Name: Sakshi Dhamapurkar

PRN No: 21510073

**High Performance Computing Lab**

**Practical No. 12**

**Title of practical: Parallel Programming using of CUDA C**

**Problem 1: Vector Addition using CUDA**

Problem Statement: Write a CUDA C program that performs element-wise addition of two vectors A and B of size N. The result of the addition should be stored in vector C.

Details:

* Initialize the vectors A and B with random numbers.
* The output vector C[i] = A[i] + B[i], where i ranges from 0 to N-1.
* Use CUDA kernels to perform the computation in parallel.
* Write the code for both serial (CPU-based) and parallel (CUDA-based) implementations.
* Measure the execution time of both the serial and CUDA implementations for different values of N (e.g., N = 10^5, 10^6, 10^7).

Task:

* Calculate and report the speedup (i.e., the ratio of CPU execution time to GPU execution time).

CODE :

%%writefile vector\_addition.cu

#include <iostream>

#include <cuda.h>

#include <cstdlib>

#include <chrono>

#define N 1000000

void initialize\_vector(float \*vec, int size) {

    for (int i = 0; i < size; i++) {

        vec[i] = static\_cast<float>(rand()) / RAND\_MAX;

    }

}

// CPU-based vector addition

void vector\_add\_cpu(float \*A, float \*B, float \*C, int size) {

    for (int i = 0; i < size; i++) {

        C[i] = A[i] + B[i];

    }

}

// CUDA kernel for vector addition

\_\_global\_\_ void vector\_add\_cuda(float \*A, float \*B, float \*C, int size) {

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

    if (idx < size) {

        C[idx] = A[idx] + B[idx];

    }

}

int main() {

    float \*A, \*B, \*C;         // Host vectors

    float \*d\_A, \*d\_B, \*d\_C;   // Device vectors

    // Allocate memory on host

    A = new float[N];

    B = new float[N];

    C = new float[N];

    // Initialize vectors

    initialize\_vector(A, N);

    initialize\_vector(B, N);

    // CPU execution time

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

    vector\_add\_cpu(A, B, C, N);

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

    float cpu\_time = std::chrono::duration<float, std::milli>(end\_cpu - start\_cpu).count();

    std::cout << "CPU Time: " << cpu\_time << " ms" << std::endl;

    // Allocate memory on device

    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, A, N \* sizeof(float), cudaMemcpyHostToDevice);

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

    // Launch CUDA kernel

    int blockSize = 256;

    int numBlocks = (N + blockSize - 1) / blockSize;

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

    vector\_add\_cuda<<<numBlocks, blockSize>>>(d\_A, d\_B, d\_C, N);

    cudaDeviceSynchronize();

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

    float gpu\_time = std::chrono::duration<float, std::milli>(end\_gpu - start\_gpu).count();

    // Copy result back to host

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

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

    std::cout << "Speedup: " << cpu\_time / gpu\_time << std::endl;

    // Free memory

    cudaFree(d\_A);

    cudaFree(d\_B);

    cudaFree(d\_C);

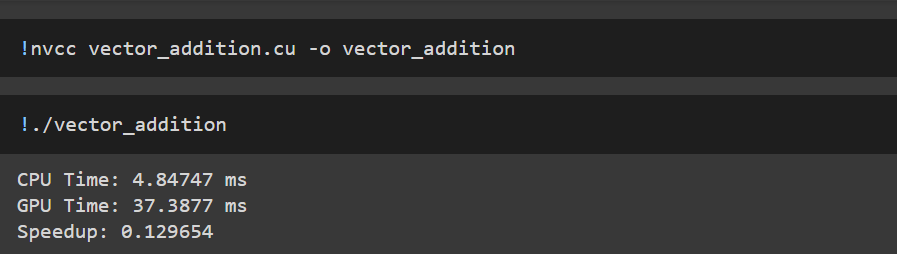
    delete[] A;

    delete[] B;

    delete[] C;

    return 0;

}



**Problem 2: Matrix Addition using CUDA**

Problem Statement: Write a CUDA C program to perform element-wise addition of two matrices A and B of size M x N. The result of the addition should be stored in matrix C.

Details:

* Initialize the matrices A and B with random values.
* The output matrix C[i][j] = A[i][j] + B[i][j] where i ranges from 0 to M-1 and j ranges from 0 to N-1.
* Write code for both serial (CPU-based) and parallel (CUDA-based) implementations.
* Measure the execution time of both implementations for various matrix sizes (e.g., 100x100, 500x500, 1000x1000).

Task:

* Calculate the speedup using the execution times of the CPU and GPU implementations.

