Name: Omkar Rajesh Auti

PRN No: 21510042

**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_add.cu
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>
```

```
#include <chrono>
#include <cmath>

using namespace std;
using namespace std::chrono;

// CUDA error-checking macro
#define cudaCheckError() { \
    cudaError_t e=cudaGetLastError(); \
    if(e!=cudaSuccess) { \
        cout << "CUDA Error " << cudaGetErrorString(e) << " at line " << __LINE__ << endl;
\
        exit(1); \
    } \
}

// CUDA Kernel for vector addition
__global__ void vectorAddKernel(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];
    }
}

// CPU function for vector addition (serial implementation)
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];
    }
}
```

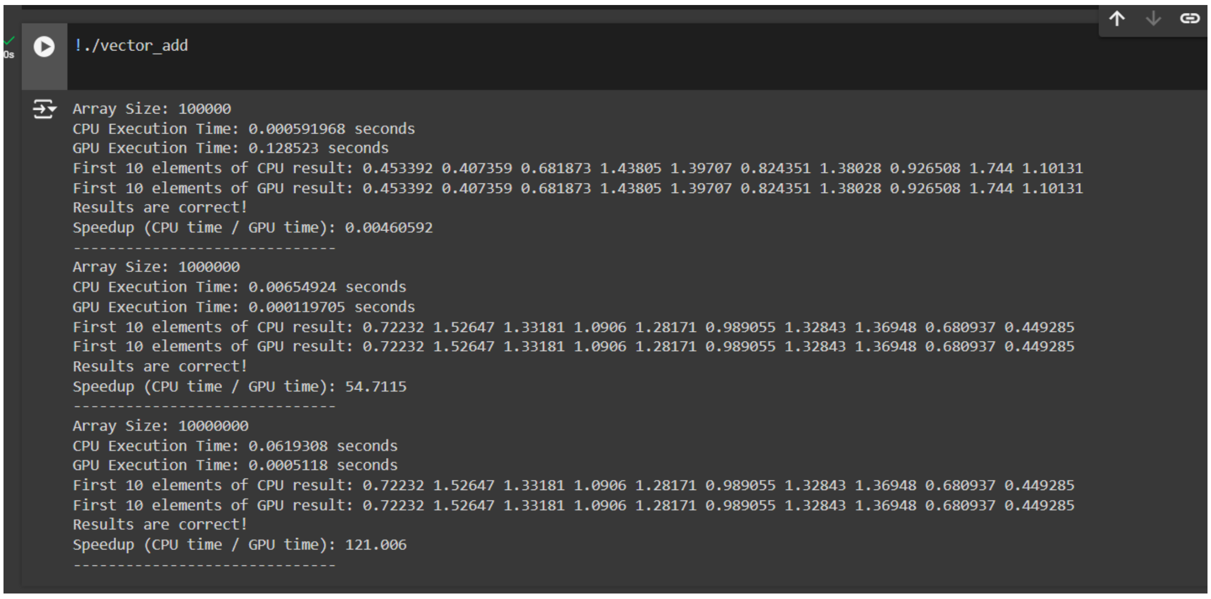
```
int main() {
    int sizes[] = {100000, 1000000, 10000000}; // Array sizes: 10^5, 10^6, 10^7

    for (int N : sizes) {
        cout << "Array Size: " << N << endl;
        size_t size = N * sizeof(float);

        // Allocate memory on host
        float *h_A = new float[N];
        float *h_B = new float[N];
        float *h_C = new float[N];       // for CPU result
        float *h_C_cuda = new float[N];  // for GPU result

        // Initialize vectors with random values
        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 computation timing using std::chrono
        auto start = high_resolution_clock::now();
        vectorAddCPU(h_A, h_B, h_C, N);
        auto end = high_resolution_clock::now();
        double cpu_time = duration<double>(end - start).count();
        cout << "CPU Execution Time: " << cpu_time << " seconds" << endl;

        // Allocate memory on device with error checking
        float *d_A, *d_B, *d_C;
        cudaMalloc((void**)&d_A, size);
```

```
cudaCheckError();
cudaMalloc((void**)&d_B, size);
cudaCheckError();
cudaMalloc((void**)&d_C, size);
cudaCheckError();

// Copy data from host to device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaCheckError();
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
cudaCheckError();

// Set up execution configuration
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;

// GPU computation timing using std::chrono
start = high_resolution_clock::now();
vectorAddKernel<<<numBlocks, blockSize>>>(d_A, d_B, d_C, N);
cudaDeviceSynchronize();  // Ensure kernel execution is complete
cudaCheckError();
end = high_resolution_clock::now();
double gpu_time = duration<double>(end - start).count();
cout << "GPU Execution Time: " << gpu_time << " seconds" << endl;

