Name: Siddhi Hagawane

PRN No: 2020BTECS00049

**High Performance Computing Lab**

**Practical No. 9**

**Title of practical:** Implementation of Matrix-matrix Multiplication (global and shared Memory), Prefix sum, 2D Convolution using CUDA C

**Problem Statement 1:**

Implement Matrix-matrix Multiplication using global memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

**Code:**

# matrix matrix multiplication

%%cu

#include<stdio.h>

#include <time.h>

const int n=10;

\_global\_ void multiply(int \*mat1, int \*mat2, int \*result, int n)

{

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

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

    if(row<n && col<n)

    {

        for(int k=0; k<n; k++)

        {

            result[row\*n+col]+=mat1[row\*n+k]\*mat2[k\*n+col];

        }

    }

}

int main()

{

    clock\_t start, end;

    start = clock();

    int mat1[n\*n];

    int mat2[n\*n];

    for(int i=0;i<n\*n;i++)

    {

        mat1[i]=i+1;

        mat2[i]=n\*n-i;

    }

    int result[n\*n];

    int \*d\_mat1, \*d\_mat2, \*d\_result;

    cudaMalloc(&d\_mat1, n\*n\*sizeof(int));

    cudaMalloc(&d\_mat2, n\*n\*sizeof(int));

    cudaMalloc(&d\_result, n\*n\*sizeof(int));

    cudaMemcpy(d\_mat1, mat1, n\*n\*sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_mat2, mat2, n\*n\*sizeof(int), cudaMemcpyHostToDevice);

    int b\_size=2;

    int g\_size=ceil(n/2.0);

    dim3 threads(b\_size, b\_size);

    dim3 blocks(g\_size, g\_size);

    multiply<<<blocks,threads>>>(d\_mat1,d\_mat2,d\_result,n);

    cudaDeviceSynchronize();

    cudaMemcpy(result, d\_result, n\*n\*sizeof(int), cudaMemcpyDeviceToHost);

    end = clock();

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

    printf("\nTime taken to execute in seconds : %f\n", duration);

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            printf("%d ",result[i\*n+j]);