Code :

%%writefile matrix\_addition.cu

#include <iostream>

#include <cuda.h>

#include <cstdlib>

#include <chrono>

#define M 1000

#define N 1000

void initialize\_matrix(float \*matrix, int rows, int cols) {

    for (int i = 0; i < rows \* cols; i++) {

        matrix[i] = static\_cast<float>(rand()) / RAND\_MAX;

    }

}

// CPU-based matrix addition

void matrix\_add\_cpu(float \*A, float \*B, float \*C, int rows, int cols) {

    for (int i = 0; i < rows \* cols; i++) {

        C[i] = A[i] + B[i];

    }

}

// CUDA kernel for matrix addition

\_\_global\_\_ void matrix\_add\_cuda(float \*A, float \*B, float \*C, int rows, int cols) {

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

    if (idx < rows \* cols) {

        C[idx] = A[idx] + B[idx];

    }

}

int main() {

    float \*A, \*B, \*C;         // Host matrices

    float \*d\_A, \*d\_B, \*d\_C;   // Device matrices

    // Allocate memory on host

    int size = M \* N;

    A = new float[size];

    B = new float[size];

    C = new float[size];

    // Initialize matrices

    initialize\_matrix(A, M, N);

    initialize\_matrix(B, M, N);

    // CPU execution time

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

    matrix\_add\_cpu(A, B, C, M, N);

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

    float cpu\_time = std::chrono::duration<float, std::milli>(end\_cpu - start\_cpu).count();

    std::cout << "CPU Time: " << cpu\_time << " ms" << std::endl;

    // Allocate memory on device

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

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

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

    // Copy data from host to device

    cudaMemcpy(d\_A, A, size \* sizeof(float), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_B, B, size \* sizeof(float), cudaMemcpyHostToDevice);

    // Launch CUDA kernel

    int blockSize = 256;

    int numBlocks = (size + blockSize - 1) / blockSize;

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

    matrix\_add\_cuda<<<numBlocks, blockSize>>>(d\_A, d\_B, d\_C, M, N);

    cudaDeviceSynchronize();

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

    float gpu\_time = std::chrono::duration<float, std::milli>(end\_gpu - start\_gpu).count();

    // Copy result back to host

    cudaMemcpy(C, d\_C, size \* sizeof(float), cudaMemcpyDeviceToHost);

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

    std::cout << "Speedup: " << cpu\_time / gpu\_time << std::endl;

    // Free memory

    cudaFree(d\_A);

    cudaFree(d\_B);

    cudaFree(d\_C);

    delete[] A;

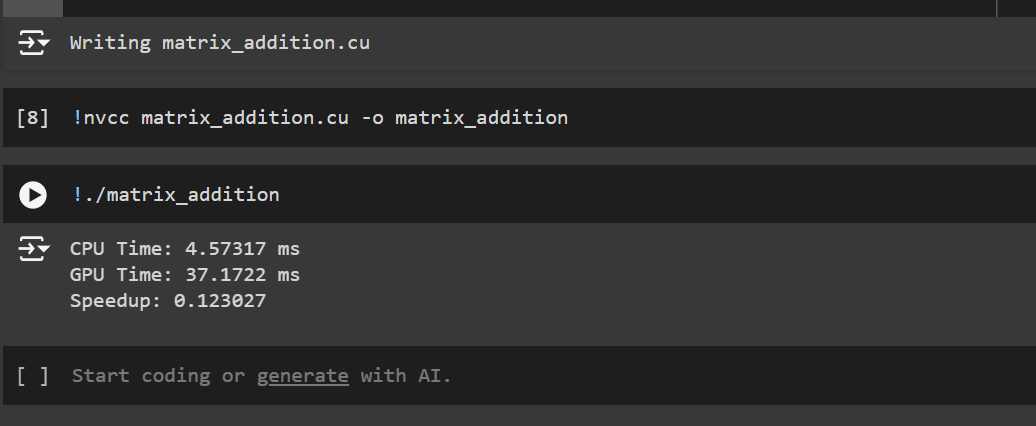
    delete[] B;

    delete[] C;

    return 0;

}

Screenshot :



**Problem 3: Dot Product of Two Vectors using CUDA**

Problem Statement: Write a CUDA C program to compute the dot product of two vectors A and B of size N. The dot product is defined as:

Details:

* Initialize the vectors A and B with random values.
* Implement the dot product calculation using both serial (CPU) and parallel (CUDA) approaches.
* Measure the execution time for both implementations with different vector sizes (e.g., N = 10^5, 10^6, 10^7).
* Use atomic operations or shared memory reduction in the CUDA kernel to compute the final sum.

Task:

* Calculate and report the speedup for different vector sizes.

Code :

**%%writefile vector\_dot\_product.cu**

**#include <iostream>**

**#include <cuda.h>**

**#include <cstdlib>**

**#include <chrono>**

**#define N 1000000**

**void initialize\_vector(float \*vec, int size) {**

**for (int i = 0; i < size; i++) {**

**vec[i] = static\_cast<float>(rand()) / RAND\_MAX;**

**}**

**}**

**// CPU-based dot product**

**float dot\_product\_cpu(float \*A, float \*B, int size) {**

**float sum = 0;**

**for (int i = 0; i < size; i++) {**

**sum += A[i] \* B[i];**

**}**

**return sum;**

**}**

**// CUDA kernel for dot product using atomicAdd**

**\_\_global\_\_ void dot\_product\_cuda(float \*A, float \*B, float \*C, int size) {**

**\_\_shared\_\_ float cache[256];**

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

**int cacheIdx = threadIdx.x;**

**float temp = 0;**

**while (idx < size) {**

**temp += A[idx] \* B[idx];**

**idx += blockDim.x \* gridDim.x;**

**}**

**cache[cacheIdx] = temp;**

**\_\_syncthreads();**

**// Reduction within the block**

**int i = blockDim.x / 2;**

**while (i != 0) {**

**if (cacheIdx < i) {**

**cache[cacheIdx] += cache[cacheIdx + i];**

**}**

**\_\_syncthreads();**

**i /= 2;**

**}**

**// Add the result of this block to global result**

**if (cacheIdx == 0) atomicAdd(C, cache[0]);**

**}**

**int main() {**

**float \*A, \*B;           // Host vectors**

**float \*d\_A, \*d\_B, \*d\_C; // Device vectors and result**

**A = new float[N];**

**B = new float[N];**

**float cpu\_result = 0;**

**float gpu\_result = 0;**

**// Initialize vectors**

**initialize\_vector(A, N);**

**initialize\_vector(B, N);**

**// CPU execution time**

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

**cpu\_result = dot\_product\_cpu(A, B, N);**

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

**float cpu\_time = std::chrono::duration<float, std::milli>(end\_cpu - start\_cpu).count();**

**std::cout << "CPU Result: " << cpu\_result << ", Time: " << cpu\_time << " ms" << std::endl;**

**// Allocate memory on device**

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

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

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

**// Copy data from host to device**

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

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

**cudaMemset(d\_C, 0, sizeof(float)); // Initialize result on device to 0**

**// Launch CUDA kernel**

**int blockSize = 256;**

**int numBlocks = (N + blockSize - 1) / blockSize;**

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

**dot\_product\_cuda<<<numBlocks, blockSize>>>(d\_A, d\_B, d\_C, N);**

**cudaDeviceSynchronize();**

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

**float gpu\_time = std::chrono::duration<float, std::milli>(end\_gpu - start\_gpu).count();**

**// Copy result back to host**

**cudaMemcpy(&gpu\_result, d\_C, sizeof(float), cudaMemcpyDeviceToHost);**

**std::cout << "GPU Result: " << gpu\_result << ", Time: " << gpu\_time << " ms" << std::endl;**

**std::cout << "Speedup: " << cpu\_time / gpu\_time << std::endl;**

**// Free memory**

**cudaFree(d\_A);**

**cudaFree(d\_B);**

**cudaFree(d\_C);**

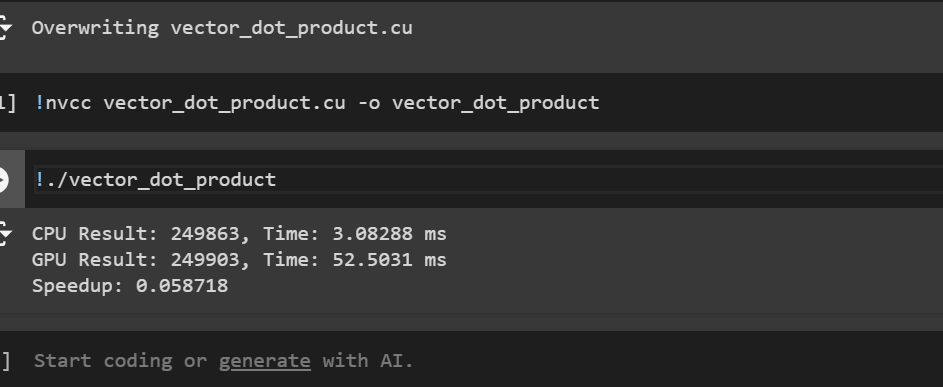
**delete[] A;**

**delete[] B;**

**return 0;**

**}**

**Screenshot :**

****

**Problem 4: Matrix Multiplication using CUDA**

Problem Statement: Write a CUDA C program to perform matrix multiplication. Given two matrices A (MxN) and B (NxP), compute the resulting matrix C (MxP) where:

Details:

* Initialize the matrices A and B with random values.
* Write code for both serial (CPU-based) and parallel (CUDA-based) implementations.
* Measure the execution time of both implementations for various matrix sizes (e.g., 100x100, 500x500, 1000x1000).

Task:

* Calculate the speedup by comparing the CPU and GPU execution times.

**Code :**

**%%writefile matrix\_multiplication.cu**

**#include <iostream>**

**#include <cuda.h>**

**#include <cstdlib>**

**#include <chrono>**

**#define M 500**

**#define N 500**

**#define P 500**

**void initialize\_matrix(float \*matrix, int rows, int cols) {**

**for (int i = 0; i < rows \* cols; i++) {**

**matrix[i] = static\_cast<float>(rand()) / RAND\_MAX;**

**}**

**}**

**// CPU-based matrix multiplication**

**void matrix\_multiply\_cpu(float \*A, float \*B, float \*C, int m, int n, int p) {**

**for (int i = 0; i < m; i++) {**

**for (int j = 0; j < p; j++) {**

**float sum = 0;**

**for (int k = 0; k < n; k++) {**

**sum += A[i \* n + k] \* B[k \* p + j];**

**}**

**C[i \* p + j] = sum;**

**}**

**}**

**}**

**// CUDA kernel for matrix multiplication**

**\_\_global\_\_ void matrix\_multiply\_cuda(float \*A, float \*B, float \*C, int m, int n, int p) {**

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

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

**if (row < m && col < p) {**

**float sum = 0;**

**for (int k = 0; k < n; k++) {**

**sum += A[row \* n + k] \* B[k \* p + col];**

**}**

**C[row \* p + col] = sum;**

**}**

**}**

**int main() {**

**float \*A, \*B, \*C;           // Host matrices**

**float \*d\_A, \*d\_B, \*d\_C;     // Device matrices**

**int sizeA = M \* N;**

**int sizeB = N \* P;**

**int sizeC = M \* P;**

**A = new float[sizeA];**

**B = new float[sizeB];**

**C = new float[sizeC];**

**// Initialize matrices**

**initialize\_matrix(A, M, N);**

**initialize\_matrix(B, N, P);**

**// CPU execution time**

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

**matrix\_multiply\_cpu(A, B, C, M, N, P);**

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

**float cpu\_time = std::chrono::duration<float, std::milli>(end\_cpu - start\_cpu).count();**

**std::cout << "CPU Time: " << cpu\_time << " ms" << std::endl;**

**// Allocate memory on device**

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

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

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

**// Copy data from host to device**

**cudaMemcpy(d\_A, A, sizeA \* sizeof(float), cudaMemcpyHostToDevice);**

**cudaMemcpy(d\_B, B, sizeB \* sizeof(float), cudaMemcpyHostToDevice);**

**// Define block and grid sizes**

**dim3 blockSize(16, 16);**

**dim3 gridSize((P + blockSize.x - 1) / blockSize.x, (M + blockSize.y - 1) / blockSize.y);**

**// Launch CUDA kernel**

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

**matrix\_multiply\_cuda<<<gridSize, blockSize>>>(d\_A, d\_B, d\_C, M, N, P);**

**cudaDeviceSynchronize();**

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

**float gpu\_time = std::chrono::duration<float, std::milli>(end\_gpu - start\_gpu).count();**

**// Copy result back to host**

**cudaMemcpy(C, d\_C, sizeC \* sizeof(float), cudaMemcpyDeviceToHost);**

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

**std::cout << "Speedup: " << cpu\_time / gpu\_time << std::endl;**

**// Free memory**

**cudaFree(d\_A);**

**cudaFree(d\_B);**

**cudaFree(d\_C);**

**delete[] A;**

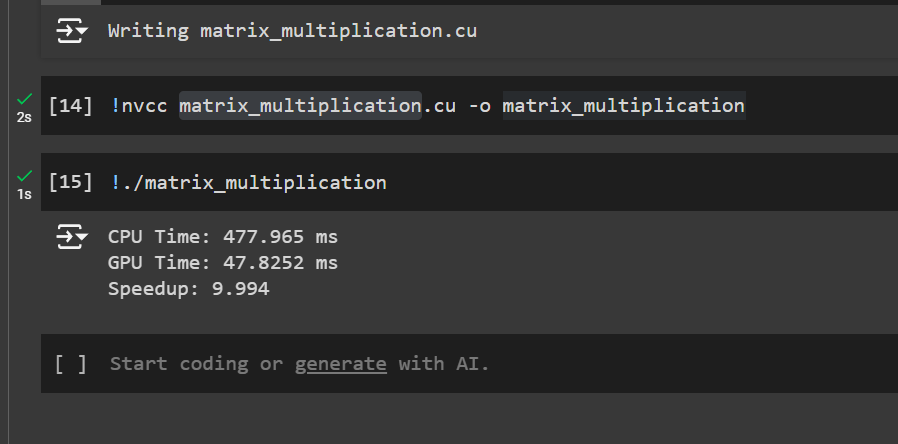
**delete[] B;**

**delete[] C;**

**return 0;**

**}**

**Screenshot :**

****