**PP LAB WEEK-10**

# DSE VI-A2 Divansh Prasad 210968140

1) Write a program in CUDA to add two matrices for the following specifications:

• Each row of the resultant matrix to be computed by one thread.

• Each column of resultant matrix to be computed by one thread

• Each element of resultant matrix to be computed by one thread

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_SIZE 10 // Maximum size of the matrices

// Kernel to add two matrices where each row of the resultant matrix is computed by one thread

\_\_global\_\_ void addMatrixRows(int \*A, int \*B, int \*C, int size) {

int row = blockIdx.x;

int col = threadIdx.x;

if (row < size && col < size) {

C[row \* size + col] = A[row \* size + col] + B[row \* size + col];

}

}

// Kernel to add two matrices where each column of the resultant matrix is computed by one thread

\_\_global\_\_ void addMatrixColumns(int \*A, int \*B, int \*C, int size) {

int row = threadIdx.x;

int col = blockIdx.x;

if (row < size && col < size) {

C[row \* size + col] = A[row \* size + col] + B[row \* size + col];

}

}

// Kernel to add two matrices where each element of the resultant matrix is computed by one thread

\_\_global\_\_ void addMatrixElements(int \*A, int \*B, int \*C, int size) {

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

int row = index / size;

int col = index % size;

if (row < size && col < size) {

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

}

}

// Function to print a matrix

void printMatrix(int \*matrix, int size) {

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

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

printf("%d ", matrix[i \* size + j]);

}

printf("\n");

}

}

int main() {

int size;

printf("Enter the size of the matrices (maximum %d): ", MAX\_SIZE);

scanf("%d", &size);

if (size <= 0 || size > MAX\_SIZE) {

printf("Invalid size!\n");

return 1;

}

int \*A, \*B, \*C;

size\_t matrixSize = size \* size \* sizeof(int);

// Allocate memory for matrices on the host

A = (int \*)malloc(matrixSize);

B = (int \*)malloc(matrixSize);

C = (int \*)malloc(matrixSize);

// Initialize random number generator

srand(time(NULL));

// Generate random matrices A and B

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

A[i] = rand() % 10; // Generate a random number between 0 and 9

B[i] = rand() % 10;

}

// Print matrices A and B

printf("Matrix A:\n");

printMatrix(A, size);

printf("\nMatrix B:\n");

printMatrix(B, size);

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

// Allocate memory for matrices on the device

cudaMalloc(&d\_A, matrixSize);

cudaMalloc(&d\_B, matrixSize);

cudaMalloc(&d\_C, matrixSize);

// Copy matrices A and B from host to device

cudaMemcpy(d\_A, A, matrixSize, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, matrixSize, cudaMemcpyHostToDevice);

addMatrixElements<<<(size \* size + 255) / 256, 256>>>(d\_A, d\_B, d\_C, size); // Each element of resultant matrix computed by one thread

// Copy resultant matrix C from device to host

cudaMemcpy(C, d\_C, matrixSize, cudaMemcpyDeviceToHost);

// Print resultant matrix C

printf("\nResultant Matrix:\n");

printMatrix(C, size);

// Free memory allocated on the device

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free memory allocated on the host

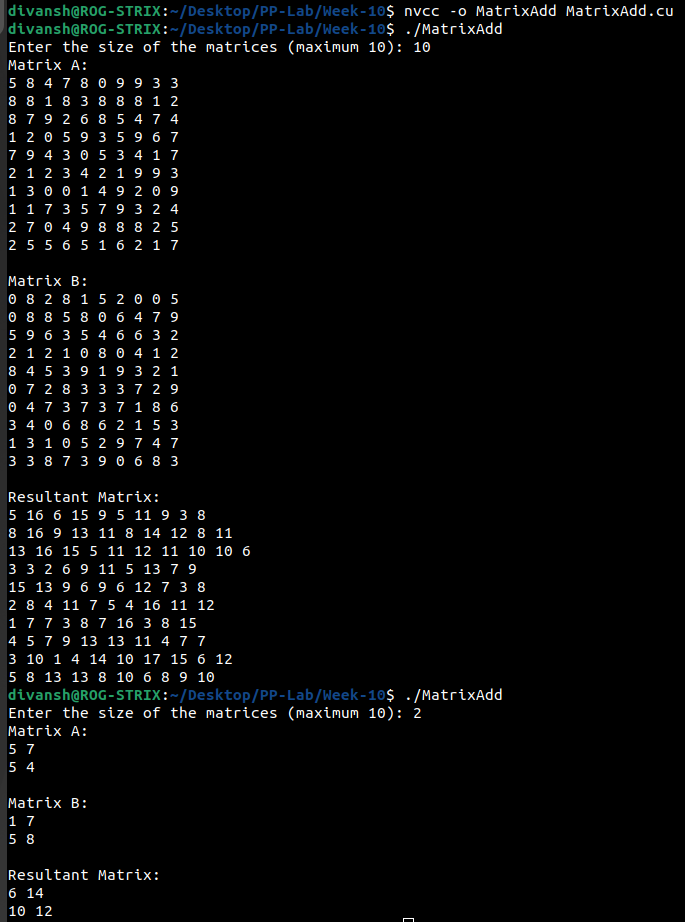
free(A);

