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:

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:

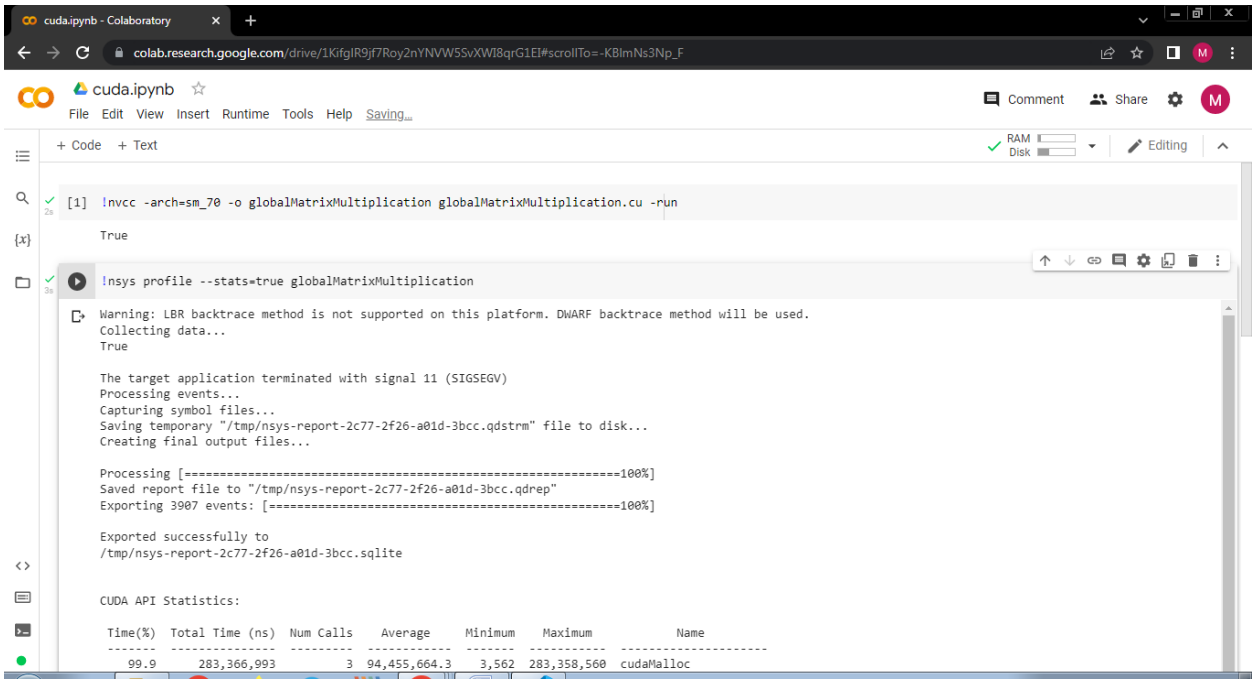
```
#include <stdio.h>

#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++)
```

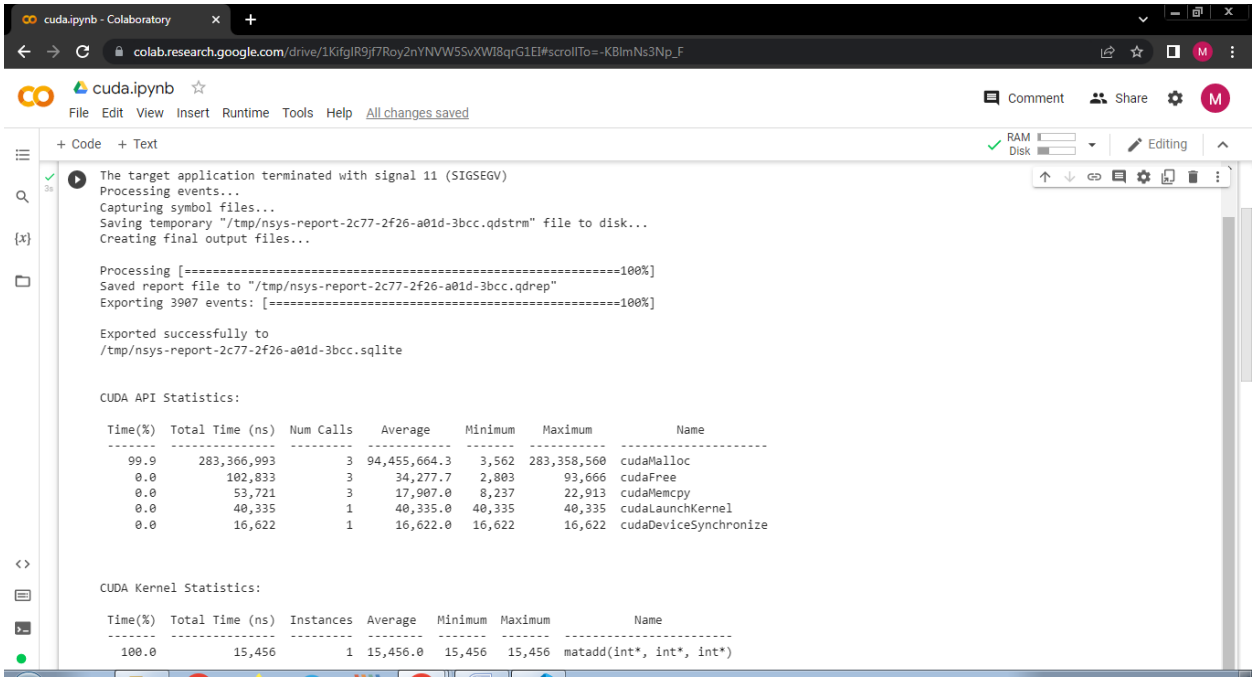
```
        {
            b[i][j] = 3;
        }
    }

    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:



Screenshot 3:



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:

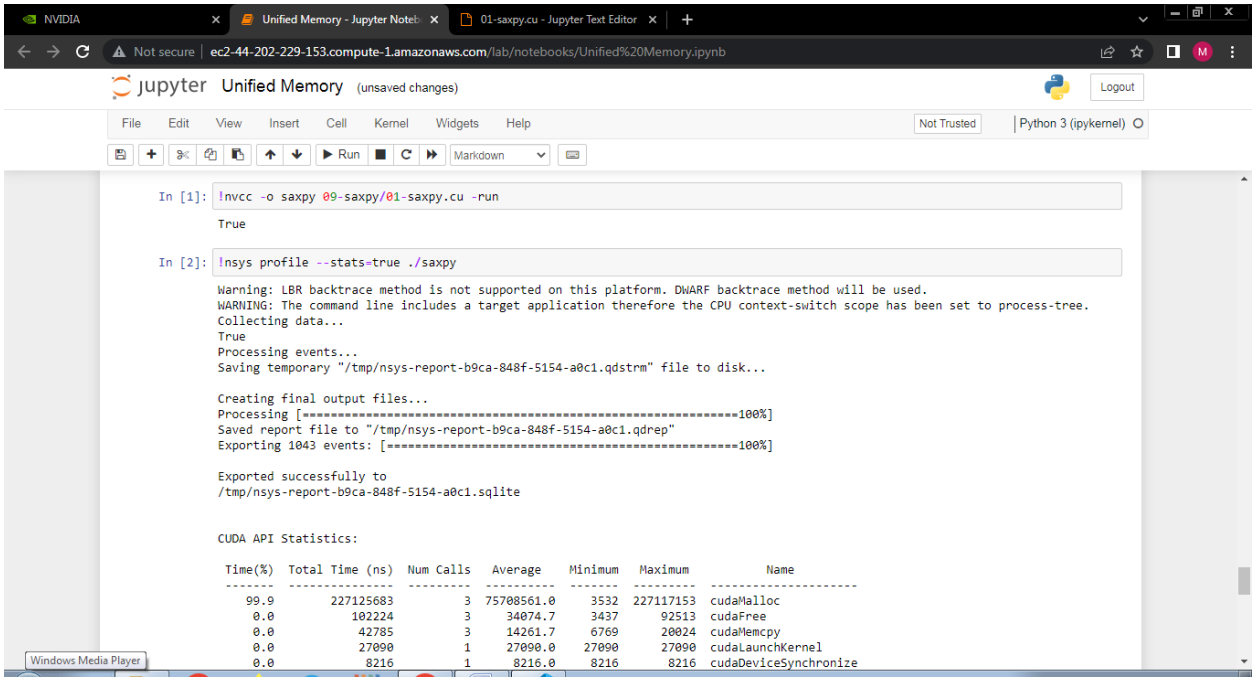
```
#include <stdio.h>

#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++)
        {
```

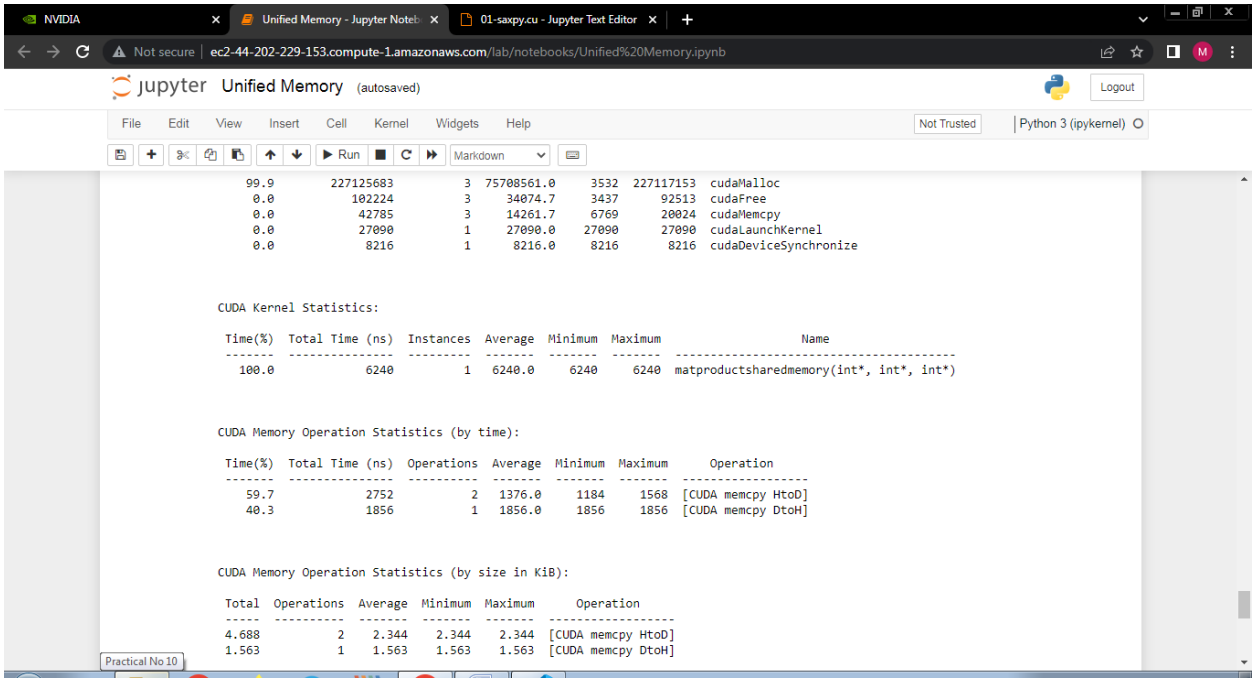
```
            a[i][j] = 2;
        }
    }

    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:



Screenshot 6:



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:

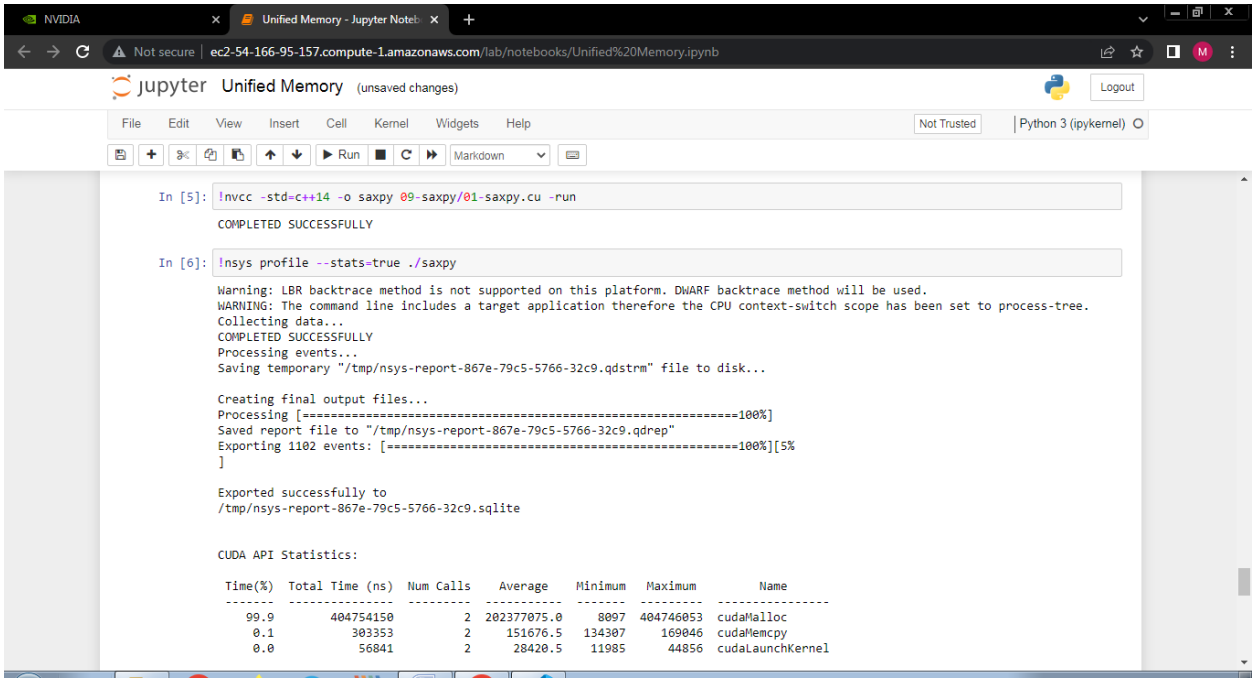
```
// This program computes prefix sum with warp divergence
#include <bits/stdc++.h>

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
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; });
```

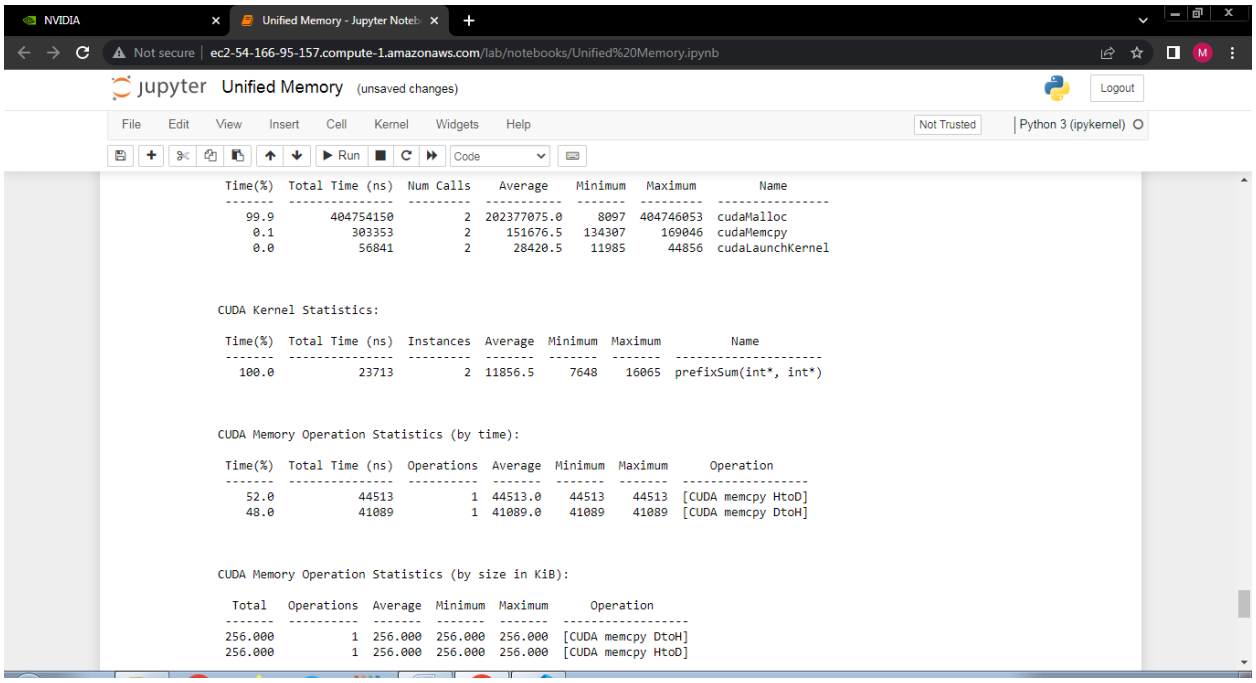
```
    // Allocate device memory
    int *d_v, *d_v_r;
    cudaMalloc(&d_v, bytes);
    cudaMalloc(&d_v_r, bytes);

    // 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;
}
```

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;
```

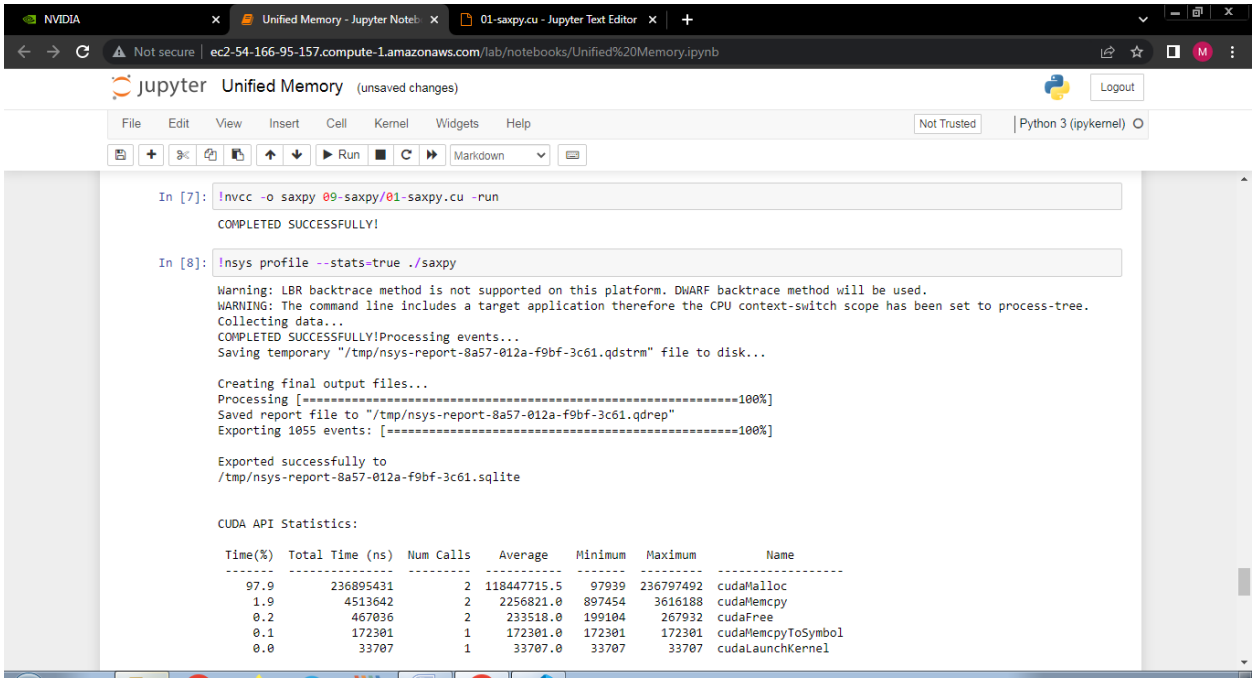
```
  // Go over each row
  for (int i = 0; i < N; i++) {
    // Go over each column
    for (int j = 0; j < N; j++) {
      // Reset the temp variable
      temp = 0;

      // 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 +
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);
```

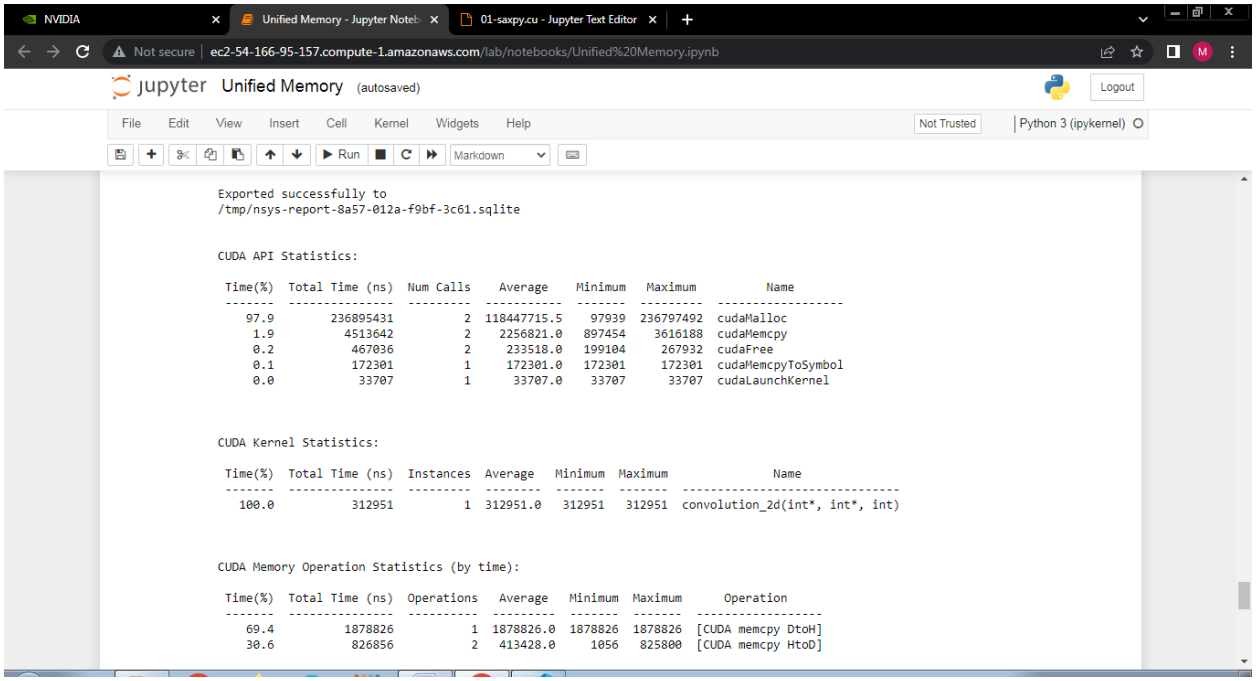
```
// Allocate device memory
int *d_matrix;
int *d_result;
cudaMalloc(&d_matrix, bytes_n);
cudaMalloc(&d_result, bytes_n);

// 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;
}
```

Screenshot 11:



Screenshot 12:



Github Link: