**C lass:** Final Year (Computer Science and Engineering)

**Year:** 2022-23 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 10**

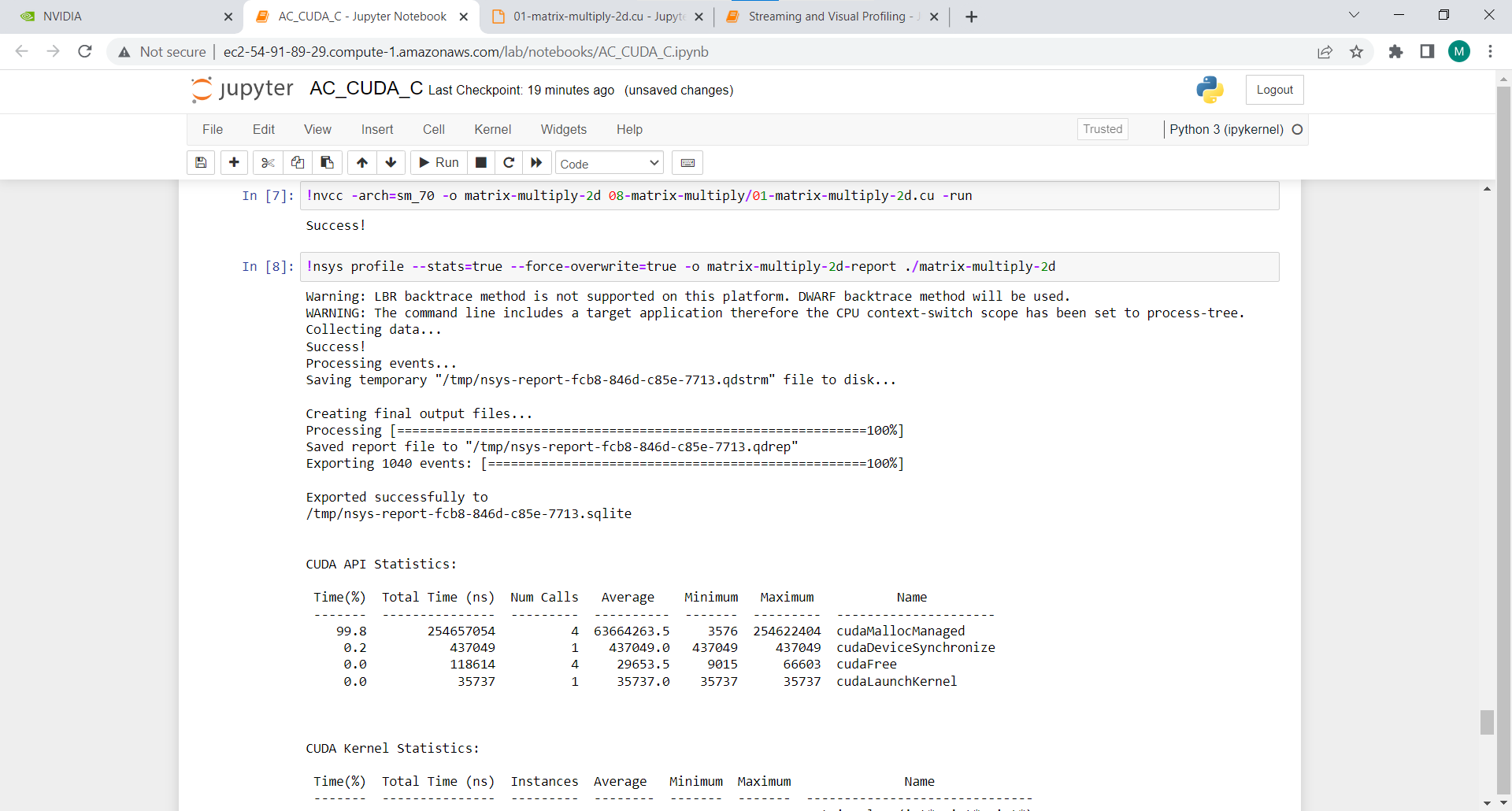
**Exam Seat No:**

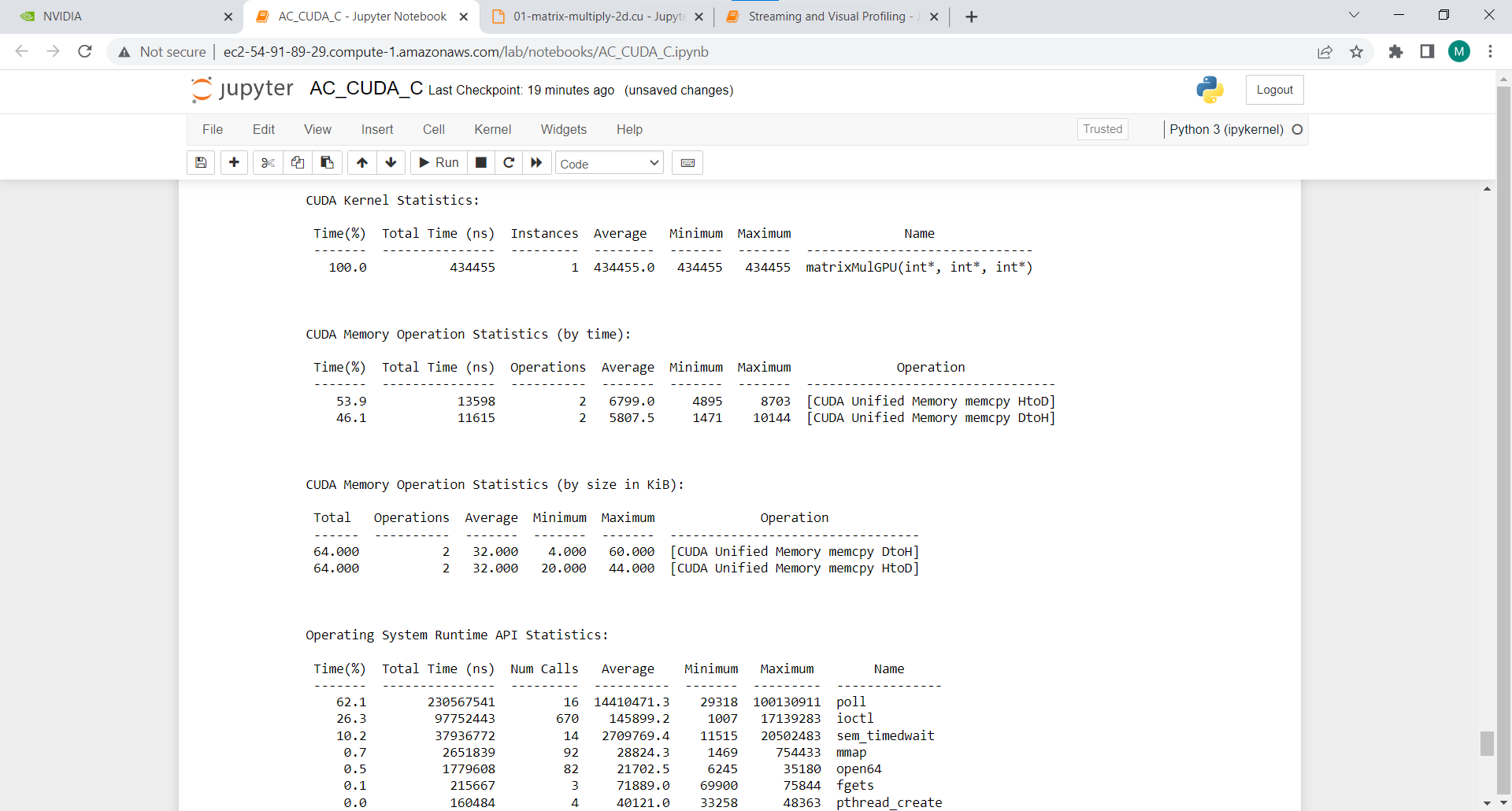
2019BTECS00033 – Teknath Jha

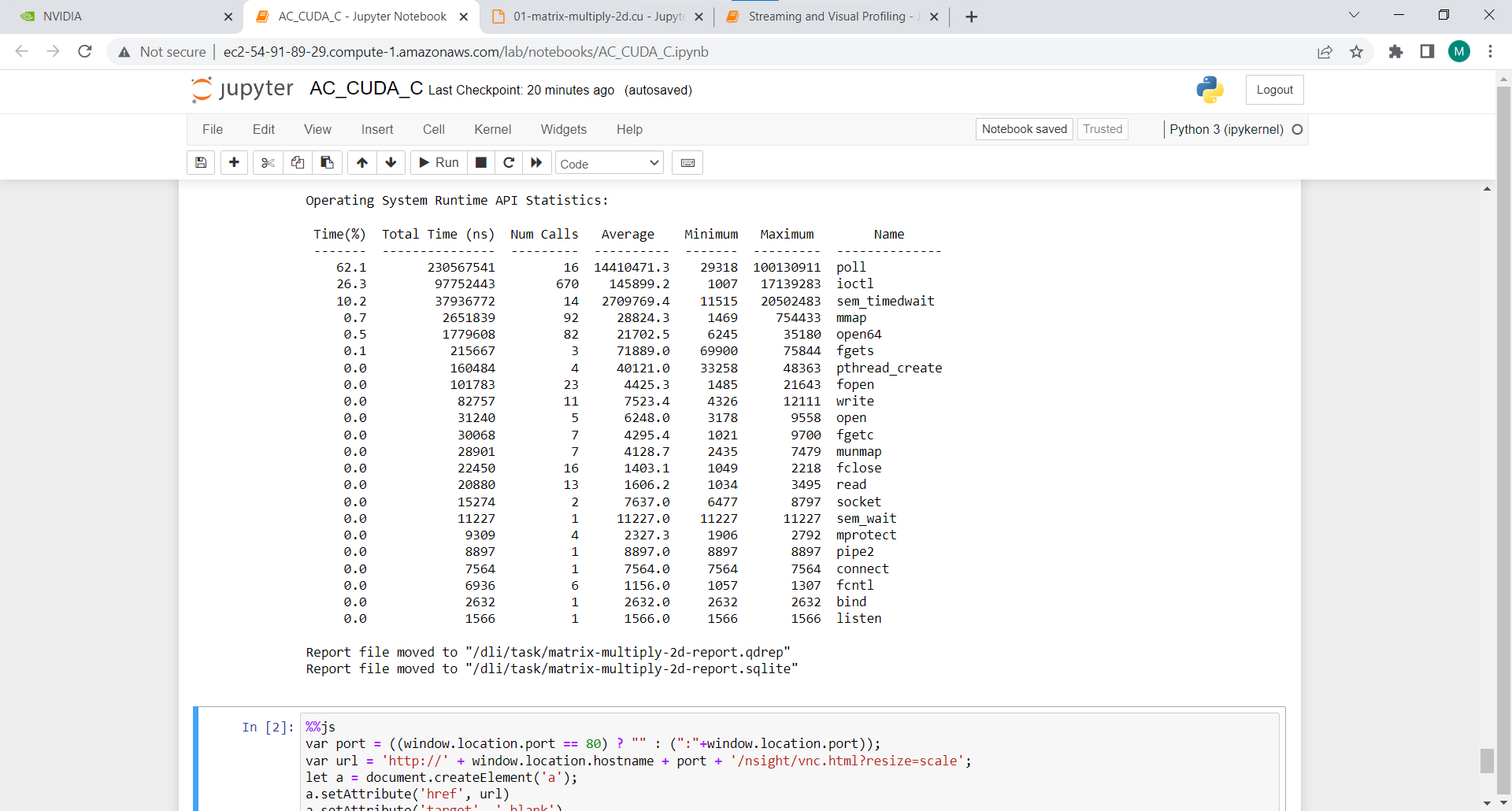
**Problem Statement 1:**

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

**Screenshot #:**







**Information #:**

#include <stdio.h>

#define N 64

\_\_global\_\_ void matrixMulGPU( int \* a, int \* b, int \* c )

{

int val = 0;

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

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

if (row < N && col < N)

{

for ( int k = 0; k < N; ++k )

val += a[row \* N + k] \* b[k \* N + col];

c[row \* N + col] = val;

}

}

void matrixMulCPU( int \* a, int \* b, int \* c )

{

int val = 0;

for( int row = 0; row < N; ++row )

for( int col = 0; col < N; ++col )

{

val = 0;

for ( int k = 0; k < N; ++k )

val += a[row \* N + k] \* b[k \* N + col];

c[row \* N + col] = val;

}

}

int main()

{

int \*a, \*b, \*c\_cpu, \*c\_gpu;

int size = N \* N \* sizeof (int); // Number of bytes of an N x N matrix

