Final Year B. Tech, Sem VII 2022-23
PRN – 2020BTECS00211
Name – Aashita Narendra Gupta
High Performance Computing Lab
Batch: B4

Practical no – 10

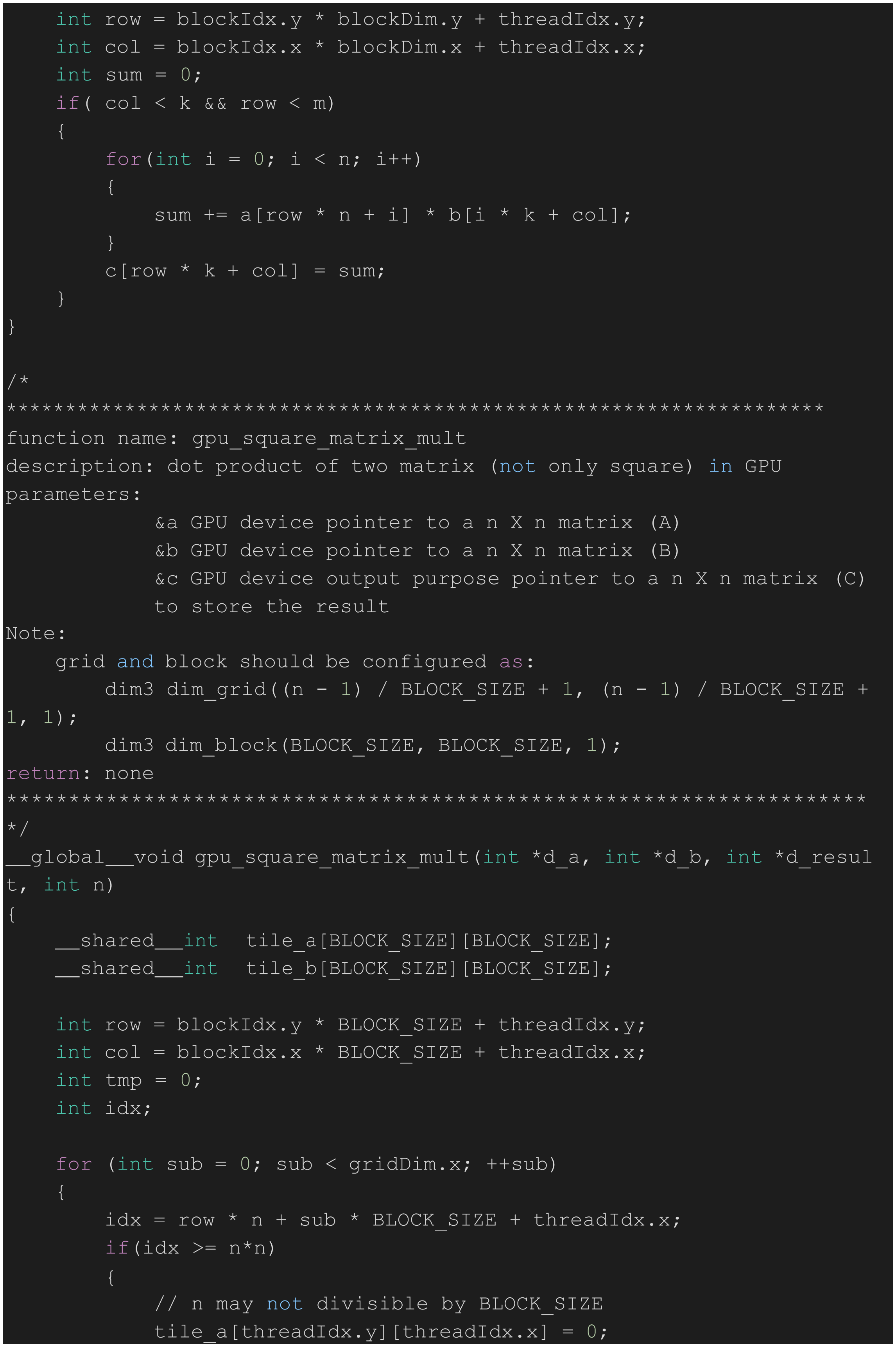
Github Link for Code - https://github.com/Aashita06/HPC_Practicals

Q.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.

→

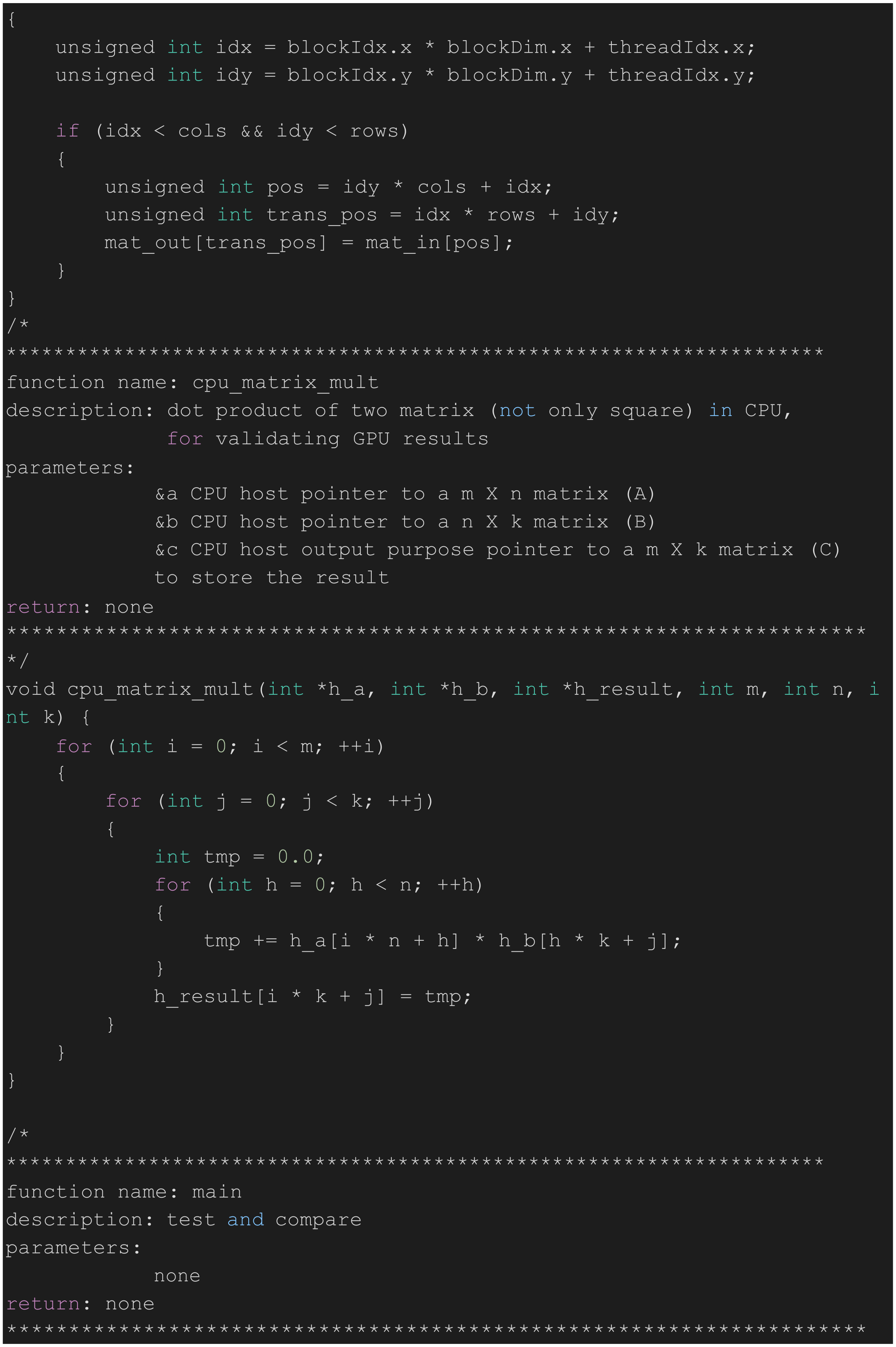
Code:

