**Final Year B. Tech., Sem VII 2022-23**

**High Performance Computing Lab**

**PRN: 2020BTECS00206**

**Full Name: SAYALI YOGESH DESAI**

**Batch: B4**

**Assignment No. 10**

**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>

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<<<1, 1>>>(c, a, b, N, SIZE);

cudaDeviceSynchronize();

checkElementsAre(12288, c, SIZE);

cudaFree(a);

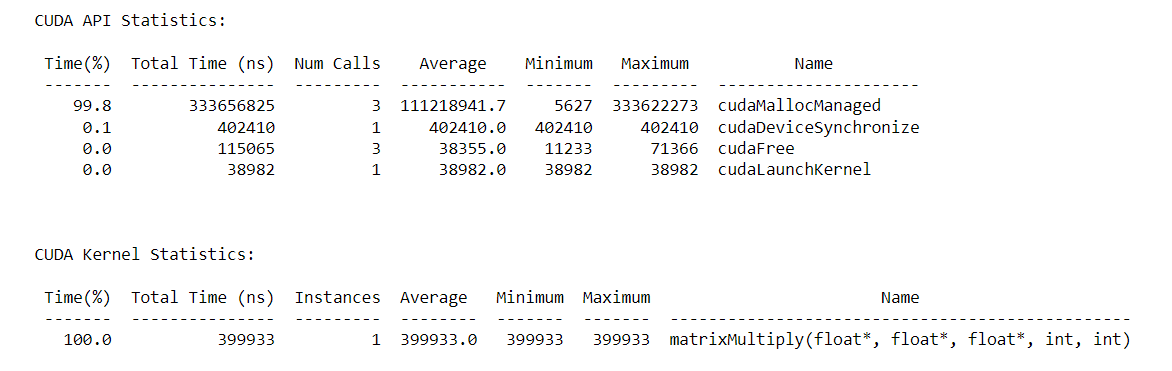
cudaFree(b);

cudaFree(c);

}

**8\*8 Matrix:**

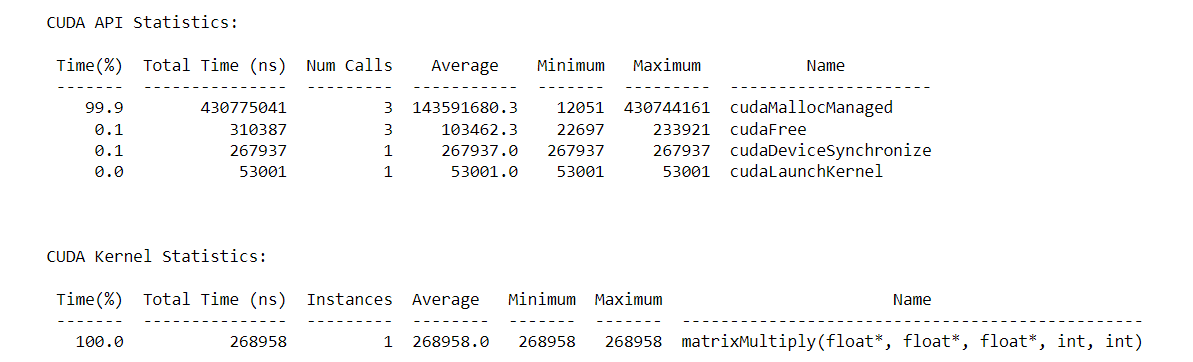
**Serial Execution Time: 399933ns**

****

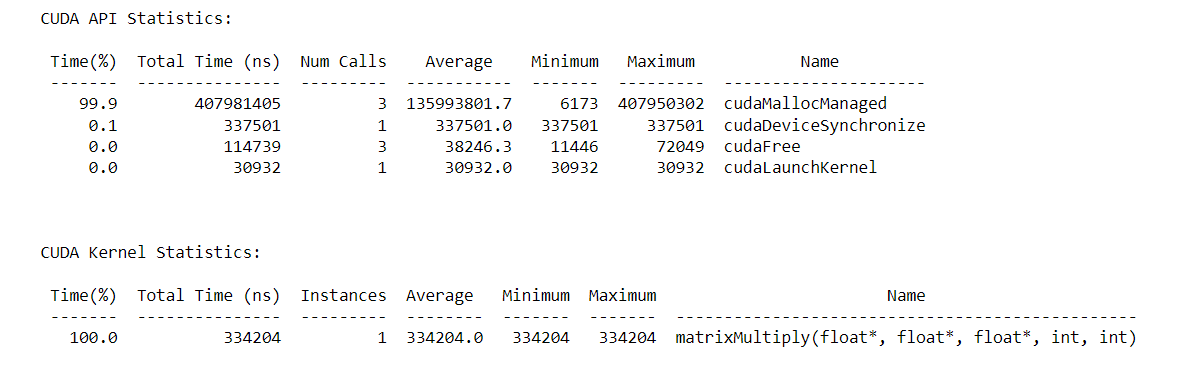
**Parallel Execution Time:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of blocks** | **Thread per blocks** | **Time (in ns)** | **Speedup** |
| 16 | 512 | 268958 | 1.4869 |
| 16 | 1024 | 334204 | 1.1966 |
| 32 | 512 | 374204 | 1.0687 |
| 32 | 1024 | 332125 | 1.2041 |

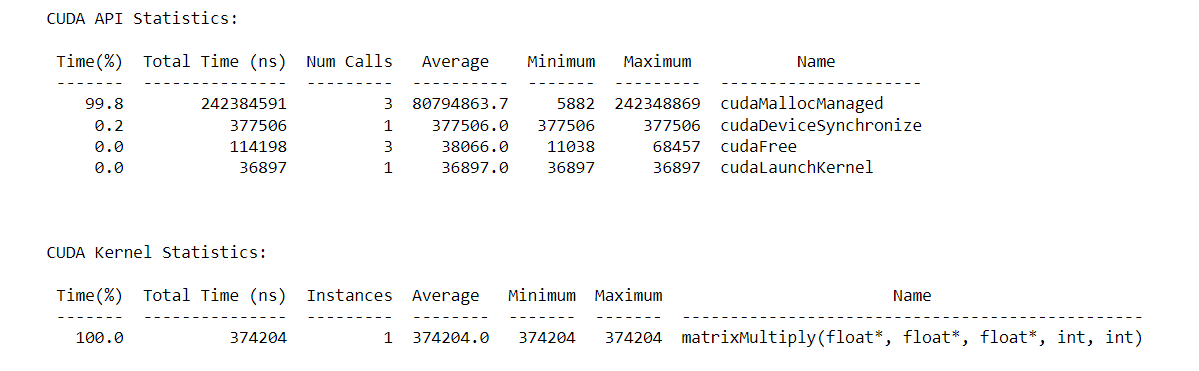
**Number of blocks: 16, Thread per blocks: 512, Execution Time: 268958**

****

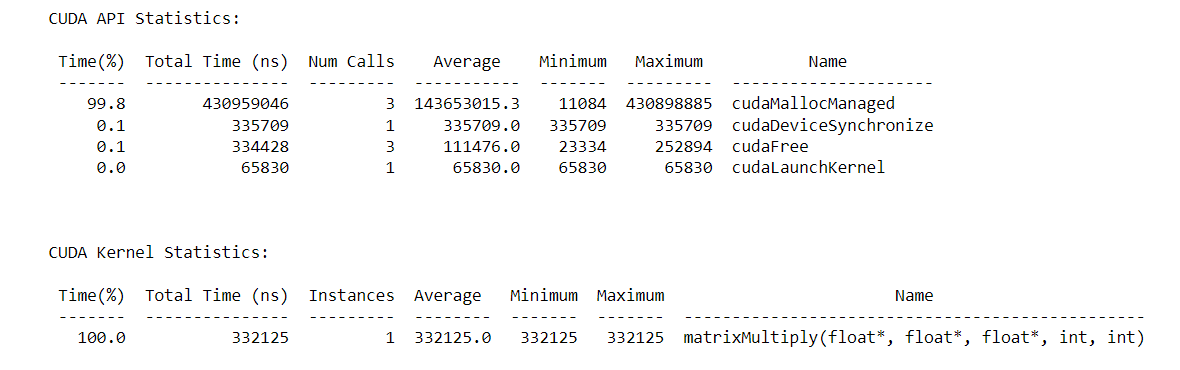
**Number of blocks: 16, Thread per blocks: 1024, Execution Time: 334204**