// Allocate memory

cudaMallocManaged (&a, size);

cudaMallocManaged (&b, size);

cudaMallocManaged (&c\_cpu, size);

cudaMallocManaged (&c\_gpu, size);

// Initialize memory

for( int row = 0; row < N; ++row )

for( int col = 0; col < N; ++col )

{

a[row\*N + col] = row;

b[row\*N + col] = col+2;

c\_cpu[row\*N + col] = 0;

c\_gpu[row\*N + col] = 0;

}

dim3 threads\_per\_block (16, 16, 1); // A 16 x 16 block threads

dim3 number\_of\_blocks ((N / threads\_per\_block.x) + 1, (N / threads\_per\_block.y) + 1, 1);

matrixMulGPU <<< number\_of\_blocks, threads\_per\_block >>> ( a, b, c\_gpu );

cudaDeviceSynchronize(); // Wait for the GPU to finish before proceeding

// Call the CPU version to check our work

matrixMulCPU( a, b, c\_cpu );

// Compare the two answers to make sure they are equal

bool error = false;

for( int row = 0; row < N && !error; ++row )

for( int col = 0; col < N && !error; ++col )

if (c\_cpu[row \* N + col] != c\_gpu[row \* N + col])

{

printf("FOUND ERROR at c[%d][%d]\n", row, col);

error = true;

break;

}

if (!error)

printf("Success!\n");

// Free all our allocated memory

cudaFree(a);

cudaFree(b);

cudaFree( c\_cpu );

cudaFree( c\_gpu );

