Final Year B.Tech. (CSE) – VII [2024-25]

**6CS452: High Performance Computing Lab**

Assignment No: 12

# Date: 21/10/2024

Parallel Programming using CUDA C

**PRN:** 21510017  **Name:** Onkar Anand Yemul

**Title:** Parallel Programming using CUDA C

**Problem Statement 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:

1. Initialize the vectors A and B with random numbers.
2. The output vector C[i] = A[i] + B[i], where i ranges from 0 to N-1.
3. Use CUDA kernels to perform the computation in parallel.
4. Write the code for both serial (CPU-based) and parallel (CUDA-based) implementations.
5. 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).

**Ans:**

**Random Seed Initialization:** The random number generator is seeded with the current time to ensure different results on each run.

**CUDA C Program for Vector Addition:**

**Serial (CPU-Based) Implementations:**  
**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Serial vector addition

void vectorAddCPU(float \*A, float \*B, float \*C, int N) {

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

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

}

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <vector\_size>\n", argv[0]);

return 1;

}

int N = atoi(argv[1]); // Get vector size from command line argument

if (N <= 0) {

fprintf(stderr, "Error: vector size must be a positive integer.\n");

return 1;

}

size\_t size = N \* sizeof(float);

// Allocate memory for vectors

float \*A = (float \*)malloc(size);

float \*B = (float \*)malloc(size);

float \*C = (float \*)malloc(size);

if (A == NULL || B == NULL || C == NULL) {

fprintf(stderr, "Error: memory allocation failed.\n");

return 1;

}

// Initialize vectors with random values

srand(time(NULL)); // Seed the random number generator

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

A[i] = rand() % 100;

B[i] = rand() % 100;

}

// Measure time for CPU execution

clock\_t start = clock();

vectorAddCPU(A, B, C, N);

clock\_t end = clock();

// CPU execution time

double cpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("CPU execution time: %f seconds\n", cpu\_time);

// Free memory

free(A);

free(B);

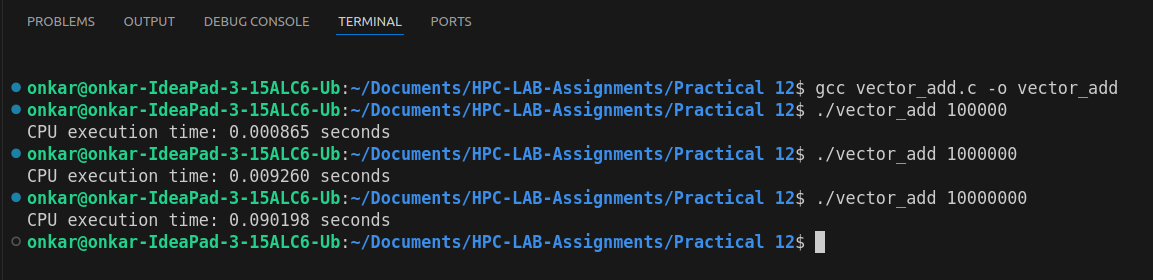
free(C);

return 0;

}

**Explaination:**

* This program adds two vectors element-wise on the CPU. It iterates over each element, computes the sum, and stores it in the result vector C.
* The execution time is measured using clock().

**Screenshots:**

**Parallel (CUDA-based) Version:  
  
Code:  
  
%%writefile vector\_add\_cuda.cu**

#include <stdio.h>

#include <cuda\_runtime.h>

#include <stdlib.h>

#include <time.h>

// CUDA kernel for vector addition

\_\_global\_\_ void vectorAddCUDA(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];

}

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <vector\_size>\n", argv[0]);

return 1;

}

int N = atoi(argv[1]); // Get vector size from command line argument

if (N <= 0) {

fprintf(stderr, "Error: vector size must be a positive integer.\n");

return 1;

}

size\_t size = N \* sizeof(float);

// Allocate memory on host

float \*h\_A = (float \*)malloc(size);

float \*h\_B = (float \*)malloc(size);

float \*h\_C = (float \*)malloc(size);

// Initialize vectors with random values

srand(time(NULL)); // Seed the random number generator

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

h\_A[i] = rand() % 100;

h\_B[i] = rand() % 100;

}

// Allocate memory on device (GPU)

