Course: High-Performance Computing Lab

Borse Vishal Subhash
2019BTECS00066
B3

Assignment - 10

que 1:

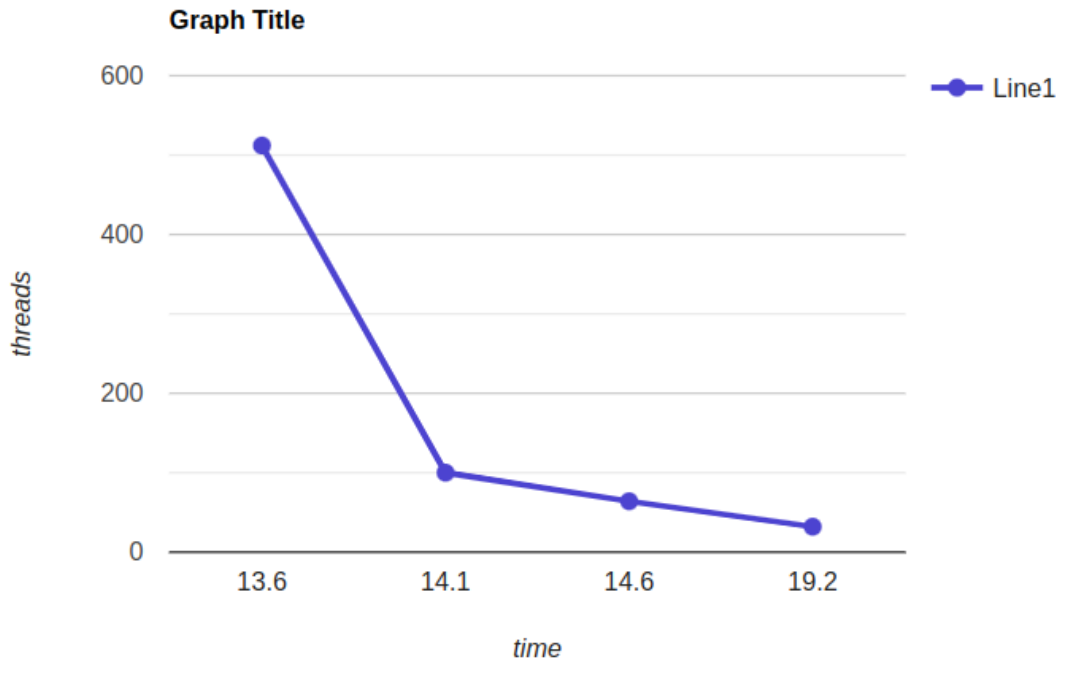
1. **Implement Matrix-matrix Multiplication using global memory in CUDA C. Analyze and tune the program for getting maximum speed up. Do Profiling and state what part of the code takes a huge amount of time to execute.**

   Code:

```
#include <stdio.h>
void initWith(float num, float *a, int SIZE)
{
   for (int i = 0; i < SIZE; ++i)
   {
       a[i] = num;
   }
}
__global__
void matrixMultiply(float *result, float *a, float *b, int N, int SIZE)
{
   int start = blockIdx.x * blockDim.x + threadIdx.x;
   int stride = gridDim.x * blockDim.x;
   for (int i = start; i < SIZE; i += stride)
   {
       int row = i / N;
       float sum = 0;
       for (int j = 0; j < N; j++)
       {
           sum += a[row * N + j] * b[N * j + row];
       } result[i] = sum;
   }
}
```

```
void checkElementsAre(float target, float *array, int SIZE)
{
   for (int i = 0; i < SIZE; i++)
   {
       if (array[i] != target)
       {
           printf("FAIL: array[%d] - %0.0f does not equal %0.0f\n", i,
array[i], target);
           exit(1);
       }
   }
   printf("SUCCESS! All values multiplied correctly.\n");
}
int main()
{
   const int N = 1024;
   const int SIZE = N * N; // sqaure matrix
   size_t size = SIZE * sizeof(float);
   float *a;
   float *b;
   float *c;
   cudaMallocManaged(&a, size);
   cudaMallocManaged(&b, size);
   cudaMallocManaged(&c, size);
   initWith(3, a, SIZE);
   initWith(4, b, SIZE);
   initWith(0, c, SIZE);
   matrixMultiply <<< 6, 96>>>(c, a, b, N, SIZE);
   cudaDeviceSynchronize();
   checkElementsAre(12288, c, SIZE);
   cudaFree(a);
   cudaFree(b);
   cudaFree(c);
}
```

N	Threads	Blocks	Time
1024	100	1024	`12886611`
1024	100	512	`13680123`
1024	100	128	`14069869`
1024	512	128	`13618225`
1024	32	128	`19194956`
1024	64	128	`14578261`
1024	16	64	`46801319`



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.