****

**Number of blocks: 32, Thread per blocks: 512, Execution Time: 374204**

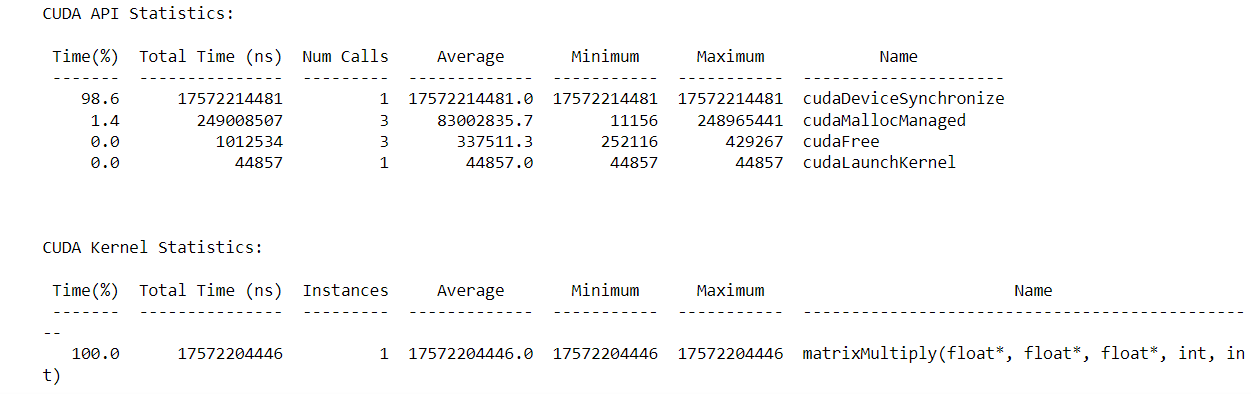
****

**Number of blocks: 32, Thread per blocks: 1024, Execution Time: 332125**

****

**1024\*1024 Matrix:**

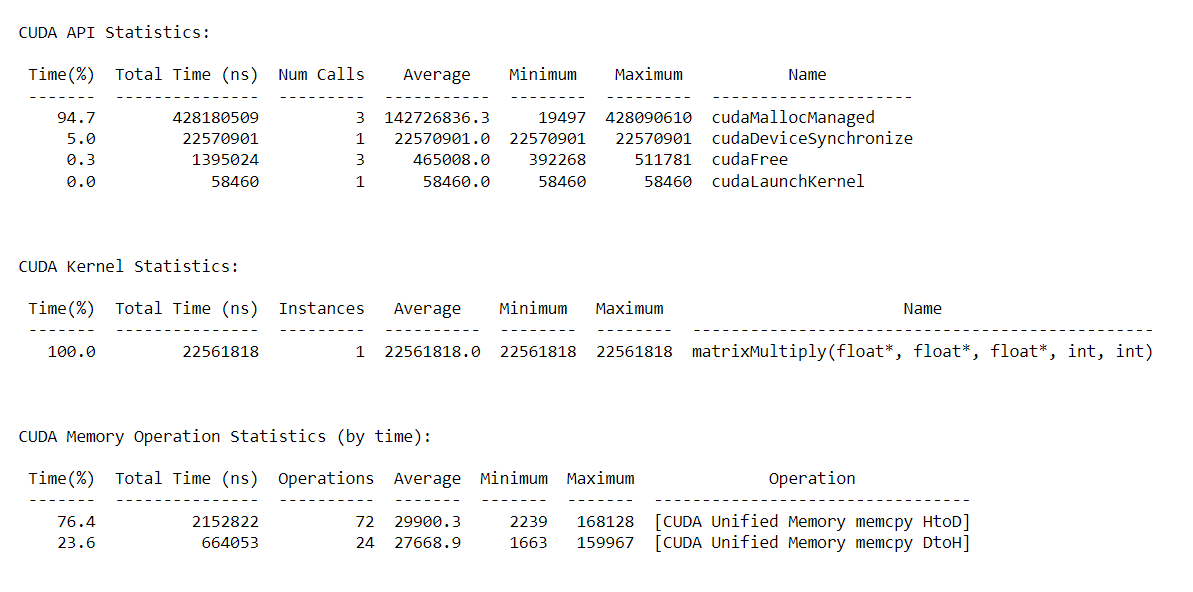
**Serial Execution Time: 17572204446ns**

****

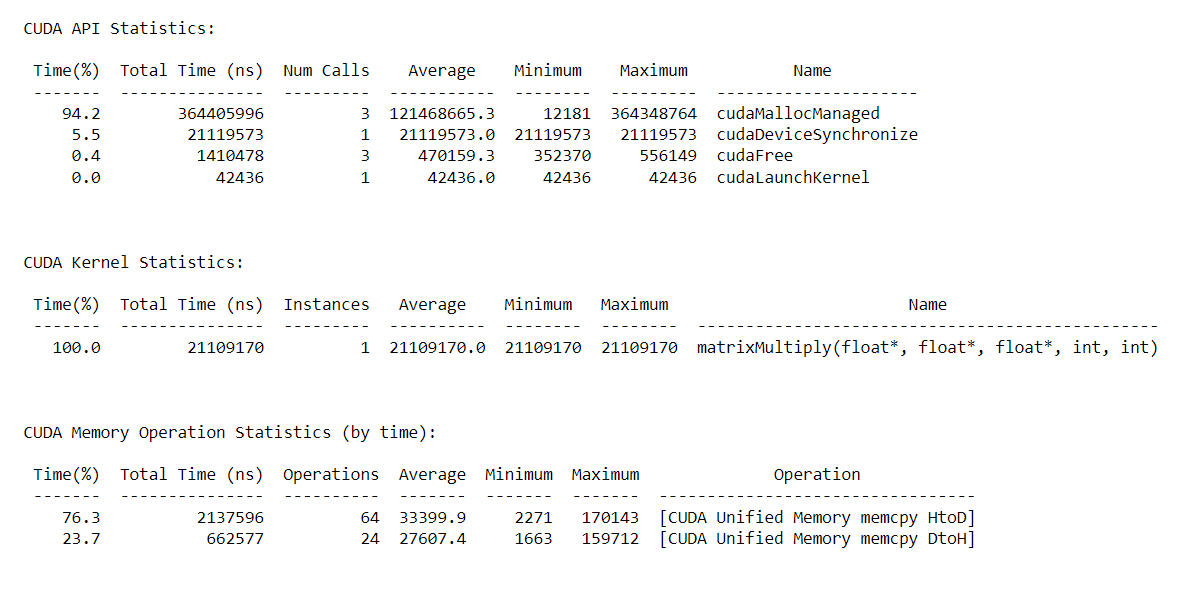
**Parallel Execution Time:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of blocks** | **Thread per blocks** | **Time (in ns)** | **Speedup** |
| 16 | 512 | 22561818 | 778.84 |
| 16 | 1024 | 21109170 | 832.44 |
| 32 | 512 | 15056306 | 1167.09 |
| 32 | 1024 | 14094717 | 1246.72 |

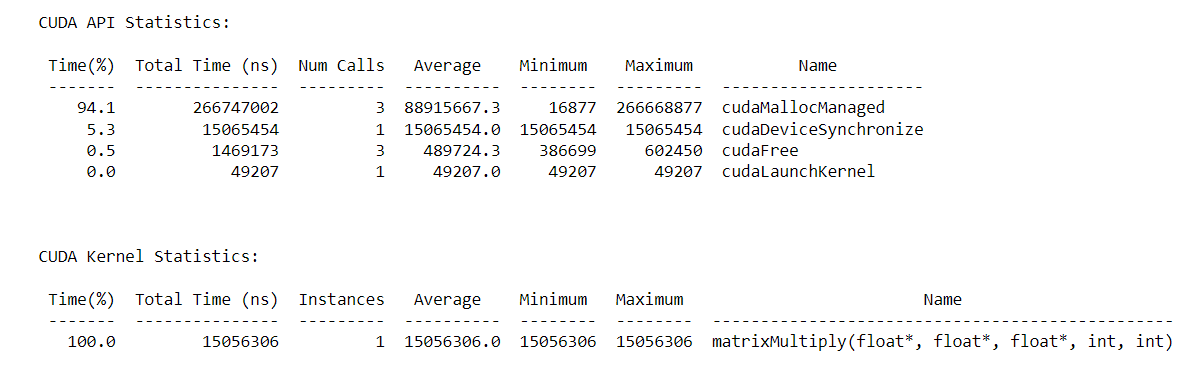
**Number of blocks: 16, Thread per blocks: 512, Execution Time: 22561818ns**



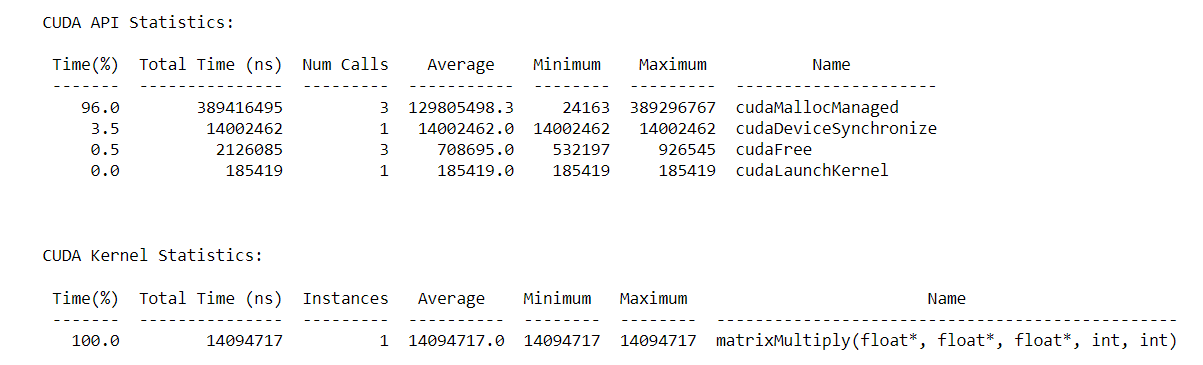
**Number of blocks: 16, Thread per blocks: 1024, Execution Time: 21109170ns**



**Number of blocks: 32, Thread per blocks: 512, Execution Time: 15056306ns**

****

**Number of blocks: 32, Thread per blocks: 1024, Execution Time: 14094717ns**

****

**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>

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<<<1, 1>>>(c, a, b, N, SIZE);

cudaDeviceSynchronize();

checkElementsAre(12288, c, SIZE);

cudaFree(a);

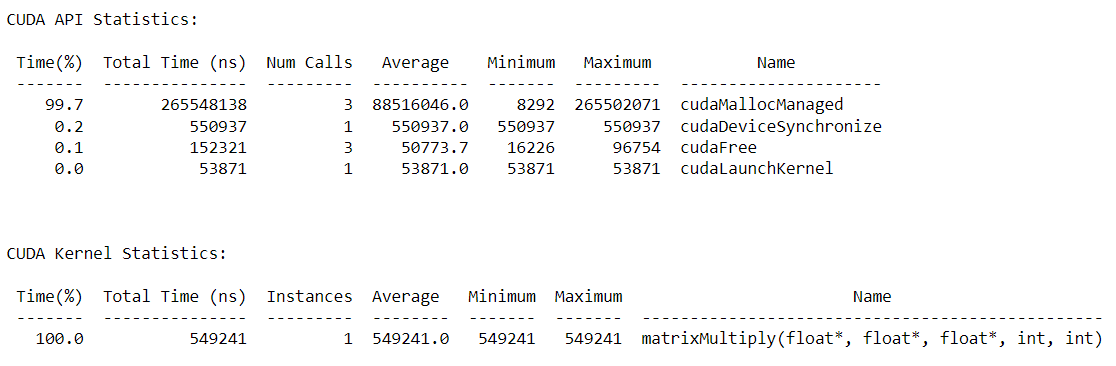
cudaFree(b);

cudaFree(c);

}

**8\*8 Matrix:**

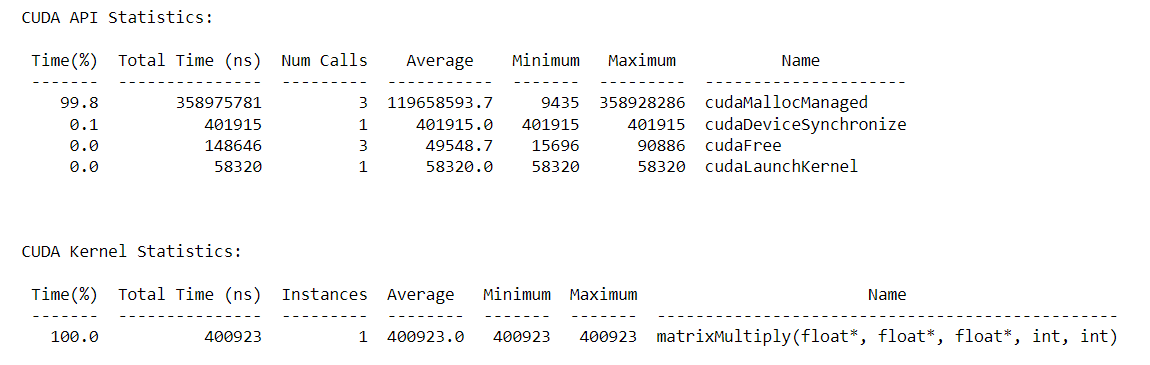
**Serial Execution Time: 549241ns**

****

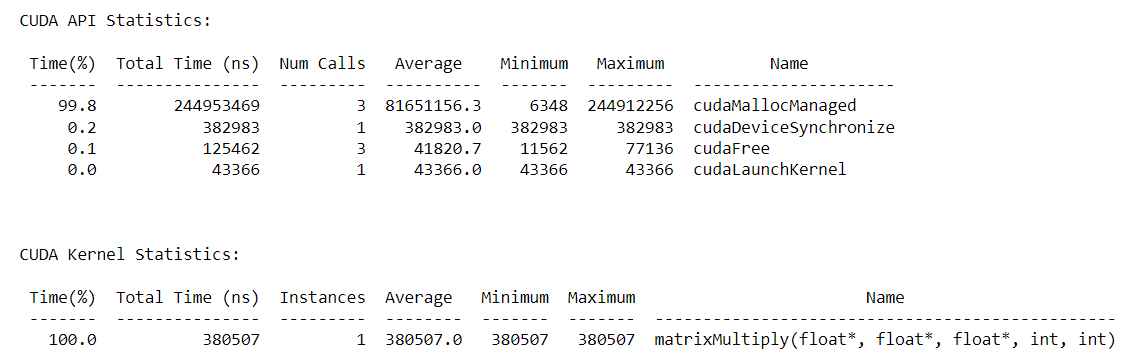
**Parallel Execution Time:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of blocks** | **Thread per blocks** | **Time (in ns)** | **Speedup** |
| 16 | 512 | 400923 | 1.3699 |
| 16 | 1024 | 380507 | 1.4434 |
| 32 | 512 | 481114 | 1.1416 |
| 32 | 1024 | 388795 | 1.4126 |

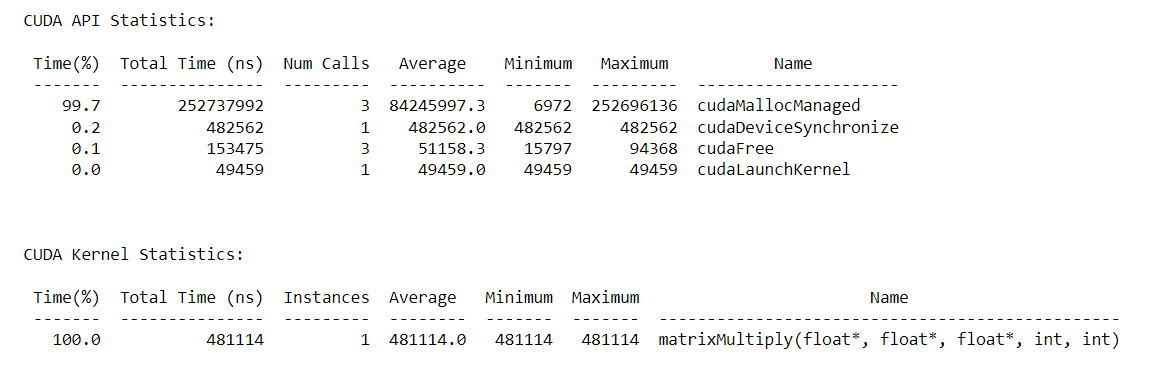
**Number of blocks: 16, Thread per blocks: 512, Execution Time: 400923ns**

****

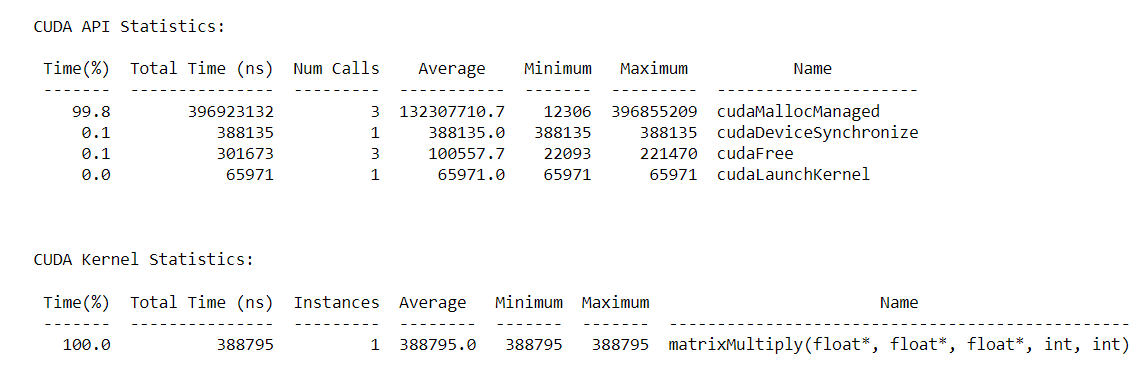
**Number of blocks: 16, Thread per blocks: 1024, Execution Time: 380507ns**

****

**Number of blocks: 32, Thread per blocks: 512, Execution Time: 481114ns**

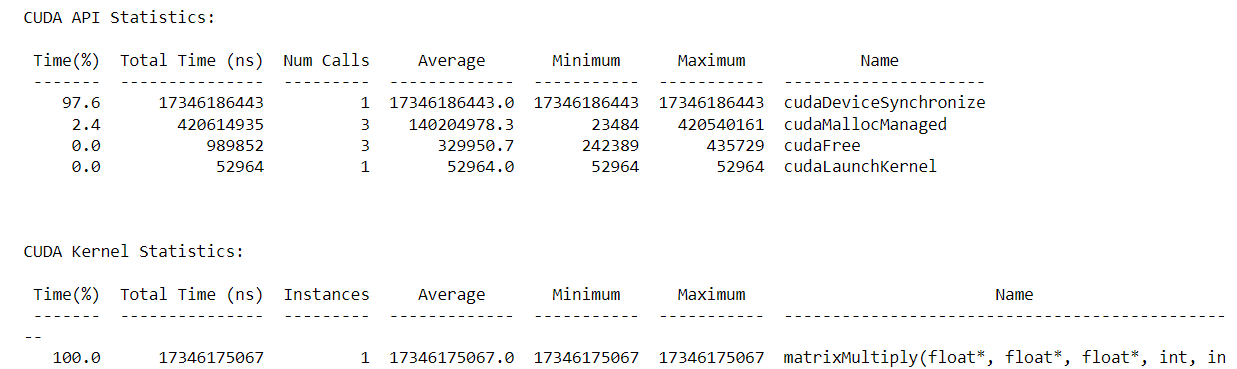
****

**Number of blocks: 32, Thread per blocks: 1024, Execution Time: 388795ns**

****

**1024\*1024 Matrix:**

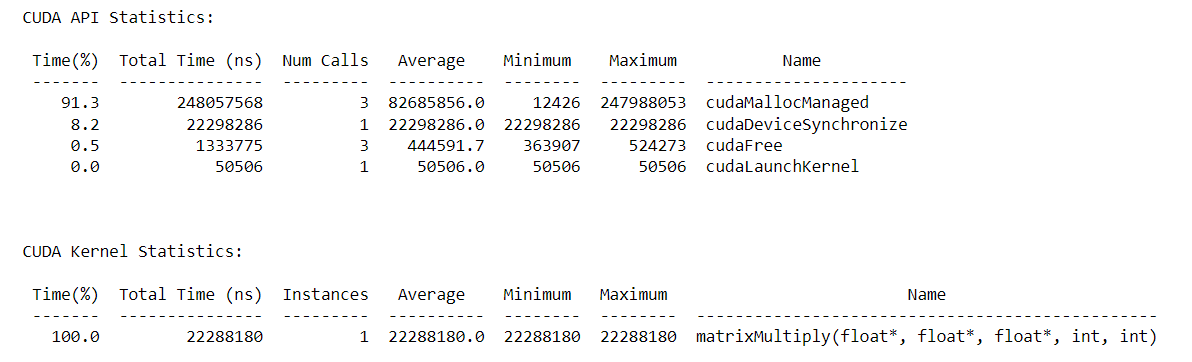
**Serial Execution Time: 17346175067ns**

****

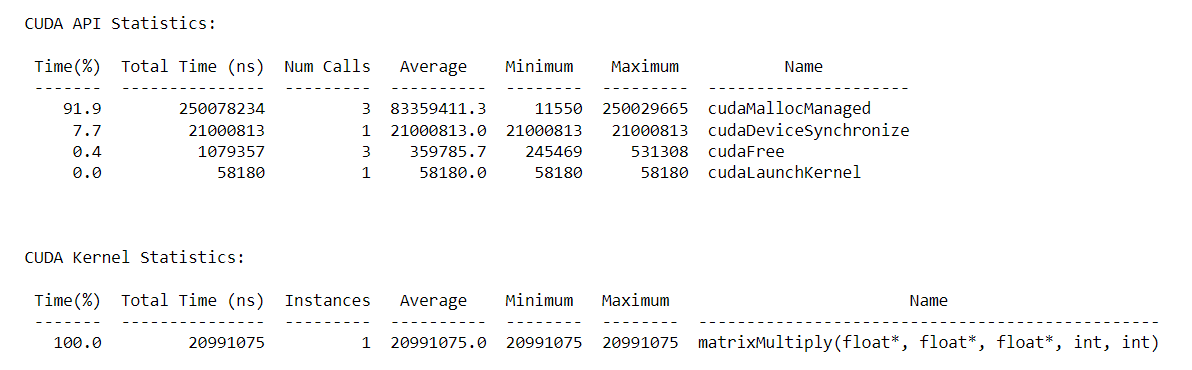
**Parallel Execution Time:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of blocks** | **Thread per blocks** | **Time (in ns)** | **Speedup** |
| 16 | 512 | 22288180 | 778.26 |
| 16 | 1024 | 20991075 | 826.35 |
| 32 | 512 | 15596802 | 1049.72 |
| 32 | 1024 | 14524471 | 1112.16 |

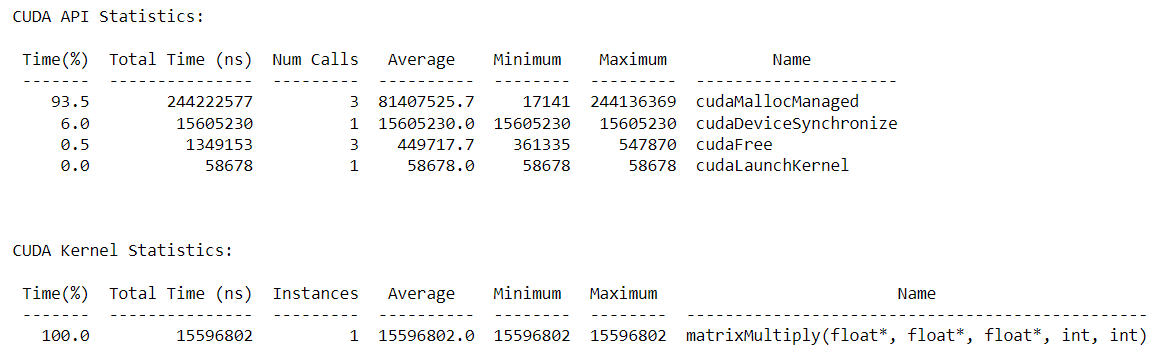
**Number of blocks: 16, Thread per blocks: 512, Execution Time: 22288180ns**

****

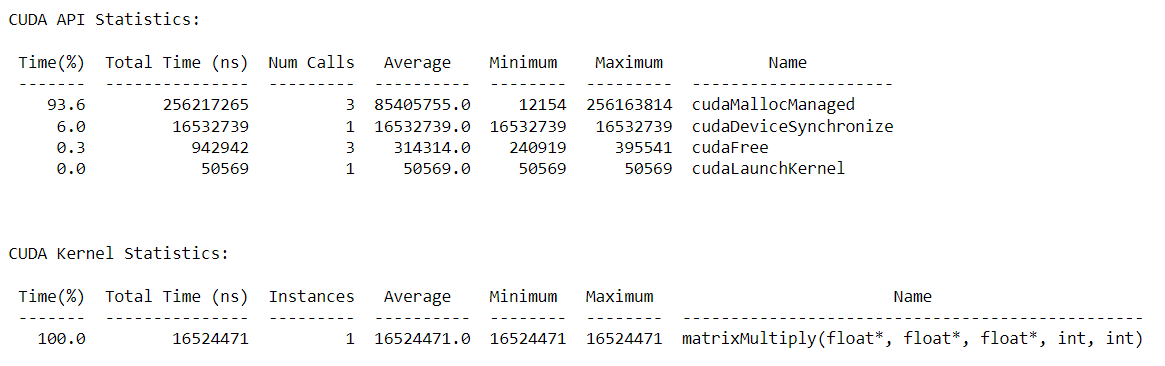
**Number of blocks: 16, Thread per blocks: 1024, Execution Time: 20991075ns**

****

**Number of blocks: 32, Thread per blocks: 512, Execution Time: 15596802ns**

****

**Number of blocks: 32, Thread per blocks: 1024, Execution Time: 16524471ns**

****

**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.**

#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 = 1;

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);

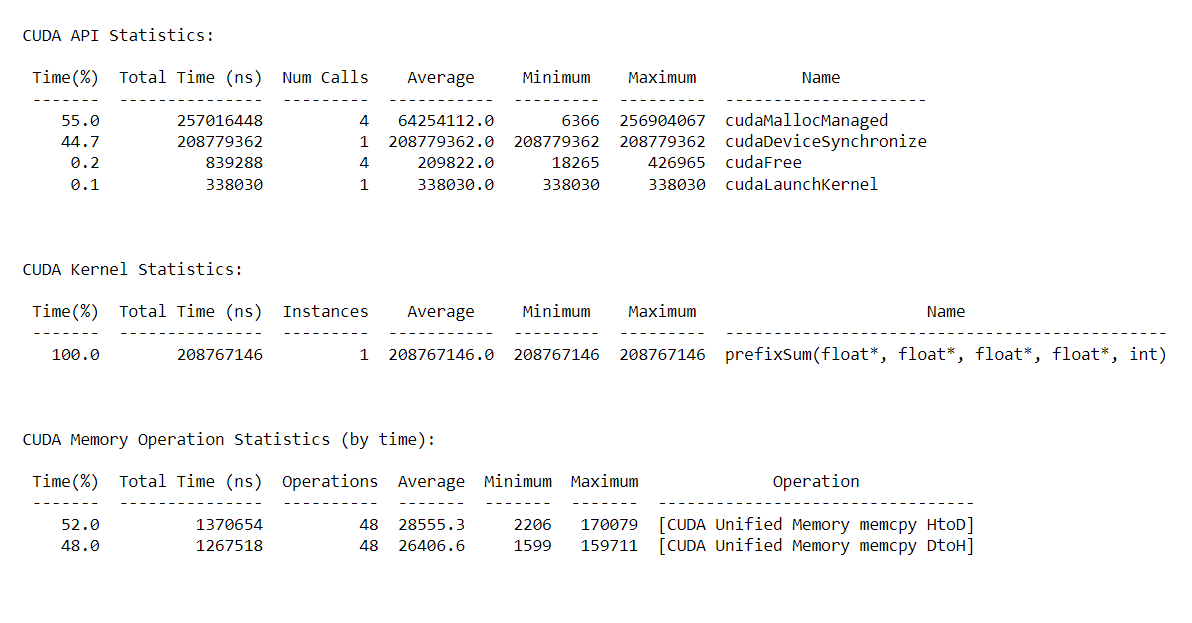
cudaFree(ttemp);

cudaFree(ptemp);

}

**Element 10^6**

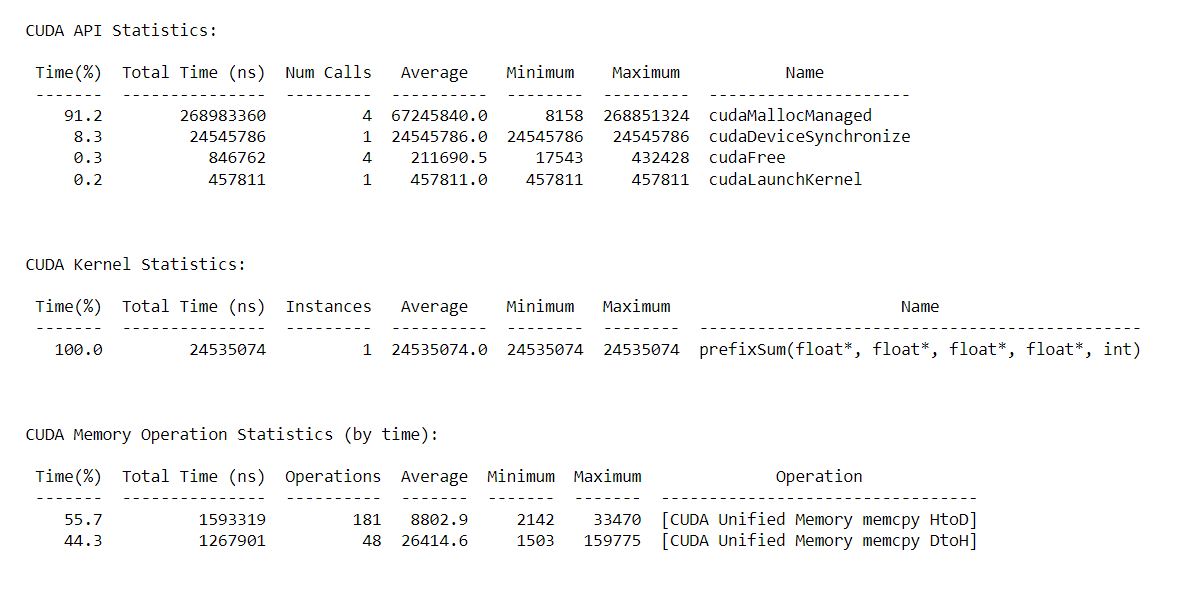
**Serial Execution Time: 208767146ns**



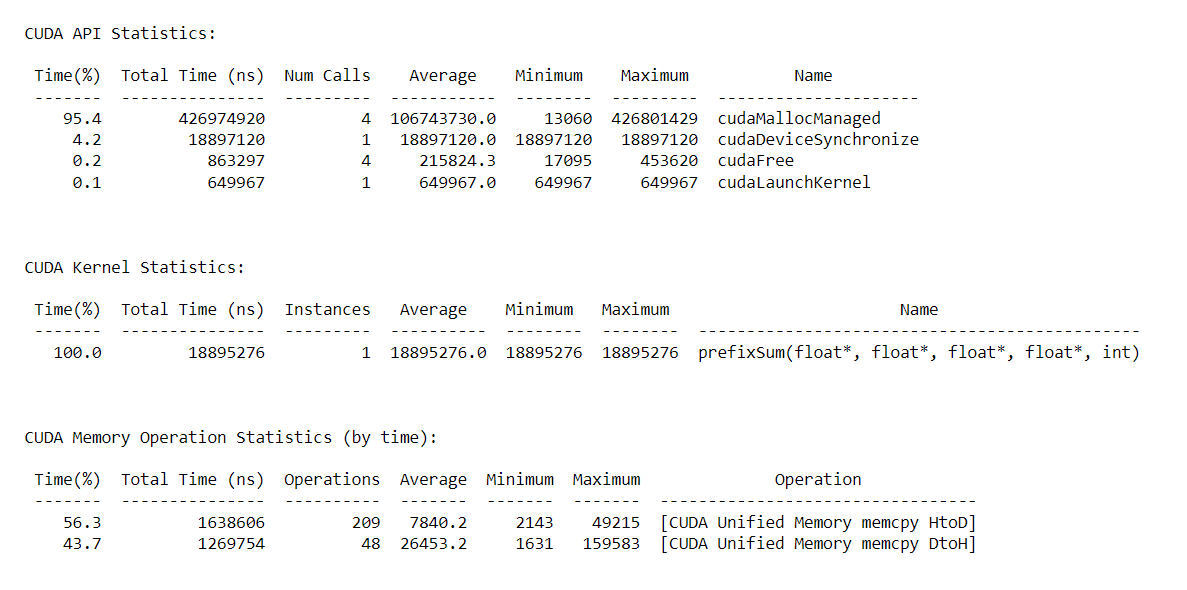
**Parallel Execution Time:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of Blocks** | **Threads per Block** | **Time(ns)** | **Speedup** |
| 1 | 32 | 24535074 | 8.5089 |
| 1 | 64 | 18895276 | 11.0486 |
| 1 | 128 | 17276600 | 12.0838 |

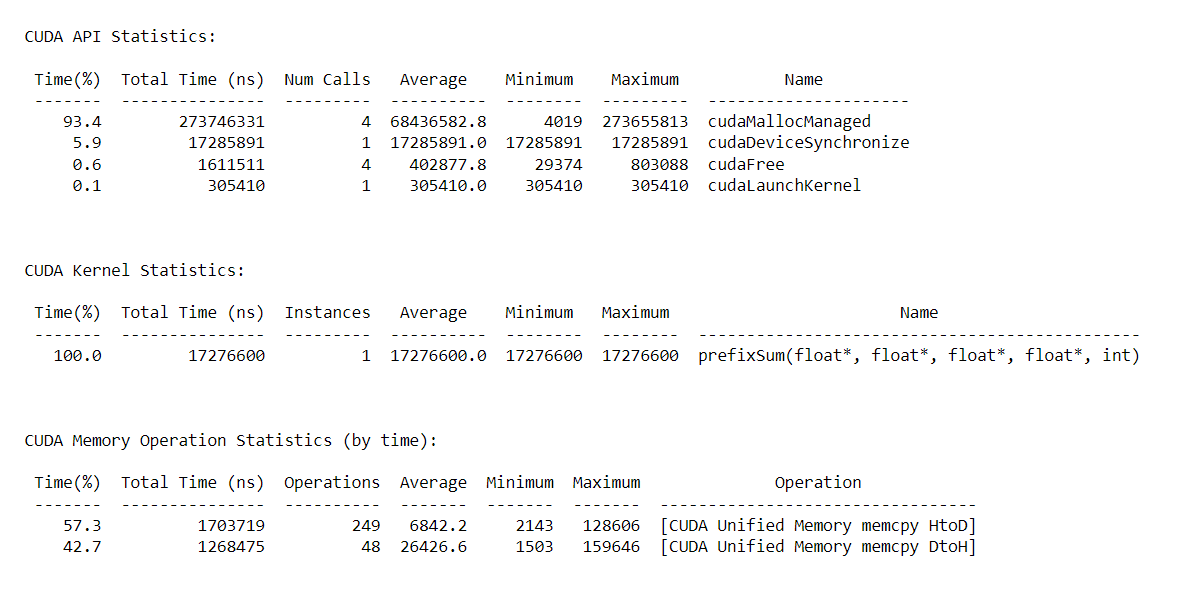
**Number of Blocks: 1, Thread per Blocks: 32, Execution Time: 24535074ns**



**Number of Blocks: 1, Thread per Blocks: 64, Execution Time: 18895276ns**



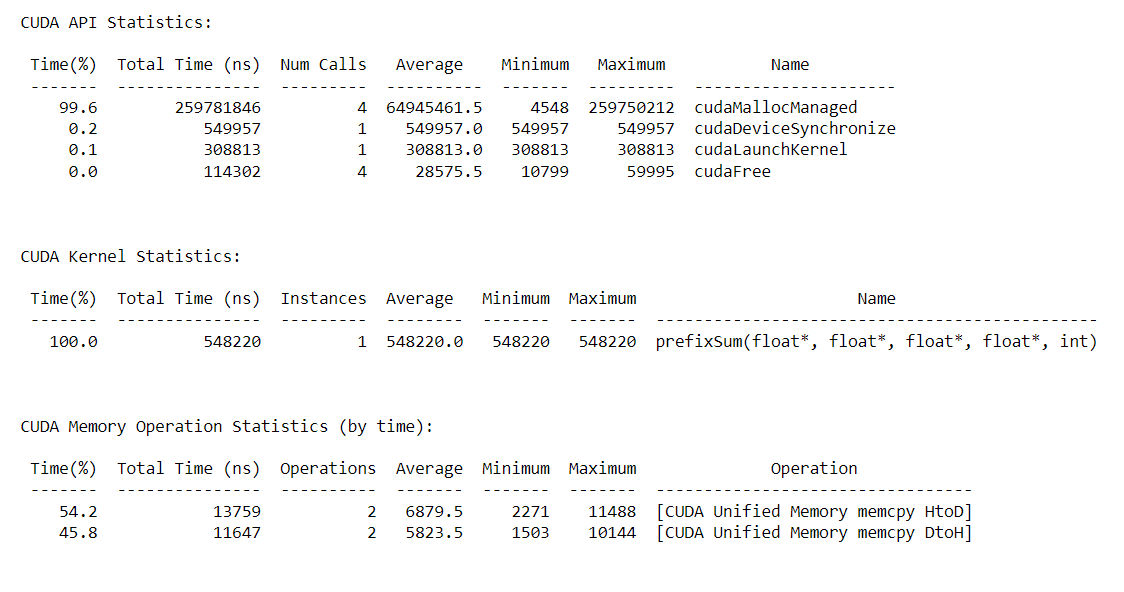
**Number of Blocks: 1, Thread per Blocks: 128, Execution Time: 17276600ns**



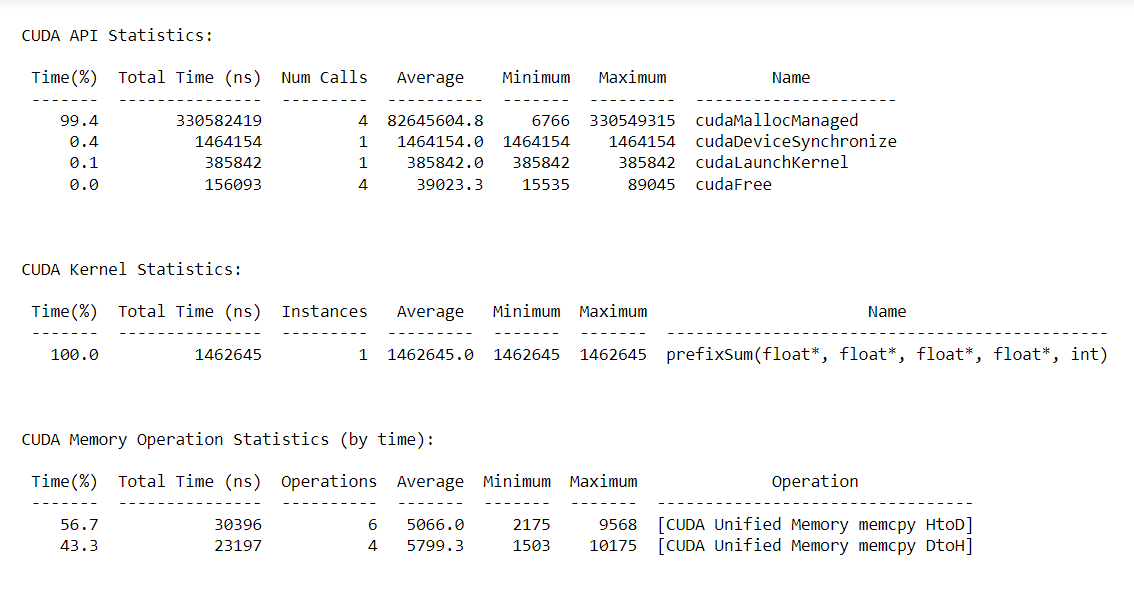
**Parallel Execution Time:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Elements** | **Number of Blocks** | **Threads per Block** | **Time(ns)** |
| 10^3 | 1 | 64 | 548220 |
| 10^4 | 1 | 64 | 1462645 |
| 10^5 | 1 | 64 | 2765066 |
| 10^6 | 1 | 64 | 19337795 |

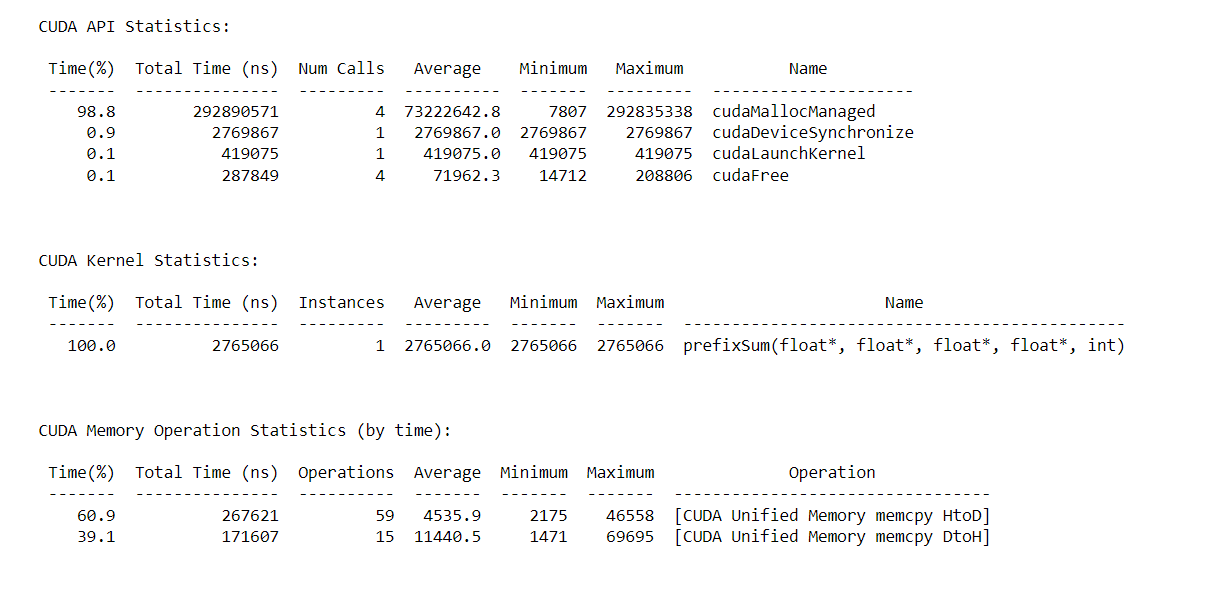
**10^3, Execution time: 548220ns**



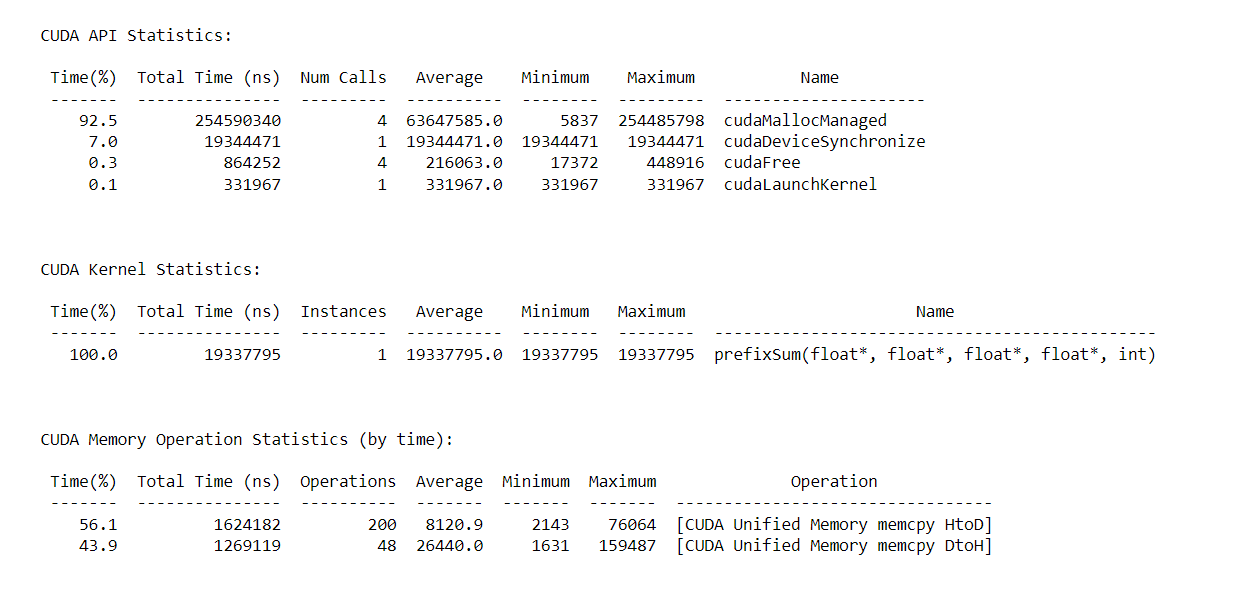
**10^4, Execution time: 1462645ns**



**10^5, Execution time: 2765066ns**



**10^6, Execution time: 19337795ns**



**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.**

#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 = 1;

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;

}

**Execution Time:**

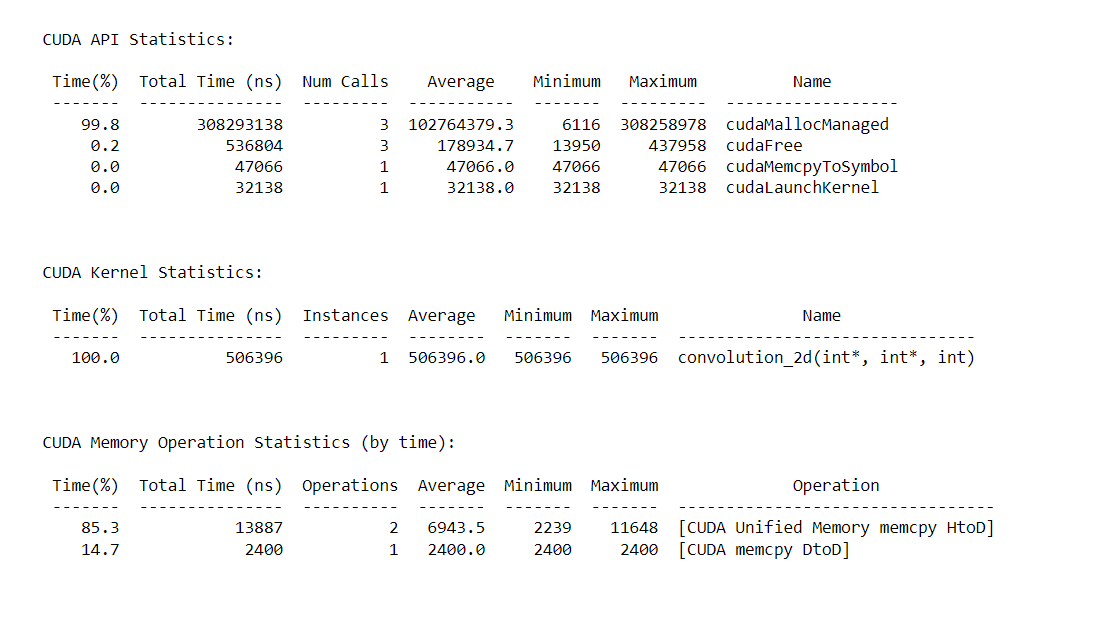
|  |  |  |
| --- | --- | --- |
| **Threads** | **2^4\*2^4** | **Speedup** |
| Serial Execution: | 506396 | - |
| 4 | 638810 | 0.7927 |
| 8 | 382717 | 1.3231 |
| 16 | 531931 | 0.9519 |
| 32 | 517883 | 0.9778 |

|  |  |  |
| --- | --- | --- |
| **Threads** | **2^5\*2^5** | **Speedup** |
| Serial Execution: | 769561 | - |
| 4 | 564605 | 1.3630 |
| 8 | 452604 | 1.7002 |
| 16 | 439085 | 1.7526 |
| 32 | 400955 | 1.9193 |

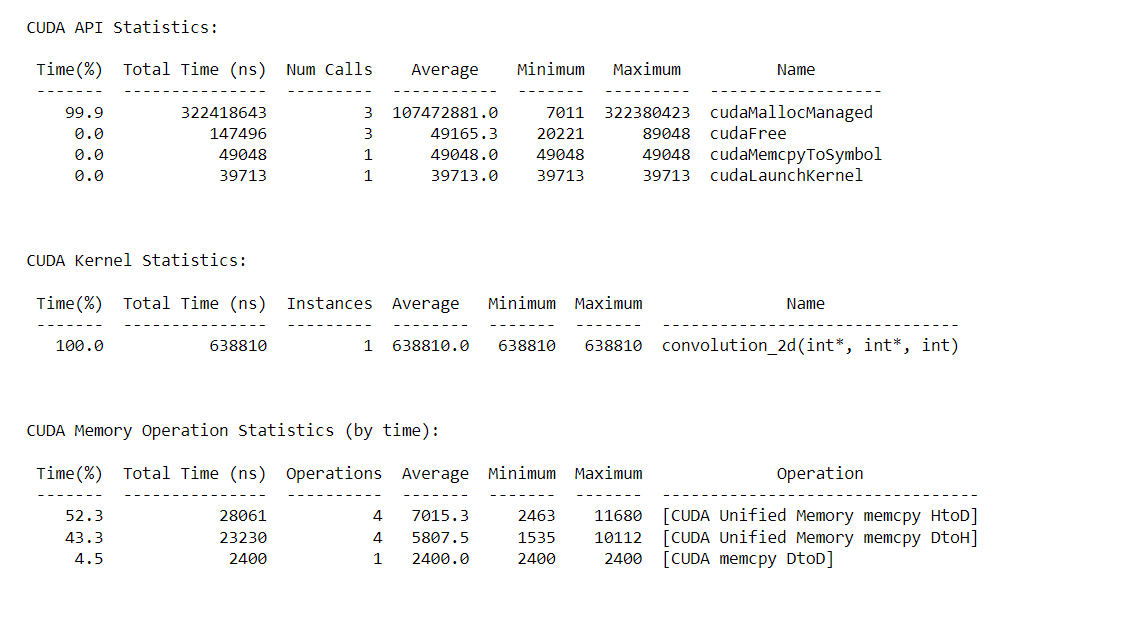
|  |  |  |
| --- | --- | --- |
| **Threads** | **2^10\*2^10** | **Speedup** |
| Serial Execution: | 14161311 | - |
| 4 | 4514933 | 3.1365 |
| 8 | 4055855 | 3.4915 |
| 16 | 460935 | 30.7230 |
| 32 | 432174 | 32.7676 |

**2^4**

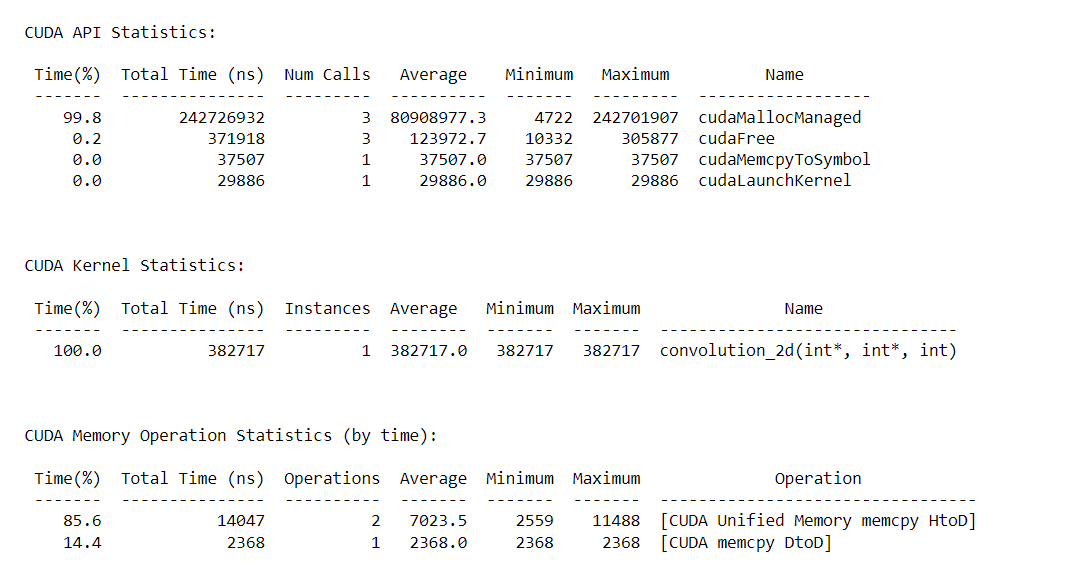
**Thread=1, Execution Time: 506396ns**



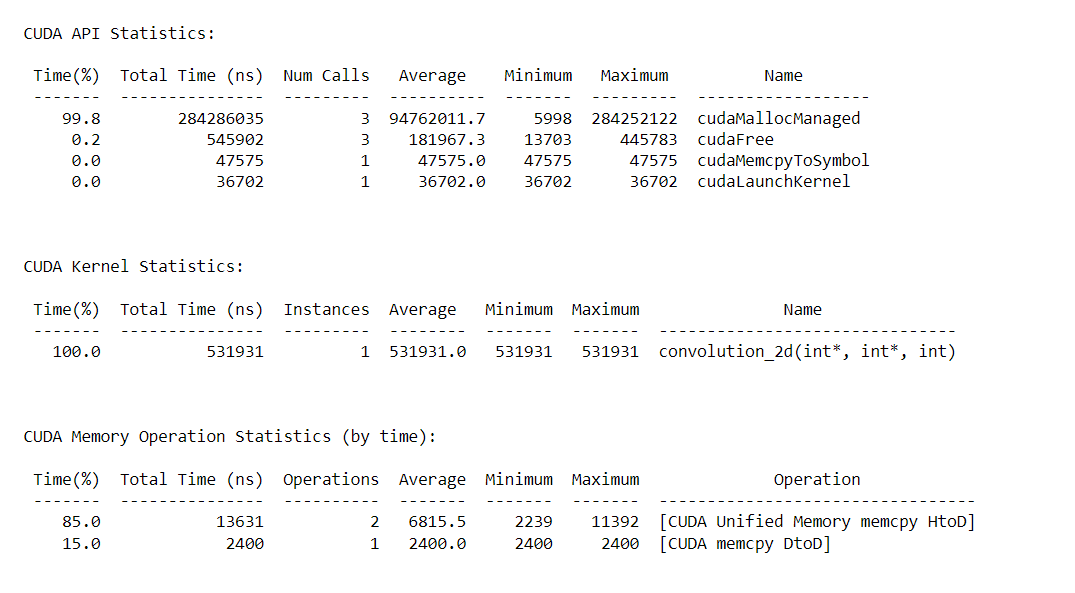
**Thread=4, Execution Time: 638810ns**



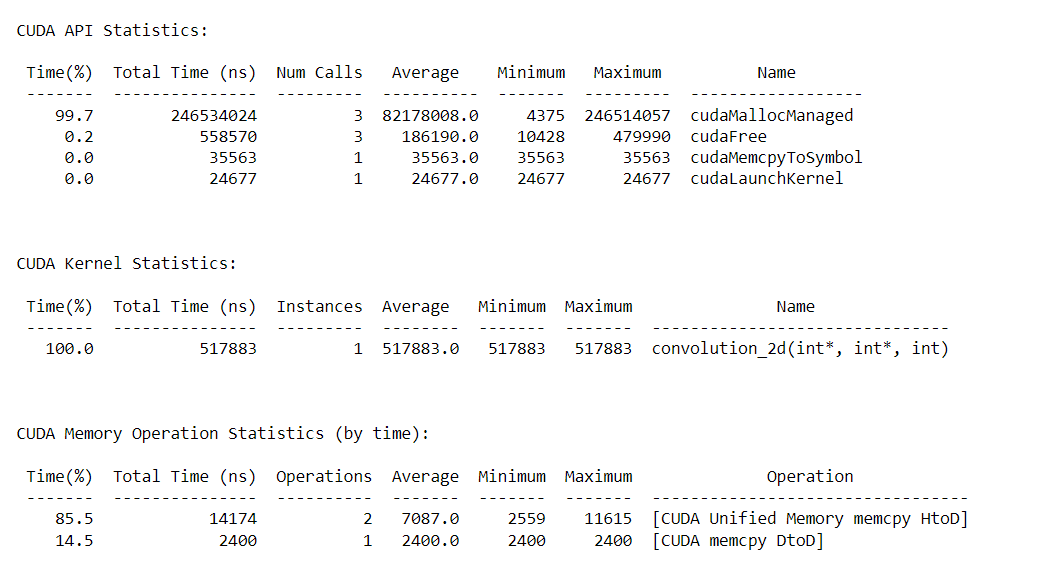
**Thread=8, Execution Time: 382717ns**



**Thread=16, Execution Time: 531931ns**

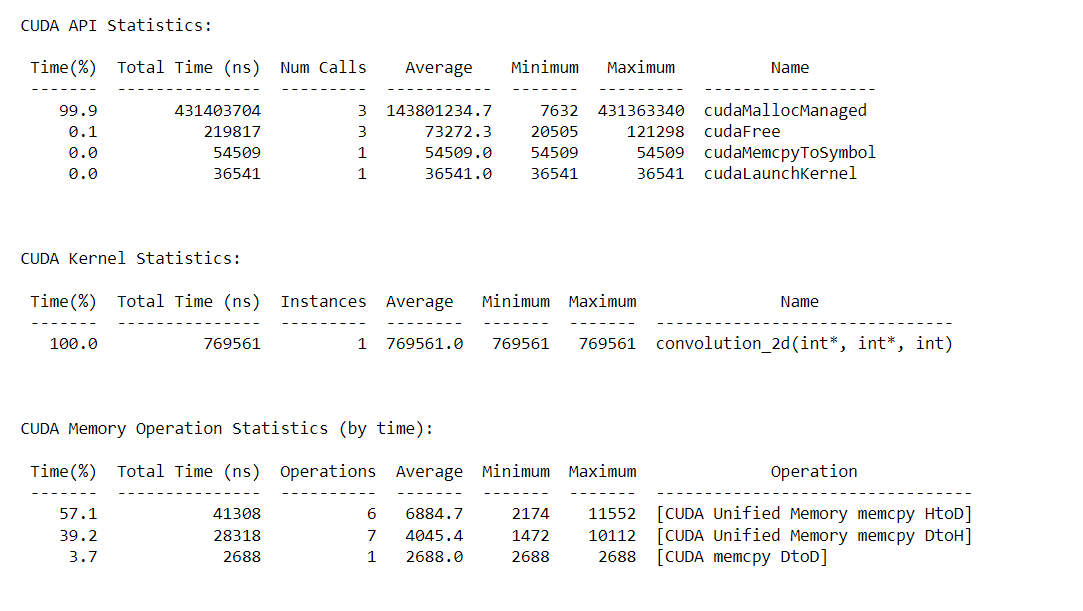


**Thread=32, Execution Time: 517883ns**

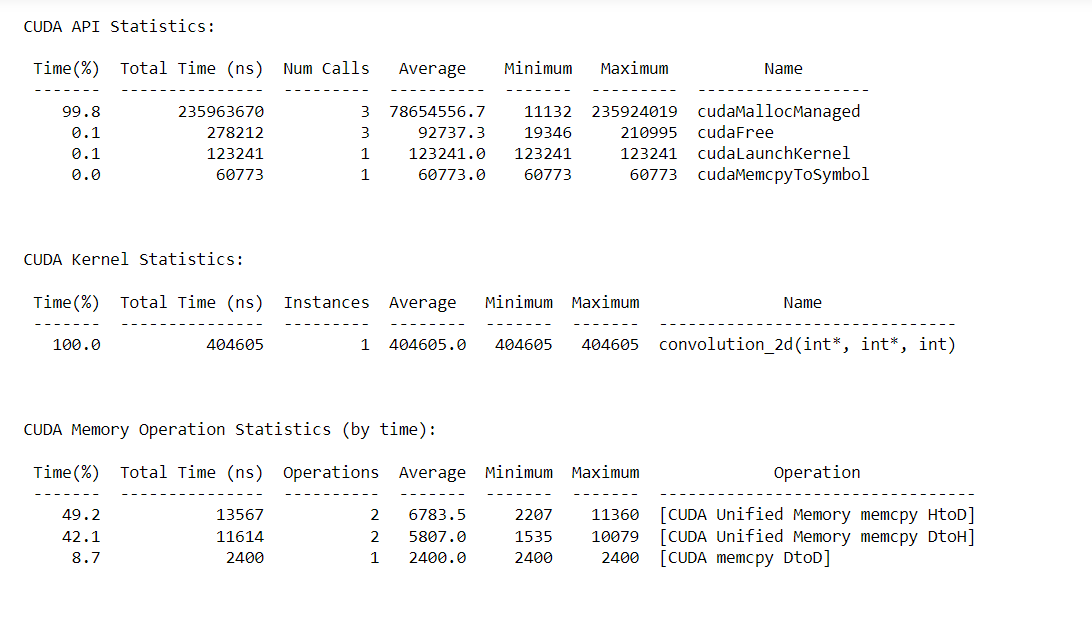


**2^5**

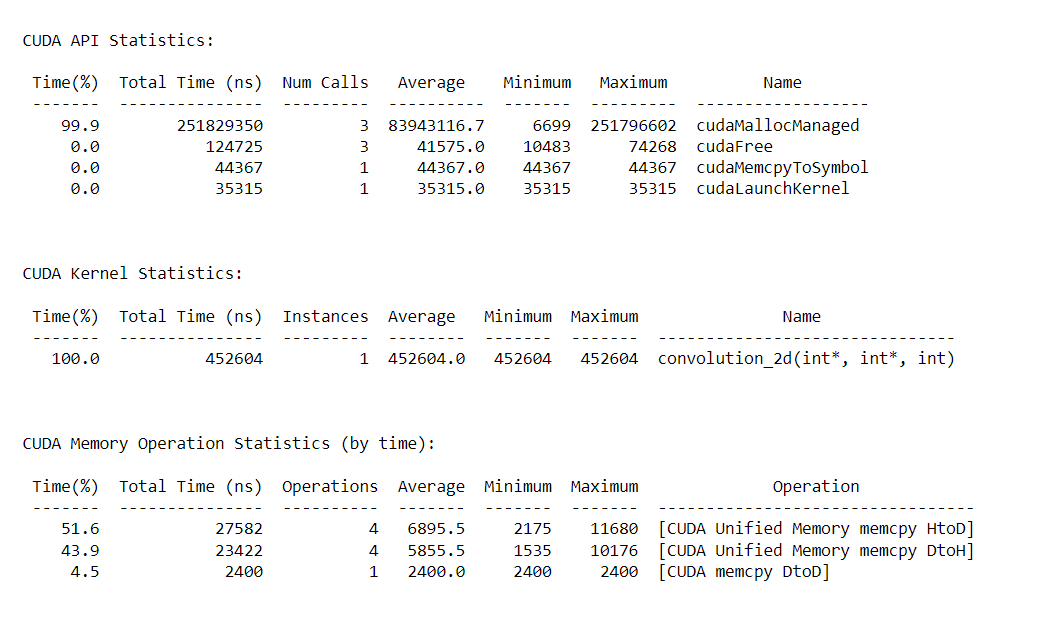
**Thread=1, Execution Time: 769561ns**



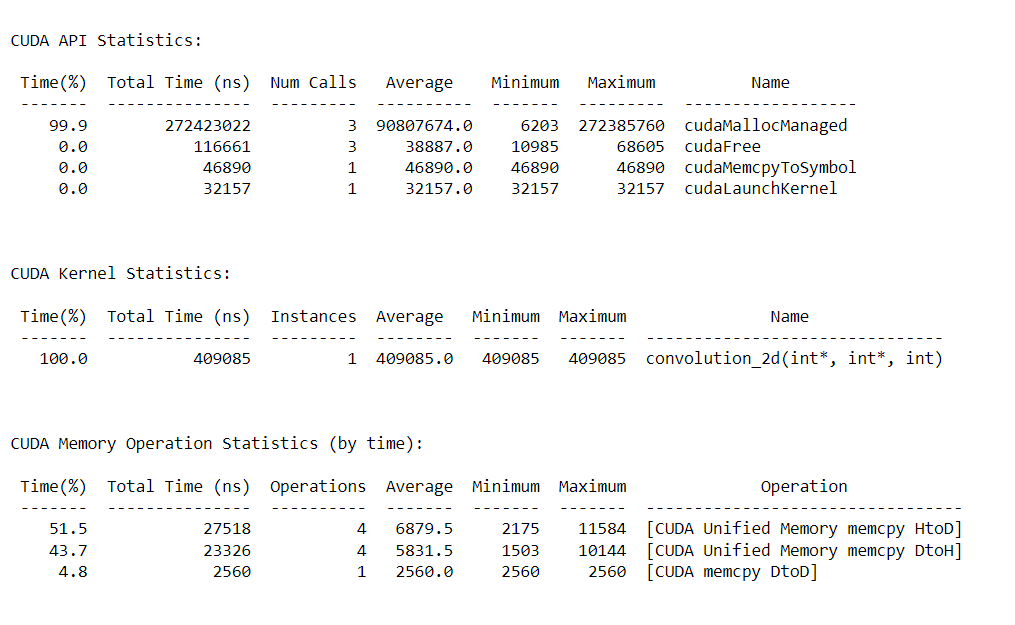
**Thread=4, Execution Time: 404605ns**



**Thread=8, Execution Time: 452604ns**



**Thread=16, Execution Time: 409085ns**



**Thread=32, Execution Time: 560955ns**