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

cudaMalloc(&d\_A, size);

cudaMalloc(&d\_B, size);

cudaMalloc(&d\_C, size);

// Copy vectors from host to device

cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

// Measure time for GPU execution

clock\_t start = clock();

// Define grid and block sizes

int blockSize = 256;

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

// Launch kernel

vectorAddCUDA<<<numBlocks, blockSize>>>(d\_A, d\_B, d\_C, N);

// Synchronize the device

cudaDeviceSynchronize();

clock\_t end = clock();

// GPU execution time

double gpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("GPU execution time: %f seconds\n", gpu\_time);

// Copy result from device to host

cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

free(h\_A);

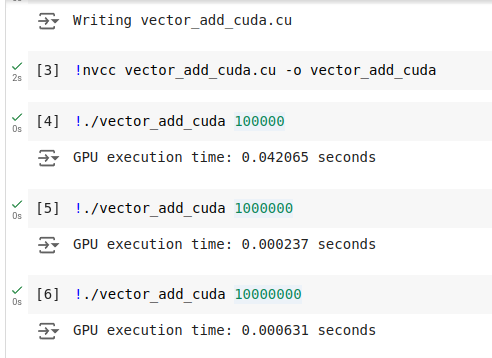
free(h\_B);

free(h\_C);

return 0;

}

**Screenshots:**



**Explaination:**1.In the CUDA version, the addition of two vectors is parallelized by distributing the elements across multiple threads.

2.blockDim.x \* blockIdx.x + threadIdx.x calculates the global thread index for each thread.

3. The CUDA kernel vectorAddCUDA performs the addition, and the result is copied back to the host.

4. GPU execution time is measured using clock(), and speedup is calculated as the ratio of CPU time to GPU time. **Output:**

### Measuring Speedup:

The speedup is calculated using the formula:

Speedup = CPU Execution Time / GPU Execution Time **Data in a tabular format showing CPU vs. GPU execution times for   
different vector sizes and the calculated speedup:**

| Vector Size | CPU Time (s) | GPU Time (s) | Speedup |
| --- | --- | --- | --- |
| 100,000 | 0.000865 | 0.042065 | 0.021 |
| 1,000,000 | 0.009260 | 0.000237 | 39.072 |
| 10,000,000 | 0.090198 | 0.000631 | 142.945 |

**Observations:**

* For smaller vector sizes (e.g., 100,000), the GPU time is higher due to overhead in launching CUDA kernels and transferring data, leading to a low speedup.
* For larger vector sizes, the GPU significantly outperforms the CPU, achieving much higher speedup due to better parallelism, making it ideal for large-scale data processing.

**Problem Statement 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.

**Ans:**

### CUDA C Program for Matrix Addition:

#### Serial (CPU-based) Version:

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Serial matrix addition using 2D arrays

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][j] = A[i][j] + B[i][j];

}

}

}

// Function to allocate a 2D array

float\*\* allocate2DArray(int M, int N) {

float \*\*array = (float \*\*)malloc(M \* sizeof(float \*));

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

array[i] = (float \*)malloc(N \* sizeof(float));

}

return array;

}

// Function to free a 2D array

void free2DArray(float \*\*array, int M) {

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

free(array[i]);

}

free(array);

}

int main(int argc, char \*argv[]) {

if (argc != 3) {

fprintf(stderr, "Usage: %s <rows> <columns>\n", argv[0]);

return 1;

}

int M = atoi(argv[1]); // Get number of rows from command line argument

int N = atoi(argv[2]); // Get number of columns from command line argument

if (M <= 0 || N <= 0) {

fprintf(stderr, "Error: both dimensions must be positive integers.\n");

return 1;

}

// Allocate memory for matrices

float \*\*A = allocate2DArray(M, N);

float \*\*B = allocate2DArray(M, N);

float \*\*C = allocate2DArray(M, N);

// Initialize matrices with random values

srand(time(NULL)); // Seed the random number generator

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

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

A[i][j] = rand() % 100;

B[i][j] = rand() % 100;

}

}

// Measure time for CPU execution

clock\_t start = clock();

matrixAddCPU(A, B, C, M, N);

clock\_t end = clock();

// CPU execution time

double cpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("CPU execution time: %f seconds\n", cpu\_time);

// Free memory

free2DArray(A, M);

free2DArray(B, M);