        }

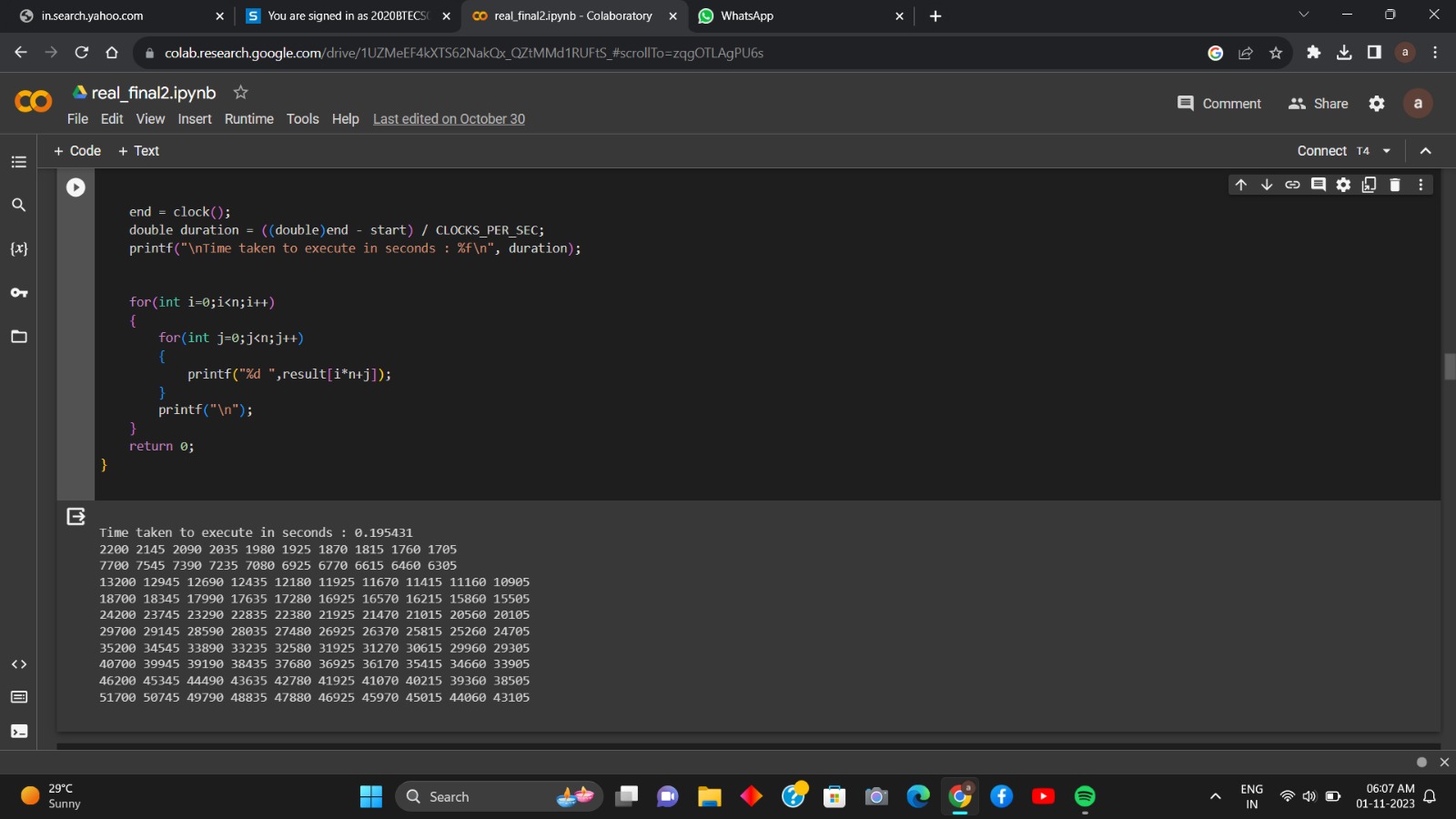
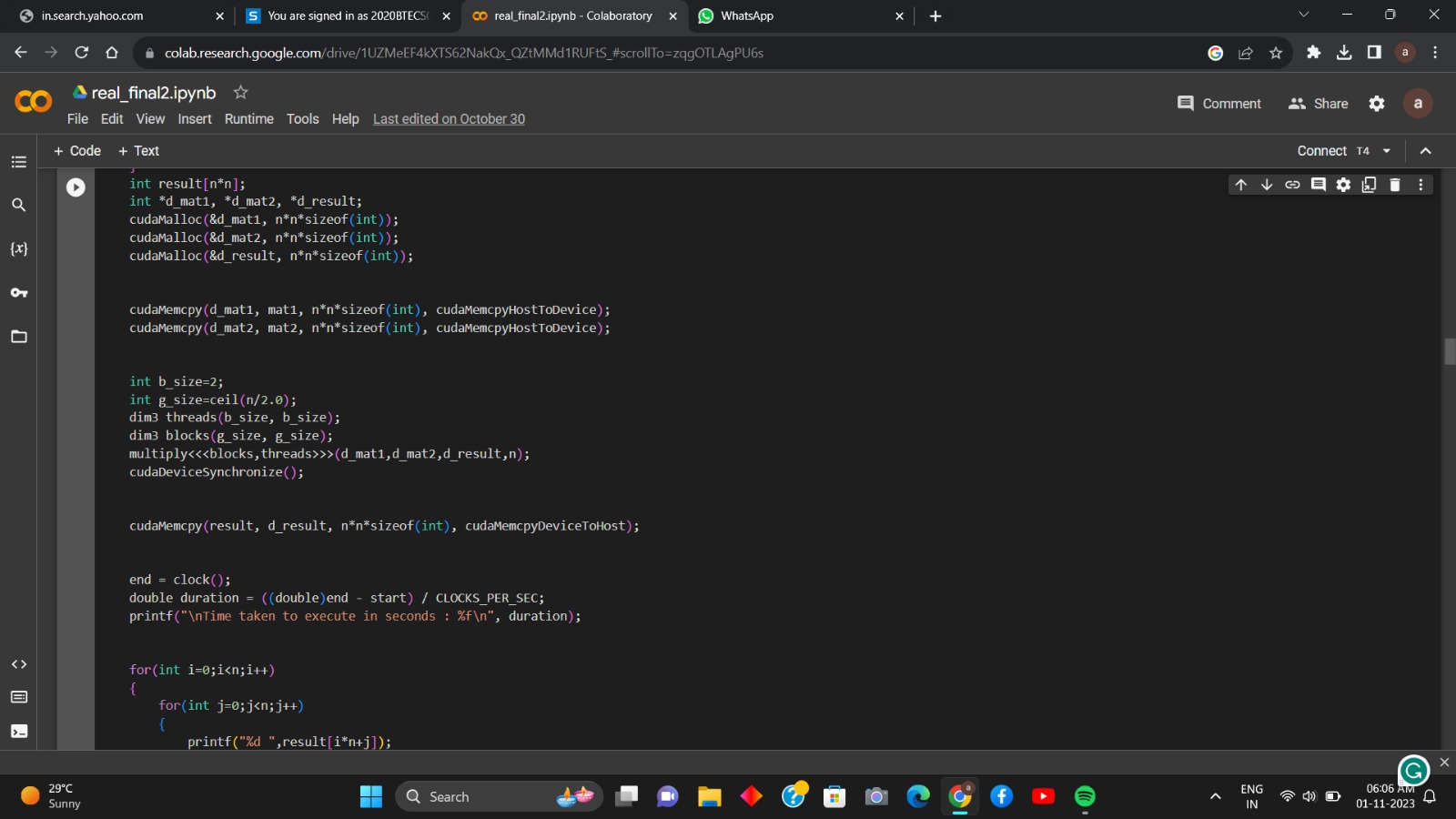
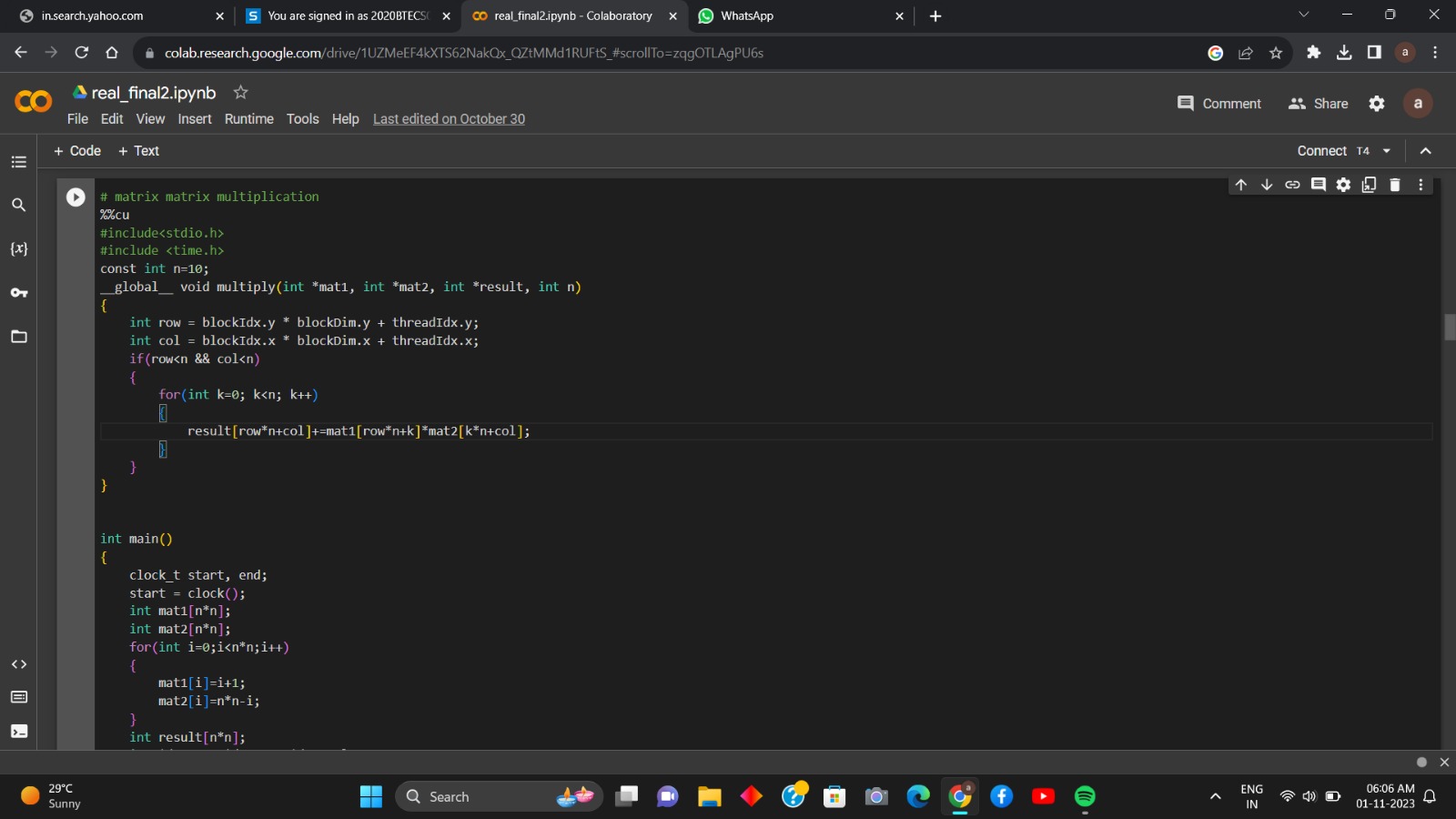
        printf("\n");