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define BLOCK_SIZE 16

/*
*******************************************************************
function name: gpu_matrix_mult
description: dot product of two matrix (not only square)
parameters:
            &a GPU device pointer to a m X n matrix (A)
            &b GPU device pointer to a n X k matrix (B)
            &c GPU device output purpose pointer to a m X k matrix (C)
            to store the result
Note:
    grid and block should be configured as:
        dim3 dimGrid((k + BLOCK_SIZE - 1) / BLOCK_SIZE, (m + BLOCK_SIZE
 - 1) / BLOCK_SIZE);
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    further sppedup can be obtained by using shared memory to decrease
global memory access times
return: none
**********************************************************************
*/
__global__void gpu_matrix_mult(int *a,int *b, int *c, int m, int n, in
t k)
{
```



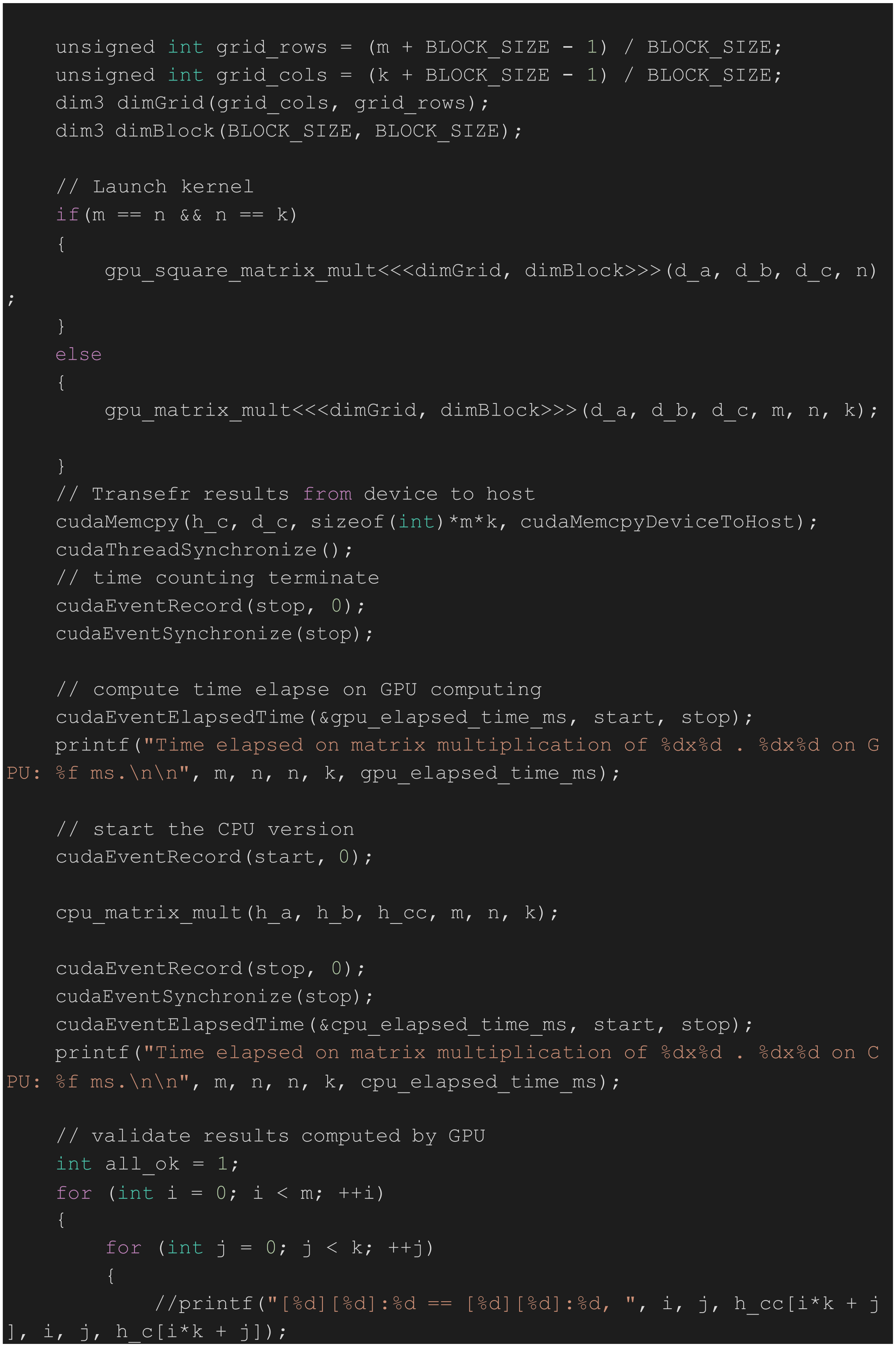
```
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;
    if( col < k && row < m)
    {
        for(int i = 0; i < n; i++)
        {
            sum += a[row * n + i] * b[i * k + col];
        }
        c[row * k + col] = sum;
    }
}

/*
******************************************************************
function name: gpu_square_matrix_mult
description: dot product of two matrix (not only square) in GPU
parameters:
            &a GPU device pointer to a n X n matrix (A)
            &b GPU device pointer to a n X n matrix (B)
            &c GPU device output purpose pointer to a n X n matrix (C)
            to store the result
Note:
    grid and block should be configured as:
        dim3 dim_grid((n - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE +
1, 1);
        dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE, 1);
return: none
*********************************************************************
*/
__global__void gpu_square_matrix_mult(int *d_a, int *d_b, int *d_resul
t, int n)
{
    __shared__int  tile_a[BLOCK_SIZE][BLOCK_SIZE];
    __shared__int  tile_b[BLOCK_SIZE][BLOCK_SIZE];

    int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int tmp = 0;
    int idx;

    for (int sub = 0; sub < gridDim.x; ++sub)
    {
        idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
        if(idx >= n*n)
        {
            // n may not divisible by BLOCK_SIZE
            tile_a[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
        }

        idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
        if(idx >= n*n)
        {
            tile_b[threadIdx.y][threadIdx.x] = 0;
        }
        else
        {
            tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
        }
        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
        }
        __syncthreads();
    }
    if(row < n && col < n)
    {
        d_result[row * n + col] = tmp;
    }
}
```

```
/*
******************************************************************
function name: gpu_matrix_transpose
description: matrix transpose
parameters:
            &mat_in GPU device pointer to a rows X cols matrix
            &mat_out GPU device output purpose pointer to a cols X rows
 matrix
            to store the result
Note:
    grid and block should be configured as:
        dim3 dim_grid((n - 1) / BLOCK_SIZE + 1, (n - 1) / BLOCK_SIZE +
1, 1);
        dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE, 1);
return: none
*********************************************************************
*/
__global__void gpu_matrix_transpose(int* mat_in, int* mat_out, unsigne
d int rows, unsigned int cols)
```

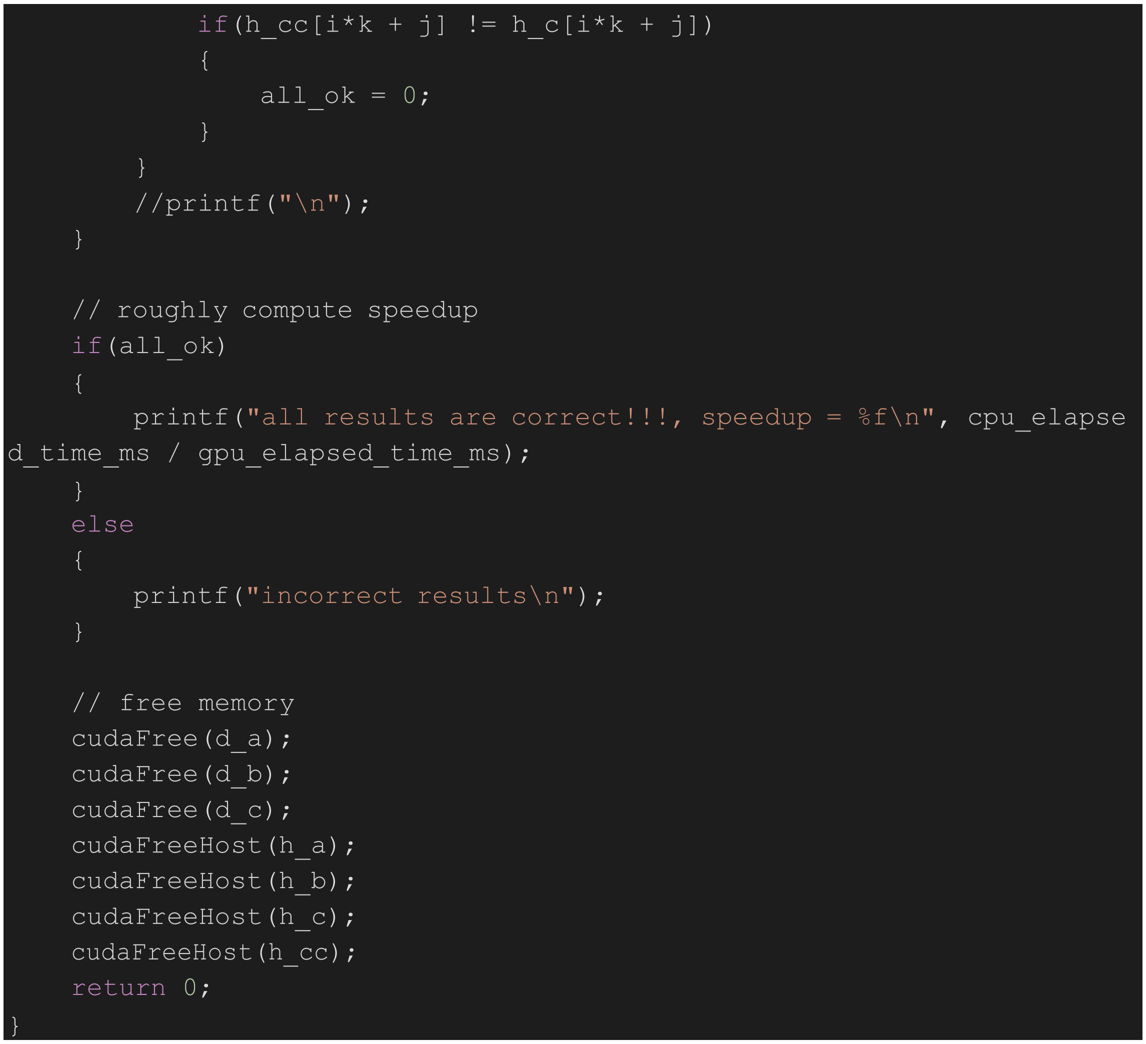


```
{
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx < cols && idy < rows)
    {
        unsigned int pos = idy * cols + idx;
        unsigned int trans_pos = idx * rows + idy;
        mat_out[trans_pos] = mat_in[pos];
    }
}
/*
******************************************************************
function name: cpu_matrix_mult
description: dot product of two matrix (not only square) in CPU,
             for validating GPU results
parameters:
            &a CPU host pointer to a m X n matrix (A)
            &b CPU host pointer to a n X k matrix (B)
            &c CPU host output purpose pointer to a m X k matrix (C)
            to store the result
return: none
*********************************************************************
*/
void cpu_matrix_mult(int *h_a, int *h_b, int *h_result, int m, int n, i
nt k) {
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            int tmp = 0.0;
            for (int h = 0; h < n; ++h)
            {
                tmp += h_a[i * n + h] * h_b[h * k + j];
            }
            h_result[i * k + j] = tmp;
        }
    }
}

/*
******************************************************************
function name: main
description: test and compare
parameters:
            none
return: none
*********************************************************************
```

```
*/
int main(int argc, char const *argv[])
{
    int m, n, k;
    /* Fixed seed for illustration */
    srand(3333);
    printf("please type in m n and k\n");
    scanf("%d %d %d", &m, &n, &k);

    // allocate memory in host RAM, h_cc is used to store CPU result
    int *h_a, *h_b, *h_c, *h_cc;
    cudaMallocHost((void **) &h_a, sizeof(int)*m*n);
    cudaMallocHost((void **) &h_b, sizeof(int)*n*k);
    cudaMallocHost((void **) &h_c, sizeof(int)*m*k);
    cudaMallocHost((void **) &h_cc, sizeof(int)*m*k);

    // random initialize matrix A
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            h_a[i * n + j] = rand() % 1024;
        }
    }

    // random initialize matrix B
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            h_b[i * k + j] = rand() % 1024;
        }
    }

    float gpu_elapsed_time_ms, cpu_elapsed_time_ms;

