PP LAB -6 : Programs on Matrix using CUDA

Name : Rajvardhan Reddy Nandyala

Reg No : 180905093

Sem : 6th

Sec : B **, Batch :** B1

Roll No : 19

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

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

b. Each column of the resultant matrix to be computed by one thread.

c. Each element of the resultant matrix to be computed by one thread.

P1)

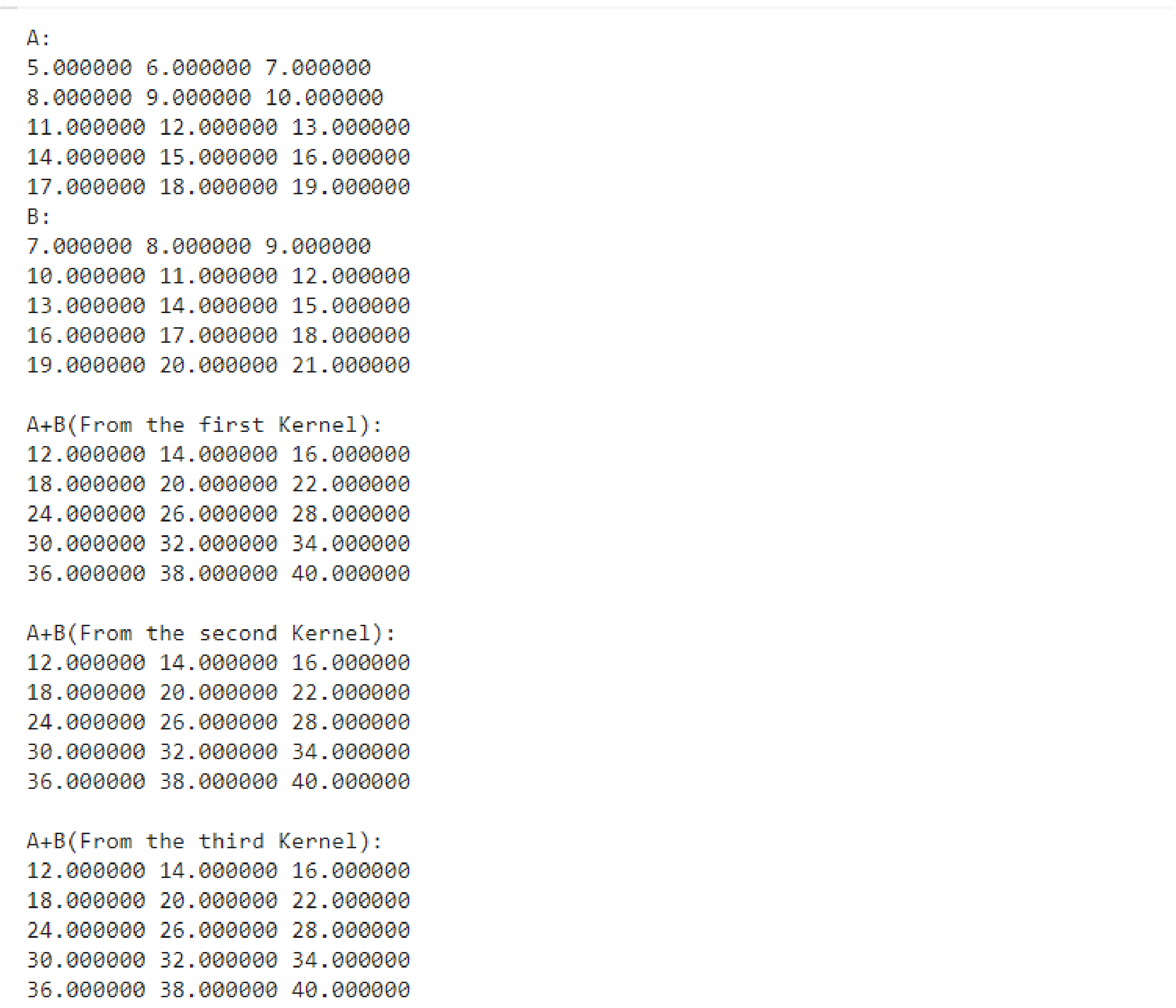
```
%%cu


#include<cuda.h>

#include<stdio.h>

#include<stdlib.h>


__global__ void matAddKernel_1a(float *a,float *b,float *c,int n)

{

   int ridA = threadIdx.x;

   int i;

   for (i=0;i<n;i++)

   {

      c[ridA*n+i] = a[ridA*n+i]+b[ridA*n+i];

   }
```

```
}

__global__ void matAddKernel_1b(float *a,float *b,float *c,int m)
{
    int col = threadIdx.x;
    int n = blockDim.x;
    int i;
    for (i=0;i<m;i++){
        c[i*n+col] = a[i*n+col]+b[i*n+col];
    }
}
__global__ void matAddKernel_1c(float *a,float *b,float *c)
{
    int row = threadIdx.x;
    int col = threadIdx.y;
    int n = blockDim.y;
    c[row*n+col] = a[row*n+col]+b[row*n+col];
}
void matAdd(float *a,float *b,float *c,int m,int n)
{
    int size = m*n*sizeof(float);
    float* d_A;
    float* d_B;
    float* d_C;

    cudaMalloc((void**) &d_A, size);
    cudaMalloc((void**) &d_B, size);
    cudaMalloc((void**) &d_C, size);

    cudaMemcpy(d_A, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, b, size, cudaMemcpyHostToDevice);

    int i,j;
    printf("A:\n");

    for (i=0;i<m;i++)
    {
        for (j=0;j<n;j++)
        {
           printf("%f ",*(a + i*n + j));
        }
        printf("\n");
    }

    printf("B:\n");

    for (i=0;i<m;i++)
    {
        for (j=0;j<n;j++)
        {
           printf("%f ",*(b + i*n + j));
        }
```

```
    printf("\n");
}

printf("\n");
matAddKernel_1a<<<1, m>>>(d_A, d_B, d_C, n);
cudaMemcpy(c, d_C, size, cudaMemcpyDeviceToHost);
printf("A+B(From the first Kernel):\n");

for (i=0;i<m;i++)
{
    for (j=0;j<n;j++)
    {
        printf("%f ",*(c + i*n + j));
    }
    printf("\n");
}

printf("\n");
matAddKernel_1b<<<1, n>>>(d_A, d_B, d_C,m);
cudaMemcpy(c, d_C, size, cudaMemcpyDeviceToHost);

printf("A+B(From the second Kernel):\n");
for (i=0;i<m;i++)
{
    for (j=0;j<n;j++)
    {
            printf("%f ",*(c + i*n + j));
        }
        printf("\n");
    }

    printf("\n");
    matAddKernel_1c<<<(1,1), (m,n)>>>(d_A, d_B, d_C);
    cudaMemcpy(c, d_C, size, cudaMemcpyDeviceToHost);

    printf("A+B(From the third Kernel):\n");
    for (i=0;i<m;i++)
    {
        for (j=0;j<n;j++)
        {
            printf("%f ",*(c + i*n + j));
        }
        printf("\n");
    }

    printf("\n");

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

```
int main()
{
    float *a,*b,*c;
    int n=3,m=5;
    int size = m*n*sizeof(float);
    a = (float*)malloc(size);
    b = (float*)malloc(size);
    c = (float*)malloc(size);
    int i,j,k=5;
    for (i=0;i<m;i++)
    {
        for (j=0;j<n;j++)
        {
            *(a + i*n + j) = float(k);
            *(b + i*n + j) = float(k+2);
            k+=1;
        }
    }
    matAdd(a,b,c,m,n);
}
```

Output :



```
A:
5.000000 6.000000 7.000000
8.000000 9.000000 10.000000
11.000000 12.000000 13.000000
14.000000 15.000000 16.000000
17.000000 18.000000 19.000000
B:
7.000000 8.000000 9.000000
10.000000 11.000000 12.000000
13.000000 14.000000 15.000000
16.000000 17.000000 18.000000
19.000000 20.000000 21.000000

A+B(From the first Kernel):
12.000000 14.000000 16.000000
18.000000 20.000000 22.000000
24.000000 26.000000 28.000000
30.000000 32.000000 34.000000
36.000000 38.000000 40.000000

A+B(From the second Kernel):
12.000000 14.000000 16.000000
18.000000 20.000000 22.000000
24.000000 26.000000 28.000000
30.000000 32.000000 34.000000
36.000000 38.000000 40.000000

A+B(From the third Kernel):
12.000000 14.000000 16.000000
18.000000 20.000000 22.000000
24.000000 26.000000 28.000000
30.000000 32.000000 34.000000
36.000000 38.000000 40.000000
```

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

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

b. Each column of the resultant matrix to be computed by one thread.

c. Each element of the resultant matrix to be computed by one thread.

P2)

```
%%cu

#include <stdio.h>
```

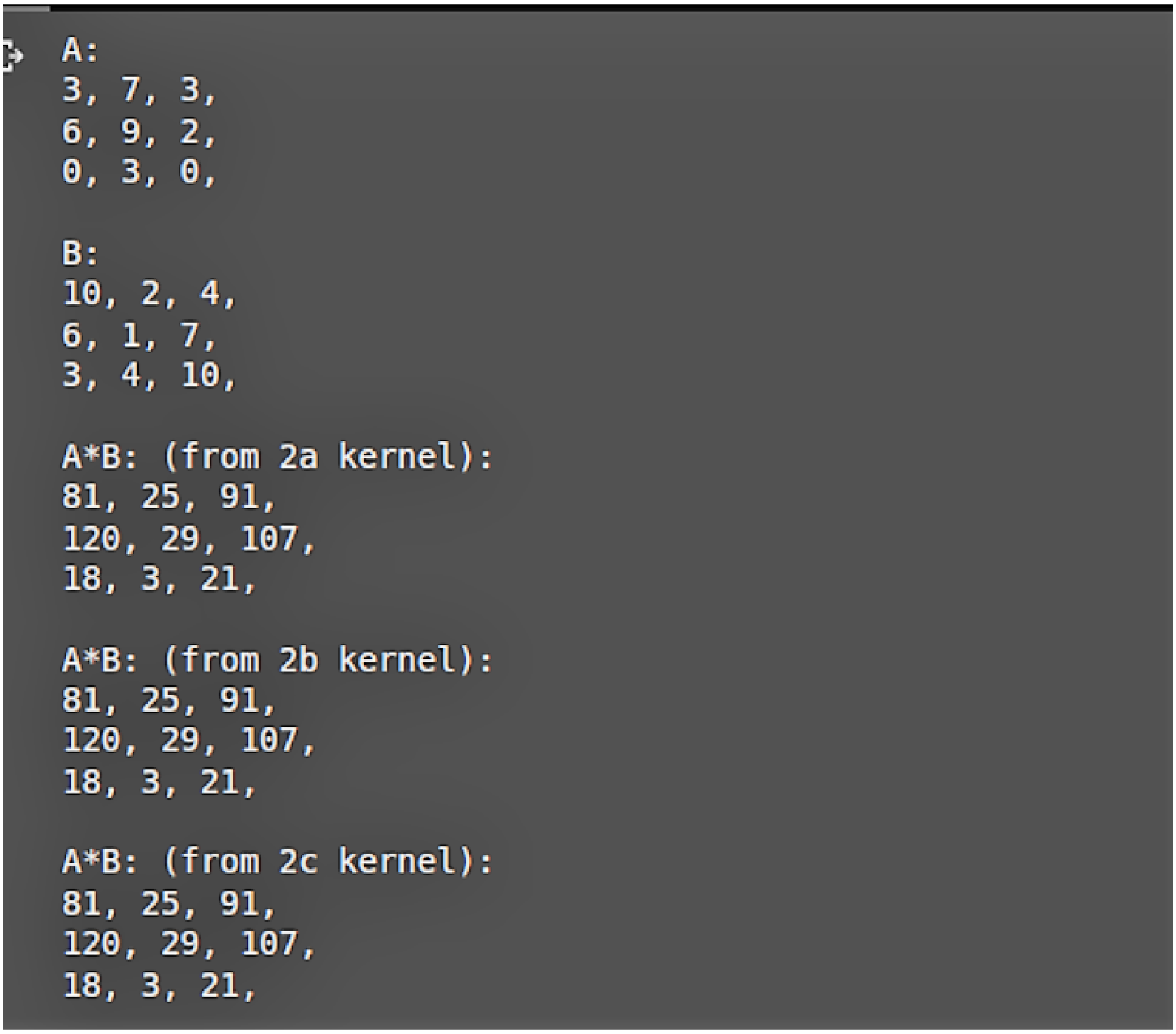
```
#include <stdlib.h>
#include <cuda.h>
__host__ __device__ void printMatrix(const char* string, int* A, int width)
{
printf("%s\n", string);
for (int i = 0; i < width; i++)
{
for (int j = 0; j < width; j++)
{
printf("%d, ", A[i*width + j]);
}
printf("\n");
}
printf("\n");
}
__host__ void clearMatrix(int* A, int width)
{
for (int i = 0; i < width; i++)
{
for (int j = 0; j < width; j++)
{
A[i*width + j] = 0;
}
}
}
__global__
```