    }

    return 0;

}

**Screenshots:**



**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Data Size (n)** | **Execution time** |
| O(n2) | 10 | 0.195431 |
| O(n2) | 100 | 0.192159 |
| O(n2) | 500 | 0.213233 |

For performing the matrix-matrix multiplication, we have used the block of n2 threads as we have to calculate n2 elements in the resultant matrix. For calculating each element of the result matrix, row of the first matrix and one column of the second matrix is needed. So, we have assigned the unique row and column to each thread. We can get the unique row and column by using the block indexes and thread indexes as they are unique as a combination.

**Problem Statement 2:**

Implement Matrix-matrix Multiplication using shared memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

**Code:**

// matrix matrix multiplication using shared memory

%%cu

#include<stdio.h>

#include <time.h>

const int n =500;

\_global\_ void multiply(int \*mat1, int \*mat2, int \*result, int n)

{

    int row = blockIdx.y;

    int col = blockIdx.x;

    int k = threadIdx.x;

    \_shared\_ int temp[1000];

    temp[k]=mat1[row\*n+k]\*mat2[k\*n+col];

    \_\_syncthreads();

    for(int i=0;i<n;i++)

    {

        result[row\*n+col]+=temp[i];

    }

}

int main()

{

    clock\_t start, end;

    start = clock();

    int mat1[n\*n];

    int mat2[n\*n];

    for(int i=0;i<n\*n;i++)

    {

        mat1[i]=i+1;

        mat2[i]=n\*n-i;

    }

    int result[n\*n];

    int \*d\_mat1, \*d\_mat2, \*d\_result;

    cudaMalloc(&d\_mat1, n\*n\*sizeof(int));

    cudaMalloc(&d\_mat2, n\*n\*sizeof(int));

    cudaMalloc(&d\_result, n\*n\*sizeof(int));

    cudaMemcpy(d\_mat1, mat1, n\*n\*sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_mat2, mat2, n\*n\*sizeof(int), cudaMemcpyHostToDevice);

    dim3 blocks(n,n);

    multiply<<<blocks,n>>>(d\_mat1,d\_mat2,d\_result,n);

    cudaDeviceSynchronize();

    cudaMemcpy(result, d\_result, n\*n\*sizeof(int), cudaMemcpyDeviceToHost);

    end = clock();

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

    printf("\nTime taken to execute in seconds : %f\n", duration);

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            printf("%d ",result[i\*n+j]);

