**SEM - VII - 2022-23**

**High Performance Computing Lab**

**Assignment - 10**

**Name: Pavan Krishnat Shinde**

**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.

#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);

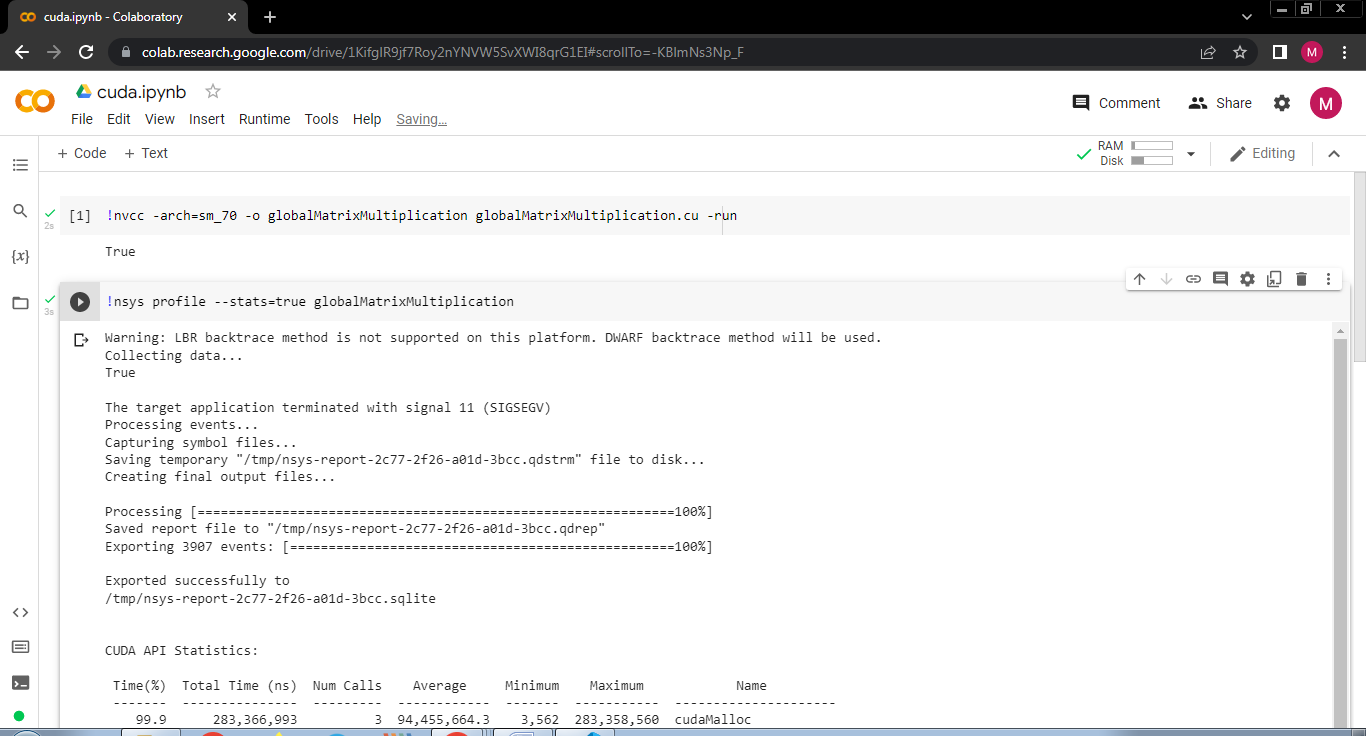
cudaFree(e);

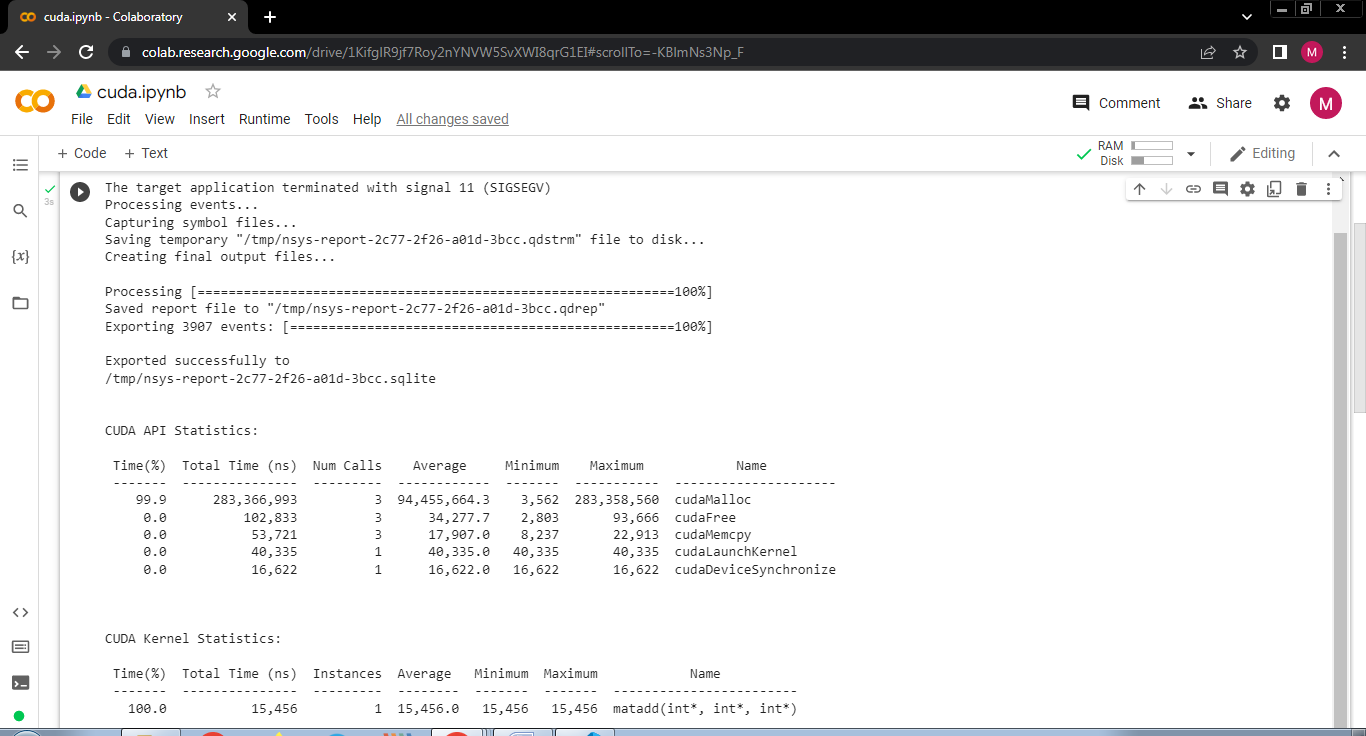
cudaFree(f);

printf("True\n");

return 0;

}





**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.

#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);

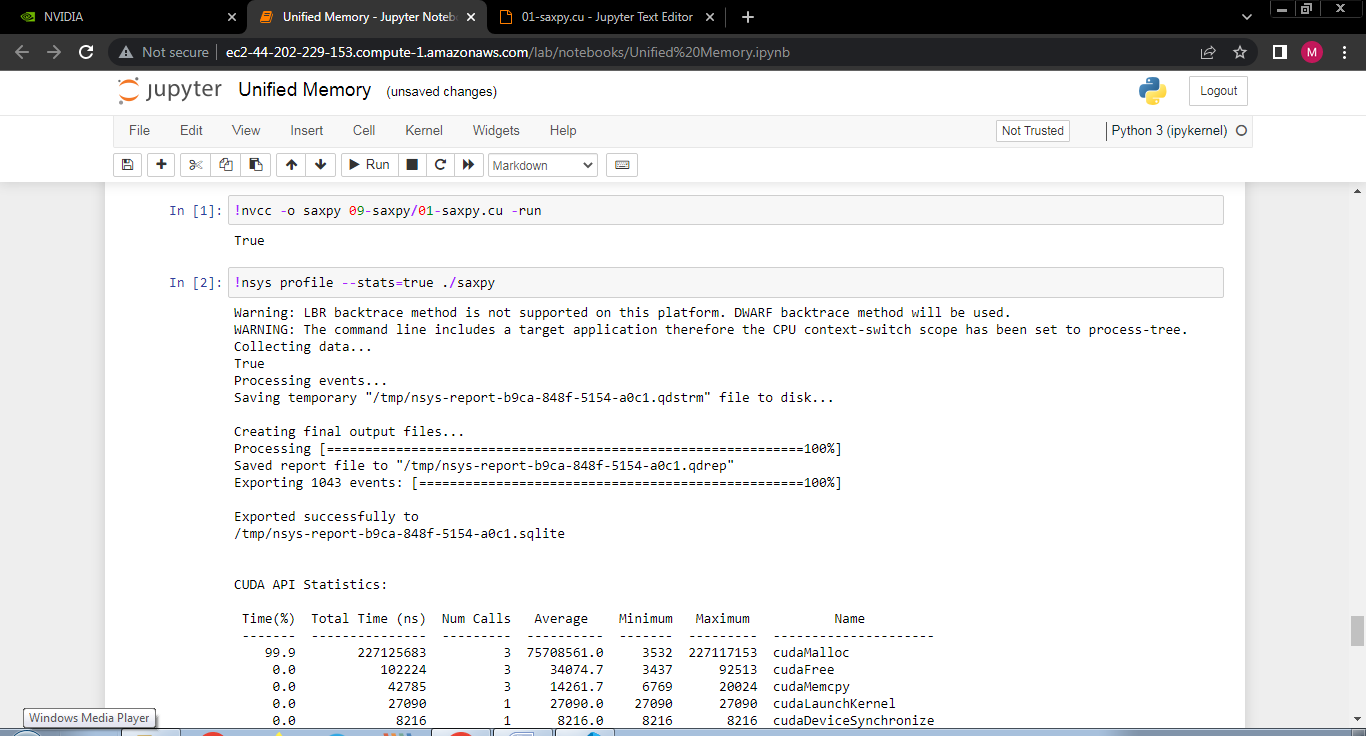
cudaFree(e);

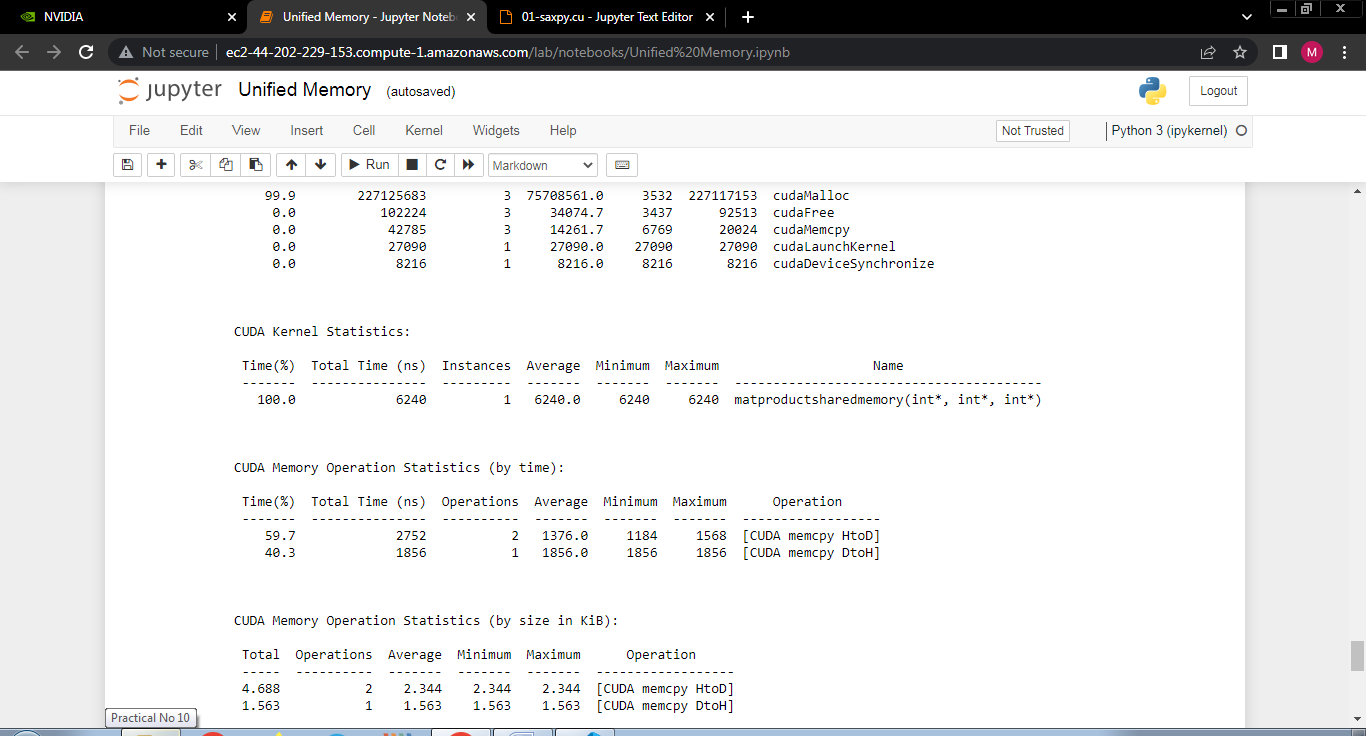
cudaFree(f);

printf("True\n");

return 0;

}





**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.

// 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);

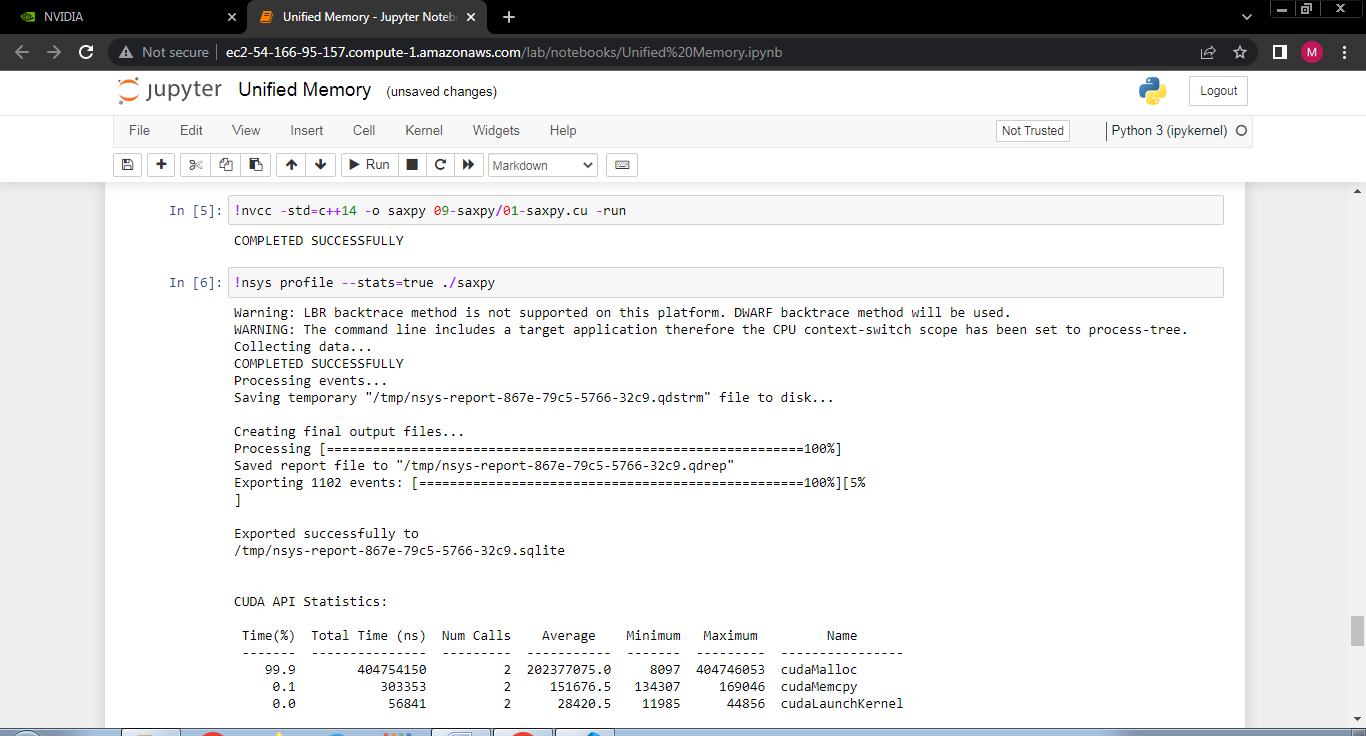
// Print the result

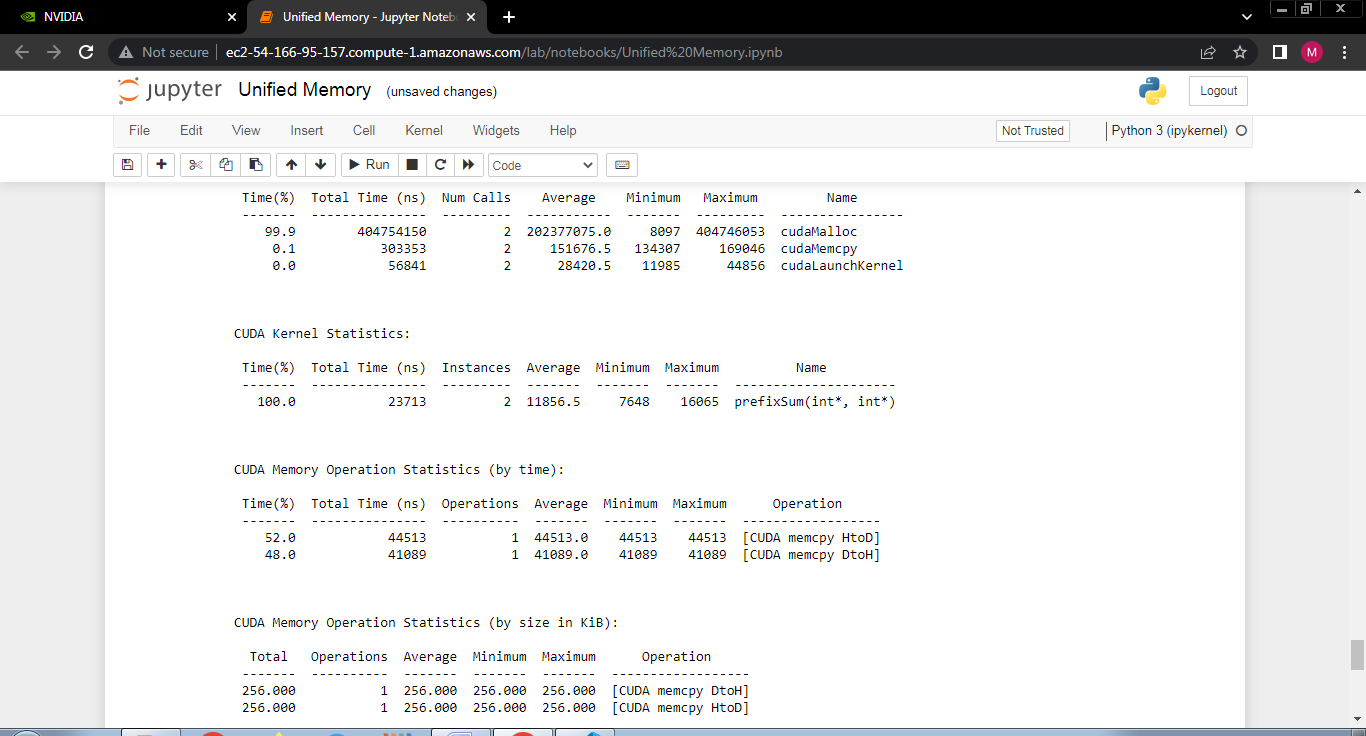
assert(h\_v\_r[0] == std::accumulate(begin(h\_v), end(h\_v), 0));

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.

// 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 + l];

}

}

}

}

// 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;

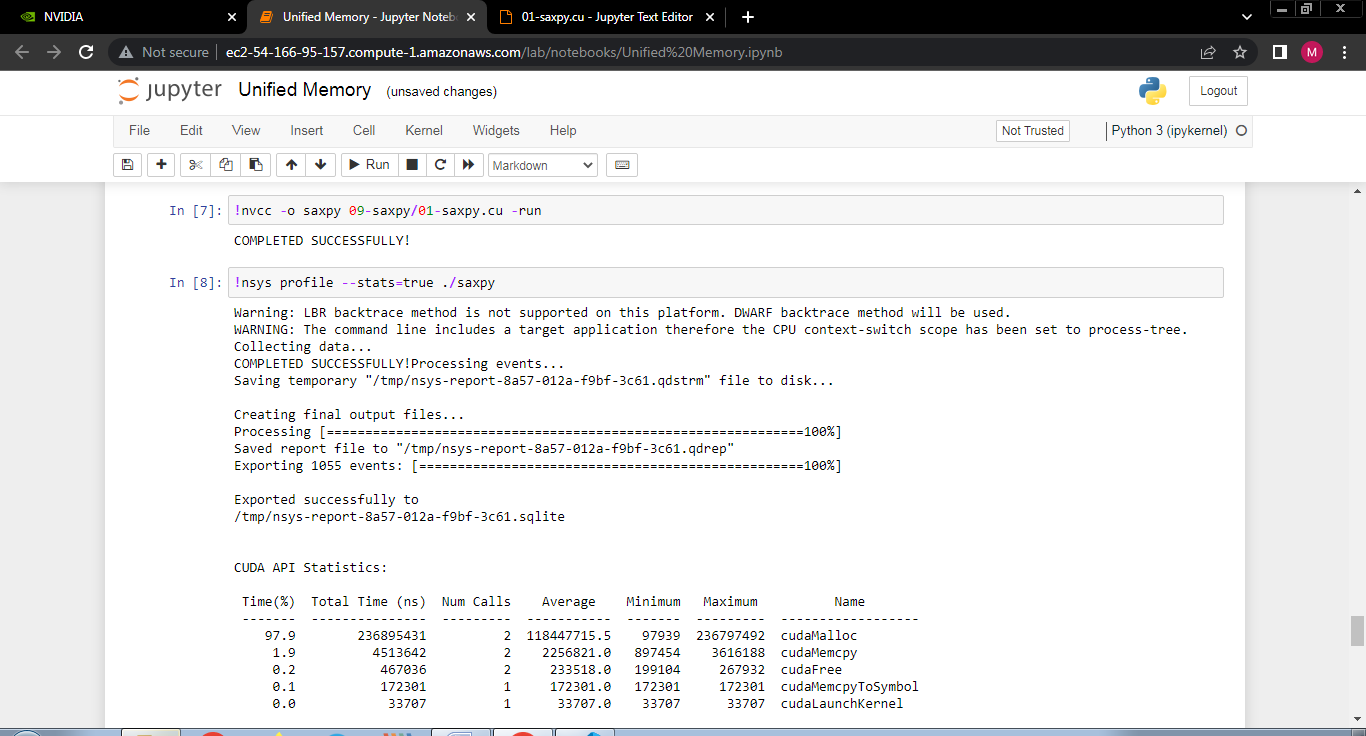
delete[] h\_mask;

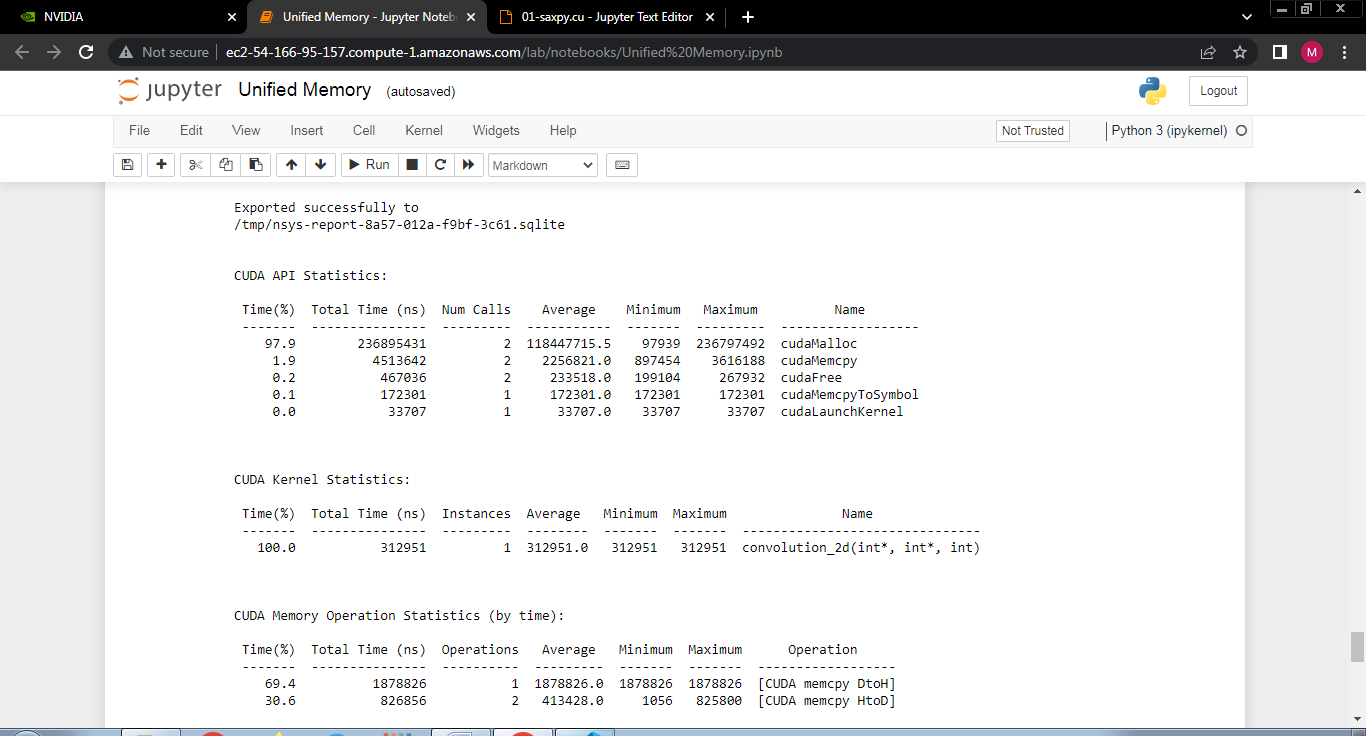
cudaFree(d\_matrix);

cudaFree(d\_result);

return 0;

}





Github Link : <https://github.com/pavanshinde7494/HPC-Assignment>