```
void multiplyMatrixKernel_2a(int* A, int* B, int* C, int width)
{
int row = threadIdx.y;
int k = 0;
for (int i = 0; i < width; i++)
{
k = 0;
for (int j = 0; j < width; j++)
{
k += A[row*width + j] * B[i + width*j];
}
C[row*width + i] = k;
}
}
__global__
void multiplyMatrixKernel_2b(int* A, int* B, int* C, int width)
{
int col = threadIdx.x;
int k = 0;
for (int i = 0; i < width; i++)
{
k = 0;
for (int j = 0; j < width; j++)
{
k += A[i*width + j] * B[col + j*width];
}
C[i*width + col] = k;
}
}
__global__
void multiplyMatrixKernel_2c(int* A, int* B, int* C, int width)
{
int col = threadIdx.x;
int row = threadIdx.y;
int k = 0;
for (int i = 0; i < width; i++)
{
k += A[row*width + i] * B[col + i*width];
}
C[row*width + col] = k;
}
void multiplyMatrix(int* h_A, int* h_B, int* h_C, int width)
{
int *d_A, *d_B, *d_C;
int size = width * width * sizeof(int);
cudaMalloc((void**) &d_A, size);
cudaMalloc((void**) &d_B, size);
cudaMalloc((void**) &d_C, size);
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);
dim3 dimBlock(1, 1, 1);
```

```
dim3 dimGrid(1, 1, 1);
dimBlock.x = 1;
dimBlock.y = width;
dimBlock.z = 1;
multiplyMatrixKernel_2a<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, width);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
printMatrix("A*B: (from 2a kernel): ", h_C, width);
clearMatrix(h_C, width);
cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);
dimBlock.x = width;
dimBlock.y = 1;
dimBlock.z = 1;
multiplyMatrixKernel_2b<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, width);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
printMatrix("A*B: (from 2b kernel): ", h_C, width);
clearMatrix(h_C, width);
cudaMemcpy(d_C, h_C, size, cudaMemcpyHostToDevice);
dimBlock.x = width;
dimBlock.y = width;
dimBlock.z = 1;
multiplyMatrixKernel_2c<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, width);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
printMatrix("A*B: (from 2c kernel): ", h_C, width);
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

```
}
int main()
{
int *A, *B, *C;
int width = 3;
int size = width * width * sizeof(int);
A = (int*) calloc(width * width, sizeof(int));
B = (int*) calloc(width * width, sizeof(int));
C = (int*) calloc(width * width, sizeof(int));
int k = 1;
for (int i = 0; i < width; i++)
{
for (int j = 0; j < width; j++)
{
A[i*width + j] = rand() % 10;
B[i*width + j] = rand() % 11;
k++;
}
}
printMatrix("A:", A, width);
printMatrix("B:", B, width);
multiplyMatrix(A, B, C, width);
return 0;
}
```

Output :



```
A:
3, 7, 3,
6, 9, 2,
0, 3, 0,

B:
10, 2, 4,
6, 1, 7,
3, 4, 10,

A*B: (from 2a kernel):
81, 25, 91,
120, 29, 107,
18, 3, 21,

A*B: (from 2b kernel):
81, 25, 91,
120, 29, 107,
18, 3, 21,

A*B: (from 2c kernel):
81, 25, 91,
120, 29, 107,
18, 3, 21,
```