Code:

```
#include <stdio.h>
void initWith(float num, float *a, int SIZE)
{
   for (int i = 0; i < SIZE; ++i)
   {
       a[i] = num;
   }
}
__global__
void matrixMultiply(float *result, float *a, float *b, int N, int SIZE)
{
   __shared__ int stride;
   if (threadIdx.x == 0)
       stride = gridDim.x * blockDim.x;
   __syncthreads();
   int start = blockIdx.x * blockDim.x + threadIdx.x;
   for (int i = start; i < SIZE; i += stride)
   {
       int row = i / N;
       float sum = 0;
       for (int j = 0; j < N; j++)
       {
           sum += a[row * N + j] * b[N * j + row];
       }
       result[i] = sum;
   }
}
void checkElementsAre(float target, float *array, int SIZE)
{
   for (int i = 0; i < SIZE; i++)
   {
       if (array[i] != target)
       {
```

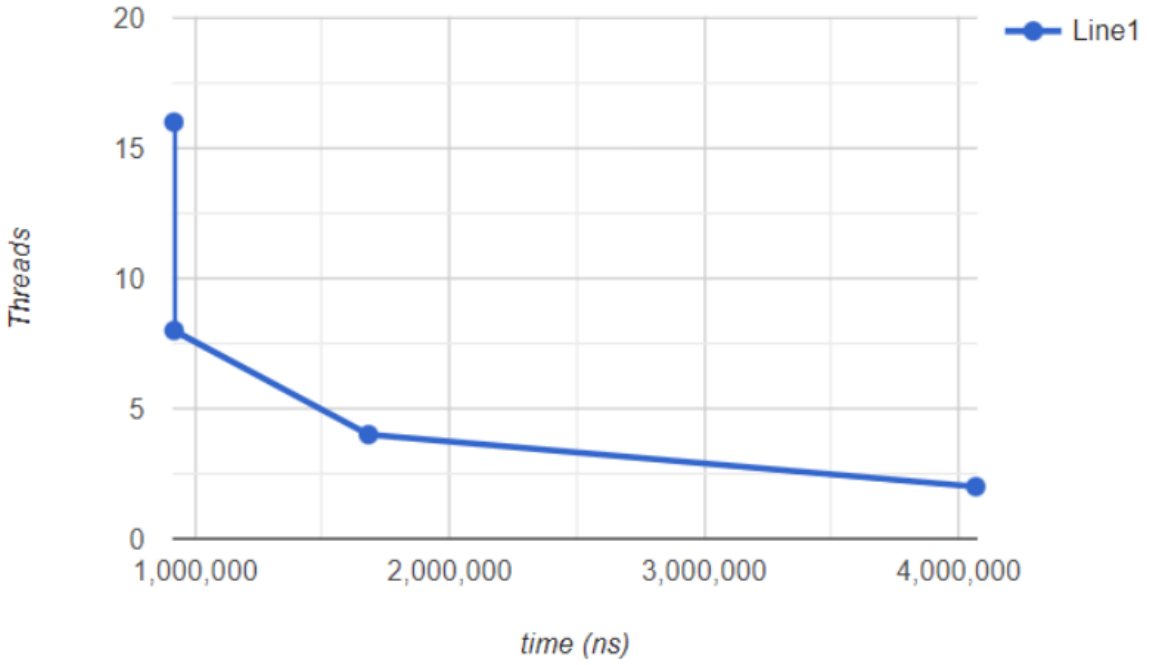
```
            printf("FAIL: array[%d] - %0.0f does not equal %0.0f\n", i,
array[i], target); exit(1);
        }
    }
    printf("SUCCESS! All values multiplied correctly.\n");
}
int main()
{
    const int N = 1024;
    const int SIZE = N * N; // sqaure matrix
    size_t size = SIZE * sizeof(float);
    float *a;
    float *b;
    float *c;
    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);
    initWith(3, a, SIZE);
    initWith(4, b, SIZE);
    initWith(0, c, SIZE);
    matrixMultiply <<< 6, 96>>>(c, a, b, N, SIZE);
    cudaDeviceSynchronize();
    checkElementsAre(12288, c, SIZE);
    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
}
```