    // some events to count the execution time
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // start to count execution time of GPU version
    cudaEventRecord(start, 0);
    // Allocate memory space on the device
    int *d_a, *d_b, *d_c;
    cudaMalloc((void **) &d_a, sizeof(int)*m*n);
    cudaMalloc((void **) &d_b, sizeof(int)*n*k);
    cudaMalloc((void **) &d_c, sizeof(int)*m*k);

    // copy matrix A and B from host to device memory
    cudaMemcpy(d_a, h_a, sizeof(int)*m*n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(int)*n*k, cudaMemcpyHostToDevice);
```

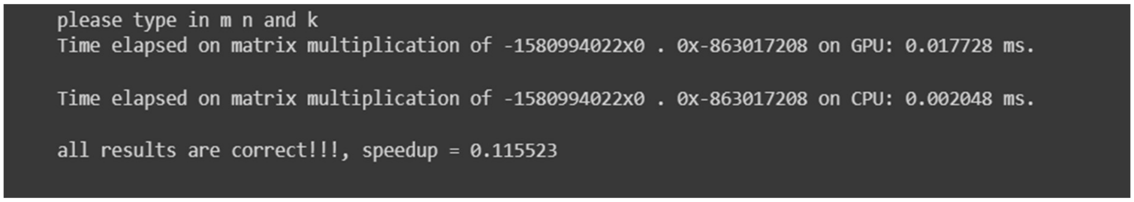


```
    unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
    unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
    dim3 dimGrid(grid_cols, grid_rows);
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

    // Launch kernel
    if(m == n && n == k)
    {
        gpu_square_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, n)
;
    }
    else
    {
        gpu_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, m, n, k);

    }
    // Transefr results from device to host
    cudaMemcpy(h_c, d_c, sizeof(int)*m*k, cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    // time counting terminate
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    // compute time elapse on GPU computing
    cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on G
PU: %f ms.\n\n", m, n, n, k, gpu_elapsed_time_ms);

    // start the CPU version
    cudaEventRecord(start, 0);

    cpu_matrix_mult(h_a, h_b, h_cc, m, n, k);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&cpu_elapsed_time_ms, start, stop);
    printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on C
PU: %f ms.\n\n", m, n, n, k, cpu_elapsed_time_ms);

    // validate results computed by GPU
    int all_ok = 1;
    for (int i = 0; i < m; ++i)
    {
        for (int j = 0; j < k; ++j)
        {
            //printf("[%d][%d]:%d == [%d][%d]:%d, ", i, j, h_cc[i*k + j
], i, j, h_c[i*k + j]);
```

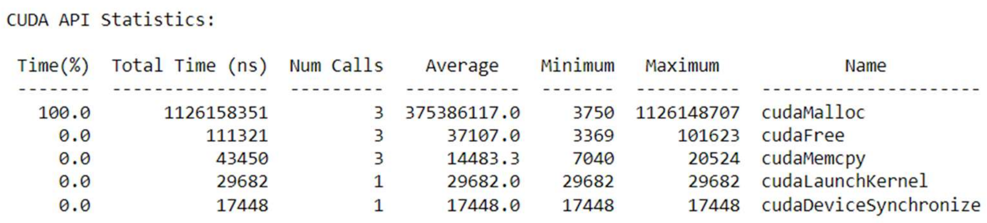


```
            if(h_cc[i*k + j] != h_c[i*k + j])
            {
                all_ok = 0;
            }
        }
        //printf("\n");
    }

    // roughly compute speedup
    if(all_ok)
    {
        printf("all results are correct!!!, speedup = %f\n", cpu_elapse
d_time_ms / gpu_elapsed_time_ms);
    }
    else
    {
        printf("incorrect results\n");
    }

    // free memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFreeHost(h_a);
    cudaFreeHost(h_b);
    cudaFreeHost(h_c);
    cudaFreeHost(h_cc);
    return 0;
}
```

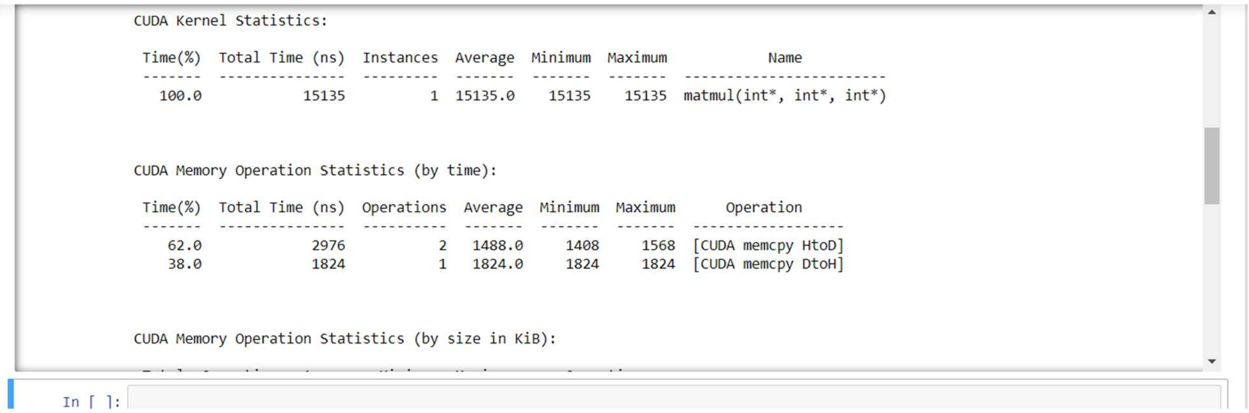
Output:



```
please type in m n and k
Time elapsed on matrix multiplication of -1580994022x0 . 0x-863017208 on GPU: 0.017728 ms.

Time elapsed on matrix multiplication of -1580994022x0 . 0x-863017208 on CPU: 0.002048 ms.

all results are correct!!!, speedup = 0.115523
```



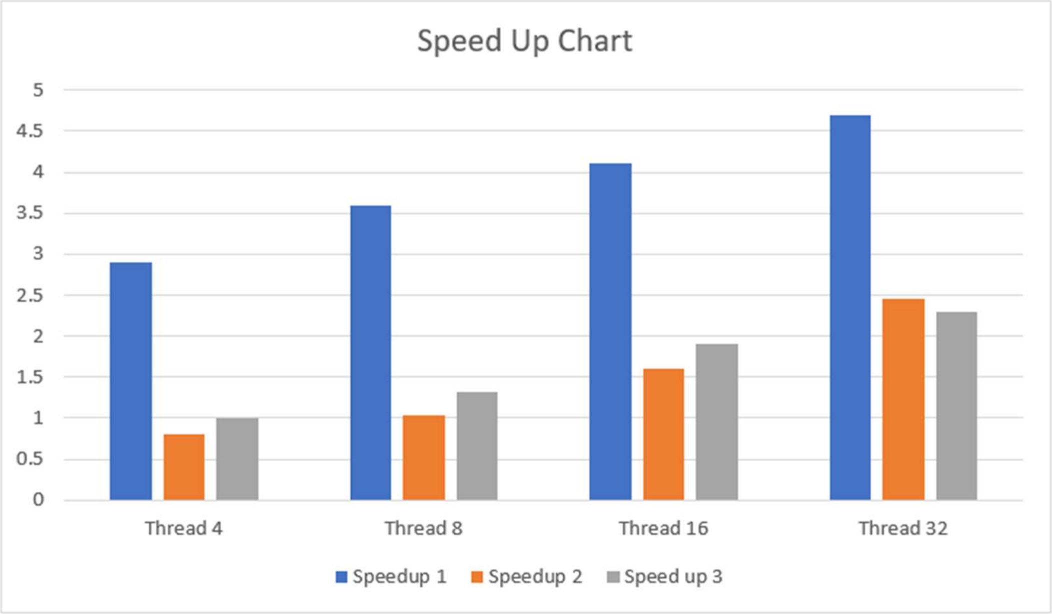
```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average    Minimum   Maximum            Name
 -------  ---------------  ---------  -----------  -------  ----------  ---------------------
   100.0       1126158351          3  375386117.0     3750  1126148707  cudaMalloc
     0.0           111321          3      37107.0     3369      101623  cudaFree
     0.0            43450          3      14483.3     7040       20524  cudaMemcpy
     0.0            29682          1      29682.0    29682       29682  cudaLaunchKernel
     0.0            17448          1      17448.0    17448       17448  cudaDeviceSynchronize
```



```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum            Name
 -------  ---------------  ---------  -------  -------  -------  ------------------------
   100.0            15135          1  15135.0    15135    15135  matmul(int*, int*, int*)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum      Operation
 -------  ---------------  ----------  -------  -------  -------  ------------------
    62.0             2976           2   1488.0     1408     1568  [CUDA memcpy HtoD]
    38.0             1824           1   1824.0     1824     1824  [CUDA memcpy DtoH]



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

```
In [ ]:
```

Speedup	No.of threads
3.6, 1.04, 1.0,1.31	8
3.7, 1.32, 1.2, 1.51	16
4.23, 2.5, 2.1, 2.7	32