}

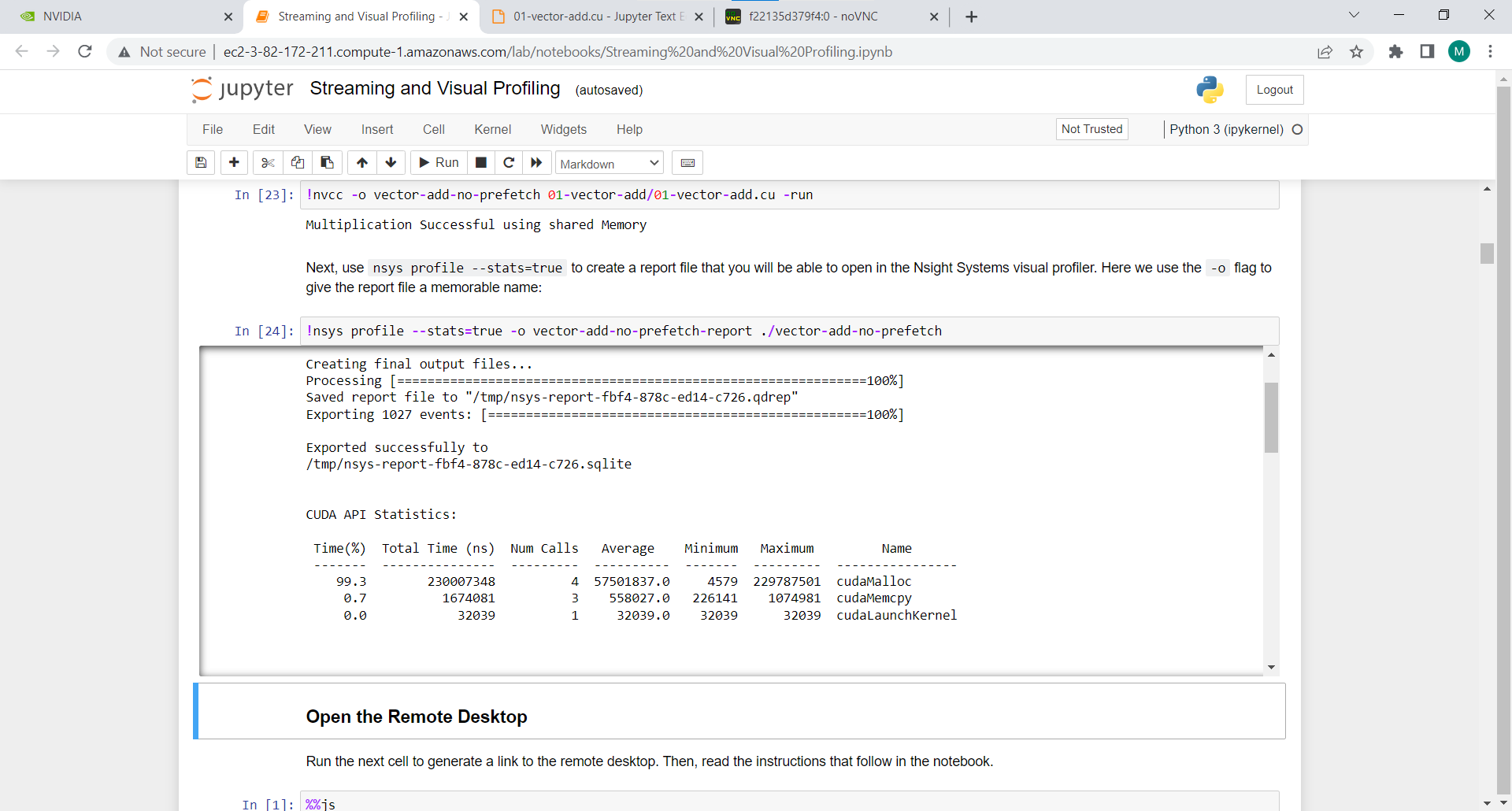
Github Link :

