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

Class: Final Year (Computer Science and Engineering)

Year: 2022-23 **Semester:** 1

Course: High Performance Computing Lab

Practical No. 10

Exam Seat No: 2019BTECS00070

Name: Prathmesh Killedar

Title of practical:

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

Information #:

```
#include <stdio.h>

#define N  64

__global__ void matrixMulGPU( int * a, int * b, int * c )
{
	int val = 0;

	int row = blockIdx.x * blockDim.x + threadIdx.x;
```

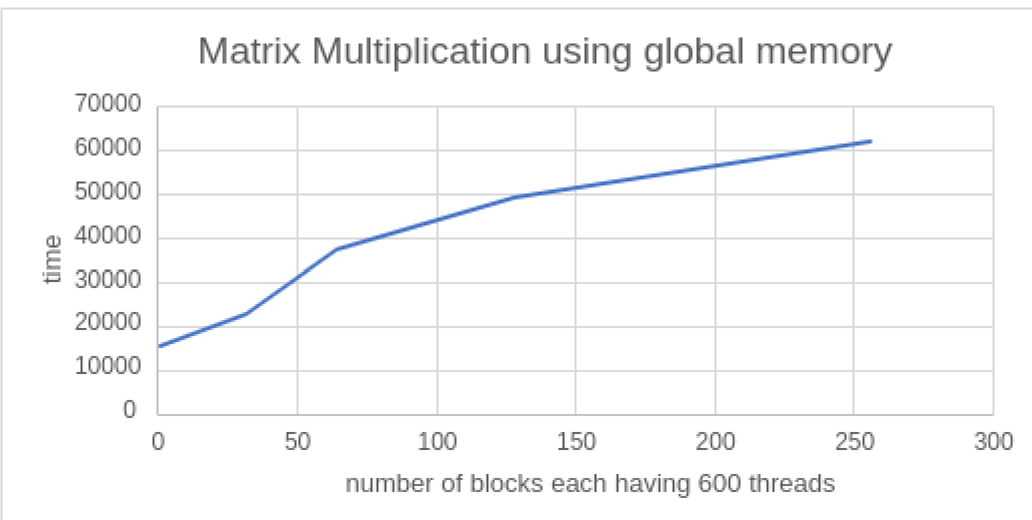
```
        int col = blockIdx.y * blockDim.y + threadIdx.y;

        if (row < N && col < N)
        {
                for ( int k = 0; k < N; ++k )
                        val += a[row * N + k] * b[k * N + col];
                c[row * N + col] = val;
        }
}

void matrixMulCPU( int * a, int * b, int * c )
{
        int val = 0;

        for( int row = 0; row < N; ++row )
                for( int col = 0; col < N; ++col )
                {
                        val = 0;
                        for ( int k = 0; k < N; ++k )
                                val += a[row * N + k] * b[k * N + col];
                        c[row * N + col] = val;
                }
}

int main()
{
        int *a, *b, *c_cpu, *c_gpu;

        int size = N * N * sizeof (int); // Number of bytes of an N x N matrix

        // Allocate memory
        cudaMallocManaged (&a, size);
        cudaMallocManaged (&b, size);
        cudaMallocManaged (&c_cpu, size);
```

```
        cudaMallocManaged (&c_gpu, size);

        // Initialize memory
        for( int row = 0; row < N; ++row )
                for( int col = 0; col < N; ++col )
                {
                        a[row*N + col] = row;
                        b[row*N + col] = col+2;
                        c_cpu[row*N + col] = 0;
                        c_gpu[row*N + col] = 0;
                }

        dim3 threads_per_block (16, 16, 1); // A 16 x 16 block threads
        dim3 number_of_blocks ((N / threads_per_block.x) + 1, (N /
threads_per_block.y) + 1, 1);

        matrixMulGPU <<< number_of_blocks, threads_per_block >>> ( a, b,
c_gpu );

        cudaDeviceSynchronize(); // Wait for the GPU to finish before
proceeding

        // Call the CPU version to check our work
        matrixMulCPU( a, b, c_cpu );

        // Compare the two answers to make sure they are equal
        bool error = false;
        for( int row = 0; row < N && !error; ++row )
                for( int col = 0; col < N && !error; ++col )
                        if (c_cpu[row * N + col] != c_gpu[row * N + col])
                        {
                                printf("FOUND ERROR at c[%d][%d]\n", row, col);
                                error = true;
                                break;
```