        }

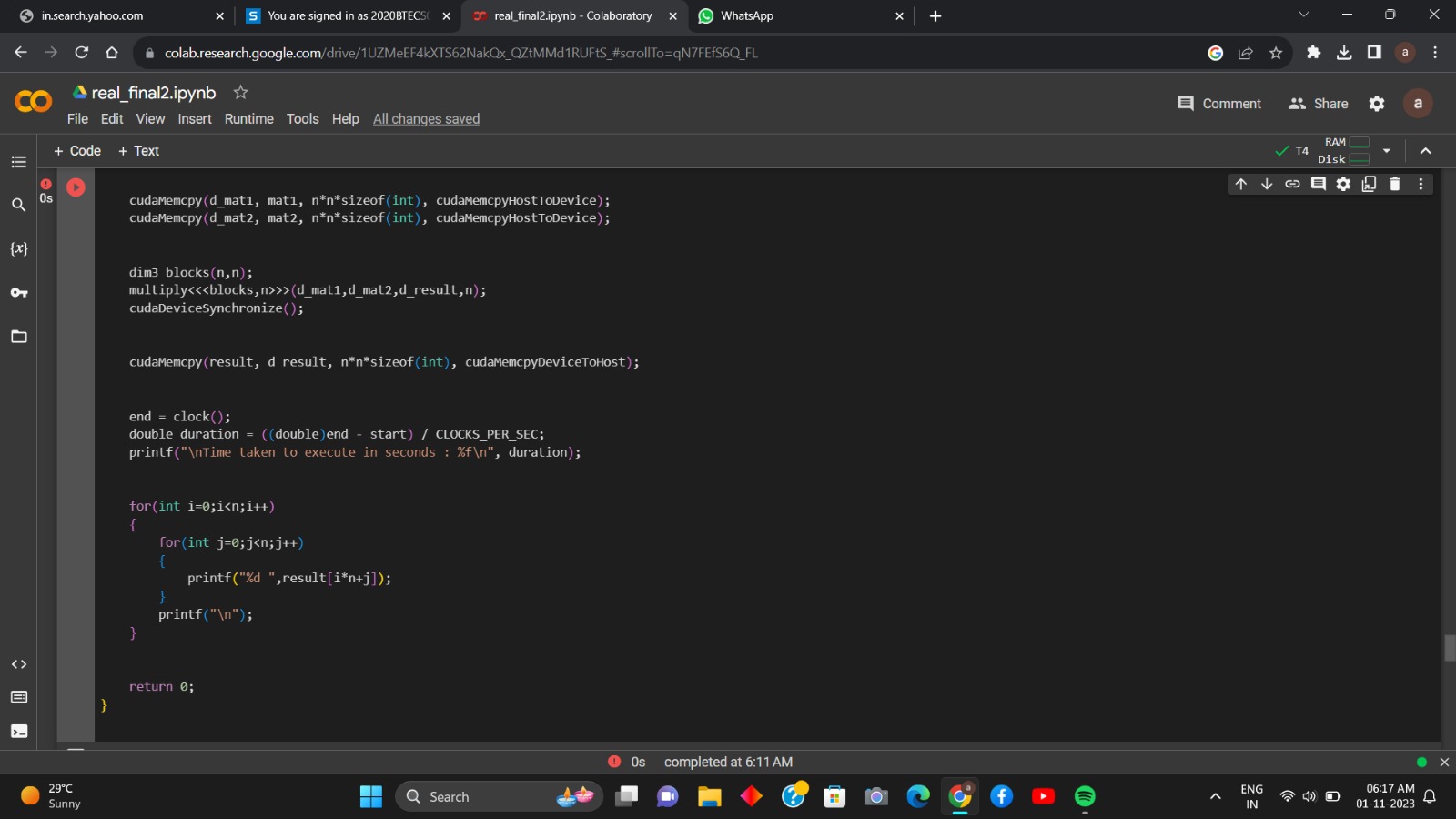
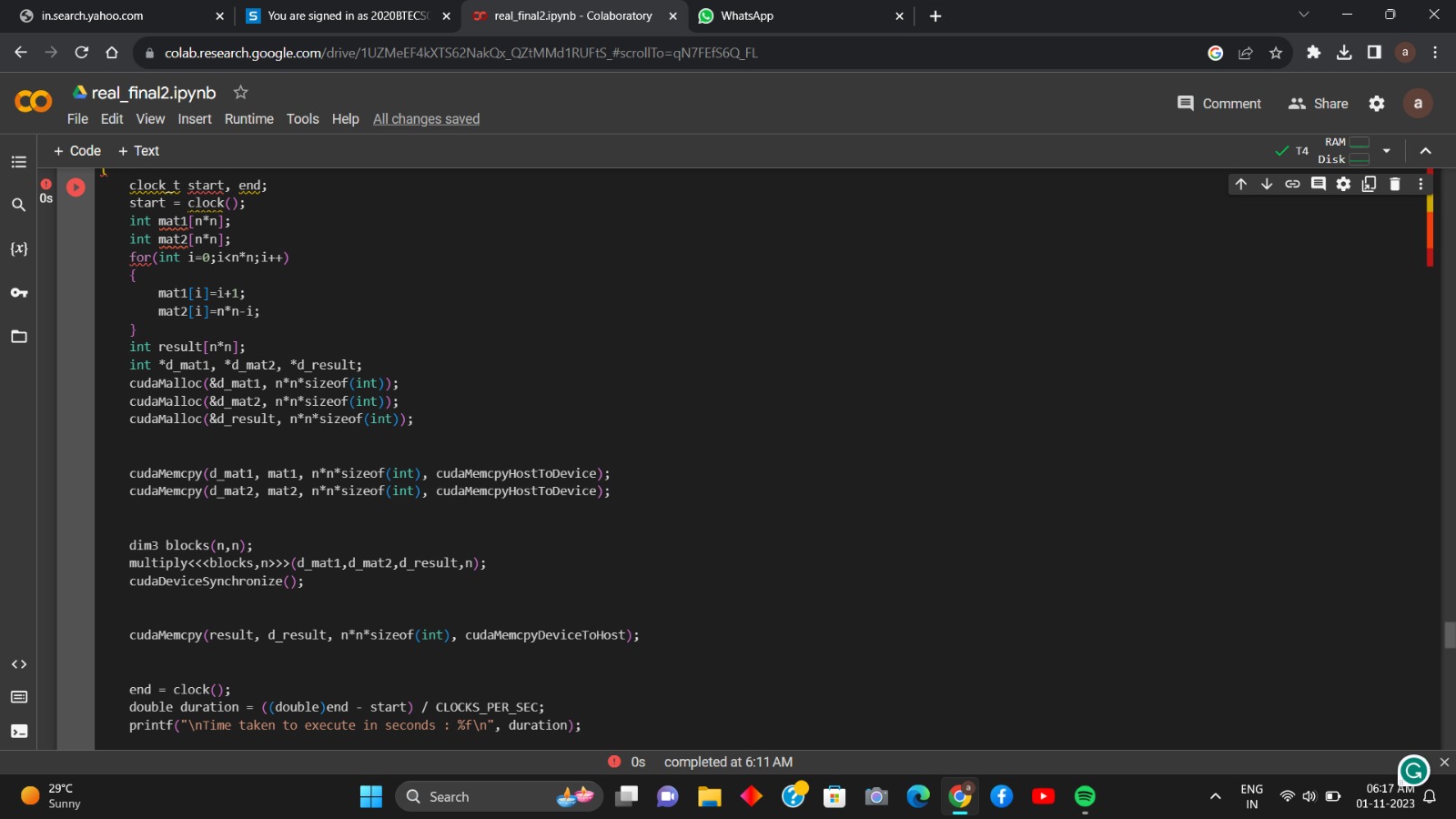
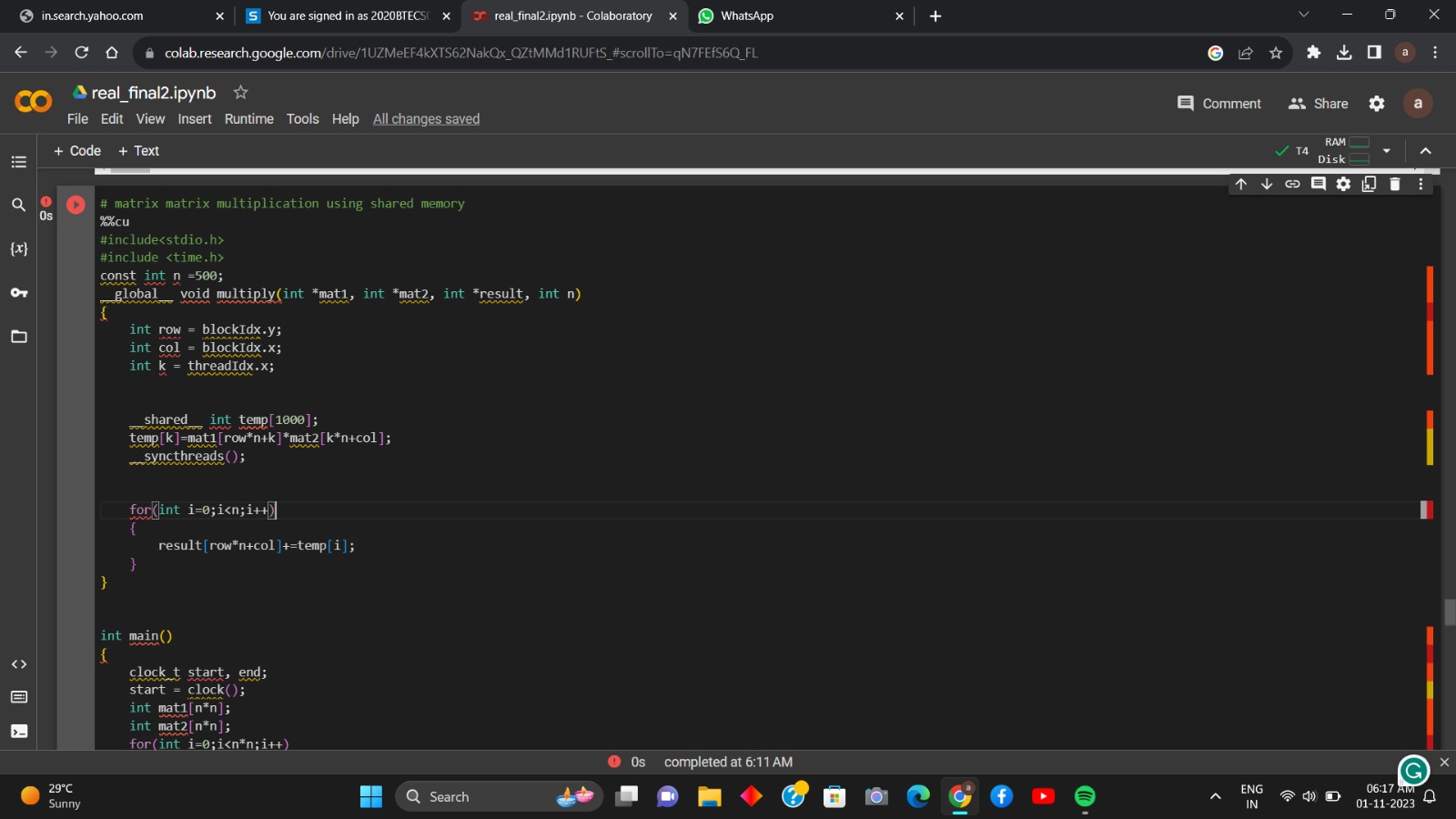
        printf("\n");

