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;
}
```

```
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-10$ nvcc -o MatrixAdd MatrixAdd.cu
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-10$ ./MatrixAdd
Enter the size of the matrices (maximum 10): 10
Matrix A:
5 8 4 7 8 0 9 9 3 3
8 8 1 8 3 8 8 8 1 2
8 7 9 2 6 8 5 4 7 4
1 2 0 5 9 3 5 9 6 7
7 9 4 3 0 5 3 4 1 7
2 1 2 3 4 2 1 9 9 3
1 3 0 0 1 4 9 2 0 9
1 1 7 3 5 7 9 3 2 4
2 7 0 4 9 8 8 8 2 5
2 5 5 6 5 1 6 2 1 7

Matrix B:
0 8 2 8 1 5 2 0 0 5
0 8 8 5 8 0 6 4 7 9
5 9 6 3 5 4 6 6 3 2
2 1 2 1 0 8 0 4 1 2
8 4 5 3 9 1 9 3 2 1
0 7 2 8 3 3 3 7 2 9
0 4 7 3 7 3 7 1 8 6
3 4 0 6 8 6 2 1 5 3
1 3 1 0 5 2 9 7 4 7
3 3 8 7 3 9 0 6 8 3

Resultant Matrix:
5 16 6 15 9 5 11 9 3 8
8 16 9 13 11 8 14 12 8 11
13 16 15 5 11 12 11 10 10 6
3 3 2 6 9 11 5 13 7 9
15 13 9 6 9 6 12 7 3 8
2 8 4 11 7 5 4 16 11 12
1 7 7 3 8 7 16 3 8 15
4 5 7 9 13 13 11 4 7 7
3 10 1 4 14 10 17 15 6 12
5 8 13 13 8 10 6 8 9 10
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-10$ ./MatrixAdd
Enter the size of the matrices (maximum 10): 2
Matrix A:
5 7
5 4

Matrix B:
1 7
5 8

Resultant Matrix:
6 14
10 12
```

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;
}
```

```
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-10$ nvcc -o MatrixMultiply MatrixMultiply.cu
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-10$ ./MatrixMultiply
Enter the size of the matrices (maximum 10): 10
Matrix A:
2 6 1 9 4 7 4 5 6 7
7 8 9 8 9 4 8 3 9 4
8 1 1 8 1 0 9 0 3 5
0 9 3 2 8 7 8 2 9 8
1 3 0 3 7 9 9 0 2 6
1 2 7 2 8 1 3 4 3 2
6 3 3 0 9 1 3 4 9 2
9 0 3 1 3 0 0 9 9 2
9 1 6 6 3 6 4 8 1 7
6 8 3 1 1 3 0 4 3 5

Matrix B:
9 9 7 0 4 6 9 4 3 2
2 0 1 4 7 4 2 1 4 5
1 4 8 7 7 0 0 8 1 3
1 1 2 2 6 8 1 1 2 3
0 3 6 6 8 9 0 2 3 2
6 1 6 8 3 1 7 6 8 2
3 6 4 7 7 0 3 2 6 7
6 5 5 8 5 1 3 3 0 8
3 4 5 7 8 3 6 9 1 6
7 6 9 2 5 0 6 8 2 6

Resultant Matrix:
191 165 246 253 300 174 193 214 161 232
217 261 351 348 442 253 226 296 202 289
154 183 183 127 214 134 157 136 115 161
184 187 286 316 359 160 201 260 196 264
147 140 212 221 235 120 156 162 182 169
84 126 188 195 221 116 73 147 79 140
143 180 224 215 264 161 154 190 99 175
180 196 215 180 214 125 175 191 54 174
243 236 302 237 274 150 215 230 145 217
160 135 180 148 193 101 152 156 95 152
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-10$ ./MatrixMultiply
Enter the size of the matrices (maximum 10): 3
Matrix A:
8 5 7
2 0 3
0 8 3

Matrix B:
7 1 0
6 0 1
2 3 1

Resultant Matrix:
100 29 12
20 11 3
54 9 11
```