N	Threads	Blocks	Time(ns)	shared memory size
1<< 8	2	128	4067591	16 * 16
1 << 8	4	64	1681160	16 * 16
1<< 8	8	32	917491	16 * 16
1 << 8	16	16	916755	16 * 16
1 << 10	16	64	12942828	16 *16
1 << 11	16	128	83486357	16 * 16
1<< 11	32	64	74789418	32 * 32
1 << 12	4	64	45854254	32 * 32
1 << 12	8	64	51990584	32 * 32
1<<12	16	64	75095919	16 * 16
1 << 12	16	256	393892197	16 * 16
1 << 12	32	64	150768098	32 * 32
1<< 13	16	512	2019049036	16 * 16

Here you can see, for N = 1<< 12 keeping the the threads constant and variable block size :

(sm = 16 * 16)for 64 block size the time is much less than 256 block size, this is because better utilization of sms is for 64 blocks.

but as we change the shared memory size to 32 and threads 32 for 64 blocks, the time increased this is because of thread sync with shared memory.



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]

3. ImplementPrefix 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 :

```
#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 = 6;
 int threadsPerBlock = 96;
 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);
}
```

For N = 10 ^ 6

If run serially

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  ---------------------
    97.5       275623254          4  68905813.5     4537  275469088  cudaMallocManaged
     2.3         6612393          1   6612393.0  6612393    6612393  cudaDeviceSynchronize
     0.1          377067          1    377067.0   377067     377067  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                      Name
 -------  ---------------  ---------  ---------  -------  -------  ---------------------------------------------
   100.0          6600768          1  6600768.0  6600768  6600768  prefixSum(float*, float*, float*, float*, int)
```

For <<<1,1>>>

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average     Minimum    Maximum           Name
 -------  ---------------  ---------  -----------  ---------  ---------  --------------------
    60.9        229171241          4   57292810.3       6083  229052749  cudaMallocManaged
    38.7        145776022          1  145776022.0  145776022  145776022  cudaDeviceSynchronize
     0.3          1176141          4     294035.3      26242     609427  cudaFree
     0.1           353577          1     353577.0     353577     353577  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances    Average     Minimum    Maximum                        Name
 -------  ---------------  ---------  -----------  ---------  ---------  ----------------------------------------
-----
   100.0        145763557          1  145763557.0  145763557  145763557  prefixSum(float*, float*, float*, float*,
int)
```

For <<<6,32>>>

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------------
    96.0        240207261          4  60051815.3     5963  240067838  cudaMallocManaged
     3.8          9604144          1   9604144.0  9604144    9604144  cudaDeviceSynchronize
     0.1           370128          1    370128.0   370128     370128  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                       Name
 -------  ---------------  ---------  ---------  -------  -------  ---------------------------------------------
   100.0          9599276          1  9599276.0  9599276  9599276  prefixSum(float*, float*, float*, float*, int)
```

For <<<6,96>>>

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------------
    97.1        232941186          4  58235296.5    12322  232658662  cudaMallocManaged
     2.6          6295650          1   6295650.0  6295650    6295650  cudaDeviceSynchronize
     0.3           741614          1    741614.0   741614     741614  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                       Name
 -------  ---------------  ---------  ---------  -------  -------  ---------------------------------------------
   100.0          6291298          1  6291298.0  6291298  6291298  prefixSum(float*, float*, float*, float*, int)
```

