Walchand College of Engineering, Sangli

Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

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

Course: High Performance Computing Lab

Practical No. 10

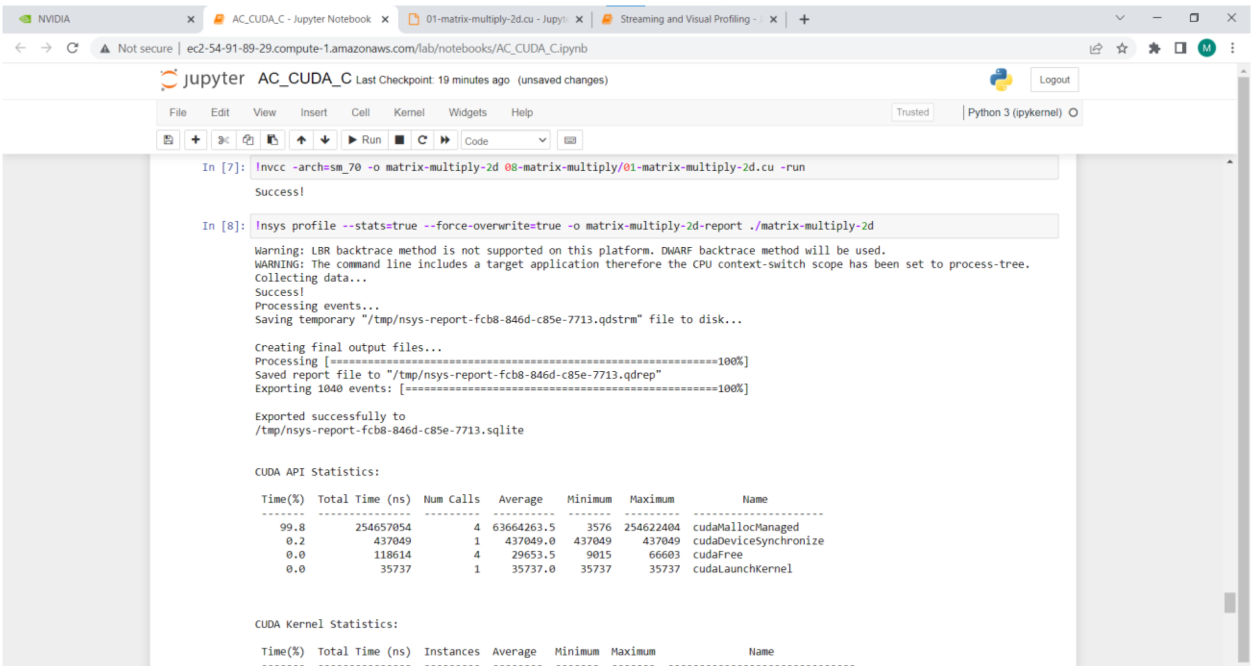
Exam Seat No: 2019BTECS00064

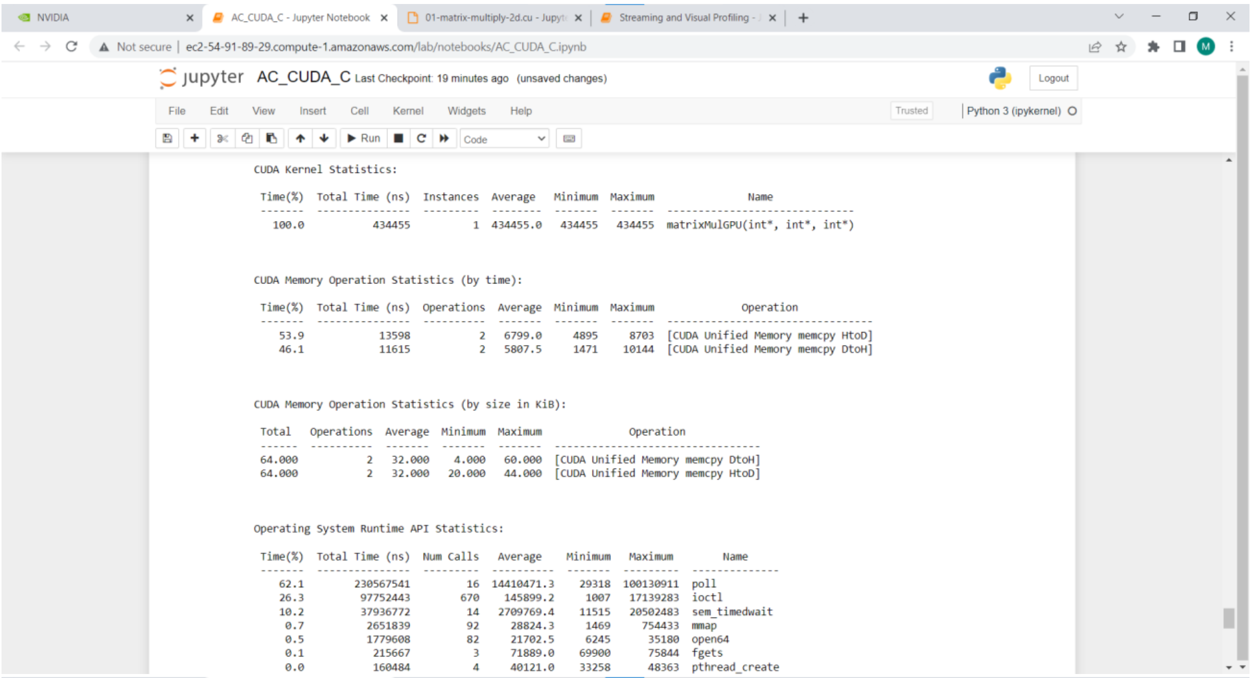
Name – Kunal Santosh Kadam

Problem Statement 1:

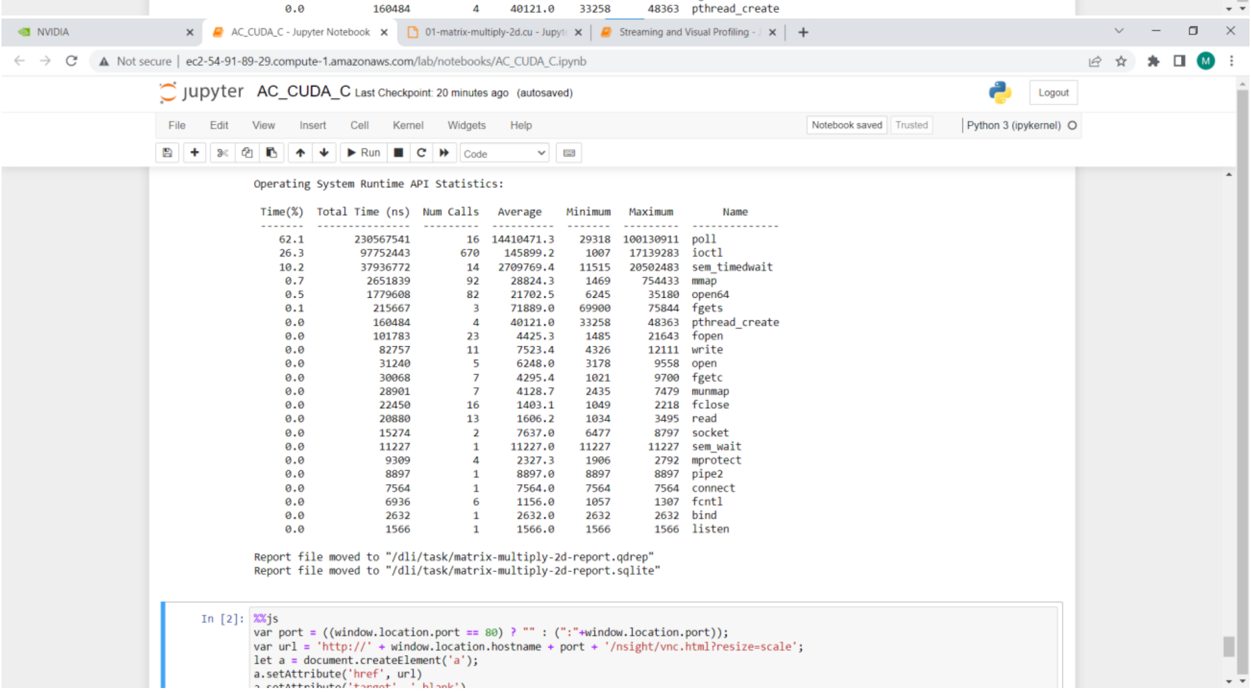
Implement Matrix-matrix Multiplication using global memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute

Screenshot #:





```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum                 Name
 -------  ---------------  ---------  --------  -------  -------  ------------------------------
   100.0           434455          1  434455.0   434455   434455  matrixMulGPU(int*, int*, int*)


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum              Operation
 -------  ---------------  ----------  -------  -------  -------  ---------------------------------
    53.9            13598           2   6799.0     4895     8703  [CUDA Unified Memory memcpy HtoD]
    46.1            11615           2   5807.5     1471    10144  [CUDA Unified Memory memcpy DtoH]


CUDA Memory Operation Statistics (by size in KiB):

 Total   Operations  Average  Minimum  Maximum              Operation
 ------  ----------  -------  -------  -------  ---------------------------------
 64.000           2   32.000    4.000   60.000  [CUDA Unified Memory memcpy DtoH]
 64.000           2   32.000   20.000   44.000  [CUDA Unified Memory memcpy HtoD]


Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    62.1        230567541         16  14410471.3    29318  100130911  poll
    26.3         97752443        670    145899.2     1007   17139283  ioctl
    10.2         37936772         14   2709769.4    11515   20502483  sem_timedwait
     0.7          2651839         92     28824.3     1469     754433  mmap
     0.5          1779608         82     21702.5     6245      35180  open64
     0.1           215667          3     71889.0    69900      75844  fgets
     0.0           160484          4     40121.0    33258      48363  pthread_create
```



```
Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum        Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------
    62.1        230567541         16  14410471.3    29318  100130911  poll
    26.3         97752443        670    145899.2     1007   17139283  ioctl
    10.2         37936772         14   2709769.4    11515   20502483  sem_timedwait
     0.7          2651839         92     28824.3     1469     754433  mmap
     0.5          1779608         82     21702.5     6245      35180  open64
     0.1           215667          3     71889.0    69900      75844  fgets
     0.0           160484          4     40121.0    33258      48363  pthread_create
     0.0           101783         23      4425.3     1485      21643  fopen
     0.0            82757         11      7523.4     4326      12111  write
     0.0            31240          5      6248.0     3178       9558  open
     0.0            30068          7      4295.4     1021       9700  fgetc
     0.0            28901          7      4128.7     2435       7479  munmap
     0.0            22450         16      1403.1     1049       2218  fclose
     0.0            20880         13      1606.2     1034       3495  read
     0.0            15274          2      7637.0     6477       8797  socket
     0.0            11227          1     11227.0    11227      11227  sem_wait
     0.0             9309          4      2327.3     1906       2792  mprotect
     0.0             8897          1      8897.0     8897       8897  pipe2
     0.0             7564          1      7564.0     7564       7564  connect
     0.0             6936          6      1156.0     1057       1307  fcntl
     0.0             2632          1      2632.0     2632       2632  bind
     0.0             1566          1      1566.0     1566       1566  listen

Report file moved to "/dli/task/matrix-multiply-2d-report.qdrep"
Report file moved to "/dli/task/matrix-multiply-2d-report.sqlite"
```

```
In [2]: %%js
var port = ((window.location.port == 80) ? "" : (":"+window.location.port));
var url = 'http://' + window.location.hostname + port + '/nsight/vnc.html?resize=scale';
let a = document.createElement('a');
a.setAttribute('href', url)
```

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
```

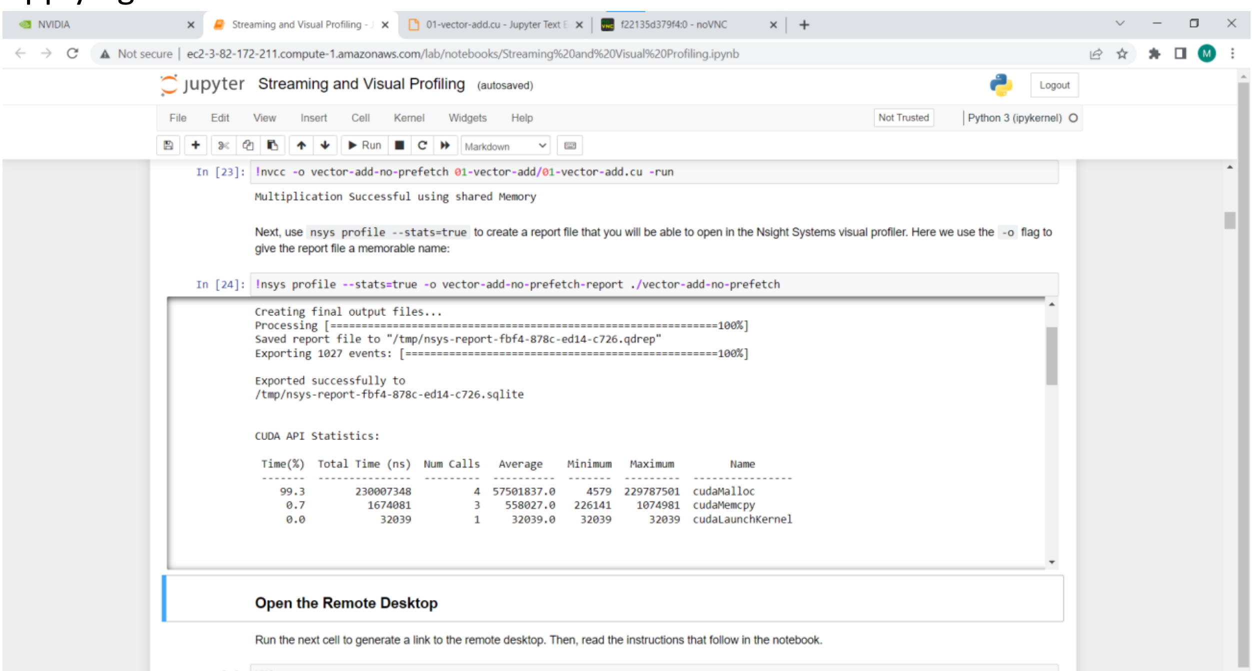
```
    matrixMulCPU( a, b, c_cpu );