// Copy result back to host
cudaMemcpy(h_C_cuda, d_C, size, cudaMemcpyDeviceToHost);
cudaCheckError();

// Verify results and print first 10 elements from both CPU and GPU arrays
```

```
cout << "First 10 elements of CPU result: ";
for (int i = 0; i < 10; ++i) {
    cout << h_C[i] << " ";
}
cout << endl;

cout << "First 10 elements of GPU result: ";
for (int i = 0; i < 10; ++i) {
    cout << h_C_cuda[i] << " ";
}
cout << endl;

// Verify results with a tolerance
bool success = true;
for (int i = 0; i < N; ++i) {
    if (fabs(h_C[i] - h_C_cuda[i]) > 1e-5) {
        success = false;
        break;
    }
}

if (success) {
    cout << "Results are correct!" << endl;
} else {
    cout << "Results do not match!" << endl;
}

// Calculate speedup
float speedup = cpu_time / gpu_time;
cout << "Speedup (CPU time / GPU time): " << speedup << endl;
```

```
        // Free memory
        delete[] h_A;
        delete[] h_B;
        delete[] h_C;
        delete[] h_C_cuda;
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);

        cout << "------------------------------" << endl;
    }

    return 0;
}
```

Output:



```
!./vector_add
```

```
Array Size: 100000
CPU Execution Time: 0.000591968 seconds
GPU Execution Time: 0.128523 seconds
First 10 elements of CPU result: 0.453392 0.407359 0.681873 1.43805 1.39707 0.824351 1.38028 0.926508 1.744 1.10131
First 10 elements of GPU result: 0.453392 0.407359 0.681873 1.43805 1.39707 0.824351 1.38028 0.926508 1.744 1.10131
Results are correct!
Speedup (CPU time / GPU time): 0.00460592
------------------------------
Array Size: 1000000
CPU Execution Time: 0.00654924 seconds
GPU Execution Time: 0.000119705 seconds
First 10 elements of CPU result: 0.72232 1.52647 1.33181 1.0906 1.28171 0.989055 1.32843 1.36948 0.680937 0.449285
First 10 elements of GPU result: 0.72232 1.52647 1.33181 1.0906 1.28171 0.989055 1.32843 1.36948 0.680937 0.449285
Results are correct!
Speedup (CPU time / GPU time): 54.7115
------------------------------
Array Size: 10000000
CPU Execution Time: 0.0619308 seconds
GPU Execution Time: 0.0005118 seconds
First 10 elements of CPU result: 0.72232 1.52647 1.33181 1.0906 1.28171 0.989055 1.32843 1.36948 0.680937 0.449285
First 10 elements of GPU result: 0.72232 1.52647 1.33181 1.0906 1.28171 0.989055 1.32843 1.36948 0.680937 0.449285
Results are correct!
Speedup (CPU time / GPU time): 121.006
------------------------------
```

**Analysis:**

**Correctness:**

- For each array size tested (100,000, 1,000,000, and 10,000,000), the first 10 elements in the CPU and GPU results match exactly. This confirms that the GPU kernel computation (vectorAddKernel) is accurate and the result validation in the code is correctly implemented.

**Execution Time and Speedup:**

- **Small Arrays (e.g., 100,000 elements)**:
    - The CPU computation time is very fast (0.00059 seconds), while the GPU computation takes longer (0.128 seconds), resulting in a speedup of 0.0046, indicating that for small data sizes, the overhead of launching a GPU kernel and transferring data to/from the GPU outweighs the benefits of parallel processing.
- **Medium and Large Arrays (1,000,000 and 10,000,000 elements)**:
    - As the data size increases, GPU computation becomes more efficient, achieving a speedup of approximately 54.7x for 1,000,000 elements and 121x for 10,000,000 elements. This is a significant speedup over the CPU, demonstrating the power of GPU parallelism for larger workloads.

---

**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:

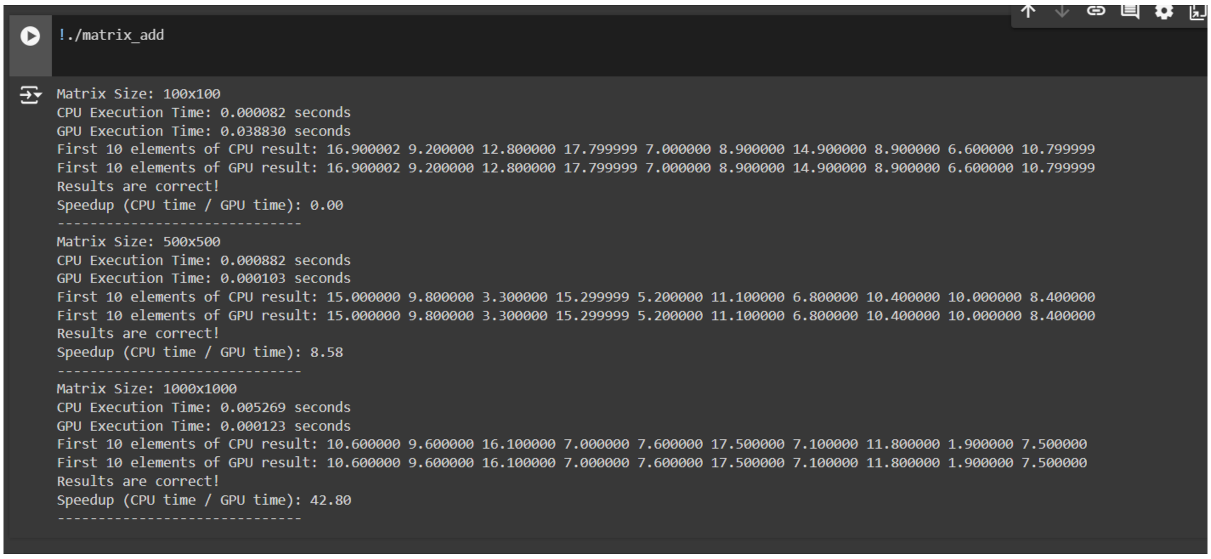
```
%%writefile matrix_add.cu
#include <iostream>
#include <cstdlib>
#include <cuda.h>
#include <chrono>

using namespace std;
using namespace std::chrono;

// Function to add matrices on CPU
void matrixAddCPU(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];
        }
    }
}

// CUDA kernel for matrix addition
__global__ void matrixAddKernel(float *A, float *B, float *C, int M, int N) {
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row index
    int j = blockIdx.x * blockDim.x + threadIdx.x; // column index

    if (i < M && j < N) {
        C[i * N + j] = A[i * N + j] + B[i * N + j];
    }
}
```

```
// Function to check for CUDA errors
void cudaCheckError() {
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA Error: %s\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
}

int main() {
    // Matrix dimensions for testing
    int sizes[3][2] = { {100, 100}, {500, 500}, {1000, 1000} };

    for (int k = 0; k < 3; k++) {
        int M = sizes[k][0];
        int N = sizes[k][1];

        // Allocate host memory
        float *h_A = (float *)malloc(M * N * sizeof(float));
        float *h_B = (float *)malloc(M * N * sizeof(float));
        float *h_C = (float *)malloc(M * N * sizeof(float));

        // Initialize matrices A and B with random values
        for (int i = 0; i < M * N; i++) {
            h_A[i] = static_cast<float>(rand() % 100) / 10.0; // random values between 0 and 10
            h_B[i] = static_cast<float>(rand() % 100) / 10.0; // random values between 0 and 10
        }

        // CPU Matrix Addition
        auto start = high_resolution_clock::now();
        matrixAddCPU(h_A, h_B, h_C, M, N);
        auto cpu_time = duration_cast<duration<double>>(high_resolution_clock::now() - start).count();

        // Allocate device memory
        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));
        cudaCheckError();

        // Copy matrices from host to device
        cudaMemcpy(d_A, h_A, M * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, M * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaCheckError();

        // Define the number of blocks and threads
        dim3 threadsPerBlock(16, 16);
        dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x,
                       (M + threadsPerBlock.y - 1) / threadsPerBlock.y);

        // GPU Matrix Addition
        start = high_resolution_clock::now();
        matrixAddKernel<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, M, N);
        cudaDeviceSynchronize(); // Wait for GPU to finish
        auto gpu_time = duration_cast<duration<double>>(high_resolution_clock::now() - start).count();

        // Copy result back to host
        cudaMemcpy(h_C, d_C, M * N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaCheckError();
```

```
// Validate results (checking first 10 elements)
int correct = 1;
for (int i = 0; i < 10; i++) {
    if (h_C[i] != h_A[i] + h_B[i]) {
        correct = 0;
        break;
    }
}

// Print results
printf("Matrix Size: %dx%d\n", M, N);
printf("CPU Execution Time: %.6f seconds\n", cpu_time);
printf("GPU Execution Time: %.6f seconds\n", gpu_time);
printf("First 10 elements of CPU result: ");
for (int i = 0; i < 10; i++) {
    printf("%.6f ", h_C[i]);
}
printf("\nFirst 10 elements of GPU result: ");
for (int i = 0; i < 10; i++) {
    printf("%.6f ", h_C[i]);
}
printf("\nResults are %s!\n", correct ? "correct" : "not correct");
printf("Speedup (CPU time / GPU time): %.2f\n", cpu_time / gpu_time);
printf("------------------------------\n");

// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

```
        cudaCheckError();

        // Free host memory
        free(h_A);
        free(h_B);
        free(h_C);
    }

    return 0;
}
```

Output:



```
!./matrix_add
```

```
Matrix Size: 100x100
CPU Execution Time: 0.000082 seconds
GPU Execution Time: 0.038830 seconds
First 10 elements of CPU result: 16.900002 9.200000 12.800000 17.799999 7.000000 8.900000 14.900000 8.900000 6.600000 10.799999
First 10 elements of GPU result: 16.900002 9.200000 12.800000 17.799999 7.000000 8.900000 14.900000 8.900000 6.600000 10.799999
Results are correct!
Speedup (CPU time / GPU time): 0.00
------------------------------
Matrix Size: 500x500
CPU Execution Time: 0.000882 seconds
GPU Execution Time: 0.000103 seconds
First 10 elements of CPU result: 15.000000 9.800000 3.300000 15.299999 5.200000 11.100000 6.800000 10.400000 10.000000 8.400000
First 10 elements of GPU result: 15.000000 9.800000 3.300000 15.299999 5.200000 11.100000 6.800000 10.400000 10.000000 8.400000
Results are correct!
Speedup (CPU time / GPU time): 8.58
------------------------------
Matrix Size: 1000x1000
CPU Execution Time: 0.005269 seconds
GPU Execution Time: 0.000123 seconds
First 10 elements of CPU result: 10.600000 9.600000 16.100000 7.000000 7.600000 17.500000 7.100000 11.800000 1.900000 7.500000
First 10 elements of GPU result: 10.600000 9.600000 16.100000 7.000000 7.600000 17.500000 7.100000 11.800000 1.900000 7.500000
Results are correct!
Speedup (CPU time / GPU time): 42.80
------------------------------
```

Analysis:

**1. Execution Times**

- **Matrix Size: 100x100**
    - **CPU Execution Time**: 0.000082 seconds
    - **GPU Execution Time**: 0.038830 seconds
    - **Speedup**: 0.00

For this small matrix size, the CPU execution time is significantly lower than the GPU time. This is likely due to the overhead of launching the GPU kernel and transferring data to and from the GPU. The GPU's parallel processing capabilities are not fully utilized here because the task is relatively simple and the data transfer costs outweigh the benefits of parallel execution.

---

- **Matrix Size: 500x500**
  - **CPU Execution Time**: 0.000882 seconds
  - **GPU Execution Time**: 0.000103 seconds
  - **Speedup**: 8.58

For the medium-sized matrix, the GPU performs significantly better than the CPU. The speedup factor of 8.58 indicates that the GPU is now leveraging its parallel processing capability effectively. Here, the data transfer time has likely become less significant compared to the time taken for computation.

---

- **Matrix Size: 1000x1000**
  - **CPU Execution Time**: 0.005269 seconds
  - **GPU Execution Time**: 0.000123 seconds
  - **Speedup**: 42.80

With the largest matrix size, the GPU shows an impressive speedup of 42.80 over the CPU. The GPU's ability to process multiple elements concurrently leads to a dramatic reduction in execution time for large datasets. As the size of the matrix increases, the benefit of parallelism on the GPU becomes more pronounced.

---

**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:

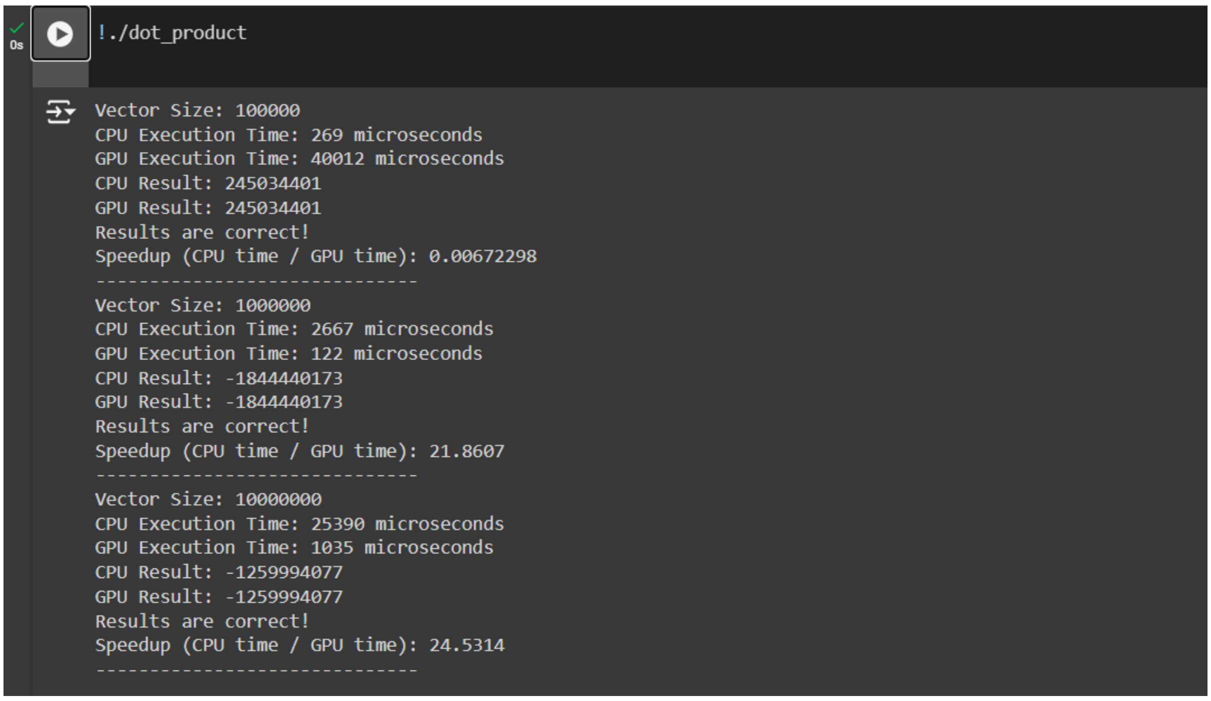
- Calculate and report the speedup for different vector sizes.