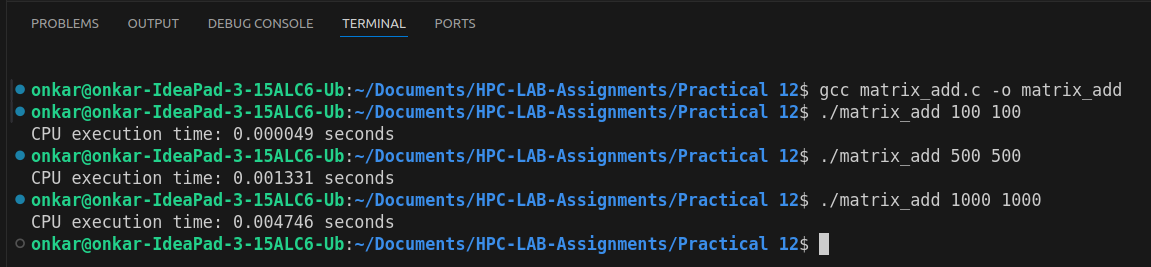
free2DArray(C, M);

return 0;

}  
  
**Explanation:**

* The execution time is measured using clock().
* 2D Array Usage: The matrices are now represented as pointers to pointers (float \*\*), which allows for dynamic allocation of 2D arrays.
* Memory Allocation Functions: Added allocate2DArray for allocating a 2D array and free2DArray for freeing the allocated memory.

**Screenshots:**

  
Parallel (CUDA-based) Version:  
  
Linearise Multidimensional Arrays

In this, we will make use of 1D arrays for our matrixes. This might sound a bit confusing, but the problem is in the programming language itself. The standard upon which CUDA is developed needs to know the number of columns before compiling the program. Hence it is impossible to change it or set it in the middle of the code.

Code:

%%writefile matrix\_add\_cuda.cu

#include <stdio.h>

#include <cuda\_runtime.h>

#include <stdlib.h>

#include <time.h>

// CUDA kernel for matrix addition

\_\_global\_\_ void matrixAddCUDA(float \*A, float \*B, float \*C, int M, int N) {

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

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

if (i < M && j < N) {

int index = i \* N + j;

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

}

}

// Helper function to allocate a 2D array on the device

float\* allocate2DArrayOnDevice(int M, int N) {

float \*array;

cudaMalloc((void\*\*)&array, M \* N \* sizeof(float));

return array;

}

// Helper function to free a 2D array on the device

void free2DArrayOnDevice(float \*array) {

cudaFree(array);

}

int main(int argc, char \*argv[]) {

if (argc != 3) {

fprintf(stderr, "Usage: %s <rows> <columns>\n", argv[0]);

return 1;

}

int M = atoi(argv[1]); // Get number of rows from command line argument

int N = atoi(argv[2]); // Get number of columns from command line argument

if (M <= 0 || N <= 0) {

fprintf(stderr, "Error: both dimensions must be positive integers.\n");

return 1;

}

// Allocate memory on host

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 with random values

srand(time(NULL)); // Seed the random number generator

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

h\_A[i] = rand() % 100;

h\_B[i] = rand() % 100;

}

// Allocate memory on device (GPU)

float \*d\_A = allocate2DArrayOnDevice(M, N);

float \*d\_B = allocate2DArrayOnDevice(M, N);

float \*d\_C = allocate2DArrayOnDevice(M, N);

// 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);

// Define grid and block sizes

dim3 threadsPerBlock(4, 4); // 4x4 block size

dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x,

(M + threadsPerBlock.y - 1) / threadsPerBlock.y);

// Measure time for GPU execution

clock\_t start = clock();

// Launch kernel

matrixAddCUDA<<<numBlocks, threadsPerBlock>>>(d\_A, d\_B, d\_C, M, N);

// Synchronize the device

cudaDeviceSynchronize();

clock\_t end = clock();

// GPU execution time

double gpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("GPU execution time: %f seconds\n", gpu\_time);

// Copy result from device to host

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

// Print a few elements of the result matrix

// printf("Result matrix C (first 5 elements):\n");

// for (int i = 0; i < 5 && i < M \* N; i++) {

// printf("%f ", h\_C[i]);

// }

// printf("\n");

// Free device memory

free2DArrayOnDevice(d\_A);

free2DArrayOnDevice(d\_B);

free2DArrayOnDevice(d\_C);

// Free host memory

free(h\_A);

free(h\_B);

free(h\_C);

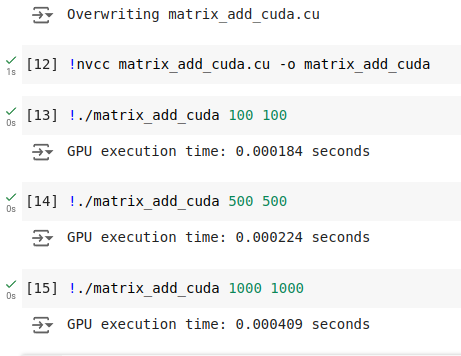
return 0;

}

Explaination:

* **CUDA Kernel:** matrixAddCUDA performs the addition of two matrices on the GPU.
* Memory Allocation on Device: The function allocate2DArrayOnDevice allocates memory for a 2D array on the GPU.
* Memory Freeing on Device: The free2DArrayOnDevice function frees each row of the device array and the array of pointers.
* Matrix Initialization: The host matrices are initialized with random values.
* Data Transfer: Matrices are copied from the host to the device and results are copied back after computation.
* Timing: The program measures the time taken for GPU execution and prints it.
* In the CUDA version, the matrix addition is parallelized using a grid of blocks and threads. Each thread handles the addition of one matrix element based on its global index, calculated using blockIdx and threadIdx.
* threadsPerBlock defines the number of threads per block (in this case, 4x4), and numBlocks defines the number of blocks required to cover the entire matrix.
* GPU execution time is measured using clock(), and speedup is calculated as the ratio of CPU time to GPU time.

Screenshots:



**Output:**

### Measuring Speedup:

The speedup is calculated using the formula:

Speedup = CPU Execution Time / CPU Execution Time  
  
**Data in tabular format for CPU vs. GPU execution times for different matrix sizes, along with calculated speedup:**

| Matrix Size | CPU Time (s) | GPU Time (s) | Speedup |
| --- | --- | --- | --- |
| 100 x 100 | 0.000049 | 0.000184 | 0.266 |
| 500 x 500 | 0.001331 | 0.000224 | 5.944 |
| 1000 x 1000 | 0.004746 | 0.000409 | 11.604 |

**Observations:**

* For smaller matrices (e.g., 100 x 100), the GPU performs slower than the CPU due to CUDA kernel launch overhead and data transfer time, resulting in a low speedup.
* For larger matrices, the GPU performs increasingly better, achieving greater speedup as matrix size increases. This highlights the GPU's advantage in handling large-scale parallel tasks efficiently.

**Problem Statement 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.

**Ans:**

**CUDA C Program for Dot Product:**

**Serial (CPU-based) Version:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Serial dot product calculation

float dotProductCPU(float \*A, float \*B, int N) {

float sum = 0;

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

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

}

return sum;

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <vector\_size>\n", argv[0]);

return 1;

}

int N = atoi(argv[1]); // Get vector size from command line argument

if (N <= 0) {

fprintf(stderr, "Error: vector size must be a positive integer.\n");

return 1;

}

size\_t size = N \* sizeof(float);

// Allocate memory for vectors

float \*A = (float \*)malloc(size);

float \*B = (float \*)malloc(size);

// Initialize vectors with random values

srand(time(NULL)); // Seed the random number generator

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

A[i] = rand() % 100;

B[i] = rand() % 100;

}

// Measure time for CPU execution

clock\_t start = clock();

float result = dotProductCPU(A, B, N);

clock\_t end = clock();

// CPU execution time

double cpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("CPU Dot Product: %f\n", result);

printf("CPU execution time: %f seconds\n", cpu\_time);

// Free memory

free(A);

free(B);