    // 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 );
}
```

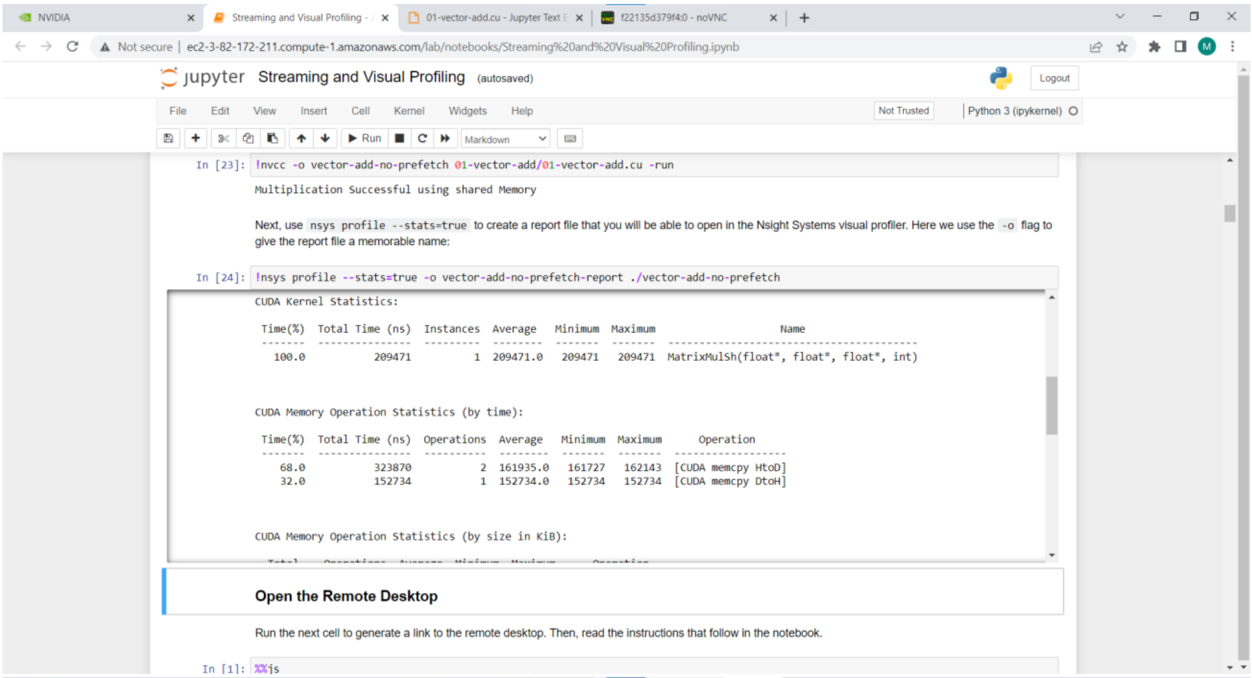
Problem Statement 2:

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.

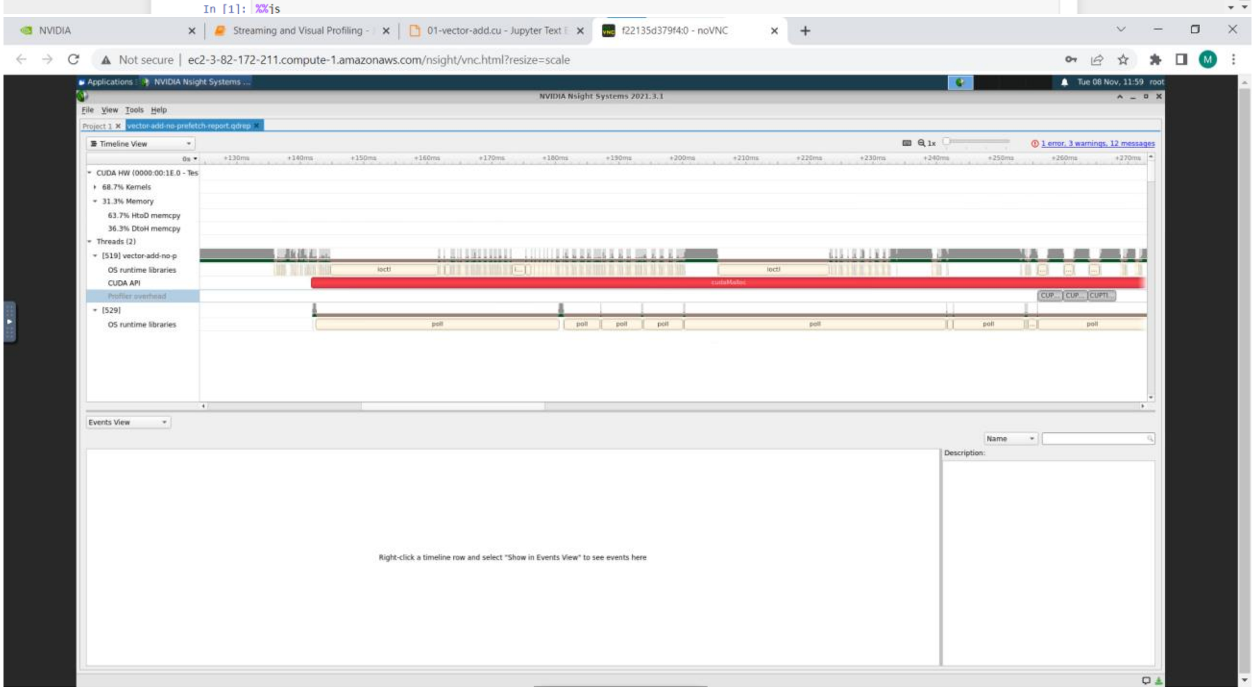
Screenshot #:

Applying 512 blocks with 32 threads each block



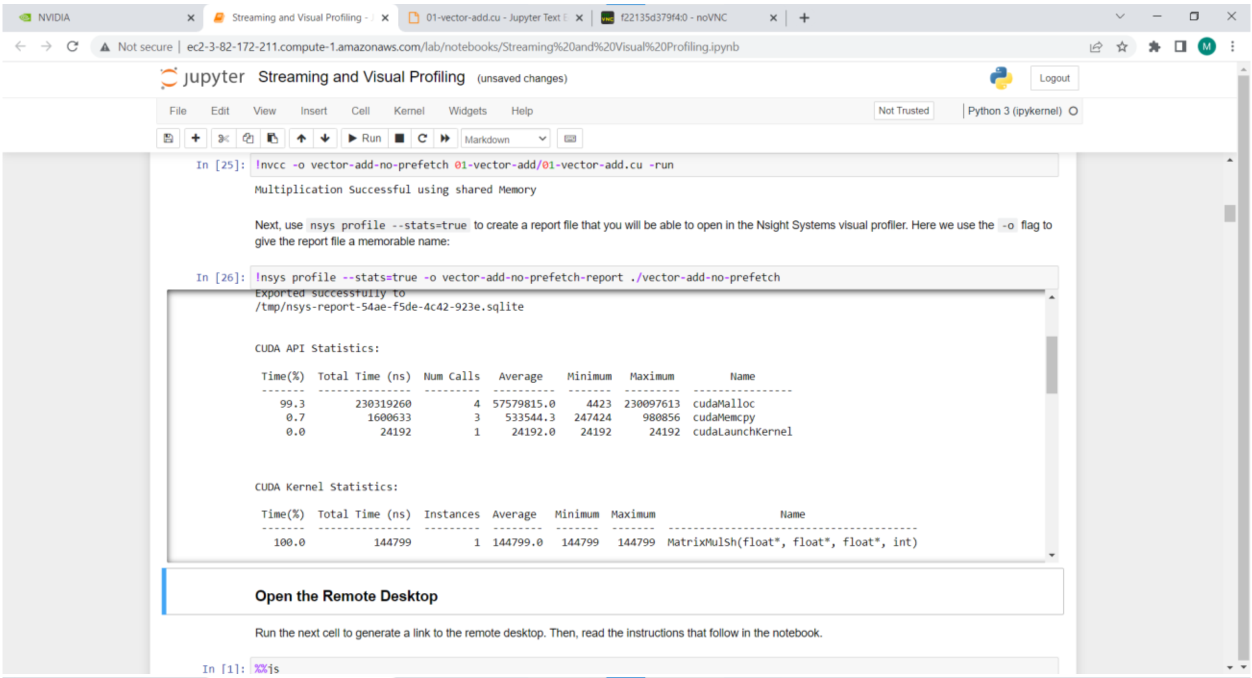


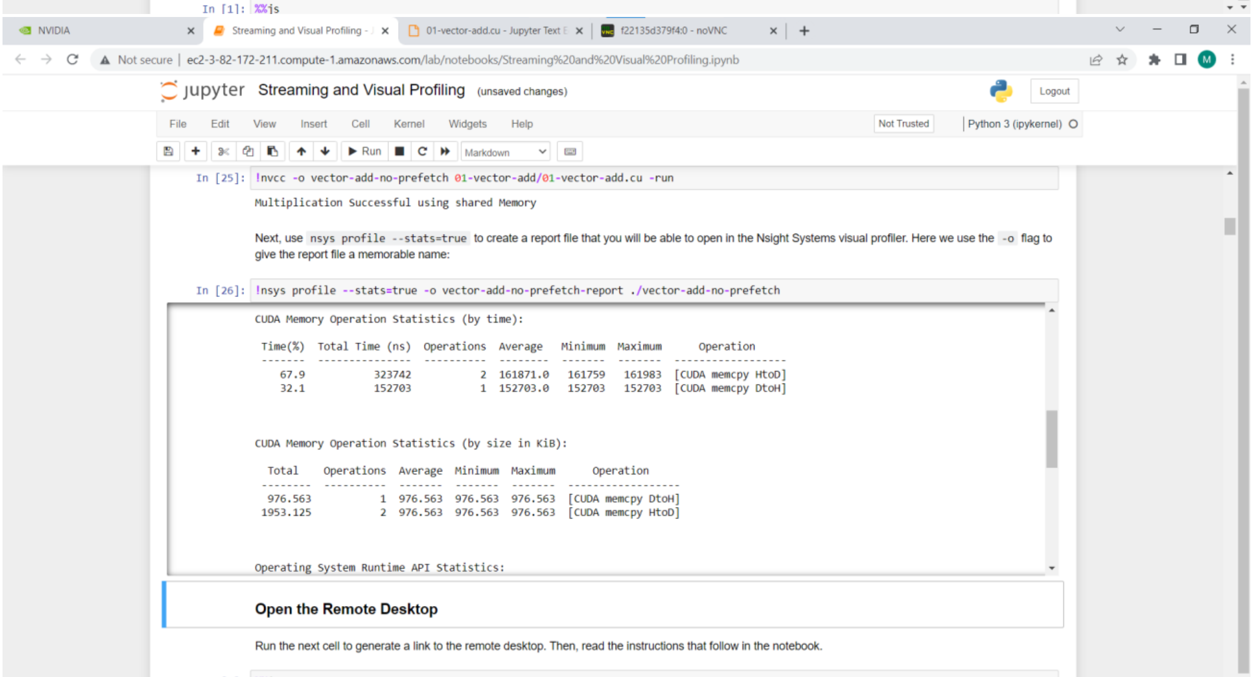
jupyter Streaming and Visual Profiling (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3 (ipykernel)
In [23]: !nvcc -o vector-add-no-prefetch 01-vector-add/01-vector-add.cu -run
Multiplication Successful using shared Memory
Next, use nsys profile --stats=true to create a report file that you will be able to open in the Nsight Systems visual profiler. Here we use the -o flag to give the report file a memorable name:
In [24]: !nsys profile --stats=true -o vector-add-no-prefetch-report ./vector-add-no-prefetch
CUDA Kernel Statistics:
Time(%) Total Time (ns) Instances Average Minimum Maximum Name
100.0 209471 1 209471.0 209471 209471 MatrixMulSh(float*, float*, float*, int)
CUDA Memory Operation Statistics (by time):
Time(%) Total Time (ns) Operations Average Minimum Maximum Operation
68.0 323870 2 161935.0 161727 162143 [CUDA memcpy HtoD]
32.0 152734 1 152734.0 152734 152734 [CUDA memcpy DtoH]
CUDA Memory Operation Statistics (by size in KiB):
Open the Remote Desktop
Run the next cell to generate a link to the remote desktop. Then, read the instructions that follow in the notebook.
In [1]: %%js



NVIDIA Nsight Systems 2021.3.1
File View Tools Help
Timeline View
CUDA HW
68.7% Kernels
31.3% Memory
63.7% HtoD memcpy
36.3% DtoH memcpy
Threads (2)
OS runtime libraries
CUDA API
Events View
Right-click a timeline row and select "Show in Events View" to see events here
Description:

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
```