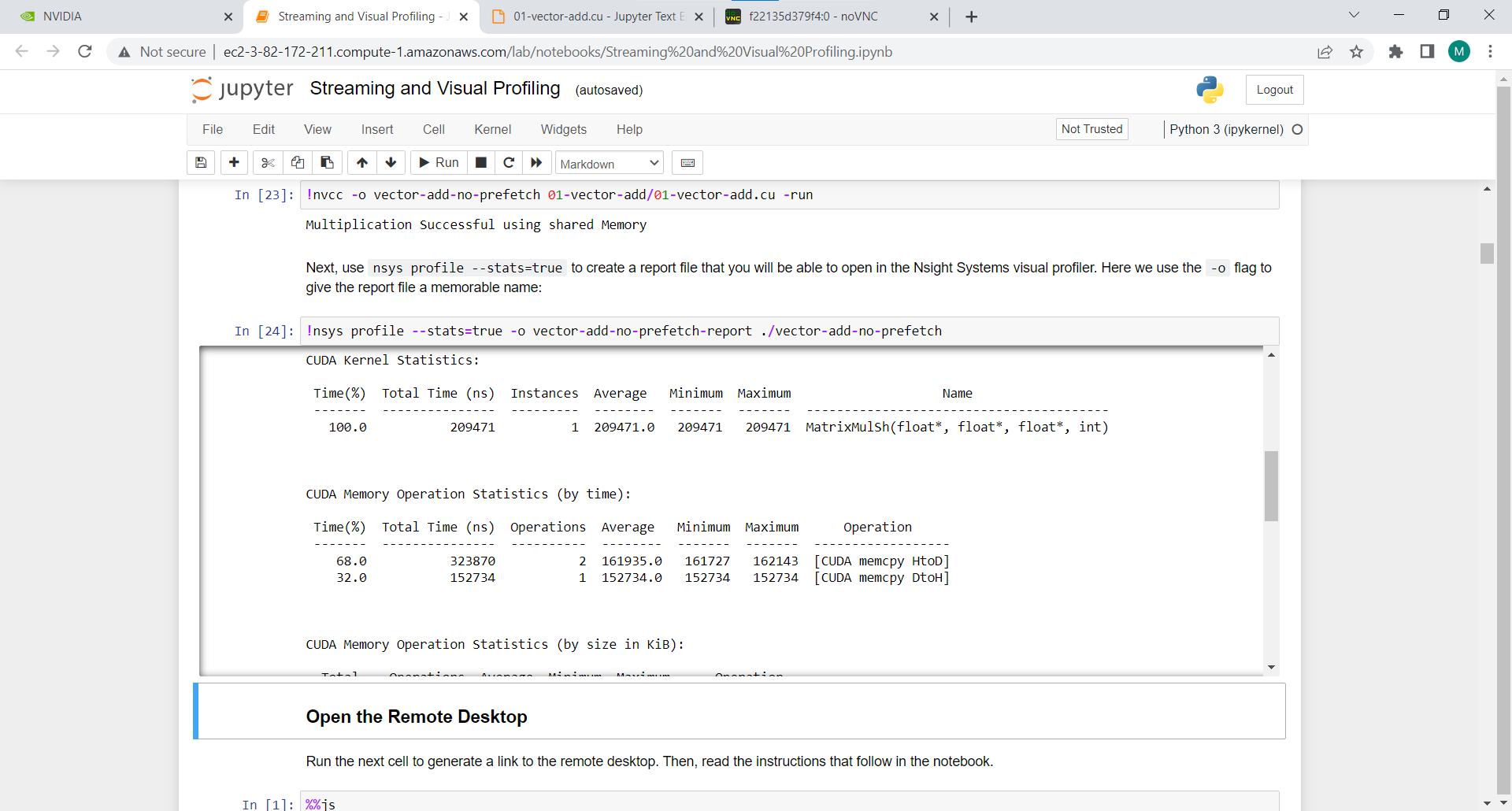
**Problem Statement 2:**

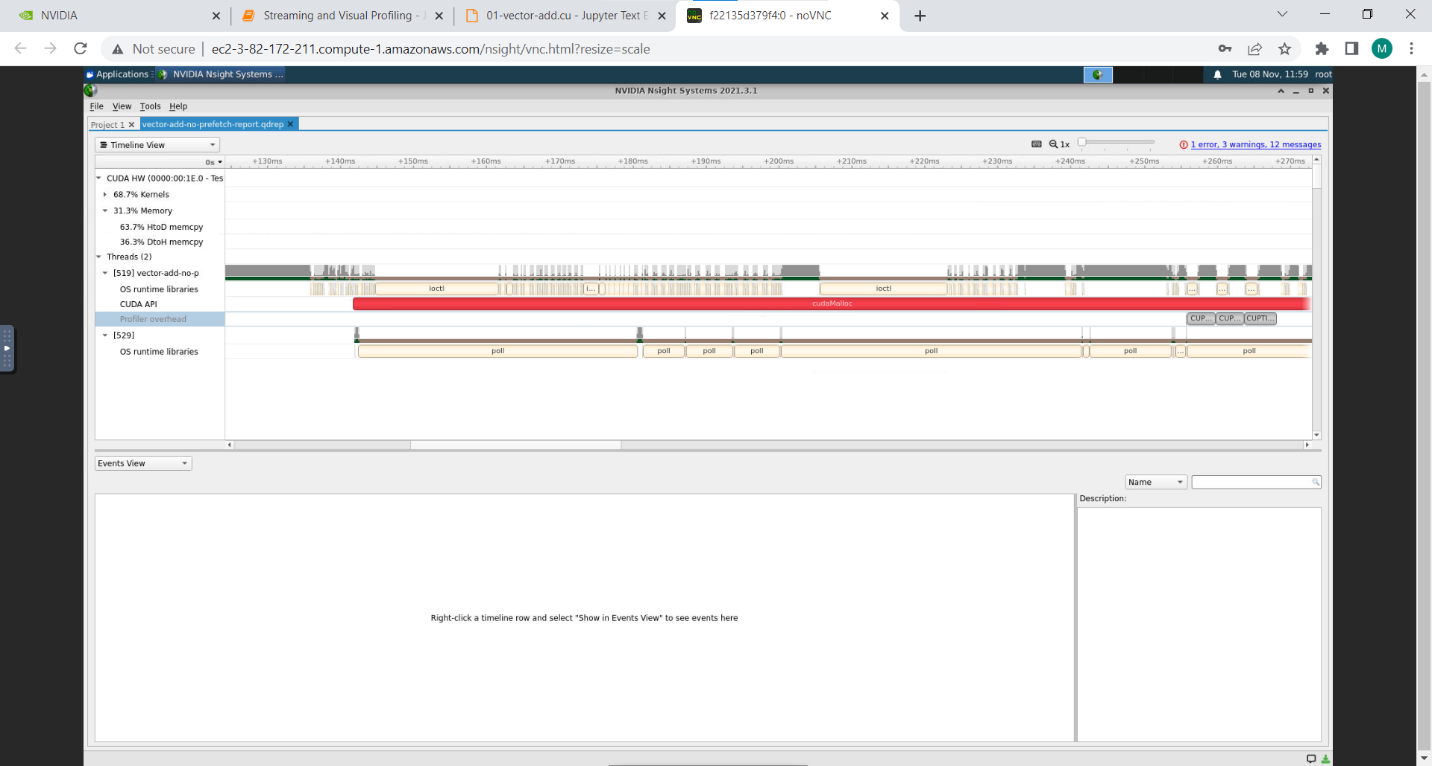
Implement Matrix-Matrix Multiplication using shared memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

**Screenshot #:**

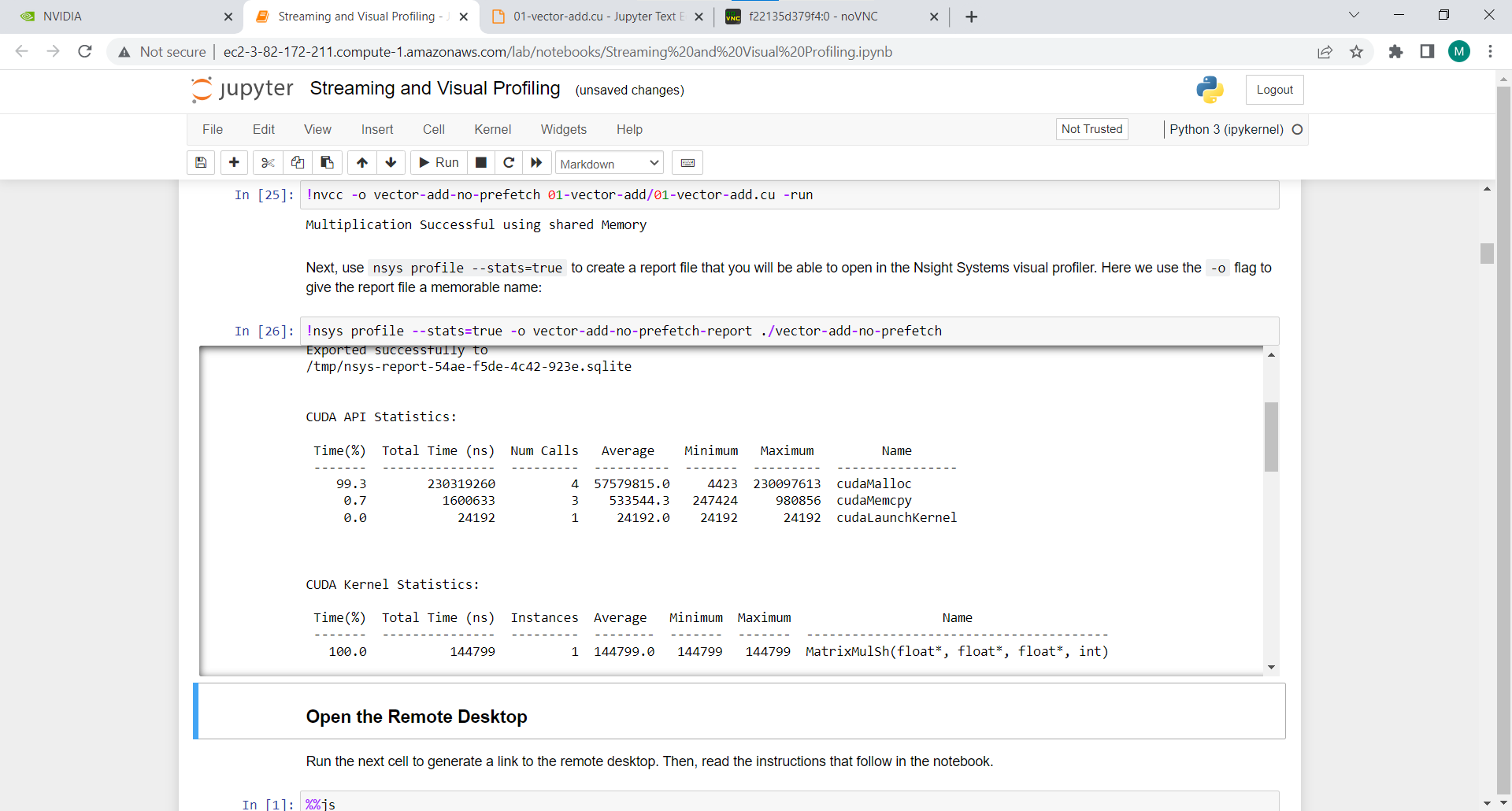
Applying 512 blocks with 32 threads each block