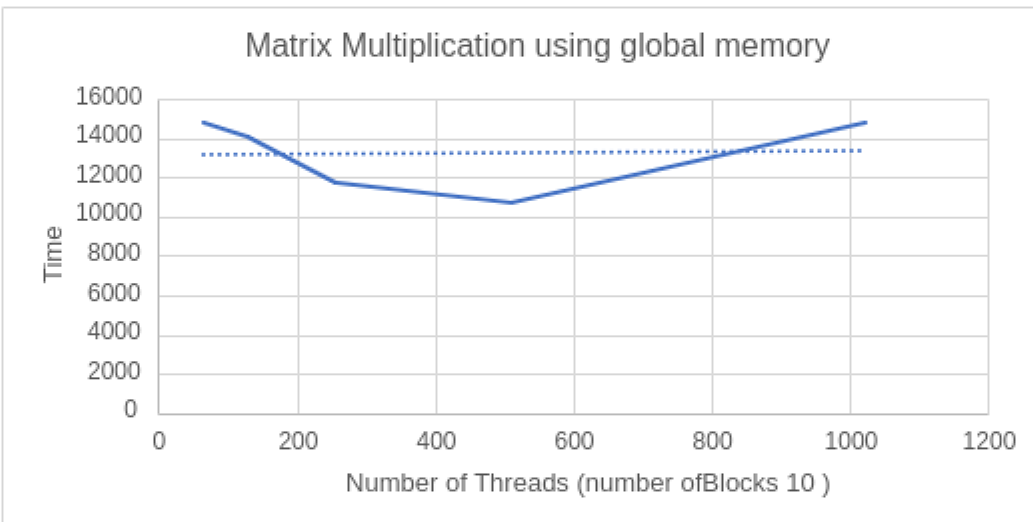
```
                    }
        if (!error)
                printf("Success!\n");

        // Free all our allocated memory
        cudaFree(a);
        cudaFree(b);
        cudaFree( c_cpu );
        cudaFree( c_gpu );
}
```

Serial execution time: 0.000224 second

Number of blocks (600 threads each)	Time required	speedup
1	15168	14.2857
32	22687	9.8734
64	37343	5.9984
128	48991	4.5722
256	61983	3.6138



Number of Threads with constant block size 10	Time required	speedup
64	14816	15.1187
128	14048	15.9453
256	11776	19.0217
512	10656	21.021
1024	14816	15.1187



Conclusion:

a. For constant number of threads we have concluded that the execution time is increasing with the increasing number of blocks
b. For constant number of block we have concluded that the execution time is decreasing until a certain point and after that it is increasing due to communication overhead by increasing the number of threads per block

Problem Statement 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 a huge amount of time to execute.

Information #:

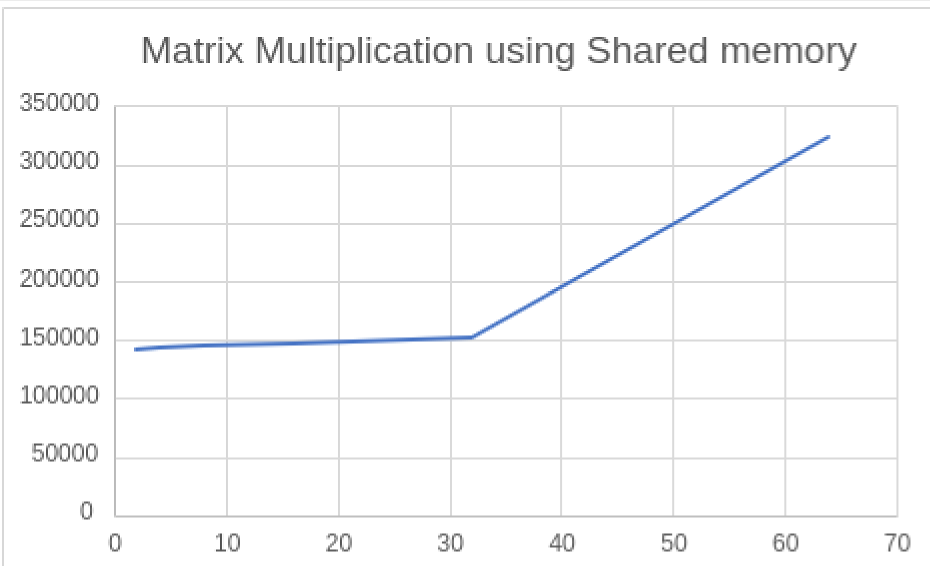
```
#include <stdio.h>
#include <math.h>
#define TILE_WIDTH 2

/*matrix multiplication kernels*/

// shared
__global__ void
MatrixMulSh( float *Md , float *Nd , float *Pd , const int WIDTH )
{

    //Taking shared array to break the MAtrix in Tile widthand fetch them
in that array per ele

    __shared__ float Mds [TILE_WIDTH][TILE_WIDTH] ;

    __shared__ float Nds [TILE_WIDTH][TILE_WIDTH] ;

    // calculate thread id
    unsigned int col = TILE_WIDTH*blockIdx.x + threadIdx.x ;
    unsigned int row = TILE_WIDTH*blockIdx.y + threadIdx.y ;

    for (int m = 0 ; m<WIDTH/TILE_WIDTH ; m++ ) // m indicate number
of phase
```

```
        {
                Mds[threadIdx.y][threadIdx.x] =  Md[row*WIDTH + 
    (m*TILE_WIDTH + threadIdx.x)]  ;
                Nds[threadIdx.y][threadIdx.x] =  Nd[ ( m*TILE_WIDTH + 
    threadIdx.y) * WIDTH + col] ;
                __syncthreads() ; // for synchronizing the threads

                // Do for tile
                for ( int k = 0; k<TILE_WIDTH ; k++ )
                        Pd[row*WIDTH + col]+= Mds[threadIdx.x][k] * 
    Nds[k][threadIdx.y] ;
                __syncthreads() ; // for synchronizing the threads

        }
    }

    // main routine
    int main ()
    {
        const int WIDTH = 500;
        float array1_h[WIDTH][WIDTH] ,array2_h[WIDTH][WIDTH],
    M_result_array_h[WIDTH][WIDTH]  ;
        float *array1_d , *array2_d ,*result_array_d  ,*M_result_array_d ; // 
    device array
        int i , j ;
        //input in host array
        for ( i = 0 ; i<WIDTH ; i++ )
        {
                for (j = 0 ; j<WIDTH ; j++ )
                {
                        array1_h[i][j] = (i + 2*j) %500 ;
                        array2_h[i][j] = (i + 3*j) %500 ;
                }
        }
```

