

assignment 1

Name: Urwah Muhammad

Roll No.: SP23-BCS-123

Section: SP23-BCS-C

Subject: Parallel and Distributed Computing

Submitted to: Sir Akhzar Nazir



**Assignment 1**

# Part 1: Hello GPU with CUDA

#include <iostream>

#include <cuda\_runtime.h>

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

int threadId = threadIdx.x; // Thread index within block

int blockId = blockIdx.x; // Block index within grid

int globalId = blockId \* blockDim.x + threadId; // Unique global thread ID

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

}

int main() {

// Launch kernel with 2 blocks, 4 threads each

int numBlocks = 2;

int threadsPerBlock = 4;

helloFromGPU << <numBlocks, threadsPerBlock >> > ();

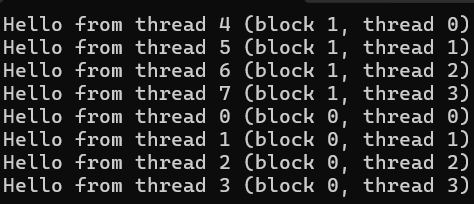
// Wait for GPU to finish before accessing results

cudaDeviceSynchronize();

return 0;

}

## Output



# Part 2: Vector Addition (CPU vs GPU)

#include <iostream>

#include <cuda\_runtime.h>

#include <chrono> // For CPU timing

#define N 10000000 // 10 million elements

// ---------------- CUDA Kernel ----------------

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

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

if (idx < n) {

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

}

}

// ---------------- Main Program ----------------

int main() {

// Allocate host memory

float\* h\_A, \* h\_B, \* h\_C\_cpu, \* h\_C\_gpu;

h\_A = new float[N];

h\_B = new float[N];

h\_C\_cpu = new float[N];

h\_C\_gpu = new float[N];

// Initialize input vectors

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

h\_A[i] = 1.0f;

h\_B[i] = 2.0f;

}

// ---------- CPU Vector Addition ----------

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

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

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

}

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

std::chrono::duration<double, std::milli> cpu\_time = end\_cpu - start\_cpu;

// ---------- Allocate GPU memory ----------

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

cudaMalloc((void\*\*)&d\_A, N \* sizeof(float));

cudaMalloc((void\*\*)&d\_B, N \* sizeof(float));

cudaMalloc((void\*\*)&d\_C, N \* sizeof(float));

// Copy data from host to device

cudaMemcpy(d\_A, h\_A, N \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, N \* sizeof(float), cudaMemcpyHostToDevice);

// ---------- GPU Vector Addition ----------

int threadsPerBlock = 256;

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

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

vectorAddGPU << <blocksPerGrid, threadsPerBlock >> > (d\_A, d\_B, d\_C, N);

cudaEventRecord(stop);

cudaMemcpy(h\_C\_gpu, d\_C, N \* sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);

float gpu\_time = 0;

cudaEventElapsedTime(&gpu\_time, start, stop);

// ---------- Verify Results ----------

bool correct = true;

for (int i = 0; i < 10; i++) { // check only first 10

if (h\_C\_cpu[i] != h\_C\_gpu[i]) {

correct = false;

break;

}

}

// ---------- Print Results ----------

std::cout << "CPU Time: " << cpu\_time.count() << " ms\n";

std::cout << "GPU Time: " << gpu\_time << " ms\n";

std::cout << "Speedup = " << cpu\_time.count() / gpu\_time << "x\n";

std::cout << "Result check: " << (correct ? "PASS " : "FAIL ") << "\n";

// Free memory

delete[] h\_A;

delete[] h\_B;

delete[] h\_C\_cpu;

delete[] h\_C\_gpu;

cudaFree(d\_A);

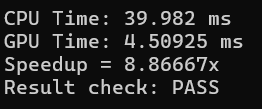
cudaFree(d\_B);

cudaFree(d\_C);

return 0;

}

## Output



# Part 3: Image Inversion (CPU vs GPU)

#include <iostream>

#include <opencv2/opencv.hpp>

#include <cuda\_runtime.h>

#include <chrono>

// ---------------- CUDA Kernel ----------------

\_\_global\_\_ void invertImage1D(unsigned char\* d\_in, unsigned char\* d\_out, int totalPixels, int channels) {

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

if (idx < totalPixels) {

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

d\_out[idx \* channels + c] = 255 - d\_in[idx \* channels + c];

}

}

}

int main() {

// ---------------- Load Image ----------------

cv::Mat input = cv::imread("input.jpg"); // <-- change filename

if (input.empty()) {

std::cerr << "Error: Could not open image!\n";

return -1;

}

int width = input.cols;

int height = input.rows;

int channels = input.channels();

int totalPixels = width \* height;

size\_t imgSize = totalPixels \* channels \* sizeof(unsigned char);

// ---------------- CPU Inversion ----------------

cv::Mat output\_cpu = input.clone();

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

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

cv::Vec3b& pixel = output\_cpu.at<cv::Vec3b>(y, x);

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

pixel[c] = 255 - pixel[c];

}

}

}

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

std::chrono::duration<double, std::milli> cpu\_time = end\_cpu - start\_cpu;

// ---------------- GPU Inversion ----------------

unsigned char\* d\_in, \* d\_out;

cudaMalloc((void\*\*)&d\_in, imgSize);

cudaMalloc((void\*\*)&d\_out, imgSize);

cudaMemcpy(d\_in, input.data, imgSize, cudaMemcpyHostToDevice);

int threadsPerBlock = 256;

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

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

invertImage1D << <blocksPerGrid, threadsPerBlock >> > (d\_in, d\_out, totalPixels, channels);

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float gpu\_time = 0;

cudaEventElapsedTime(&gpu\_time, start, stop);

cv::Mat output\_gpu(height, width, input.type());

cudaMemcpy(output\_gpu.data, d\_out, imgSize, cudaMemcpyDeviceToHost);

// ---------------- Verify Results (Fixed) ----------------

bool identical = (cv::norm(output\_cpu, output\_gpu, cv::NORM\_L2) == 0);

// ---------------- Save Results ----------------

cv::imwrite("output\_cpu.jpg", output\_cpu);

cv::imwrite("output\_gpu.jpg", output\_gpu);

// ---------------- Print Times ----------------

std::cout << "CPU Time: " << cpu\_time.count() << " ms\n";

std::cout << "GPU Time: " << gpu\_time << " ms\n";

std::cout << "Speedup = " << cpu\_time.count() / gpu\_time << "x\n";

std::cout << "Images identical? " << (identical ? "YES " : "NO ") << "\n";

// ---------------- Cleanup ----------------

cudaFree(d\_in);

cudaFree(d\_out);

return 0;

}

## Output