    }

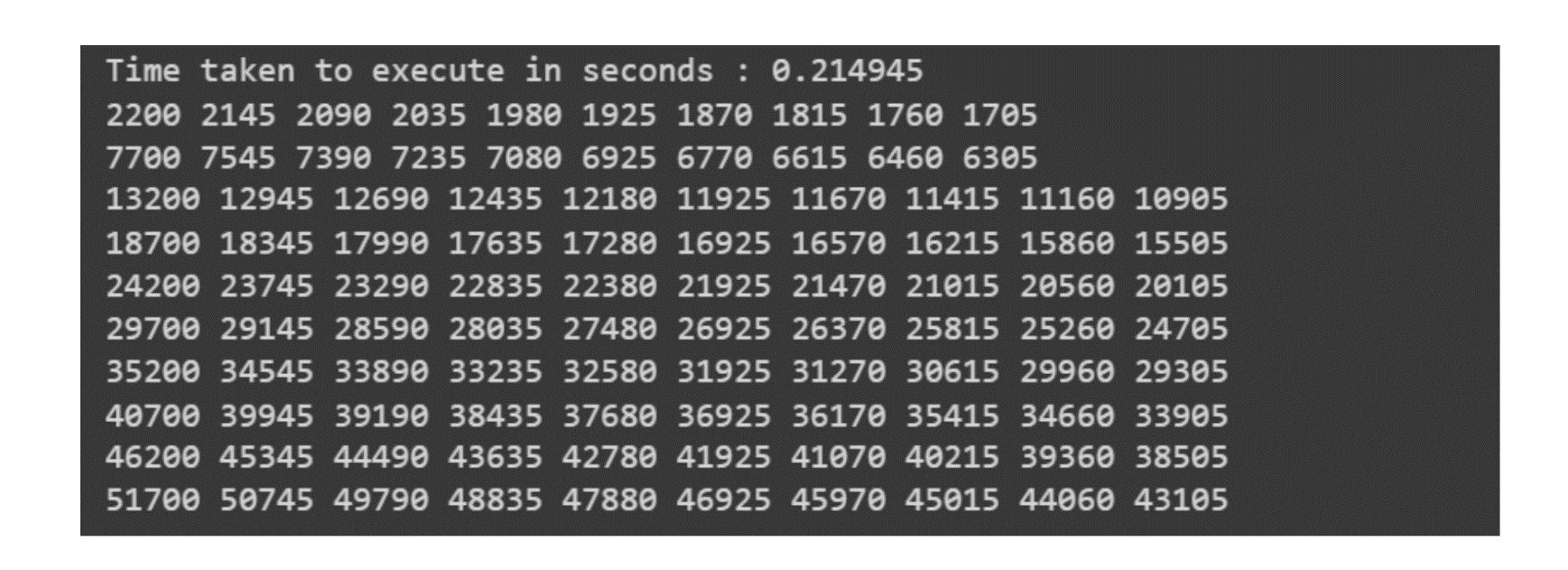
    return 0;

}

**Screenshots:**



**Output:**



**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Data Size (n)** | **Execution time** |
| O(n3) | 10 | 0.214945 |
| O(n3) | 100 | 0.196268 |
| O(n3) | 500 | 0.287709 |

For performing the matrix-matrix multiplication, we have used the n2 blocks of nthreads each as we have to calculate n2 elements in the resultant matrix. In this computation, for calculating each element of the result matrix, we need n threads.

In this implementation, the matrix multiplication is computed using a grid of thread blocks. Each thread block is responsible for computing a sub-matrix of the resulting matrix. Threads within a block cooperate to load the necessary data into shared memory and perform the matrix multiplication. The use of shared memory minimizes global memory accesses, leading to significant speedups.

**Anaylsis by comparing shared and global version:**

While using the global memory version of the program, one element of the result matrix is calculated by the one thread. But, in case of the shared memory each element is calculated by one block consisting of n threads each.

We can get the significant speedup in case of the shared memory program as it reduces the global memory accesses and increases the local memory accesses.

**Problem Statement 3:**

Implement Prefix sum using CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

**Code:**

%%cu

#include<stdio.h>

#include <time.h>

const int n =1000;

\_global\_ void calculate2DConvolution(int \*image, int \*mask, int \*result, int n, int maskdim)

{

    int offset = maskdim/2;

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

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

    if(row<n && col<n)

    {

        int start\_row = row - offset;

        int start\_col = col - offset;

        for(int i=0;i<maskdim;i++)

        {

            for(int j=0;j<maskdim;j++)

            {

                if(start\_row+i>=0 && start\_row+i<n && start\_col+j>=0 && start\_col+j<n)

                {

                    int cr = start\_row+i, cc = start\_col+j;

                    result[row\*n+col]+=image[cr\*n+cc]\*mask[i\*maskdim+j];

                }

            }

        }

    }

}

int main()

{

    clock\_t start, end;

    start = clock();

    int maskdim =3;

    int image[n\*n];

    for(int i=0;i<n\*n;i++)

    {

        int x=i/n;

        int y=i%n;

        image[i]=min(x,y);

    }

    int mask[]={1,2,3,4,5,6,7,8,9};

    int result[n\*n];

    int \*d\_image, \*d\_mask, \*d\_result;

    cudaMalloc(&d\_image,n\*n\*sizeof(int));

    cudaMalloc(&d\_mask, maskdim\*maskdim\*sizeof(int));

    cudaMalloc(&d\_result, n\*n\*sizeof(int));

    cudaMemcpy(d\_image, image, n\*n\*sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_mask, mask, maskdim\*maskdim\*sizeof(int), cudaMemcpyHostToDevice);

    int thread=2;

    int block=ceil((n\*1.0)/thread);

    dim3 blocks(block,block);

    dim3 threads(thread,thread);

    calculate2DConvolution<<<blocks,threads>>>(d\_image,d\_mask,d\_result,n,maskdim);

    cudaDeviceSynchronize();

    cudaMemcpy(result, d\_result, n\*n\*sizeof(int), cudaMemcpyDeviceToHost);

    end = clock();

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

    printf("\nTime taken to execute in seconds : %f\n", duration);

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            printf("%d ",result[i\*n+j]);

        }

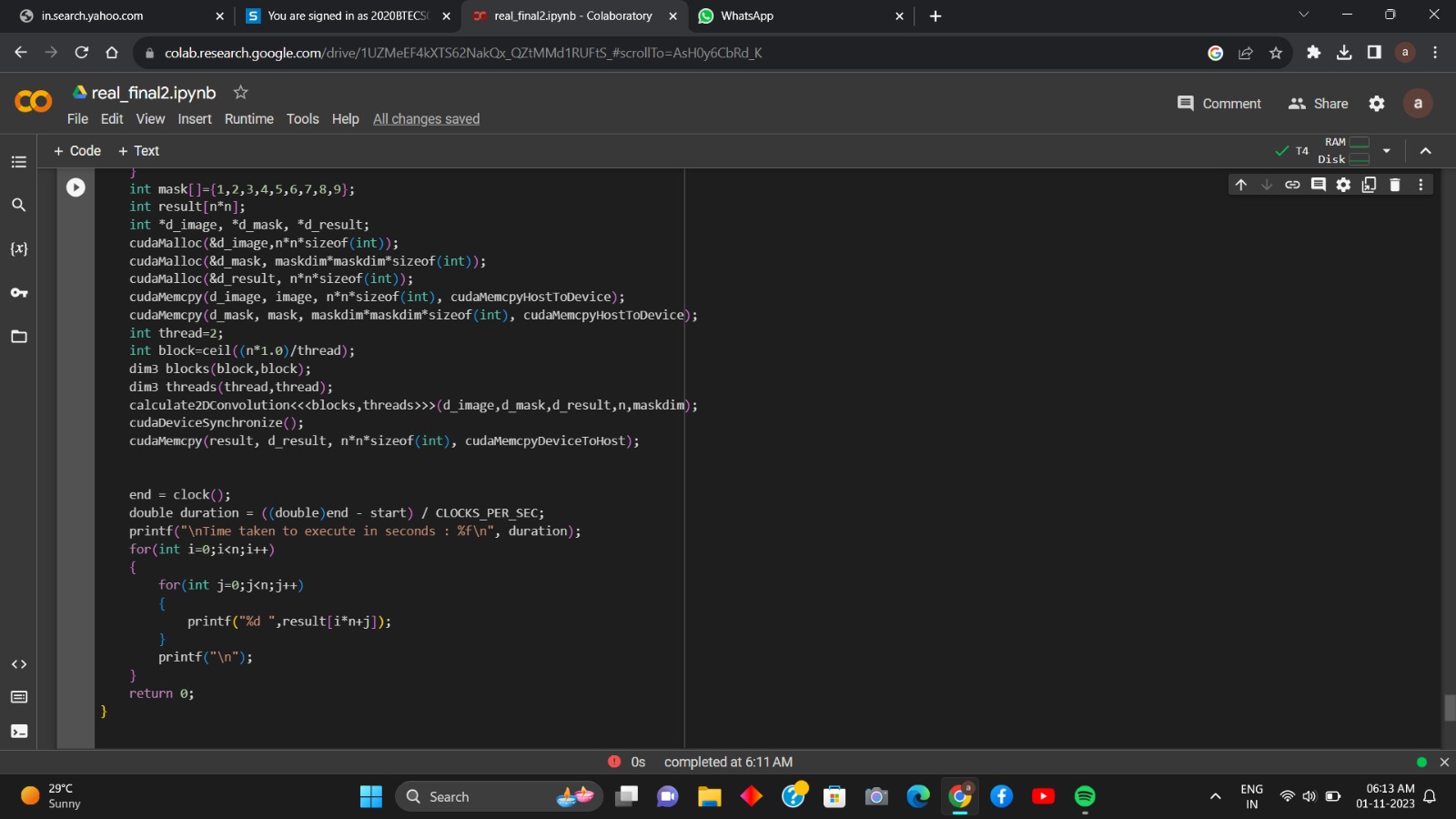
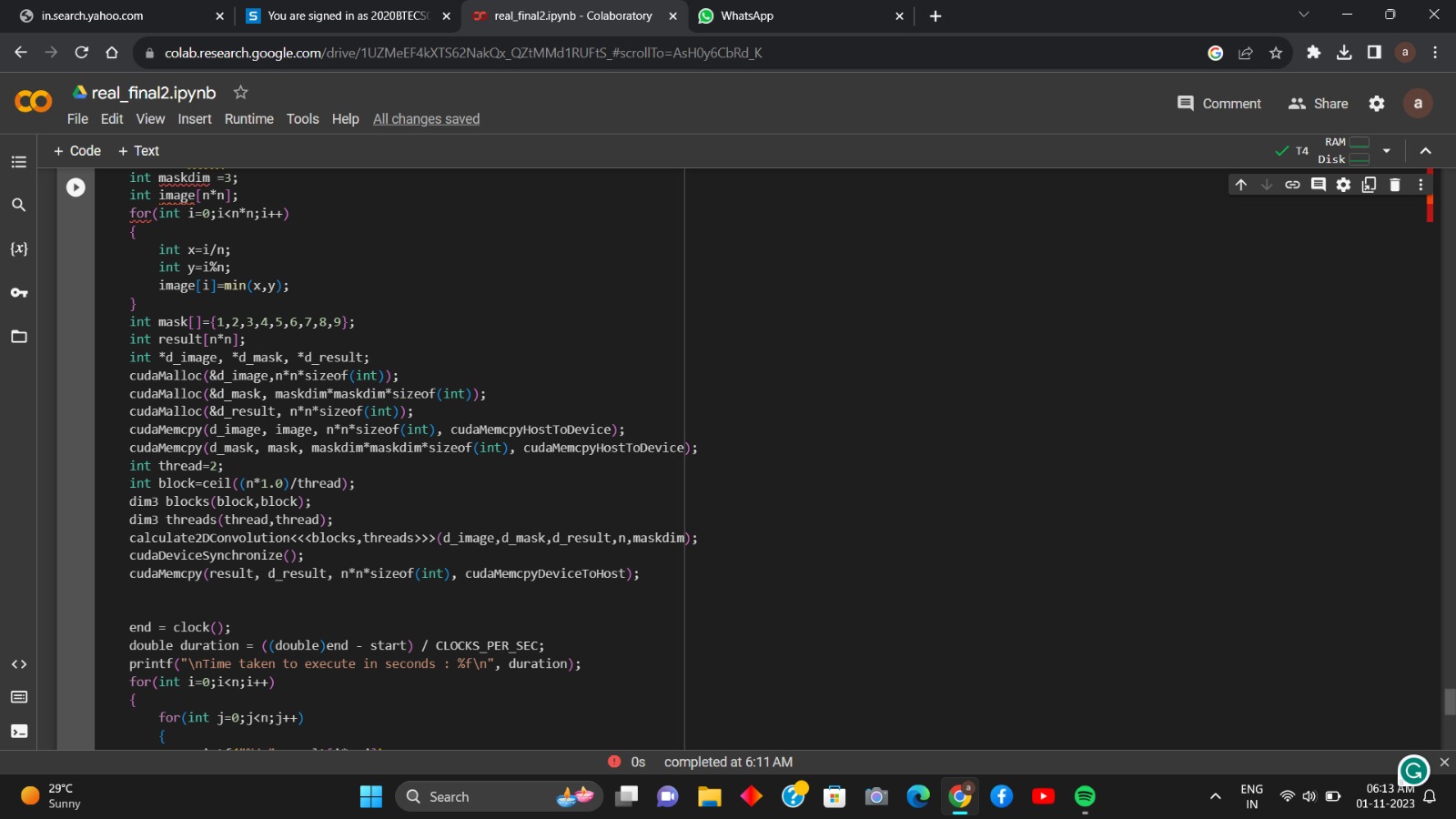
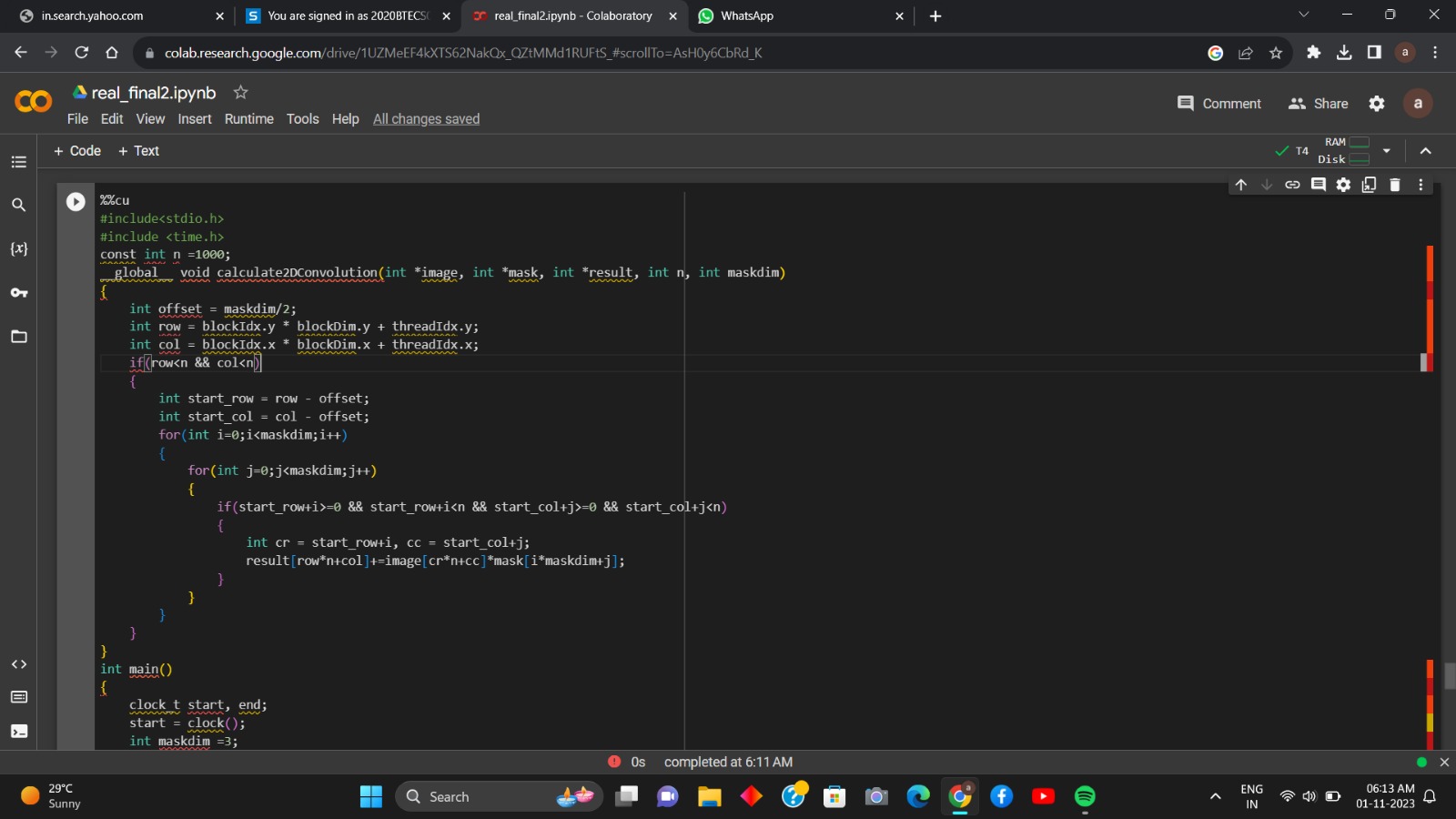
        printf("\n");