free(B);

free(C);

return 0;

}



2) Write a program in CUDA to multiply two matrices for the following specifications:

• Each row of the resultant matrix to be computed by one thread.

• Each column of resultant matrix to be computed by one thread

• Each element of resultant matrix to be computed by one thread

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_SIZE 10 // Maximum size of the matrices

// Kernel to multiply two matrices where each row of the resultant matrix is computed by one thread

\_\_global\_\_ void multiplyMatrixRows(int \*A, int \*B, int \*C, int size) {

int row = blockIdx.x;

int col = threadIdx.x;

if (row < size && col < size) {

int sum = 0;

for (int k = 0; k < size; ++k) {

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

}

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

}

}

// Kernel to multiply two matrices where each column of the resultant matrix is computed by one thread

\_\_global\_\_ void multiplyMatrixColumns(int \*A, int \*B, int \*C, int size) {

int row = threadIdx.x;

int col = blockIdx.x;

if (row < size && col < size) {

int sum = 0;

for (int k = 0; k < size; ++k) {

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

}

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

}

}

// Kernel to multiply two matrices where each element of the resultant matrix is computed by one thread

\_\_global\_\_ void multiplyMatrixElements(int \*A, int \*B, int \*C, int size) {

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

int row = index / size;

int col = index % size;

if (row < size && col < size) {

int sum = 0;

for (int k = 0; k < size; ++k) {

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

}

C[index] = sum;

}

}

// Function to print a matrix

void printMatrix(int \*matrix, int size) {

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

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

printf("%d ", matrix[i \* size + j]);

}

printf("\n");

}

}

int main() {

int size;

printf("Enter the size of the matrices (maximum %d): ", MAX\_SIZE);

scanf("%d", &size);

if (size <= 0 || size > MAX\_SIZE) {

printf("Invalid size!\n");

return 1;

}

int \*A, \*B, \*C;

size\_t matrixSize = size \* size \* sizeof(int);

// Allocate memory for matrices on the host

A = (int \*)malloc(matrixSize);

B = (int \*)malloc(matrixSize);

C = (int \*)malloc(matrixSize);

// Initialize random number generator

srand(time(NULL));

// Generate random matrices A and B

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

A[i] = rand() % 10; // Generate a random number between 0 and 9

B[i] = rand() % 10;

}

// Print matrices A and B

printf("Matrix A:\n");

printMatrix(A, size);

printf("\nMatrix B:\n");

printMatrix(B, size);

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

// Allocate memory for matrices on the device

cudaMalloc(&d\_A, matrixSize);

cudaMalloc(&d\_B, matrixSize);

cudaMalloc(&d\_C, matrixSize);

// Copy matrices A and B from host to device

cudaMemcpy(d\_A, A, matrixSize, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, matrixSize, cudaMemcpyHostToDevice);

// Launch kernels to multiply matrices using different thread specifications

// Uncomment the desired kernel call

// multiplyMatrixRows<<<size, size>>>(d\_A, d\_B, d\_C, size); // Each row of resultant matrix computed by one thread

// multiplyMatrixColumns<<<size, size>>>(d\_A, d\_B, d\_C, size); // Each column of resultant matrix computed by one thread

multiplyMatrixElements<<<(size \* size + 255) / 256, 256>>>(d\_A, d\_B, d\_C, size); // Each element of resultant matrix computed by one thread

// Copy resultant matrix C from device to host

cudaMemcpy(C, d\_C, matrixSize, cudaMemcpyDeviceToHost);

// Print resultant matrix C

printf("\nResultant Matrix:\n");

printMatrix(C, size);

// Free memory allocated on the device

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free memory allocated on the host

free(A);

free(B);

free(C);

return 0;

}

