**Practical 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.

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.

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.

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 "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* *numCCols*)

{

*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);

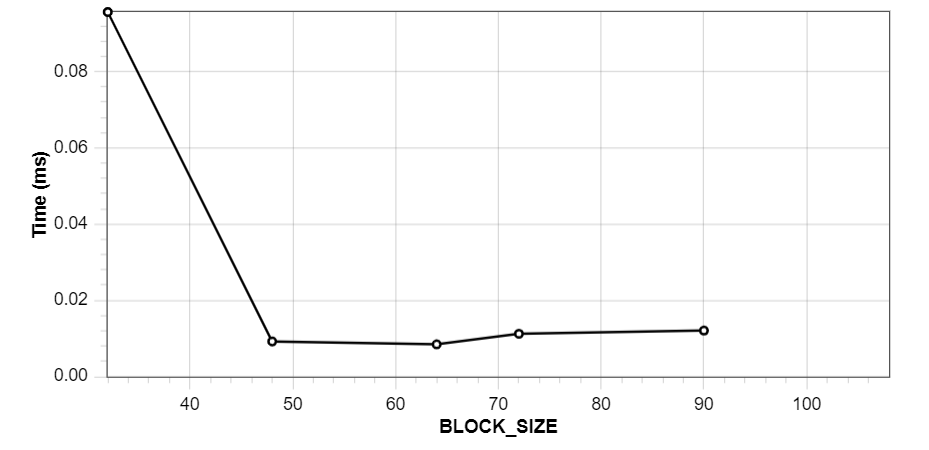
    cudaFree(d\_B);

    cudaFree(d\_C);

    return EXIT\_SUCCESS;

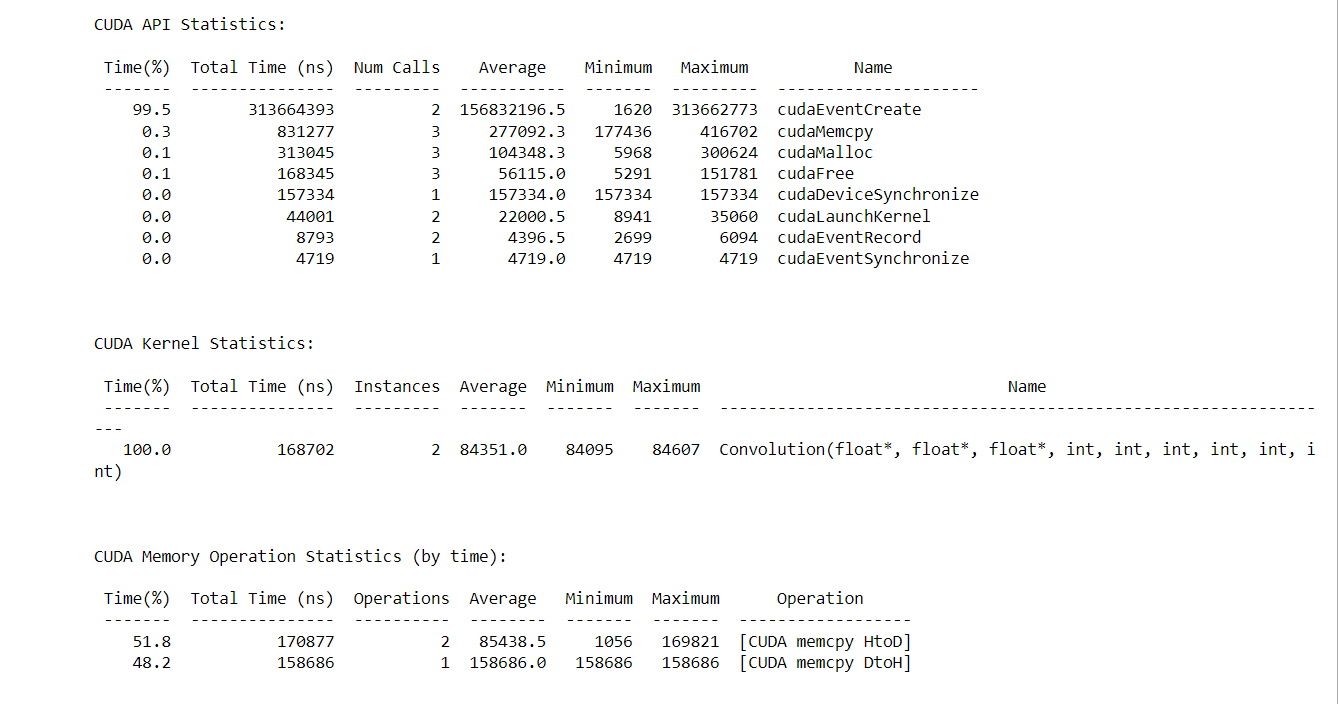
}

For different values of block sizes, we have following trend in time:



Hence, we observe that taking **BLOCK\_SIZE = 64** would be optimum.

**Profiling:**

****