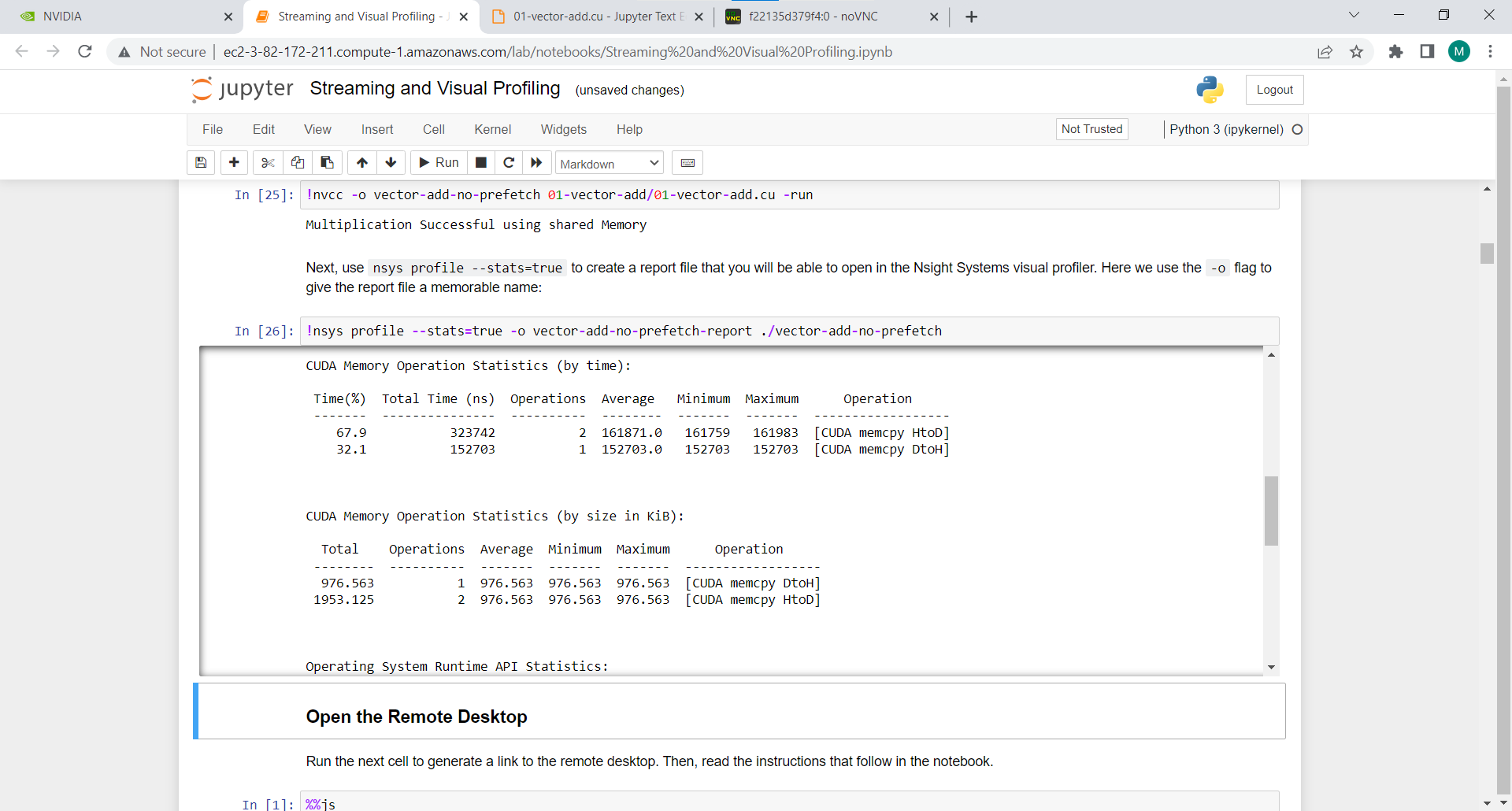






Applying 256 blocks with 16 threads each block





**Information #:**

#include <stdio.h>

#include <math.h>

#define TILE\_WIDTH 2

/\*matrix multiplication kernels\*/

// shared

\_\_global\_\_ void

MatrixMulSh( float \*Md , float \*Nd , float \*Pd , const int WIDTH )

{

//Taking shared array to break the MAtrix in Tile widht and fatch them in that array per ele

\_\_shared\_\_ float Mds [TILE\_WIDTH][TILE\_WIDTH] ;

\_\_shared\_\_ float Nds [TILE\_WIDTH][TILE\_WIDTH] ;

// calculate thread id

unsigned int col = TILE\_WIDTH\*blockIdx.x + threadIdx.x ;

unsigned int row = TILE\_WIDTH\*blockIdx.y + threadIdx.y ;

for (int m = 0 ; m<WIDTH/TILE\_WIDTH ; m++ ) // m indicate number of phase

{

Mds[threadIdx.y][threadIdx.x] = Md[row\*WIDTH + (m\*TILE\_WIDTH + threadIdx.x)] ;

Nds[threadIdx.y][threadIdx.x] = Nd[ ( m\*TILE\_WIDTH + threadIdx.y) \* WIDTH + col] ;

\_\_syncthreads() ; // for syncronizeing the threads

// Do for tile

for ( int k = 0; k<TILE\_WIDTH ; k++ )

Pd[row\*WIDTH + col]+= Mds[threadIdx.x][k] \* Nds[k][threadIdx.y] ;

\_\_syncthreads() ; // for syncronizeing the threads

}

}

// main routine

int main ()

{

const int WIDTH = 500;

float array1\_h[WIDTH][WIDTH] ,array2\_h[WIDTH][WIDTH], M\_result\_array\_h[WIDTH][WIDTH] ;

float \*array1\_d , \*array2\_d ,\*result\_array\_d ,\*M\_result\_array\_d ; // device array

int i , j ;

//input in host array

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

{

for (j = 0 ; j<WIDTH ; j++ )

{

array1\_h[i][j] = (i + 2\*j) %500 ;

array2\_h[i][j] = (i + 3\*j) %500 ;

}

}

//create device array cudaMalloc ( (void \*\*)&array\_name, sizeofmatrixinbytes) ;

cudaMalloc((void \*\*) &array1\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

cudaMalloc((void \*\*) &array2\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

//copy host array to device array; cudaMemcpy ( dest , source , WIDTH , direction )

cudaMemcpy ( array1\_d , array1\_h , WIDTH\*WIDTH\*sizeof (int) , cudaMemcpyHostToDevice ) ;

cudaMemcpy ( array2\_d , array2\_h , WIDTH\*WIDTH\*sizeof (int) , cudaMemcpyHostToDevice ) ;

//allocating memory for resultent device array

cudaMalloc((void \*\*) &result\_array\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

cudaMalloc((void \*\*) &M\_result\_array\_d , WIDTH\*WIDTH\*sizeof (int) ) ;

MatrixMulSh<<<512,32>>> ( array1\_d , array2\_d ,M\_result\_array\_d , WIDTH) ;