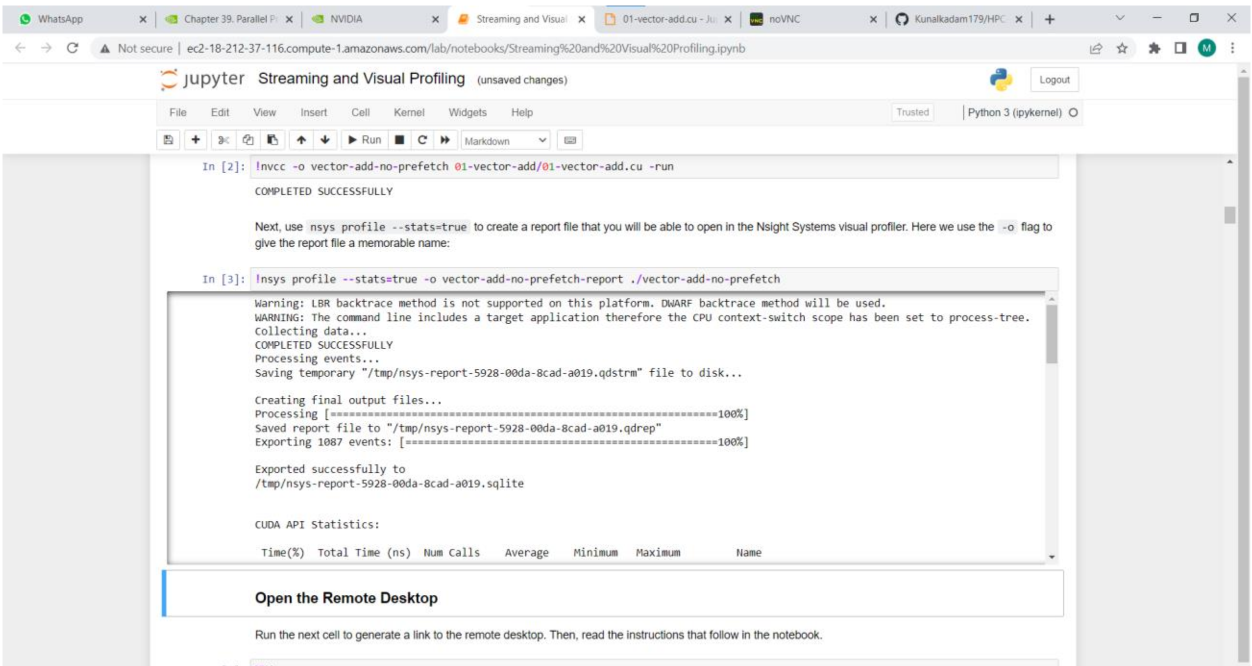
```
                for ( int k = 0; k<TILE_WIDTH ; k++ )
                        Pd[row*WIDTH + col]+= Mds[threadIdx.x][k] *
Nds[k][threadIdx.y] ;
                __syncthreads() ; // for syncronizeing the threads

        }
}

// 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) ) ;
```

