Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

High Performance Computing Lab

Class: Final Year (Computer Science and Engineering) **Year:** 2022-23

PRN: 2019BTECS00089 – Piyush Pramod Mhaske **Batch**: B3

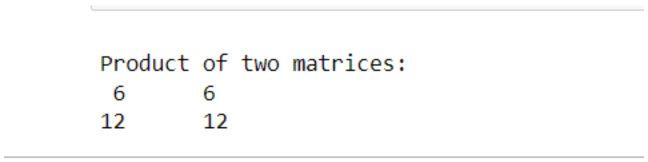
Practical 10

que 1:

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.

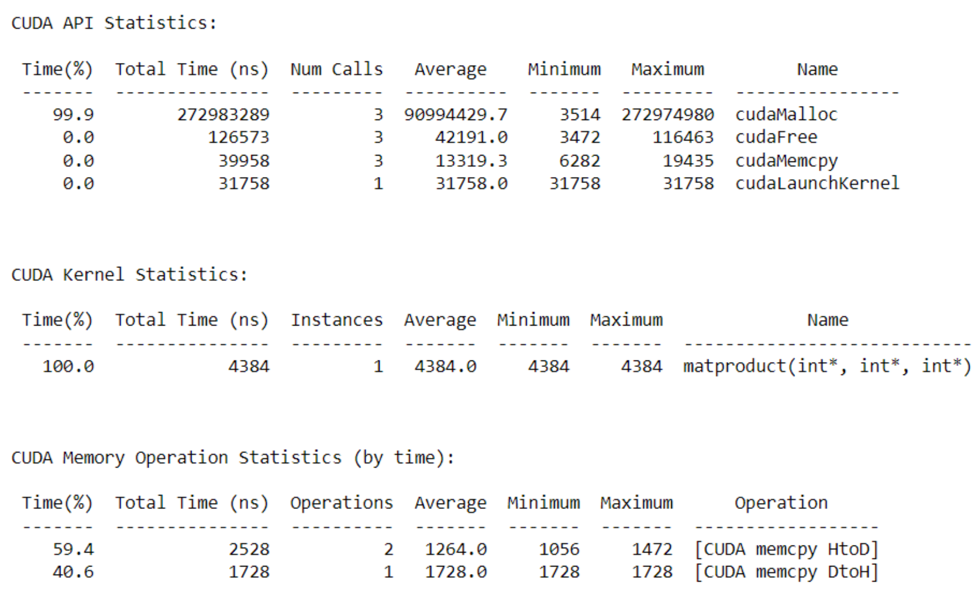
```
#include<stdio.h>
#include<cuda.h>
#define row1 2 /* Number of rows of first matrix */
#define col1 3 /* Number of columns of first matrix */
#define row2 3 /* Number of rows of second matrix */
#define col2 2 /* Number of columns of second matrix */

__global__ void matproduct(int *l,int *m, int *n)
{
    int x=blockIdx.x;
    int y=blockIdx.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]={{1,1,1},{2,2,2}};
    int b[row2][col2]={{1,1},{2,2},{3,3}};
    int c[row1][col2];
    int *d,*e,*f;
    int i,j;

    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);
/* Here we are defining two dimensional Grid(collection of blocks) structure. Syntax is dim3 grid(no. of columns,no. of rows) */

    matproduct<<<grid,1>>>(d,e,f);

 cudaMemcpy(c,f,row1*col2*sizeof(int),cudaMemcpyDeviceToHost);
    printf("\nProduct of two matrices:\n ");
    for(i=0;i<row1;i++)
    {
        for(j=0;j<col2;j++)
        {
              printf("%d\t",c[i][j]);
        }
        printf("\n");
    }

    cudaFree(d);
    cudaFree(e);
    cudaFree(f);

    return 0;
}
```