// all gpu function blocked till kernel is working

//copy back result\_array\_d to result\_array\_h

cudaMemcpy(M\_result\_array\_h , M\_result\_array\_d , WIDTH\*WIDTH\*sizeof(int) ,cudaMemcpyDeviceToHost) ;

printf("Multiplication Successful using shared Memory");

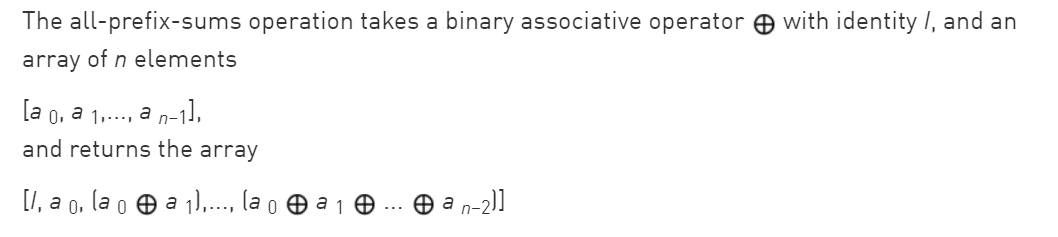
}

**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.

**Screenshot #:**

**SRC : NVIDIA** Blelloch (1990) describes all-prefix-sums as a good example of a computation that seems inherently sequential, but for which there is an efficient parallel algorithm.

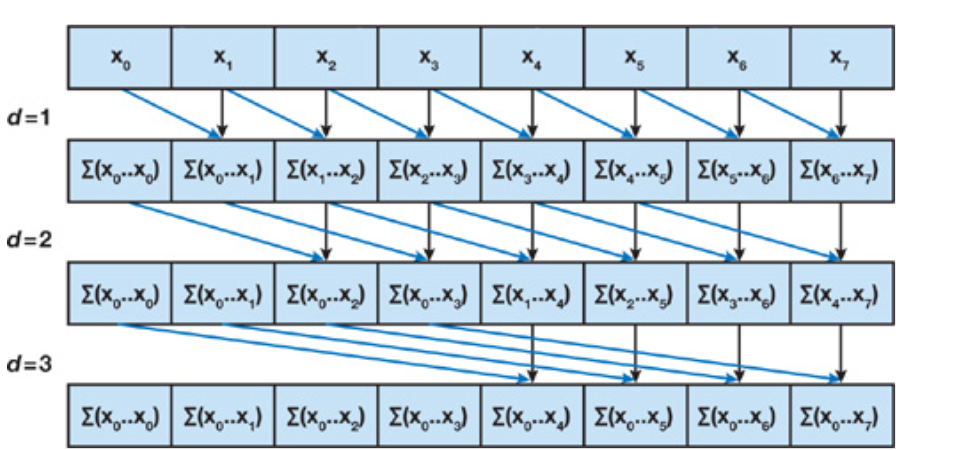
****

out[0] := 0 for k := 1 to n do out[k] := in[k-1] + out[k-1]

This code performs exactly n adds for an array of length n; this is the minimum number of adds required to produce the scanned array.

A parallel computation is work-efficient if it does asymptotically no more work (add operations, in this case) than the sequential version. In other words the two implementations should have the same work complexity, O(n).

Scan Algorithm:



Will works as like :

   \_\_global\_\_ void scan(float \*g\_odata, float \*g\_idata, int n) {

     extern \_\_shared\_\_ float temp[]; *// allocated on invocation*

     int thid = threadIdx.x;   int pout = 0, pin = 1; *// Load input into shared memory.*

*// This is exclusive scan, so shift right by one*

*// and set first element to 0*

        temp[pout\*n + thid] = (thid > 0) ? g\_idata[thid-1] : 0;

        \_\_syncthreads();   for (int offset = 1; offset < n; offset \*= 2)

        {

            pout = 1 - pout;

*// swap double buffer indices*

            pin = 1 - pout;

            if (thid >= offset)

                temp[pout\*n+thid] += temp[pin\*n+thid - offset];

            else  temp[pout\*n+thid] = temp[pin\*n+thid];

            \_\_syncthreads();

        }

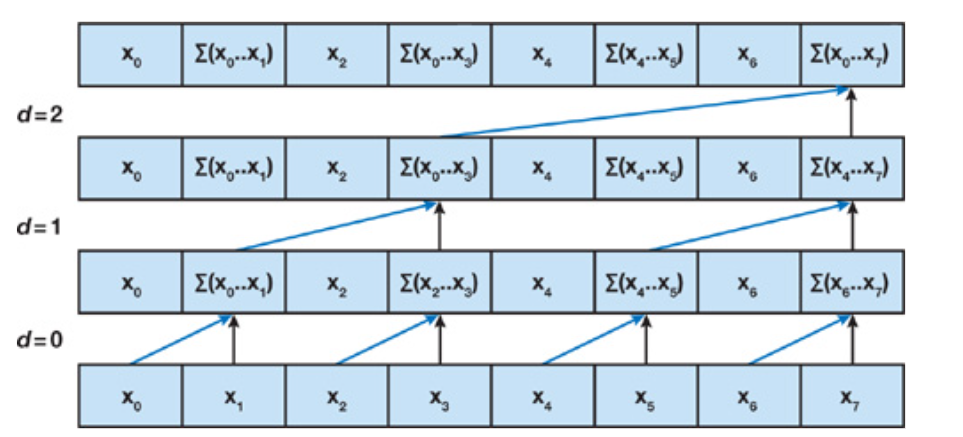
        g\_odata[thid] = temp[pout\*n+thid];