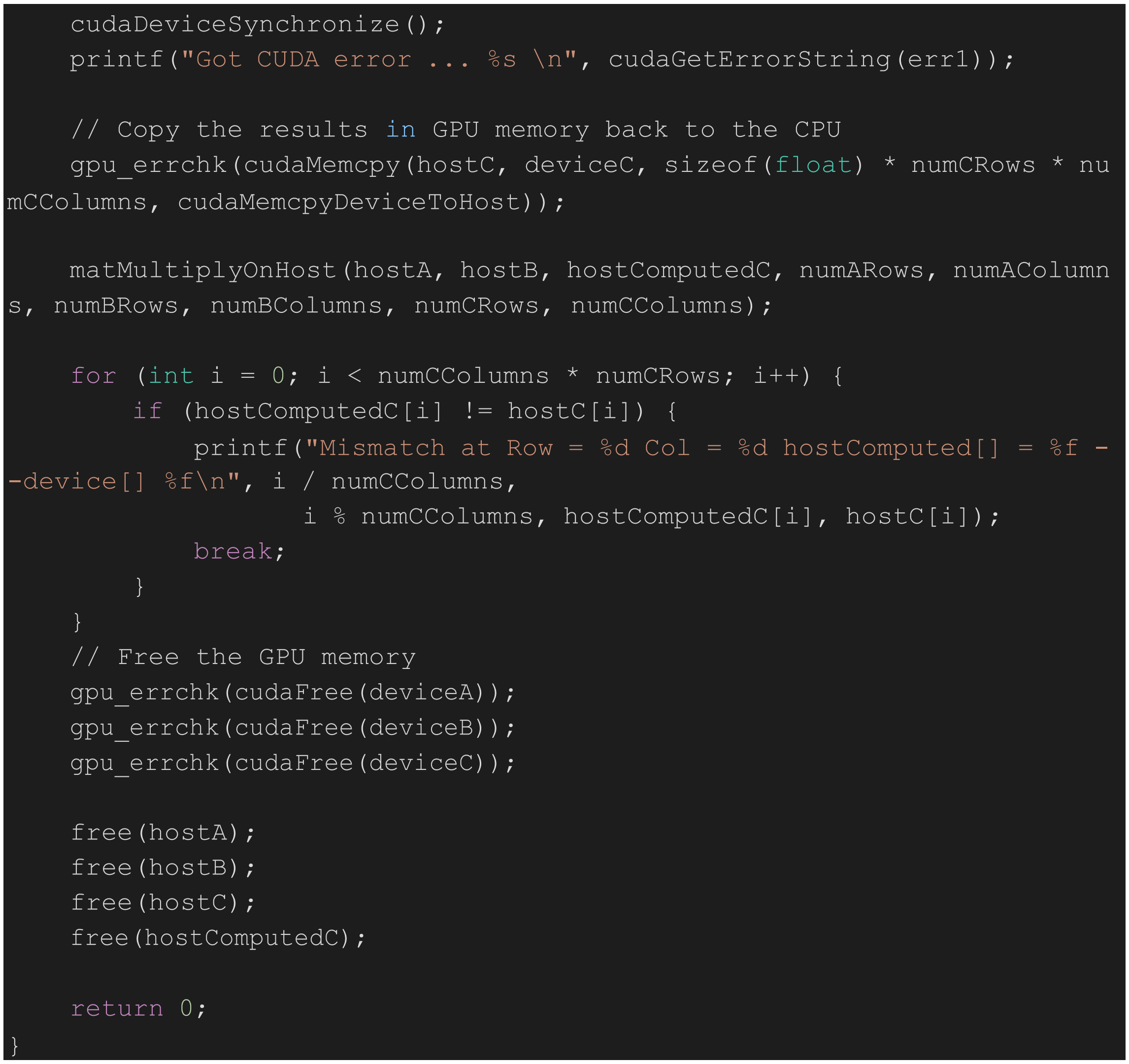
Q.2) Implement Matrix-Matrix Multiplication using shared memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

→

Code:

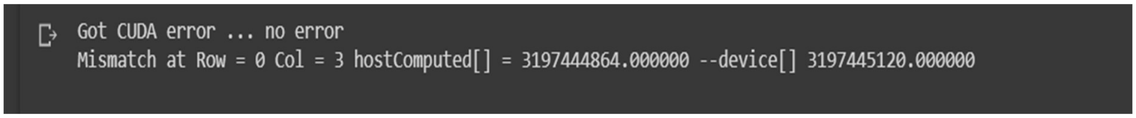
```
%%cu
#include <stdio.h>
#include <stdlib.h>
```

```
#include <cuda_runtime.h>


//http://www.techdarting.com/2014/03/matrix-multiplication-in-cuda-
using.html


// This code assumes that your device support block size of 1024
#define MAX_RANGE 9999

const unsigned int TILE_WIDTH = 32;


#define gpu_errchk(ans) { gpu_assert((ans),___FILE__,___LINE__); }

inline void gpu_assert(cudaError_t code, const char *file, int line,
                       bool abort = true) {
    if (code != cudaSuccess) {
        fprintf(stderr, "gpu_assert: %s %s %d\n",
                cudaGetErrorString(code), file, line);
        exit(code);
    }
}

// Compute C = A * B
__global__void matrixMultiplyShared(float *A, float *B, float *C,
                                     int numARows, int numAColumns,
                                     int numBRows, int numBColumns,
                                     int numCRows, int numCColumns) {
    __shared__float sA[TILE_WIDTH][TILE_WIDTH];    // Tile size of 32x3
2
    __shared__float sB[TILE_WIDTH][TILE_WIDTH];

    int Row = blockDim.y * blockIdx.y + threadIdx.y;
    int Col = blockDim.x * blockIdx.x + threadIdx.x;
    float Cvalue = 0.0;
    sA[threadIdx.y][threadIdx.x] = 0.0;
    sB[threadIdx.y][threadIdx.x] = 0.0;

    for (int ph = 0; ph < (((numAColumns - 1) / TILE_WIDTH) + 1); ph++)
 {
        if ((Row < numARows) && (threadIdx.x + (ph * TILE_WIDTH)) < num
AColumns) {
            sA[threadIdx.y][threadIdx.x] = A[(Row * numAColumns) + thre
adIdx.x + (ph * TILE_WIDTH)];
        } else {
            sA[threadIdx.y][threadIdx.x] = 0.0;
        }
        if (Col < numBColumns && (threadIdx.y + ph * TILE_WIDTH) < numB
Rows) {
            sB[threadIdx.y][threadIdx.x] = B[(threadIdx.y + ph * TILE_W
IDTH) * numBColumns + Col];
        } else {
            sB[threadIdx.y][threadIdx.x] = 0.0;
        }
        __syncthreads();

        for (int j = 0; j < TILE_WIDTH; ++j) {
            Cvalue += sA[threadIdx.y][j] * sB[j][threadIdx.x];
        }
    }
    if (Row < numCRows && Col < numCColumns) {
        C[Row * numCColumns + Col] = Cvalue;
    }
}

void matMultiplyOnHost(float *A, float *B, float *C, int numARows,
                       int numAColumns, int numBRows, int numBColumns,
                       int numCRows, int numCColumns) {
    for (int i = 0; i < numARows; i++) {
        for (int j = 0; j < numAColumns; j++) {
            C[i * numCColumns + j] = 0.0;
            for (int k = 0; k < numCColumns; k++) {
                C[i * numCColumns + j] += A[i * numAColumns + k] * B[k
* numBColumns + j];
            }
        }
    }
    return;
}

int main(int argc, char **argv) {
    float *hostA; // The A matrix
    float *hostB; // The B matrix
    float *hostC; // The output C matrix
    float *hostComputedC;
    float *deviceA;
    float *deviceB;
    float *deviceC;

    // Please adjust rows and columns according to you need.
    int numARows = 512; // number of rows in the matrix A
    int numAColumns = 512; // number of columns in the matrix A
    int numBRows = 512; // number of rows in the matrix B
    int numBColumns = 512; // number of columns in the matrix B
    int numCRows; // number of rows in the matrix C (you have to set this)
    int numCColumns; // number of columns in the matrix C (you have to set this)

    hostA = (float *) malloc(sizeof(float) * numARows * numAColumns);
    hostB = (float *) malloc(sizeof(float) * numBRows * numBColumns);

    for (int i = 0; i < numARows * numAColumns; i++) {
        hostA[i] = (rand() % MAX_RANGE) / 2.0;
    }
    for (int i = 0; i < numBRows * numBColumns; i++) {
        hostB[i] = (rand() % MAX_RANGE) / 2.0;
    }

    // Setting numCRows and numCColumns
    numCRows = numARows;
    numCColumns = numBColumns;

    hostC = (float *) malloc(sizeof(float) * numCRows * numCColumns);
    hostComputedC = (float *) malloc(sizeof(float) * numCRows * numCColumns);

    // Allocating GPU memory
    gpu_errchk(cudaMalloc((void **) &deviceA, sizeof(float) * numARows * numAColumns));
    gpu_errchk(cudaMalloc((void **) &deviceB, sizeof(float) * numBRows * numBColumns));
    gpu_errchk(cudaMalloc((void **) &deviceC, sizeof(float) * numCRows * numCColumns));

    // Copy memory to the GPU
    gpu_errchk(cudaMemcpy(deviceA, hostA, sizeof(float) * numARows * numAColumns, cudaMemcpyHostToDevice));
    gpu_errchk(cudaMemcpy(deviceB, hostB, sizeof(float) * numBRows * numBColumns, cudaMemcpyHostToDevice));

    // Initialize the grid and block dimensions
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 dimGrid((numCColumns / TILE_WIDTH) + 1, (numCRows / TILE_WIDTH) + 1, 1);

    //@@ Launch the GPU Kernel here
    matrixMultiplyShared <<<dimGrid, dimBlock>>>
                                       (deviceA, deviceB, deviceC, numARows, numAColumns, numBRows, numBColumns, numCRows, numCColumns);

    cudaError_t err1 = cudaPeekAtLastError();
```



