

Class: Final Year (Computer Science and Engineering)

Year: 2022-23

Semester: 1

Course: High Performance Computing Lab

Practical No. 10

Exam Seat No:

2019BTECS00070 – Prathmesh Killedar

Title of practical:

Implementation of Matrix-matrix Multiplication, 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.

Screenshot 1:

```
#include <stdio.h>

#define row1 20
#define col1 30
#define row2 30
#define col2 20

__global__ void matmul(int *l, int *m, int *n)
{
    int x = threadIdx.x;
    int y = threadIdx.y;

    int k;

    n[col2 * y + x] = 0;
```

```
    for (k = 0; k < col1; k++)
    {
        n[col2 * y + x] = n[col2 * y + x] + l[col1 * y + k] * m[col2 *
k + x];
    }
}

int main()
{
    int a[row1][col1];
    int b[row2][col2];
    int c[row1][col2];
    int *d, *e, *f;
    int i, j;

    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < col1; j++)
        {
            a[i][j] = 2;
        }
    }

    for (i = 0; i < row2; i++)
    {
        for (j = 0; j < col2; j++)
```

```
    {  
        b[i][j] = 3;  
    }  
}  
  
cudaMalloc((void **)&d, row1 * col1 * sizeof(int));  
cudaMalloc((void **)&e, row2 * col2 * sizeof(int));  
cudaMalloc((void **)&f, row1 * col2 * sizeof(int));  
  
    cudaMemcpy(d, a, row1 * col1 * sizeof(int),  
cudaMemcpyHostToDevice);  
    cudaMemcpy(e, b, row2 * col2 * sizeof(int),  
cudaMemcpyHostToDevice);  
  
    dim3 threadBlock(col2, row1);  
  
    matmul<<<1, threadBlock>>>(d, e, f);  
    cudaDeviceSynchronize();  
  
    cudaMemcpy(c, f, row1 * col2 * sizeof(int),  
cudaMemcpyDeviceToHost);  
  
    for (i = 0; i < row1; i++)  
    {  
        for (j = 0; j < col2; j++)  
        {
```

```
        if (c[i][j] != 180)
        {
            printf("False\n");
            return -1;
        }
    }
}

cudaFree(d);
cudaFree(e);
cudaFree(f);

printf("True\n");
return 0;
}
```

Screenshot 2:

The screenshot shows a Google Colaboratory notebook interface. The top toolbar includes options for File, Edit, View, Insert, Runtime, Tools, Help, and Saving. The code cell contains the following commands:

```
[1] !nvcc -arch=sm_70 -o globalMatrixMultiplication globalMatrixMultiplication.cu -run
```

The output shows the program executed successfully, printing "True". Below this, the nsys profiling command is executed:

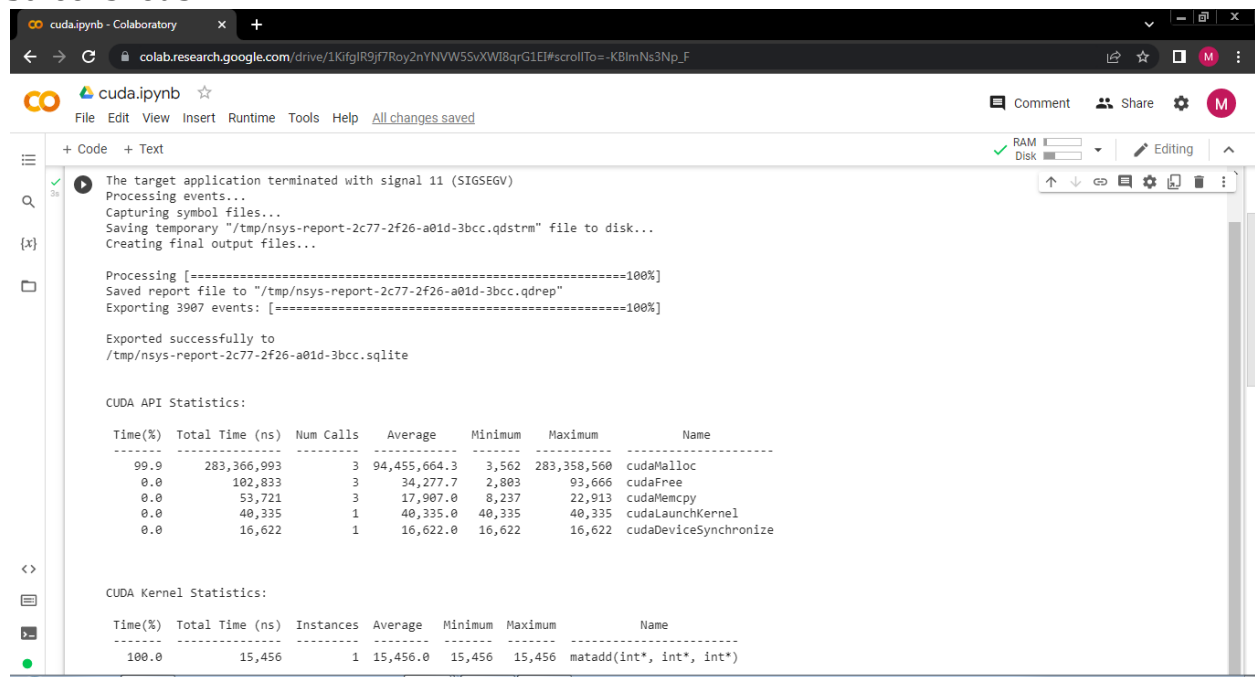
```
!nsys profile --stats=true globalMatrixMultiplication
```