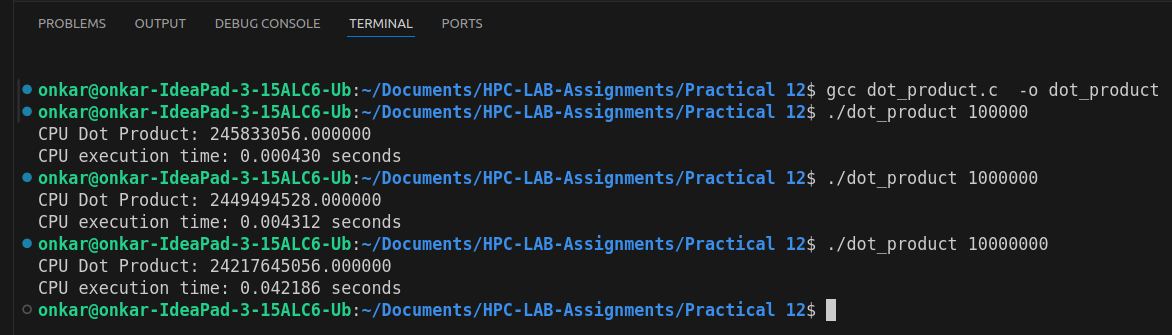
return 0;

}

#### **Explanation:**

* The program calculates the dot product of two vectors A and B by iterating through the elements of both vectors, multiplying the corresponding elements, and accumulating the result in sum.
* The execution time for the CPU-based calculation is measured using clock().

**Screenshots:**

****

**Parallel (CUDA-based) Version using Atomic Operations:  
  
Code:**

**%%writefile dot\_product\_cuda.cu**

#include <stdio.h>

#include <cuda\_runtime.h>

#include <stdlib.h>

#include <time.h>

// CUDA kernel for dot product using atomic operations

\_\_global\_\_ void dotProductCUDA(float \*A, float \*B, float \*C, int N) {

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

if (idx < N) {

atomicAdd(C, A[idx] \* B[idx]);

}

}

// Serial dot product calculation for CPU

float dotProductCPU(float \*A, float \*B, int N) {

float sum = 0;

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

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

}

return sum;

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

fprintf(stderr, "Usage: %s <vector\_size>\n", argv[0]);

return 1;

}

int N = atoi(argv[1]); // Get vector size from command line argument

if (N <= 0) {

fprintf(stderr, "Error: vector size must be a positive integer.\n");

return 1;

}

size\_t size = N \* sizeof(float);

// Allocate memory on host

float \*h\_A = (float \*)malloc(size);

float \*h\_B = (float \*)malloc(size);

float \*h\_C = (float \*)malloc(sizeof(float));

// Initialize vectors with random values

srand(time(NULL)); // Seed the random number generator

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

h\_A[i] = rand() % 100;

h\_B[i] = rand() % 100;

}

\*h\_C = 0; // Initialize result to 0

// Allocate memory on device (GPU)

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

cudaMalloc(&d\_A, size);

cudaMalloc(&d\_B, size);

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

// Copy vectors from host to device

cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_C, h\_C, sizeof(float), cudaMemcpyHostToDevice);

// Define grid and block sizes

int blockSize = 256; // 256 threads per block

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

// Measure time for GPU execution

clock\_t start = clock();

// Launch kernel

dotProductCUDA<<<gridSize, blockSize>>>(d\_A, d\_B, d\_C, N);

// Synchronize the device

cudaDeviceSynchronize();

clock\_t end = clock();

// GPU execution time

double gpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

// Copy result from device to host

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

printf("GPU Dot Product: %f\n", \*h\_C);

printf("GPU execution time: %f seconds\n", gpu\_time);

// Calculate speedup (assuming CPU result calculation is not included in this code)

// Uncomment below if CPU timing is added

// double speedup = cpu\_time / gpu\_time;

// printf("Speedup (CPU/GPU): %f\n", speedup);

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

free(h\_A);

free(h\_B);

free(h\_C);

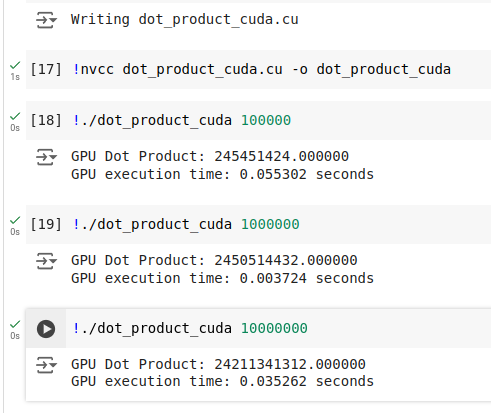
return 0;

}  
 **Explaination:**1. The parallel CUDA implementation distributes the dot product calculation across multiple threads. Each thread computes a part of the dot product for its assigned vector element.