Output :



```
Product of two matrices:
 6       6
12      12
```

Profilling :



```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average    Minimum   Maximum          Name
 -------  ---------------  ---------  ----------  -------  ---------  ----------------
    99.9        272983289          3  90994429.7     3514  272974980  cudaMalloc
     0.0           126573          3     42191.0     3472     116463  cudaFree
     0.0            39958          3     13319.3     6282      19435  cudaMemcpy
     0.0            31758          1     31758.0    31758      31758  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum              Name
 -------  ---------------  ---------  -------  -------  -------  ----------------------------
   100.0             4384          1   4384.0     4384     4384  matproduct(int*, int*, int*)


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum      Operation
 -------  ---------------  ----------  -------  -------  -------  ------------------
    59.4             2528           2   1264.0     1056     1472  [CUDA memcpy HtoD]
    40.6             1728           1   1728.0     1728     1728  [CUDA memcpy DtoH]
```

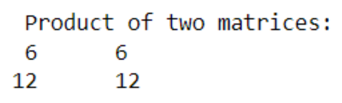
Que 2:

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>
#include<cuda.h>
#define row1 2 /* Number of rows of first matrix */
#define col1 3 /* Number of columns of first matrix */
#define row2 3 /* Number of rows of second matrix */
#define col2 2 /* Number of columns of second matrix */

__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();
```

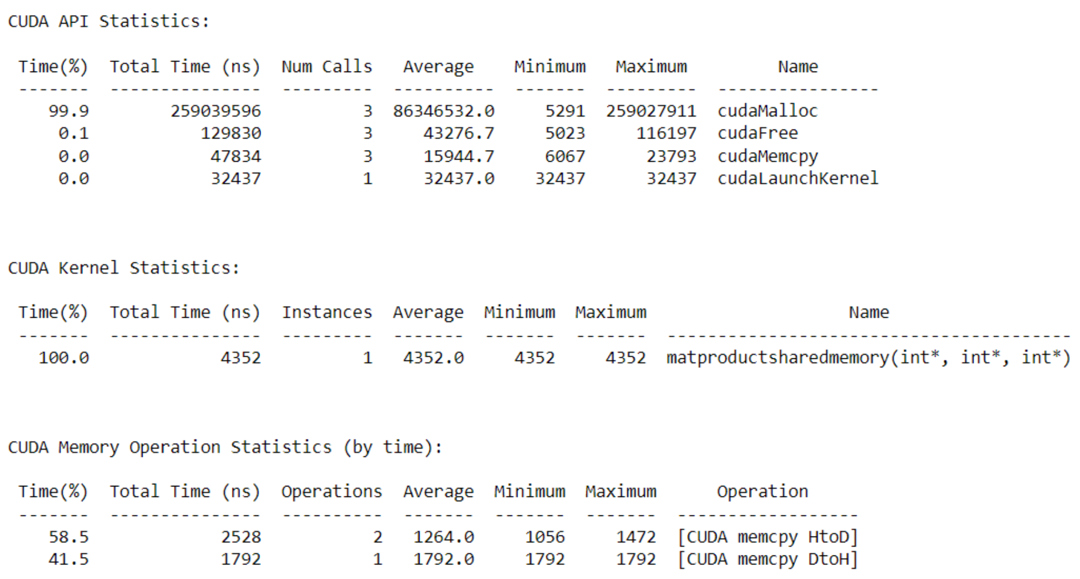
```
   for(i=0;i<col1;i++)
   n[col2*y+x]=n[col2*y+x]+p[i];
 }

 int main()
 {
     int a[row1][col1]={{1,1,1},{2,2,2}};
     int b[row2][col2]={{1,1},{2,2},{3,3}};
     int c[row1][col2];
     int *d,*e,*f;
     int i,j;

    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);
 /* Here we are defining two dimensional Grid(collection of blocks) structure. Syntax is dim3 grid(no. of columns,no. of rows) */

 matproductsharedmemory<<<grid,col1>>>(d,e,f);

  cudaMemcpy(c,f,row1*col2*sizeof(int),cudaMemcpyDeviceToHost);

  printf("\n Product of two matrices:\n ");
     for(i=0;i<row1;i++)
     {
         for(j=0;j<col2;j++)
         {
               printf("%d\t",c[i][j]);
         }
         printf("\n");
     }

     cudaFree(d);
     cudaFree(e);
     cudaFree(f);

     return 0;
 }
```

Output:



```
 Product of two matrices:
 6      6
12      12
```

Profiling:



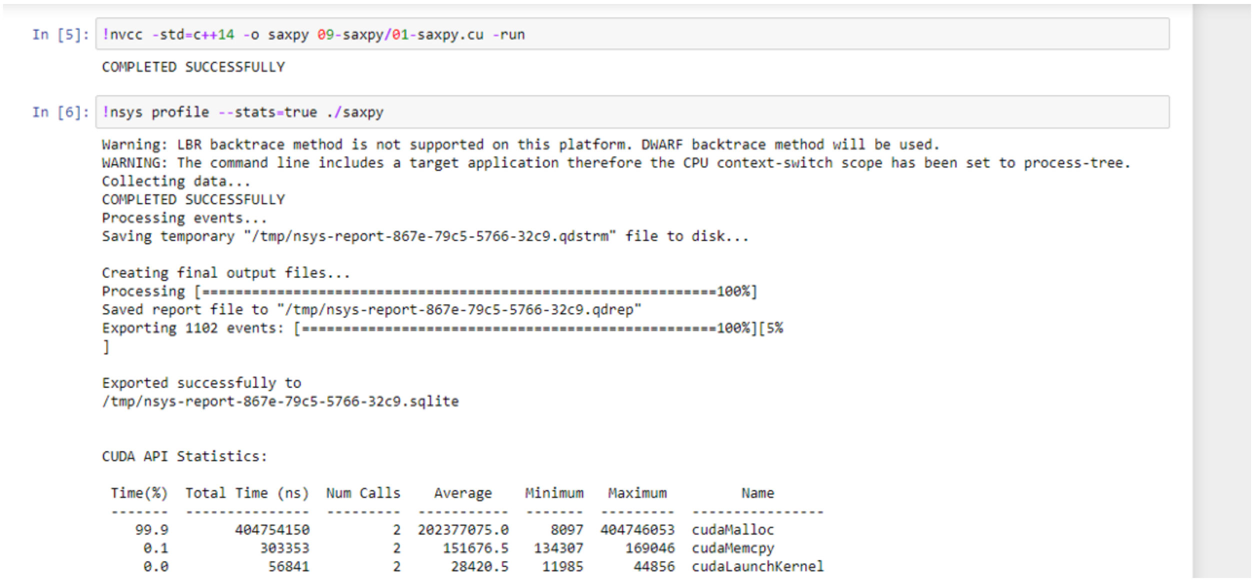
```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average    Minimum   Maximum         Name
 -------  ---------------  ---------  ----------  -------  ---------  ----------------
    99.9        259039596          3  86346532.0     5291  259027911  cudaMalloc
     0.1           129830          3     43276.7     5023     116197  cudaFree
     0.0            47834          3     15944.7     6067      23793  cudaMemcpy
     0.0            32437          1     32437.0    32437      32437  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum                   Name
 -------  ---------------  ---------  -------  -------  -------  ----------------------------------------
   100.0             4352          1   4352.0     4352     4352  matproductsharedmemory(int*, int*, int*)


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum      Operation
 -------  ---------------  ----------  -------  -------  -------  ------------------
    58.5             2528           2   1264.0     1056     1472  [CUDA memcpy HtoD]
    41.5             1792           1   1792.0     1792     1792  [CUDA memcpy DtoH]
```

Que 3:

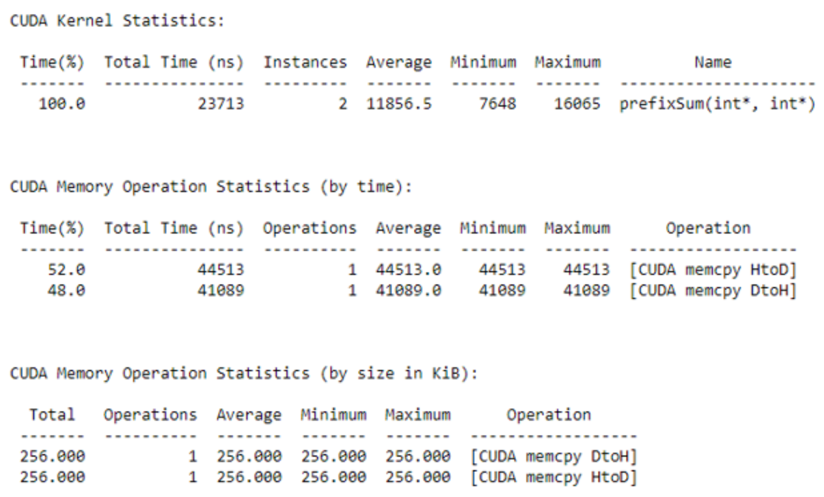
Implement Prefix sum using CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

```
// 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;
}
```

Output:



```
In [5]: !nvcc -std=c++14 -o saxpy 09-saxpy/01-saxpy.cu -run

        COMPLETED SUCCESSFULLY

In [6]: !nsys profile --stats=true ./saxpy

        Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
        WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-tree.
        Collecting data...
        COMPLETED SUCCESSFULLY
        Processing events...
        Saving temporary "/tmp/nsys-report-867e-79c5-5766-32c9.qdstrm" file to disk...

        Creating final output files...
        Processing [==============================================================100%]
        Saved report file to "/tmp/nsys-report-867e-79c5-5766-32c9.qdrep"
        Exporting 1102 events: [==================================================100%][5%
        ]

        Exported successfully to
        /tmp/nsys-report-867e-79c5-5766-32c9.sqlite


        CUDA API Statistics:

         Time(%)  Total Time (ns)  Num Calls    Average     Minimum   Maximum         Name
         -------  ---------------  ---------  -----------  -------  ---------  ----------------
            99.9        404754150          2  202377075.0     8097  404746053  cudaMalloc
             0.1           303353          2     151676.5   134307     169046  cudaMemcpy
             0.0            56841          2      28420.5    11985      44856  cudaLaunchKernel
```



```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum          Name
 -------  ---------------  ---------  -------  -------  -------  ---------------------
   100.0            23713          2  11856.5     7648    16065  prefixSum(int*, int*)


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum      Operation
 -------  ---------------  ----------  -------  -------  -------  ------------------
    52.0            44513           1  44513.0    44513    44513  [CUDA memcpy HtoD]
    48.0            41089           1  41089.0    41089    41089  [CUDA memcpy DtoH]


CUDA Memory Operation Statistics (by size in KiB):

  Total    Operations  Average  Minimum  Maximum      Operation
 -------   ----------  -------  -------  -------  ------------------
 256.000            1  256.000  256.000  256.000  [CUDA memcpy DtoH]
 256.000            1  256.000  256.000  256.000  [CUDA memcpy HtoD]
```

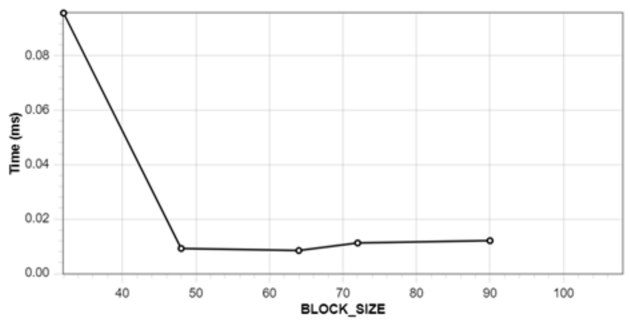
Que 4:

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.

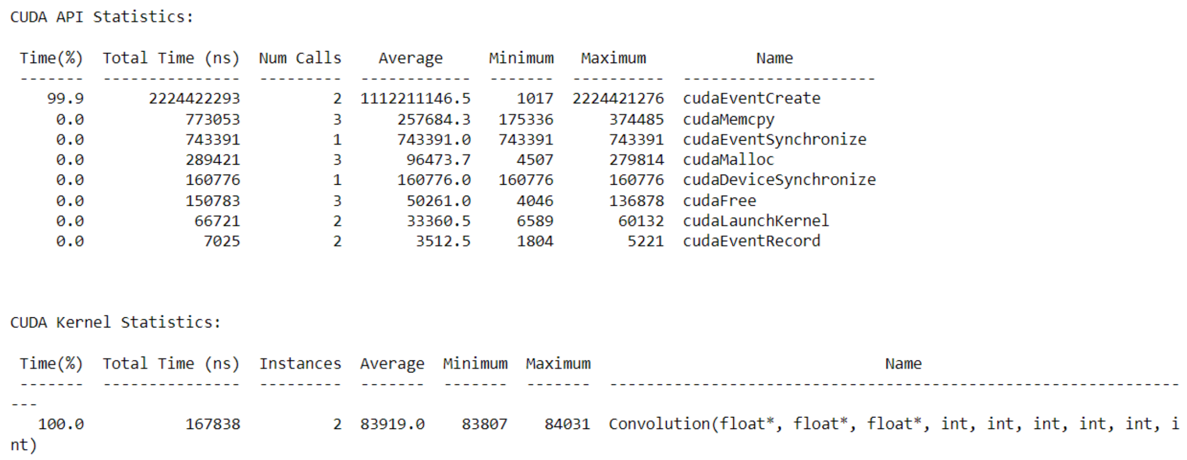
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <cstdlib>
#include <time.h>

#define BLOCK_SIZE 32
#define WA 512
#define HA 512
#define HC 3
#define WC 3
#define WB (WA - WC + 1)
#define HB (HA - HC + 1)

__global__ void Convolution(float* A, float* B, float* C, int numARows, int numACols, int numBRows, int numBCols, int numCRows, int numCCol
{
    int col = blockIdx.x * (BLOCK_SIZE - WC + 1) + threadIdx.x;
    int row = blockIdx.y * (BLOCK_SIZE - WC + 1) + threadIdx.y;
    int row_i = row - WC + 1;
    int col_i = col - WC + 1;

    float tmp = 0;

    __shared__ float shm[BLOCK_SIZE][BLOCK_SIZE];

    if (row_i < WA && row_i >= 0 && col_i < WA && col_i >= 0)
    {
        shm[threadIdx.y][threadIdx.x] = A[col_i * WA + row_i];
    }
    else
    {
        shm[threadIdx.y][threadIdx.x] = 0;
    }

    __syncthreads();

    if (threadIdx.y < (BLOCK_SIZE - WC + 1) && threadIdx.x < (BLOCK_SIZE - WC + 1) && row < (WB - WC + 1) && col < (WB - WC + 1))
    {
        for (int i = 0; i< WC;i++)
            for (int j = 0;j<WC;j++)
                tmp += shm[threadIdx.y + i][threadIdx.x + j] * C[j*WC + i];
        B[col*WB + row] = tmp;
    }
}

void randomInit(float* data, int size)
{
    for (int i = 0; i < size; ++i)
        data[i] = rand() / (float)RAND_MAX;
}

int main(int argc, char** argv)
{
    srand(2006);
    cudaError_t error;
    cudaEvent_t start_G, stop_G;

    cudaEventCreate(&start_G);
    cudaEventCreate(&stop_G);

    unsigned int size_A = WA * HA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float* h_A = (float*)malloc(mem_size_A);

    unsigned int size_B = WB * HB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float* h_B = (float*)malloc(mem_size_B);

    unsigned int size_C = WC * HC;
    unsigned int mem_size_C = sizeof(float) * size_C;
    float* h_C = (float*)malloc(mem_size_C);

    randomInit(h_A, size_A);
    randomInit(h_C, size_C);

    float* d_A;
    float* d_B;
    float* d_C;

    cudaMalloc((void**)&d_A, mem_size_A);

    cudaMalloc((void**)&d_B, mem_size_B);

    cudaMalloc((void**)&d_C, mem_size_C);

    cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

    cudaMemcpy(d_C, h_C, mem_size_C, cudaMemcpyHostToDevice);

    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid((WB - 1) / (BLOCK_SIZE - WC + 1), (WB - 1) / (BLOCK_SIZE - WC + 1));

    Convolution << < grid, threads >> >(d_A, d_B, d_C, HA, WA, HB, WB, HC, WC);

    cudaEventRecord(start_G);

    Convolution << < grid, threads >> >(d_A, d_B, d_C, HA, WA, HB, WB, HC, WC);

    cudaDeviceSynchronize();

    cudaEventRecord(stop_G);

    cudaEventSynchronize(stop_G);

    cudaMemcpy(h_B, d_B, mem_size_B, cudaMemcpyDeviceToHost);

    float miliseconds = 0;
    cudaEventElapsedTime(&miliseconds, start_G, stop_G);

    printf("Time took to compute matrix A of dimensions %d x %d  on GPU is %f ms \n \n \n", WA, HA, miliseconds);

//  for (int i = 0;i < HB;i++)
//  {
//      for (int j = 0;j < WB;j++)
//      {
//          printf("%f ", h_B[i*HB + j]);
//      }
//      printf("\n");
//  }

    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    return EXIT_SUCCESS;
}
```

Output :

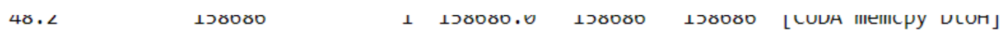
For different Values we get this trend:



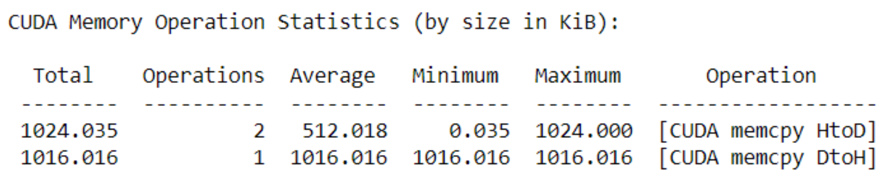
Statistics:



```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average     Minimum   Maximum            Name
 -------  ---------------  ---------  ------------  -------  ----------  ---------------------
    99.9       2224422293          2  1112211146.5     1017  2224421276  cudaEventCreate
     0.0           773053          3      257684.3   175336      374485  cudaMemcpy
     0.0           743391          1      743391.0   743391      743391  cudaEventSynchronize
     0.0           289421          3       96473.7     4507      279814  cudaMalloc
     0.0           160776          1      160776.0   160776      160776  cudaDeviceSynchronize
     0.0           150783          3       50261.0     4046      136878  cudaFree
     0.0            66721          2       33360.5     6589       60132  cudaLaunchKernel
     0.0             7025          2        3512.5     1804        5221  cudaEventRecord


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum                              Name
 -------  ---------------  ---------  -------  -------  -------  --------------------------------------------------------------
---
   100.0           167838          2  83919.0    83807    84031  Convolution(float*, float*, float*, int, int, int, int, int, i
nt)
```



```
     48.2           158686           1  158686.0   158686   158686  [CUDA memcpy DtoH]
```



```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average   Minimum   Maximum        Operation
 --------    ----------  --------  --------  --------  ------------------
 1024.035             2   512.018     0.035  1024.000  [CUDA memcpy HtoD]
 1016.016             1  1016.016  1016.016  1016.016  [CUDA memcpy DtoH]
```