The nsys output includes a warning about the LBR backtrace method, followed by event collection and processing. It reports that the target application terminated with signal 11 (SIGSEGV). The final output shows the export of 3907 events to a SQLite file.

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
99.9	283,366,993	3	94,455,664.3	3,562	283,358,560	cudaMalloc

Screenshot 3:



The target application terminated with signal 11 (SIGSEGV)
Processing events...
Capturing symbol files...
Saving temporary "/tmp/nsys-report-2c77-2f26-a01d-3bcc.qdstrm" file to disk...
Creating final output files...

Processing [=====100%]
Saved report file to "/tmp/nsys-report-2c77-2f26-a01d-3bcc.qdrep"
Exporting 3907 events: [=====100%]

Exported successfully to
/tmp/nsys-report-2c77-2f26-a01d-3bcc.sqlite

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
99.9	283,366,993	3	94,455,664.3	3,562	283,358,560	cudaMalloc
0.0	102,833	3	34,277.7	2,803	93,666	cudaFree
0.0	53,721	3	17,907.0	8,237	22,913	cudaMemcpy
0.0	40,335	1	40,335.0	40,335	40,335	cudaLaunchKernel
0.0	16,622	1	16,622.0	16,622	16,622	cudaDeviceSynchronize

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	15,456	1	15,456.0	15,456	15,456	matadd(int*, int*, int*)

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 4:

```
#include <stdio.h>

#define row1 20
#define col1 30
#define row2 30
#define col2 20

__global__ void matproductsharedmemory(int *l, int *m, int *n)
{
    int x = blockIdx.x;
    int y = blockIdx.y;
    __shared__ int p[col1];
```

```
int i;
int k = threadIdx.x;

n[col2 * y + x] = 0;

p[k] = l[col1 * y + k] * m[col2 * k + x];

__syncthreads();

for (i = 0; i < col1; i++)
    n[col2 * y + x] = n[col2 * y + x] + p[i];
}

int main()
{
    int a[row1][col1];
    int b[row2][col2];
    int c[row1][col2];
    int *d, *e, *f;
    int i, j;

    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < col1; j++)
        {
```

```
        a[i][j] = 2;
    }
}

for (i = 0; i < row2; i++)
{
    for (j = 0; j < col2; j++)
    {
        b[i][j] = 3;
    }
}

cudaMalloc((void **)&d, row1 * col1 * sizeof(int));
cudaMalloc((void **)&e, row2 * col2 * sizeof(int));
cudaMalloc((void **)&f, row1 * col2 * sizeof(int));

cudaMemcpy(d, a, row1 * col1 * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(e, b, row2 * col2 * sizeof(int),
cudaMemcpyHostToDevice);

dim3 grid(col2, row1);

matproductsharedmemory<<<grid, col1>>>(d, e, f);
cudaDeviceSynchronize();
```