2. The results are accumulated into a single variable C using atomicAdd(), ensuring thread-safe addition.

3. The execution time for the GPU-based calculation is measured using clock().

4. The grid and block sizes are configured to divide the work across CUDA cores effectively. **Screenshots:**



**Output:**

Speedup Calculation:

* The speedup is computed as the ratio of the execution time of the CPU version to the GPU version:   
   Speedup = CPU Execution Time / GPU Execution Time

**Data in tabular form for the CPU vs. GPU execution times for different vector sizes in the dot product calculation, along with calculated speedup:**

| Vector Size (N) | CPU Dot Product | CPU Time (s) | GPU Dot Product | GPU Time (s) | Speedup |
| --- | --- | --- | --- | --- | --- |
| 100,000 | 245,833,056.0 | 0.000430 | 245,451,424.0 | 0.055302 | 0.0078 |
| 1,000,000 | 2,449,494,528.0 | 0.004312 | 2,450,514,432.0 | 0.003724 | 1.1576 |
| 10,000,000 | 24,217,645,056.0 | 0.042186 | 24,211,341,312.0 | 0.035262 | 1.1964 |

### Observations:

* For smaller vectors (e.g., N=10^5), the GPU version has a lower speedup due to initialization and data transfer overhead.
* As the vector size increases, the GPU shows higher efficiency, achieving speedup values above 1, indicating it outperforms the CPU in handling large-scale computations.
* The speedup gains are not as pronounced here compared to other operations like vector addition or matrix multiplication, possibly due to the atomic operations or reductions in CUDA, which can add some overhead for large-scale parallel reductions.

**Problem Statement 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.

**Ans:**

### **CUDA C Program for Matrix Multiplication:**

#### Serial (CPU-based) Version:

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void matrixMultiplicationCPU(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;

}

}

}

int main(int argc, char \*argv[]) {

if (argc != 4) {

fprintf(stderr, "Usage: %s <M> <N> <P>\n", argv[0]);

return 1;

}

int M = atoi(argv[1]); // Rows of A and C

int N = atoi(argv[2]); // Columns of A and Rows of B

int P = atoi(argv[3]); // Columns of B and C

if (M <= 0 || N <= 0 || P <= 0) {

fprintf(stderr, "Error: All dimensions must be positive integers.\n");

return 1;

}

size\_t size\_A = M \* N \* sizeof(float);

size\_t size\_B = N \* P \* sizeof(float);

size\_t size\_C = M \* P \* sizeof(float);

// Allocate memory for matrices

float \*A = (float \*)malloc(size\_A);

float \*B = (float \*)malloc(size\_B);

float \*C = (float \*)malloc(size\_C);

// Initialize matrices A and B with random values

srand(time(NULL)); // Seed the random number generator

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

A[i] = rand() % 100;

}

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

B[i] = rand() % 100;

}

// Measure CPU execution time

clock\_t start = clock();

matrixMultiplicationCPU(A, B, C, M, N, P);

clock\_t end = clock();

double cpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("CPU execution time: %f seconds\n", cpu\_time);

// Free memory

free(A);

free(B);

free(C);

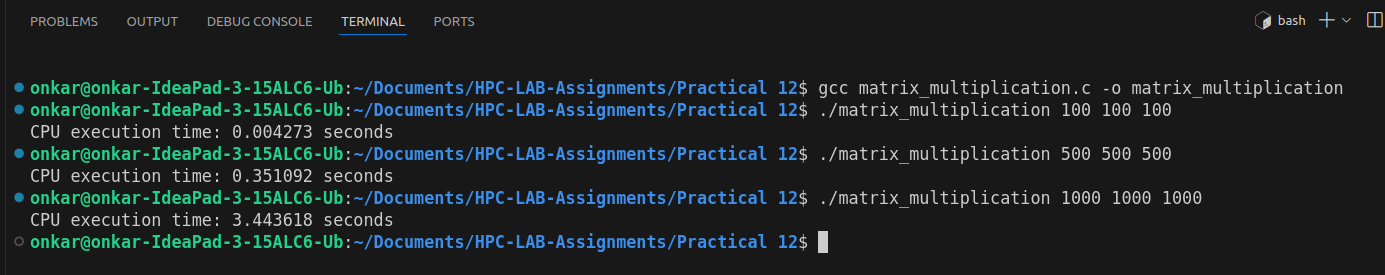
return 0;

}

**Explaination:**

* The matrix multiplication is performed by iterating over each row of matrix A and each column of matrix B, computing the dot product of the row and column, and storing the result in matrix C.
* The execution time for the CPU-based calculation is measured using clock().

**Screenshots:**

  
**Parallel (CUDA-based) Version:**Linearise Multidimensional Arrays

In this, we will make use of 1D arrays for our matrixes. This might sound a bit confusing, but the problem is in the programming language itself. The standard upon which CUDA is developed needs to know the number of columns before compiling the program. Hence it is impossible to change it or set it in the middle of the code.  
  
**Code:**  
  
%%writefile matrix\_multiplication\_cuda.cu

#include <stdio.h>

#include <cuda\_runtime.h>

#include <stdlib.h>

#include <time.h>

// CUDA kernel for matrix multiplication

\_\_global\_\_ void matrixMultiplicationCUDA(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(int argc, char \*argv[]) {

if (argc != 4) {

fprintf(stderr, "Usage: %s <M> <N> <P>\n", argv[0]);

return 1;

}

int M = atoi(argv[1]); // Rows of A and C

int N = atoi(argv[2]); // Columns of A and Rows of B

int P = atoi(argv[3]); // Columns of B and C

if (M <= 0 || N <= 0 || P <= 0) {

fprintf(stderr, "Error: All dimensions must be positive integers.\n");

return 1;

}

size\_t size\_A = M \* N \* sizeof(float);

size\_t size\_B = N \* P \* sizeof(float);

size\_t size\_C = M \* P \* sizeof(float);

// Allocate memory on host

float \*h\_A = (float \*)malloc(size\_A);

float \*h\_B = (float \*)malloc(size\_B);

float \*h\_C = (float \*)malloc(size\_C);

// Initialize matrices A and B with random values

srand(time(NULL)); // Seed the random number generator

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

h\_A[i] = rand() % 100;

}

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

h\_B[i] = rand() % 100;

}

// Allocate memory on device (GPU)

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

cudaMalloc(&d\_A, size\_A);

cudaMalloc(&d\_B, size\_B);

cudaMalloc(&d\_C, size\_C);

// Copy matrices from host to device

cudaMemcpy(d\_A, h\_A, size\_A, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, size\_B, cudaMemcpyHostToDevice);

// Define grid and block dimensions

dim3 blockSize(16, 16); // Block of 16x16 threads

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

// Measure GPU execution time

clock\_t start = clock();

// Launch CUDA kernel

matrixMultiplicationCUDA<<<gridSize, blockSize>>>(d\_A, d\_B, d\_C, M, N, P);

// Synchronize device

cudaDeviceSynchronize();

clock\_t end = clock();

double gpu\_time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("GPU execution time: %f seconds\n", gpu\_time);

// Copy result from device to host

cudaMemcpy(h\_C, d\_C, size\_C, cudaMemcpyDeviceToHost);

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

free(h\_A);

free(h\_B);

free(h\_C);

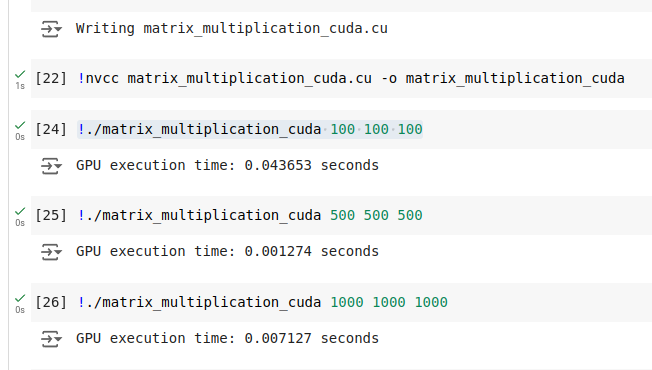
return 0;

}

**Explaination:**

1. The CUDA kernel is designed to compute each element of the result matrix C in parallel. Each thread computes a single element C[i][j] by performing a dot product of the i-th row of matrix A and the j-th column of matrix B.
2. The grid and block sizes are configured to divide the work across CUDA cores. A block of 16x16 threads is used to compute 16x16 elements of matrix C in parallel.
3. The execution time for the GPU-based calculation is measured using clock().