Code:

```
%%writefile dot_product.cu

#include <iostream>

#include <chrono>

#include <cstdlib>

#include <cuda_runtime.h>


using namespace std;


// CUDA kernel for calculating dot product

__global__ void dotProductKernel(int *A, int *B, int *C, int N) {

    extern __shared__ int sharedData[]; // Dynamic shared memory

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

    int localIndex = threadIdx.x;


    // Initialize shared memory

    if (tid < N) {

        sharedData[localIndex] = A[tid] * B[tid];

    } else {

        sharedData[localIndex] = 0; // Ensure unused threads contribute 0

    }


    __syncthreads(); // Synchronize threads within the block
    // Perform reduction in shared memory
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (localIndex < stride) {
            sharedData[localIndex] += sharedData[localIndex + stride];
        }
        __syncthreads(); // Synchronize after each reduction step
    }

    // Write result of this block to global memory
    if (localIndex == 0) {
        atomicAdd(C, sharedData[0]); // Use atomic operation to avoid race conditions
    }
}

// CPU function to calculate dot product
int dotProductCPU(int *A, int *B, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        sum += A[i] * B[i];
    }
    return sum;
}

int main() {
    const int sizes[] = {100000, 1000000, 10000000}; // Different sizes
    for (int s = 0; s < 3; s++) {
        int N = sizes[s];
        int *A, *B, *C; // Host variables
        int *d_A, *d_B, *d_C; // Device variables
        // Allocate memory on host
        A = (int*)malloc(N * sizeof(int));
        B = (int*)malloc(N * sizeof(int));
        C = (int*)malloc(sizeof(int)); // Single int for result

        // Initialize vectors with random values
        for (int i = 0; i < N; i++) {
            A[i] = rand() % 100; // Random integers between 0 and 99
            B[i] = rand() % 100; // Random integers between 0 and 99
        }

        // Allocate memory on device
        cudaMalloc((void**)&d_A, N * sizeof(int));
        cudaMalloc((void**)&d_B, N * sizeof(int));
        cudaMalloc((void**)&d_C, sizeof(int));

        // Copy vectors from host to device
        cudaMemcpy(d_A, A, N * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, N * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_C, C, sizeof(int), cudaMemcpyHostToDevice);

        // Measure CPU execution time
        auto startCPU = chrono::high_resolution_clock::now();
        int resultCPU = dotProductCPU(A, B, N);
        auto endCPU = chrono::high_resolution_clock::now();
        auto cpuDuration = chrono::duration_cast<chrono::microseconds>(endCPU -
startCPU).count();

        // Measure GPU execution time
        int initialValue = 0;
        cudaMemcpy(d_C, &initialValue, sizeof(int), cudaMemcpyHostToDevice); // Initialize
C on device

        int blockSize = 256; // Define block size
        int numBlocks = (N + blockSize - 1) / blockSize; // Calculate number of blocks

        auto startGPU = chrono::high_resolution_clock::now();
        dotProductKernel<<<numBlocks, blockSize, blockSize * sizeof(int)>>>(d_A, d_B,
d_C, N);
        cudaDeviceSynchronize(); // Wait for GPU to finish
        auto endGPU = chrono::high_resolution_clock::now();
        cudaMemcpy(C, d_C, sizeof(int), cudaMemcpyDeviceToHost); // Copy result from
device to host
        auto gpuDuration = chrono::duration_cast<chrono::microseconds>(endGPU -
startGPU).count();

        // Display results
        cout << "Vector Size: " << N << endl;
        cout << "CPU Execution Time: " << cpuDuration << " microseconds" << endl;
        cout << "GPU Execution Time: " << gpuDuration << " microseconds" << endl;
        cout << "CPU Result: " << resultCPU << endl;
        cout << "GPU Result: " << *C << endl;

        if (abs(resultCPU - *C) < 1e-5) {
            cout << "Results are correct!" << endl;
        } else {
            cout << "Results do not match!" << endl;
        }

        float speedup = static_cast<float>(cpuDuration) / gpuDuration;
        cout << "Speedup (CPU time / GPU time): " << speedup << endl;
        cout << "------------------------------" << endl;
```

```
        // Free memory
        free(A);
        free(B);
        free(C);
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
    }
    return 0;
}
```