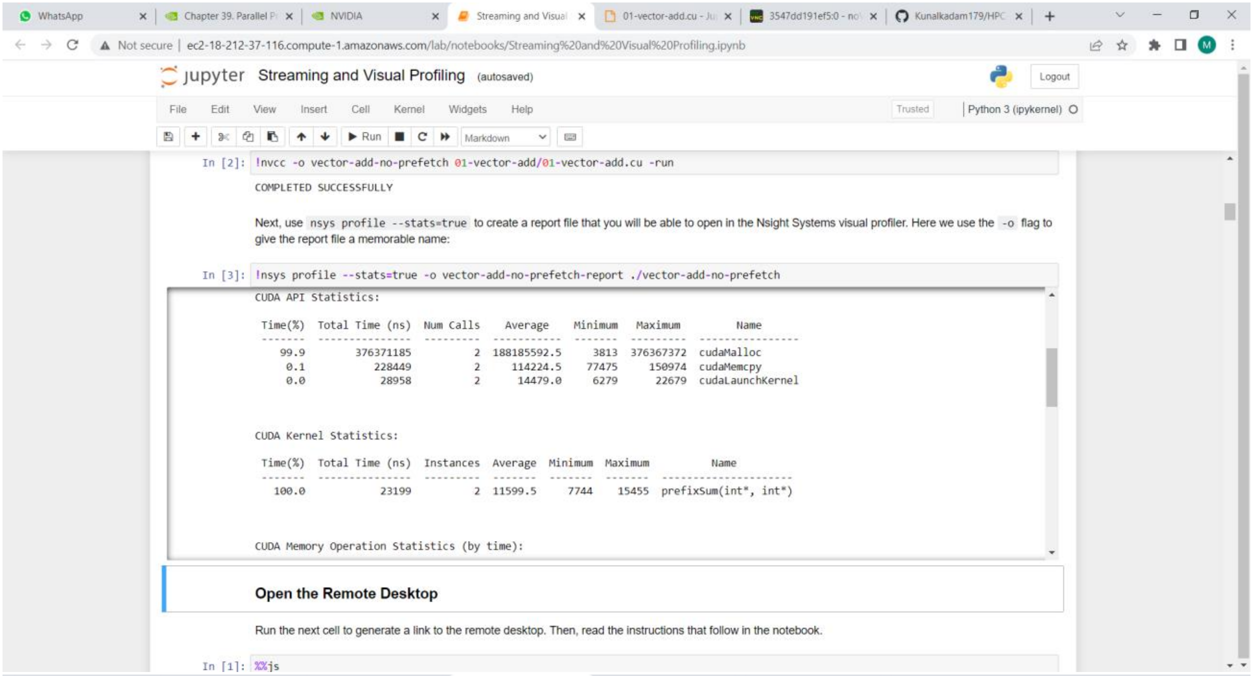
```
    //copy host array to device array; cudaMemcpy ( dest , source ,
WIDTH , direction )

    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");


}
```