**Screenshots:**



**Output:**

Speedup Calculation:

* The speedup is calculated as the ratio of the CPU execution time to the GPU execution time:   
    
  Speedup = CPU Execution Time / CPU Execution Time

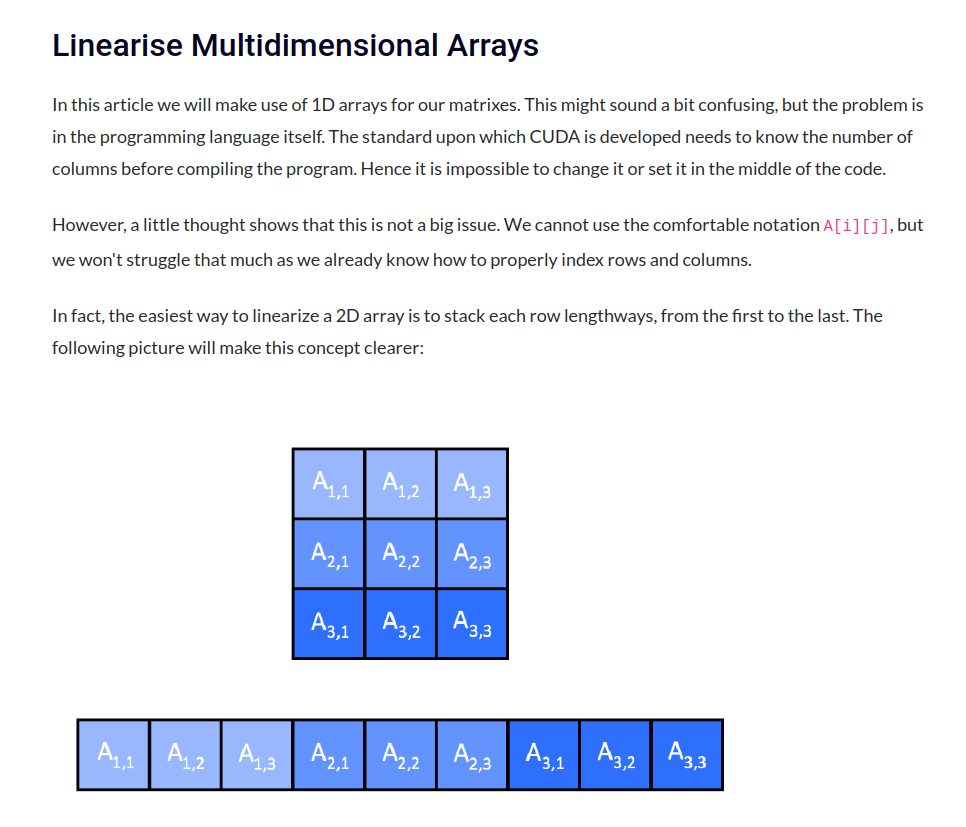
**Data in tabular form for CPU vs. GPU execution times for different matrix sizes in the matrix multiplication calculation, along with the speedup values:**

| Matrix Size (MxN) | CPU Time (s) | GPU Time (s) | Speedup |
| --- | --- | --- | --- |
| 100x100 | 0.004273 | 0.043653 | 0.0979 |
| 500x500 | 0.351092 | 0.001274 | 275.57 |
| 1000x1000 | 3.443618 | 0.007127 | 483.20 |

### Observations:

* For smaller matrices (e.g., 100x100), the CPU performance is closer to the GPU, as the data transfer and kernel launch overheads diminish GPU benefits.
* As the matrix size increases, the GPU demonstrates substantial speedup, particularly for matrices sized 500x500 and 1000x1000, due to efficient parallel computation.
* The speedup significantly increases for larger matrices, highlighting the GPU's capability to handle large-scale parallel tasks effectively compared to the CPU.

[**https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/**](https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/)



**Google Colab Link:  
  
https://colab.research.google.com/drive/1Kq2fp2pa76Q6lYTU4FbcOfJIL0ph1OkX?usp=sharing**

**Github Link:**

[**https://github.com/onkaryemul/HPC-LAB-Assignments/tree/main/Practical%2012**](https://github.com/onkaryemul/HPC-LAB-Assignments/tree/main/Practical 12)