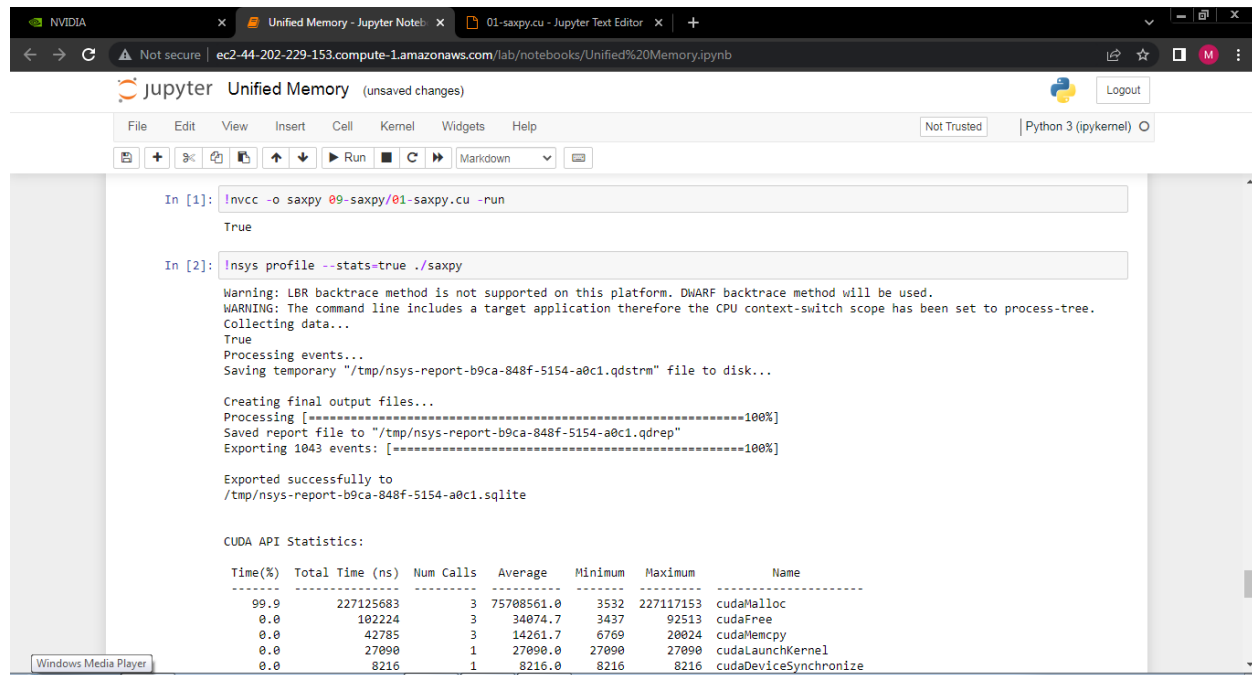
```
    cudaMemcpy(c, f, row1 * col2 * sizeof(int),
cudaMemcpyDeviceToHost);

    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < col2; j++)
        {
            if (c[i][j] != 180)
            {
                printf("False\n");
                return -1;
            }
        }
    }

    cudaFree(d);
    cudaFree(e);
    cudaFree(f);

    printf("True\n");
    return 0;
}
```


Screenshot 5:



The screenshot shows a Jupyter Notebook interface with two input cells. The first cell contains the command `!nvcc -o saxpy 09-saxpy/01-saxpy.cu -run`. The second cell contains `!nsys profile --stats=true ./saxpy`. The output of the second cell displays CUDA API statistics, including a table of time percentages, total times, number of calls, average, minimum, and maximum values for various CUDA functions.

```
In [1]: !nvcc -o saxpy 09-saxpy/01-saxpy.cu -run

True

In [2]: !nsys profile --stats=true ./saxpy

Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-tree.
Collecting data...
True
Processing events...
Saving temporary "/tmp/nsys-report-b9ca-848f-5154-a0c1.qdstrm" file to disk...

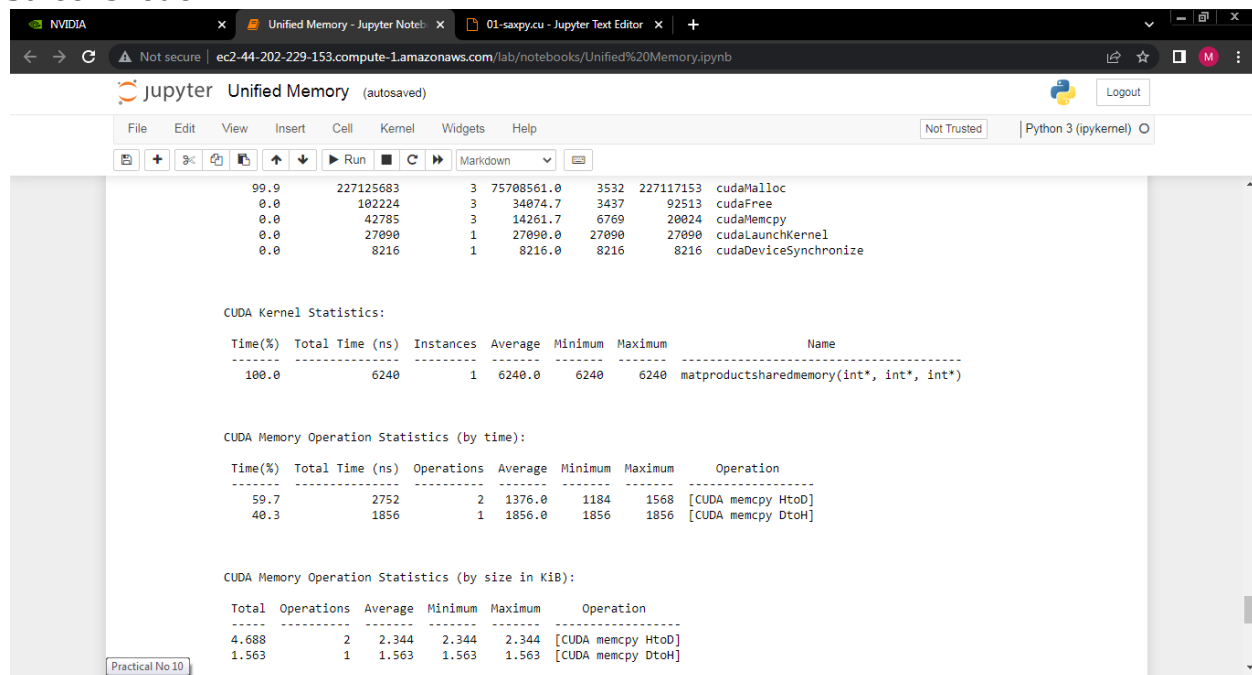
Creating final output files...
Processing [=====100%]
Saved report file to "/tmp/nsys-report-b9ca-848f-5154-a0c1.qdrep"
Exporting 1043 events: [=====100%]

Exported successfully to
/tmp/nsys-report-b9ca-848f-5154-a0c1.sqlite

CUDA API Statistics:

Time(%) Total Time (ns) Num Calls Average Minimum Maximum Name
-----
99.9      227125683      3 75708561.0 3532 227117153 cudaMalloc
0.0       102224      3  34074.7 3437  92513  cudaFree
0.0       42785      3  14261.7 6769  20024  cudaMemcpy
0.0       27090      1  27090.0 27090  27090  cudaLaunchKernel
0.0        8216      1   8216.0 8216   8216  cudaDeviceSynchronize
```

Screenshot 6:



The screenshot shows the same Jupyter Notebook interface as Screenshot 5, but with the output of the `!nsys profile` command expanded to show more detailed statistics, including CUDA Kernel Statistics and CUDA Memory Operation Statistics (by time and by size).

```
99.9      227125683      3 75708561.0 3532 227117153 cudaMalloc
0.0       102224      3  34074.7 3437  92513  cudaFree
0.0       42785      3  14261.7 6769  20024  cudaMemcpy
0.0       27090      1  27090.0 27090  27090  cudaLaunchKernel
0.0        8216      1   8216.0 8216   8216  cudaDeviceSynchronize

CUDA Kernel Statistics:

Time(%) Total Time (ns) Instances Average Minimum Maximum Name
-----
100.0      6240      1  6240.0 6240  6240  matproductsharedmemory(int*, int*, int*)

CUDA Memory Operation Statistics (by time):

Time(%) Total Time (ns) Operations Average Minimum Maximum Operation
-----
59.7       2752      2  1376.0 1184  1568  [CUDA memcpy HtoD]
40.3       1856      1  1856.0 1856  1856  [CUDA memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

Total Operations Average Minimum Maximum Operation
-----
4.688      2  2.344 2.344 2.344  [CUDA memcpy HtoD]
1.563      1  1.563 1.563 1.563  [CUDA memcpy DtoH]
```

