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);

dim3threadBlock(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);

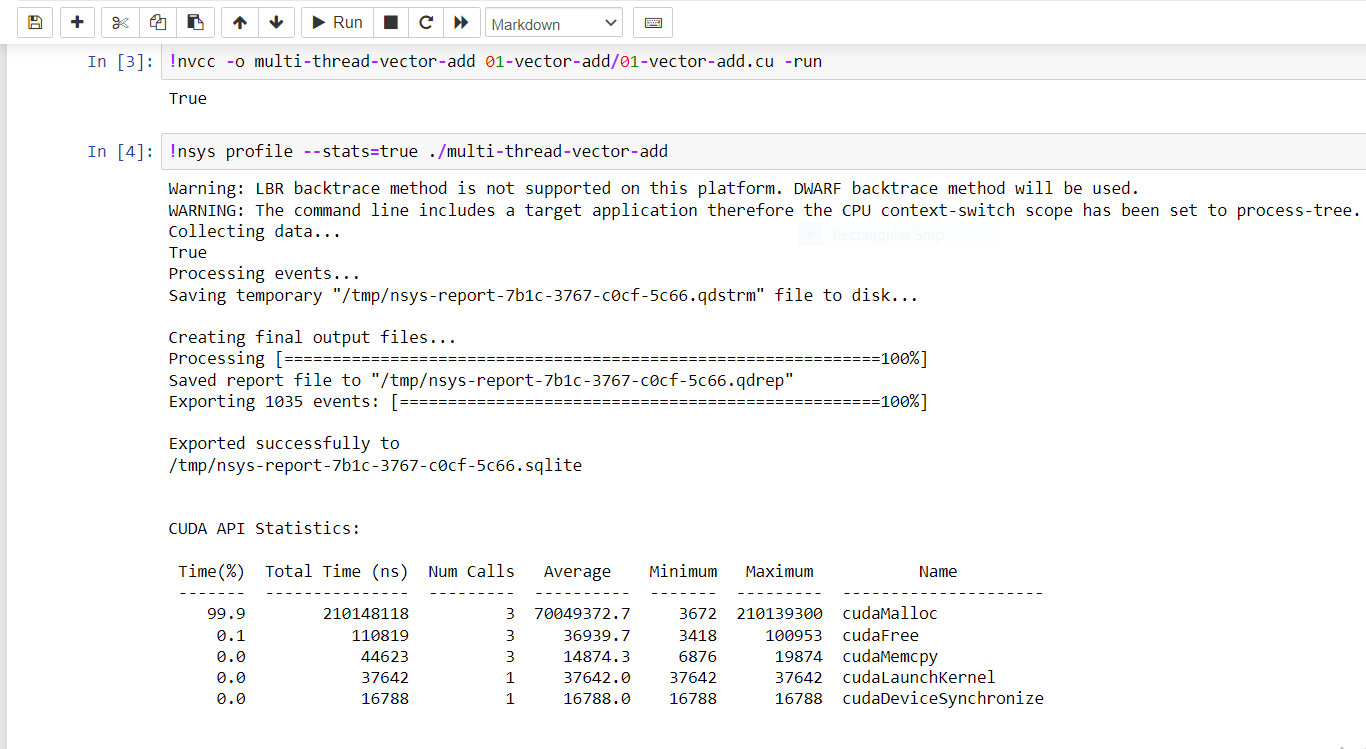
cudaFree(f);

printf("True\n");

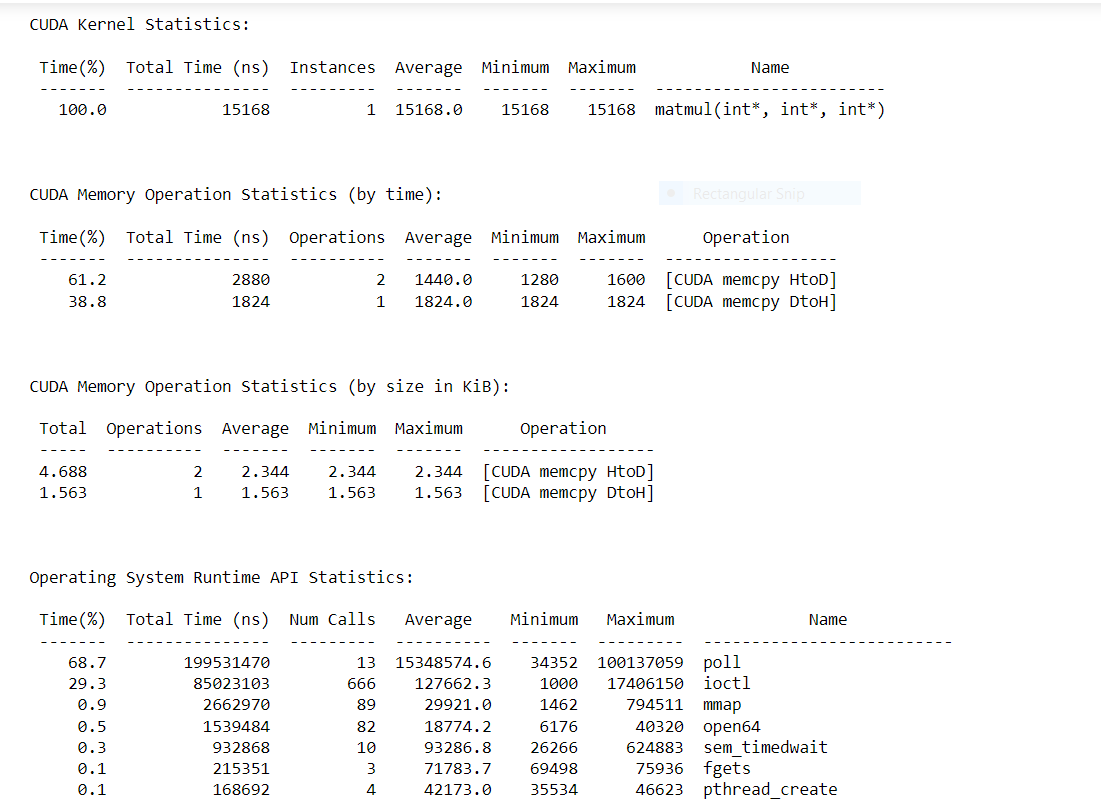
return 0;

}

**Screenshot 2:**

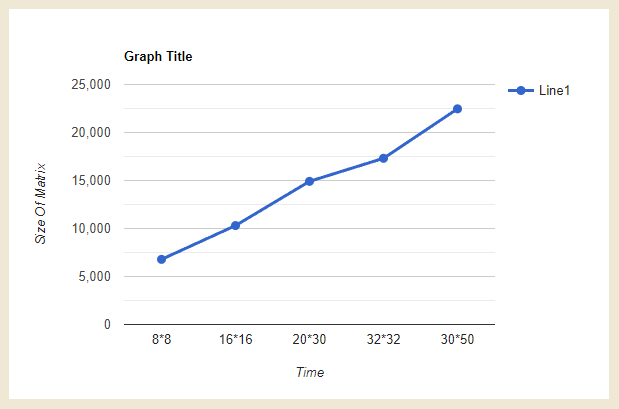


**Screenshot 3:**



**Analysis –**

|  |  |
| --- | --- |
| **Problem Size** | **Time Required** |
| 8\*8 | 6784 |
| 16\*16 | 10304 |
| 20\*30 | 14911 |
| 32\*32 | 17312 |
| 30\*50 | 22464 |



#include <stdio.h>

#include <string.h>

#include <cuda.h>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include "math.h"

#include "time.h"

#include <iostream>

#include <fstream>

#include <iomanip>

#define BLOCK\_SIZE 32

voidprint\_matrices(float\* matrix, char\*file\_Name, intx\_dim, inty\_dim, int dim)

{

    std::ofstreamoutFile;

    outFile.open (file\_Name);

    outFile<<std::fixed;

    outFile<<std::setprecision(2);

    for (int i = 0; i <x\_dim; i++) {

        for (int j = 0; j <y\_dim; j++) {

            outFile<< matrix[i \* dim + j] <<" ";

        }

        outFile<<std::endl;

    }

}

\_\_host\_\_voidcpu\_matrix\_mult(float\*h\_a, float\*h\_b, float\*h\_result, int m) {

    for (int i = 0; i < m; ++i)

    {

        for (int j = 0; j < m; ++j)

        {

            floattmp = 0.0;

            for (int h = 0; h < m; ++h)

            {

                tmp += h\_a[i \* m + h] \* h\_b[h \* m + j];

            }

            h\_result[i \* m + j] = tmp;

        }

    }

}

\_\_host\_\_intfill(float\*\*Lmatrix, float\*\*Rmatrix, intLdimX, intLdimY, intRdimX, intRdimY) {

    intsqr\_dim\_X, sqr\_dim\_Y, size;

    sqr\_dim\_X = RdimX;

    if (LdimX>RdimX) {

        sqr\_dim\_X = LdimX;

    }

    sqr\_dim\_Y = RdimY;

    if (LdimY>RdimY) {

        sqr\_dim\_Y = LdimY;

    }

    size = sqr\_dim\_Y;

    if (sqr\_dim\_X>sqr\_dim\_Y) {

        size = sqr\_dim\_X;

    }

    int temp = size / BLOCK\_SIZE + (size % BLOCK\_SIZE == 0 ? 0 : 1);

    size = temp \* BLOCK\_SIZE;

    size\_tpt\_size = size \* size \* sizeof(float);

    \*Lmatrix = (float \*) malloc(pt\_size);

    \*Rmatrix = (float \*) malloc(pt\_size);

    memset(\*Lmatrix, 0, pt\_size);

    memset(\*Rmatrix, 0, pt\_size);

    for (int i = 0; i <LdimX; i++) {

        for (int j = 0; j <LdimY; j++) {

            int dummy = size \* i + j;

            (\*Lmatrix)[dummy] = sinf(dummy);

        }

    }

    for (int i = 0; i <RdimX; i++) {

        for (int j = 0; j <RdimY; j++) {

            int dummy = size \* i + j;

            (\*Rmatrix)[dummy] = cosf(dummy);

        }

    }

    return size;

}

\_\_global\_\_voidmultiply(float\*left, float\*right, float\*res, int dim) {

    inti,j;

    float temp = 0;

    \_\_shared\_\_floatLeft\_shared\_t [BLOCK\_SIZE][BLOCK\_SIZE];

    \_\_shared\_\_floatRight\_shared\_t[BLOCK\_SIZE][BLOCK\_SIZE];

    // Row i of matrix left

    int row = blockIdx.y \* blockDim.y + threadIdx.y;

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

    for (inttileNUM = 0; tileNUM<gridDim.x; tileNUM++) {

        // Column j of matrix left

        j = tileNUM \* BLOCK\_SIZE + threadIdx.x;

        i = tileNUM \* BLOCK\_SIZE + threadIdx.y;

        // Load left[i][j] to shared mem

        Left\_shared\_t[threadIdx.y][threadIdx.x] = left[row \* dim + j];// Coalesced access

        // Load right[i][j] to shared mem

        Right\_shared\_t[threadIdx.y][threadIdx.x] = right[i \* dim + col]; // Coalesced access

        // Synchronize before computation

        \_\_syncthreads();

        // Accumulate one tile of res from tiles of left and right in shared mem

        for (int k = 0; k <BLOCK\_SIZE; k++) {

            temp += Left\_shared\_t[threadIdx.y][k] \* Right\_shared\_t[k][threadIdx.x]; //no shared memory bank conflict

        }

        // Synchronize

        \_\_syncthreads();

    }

    // Store accumulated value to res

    res[row \* dim + col] = temp;

}

int main(void)

{

    intLeft\_matrix\_x, Left\_matrix\_y, Right\_matrix\_x, Right\_matrix\_y, Left\_vector\_size, Right\_vector\_size;

    float \*Left\_Vector\_h, \*Right\_Vector\_h, \*Left\_Vector\_d, \*Right\_Vector\_d, \*Res\_h, \*Res\_d, \*CPU;

    Left\_matrix\_x = Left\_matrix\_y= Right\_matrix\_x= Right\_matrix\_y= 1024;

    int dim = fill(&Left\_Vector\_h, &Right\_Vector\_h, Left\_matrix\_x, Left\_matrix\_y, Right\_matrix\_x, Right\_matrix\_y);

    size\_tvector\_size;

    vector\_size = dim\*dim \* sizeof(float);

    Res\_h = (float \*) malloc(vector\_size);

    CPU = (float \*) malloc(vector\_size);

    cudaMalloc((void \*\*) &Left\_Vector\_d, vector\_size);

    cudaMalloc((void \*\*) &Right\_Vector\_d, vector\_size);

    cudaMalloc((void \*\*) &Res\_d, vector\_size);

    cudaMemcpy(Left\_Vector\_d, Left\_Vector\_h, vector\_size, cudaMemcpyHostToDevice);

    cudaMemcpy(Right\_Vector\_d, Right\_Vector\_h, vector\_size, cudaMemcpyHostToDevice);

    dim3Block\_dim(BLOCK\_SIZE, BLOCK\_SIZE);

    dim3Grid\_dim(dim / BLOCK\_SIZE, dim / BLOCK\_SIZE);

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start,0);

    multiply<<<Grid\_dim, Block\_dim>>> (Left\_Vector\_d, Right\_Vector\_d, Res\_d, dim);

    cudaEventRecord(stop,0);

    cudaEventSynchronize(stop);

    float et;

    cudaEventElapsedTime(&et, start, stop);

    cudaEventDestroy(start);

    cudaEventDestroy(stop);

    cudaMemcpy(Res\_h, Res\_d, vector\_size, cudaMemcpyDeviceToHost);

    clock\_t begin = clock();

    cpu\_matrix\_mult(Left\_Vector\_h,Right\_Vector\_h,CPU,dim);

    clock\_t end = clock();

    doubletime\_spent = (double)1000\*(end - begin) / CLOCKS\_PER\_SEC;

    printf("GPU time= %fms\n", et);

    printf("CPU time= %lf ms\n", time\_spent);

    print\_matrices(Res\_h,"GPU\_out",Left\_matrix\_x,Right\_matrix\_y,dim);

    print\_matrices(CPU,"CPU\_out",Left\_matrix\_x,Right\_matrix\_y,dim);

    booleqaul = true;

    for (int i=0;i<Left\_matrix\_x&&eqaul;i++){

        for (int j = 0; j <Right\_matrix\_y&&eqaul; j++) {

            if (abs(Res\_h[i\*dim+j]-CPU[i\*dim+j]) >0.001)

            {

                eqaul = false;

                printf("NOT EQUAL\n");

            }

        }

    }

    if (eqaul)

    {

        std::cout<<"Results are equal!"<<std::endl;

    }

    else

    {

        std::cout<<"Results are NOT equal!"<<std::endl;

    }

    free(Left\_Vector\_h);

    free(Right\_Vector\_h);

    free(Res\_h);

    free(CPU);

    cudaFree(Left\_Vector\_d);

    cudaFree(Right\_Vector\_d);

    cudaFree(Res\_d);

}

Shared Memory Matrix matrixmultiplication :

8 blocks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No of elements | 16 | 256 | 512 | 1024 |
| CPU execution in ms | 0.015000 | 53.750000 | 888.633000 | 7180.0000 |
| GPU execution  In ms | 0.018144 | 0.1580160 | 1.17350400 | 9.1251000 |
| SpeedUp | 0.8267 | 340.18 | 757.34 | 786.84 |

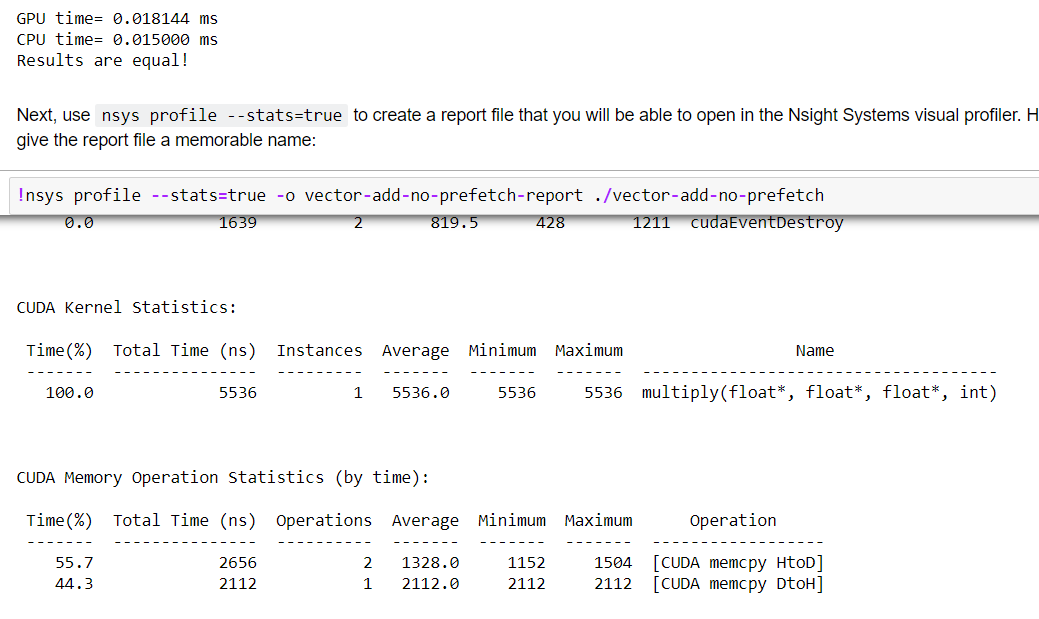
16 blocks :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No of elements | 16 | 256 | 512 | 1024 |
| CPU execution in ms | 0.017824 | 54.507000 | 711.363000 | 7623.437000 |
| GPU execution  In ms | 0.015000 | 0.112672 | 0.746944 | 5.779552 |
| SpeedUp | 1.18 | 446.7 | 952.36 | 1319.72 |

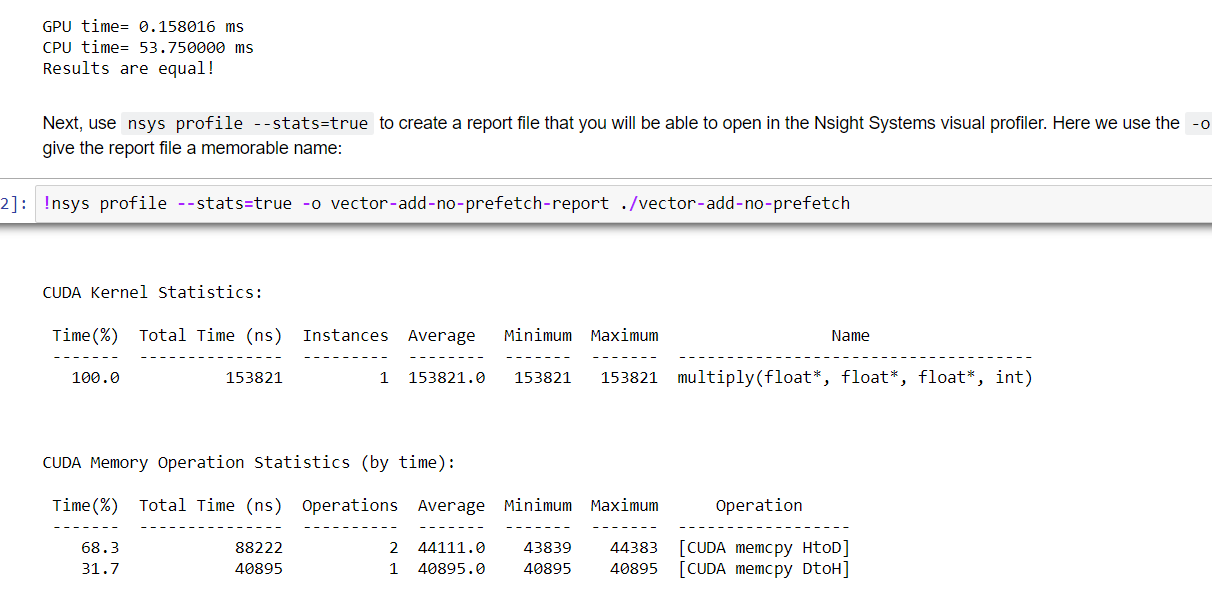
32 blocks:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No of elements | 16 | 256 | 512 | 1024 |
| CPU execution in ms | 0.099000 | 53.936000 | 793.646000 | 7338.183000 |
| GPU execution  In ms | 0.021792 | 0.114752 | 0.708256 | 5.286784 |
| SpeedUp | 4.56 | 470.23 | 1120.56 | 1388.02399 |

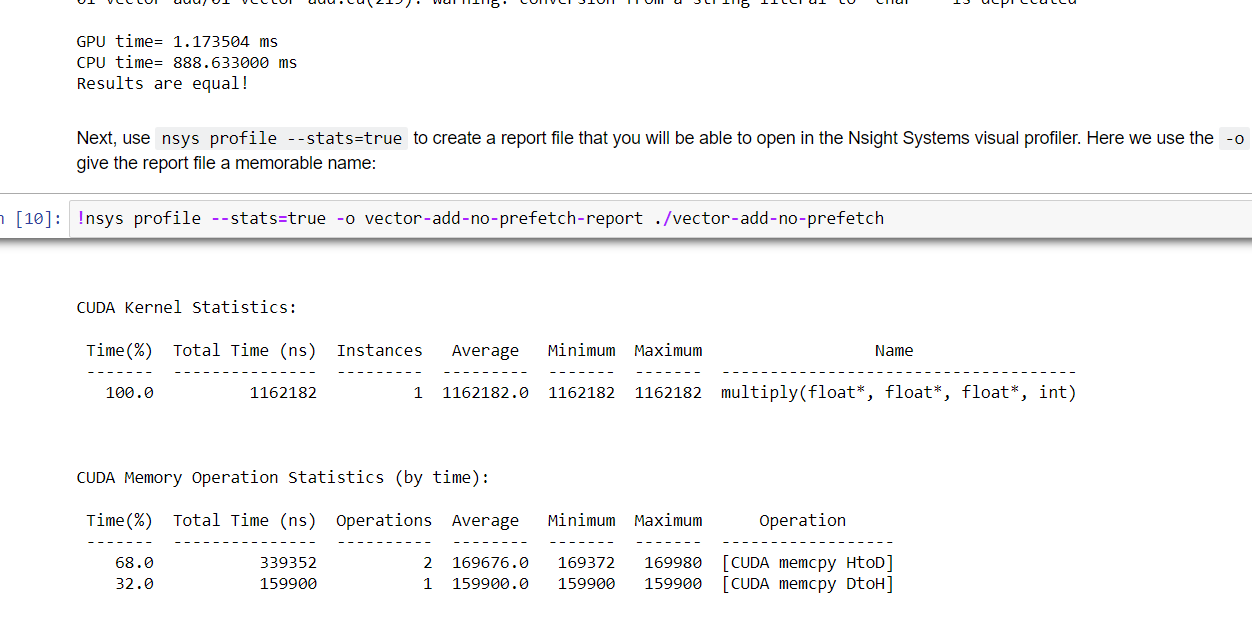
16



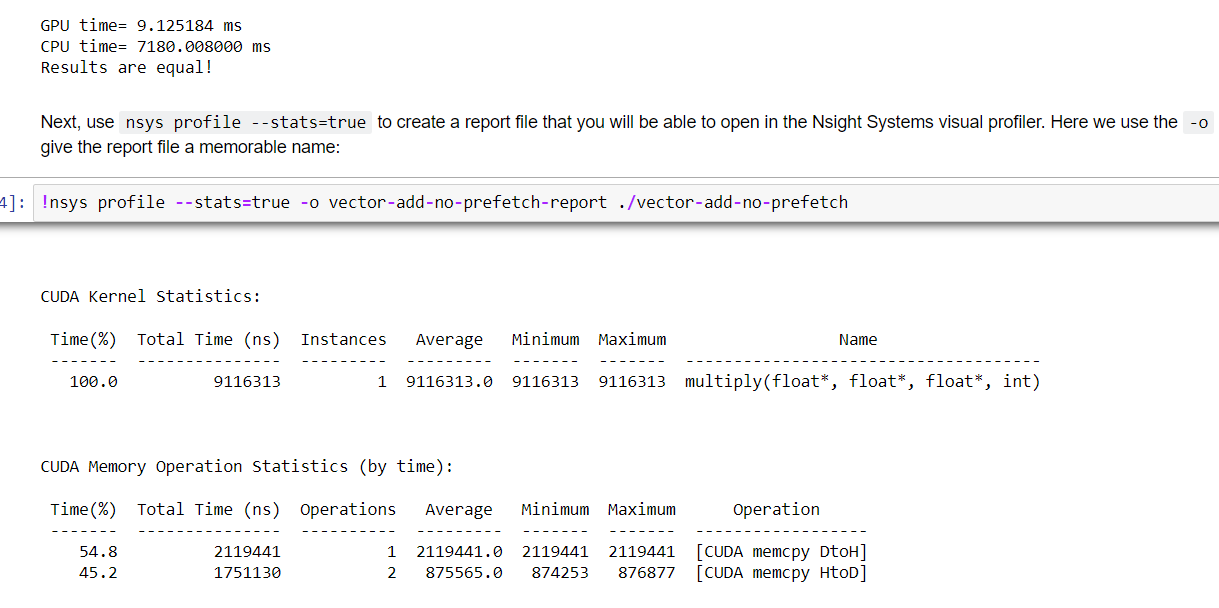
256:



512 :



1024 :



Conclusion :

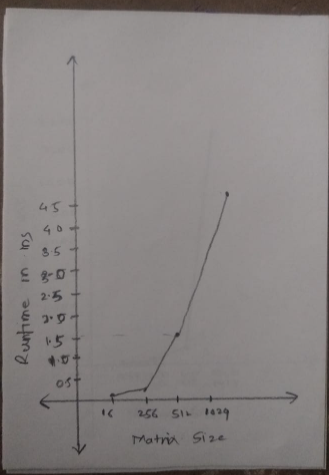
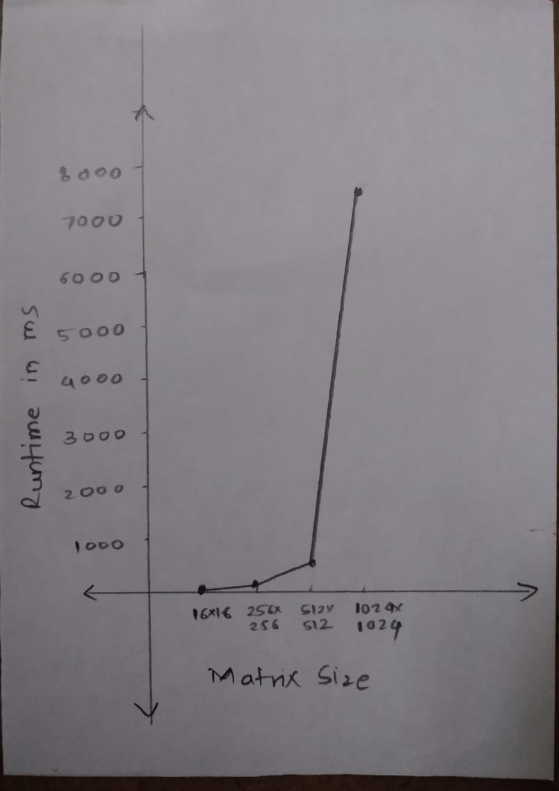
As we can see that by keeping the number of block tile constant and increasing matrix dimension speed up of algo is increasing.

Also as we can goes on increasing the no of elements the time for host to device and device to host memory transfer time also goes on increasing.

This show that GPU is more suitable for large no of computation.

As compared to the global memory it takes less time.

Memory is not a problem when global memory is considered because 1024x1024 matrix needs 4MB of space. However, shared memory size is limited and in order to provide concurrent execution of blocks in one SM shared memory must be divided wisely



Sequential Execution Parallel Execution

Prefix Sum

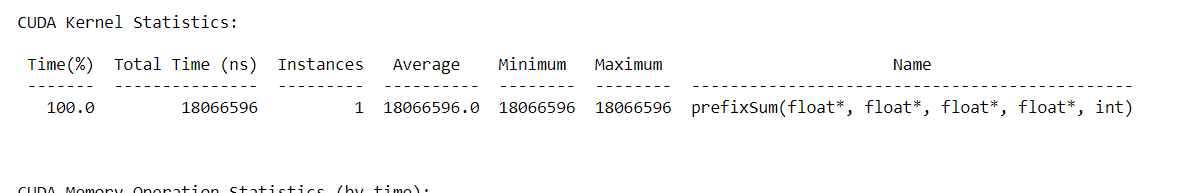
Prefix sum Program :

For given N we are changing number of threads

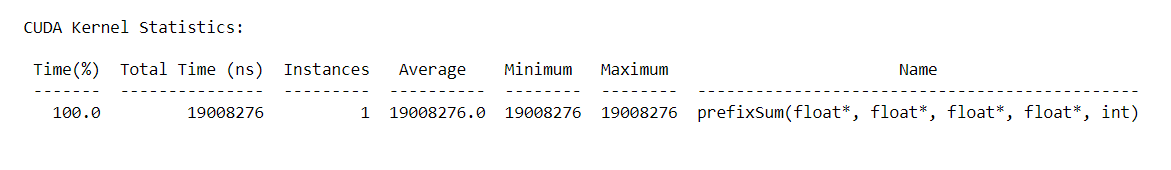
For 10^6 elements :

|  |  |  |  |
| --- | --- | --- | --- |
| No of Blocks | 1 32 | 1 64 | 1 128 |
| Time GPU Kernal in ns | 0.023671988 | 0.019737041 | 0.018066596 |
|  |  |  |  |
| No of Blocks | 1 256 | 1 512 | 1 1 |
| Time GPU Kernal in ns | 0.014066596 | 0.010667050 | 0.226743678 |

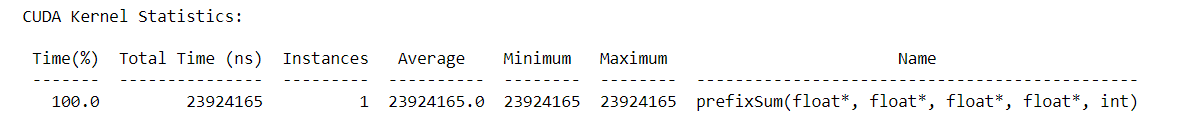
128



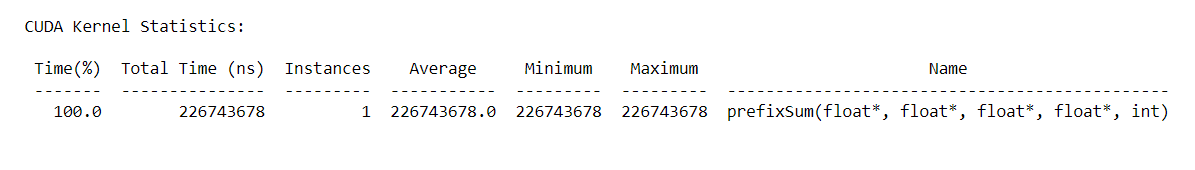
64



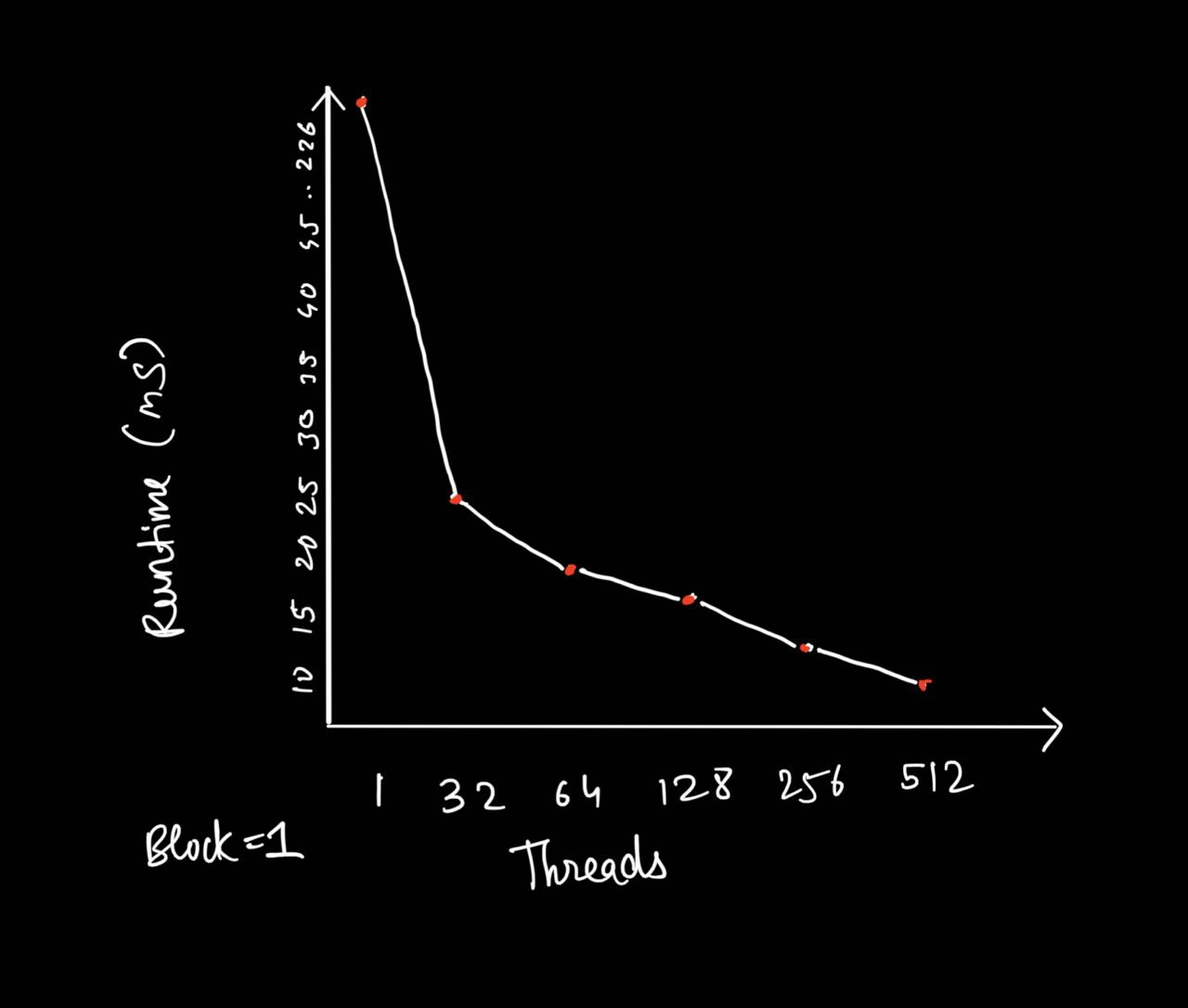
32



1



As we can see that by keeping the number of blocks constant, increasing the number of threads for the same size the time is decreasing .

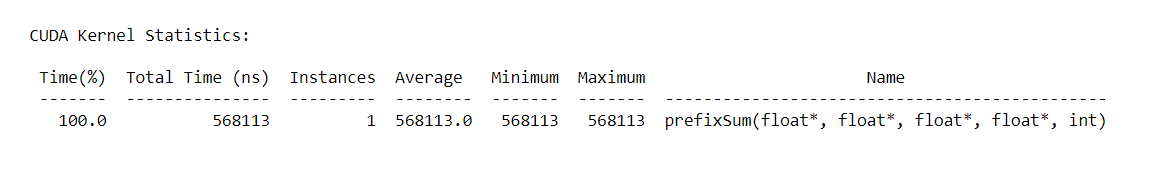


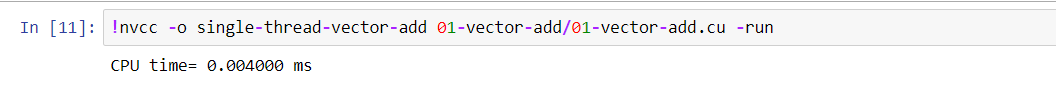
For given blocks and threads changing number of elements

1 block and 64 threads

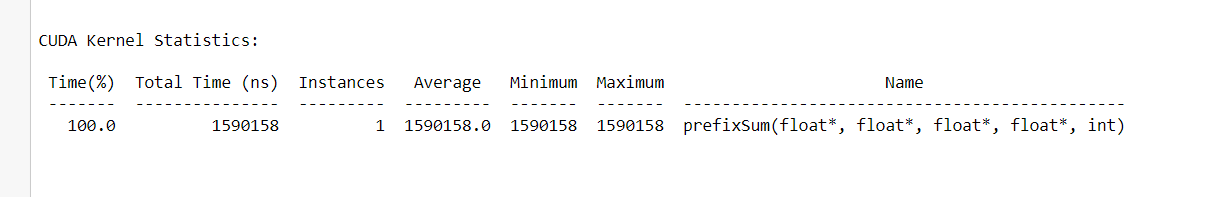
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No of elements | 1000 | 10000 | 100000 | 1000000 |
| CPU execution in ns | 4000 | 32000 | 312000 | 3041000 |
| GPU execution  In ns | 568113 | 1590158 | 2503075 | 19737041 |
| SpeedUp | 0.007 | 0.02 | 0.125 | 0.154 |

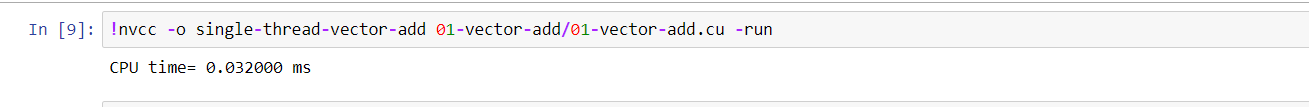
10^3



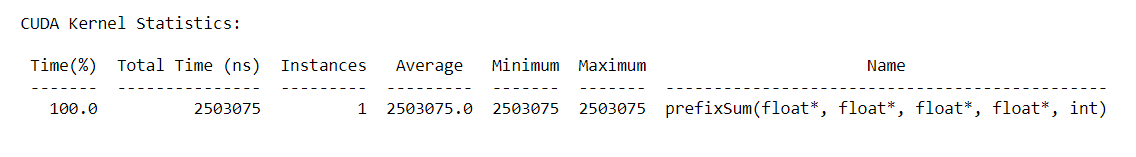


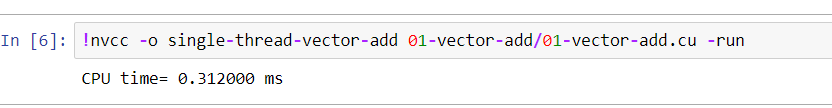
10^4



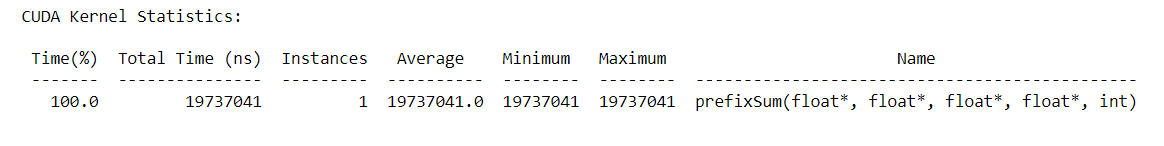


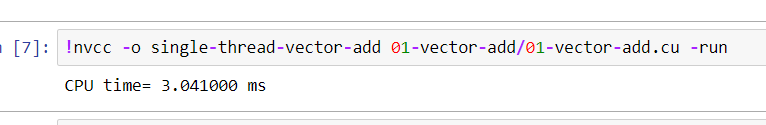
10^5

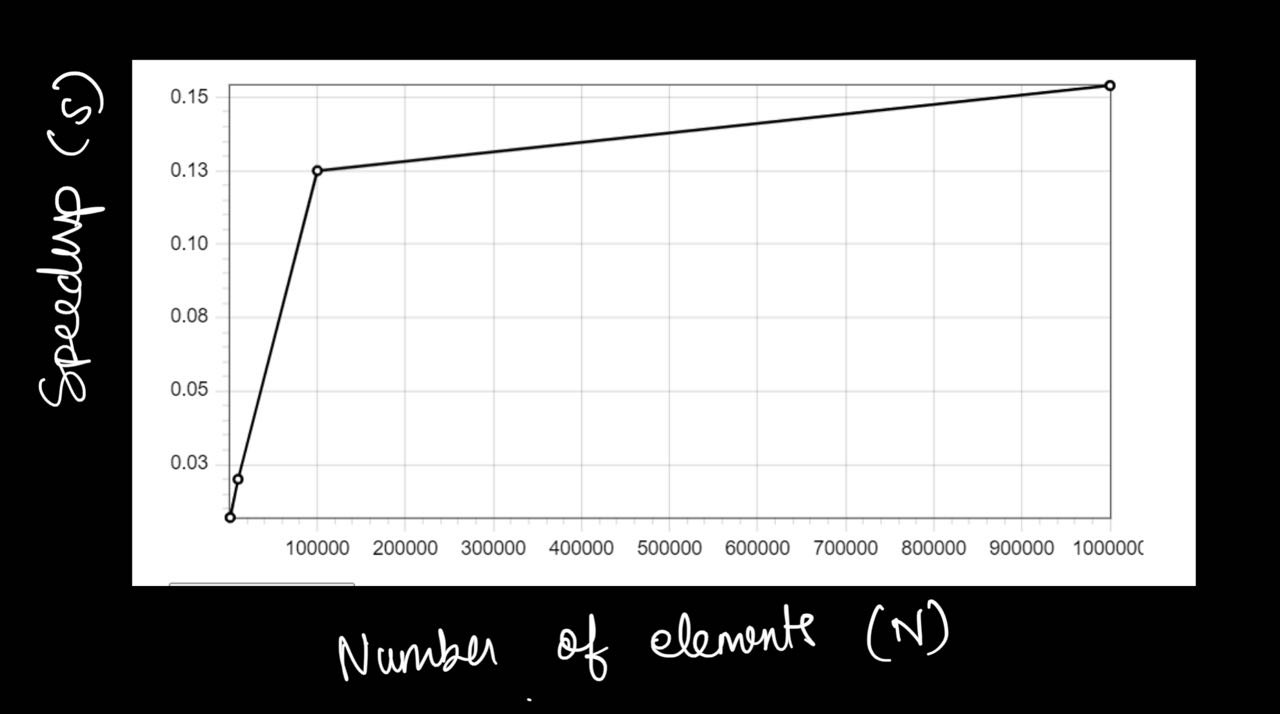




10^6







Conclusion :

As we can see that by keeping the number of block and threads constant and

increasing the dimension of the array , the speed up of the algorithm is increasing.

Also as we can go on increasing the number of elements the time for host to device and device to host memory transfer time also goes on increasing.

This shows that the GPU is more suitable for large computation.

Memory is not a problem when global memory is considered because 1024x1024 matrix needs 4MB of space. However, shared memory size is limited and in order to provide concurrent execution of blocks in one SM shared memory must be divided wisely

**Problem Statement 4:**

Implement 2D Convolution using shared memoryusing 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

#defineMASK\_DIM7

// Amount the the matrix will hang over the matrix

#defineMASK\_OFFSET (MASK\_DIM / 2)

// Allocate mask in constant memory

\_\_constant\_\_intmask[7\*7];

// 2D Convolution Kernel

// Takes:

//  matrix: Input matrix

//  result: Convolution result

//  N:      Dimensions of the matrices

\_\_global\_\_voidconvolution\_2d(int\*matrix, int\*result, intN) {

  // 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

  intstart\_r= row - MASK\_OFFSET;

  intstart\_c= col - MASK\_OFFSET;

  // Temp value for accumulating the result

  int temp =0;

  // Iterate over all the rows

  for (inti=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)

voidinit\_matrix(int\*m, intn) {

  for (inti=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

voidverify\_result(int\*m, int\*mask, int\*result, intN) {

  // Temp value for accumulating results

  int temp;

  // Intermediate value for more readable code

  intoffset\_r;

  intoffset\_c;

  // Go over each row

  for (inti=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);

    }

  }

}

intmain() {

  // Dimensions of the matrix (2 ^ 10 x 2 ^ 10)

  int N =1<<10;

  // Size of the matrix (in bytes)

  size\_tbytes\_n= N \* N \*sizeof(int);

  // Allocate the matrix and initialize it

  int\*matrix =newint[N \* N];

  int\*result =newint[N \* N];

  init\_matrix(matrix, N);

  // Size of the mask in bytes

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

  // Allocate the mask and initialize it

  int\*h\_mask=newint[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

  dim3block\_dim(THREADS, THREADS);

  dim3grid\_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);

  return0;

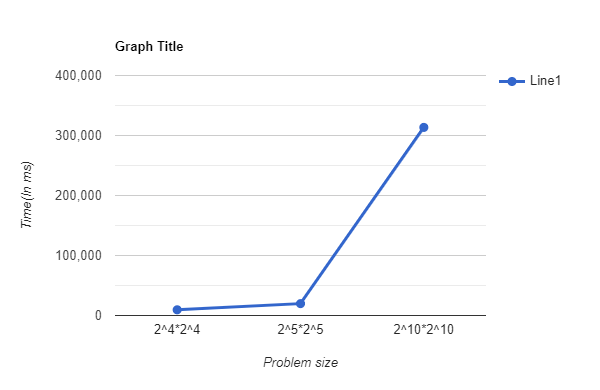
}

**Screenshot 11:**

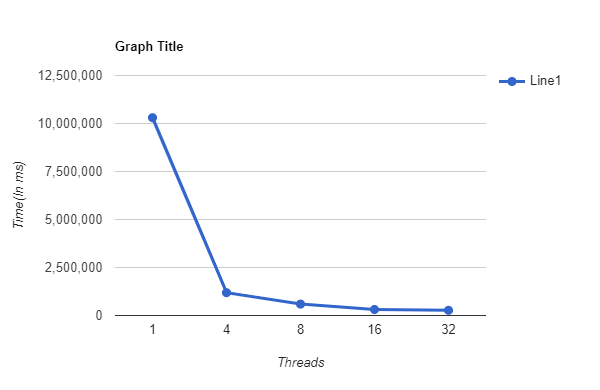
**Stats:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Problem size** | 2^4\* 2^4 | 2^5 \* 2^5 | 2 ^ 10 \* 2^10 |
| **No.of Threads** | **time** | **time** | **time** |
| **1** | 11296 | 18143 | 10303673 |
| **4** | 12064 | 12383 | 1187117 |
| **8** | 12256 | 12256 | 593431 |
| **16** | 9503 | 11776 | 313499 |
| **32** |  | 11488 | 270364 |

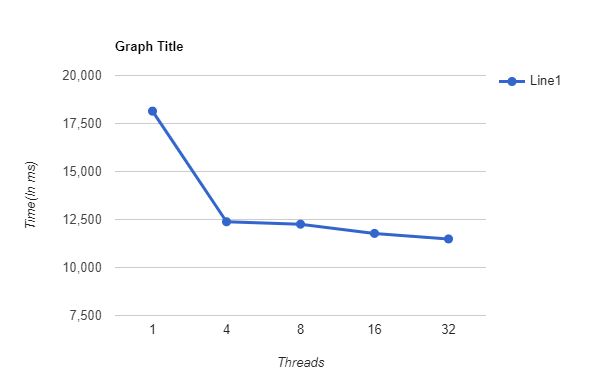
**Keeping thread constant graph against time and N**



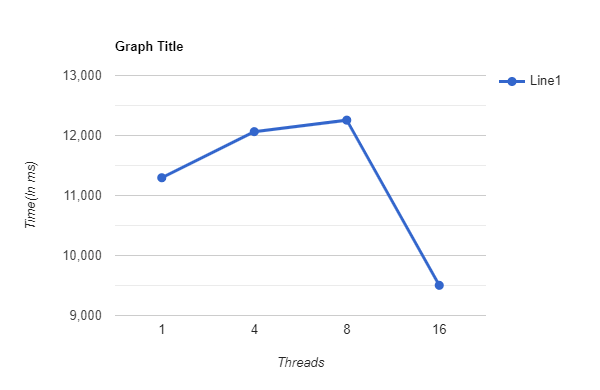
**Keeping problem size constant : 2^10 \* 2^10 matrix**



**2^5 \* 2^5 matrix**

****

**2^4 \* 2^4 matrix**

****