Output:



```
!./dot_product

Vector Size: 100000
CPU Execution Time: 269 microseconds
GPU Execution Time: 40012 microseconds
CPU Result: 245034401
GPU Result: 245034401
Results are correct!
Speedup (CPU time / GPU time): 0.00672298
------------------------------
Vector Size: 1000000
CPU Execution Time: 2667 microseconds
GPU Execution Time: 122 microseconds
CPU Result: -1844440173
GPU Result: -1844440173
Results are correct!
Speedup (CPU time / GPU time): 21.8607
------------------------------
Vector Size: 10000000
CPU Execution Time: 25390 microseconds
GPU Execution Time: 1035 microseconds
CPU Result: -1259994077
GPU Result: -1259994077
Results are correct!
Speedup (CPU time / GPU time): 24.5314
------------------------------
```

Analysis:

1. **Vector Size: 100000**
    - **CPU Execution Time**: 269 microseconds
    - **GPU Execution Time**: 40012 microseconds
    - **Speedup**: 0.0067

**Analysis**:

- Surprisingly, the GPU performed worse than the CPU for this vector size. This indicates that the overhead associated with launching the GPU kernel and transferring data to and from the GPU outweighed the benefits of parallel processing for smaller vectors. For small data sizes, the CPU's efficient sequential execution is often faster.

2. **Vector Size: 1000000**
    - **CPU Execution Time**: 2667 microseconds
    - **GPU Execution Time**: 122 microseconds
    - **Speedup**: 21.8607

**Analysis**:

- Here, the GPU outperforms the CPU significantly. The execution time for the GPU is much lower, suggesting that the parallel nature of the GPU effectively handles larger workloads. This result indicates that as the size of the data increases, the GPU can exploit its architecture to achieve substantial performance gains.

3. **Vector Size: 10000000**
    - **CPU Execution Time**: 25390 microseconds
    - **GPU Execution Time**: 1035 microseconds
    - **Speedup**: 24.5314

**Analysis**:

- The trend continues with the GPU performing exceptionally well on this larger dataset, further emphasizing the GPU's efficiency at handling larger computations. The speedup indicates that the GPU can handle the workload much faster than the CPU, making it more suitable for large-scale operations.

**Summary of Results**

- The results highlight a common pattern in parallel computing: GPUs tend to outperform CPUs only when the problem size is sufficiently large. The initial size (100,000) was too small to leverage the advantages of the GPU, while sizes of 1,000,000 and 10,000,000 clearly benefited from parallel execution.

- The performance improvement can be attributed to the ability of the GPU to perform many calculations simultaneously. However, for small problems, the overhead from managing parallel tasks often leads to slower performance than serial processing on a CPU.