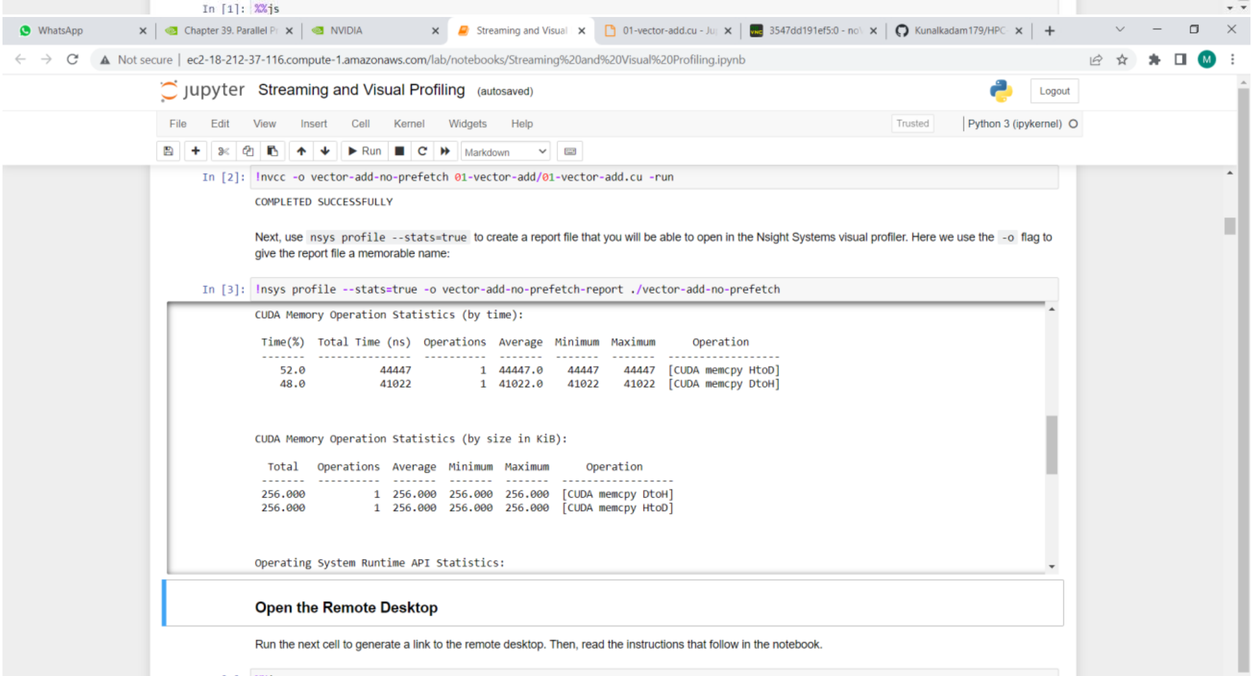
Problem Statement 3:

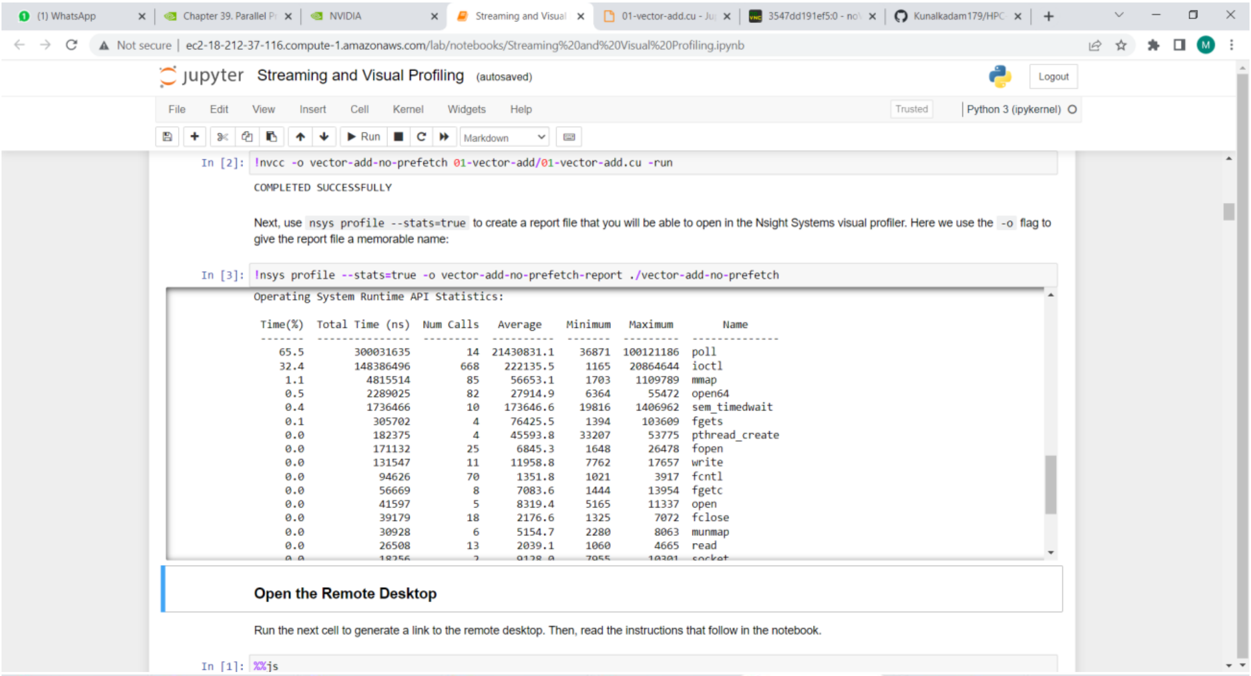
Implement Prefix sum using CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

Screenshot #:




jupyter Streaming and Visual Profiling (autosaved)
In [2]: !nvcc -o vector-add-no-prefetch 01-vector-add/01-vector-add.cu -run
COMPLETED SUCCESSFULLY
Next, use nsys profile --stats=true to create a report file that you will be able to open in the Nsight Systems visual profiler. Here we use the -o flag to give the report file a memorable name:
In [3]: !nsys profile --stats=true -o vector-add-no-prefetch-report ./vector-add-no-prefetch
CUDA API Statistics:
Time(%) Total Time (ns) Num Calls Average Minimum Maximum Name
99.9 376371185 2 188185592.5 3813 376367372 cudaMalloc
0.1 228449 2 114224.5 77475 150974 cudaMemcpy
0.0 28958 2 14479.0 6279 22679 cudaLaunchKernel
CUDA Kernel Statistics:
Time(%) Total Time (ns) Instances Average Minimum Maximum Name
100.0 23199 2 11599.5 7744 15455 prefixSum(int*, int*)
CUDA Memory Operation Statistics (by time):
Open the Remote Desktop
Run the next cell to generate a link to the remote desktop. Then, read the instructions that follow in the notebook.
In [1]: %%js


jupyter Streaming and Visual Profiling (autosaved)
In [2]: !nvcc -o vector-add-no-prefetch 01-vector-add/01-vector-add.cu -run
COMPLETED SUCCESSFULLY
Next, use nsys profile --stats=true to create a report file that you will be able to open in the Nsight Systems visual profiler. Here we use the -o flag to give the report file a memorable name:
In [3]: !nsys profile --stats=true -o vector-add-no-prefetch-report ./vector-add-no-prefetch
CUDA Memory Operation Statistics (by time):
Time(%) Total Time (ns) Operations Average Minimum Maximum Operation
52.0 44447 1 44447.0 44447 44447 [CUDA memcpy HtoD]
48.0 41022 1 41022.0 41022 41022 [CUDA memcpy DtoH]
CUDA Memory Operation Statistics (by size in KiB):
Total Operations Average Minimum Maximum Operation
256.000 1 256.000 256.000 256.000 [CUDA memcpy DtoH]
256.000 1 256.000 256.000 256.000 [CUDA memcpy HtoD]
Operating System Runtime API Statistics:
Open the Remote Desktop
Run the next cell to generate a link to the remote desktop. Then, read the instructions that follow in the notebook.
In [1]: %%js



```
In [2]: !nvcc -o vector-add-no-prefetch 01-vector-add/01-vector-add.cu -run
COMPLETED SUCCESSFULLY
```