*// write output*

    }

**Another more efficient algorithm is :**

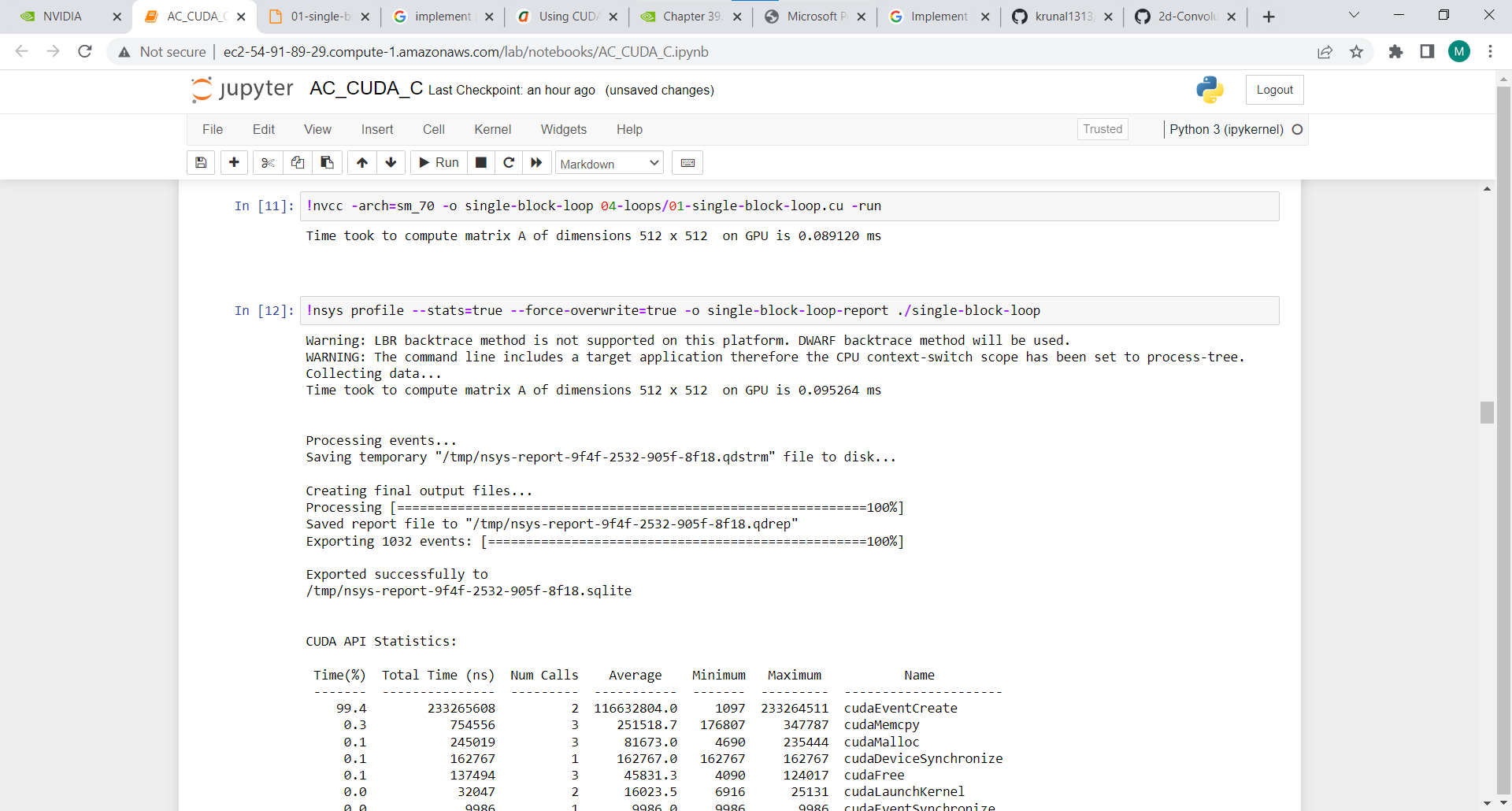
**The Up-Sweep (Reduce) Phase of a Work-Efficient Sum Scan Algorithm**