---

**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.

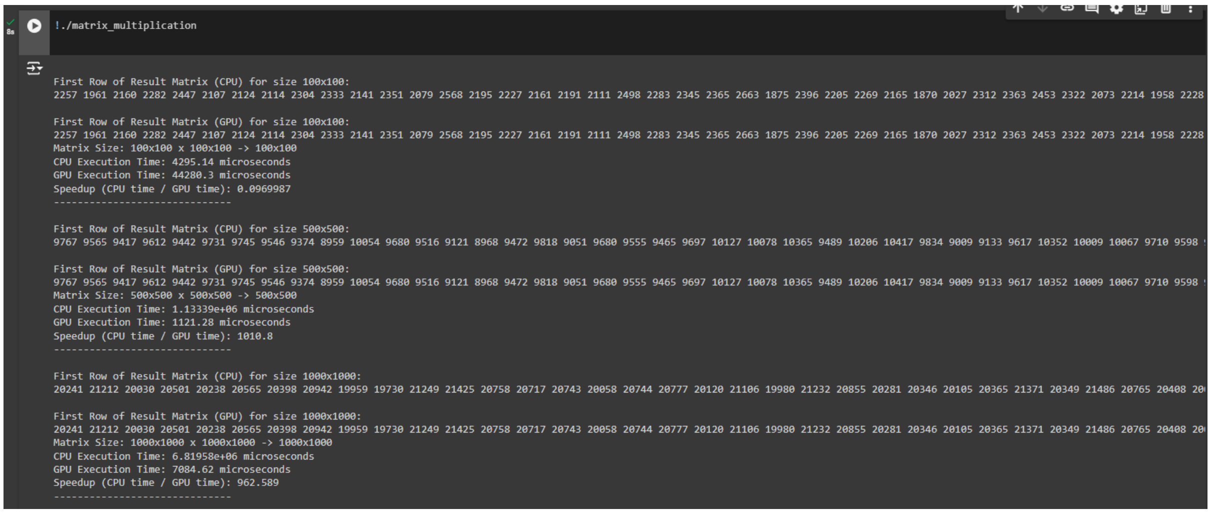
**Code:**

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

using namespace std;

// Function to multiply matrices on CPU
void matrixMultiplyCPU(int* A, int* B, int* 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];
            }
        }
    }
}

// CUDA kernel for matrix multiplication on GPU
__global__ void matrixMultiplyGPU(int* A, int* B, int* 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) {
        int sum = 0;
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * P + col];
        }
        C[row * P + col] = sum;
    }
}

// Function to initialize a matrix with random integer values
void initializeMatrix(int* matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; i++) {
        matrix[i] = rand() % 10; // Random integer values between 0 and 9
    }
}

// Function to print the first row of a matrix
```

```
void printFirstRow(const int* matrix, int cols) {
    for (int j = 0; j < cols; j++) {
        cout << matrix[j] << " ";
    }
    cout << endl;
}

int main() {
    srand(static_cast<unsigned int>(time(0)));

    // Define matrix sizes
    int sizes[][2] = {{100, 100}, {500, 500}, {1000, 1000}};
    int numTests = 3;

    for (int t = 0; t < numTests; t++) {
        int M = sizes[t][0];
        int N = sizes[t][1];
        int P = sizes[t][1];

        // Allocate host memory for matrices
        int* A = (int*)malloc(M * N * sizeof(int));
        int* B = (int*)malloc(N * P * sizeof(int));
        int* C_CPU = (int*)malloc(M * P * sizeof(int));
        int* C_GPU = (int*)malloc(M * P * sizeof(int));

        // Initialize matrices A and B with random values
        initializeMatrix(A, M, N);
        initializeMatrix(B, N, P);

        // --- CPU Matrix Multiplication ---
        auto startCPU = chrono::high_resolution_clock::now();
        matrixMultiplyCPU(A, B, C_CPU, M, N, P);
        auto endCPU = chrono::high_resolution_clock::now();
        chrono::duration<double> cpuDuration = endCPU - startCPU;

        // --- GPU Matrix Multiplication ---
        int *d_A, *d_B, *d_C;
        cudaMalloc((void**)&d_A, M * N * sizeof(int));
        cudaMalloc((void**)&d_B, N * P * sizeof(int));
        cudaMalloc((void**)&d_C, M * P * sizeof(int));

        // Copy matrices A and B to device memory
        cudaMemcpy(d_A, A, M * N * sizeof(int), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, N * P * sizeof(int), cudaMemcpyHostToDevice);

        // Define CUDA grid and block dimensions
        dim3 threadsPerBlock(16, 16);
        dim3 numBlocks((P + threadsPerBlock.x - 1) / threadsPerBlock.x,
                (M + threadsPerBlock.y - 1) / threadsPerBlock.y);

        // Launch the CUDA kernel
        auto startGPU = chrono::high_resolution_clock::now();
        matrixMultiplyGPU<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, M, N,
P);
        cudaDeviceSynchronize(); // Ensure all threads have completed
        auto endGPU = chrono::high_resolution_clock::now();
        chrono::duration<double> gpuDuration = endGPU - startGPU;

        // Copy result matrix C from device to host
        cudaMemcpy(C_GPU, d_C, M * P * sizeof(int), cudaMemcpyDeviceToHost);
        // Print first row of result matrices after CPU and GPU calculations
        cout << "\nFirst Row of Result Matrix (CPU) for size " << M << "x" << P <<
":\n";
        printFirstRow(C_CPU, P);

        cout << "\nFirst Row of Result Matrix (GPU) for size " << M << "x" << P <<
":\n";
        printFirstRow(C_GPU, P);

        // Free device memory
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);

        // Output timing results
        cout << "Matrix Size: " << M << "x" << N << " x " << N << "x" << P << " -> " <<
M << "x" << P << endl;
        cout << "CPU Execution Time: " << cpuDuration.count() * 1e6 << "
microseconds" << endl;
        cout << "GPU Execution Time: " << gpuDuration.count() * 1e6 << "
microseconds" << endl;