Next, use `nsys profile --stats=true` to create a report file that you will be able to open in the Nsight Systems visual profiler. Here we use the `-o` flag to give the report file a memorable name:

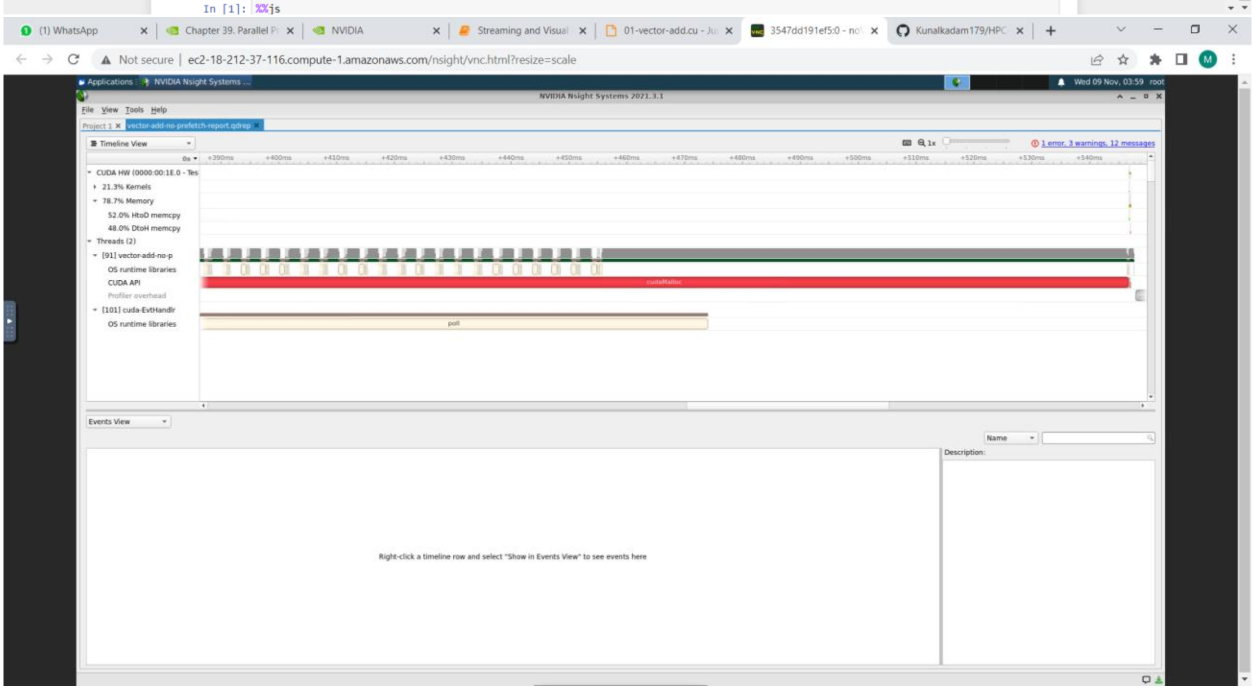
```
In [3]: !nsys profile --stats=true -o vector-add-no-prefetch-report ./vector-add-no-prefetch
Operating System Runtime API Statistics:
```

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
65.5	300031635	14	21430831.1	36871	100121186	poll
32.4	148386496	668	222135.5	1165	20864644	ioctl
1.1	4815514	85	56653.1	1703	1109789	mmap
0.5	2289025	82	27914.9	6364	55472	open64
0.4	1736466	10	173646.6	19816	1406962	sem_timedwait
0.1	305702	4	76425.5	1394	103609	fgets
0.0	182375	4	45593.8	33207	53775	pthread_create
0.0	171132	25	6845.3	1648	26478	fopen
0.0	131547	11	11958.8	7762	17657	write
0.0	94626	70	1351.8	1021	3917	fcntl
0.0	56669	8	7083.6	1444	13954	fgetc
0.0	41597	5	8319.4	5165	11337	open
0.0	39179	18	2176.6	1325	7072	fclose
0.0	30928	6	5154.7	2280	8063	munmap
0.0	26508	13	2039.1	1060	4665	read

Open the Remote Desktop

Run the next cell to generate a link to the remote desktop. Then, read the instructions that follow in the notebook.

```
In [1]: %%js
```



Information #:

```
// This program computes prefix sum with warp divergence
#include <bits/stdc++.h>

using std::accumulate;
using std::generate;
using std::cout;
using std::vector;

#define SHMEM_SIZE 256

__global__ void prefixSum(int *v, int *v_r) {
   // Allocate shared memory
   __shared__ int partial_sum[SHMEM_SIZE];

   // Calculate thread ID
   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   // Load elements into shared memory
   partial_sum[threadIdx.x] = v[tid];
   __syncthreads();

   // Iterate of log base 2 the block dimension
   for (int s = 1; s < blockDim.x; s *= 2) {
      // Reduce the threads performing work by half previous the previous
      // iteration each cycle
      if (threadIdx.x % (2 * s) == 0) {
         partial_sum[threadIdx.x] += partial_sum[threadIdx.x + s];
      }
      __syncthreads();
   }

   // Let the thread 0 for this block write it's result to main memory
```