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 7:

```
// 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
        // 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 indexed 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);

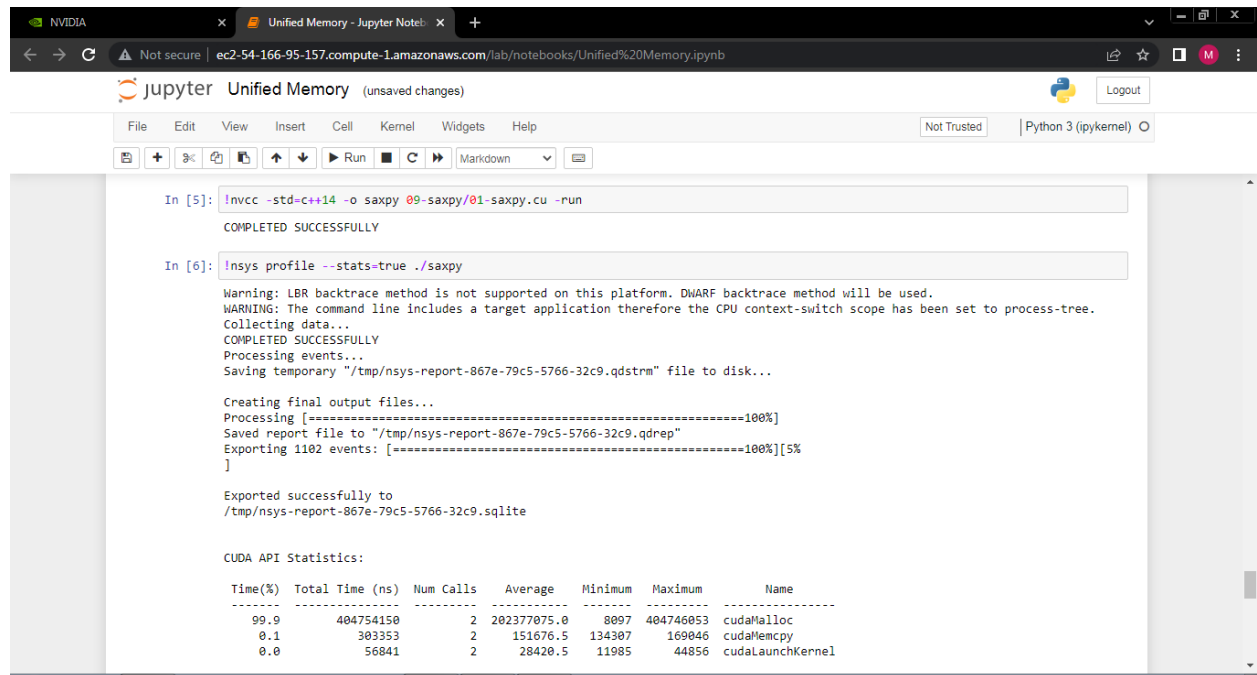

// Print the result

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

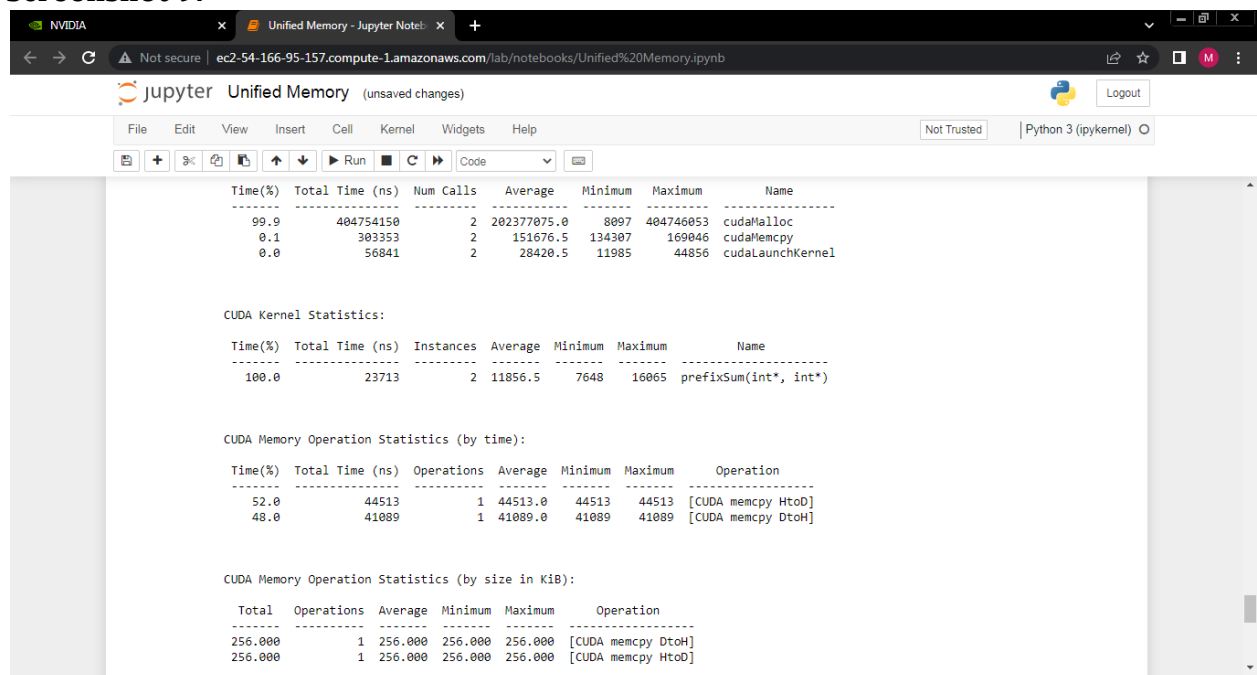
cout << "COMPLETED SUCCESSFULLY\n";
```