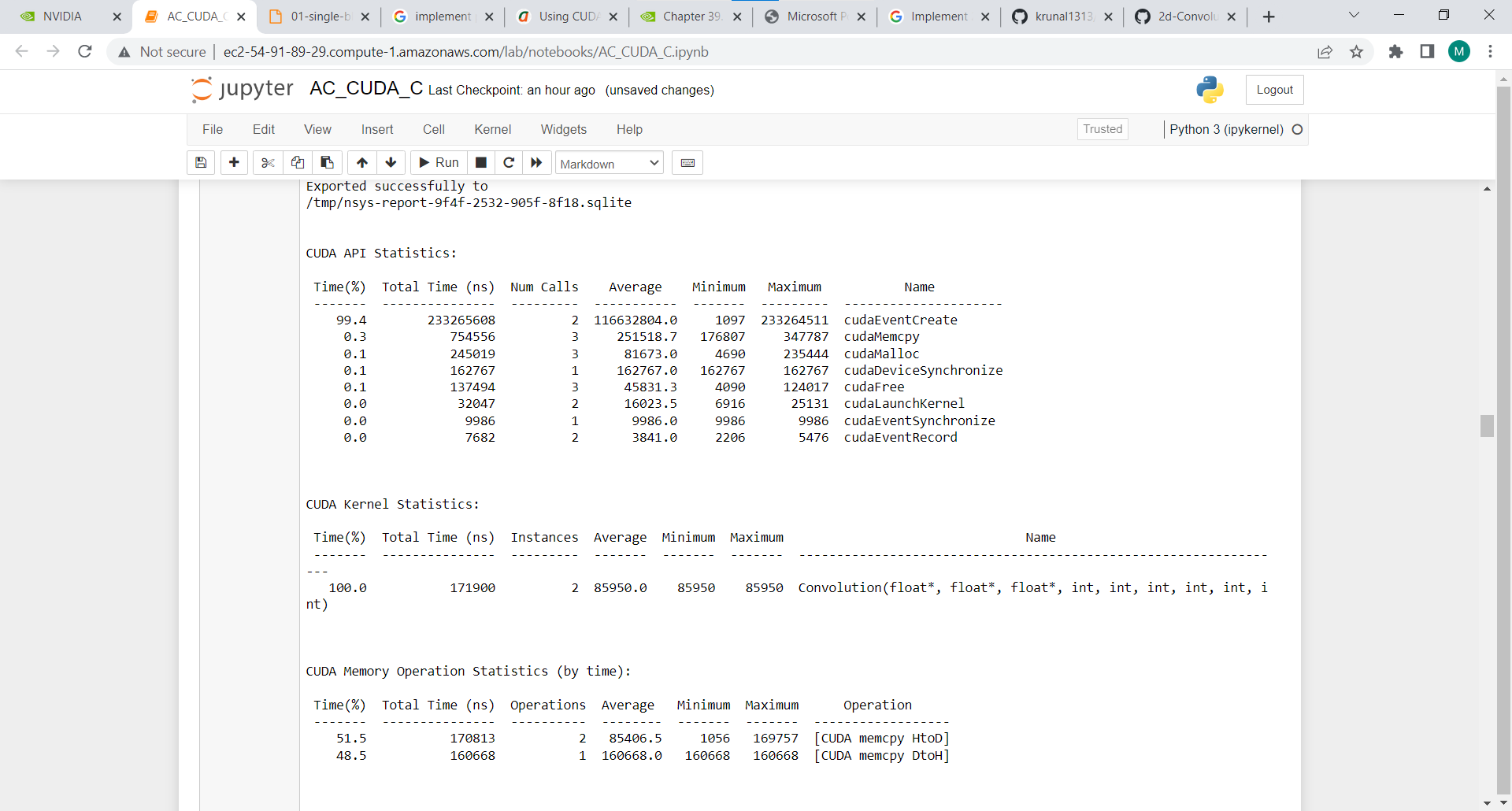


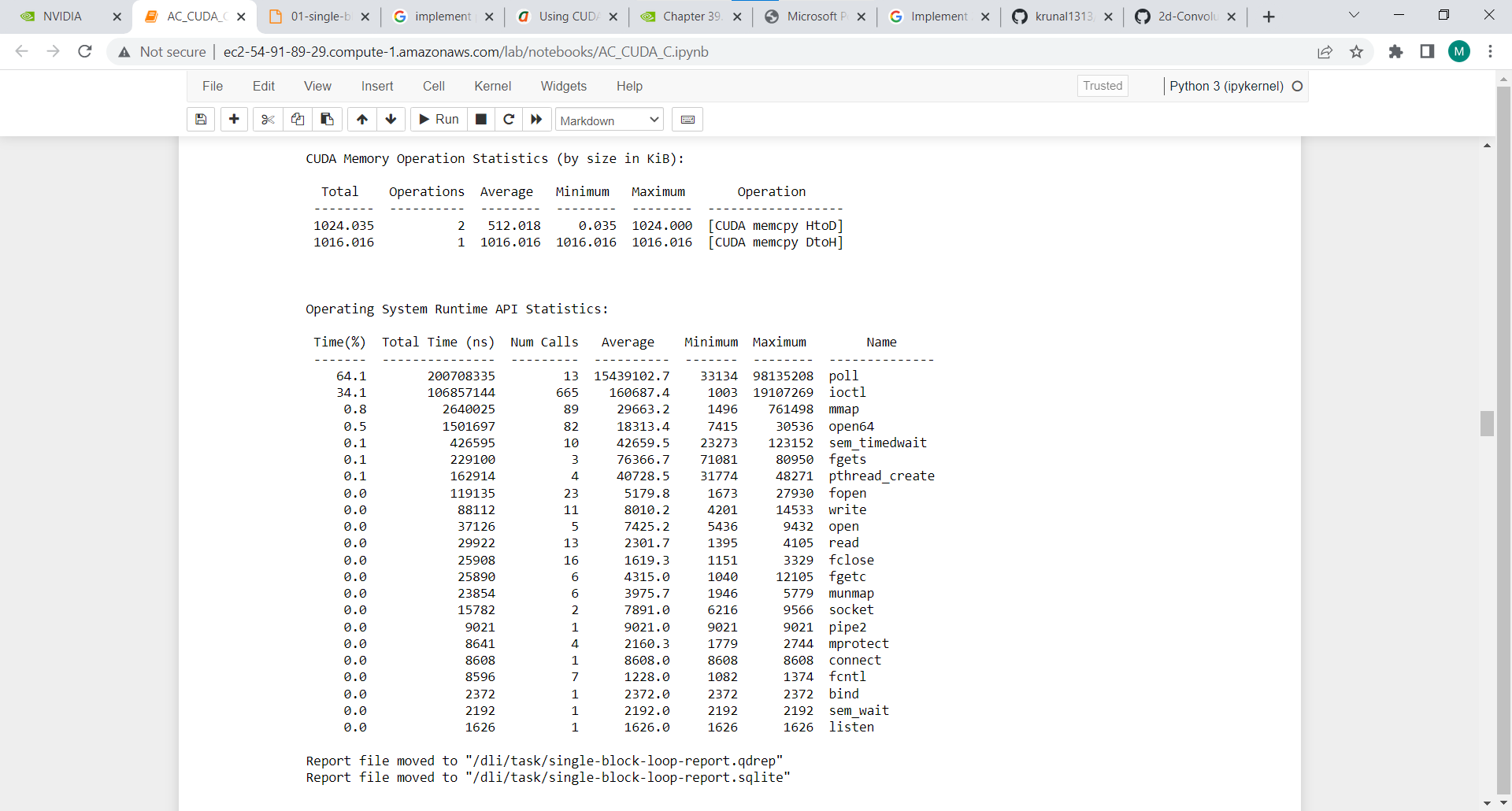
**Problem Statement 4:**

Implement 2D Convolution using shared memory 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.

**Screenshot #:**







**Information #:**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <cstdlib>

#include <time.h>

#define BLOCK\_SIZE 32

#define WA 512

#define HA 512

#define HC 3

#define WC 3

#define WB (WA - WC + 1)

#define HB (HA - HC + 1)

\_\_global\_\_ void Convolution(float\* A, float\* B, float\* C, int numARows, int numACols, int numBRows, int numBCols, int numCRows, int numCCols)

{

int col = blockIdx.x \* (BLOCK\_SIZE - WC + 1) + threadIdx.x;

int row = blockIdx.y \* (BLOCK\_SIZE - WC + 1) + threadIdx.y;

int row\_i = row - WC + 1;

int col\_i = col - WC + 1;

float tmp = 0;

\_\_shared\_\_ float shm[BLOCK\_SIZE][BLOCK\_SIZE];

if (row\_i < WA && row\_i >= 0 && col\_i < WA && col\_i >= 0)

{

shm[threadIdx.y][threadIdx.x] = A[col\_i \* WA + row\_i];

}

else

{

shm[threadIdx.y][threadIdx.x] = 0;

}

\_\_syncthreads();

if (threadIdx.y < (BLOCK\_SIZE - WC + 1) && threadIdx.x < (BLOCK\_SIZE - WC + 1) && row < (WB - WC + 1) && col < (WB - WC + 1))

{

for (int i = 0; i< WC;i++)

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

tmp += shm[threadIdx.y + i][threadIdx.x + j] \* C[j\*WC + i];

B[col\*WB + row] = tmp;

}

}

void randomInit(float\* data, int size)

{

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

data[i] = rand() / (float)RAND\_MAX;

}

int main(int argc, char\*\* argv)

{

srand(2006);

cudaError\_t error;

cudaEvent\_t start\_G, stop\_G;

cudaEventCreate(&start\_G);

cudaEventCreate(&stop\_G);

unsigned int size\_A = WA \* HA;

unsigned int mem\_size\_A = sizeof(float) \* size\_A;

float\* h\_A = (float\*)malloc(mem\_size\_A);

unsigned int size\_B = WB \* HB;

unsigned int mem\_size\_B = sizeof(float) \* size\_B;

float\* h\_B = (float\*)malloc(mem\_size\_B);

unsigned int size\_C = WC \* HC;

unsigned int mem\_size\_C = sizeof(float) \* size\_C;

float\* h\_C = (float\*)malloc(mem\_size\_C);

randomInit(h\_A, size\_A);

randomInit(h\_C, size\_C);

float\* d\_A;

float\* d\_B;

float\* d\_C;

error = cudaMalloc((void\*\*)&d\_A, mem\_size\_A);

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in cudaMalloc for A\n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

error = cudaMalloc((void\*\*)&d\_B, mem\_size\_B);

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in cudaMalloc for B\n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

error = cudaMalloc((void\*\*)&d\_C, mem\_size\_C);

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in cudaMalloc for C\n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

error = cudaMemcpy(d\_A, h\_A, mem\_size\_A, cudaMemcpyHostToDevice);

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in cudaMemcpy for A\n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

error = cudaMemcpy(d\_C, h\_C, mem\_size\_C, cudaMemcpyHostToDevice);

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in cudaMemcpy for C\n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

dim3 threads(BLOCK\_SIZE, BLOCK\_SIZE);

dim3 grid((WB - 1) / (BLOCK\_SIZE - WC + 1), (WB - 1) / (BLOCK\_SIZE - WC + 1));

Convolution << < grid, threads >> >(d\_A, d\_B, d\_C, HA, WA, HB, WB, HC, WC);

cudaEventRecord(start\_G);

Convolution << < grid, threads >> >(d\_A, d\_B, d\_C, HA, WA, HB, WB, HC, WC);

error = cudaGetLastError();

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in launching kernel\n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

error = cudaDeviceSynchronize();

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in cudaDeviceSynchronize \n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

cudaEventRecord(stop\_G);

cudaEventSynchronize(stop\_G);

error = cudaMemcpy(h\_B, d\_B, mem\_size\_B, cudaMemcpyDeviceToHost);

if (error != cudaSuccess)

{

fprintf(stderr, "GPUassert: %s in cudaMemcpy for B\n", cudaGetErrorString(error));

return EXIT\_FAILURE;

}

float miliseconds = 0;

cudaEventElapsedTime(&miliseconds, start\_G, stop\_G);

printf("Time took to compute matrix A of dimensions %d x %d on GPU is %f ms \n \n \n", WA, HA, miliseconds);

//for (int i = 0;i < HB;i++)

//{

// for (int j = 0;j < WB;j++)

// {

// printf("%f ", h\_B[i\*HB + j]);

// }

// printf("\n");

//}

free(h\_A);

free(h\_B);

free(h\_C);

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

return EXIT\_SUCCESS;

}

**Github Link:**