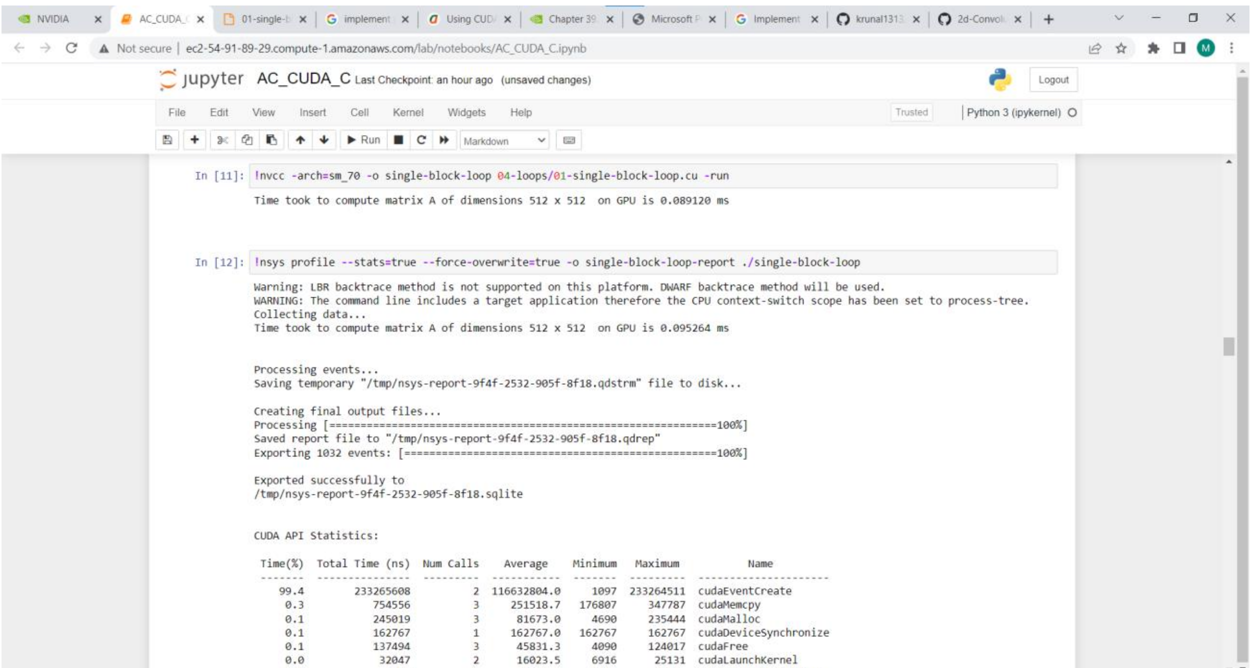
```
    // Result is inexed by this block
    if (threadIdx.x == 0) {
        v_r[blockIdx.x] = partial_sum[0];
    }
}

int main() {
    // Vector size
    int N = 1 << 16;
    size_t bytes = N * sizeof(int);

    // Host data
    vector<int> h_v(N);
    vector<int> h_v_r(N);

  // Initialize the input data
  generate(begin(h_v), end(h_v), [](){ return rand() % 10; });

    // Allocate device memory
    int *d_v, *d_v_r;
    cudaMalloc(&d_v, bytes);
    cudaMalloc(&d_v_r, bytes);

    // Copy to device
    cudaMemcpy(d_v, h_v.data(), bytes, cudaMemcpyHostToDevice);

    // TB Size
    const int TB_SIZE = 256;

    // Grid Size (No padding)
    int GRID_SIZE = N / TB_SIZE;

    // Call kernels
    prefixSum<<<GRID_SIZE, TB_SIZE>>>(d_v, d_v_r);
```

```
    prefixSum<<<1, TB_SIZE>>> (d_v_r, d_v_r);

    // Copy to host;
    cudaMemcpy(h_v_r.data(), d_v_r, bytes, cudaMemcpyDeviceToHost);

    // Print the result
    assert(h_v_r[0] == std::accumulate(begin(h_v), end(h_v), 0));

    cout << "COMPLETED SUCCESSFULLY\n";

    return 0;
}
```

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 #:



```
Exported successfully to
/tmp/nsys-report-9f4f-2532-905f-8f18.sqlite


CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average    Minimum   Maximum           Name
 -------  ---------------  ---------  -----------  -------  ---------  ---------------------
    99.4        233265608          2  116632804.0     1097  233264511  cudaEventCreate
     0.3           754556          3     251518.7   176807     347787  cudaMemcpy
     0.1           245019          3      81673.0     4690     235444  cudaMalloc
     0.1           162767          1     162767.0   162767     162767  cudaDeviceSynchronize
     0.1           137494          3      45831.3     4090     124017  cudaFree
     0.0            32047          2      16023.5     6916      25131  cudaLaunchKernel
     0.0             9986          1       9986.0     9986       9986  cudaEventSynchronize
     0.0             7682          2       3841.0     2206       5476  cudaEventRecord



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum                            Name
 -------  ---------------  ---------  -------  -------  -------  --------------------------------------------------------------------
---
   100.0           171900          2  85950.0    85950    85950  Convolution(float*, float*, float*, int, int, int, int, int, i
nt)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average   Minimum  Maximum      Operation
 -------  ---------------  ----------  --------  -------  -------  ------------------
    51.5           170813           2   85406.5     1056   169757  [CUDA memcpy HtoD]
    48.5           160668           1  160668.0   160668   160668  [CUDA memcpy DtoH]
```

```
CUDA Memory Operation Statistics (by size in KiB):

   Total    Operations  Average   Minimum   Maximum       Operation
 --------  ----------  --------  --------  --------  ------------------
 1024.035           2   512.018     0.035  1024.000  [CUDA memcpy HtoD]
 1016.016           1  1016.016  1016.016  1016.016  [CUDA memcpy DtoH]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average    Minimum  Maximum        Name
 -------  ---------------  ---------  ----------  -------  --------  --------------
    64.1        200708335         13  15439102.7    33134  98135208  poll
    34.1        106857144        665    160687.4     1003  19107269  ioctl
     0.8          2640025         89     29663.2     1496    761498  mmap
     0.5          1501697         82     18313.4     7415     30536  open64
     0.1           426595         10     42659.5    23273    123152  sem_timedwait
     0.1           229100          3     76366.7    71081     80950  fgets
     0.1           162914          4     40728.5    31774     48271  pthread_create
     0.0           119135         23      5179.8     1673     27930  fopen
     0.0            88112         11      8010.2     4201     14533  write
     0.0            37126          5      7425.2     5436      9432  open
     0.0            29922         13      2301.7     1395      4105  read
     0.0            25908         16      1619.3     1151      3329  fclose
     0.0            25890          6      4315.0     1040     12105  fgetc
     0.0            23854          6      3975.7     1946      5779  munmap
     0.0            15782          2      7891.0     6216      9566  socket
     0.0             9021          1      9021.0     9021      9021  pipe2
     0.0             8641          4      2160.3     1779      2744  mprotect
     0.0             8608          1      8608.0     8608      8608  connect
     0.0             8596          7      1228.0     1082      1374  fcntl
     0.0             2372          1      2372.0     2372      2372  bind
     0.0             2192          1      2192.0     2192      2192  sem_wait
     0.0             1626          1      1626.0     1626      1626  listen

Report file moved to "/dli/task/single-block-loop-report.qdrep"
Report file moved to "/dli/task/single-block-loop-report.sqlite"
```

Information #:

```
#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:

https://github.com/Kunalkadam179/HPC-Assignment/tree/main/Assignment%20-%2010