Screenshot 8:



Screenshot 9:



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 10:

```
// This program implements 2D convolution in CUDA
#include <bits/stdc++.h>

// 7 x 7 convolutional mask
#define MASK_DIM 7

// Amount the the matrix will hang over the matrix
#define MASK_OFFSET (MASK_DIM / 2)

// Allocate mask in constant memory
__constant__ int mask[7 * 7];

// 2D Convolution Kernel
// Takes:
// matrix: Input matrix
// result: Convolution result
// N:      Dimensions of the matrices
__global__ void convolution_2d(int *matrix, int *result, int N) {
    // Calculate the global thread positions
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Starting index for calculation
```

```
int start_r = row - MASK_OFFSET;
int start_c = col - MASK_OFFSET;

// Temp value for accumulating the result
int temp = 0;

// Iterate over all the rows
for (int i = 0; i < MASK_DIM; i++) {
    // Go over each column
    for (int j = 0; j < MASK_DIM; j++) {
        // Range check for rows
        if ((start_r + i) >= 0 && (start_r + i) < N) {
            // Range check for columns
            if ((start_c + j) >= 0 && (start_c + j) < N) {
                // Accumulate result
                temp += matrix[(start_r + i) * N + (start_c + j)] *
                    mask[i * MASK_DIM + j];
            }
        }
    }
}

// Write back the result
result[row * N + col] = temp;
}
```

```
// Initializes an n x n matrix with random numbers
// Takes:
// m : Pointer to the matrix
// n : Dimension of the matrix (square)
void init_matrix(int *m, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            m[n * i + j] = rand() % 100;
        }
    }
}

// Verifies the 2D convolution result on the CPU
// Takes:
// m:      Original matrix
// mask:   Convolutional mask
// result: Result from the GPU
// N:      Dimensions of the matrix
void verify_result(int *m, int *mask, int *result, int N) {
    // Temp value for accumulating results
    int temp;

    // Intermediate value for more readable code
    int offset_r;
    int offset_c;
```



```
// Go over each row
for (int i = 0; i < N; i++) {
    // Go over each column
    for (int j = 0; j < N; j++) {
        // Reset the temp variable
        temp = 0;

        // Go over each mask row
        for (int k = 0; k < MASK_DIM; k++) {
            // Update offset value for row
            offset_r = i - MASK_OFFSET + k;

            // Go over each mask column
            for (int l = 0; l < MASK_DIM; l++) {
                // Update offset value for column
                offset_c = j - MASK_OFFSET + l;

                // Range checks if we are hanging off the matrix
                if (offset_r >= 0 && offset_r < N) {
                    if (offset_c >= 0 && offset_c < N) {
                        // Accumulate partial results
                        temp += m[offset_r * N + offset_c] * mask[k * MASK_DIM +
1];
                    }
                }
            }
        }
    }
}
```

```
    }  
    // Fail if the results don't match  
    assert(result[i * N + j] == temp);  
}  
}  
}  
  
int main() {  
    // Dimensions of the matrix (2 ^ 10 x 2 ^ 10)  
    int N = 1 << 10;  
  
    // Size of the matrix (in bytes)  
    size_t bytes_n = N * N * sizeof(int);  
  
    // Allocate the matrix and initialize it  
    int *matrix = new int[N * N];  
    int *result = new int[N * N];  
    init_matrix(matrix, N);  
  
    // Size of the mask in bytes  
    size_t bytes_m = MASK_DIM * MASK_DIM * sizeof(int);  
  
    // Allocate the mask and initialize it  
    int *h_mask = new int[MASK_DIM * MASK_DIM];  
    init_matrix(h_mask, MASK_DIM);
```

```
// Allocate device memory

int *d_matrix;
int *d_result;

cudaMalloc(&d_matrix, bytes_n);
cudaMalloc(&d_result, bytes_n);


// Copy data to the device

cudaMemcpy(d_matrix, matrix, bytes_n, cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(mask, h_mask, bytes_m);


// Calculate grid dimensions

int THREADS = 16;
int BLOCKS = (N + THREADS - 1) / THREADS;


// Dimension launch arguments

dim3 block_dim(THREADS, THREADS);
dim3 grid_dim(BLOCKS, BLOCKS);


// Perform 2D Convolution

convolution_2d<<<grid_dim, block_dim>>>(d_matrix, d_result, N);


// Copy the result back to the CPU

cudaMemcpy(result, d_result, bytes_n, cudaMemcpyDeviceToHost);


// Functional test

verify_result(matrix, h_mask, result, N);
```

