Name: Manish Namdev Barage

PRN No: 22520007(B6)

High Performance Computing Lab

Practical No. 12

<u>Title of practical: Parallel Programming using of CUDA C</u>

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.

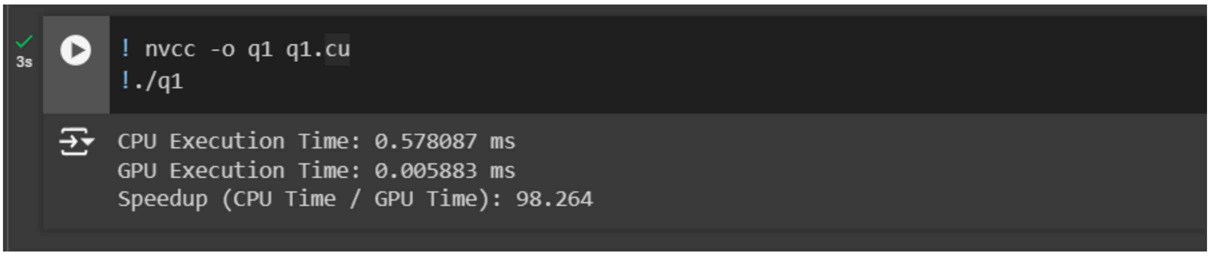
Code :

```
%%writefile q1.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>
#include <ctime>
#include <chrono>
using namespace std;
__global__ void vectorAddCUDA(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() {
```

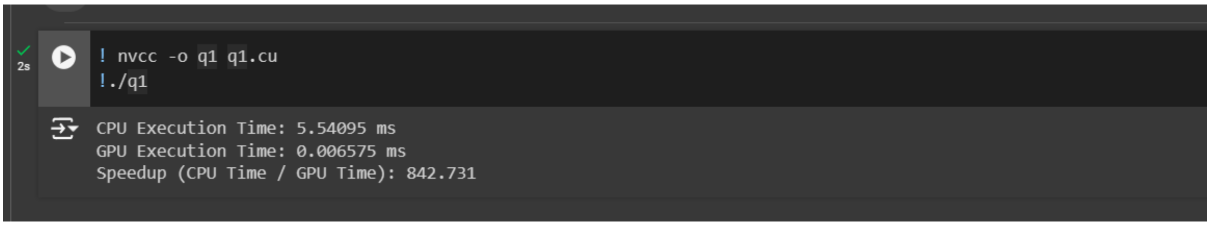
```
int N = 1000000;
size_t size = N * sizeof(float);

float *h_A = (float*)malloc(size);
float *h_B = (float*)malloc(size);
float *h_C_cpu = (float*)malloc(size); // Result for CPU
float *h_C_gpu = (float*)malloc(size); // Result for GPU

srand(time(0));
for (int i = 0; i < N; i++) {
    h_A[i] = static_cast<float>(rand()) / RAND_MAX;
    h_B[i] = static_cast<float>(rand()) / RAND_MAX;
}

// CPU (Serial) Execution
auto start_cpu = chrono::high_resolution_clock::now();
vectorAddCPU(h_A, h_B, h_C_cpu, N);
auto end_cpu = chrono::high_resolution_clock::now();
chrono::duration<float, milli> cpu_duration = end_cpu - start_cpu;
cout << "CPU Execution Time: " << cpu_duration.count() << " ms" << endl;

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);

// Define block and grid sizes
int blockSize = 256;
int gridSize = (N + blockSize - 1) / blockSize;

// GPU (CUDA) Execution
```

```
    auto start_gpu = chrono::high_resolution_clock::now();
    vectorAddCUDA<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();
    auto end_gpu = chrono::high_resolution_clock::now();
    chrono::duration<float, milli> gpu_duration = end_gpu - start_gpu;
    cout << "GPU Execution Time: " << gpu_duration.count() << " ms" << endl;

    cudaMemcpy(h_C_gpu, d_C, size, cudaMemcpyDeviceToHost);

    float speedup = cpu_duration.count() / gpu_duration.count();
    cout << "Speedup (CPU Time / GPU Time): " << speedup << endl;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C_cpu);
    free(h_C_gpu);
    return 0;
}
```

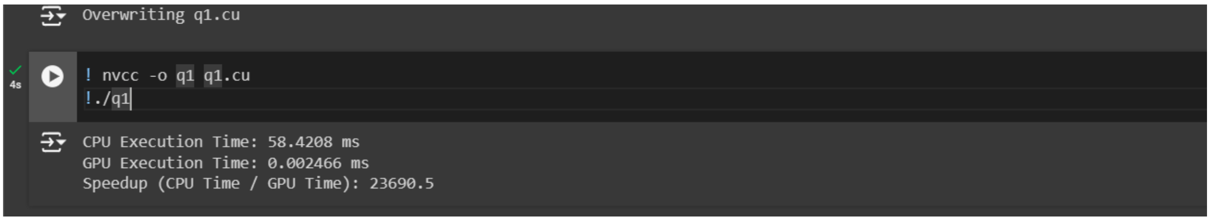
1) 10^5



2) 10^6



3) 10^7



```
Overwriting q1.cu
! nvcc -o q1 q1.cu
!./q1
CPU Execution Time: 58.4208 ms
GPU Execution Time: 0.002466 ms
Speedup (CPU Time / GPU Time): 23690.5
```

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.

```
%%writefile q2.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;

__global__ void matrixAddCUDA(const float* A, const float* B, float* C, int M, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N) {
        int index = row * N + col;
        C[index] = A[index] + B[index];
    }
}

void matrixAddCPU(const float* A, const float* B, float* C, int M, int N) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            int index = i * N + j;
```

```
            C[index] = A[index] + B[index];
        }
    }
}

int main() {
    int M = 100;
    int N = 100;
    size_t size = M * N * sizeof(float);

    // Allocate host memory
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C_cpu = (float*)malloc(size);
    float *h_C_gpu = (float*)malloc(size);



    srand(time(0));
    for (int i = 0; i < M * N; i++) {
        h_A[i] = static_cast<float>(rand()) / RAND_MAX;
        h_B[i] = static_cast<float>(rand()) / RAND_MAX;
    }

    // CPU (Serial) Execution
    auto start_cpu = chrono::high_resolution_clock::now();
    matrixAddCPU(h_A, h_B, h_C_cpu, M, N);
    auto end_cpu = chrono::high_resolution_clock::now();
    chrono::duration<float, milli> cpu_duration = end_cpu - start_cpu;
    cout << "CPU Execution Time: " << cpu_duration.count() << " ms" << endl;

    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);

    dim3 blockSize(16, 16); // 16x16 threads per block
    dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (M + blockSize.y - 1) / blockSize.y);

    // GPU (CUDA) Execution
    auto start_gpu = chrono::high_resolution_clock::now();
    matrixAddCUDA<<<gridSize, blockSize>>>(d_A, d_B, d_C, M, N);
    cudaDeviceSynchronize(); // Ensure kernel has finished executing
    auto end_gpu = chrono::high_resolution_clock::now();

    chrono::duration<float, milli> gpu_duration = end_gpu - start_gpu;
    cout << "GPU Execution Time: " << gpu_duration.count() << " ms" << endl;

    cudaMemcpy(h_C_gpu, d_C, size, cudaMemcpyDeviceToHost);

    float speedup = cpu_duration.count() / gpu_duration.count();
    cout << "Speedup (CPU Time / GPU Time): " << speedup << endl;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C_cpu);
    free(h_C_gpu);

    return 0;
}
```

1) 100 X 100

```
!nvcc -o q2 q2.cu
!./q2

CPU Execution Time: 0.088439 ms
GPU Execution Time: 0.002779 ms
Speedup (CPU Time / GPU Time): 31.824
```

2) 500 X 500

```
!nvcc -o q2 q2.cu
!./q2

CPU Execution Time: 2.49819 ms
GPU Execution Time: 0.008688 ms
Speedup (CPU Time / GPU Time): 287.545
```

3) 1000 X 1000

```
!nvcc -o q2 q2.cu
!./q2

CPU Execution Time: 6.67393 ms
GPU Execution Time: 0.013812 ms
Speedup (CPU Time / GPU Time): 483.198
```

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:

```
%%writefile q3.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;

__global__ void dotProductCUDA(const float* A, const float* B, float* result, int N) {
    __shared__ float cache[256]; // Shared memory for partial sums
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int cacheIndex = threadIdx.x;
```

```
    float temp = 0.0;
    while (tid < N) {
        temp += A[tid] * B[tid];
        tid += blockDim.x * gridDim.x;
    }

    cache[cacheIndex] = temp;
    __syncthreads();

    int i = blockDim.x / 2;
    while (i != 0) {
        if (cacheIndex < i) {
            cache[cacheIndex] += cache[cacheIndex + i];
        }
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0) {
        atomicAdd(result, cache[0]);
    }
}

float dotProductCPU(const float* A, const float* B, int N) {
    float result = 0.0;
    for (int i = 0; i < N; ++i) {
        result += A[i] * B[i];
    }
    return result;
}

int main() {
    int N = 100000;
```

```
size_t size = N * sizeof(float);

float *h_A = (float*)malloc(size);
float *h_B = (float*)malloc(size);

srand(time(0));
for (int i = 0; i < N; i++) {
    h_A[i] = static_cast<float>(rand()) / RAND_MAX;
    h_B[i] = static_cast<float>(rand()) / RAND_MAX;
}

// CPU (Serial) Execution
auto start_cpu = chrono::high_resolution_clock::now();
float cpu_result = dotProductCPU(h_A, h_B, N);
auto end_cpu = chrono::high_resolution_clock::now();
chrono::duration<float, milli> cpu_duration = end_cpu - start_cpu;
cout << "CPU Execution Time: " << cpu_duration.count() << " ms" << endl;
cout << "CPU Dot Product Result: " << cpu_result << endl;

float *d_A, *d_B, *d_result;
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);
cudaMalloc((void**)&d_result, sizeof(float));

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
cudaMemset(d_result, 0, sizeof(float));

int blockSize = 256;
int gridSize = (N + blockSize - 1) / blockSize;

// GPU (CUDA) Execution
auto start_gpu = chrono::high_resolution_clock::now();
dotProductCUDA<<<gridSize, blockSize>>>(d_A, d_B, d_result, N);
```

```cpp
    cudaDeviceSynchronize(); // Ensure kernel has finished executing
    auto end_gpu = chrono::high_resolution_clock::now();

    float gpu_result;
    cudaMemcpy(&gpu_result, d_result, sizeof(float), cudaMemcpyDeviceToHost);

    chrono::duration<float, milli> gpu_duration = end_gpu - start_gpu;

    float speedup = cpu_duration.count() / gpu_duration.count();
    cout << "Speedup (CPU Time / GPU Time): " << speedup << endl;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_result);
    free(h_A);
    free(h_B);

    return 0;
}
```

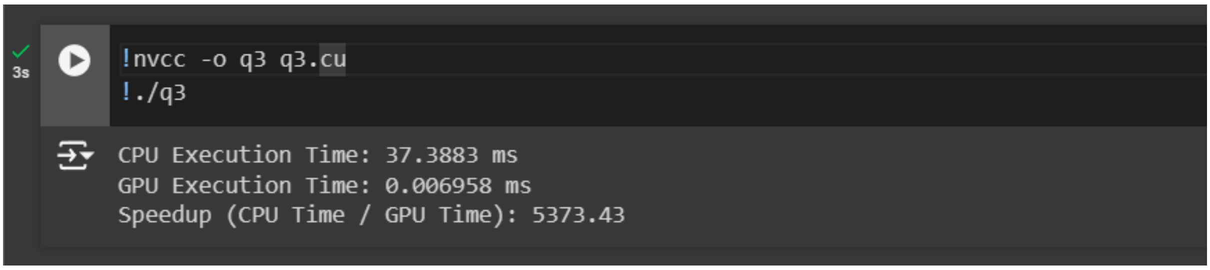
1) 10^5

```
!nvcc -o q3 q3.cu
!./q3

CPU Execution Time: 0.441441 ms
GPU Execution Time: 0.008717 ms
Speedup (CPU Time / GPU Time): 50.6414
```

2) 10^6

```
!nvcc -o q3 q3.cu
!./q3

CPU Execution Time: 3.70576 ms
GPU Execution Time: 0.009961 ms
Speedup (CPU Time / GPU Time): 372.027
```

3) 10^7



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:

```
%%writefile q4.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;

__global__ void matrixMultiplyCUDA(const float* A, const 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 = 0.0;
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * P + col];
        }
        C[row * P + col] = sum;
    }
}
```

```
void matrixMultiplyCPU(const float* A, const 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 = 0.0;
            for (int k = 0; k < N; ++k) {
                sum += A[i * N + k] * B[k * P + j];
            }
            C[i * P + j] = sum;
        }
    }
}

int main() {
    int M = 100;
    int N = 100;
    int P = 100;
    size_t sizeA = M * N * sizeof(float);
    size_t sizeB = N * P * sizeof(float);
    size_t sizeC = M * P * sizeof(float);

    float *h_A = (float*)malloc(sizeA);
    float *h_B = (float*)malloc(sizeB);
    float *h_C_cpu = (float*)malloc(sizeC); // Result for CPU
    float *h_C_gpu = (float*)malloc(sizeC); // Result for GPU

    srand(time(0));
    for (int i = 0; i < M * N; i++) {
        h_A[i] = static_cast<float>(rand()) / RAND_MAX;
    }
    for (int i = 0; i < N * P; i++) {
        h_B[i] = static_cast<float>(rand()) / RAND_MAX;
    }

    // CPU (Serial) Execution
```

```
auto start_cpu = chrono::high_resolution_clock::now();
matrixMultiplyCPU(h_A, h_B, h_C_cpu, M, N, P);
auto end_cpu = chrono::high_resolution_clock::now();
chrono::duration<float, milli> cpu_duration = end_cpu - start_cpu;
cout << "CPU Execution Time: " << cpu_duration.count() << " ms" << endl;

float *d_A, *d_B, *d_C;
cudaMalloc((void**)&d_A, sizeA);
cudaMalloc((void**)&d_B, sizeB);
cudaMalloc((void**)&d_C, sizeC);

cudaMemcpy(d_A, h_A, sizeA, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, sizeB, cudaMemcpyHostToDevice);

dim3 blockSize(16, 16);
dim3 gridSize((P + blockSize.x - 1) / blockSize.x, (M + blockSize.y - 1) / blockSize.y);

// GPU (CUDA) Execution
auto start_gpu = chrono::high_resolution_clock::now();
matrixMultiplyCUDA<<<gridSize, blockSize>>>(d_A, d_B, d_C, M, N, P);
cudaDeviceSynchronize();
auto end_gpu = chrono::high_resolution_clock::now();

chrono::duration<float, milli> gpu_duration = end_gpu - start_gpu;
cout << "GPU Execution Time: " << gpu_duration.count() << " ms" << endl;

cudaMemcpy(h_C_gpu, d_C, sizeC, cudaMemcpyDeviceToHost);

float speedup = cpu_duration.count() / gpu_duration.count();
cout << "Speedup (CPU Time / GPU Time): " << speedup << endl;

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

```
    free(h_A);

    free(h_B);

    free(h_C_cpu);

    free(h_C_gpu);


    return 0;

}
```

1) 100 X 100

```
[45] !nvcc -o q4 q4.cu
     !./q4

CPU Execution Time: 6.97269 ms
GPU Execution Time: 0.010718 ms
Speedup (CPU Time / GPU Time): 650.559
```

2) 500 X 500

```
!nvcc -o q4 q4.cu
!./q4

CPU Execution Time: 569.779 ms
GPU Execution Time: 0.002067 ms
Speedup (CPU Time / GPU Time): 275655
```

3) 1000 X 1000

```
!nvcc -o q4 q4.cu
!./q4

CPU Execution Time: 7745.16 ms
GPU Execution Time: 0.011274 ms
Speedup (CPU Time / GPU Time): 686993
```