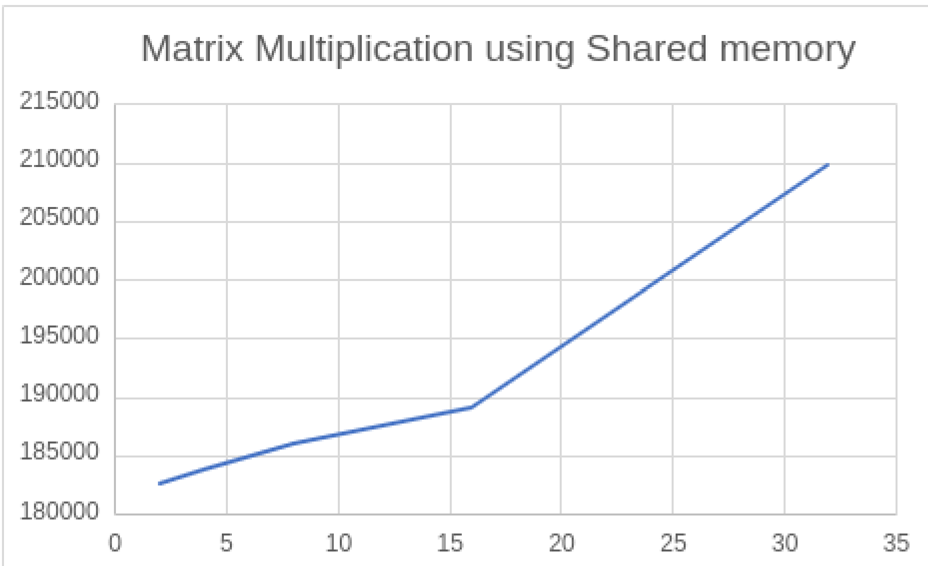
```
//create device array cudaMalloc ( (void **)&array_name, sizeofmatrixinbytes) ;

cudaMalloc((void **) &array1_d , WIDTH*WIDTH*sizeof (int) ) ;

cudaMalloc((void **) &array2_d , WIDTH*WIDTH*sizeof (int) ) ;



//copy host array to device array; cudaMemcpy ( dest , source , WIDTH , direction )

cudaMemcpy ( array1_d , array1_h , WIDTH*WIDTH*sizeof (int) , cudaMemcpyHostToDevice ) ;

cudaMemcpy ( array2_d , array2_h , WIDTH*WIDTH*sizeof (int) , cudaMemcpyHostToDevice ) ;



//allocating memory for resultent device array

cudaMalloc((void **) &result_array_d , WIDTH*WIDTH*sizeof (int) ) ;

cudaMalloc((void **) &M_result_array_d , WIDTH*WIDTH*sizeof (int) ) ;

MatrixMulSh<<<512,32>>> ( array1_d , array2_d ,M_result_array_d , WIDTH) ;

// all gpu function blocked till kernel is working
//copy back result_array_d to result_array_h
```

```
        cudaMemcpy(M_result_array_h , M_result_array_d ,
WIDTH*WIDTH*sizeof(int) ,cudaMemcpyDeviceToHost) ;

    printf("Multiplication Successful using shared Memory");

}
```

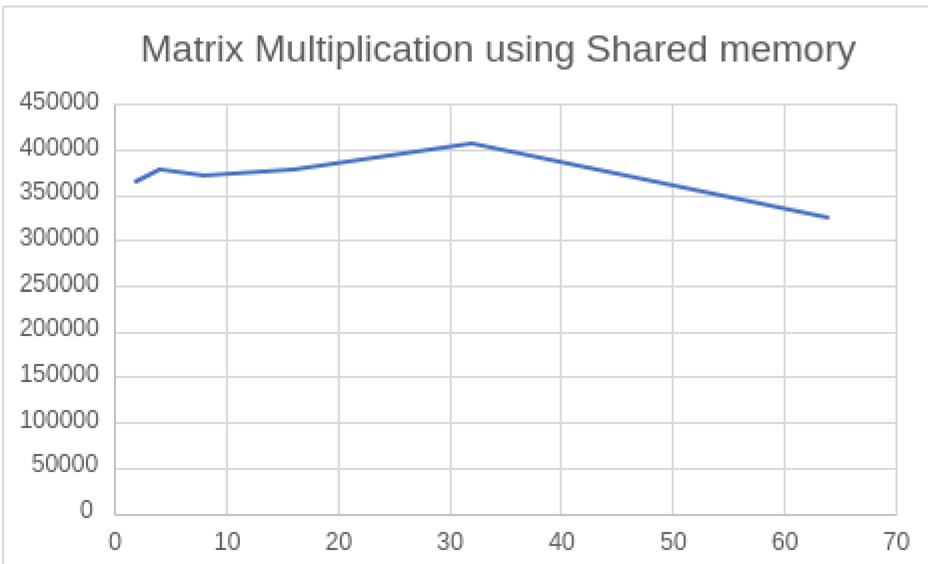
Number of Threads with constant block size 256	Time required
2	141692
4	143356
8	145020
16	146556
32	151196
64	323192



Number of Threads with constant block size 512	Time required
2	182555
4	183771
8	185947
16	189019
32	209755



Number of Threads with constant block size 1024	Time required
2	364726
4	377142
8	372630
16	378135
32	406197
64	324248



Conclusion:

a. For constant number of blocks we have concluded that the execution time is increasing with the increasing number of threads
b. For constant number of threads per block at a partic

Problem Statement 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.