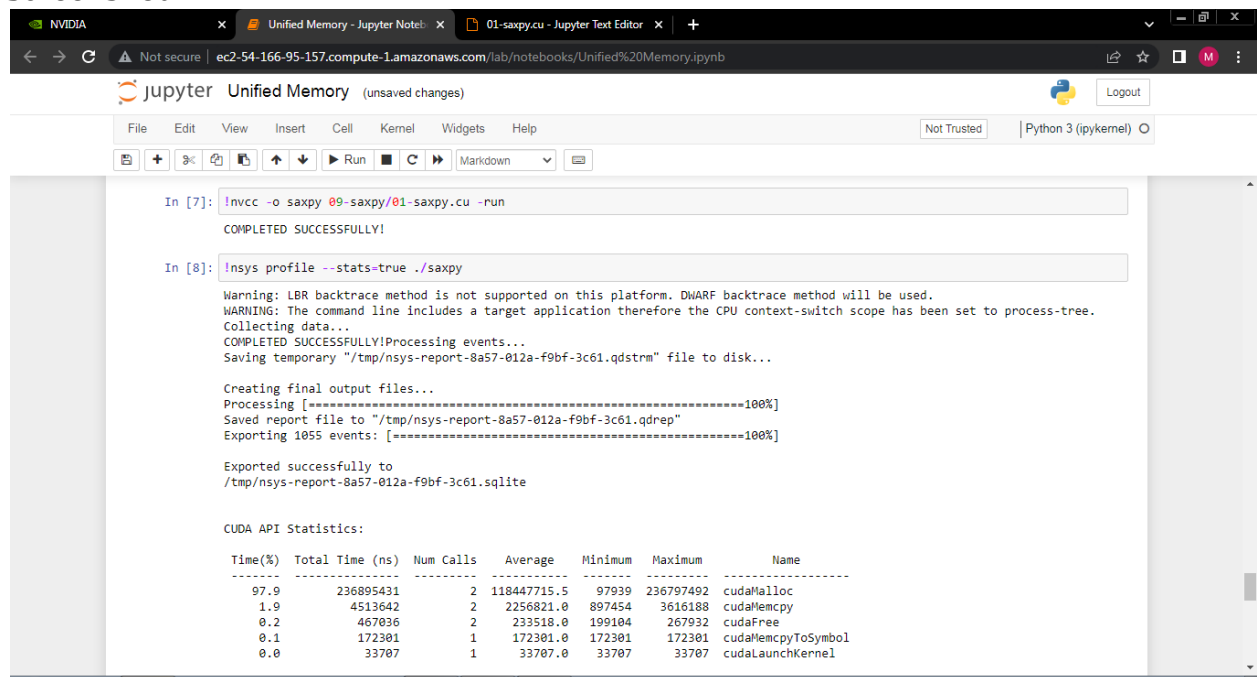
```
std::cout << "COMPLETED SUCCESSFULLY!";

// Free the memory we allocated
delete[] matrix;
delete[] result;
delete[] h_mask;

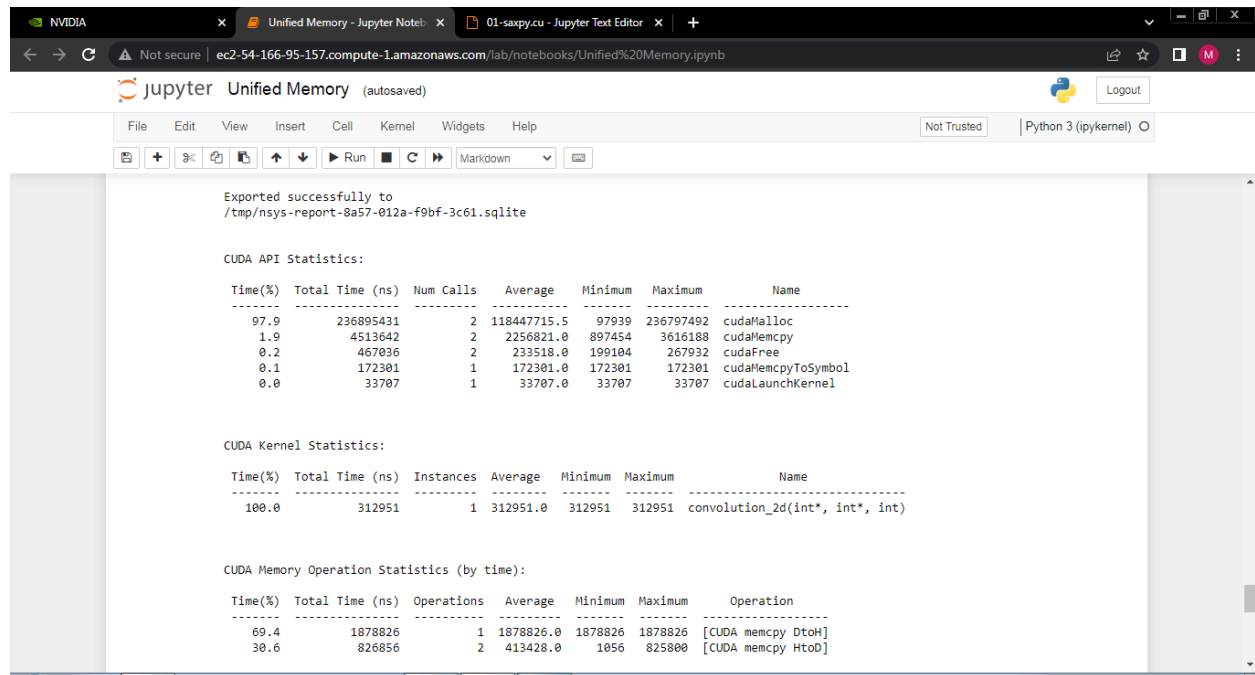
cudaFree(d_matrix);
cudaFree(d_result);

return 0;
}
```

Screenshot 11:



Screenshot 12:



The screenshot shows a Jupyter Notebook interface with a dark theme. The browser address bar indicates the notebook is running on an Amazon EC2 instance. The notebook content displays the following:

```
Exported successfully to
/tmp/nsys-report-8a57-012a-f9bf-3c61.sqlite
```

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
97.9	236895431	2	118447715.5	97939	236797492	cudaMalloc
1.9	4513642	2	2256821.0	897454	3616188	cudaMemcpy
0.2	467036	2	233518.0	199104	267932	cudaFree
0.1	172301	1	172301.0	172301	172301	cudaMemcpyToSymbol
0.0	33707	1	33707.0	33707	33707	cudaLaunchKernel

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	312951	1	312951.0	312951	312951	convolution_2d(int*, int*, int)

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
69.4	1878826	1	1878826.0	1878826	1878826	[CUDA memcpy DtoH]
30.6	826856	2	413428.0	1056	825800	[CUDA memcpy HtoD]

Github Link: