**Class:** Final Year (Computer Science and Engineering)

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 9**

**Name:** Sumit Narake

**PRN:**2020BTECS00023

**Batch :** B2

**Title of practical:** Implementation of Matrix-matrix Multiplication (global and shared Memory), 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.

**Code:**

#include <stdio.h>

void initWith(float num, float \*a, int SIZE)

{

    for (int i = 0; i < SIZE; ++i)

    {

        a[i] = num;

    }

}

\_\_global\_\_ void matrixMultiply(float \*result, float \*a, float \*b, int N, int SIZE)

{

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

    int stride = gridDim.x \* blockDim.x;

    for (int i = start; i < SIZE; i += stride)

    {

        int row = i / N;

        float sum = 0;

        for (int j = 0; j < N; j++)

        {

            sum += a[row \* N + j] \* b[N \* j + row];

        }

        result[i] = sum;

    }

}

void checkElementsAre(float target, float \*array, int SIZE)

{

    for (int i = 0; i < SIZE; i++)

    {

        if (array[i] != target)

        {

            printf("FAIL: array[%d] - %0.0f does not equal %0.0f\n", i,

                   array[i], target);

            exit(1);

        }

    }

    printf("SUCCESS! All values multiplied correctly.\n");

}

int main()

{

    const int N = 1024;

    const int SIZE = N \* N; // sqaure matrix

    size\_t size = SIZE \* sizeof(float);

    float \*a;

    float \*b;

    float \*c;

    cudaMallocManaged(&a, size);

    cudaMallocManaged(&b, size);

    cudaMallocManaged(&c, size);

    initWith(3, a, SIZE);

    initWith(4, b, SIZE);

    initWith(0, c, SIZE);

    matrixMultiply<<<100, 1024>>>(c, a, b, N, SIZE);

    cudaDeviceSynchronize();

    checkElementsAre(12288, c, SIZE);

    cudaFree(a);

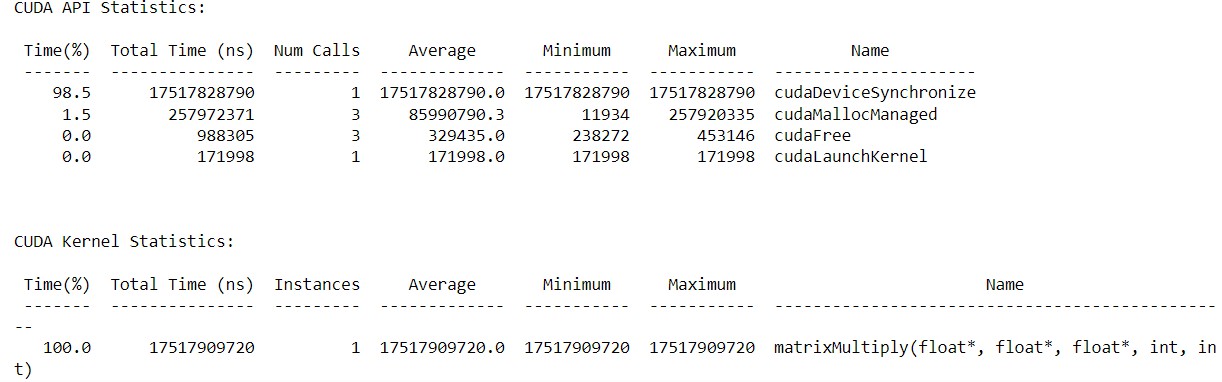
    cudaFree(b);

    cudaFree(c);

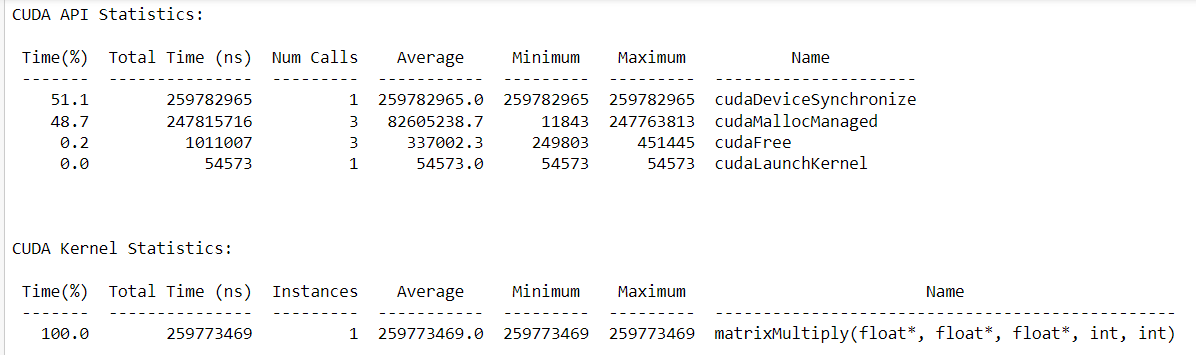
}

**Screenshots:**

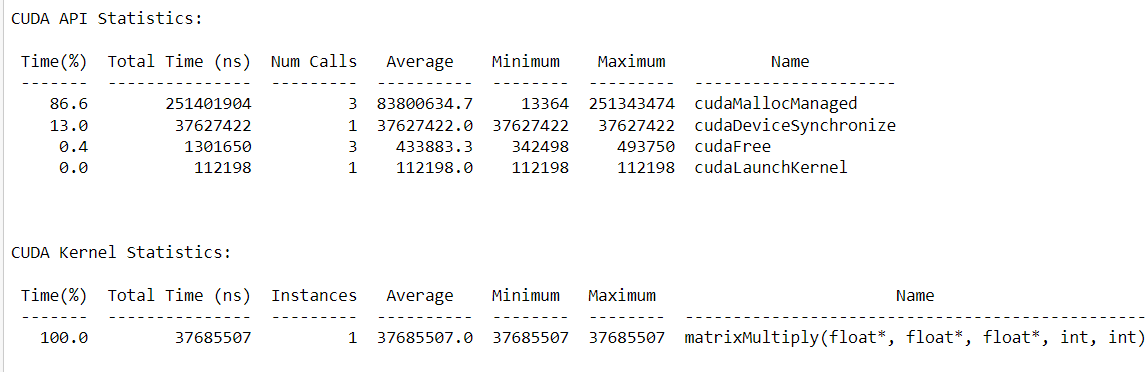
**For <<<1,1>>>**



**For <<<2,64>>>**



**For <<<8,1024>>>**



**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Block Size** | **Execution time** |
| 1 | 1 | 1.75 |
| 2 | 64 | 0.28 |
| 8 | 1024 | 0.25 |

For performing the matrix-matrix multiplication, we have used the block of n2threads as we must calculate n2elements in the resultant matrix. For calculating each element of the result matrix, row of the first matrix and one column of the second matrix is needed. So, we have assigned the unique row and column to each thread. We can get the unique row and column by using the block indexes and thread indexes as they are unique as a combination.

**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.

**Code:**

#include <stdio.h>

void initWith(float num, float \*a, int SIZE)

{

    for (int i = 0; i < SIZE; ++i)

    {

        a[i] = num;

    }

}

\_\_global\_\_ void matrixMultiply(float \*result, float \*a, float \*b, int N, int SIZE)

{

    \_\_shared\_\_ int stride;

    if (threadIdx.x == 0)

        stride = gridDim.x \* blockDim.x;

    \_\_syncthreads();

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

    for (int i = start; i < SIZE; i += stride)

    {

        int row = i / N;

        float sum = 0;

        for (int j = 0; j < N; j++)

        {

            sum += a[row \* N + j] \* b[N \* j + row];

        }

        result[i] = sum;

    }

}

void checkElementsAre(float target, float \*array, int SIZE)

{

    for (int i = 0; i < SIZE; i++)

    {

        if (array[i] != target)

        {

            printf("FAIL: array[%d] - %0.0f does not equal %0.0f\n", i,

                   array[i], target);

            exit(1);

        }

    }

    printf("SUCCESS! All values multiplied correctly.\n");

}

int main()

{

    const int N = 1024;

    const int SIZE = N \* N; // sqaure matrix

    size\_t size = SIZE \* sizeof(float);

    float \*a;

    float \*b;

    float \*c;

    cudaMallocManaged(&a, size);

    cudaMallocManaged(&b, size);

    cudaMallocManaged(&c, size);

    initWith(3, a, SIZE);

    initWith(4, b, SIZE);

    initWith(0, c, SIZE);

    matrixMultiply<<<100, 1024>>>(c, a, b, N, SIZE);

    cudaDeviceSynchronize();

    checkElementsAre(12288, c, SIZE);

    cudaFree(a);

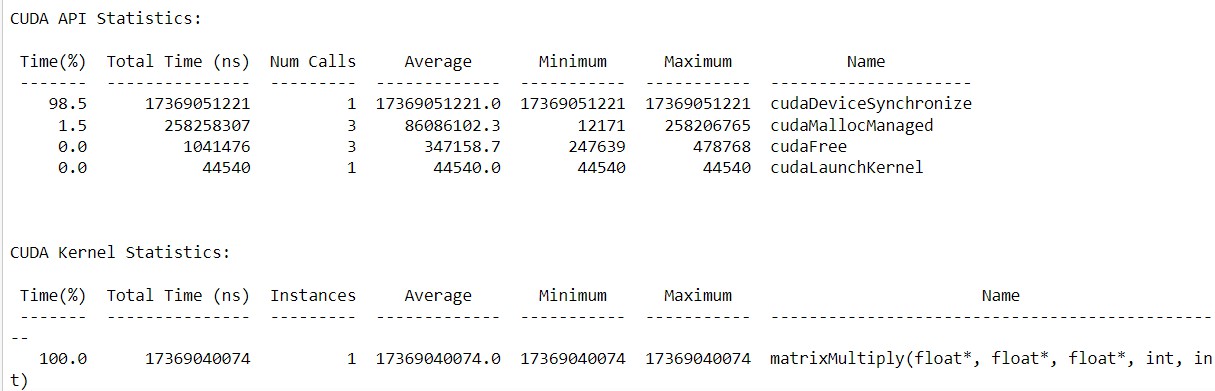
    cudaFree(b);

    cudaFree(c);

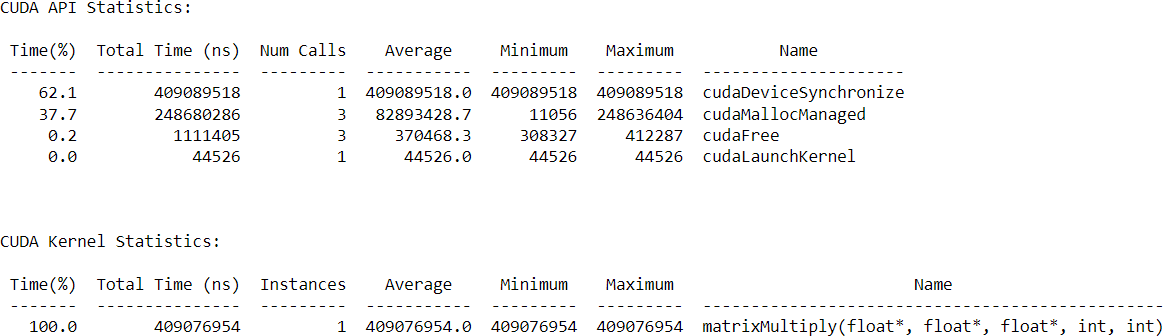
}

**Screenshots:**

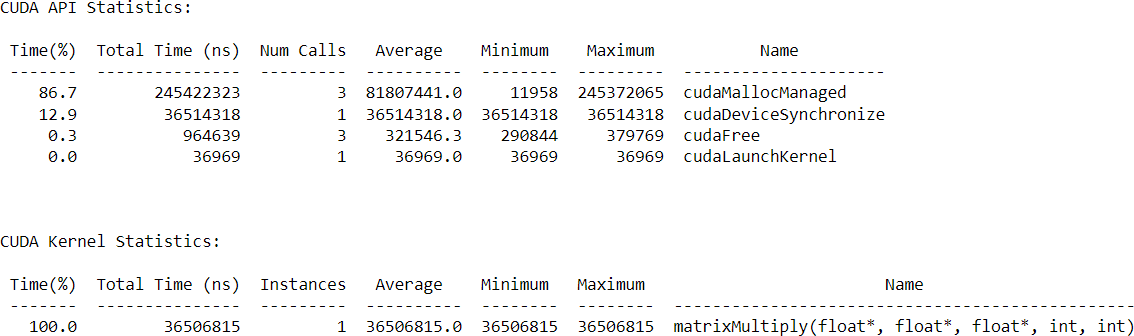
**For <<<1,1>>>**



**For<<<2,32>>>**



**For <<<8,1024>>>**



**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Block Size** | **Execution time** |
| 1 | 1 | 1.73 |
| 2 | 64 | 0.40 |
| 8 | 1024 | 0.24 |

For performing the matrix-matrix multiplication, we have used the n2 blocks of nthreads each as we have to calculate n2elements in the resultant matrix. In this computation, for calculating each element of the result matrix, we need n threads.

In this implementation, the matrix multiplication is computed using a grid of thread blocks. Each thread block is responsible for computing a sub-matrix of the resulting matrix. Threads within a block cooperate to load the necessary data into shared memory and perform the matrix multiplication. The use of shared memory minimizes global memory accesses, leading to significant speedups.

**Analysis by comparing shared and global version:**

While using the global memory version of the program, one element of the result matrix is calculated by the one thread. But, in case of the shared memory each element is calculated by one block consisting of n threads each.

We can get the significant speedup in case of the shared memory program as it reduces the global memory accesses and increases the local memory accesses.

**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.

**Code:**

#include <stdio.h>

void initWith(float val, float \*arr, int N)

{

    for (int i = 0; i < N; i++)

    {

        arr[i] = val;

    }

}

\_\_global\_\_ void prefixSum(float \*arr, float \*res, float \*ptemp, float \*ttemp, int N)

{

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

    int totalThreads = gridDim.x \* blockDim.x;

    int elementsPerThread = ceil(1.0 \* N / totalThreads);

    int start = threadId \* elementsPerThread;

    int count = 0;

    float \*sums = new float[elementsPerThread];

    float sum = 0;

    for (int i = start; i < N && count < elementsPerThread; i++, count++)

    {

        sum += arr[i];

        sums[count] = sum;

    }

    float localSum;

    if (count)

        localSum = sums[count - 1];

    else

        localSum = 0;

    ptemp[threadId] = localSum;

    ttemp[threadId] = localSum;

    \_\_syncthreads();

    if (totalThreads == 1)

    {

        for (int i = 0; i < N; i++)

            res[i] = sums[i];

    }

    else

    {

        int d = 0; // log2(totalThreads)

        int x = totalThreads;

        while (x > 1)

        {

            d++;

            x = x >> 1;

        }

        x = 1;

        for (int i = 0; i < 2 \* d; i++)

        {

            int tsum = ttemp[threadId];

            \_\_syncthreads();

            int newId = threadId / x;

            if (newId % 2 == 0)

            {

                int nextId = threadId + x;

                ptemp[nextId] += tsum;

                ttemp[nextId] += tsum;

            }

            else

            {

                int nextId = threadId - x;

                ttemp[nextId] += tsum;

            }

            x = x << 1;

        }

        \_\_syncthreads();

        float diff = ptemp[threadId] - localSum;

        for (int i = start, j = 0; i < N && j < count; i++, j++)

        {

            res[i] = sums[j] + diff;

        }

    }

}

void checkRes(float \*arr, float \*res, int N, float \*ptemp, float \*ttemp)

{

    float sum = 0;

    for (int i = 0; i < N; i++)

    {

        sum += arr[i];

        if (sum != res[i])

        {

            printf("FAIL: res[%d] - %0.0f does not equal %0.0f\n", i, res[i],

                   sum);

            exit(1);

        }

    }

    printf("SUCCESS! All prefix sums added correctly.\n");

}

int main()

{

    const int N = 1000000;

    size\_t size = N \* sizeof(float);

    float \*arr;

    float \*res;

    cudaMallocManaged(&arr, size);

    cudaMallocManaged(&res, size);

    initWith(2, arr, N);

    initWith(0, res, N);

    int blocks = 1;

    int threadsPerBlock = 32;

    int totalThreads = blocks \* threadsPerBlock;

    float \*ptemp;

    float \*ttemp;

    cudaMallocManaged(&ptemp, totalThreads \* sizeof(float));

    cudaMallocManaged(&ttemp, totalThreads \* sizeof(float));

    prefixSum<<<blocks, threadsPerBlock>>>(arr, res, ptemp, ttemp, N);

    cudaDeviceSynchronize();

    checkRes(arr, res, N, ptemp, ttemp);

    cudaFree(arr);

    cudaFree(res);

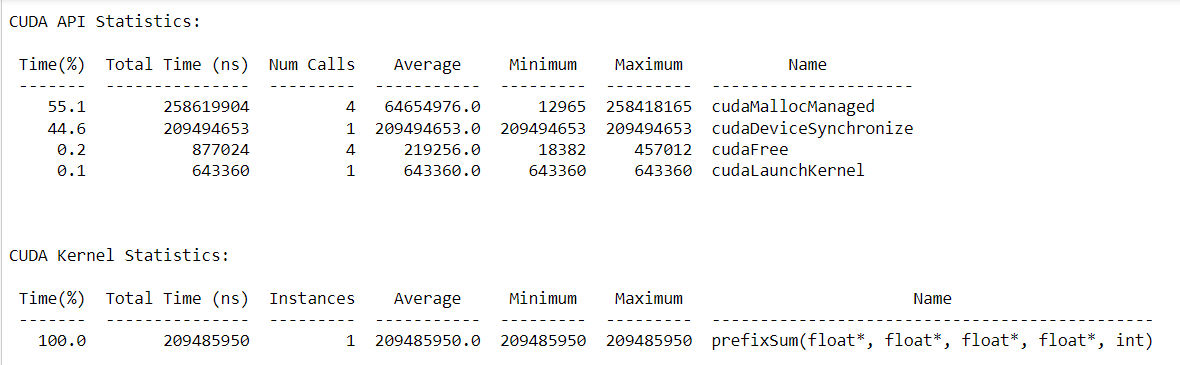
    cudaFree(ttemp);

    cudaFree(ptemp);

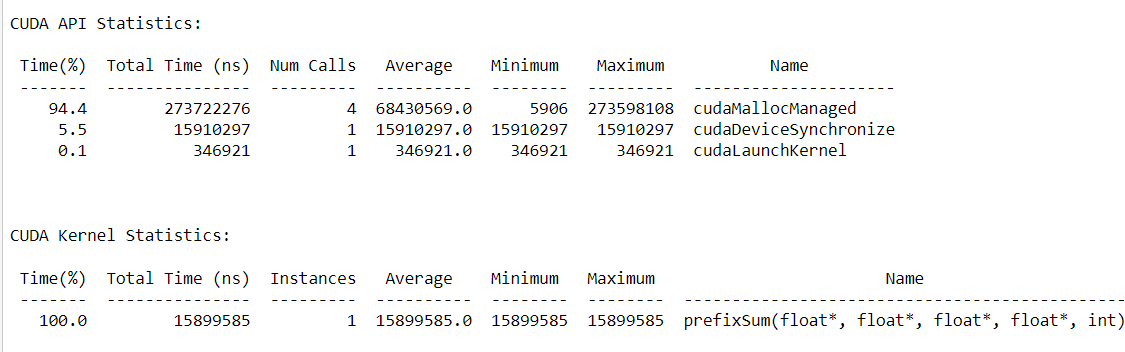
}

**Screenshots:**

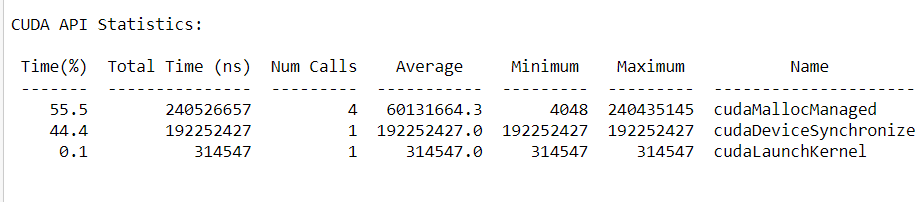
**For <<<1,1>>>**



**For <<<2,32>>>**



**For <<<8,1024>>>**



**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Block Size** | **Execution time** |
| 1 | 1 | 0.25 |
| 2 | 64 | 0.27 |
| 8 | 1024 | 0.24 |

When implementing 2D convolution using shared memory in CUDA, the computation is distributed among threads within a block, with each thread responsible for computing an element of the output matrix. The implementation involves loading the necessary data into shared memory to minimize global memory transactions and maximize memory access efficiency.

**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.

**Code:**

#include <stdio.h>

#define MASK\_DIM 7

#define MASK\_OFFSET (MASK\_DIM / 2)

\_\_constant\_\_ int mask[7 \* 7];

\_\_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;

}

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;

        }

    }

}

void verify\_result(int \*m, int \*mask, int \*result, int N)

{

    int temp;

    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

            if (result[i \* N + j] != temp)

            {

                printf("Check failed");

                return;

            }

        }

    }

}

int main()

{

    int N = 1 << 10; // 2^10

    size\_t bytes\_n = N \* N \* sizeof(int);

    size\_t bytes\_m = MASK\_DIM \* MASK\_DIM \* sizeof(int);

    int \*matrix;

    int \*result;

    int \*h\_mask;

    cudaMallocManaged(&matrix, bytes\_n);

    cudaMallocManaged(&result, bytes\_n);

    cudaMallocManaged(&h\_mask, bytes\_m);

    init\_matrix(matrix, N);

    init\_matrix(mask, MASK\_DIM);

    cudaMemcpyToSymbol(mask, h\_mask, bytes\_m);

    // Calculate grid dimensions

    int THREADS = 32;

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

    // Dimension launch arguments

    dim3 block\_dim(THREADS, THREADS);

    dim3 grid\_dim(BLOCKS, BLOCKS);

    convolution\_2d<<<grid\_dim, block\_dim>>>(matrix, result, N);

    verify\_result(matrix, h\_mask, result, N);

    printf("COMPLETED SUCCESSFULLY!");

    cudaFree(matrix);

    cudaFree(result);

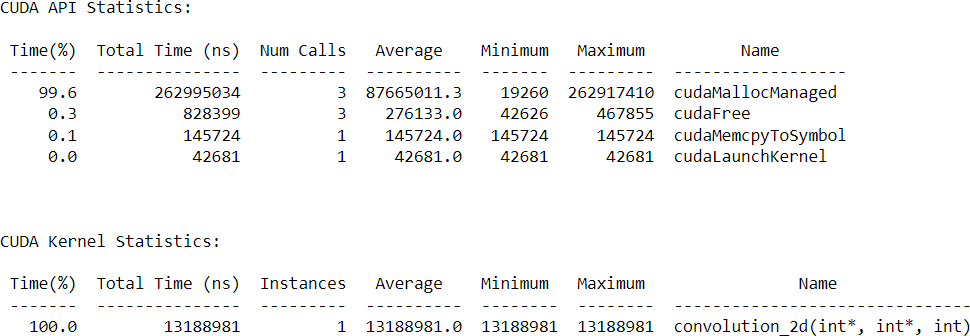
    cudaFree(h\_mask);

    return 0;

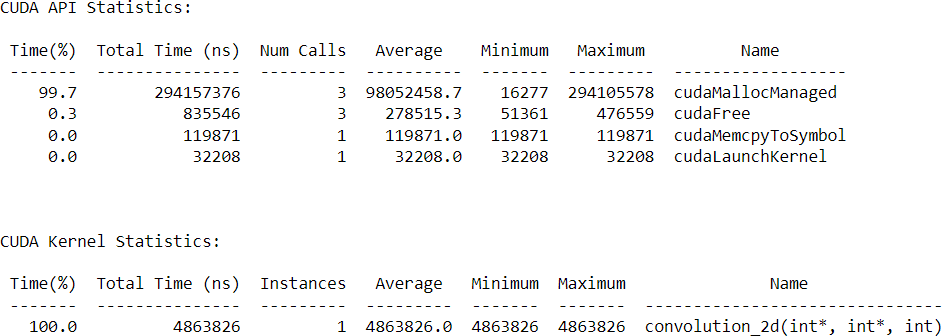
}

**Screenshots:**

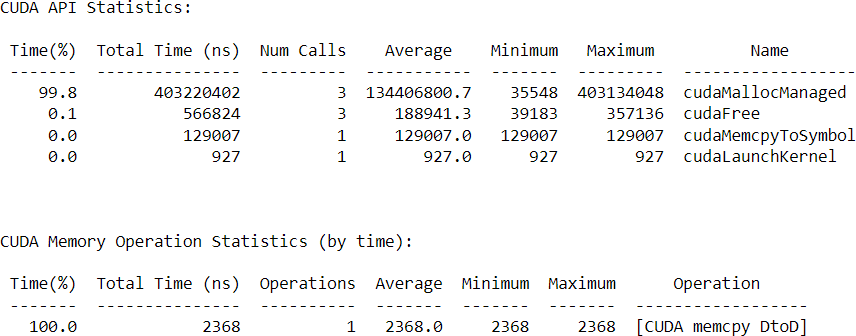
**Thread : 1**



**Thread : 32**



**Thread : 256**



**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Block Size** | **Execution time** |
| 1 | 1 | 1.75 |
| 2 | 64 | 0.28 |
| 8 | 1024 | 0.25 |