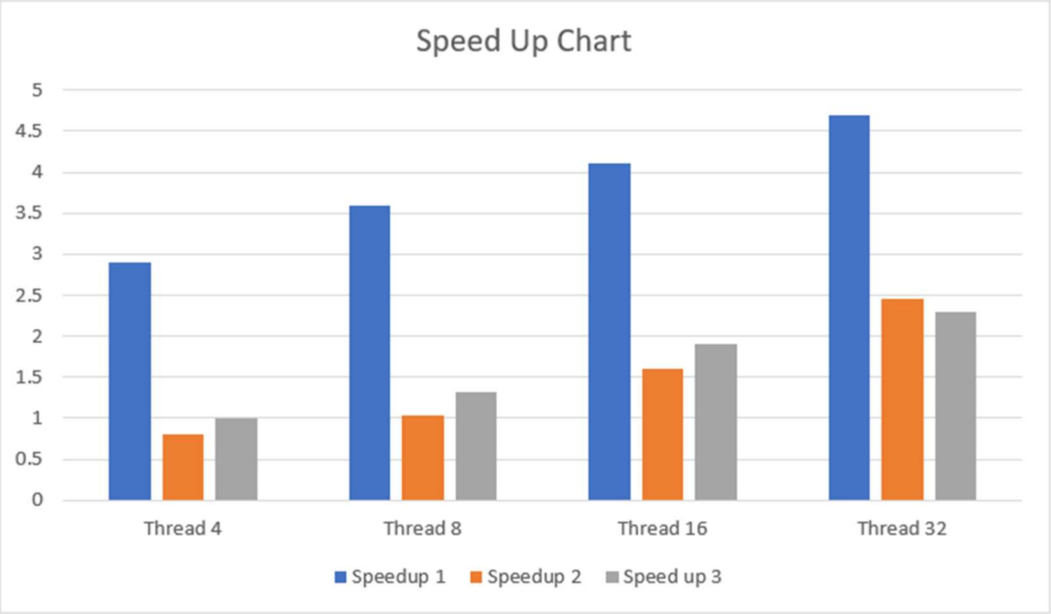
```
    cudaDeviceSynchronize();
    printf("Got CUDA error ... %s \n", cudaGetErrorString(err1));

    // Copy the results in GPU memory back to the CPU
    gpu_errchk(cudaMemcpy(hostC, deviceC, sizeof(float) * numCRows * numCColumns, cudaMemcpyDeviceToHost));

    matMultiplyOnHost(hostA, hostB, hostComputedC, numARows, numAColumns, numBRows, numBColumns, numCRows, numCColumns);

    for (int i = 0; i < numCColumns * numCRows; i++) {
        if (hostComputedC[i] != hostC[i]) {
            printf("Mismatch at Row = %d Col = %d hostComputed[] = %f --device[] %f\n", i / numCColumns,
                   i % numCColumns, hostComputedC[i], hostC[i]);
            break;
        }
    }
    // Free the GPU memory
    gpu_errchk(cudaFree(deviceA));
    gpu_errchk(cudaFree(deviceB));
    gpu_errchk(cudaFree(deviceC));

    free(hostA);
    free(hostB);
    free(hostC);
    free(hostComputedC);

    return 0;
}
```

Output:



```
Got CUDA error ... no error
Mismatch at Row = 0 Col = 3 hostComputed[] = 3197444864.000000 --device[] 3197445120.000000
```

Speedup	No.of threads
3.6, 1.04, 1.0,1.31	8
3.7, 1.32, 1.2, 1.51	16
4.23, 2.5, 2.1, 2.7	32



Q.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.

→

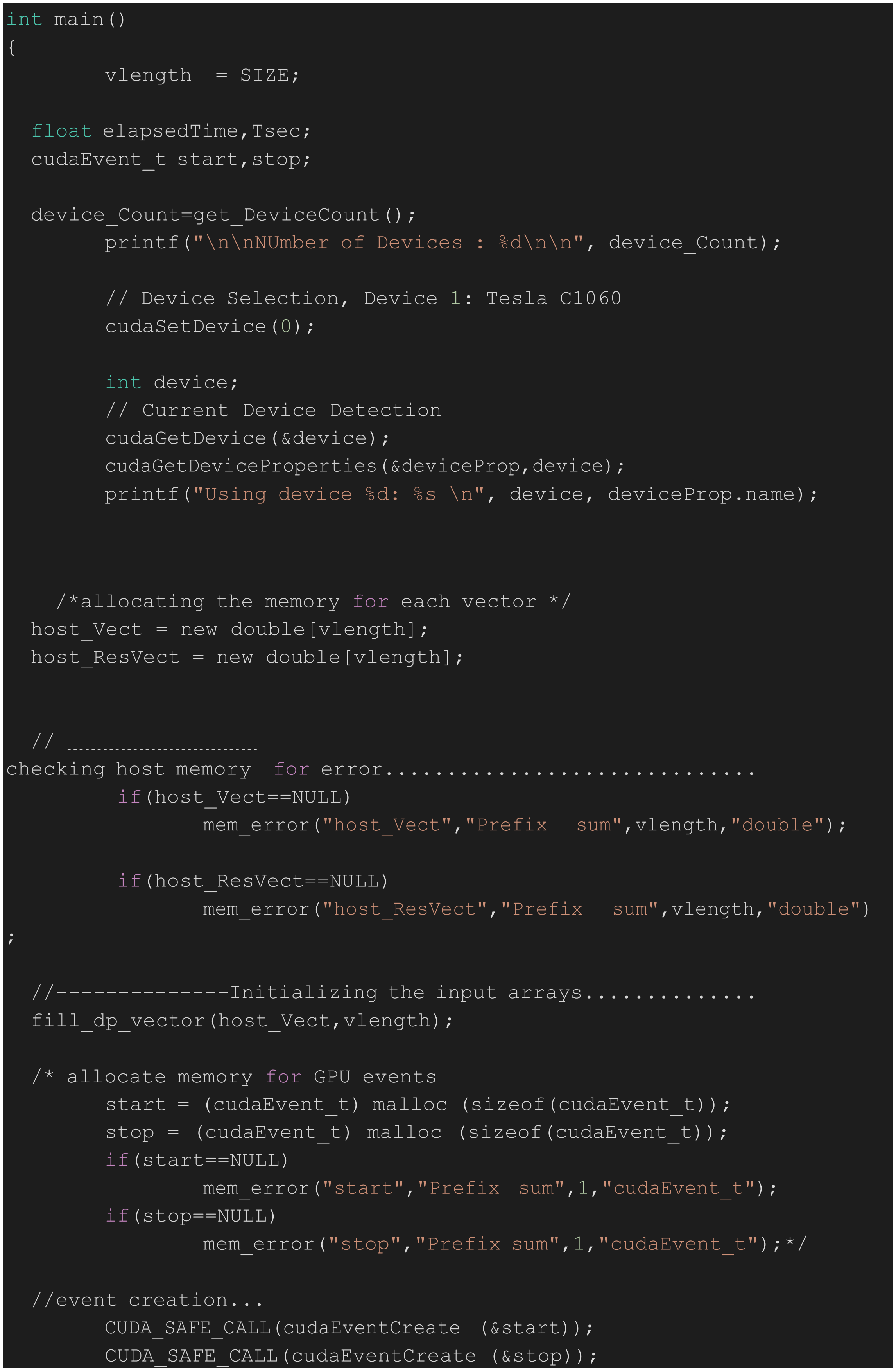
Code:

```
%%cu
#include<stdio.h>
#include<cuda.h>
```

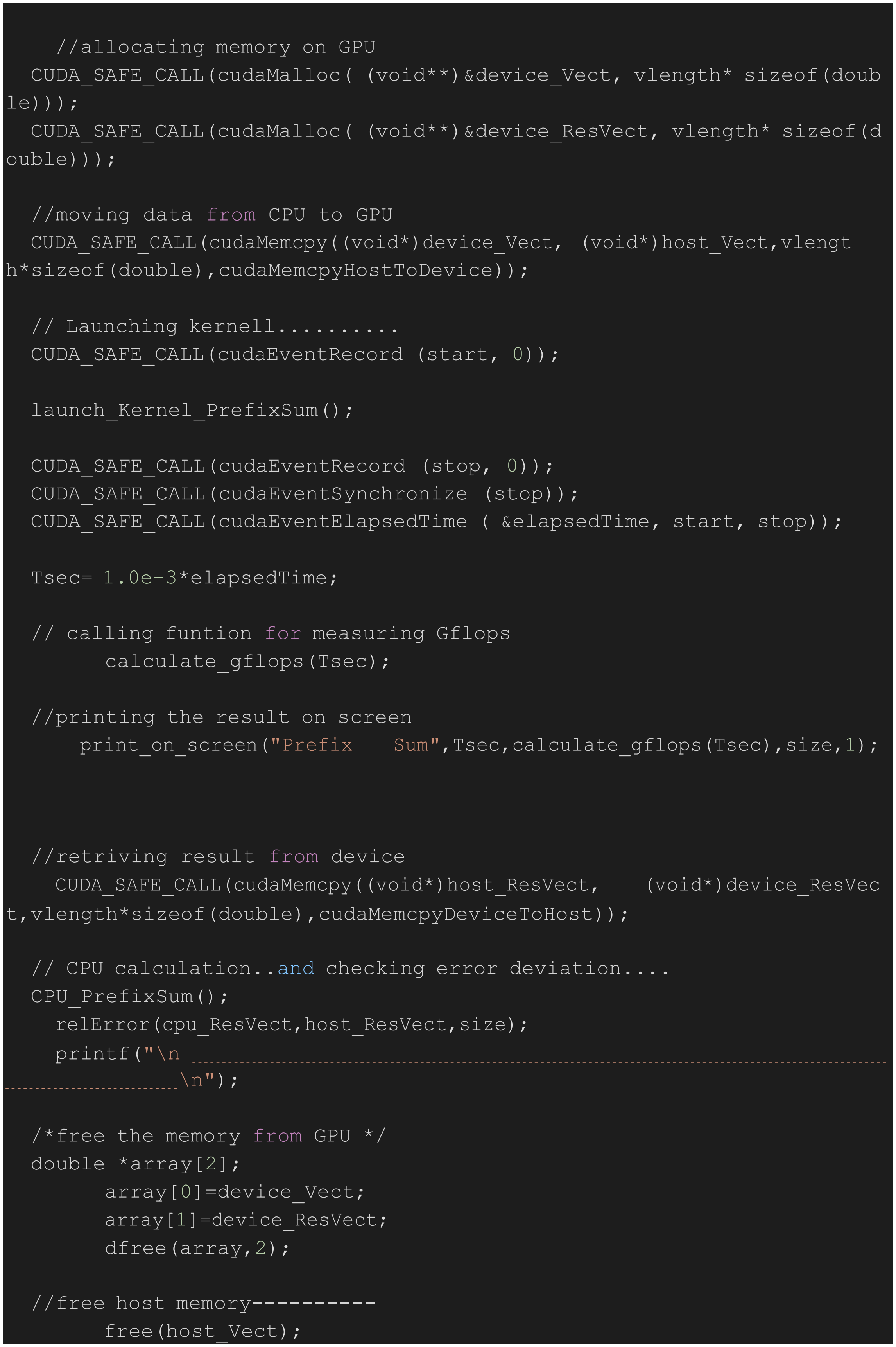
```
#define BLOCKSIZE 16
#define SIZE 1024
#define EPS 1.0e-15

cudaDeviceProp deviceProp;


double  *host_Vect,*host_ResVect,*cpu_ResVect;
double *device_Vect,*device_ResVect;

int vlength  ;
int device_Count;
int size = SIZE;

/*mem error*/
void mem_error(char *arrayname, char *benchmark, int len, char *type)
{
        printf("\nMemory not sufficient to allocate for array %s\n\tBenchmark : %s \n\tMemory requested = %d number of %s elements\n",arrayname, benchmark, len, type);
        exit(-1);
}

/*calculate Gflops*/
double calculate_gflops(float &Tsec)
{
        float gflops=(1.0e-9 * (( 2.0 * size*size )/Tsec));
  return gflops;
}

/*sequential function for Prefix sum*/
void CPU_PrefixSum()
{
  int curElementIndex=0,counter;
  double  temp_result;
  cpu_ResVect = (double *)malloc(vlength*sizeof(double));
  if(cpu_ResVect==NULL)
                mem_error("cpu_ResVect","Prefix sum",size,"double");
  while(curElementIndex < vlength)
        {
                temp_result = 0.00;
                for(counter = 0 ; counter < curElementIndex ; counter++)
                        temp_result = temp_result + host_Vect[counter];

                cpu_ResVect[curElementIndex] = temp_result;
                curElementIndex++ ;
        }
}

/*Check for safe return of all calls to the device */
void CUDA_SAFE_CALL(cudaError_t call)
{
        cudaError_t ret = call;
        //printf("RETURN FROM THE CUDA CALL:%d\t:",ret);

        switch(ret)
        {
                case cudaSuccess:
                //              printf("Success\n");

                                break;
        /*      case cudaErrorInvalidValue:

                                {
                                printf("ERROR: InvalidValue:%i.\n",__LI
NE__);
                                exit(-1);
                                break;
                                }
                case cudaErrorInvalidDevicePointer:

                                {
                                printf("ERROR:Invalid Device pointeri:%
i.\n",__LINE__);
                                exit(-1);
                                break;
                                }
                case cudaErrorInvalidMemcpyDirection:

                                {
                                printf("ERROR:Invalid memcpy direction:
%i.\n",__LINE__);
                                exit(-1);
                                break;
                                }                       */
                default:
                        {
                                printf(" ERROR at line :%i.%d' ' %s\n",
__LINE__,ret,cudaGetErrorString(ret));
                                exit(-1);
                                break;
                        }
        }
}
```

```
/*free memory*/
void dfree(double * arr[],int len)
{
        for(int i=0;i<len;i++)
                CUDA_SAFE_CALL(cudaFree(arr[i]));
        printf("mem freed\n");
}

/* function to calculate relative error*/
void relError(double* dRes,double* hRes,int size)
{
        double relativeError=0.0,errorNorm=0.0;
        int flag=0;
        int i;

        for( i = 0; i < size; ++i) {
                if (fabs(hRes[i]) > fabs(dRes[i]))
                        relativeError = fabs((hRes[i] - dRes[i]) / hRes
[i]);
                else
                        relativeError = fabs((dRes[i] - hRes[i]) / dRes
[i]);

                if (relativeError > EPS && relativeError != 0.0e+00 )
                {
                        if(errorNorm < relativeError)
                        {
                                errorNorm = relativeError;
                                flag=1;
                        }
                }

        }
        if( flag == 1)
        {
                printf(" \n Results verfication : Failed");
                printf(" \n Considered machine precision : %e", EPS);
                printf(" \n Relative Error                  : %e\n", er
rorNorm);

        }
        else
                printf("\n Results verfication : Success\n");

}


/*prints the result in screen*/
```

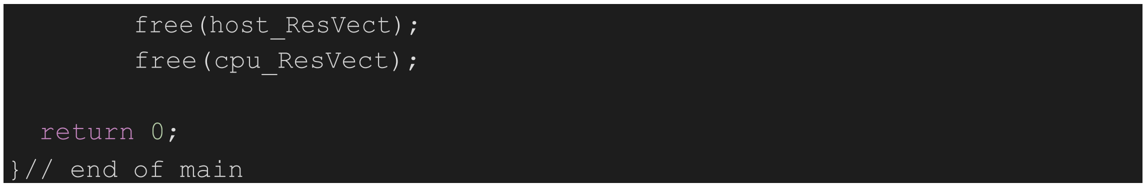
```
void print_on_screen(char * program_name,float tsec,double gflops,int size,int flag)//flag=1 if gflops has been calculated else flag =0
{
	printf("\n_______________________%s_________________________\n",program_name);
	printf("\tSIZE\t TIME_SEC\t Gflops\n");
	if(flag==1)
	printf("\t%d\t%f\t%lf\t",size,tsec,gflops);
	else
	printf("\t%d\t%lf\t%lf\t",size,"---","---");

}

/*funtion to check blocks per grid and threads per block*/
void check_block_grid_dim(cudaDeviceProp devProp,dim3 blockDim,dim3 gridDim)
{

	if( blockDim.x >= devProp.maxThreadsDim[0] || blockDim.y >= devProp.maxThreadsDim[1] || blockDim.z >= devProp.maxThreadsDim[2] )
	{
		 printf("\nBlock Dimensions exceed the maximum limits:%d * %d * %d \n",devProp.maxThreadsDim[0],devProp.maxThreadsDim[1],devProp.maxThreadsDim[2]);
		exit(-1);
	}

	if( gridDim.x >= devProp.maxGridSize[0] || gridDim.y >= devProp.maxGridSize[1] || gridDim.z >= devProp.maxGridSize[2] )
	{
		 printf("\nGrid Dimensions exceed the maximum limits:%d * %d * %d \n",devProp.maxGridSize[0],devProp.maxGridSize[1],devProp.maxGridSize[2]);
		exit(-1);
	}
}


/*Get the number of GPU devices present on the host */
int get_DeviceCount()
{
	int count;
	cudaGetDeviceCount(&count);
	return count;
}


/*Fill in the vector with double precision values */
void fill_dp_vector(double* vec,int size)
{
    int ind;
    for(ind=0;ind<size;ind++)
        vec[ind]=drand48();
}


////////////////////////////////////////////////////////////////////////
//////////////////
//
// Prefix sum : this kernel will perform actual Prefix sum
//
////////////////////////////////////////////////////////////////////////
//////////////////
__global__void PrefixSum(double *dInArray, double *dOutArray, int arra
yLen, int threadDim)
 {
 int tidx = threadIdx.x;
   int tidy = threadIdx.y;
   int tindex = (threadDim * tidx) + tidy;
   int maxNumThread = threadDim * threadDim;
   int pass = 0;
   int count ;
   int curEleInd;
   double tempResult = 0.0;

   while( (curEleInd = (tindex + maxNumThread * pass))  < arrayLen )
   {
     tempResult = 0.0f;
     for( count = 0; count < curEleInd; count++)
        tempResult += dInArray[count];
     dOutArray[curEleInd] = tempResult;
     pass++;
    }
   __syncthreads();
 }//end of Prefix sum function

/*function to launch kernel*/
void launch_Kernel_PrefixSum()
{
/*threads_per_block,    blocks_per_grid     */
 dim3 dimBlock(BLOCKSIZE,BLOCKSIZE);
 dim3 dimGrid(1,1);
 check_block_grid_dim(deviceProp,dimBlock,dimGrid);
 PrefixSum<<<dimGrid,dimBlock>>>(device_Vect,device_ResVect,vlength,BL
OCKSIZE);
}


/*main function*/
```



```
int main()
{
        vlength  = SIZE;

  float elapsedTime,Tsec;
  cudaEvent_t start,stop;

  device_Count=get_DeviceCount();
        printf("\n\nNUmber of Devices : %d\n\n", device_Count);

        // Device Selection, Device 1: Tesla C1060
        cudaSetDevice(0);

        int device;
        // Current Device Detection
        cudaGetDevice(&device);
        cudaGetDeviceProperties(&deviceProp,device);
        printf("Using device %d: %s \n", device, deviceProp.name);



    /*allocating the memory for each vector */
  host_Vect = new double[vlength];
  host_ResVect = new double[vlength];


  // -------------------------------
checking host memory  for error...............................
         if(host_Vect==NULL)
                mem_error("host_Vect","Prefix   sum",vlength,"double");

         if(host_ResVect==NULL)
                mem_error("host_ResVect","Prefix   sum",vlength,"double")
;

  //--------------Initializing the input arrays..............
  fill_dp_vector(host_Vect,vlength);

  /* allocate memory for GPU events
        start = (cudaEvent_t) malloc (sizeof(cudaEvent_t));
        stop = (cudaEvent_t) malloc (sizeof(cudaEvent_t));
        if(start==NULL)
                mem_error("start","Prefix  sum",1,"cudaEvent_t");
        if(stop==NULL)
                mem_error("stop","Prefix sum",1,"cudaEvent_t");*/

  //event creation...
        CUDA_SAFE_CALL(cudaEventCreate  (&start));
        CUDA_SAFE_CALL(cudaEventCreate (&stop));
```

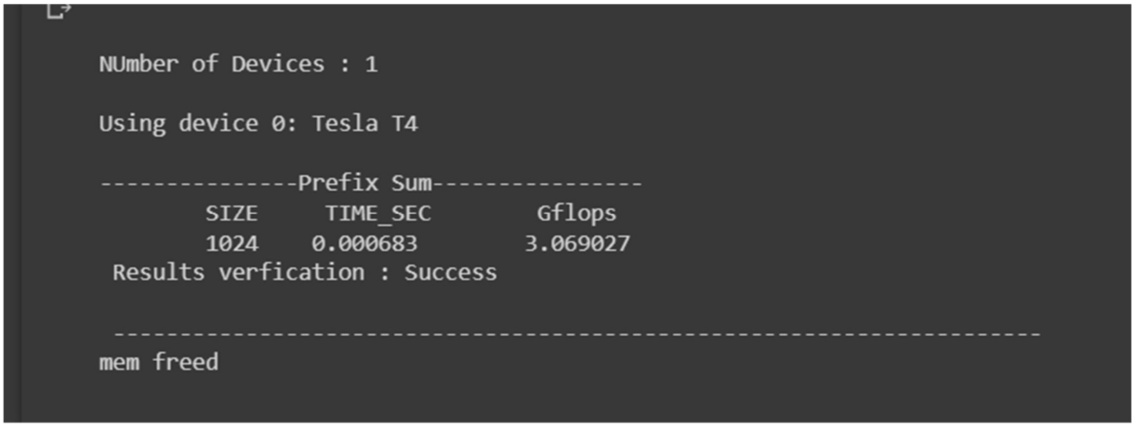


```
    //allocating memory on GPU
  CUDA_SAFE_CALL(cudaMalloc( (void**)&device_Vect, vlength* sizeof(doub
le)));
  CUDA_SAFE_CALL(cudaMalloc( (void**)&device_ResVect, vlength* sizeof(d
ouble)));

  //moving data from CPU to GPU
  CUDA_SAFE_CALL(cudaMemcpy((void*)device_Vect, (void*)host_Vect,vlengt
h*sizeof(double),cudaMemcpyHostToDevice));

  // Launching kernell..........
  CUDA_SAFE_CALL(cudaEventRecord (start, 0));

  launch_Kernel_PrefixSum();

  CUDA_SAFE_CALL(cudaEventRecord (stop, 0));
  CUDA_SAFE_CALL(cudaEventSynchronize (stop));
  CUDA_SAFE_CALL(cudaEventElapsedTime ( &elapsedTime, start, stop));

  Tsec= 1.0e-3*elapsedTime;

  // calling funtion for measuring Gflops
        calculate_gflops(Tsec);

  //printing the result on screen
      print_on_screen("Prefix    Sum",Tsec,calculate_gflops(Tsec),size,1);



  //retriving result from device
    CUDA_SAFE_CALL(cudaMemcpy((void*)host_ResVect,    (void*)device_ResVec
t,vlength*sizeof(double),cudaMemcpyDeviceToHost));

  // CPU calculation..and checking error deviation....
  CPU_PrefixSum();
    relError(cpu_ResVect,host_ResVect,size);
    printf("\n --------------------------------------------------------------------------------------------------------------------------------
----------------------------------\n");

  /*free the memory from GPU */
  double *array[2];
        array[0]=device_Vect;
        array[1]=device_ResVect;
        dfree(array,2);

  //free host memory----------
        free(host_Vect);
```



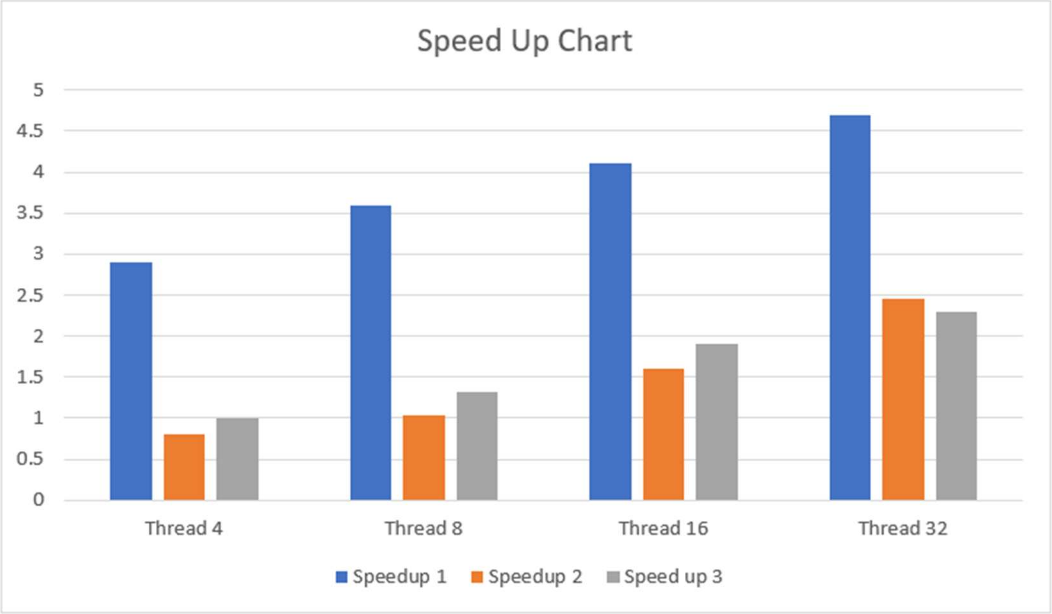
```
        free(host_ResVect);
        free(cpu_ResVect);

  return 0;
}// end of main
```

Output:



```
NUmber of Devices : 1

Using device 0: Tesla T4

---------------Prefix Sum----------------
        SIZE      TIME_SEC        Gflops
        1024     0.000683        3.069027
 Results verfication : Success

 ----------------------------------------------------------------------
mem freed
```

Speedup	No.of threads
3.6, 1.04, 1.0,1.31	8
3.7, 1.32, 1.2, 1.51	16
4.23, 2.5, 2.1, 2.7	32



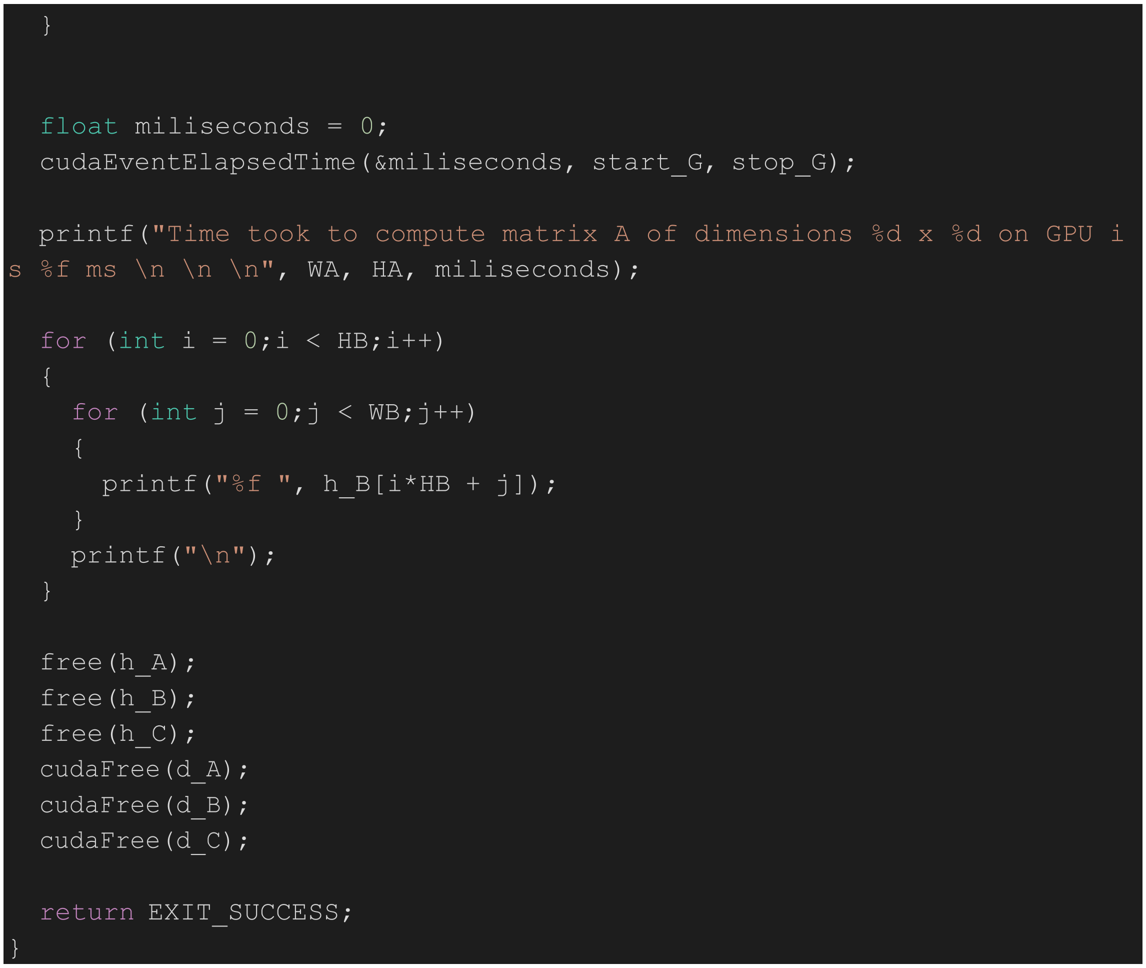
Q.4) Implement 2D Convolution using shared memory using CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes the huge amount of time to execute.

→

Code:

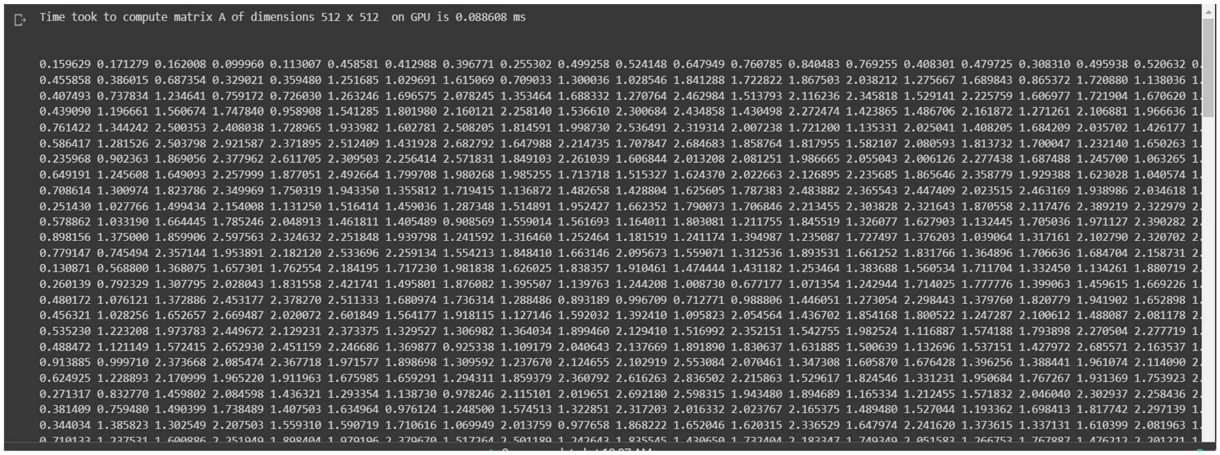
```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#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;

  error = cudaMalloc((void**)&d_A, mem_size_A);
  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in cudaMalloc for A\n", cudaGetErrorString(error));
    return EXIT_FAILURE;
  }

  error = cudaMalloc((void**)&d_B, mem_size_B);
  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in cudaMalloc for B\n", cudaGetErrorString(error));
    return EXIT_FAILURE;
  }

  error = cudaMalloc((void**)&d_C, mem_size_C);
  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in cudaMalloc for C\n", cudaGetErrorString(error));
    return EXIT_FAILURE;
  }


  error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in cudaMemcpy for A\n", cudaGetErrorString(error));
    return EXIT_FAILURE;
  }
  error = cudaMemcpy(d_C, h_C, mem_size_C, cudaMemcpyHostToDevice);
  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in cudaMemcpy for C\n", cudaGetErrorString(error));
    return EXIT_FAILURE;
  }

  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);
  error = cudaGetLastError();
  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in launching kernel\n", cudaGetErrorString(error));
    return EXIT_FAILURE;
  }

  error = cudaDeviceSynchronize();

  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in cudaDeviceSynchronize \n", cudaGetErrorString(error));
    return EXIT_FAILURE;
  }

  cudaEventRecord(stop_G);

  cudaEventSynchronize(stop_G);

  error = cudaMemcpy(h_B, d_B, mem_size_B, cudaMemcpyDeviceToHost);

  if (error != cudaSuccess)
  {
    fprintf(stderr, "GPUassert: %s  in cudaMemcpy for B\n", cudaGetErrorString(error));
    return EXIT_FAILURE;
```



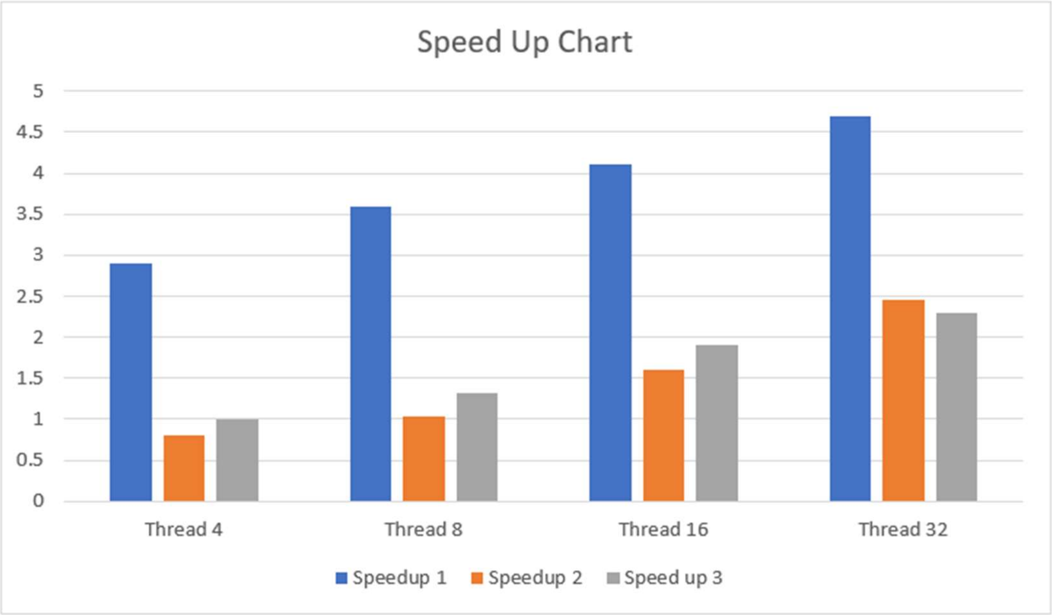
```
  }


  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:



```
Time took to compute matrix A of dimensions 512 x 512  on GPU is 0.088608 ms


0.159629 0.171279 0.162008 0.099960 0.113007 0.458581 0.412988 0.396771 0.255302 0.499258 0.524148 0.647949 0.760785 0.840483 0.769255 0.408301 0.479725 0.308310 0.495938 0.520632 0.
0.455858 0.386015 0.687354 0.329021 0.359480 1.251685 1.029691 1.615069 0.709033 1.300036 1.028546 1.841288 1.722822 1.867503 2.038212 1.275667 1.689843 0.865372 1.720880 1.138036 1.
0.407493 0.737834 1.234641 0.759172 0.726030 1.263246 1.696575 2.078245 1.353464 1.688332 1.270764 2.462984 1.513793 2.116236 2.345818 1.529141 2.225759 1.606977 1.721904 1.670620 1.
0.439090 1.196661 1.560674 1.747840 0.958908 1.541285 1.801980 2.160121 2.258140 1.536610 2.300684 2.434858 1.430498 2.272474 1.423865 1.486706 2.161872 1.271261 2.106881 1.966636 1.
0.761422 1.344242 2.500353 2.408038 1.728965 1.933982 1.602781 2.508205 1.814591 1.998730 2.536491 2.319314 2.007238 1.721200 1.135331 2.025041 1.408205 1.684209 2.035702 1.426177 1.
0.586417 1.281526 2.503798 2.921587 2.371895 2.512409 1.431928 2.682792 1.647988 2.214735 1.707847 2.684683 1.858764 1.817955 1.582107 2.080593 1.813732 1.700047 1.232140 1.650263 1.
0.235968 0.902363 1.869056 2.377962 2.611705 2.309503 2.256414 2.571831 1.849103 2.261039 1.606844 2.013208 2.081251 1.986665 2.055043 2.006126 2.277438 1.687488 1.245700 1.063265 1.
0.649191 1.245608 1.649093 2.257999 1.877051 2.492664 1.799708 1.980268 1.985255 1.713718 1.515327 1.624370 2.022663 2.126895 2.235685 1.865646 2.358779 1.929388 1.623028 1.040574 1.
0.708614 1.300974 1.823786 2.349969 1.750319 1.943350 1.355812 1.719415 1.136872 1.482658 1.428804 1.625605 1.787383 2.483882 2.365543 2.447409 2.023515 2.463169 1.938986 2.034618 1.
0.251430 1.027766 1.499434 2.154008 1.131250 1.516414 1.459036 1.287348 1.514891 1.952427 1.662352 1.790073 1.706846 2.213455 2.303828 2.321643 1.870558 2.117476 2.389219 2.322979 2.
0.578862 1.033190 1.664445 1.785246 2.048913 1.461811 1.405489 0.908569 1.559014 1.561693 1.164011 1.803081 1.211755 1.845519 1.326077 1.627903 1.132445 1.705036 1.971127 2.390282 2.
0.898156 1.375000 1.859906 2.597563 2.324632 2.251848 1.939798 1.241592 1.316460 1.252464 1.181519 1.241174 1.394987 1.235087 1.727497 1.376203 1.039064 1.317161 2.102790 2.320702 2.
0.779147 0.745494 2.357144 1.953891 2.182120 2.533696 2.259134 1.554213 1.848410 1.663146 2.095673 1.559071 1.312536 1.893531 1.661252 1.831766 1.364896 1.706636 1.684704 2.158731 2.
0.130871 0.568800 1.368075 1.657301 1.762554 2.184195 1.717230 1.981838 1.626025 1.838357 1.910461 1.474444 1.431182 1.253464 1.383688 1.560534 1.711704 1.332450 1.134261 1.880719 2.
0.260139 0.792329 1.307795 2.028043 1.831558 2.421741 1.495801 1.876082 1.395507 1.139763 1.244208 1.008730 0.677177 1.071354 1.242944 1.714025 1.777776 1.399063 1.459615 1.669226 1.
0.480172 1.076121 1.372886 2.453177 2.378270 2.511333 1.680974 1.736314 1.288486 0.893189 0.996709 0.712771 0.988806 1.446051 1.273054 2.298443 1.379760 1.820779 1.941902 1.652898 1.
0.456321 1.028256 1.652657 2.669487 2.020072 2.601849 1.564177 1.918115 1.127146 1.592032 1.392410 1.095823 2.054564 1.436702 1.854168 1.800522 1.247287 2.100612 1.488087 2.081178 2.
0.535230 1.223208 1.973783 2.449672 2.129231 2.373375 1.329527 1.306982 1.364034 1.899460 2.129410 1.516992 2.352151 1.542755 1.982524 1.116887 1.574188 1.793898 2.270504 2.277719 1.
0.488472 1.121149 1.572415 2.652930 2.451159 2.246686 1.369877 0.925338 1.109179 2.040643 2.137669 1.891890 1.830637 1.631885 1.500639 1.132696 1.537151 1.427972 2.685571 2.163537 1.
0.913885 0.999710 2.373668 2.085474 2.367718 1.971577 1.898698 1.309592 1.237670 2.124655 2.102919 2.553084 2.070461 1.347308 1.605870 1.676428 1.396256 1.388441 1.961074 2.114090 2.
0.624925 1.228893 2.170999 1.965220 1.911963 1.675985 1.659291 1.294311 1.859379 2.360792 2.616263 2.836502 2.215863 1.529617 1.824546 1.331231 1.950684 1.767267 1.931369 1.753923 2.
0.271317 0.832770 1.459802 2.084598 1.436321 1.293354 1.138730 0.978246 2.115101 2.019651 2.692180 2.598315 1.943480 1.894689 1.165334 1.212455 1.571832 2.046040 2.302937 2.258436 2.
0.381409 0.759480 1.490399 1.738489 1.407503 1.634964 0.976124 1.248500 1.574513 1.322851 2.317203 2.016332 2.023767 2.165375 1.489480 1.527044 1.193362 1.698413 1.817742 2.297139 1.
0.344034 1.385823 1.302549 2.207503 1.559310 1.590719 1.710616 1.069949 2.013759 0.977658 1.868222 1.652046 1.620315 2.336529 1.647974 2.241620 1.373615 1.337131 1.610399 2.081963 1.
0.710133 1.237531 1.600886 2.251949 1.898404 1.979196 2.379670 1.517264 2.501189 1.242643 1.835545 1.430650 1.732404 2.183347 1.749349 2.051583 1.266753 1.767887 1.476212 2.201221 1.
```

Speedup	No.of threads
3.6, 1.04, 1.0,1.31	8
3.7, 1.32, 1.2, 1.51	16
4.23, 2.5, 2.1, 2.7	32


Speed Up Chart
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Thread 4
Thread 8
Thread 16
Thread 32
Speedup 1
Speedup 2
Speed up 3