Information #:

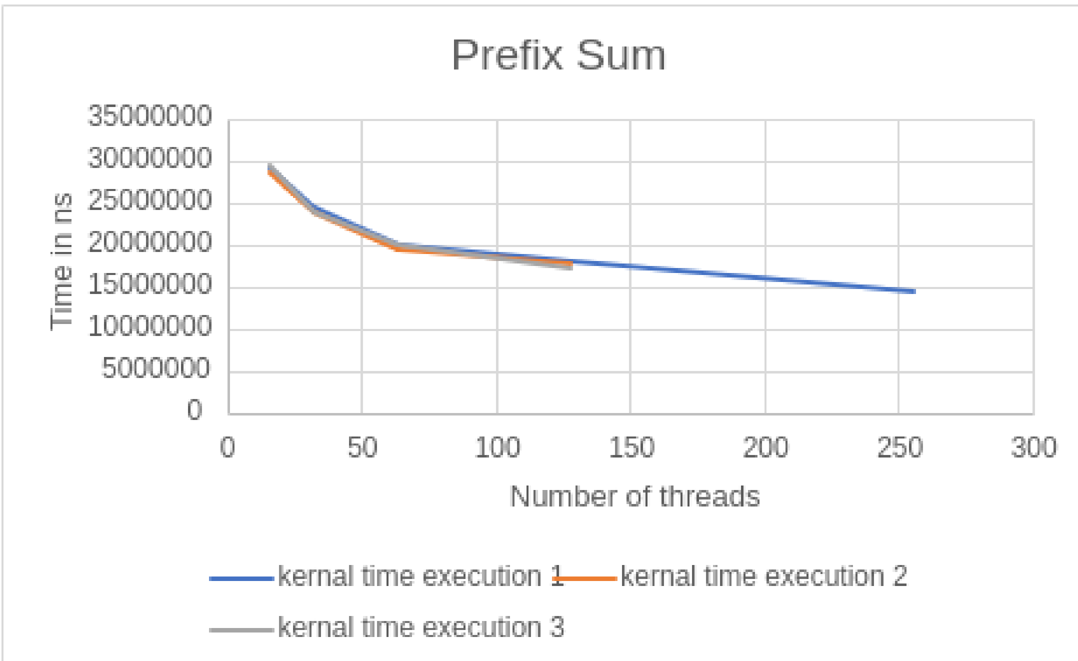
```
#include <stdio.h>

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 = 32;
    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);
}
```

No of threads (1 block)	kernal time execution 1	kernal time execution 2	kernal time execution 3
16	29185007	28690201	29448042
32	24658464	23898575	24076521
64	20158082	19436169	20129785
128	18123523	17824168	17406672
256	14619018		



Conclusion:
As there is lack of synchronisation in blocks but there is synchronisation in threads, so for the prefix sum problem we consider only one block with varying number of threads. So by observing the above graph we have concluded that as the number of threads increases execution time is decreasing.

Problem Statement 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.

Information #:

```
#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++)
```

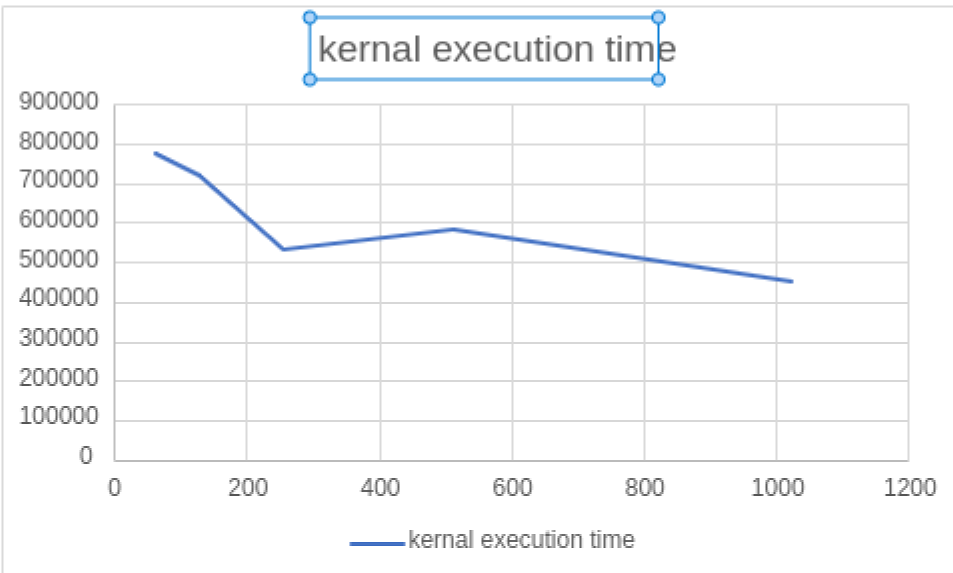
```
    {
      // Go over each column
      for (int j = 0; j < MASK_DIM; j++)
      {
        // Range check for rows
        if ((start_r + i) >= 0 && (start_r + i) < N)
        {
          // Range check for columns
          if ((start_c + j) >= 0 && (start_c + j) < N)
          {
            // 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
```

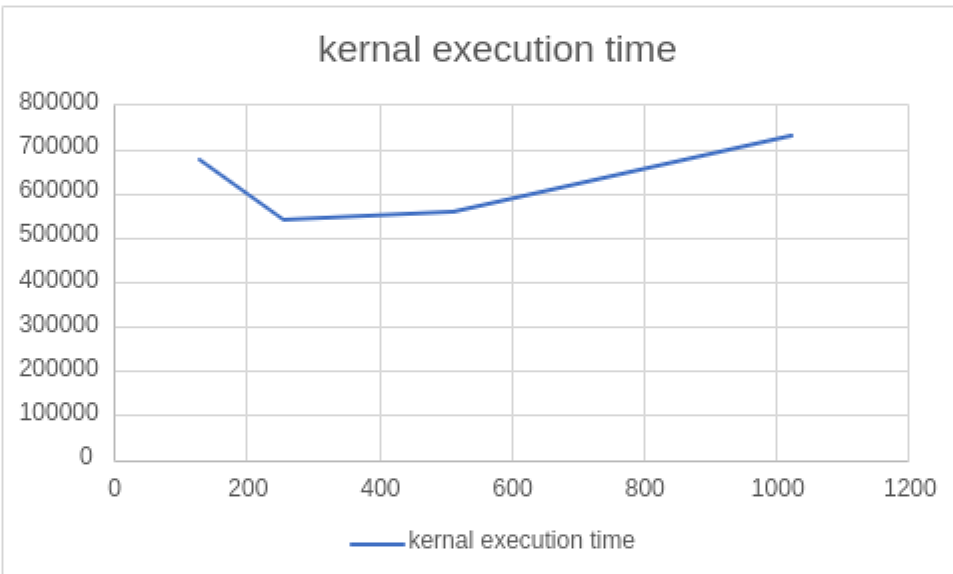
```
                    temp += m[offset_r * N + offset_c] * mask[k * MASK_DIM +
l];
                }
            }
        }
    }
    // Fail if the results don't match
    if (result[i * N + j] != temp)
    {
        printf("Check failed");
        return;
    }
  }
 }
}

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 = 64;
    //int BLOCKS = (N + THREADS - 1) / THREADS;

    // Dimension launch arguments
    //dim3 block_dim(THREADS, THREADS);
    //dim3 grid_dim(BLOCKS, BLOCKS);

    //printf("%d %d",grid_dim.y,block_dim.y);

      convolution_2d<<<128, 1024>>>(matrix, result, N);

    verify_result(matrix, h_mask, result, N);

    printf("COMPLETED SUCCESSFULLY!");

    cudaFree(matrix);
    cudaFree(result);
    cudaFree(h_mask);

    return 0;
}
```

Blocks (thread constant 128)	kernal execution time
64	777744
128	720112
256	533781
512	582900
1024	448567



Threads(Block Constant 128)	kernal execution time
128	674482
256	539349
512	559860
1024	731410



Github Link:https://github.com/killedar27/HPC-assignments/tree/main/assignment10