Practical No 12

PRN: 22520005

Name: Aftab Imtiyaj Bhadgaonkar

Batch: B6

Course: High Performance Computing Lab

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:

```
code = '''
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>

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

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

```
int main() {
```

```
    int N = 10000000;
    printf("Current input size: %d\\n", N);
    float *h_A, *h_B, *h_C;
    h_A = (float *)malloc(N * sizeof(float));
    h_B = (float *)malloc(N * sizeof(float));
    h_C = (float *)malloc(N * sizeof(float));

    srand(time(0));
    for (int i = 0; i < N; i++) {
        h_A[i] = rand() % 100; // Random float numbers
        h_B[i] = rand() % 100;
    }

    clock_t start_cpu = clock();
    cpuVectorAdd(h_A, h_B, h_C, N);
    clock_t end_cpu = clock();
    double cpu_time = double(end_cpu - start_cpu) / CLOCKS_PER_SEC;

    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));

    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    clock_t start_gpu = clock();
    vectorAdd<<<numBlocks, blockSize>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize(); // Wait for GPU to finish
    clock_t end_gpu = clock();
    double gpu_time = double(end_gpu - start_gpu) / CLOCKS_PER_SEC;

    cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);

    for (int i = 0; i < 10; i++) { // Display first 10 results
        printf("%.2f + %.2f = %.2f\\n", h_A[i], h_B[i], h_C[i]);
    }

    printf("CPU time: %f seconds\\n", cpu_time);
    printf("GPU time: %f seconds\\n", gpu_time);
    printf("Speedup: %f\\n", cpu_time / gpu_time);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
'''

with open('vector_addition.cu', 'w') as f:
    f.write(code)

!nvcc vector_addition.cu -o vector_add
!./vector_add
```

Output:

Input N: 10^7

```
Current input size: 10000000
94.00 + 41.00 = 135.00
52.00 + 72.00 = 124.00
25.00 + 77.00 = 102.00
73.00 + 97.00 = 170.00
47.00 + 76.00 = 123.00
95.00 + 74.00 = 169.00
97.00 + 90.00 = 187.00
28.00 + 92.00 = 120.00
10.00 + 79.00 = 89.00
62.00 + 88.00 = 150.00
CPU time: 0.061550 seconds
GPU time: 0.000671 seconds
Speedup: 91.728763
```

Input N: 10^9

```
Current input size: 1000000000
76.00 + 64.00 = 140.00
12.00 + 10.00 = 22.00
34.00 + 91.00 = 125.00
99.00 + 10.00 = 109.00
4.00 + 6.00 = 10.00
94.00 + 7.00 = 101.00
72.00 + 94.00 = 166.00
55.00 + 74.00 = 129.00
67.00 + 78.00 = 145.00
85.00 + 32.00 = 117.00
CPU time: 5.750296 seconds
GPU time: 0.045569 seconds
Speedup: 126.188769
```

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:

```
code = '''
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>

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

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

```
int main() {
    int M = 10000;
    int N = 10000;
    printf("Current matrix size: %d x %d\\n", M, N);
    float *h_A, *h_B, *h_C;
    h_A = (float *)malloc(M * N * sizeof(float));
    h_B = (float *)malloc(M * N * sizeof(float));
    h_C = (float *)malloc(M * N * sizeof(float));
```

```
    srand(time(0));
    for (int i = 0; i < M * N; i++) {
        h_A[i] = rand() % 100;
        h_B[i] = rand() % 100;
    }
```

```
    clock_t start_cpu = clock();
```

```
    cpuMatrixAdd(h_A, h_B, h_C, M, N);
    clock_t end_cpu = clock();
    double cpu_time = double(end_cpu - start_cpu) / CLOCKS_PER_SEC;

    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, M * N * sizeof(float));
    cudaMalloc((void **)&d_B, M * N * sizeof(float));
    cudaMalloc((void **)&d_C, M * N * sizeof(float));

    cudaMemcpy(d_A, h_A, M * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, M * N * sizeof(float), cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (M + threadsPerBlock.y - 1) /
threadsPerBlock.y);
    clock_t start_gpu = clock();
    matrixAddKernel<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, M, N);
    cudaDeviceSynchronize();
    clock_t end_gpu = clock();
    double gpu_time = double(end_gpu - start_gpu) / CLOCKS_PER_SEC;

    cudaMemcpy(h_C, d_C, M * N * sizeof(float), cudaMemcpyDeviceToHost);

    printf("CPU time: %f seconds\\n", cpu_time);
    printf("GPU time: %f seconds\\n", gpu_time);
    printf("Speedup: %f\\n", cpu_time / gpu_time);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
'''

with open('matrix_addition.cu', 'w') as f:
    f.write(code)

!nvcc matrix_addition.cu -o matrix_add
!./matrix_add
```

Output:

Input N: 1000 x 1000

```
Current matrix size: 1000 x 1000
CPU time: 0.005045 seconds
GPU time: 0.000496 seconds
Speedup: 10.171371
```

Input N: 10000 x 10000

```
Current matrix size: 10000 x 10000
CPU time: 0.538746 seconds
GPU time: 0.014548 seconds
Speedup: 37.032307
```

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:

```
code = '''
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>

__global__ void dotProductKernel(float *A, float *B, float *C, int N) {
    extern __shared__ float sharedData[];
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        sharedData[threadIdx.x] = A[i] * B[i];
    } else {
        sharedData[threadIdx.x] = 0;
    }
    __syncthreads();
```

```
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (threadIdx.x < stride) {
            sharedData[threadIdx.x] += sharedData[threadIdx.x + stride];
        }
        __syncthreads();
    }
```

```
    if (threadIdx.x == 0) {
        atomicAdd(C, sharedData[0]);
    }
}
```

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

```
int main() {
    int N = 10000000;
    printf("Current vector size: %d\\n", N);
    float *h_A, *h_B, *h_C;
    h_A = (float *)malloc(N * sizeof(float));
    h_B = (float *)malloc(N * sizeof(float));
```

```
    h_C = (float *)malloc(sizeof(float));

    srand(time(0));
    for (int i = 0; i < N; i++) {
        h_A[i] = rand() % 100;
        h_B[i] = rand() % 100;
    }

    clock_t start_cpu = clock();
    cpuDotProduct(h_A, h_B, h_C, N);
    clock_t end_cpu = clock();
    double cpu_time = double(end_cpu - start_cpu) / CLOCKS_PER_SEC;

    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, sizeof(float));

    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    clock_t start_gpu = clock();
    dotProductKernel<<<numBlocks, blockSize, blockSize * sizeof(float)>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();
    clock_t end_gpu = clock();
    double gpu_time = double(end_gpu - start_gpu) / CLOCKS_PER_SEC;

    cudaMemcpy(h_C, d_C, sizeof(float), cudaMemcpyDeviceToHost);

    printf("CPU result: %f\\n", *h_C);
    printf("GPU result: %f\\n", *h_C);
    printf("CPU time: %f seconds\\n", cpu_time);
    printf("GPU time: %f seconds\\n", gpu_time);
    printf("Speedup: %f\\n", cpu_time / gpu_time);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
'''

with open('dot_product.cu', 'w') as f:
    f.write(code)

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

Output:

Input N: 10^6

```
Current vector size: 1000000
CPU result: 4891603968.000000
GPU result: 4891603968.000000
CPU time: 0.002926 seconds
GPU time: 0.000280 seconds
Speedup: 10.450000
```

Input N: 10^7

```
Current vector size: 10000000
CPU result: 48719536128.000000
GPU result: 48719536128.000000
CPU time: 0.031295 seconds
GPU time: 0.001192 seconds
Speedup: 26.254195
```

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:

```
code = '''
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>

__global__ void matrixMultiplyKernel(float *A, float *B, float *C, int M, int N, int P) {
    int row = blockDim.y * blockIdx.y + threadIdx.y;
    int col = blockDim.x * blockIdx.x + threadIdx.x;
    float sum = 0;
```

```
    if (row < M && col < P) {
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * P + col];
        }
        C[row * P + col] = sum;
    }
}
```

```
void cpuMatrixMultiply(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++) {
            C[i * P + j] = 0;
            for (int k = 0; k < N; k++) {
                C[i * P + j] += A[i * N + k] * B[k * P + j];
            }
        }
    }
}
```

```
void initializeMatrix(float *matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; i++) {
        matrix[i] = (float)(rand() % 100); // Random values
    }
}
```

```
int main() {
    int matrixSizes[3][2] = {{100, 100}, {500, 500}, {1000, 1000}};
    srand(time(0));
```

```
    for (int size = 0; size < 3; size++) {
        int M = matrixSizes[size][0];
        int N = matrixSizes[size][1];
        int P = N; // For square multiplication

        float *h_A = (float *)malloc(M * N * sizeof(float));
        float *h_B = (float *)malloc(N * P * sizeof(float));
        float *h_C_cpu = (float *)malloc(M * P * sizeof(float));
        float *h_C_gpu = (float *)malloc(M * P * sizeof(float));

        initializeMatrix(h_A, M, N);
        initializeMatrix(h_B, N, P);

        clock_t start_cpu = clock();
        cpuMatrixMultiply(h_A, h_B, h_C_cpu, M, N, P);
        clock_t end_cpu = clock();
        double cpu_time = double(end_cpu - start_cpu) / CLOCKS_PER_SEC;

        float *d_A, *d_B, *d_C;
        cudaMalloc((void **)&d_A, M * N * sizeof(float));
        cudaMalloc((void **)&d_B, N * P * sizeof(float));
        cudaMalloc((void **)&d_C, M * P * sizeof(float));

        cudaMemcpy(d_A, h_A, M * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, N * P * sizeof(float), cudaMemcpyHostToDevice);

        dim3 threadsPerBlock(16, 16);
        dim3 numBlocks((P + threadsPerBlock.x - 1) / threadsPerBlock.x, (M + threadsPerBlock.y - 1) /
threadsPerBlock.y);

        clock_t start_gpu = clock();
        matrixMultiplyKernel<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, M, N, P);
        cudaDeviceSynchronize(); // Wait for GPU to finish
        clock_t end_gpu = clock();
        double gpu_time = double(end_gpu - start_gpu) / CLOCKS_PER_SEC;

        cudaMemcpy(h_C_gpu, d_C, M * P * sizeof(float), cudaMemcpyDeviceToHost);

        // Compare results (optional)
        for (int i = 0; i < M * P; i++) {
            if (fabs(h_C_cpu[i] - h_C_gpu[i]) > 1e-5) {
                printf("Results do not match!\\n");
                break;
            }
        }

        printf("Matrix Size: %dx%d\\n", M, P);
        printf("CPU time: %f seconds\\n", cpu_time);
        printf("GPU time: %f seconds\\n", gpu_time);
        printf("Speedup: %f\\n", cpu_time / gpu_time);

        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;
}
```

```
'''
```

```
with open('matrix_multiplication.cu', 'w') as f:
    f.write(code)
```

```
!nvcc matrix_multiplication.cu -o matrix_mult
!./matrix_mult
```

Output:

Input N: 100, 500, 1000

```
Matrix Size: 100x100
CPU time: 0.005078 seconds
GPU time: 0.000219 seconds
Speedup: 23.187215

Matrix Size: 500x500
CPU time: 0.718720 seconds
GPU time: 0.001120 seconds
Speedup: 641.714286

Matrix Size: 1000x1000
CPU time: 6.829388 seconds
GPU time: 0.007077 seconds
Speedup: 965.011728
```