    }

    return 0;

}

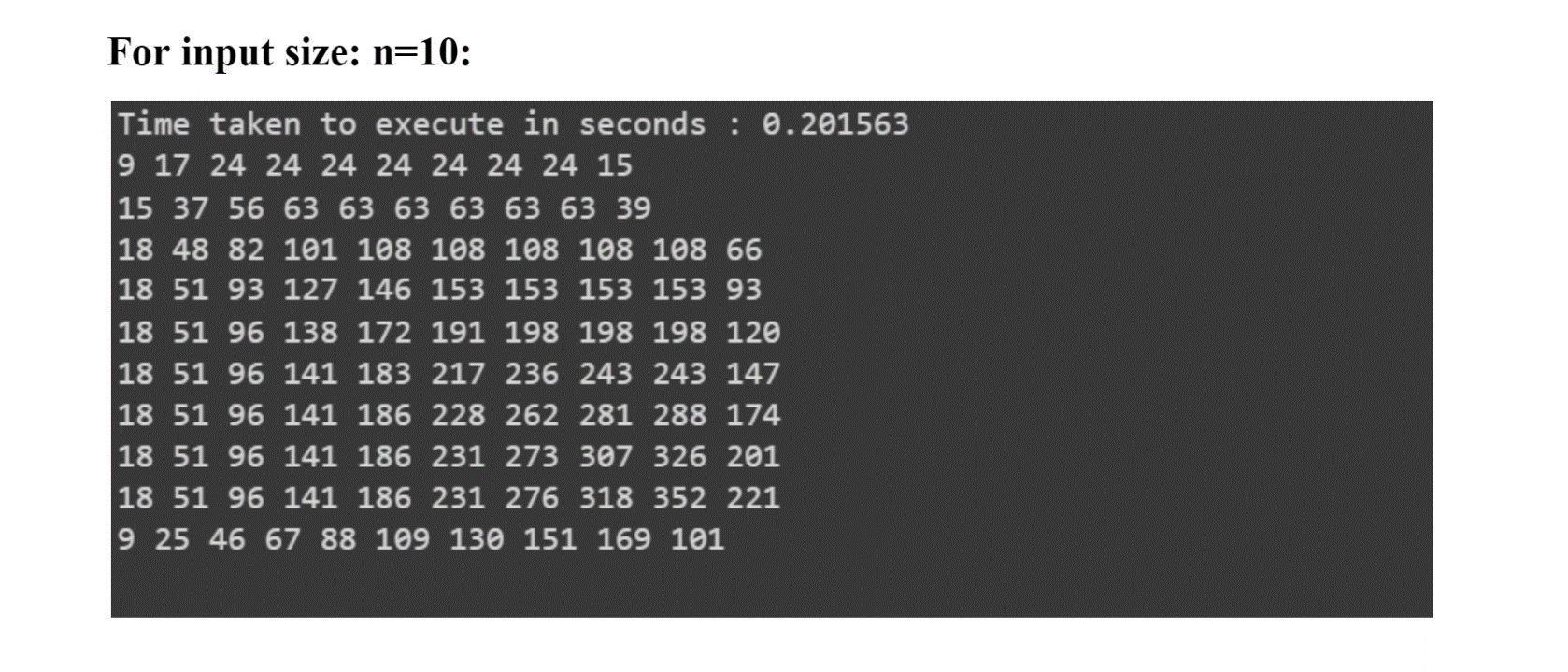
**Screenshots:**



**Analysis:**

When implementing 2D convolution using shared memory in CUDA, the computation is distributed among threads within a block, with each thread responsible for computing an element of the output matrix. The implementation involves loading the necessary data into shared memory to minimize global memory transactions and maximize memory access efficiency.

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Data Size (n)** | **Execution time** |
| O(n2) | 10 | 0.201563 |
| O(n2) | 100 | 0.224688 |
| O(n2) | 1000 |  |



**GitHub Link:**

<https://github.com/Siddhish16/HPC-Assignments>