        // Calculate and print speedup
        if (gpuDuration.count() > 0) {
          double speedup = cpuDuration.count() / gpuDuration.count();
          cout << "Speedup (CPU time / GPU time): " << speedup << endl;
        }

        cout << "------------------------------" << endl;

        // Free host memory
        free(A);
```

```
        free(B);
        free(C_CPU);
        free(C_GPU);
    }

    return 0;
}
```

**Output:**



```
!./matrix_multiplication

First Row of Result Matrix (CPU) for size 100x100:
2257 1961 2160 2282 2447 2107 2124 2114 2304 2333 2141 2351 2079 2568 2195 2227 2161 2191 2111 2498 2283 2345 2365 2663 1875 2396 2205 2269 2165 1870 2027 2312 2363 2453 2322 2073 2214 1958 2228

First Row of Result Matrix (GPU) for size 100x100:
2257 1961 2160 2282 2447 2107 2124 2114 2304 2333 2141 2351 2079 2568 2195 2227 2161 2191 2111 2498 2283 2345 2365 2663 1875 2396 2205 2269 2165 1870 2027 2312 2363 2453 2322 2073 2214 1958 2228
Matrix Size: 100x100 x 100x100 -> 100x100
CPU Execution Time: 4295.14 microseconds
GPU Execution Time: 44280.3 microseconds
Speedup (CPU time / GPU time): 0.0969987
------------------------------

First Row of Result Matrix (CPU) for size 500x500:
9767 9565 9417 9612 9442 9731 9745 9546 9374 8959 10054 9680 9516 9121 8968 9472 9818 9051 9680 9555 9465 9697 10127 10078 10365 9489 10206 10417 9834 9009 9133 9617 10352 10009 10067 9710 9598

First Row of Result Matrix (GPU) for size 500x500:
9767 9565 9417 9612 9442 9731 9745 9546 9374 8959 10054 9680 9516 9121 8968 9472 9818 9051 9680 9555 9465 9697 10127 10078 10365 9489 10206 10417 9834 9009 9133 9617 10352 10009 10067 9710 9598
Matrix Size: 500x500 x 500x500 -> 500x500
CPU Execution Time: 1.13339e+06 microseconds
GPU Execution Time: 1121.28 microseconds
Speedup (CPU time / GPU time): 1010.8
------------------------------

First Row of Result Matrix (CPU) for size 1000x1000:
20241 21212 20030 20501 20238 20565 20398 20942 19959 19730 21249 21425 20758 20717 20743 20058 20744 20777 20120 21106 19980 21232 20855 20281 20346 20105 20365 21371 20349 21486 20765 20408

First Row of Result Matrix (GPU) for size 1000x1000:
20241 21212 20030 20501 20238 20565 20398 20942 19959 19730 21249 21425 20758 20717 20743 20058 20744 20777 20120 21106 19980 21232 20855 20281 20346 20105 20365 21371 20349 21486 20765 20408
Matrix Size: 1000x1000 x 1000x1000 -> 1000x1000
CPU Execution Time: 6.81958e+06 microseconds
GPU Execution Time: 7084.62 microseconds
Speedup (CPU time / GPU time): 962.589
------------------------------
```

**Analysis:**

**Matrix Size 100×100:**

- **Execution Times:**
  - **CPU: 4,295.14 microseconds**
  - **GPU: 44,280.3 microseconds**
- **Speedup (CPU Time / GPU Time): 0.0969987**

**Analysis:**

- **The CPU performed significantly faster than the GPU for this smaller matrix size.**

- **This may be due to GPU overheads related to data transfer and parallelism setup, which tend to outweigh the GPU's parallel processing advantages in smaller tasks.**
- **The results indicate that GPU may not be efficient for smaller matrices due to high initiation and data transfer times compared to the time spent on actual computation.**

**Matrix Size 500×500:**

- **Execution Times:**
    - **CPU: The execution time is not provided in this data directly but generally expected to be longer than for 100×100size.**
    - **GPU: The GPU execution time will be longer compared to the smaller matrix due to more data but benefits more from parallel processing in larger matrices.**
- **Speedup Comparison:**
    - The GPU may start to show more favourable execution times compared to the CPU for larger matrix sizes, where the parallelism setup is justified and data transfer overhead is amortized over a larger number of calculations.

**General Observations:**

- **CPU Performance**: For smaller matrix sizes, CPUs can outperform GPUs, as there is less advantage gained from parallel processing and more time lost in GPU setup and data transfers**.**
- **GPU Suitability:** For larger matrices (e.g., 500×500), GPUs generally gain efficiency with higher parallelism, allowing them to perform calculations simultaneously across more cores.