1;

}

int width = input.cols;

int height = input.rows;

int channels = input.channels();

int img\_size = width \* height \* channels;

unsigned char\* h\_input = input.data;

unsigned char\* h\_output = new unsigned char[img\_size];

// CPU inversion

for (int i = 0; i < img\_size; i++) {

h\_output[i] = 255 - h\_input[i];

}

Mat cpu\_img(height, width, CV\_8UC3, h\_output);

imwrite("output\_cpu.jpg", cpu\_img.clone());

// GPU inversion

unsigned char \*d\_input, \*d\_output;

cudaMalloc((void\*\*)&d\_input, img\_size);

cudaMalloc((void\*\*)&d\_output, img\_size);

cudaMemcpy(d\_input, h\_input, img\_size, cudaMemcpyHostToDevice);

dim3 block(16, 16);

dim3 grid((width + block.x - 1) / block.x, (height + block.y - 1) / block.y);

invertImage<<<grid, block>>>(d\_input, d\_output, width, height, channels);

cudaDeviceSynchronize();

cudaMemcpy(h\_output, d\_output, img\_size, cudaMemcpyDeviceToHost);

Mat gpu\_img(height, width, CV\_8UC3, h\_output);

imwrite("output\_gpu.jpg", gpu\_img.clone());

cudaFree(d\_input);

cudaFree(d\_output);

delete[] h\_output;

cout << "Done: output\_cpu.jpg and output\_gpu.jpg generated" << endl;

return 0;

}

**OUTPUT:**

****