for<<<6,160>>>

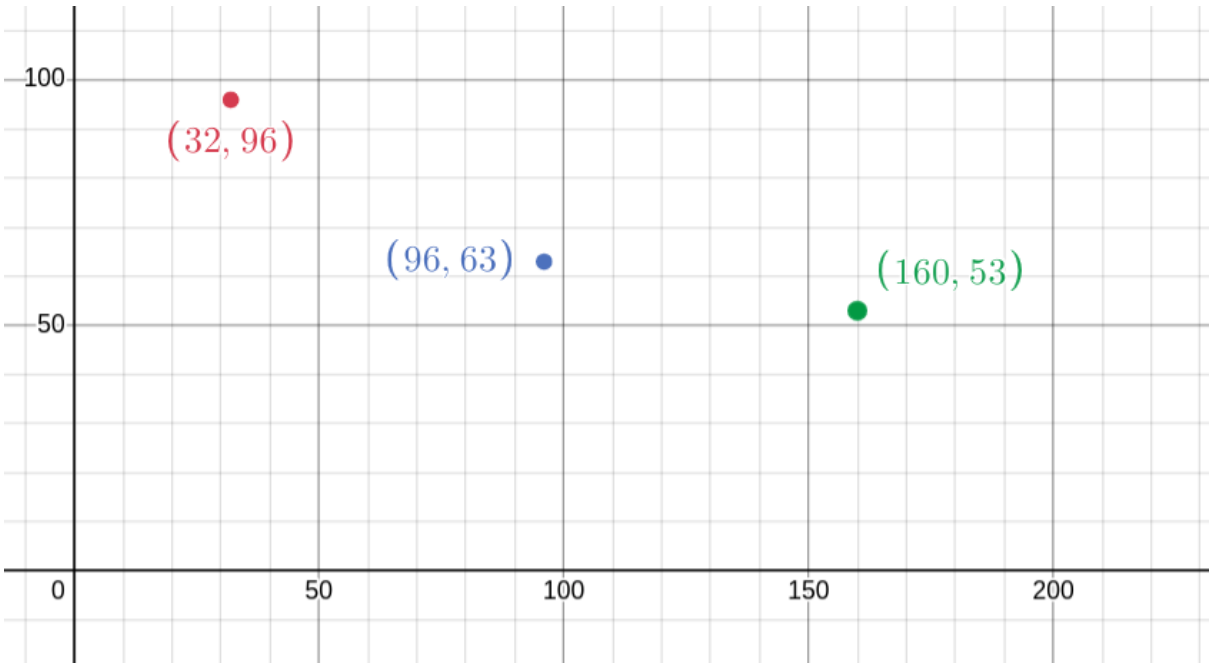
```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------------
    97.7        255821336          4  63955334.0    14973  255554105  cudaMallocManaged
     2.0          5328455          1   5328455.0  5328455    5328455  cudaDeviceSynchronize
     0.3           718446          1    718446.0   718446     718446  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                      Name
 -------  ---------------  ---------  ---------  -------  -------  ---------------------------------------------
   100.0          5325976          1  5325976.0  5325976  5325976  prefixSum(float*, float*, float*, float*, int)
```

No of Block	No of thread	Runtime	
serial	serial	6600768.0	6.6
1	1	145763557.0	145.8
6	32	9599276.0	9.6
6	96	6291298.0	6.3
6	160	5325976.0	5.3



X - number of thread
Y - Runtime
Number of Blocks is constant

For N = 10^6

For <<<2,32>>>

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum    Maximum           Name
 -------  ---------------  ---------  ----------  --------  ---------  --------------------
    93.4        240537604          4  60134401.0      5882  240411456  cudaMallocManaged
     6.5         16761041          1  16761041.0  16761041   16761041  cudaDeviceSynchronize
     0.1           372952          1    372952.0    372952     372952  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average    Minimum   Maximum                       Name
 -------  ---------------  ---------  ----------  --------  --------  ------------------------------------------
--
   100.0         16747733          1  16747733.0  16747733  16747733  prefixSum(float*, float*, float*, float*, in
t)
```

For <<<8,32>>>

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------------
    96.9        246856846          4  61714211.5     3940  246733482  cudaMallocManaged
     2.9          7501688          1   7501688.0  7501688    7501688  cudaDeviceSynchronize
     0.1           366461          1    366461.0   366461     366461  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                       Name
 -------  ---------------  ---------  ---------  -------  -------  ------------------------------------------------
   100.0          7494626          1  7494626.0  7494626  7494626  prefixSum(float*, float*, float*, float*, int)
```

for<<<16,32>>>

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------------
    97.9        305677506          4  76419376.5     6690  305438135  cudaMallocManaged
     2.0          6186105          1   6186105.0  6186105    6186105  cudaDeviceSynchronize
     0.1           467282          1    467282.0   467282     467282  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                       Name
 -------  ---------------  ---------  ---------  -------  -------  ------------------------------------------------
   100.0          6180260          1  6180260.0  6180260  6180260  prefixSum(float*, float*, float*, float*, int)
```

No of Block	No of thread	Runtime	
serial	Serial	6600768.0	6.6
1	1	145763557.0	145.8
2	32	16747733.0	16.7
8	32	7494626.0	7.5
16	32	6180260.0	6.2

X - number of blocks
Y - runtime
Keeping number of threads constant

Execution time: As expected, the execution time of kernel decreases as the total threads (blocks * threads) increases. Total time taken by program also follows a similar trend. Memory allocation and deallocation: All instances seem to follow random fluctuations since the same amount of memory is being allocated outside any parallel region. But is pretty much constant Memory Migration in Unified Memory: Copying memory from Host to Device takes almost 1.33 times more time than Device to Host. But the time required for these operations is pretty much constant across changed kernel configuration. Most time consuming part: Allocation of memory by cudaMallocManaged takes largest time in the entire program in most cases

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.

```
Code :
#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 columnoffset_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 = 32;
    int BLOCKS = (N + THREADS - 1) / THREADS;
// Dimension launch arguments
    dim3 block_dim(THREADS, THREADS);
    dim3 grid_dim(BLOCKS, BLOCKS);
```

```
    convolution_2d <<< grid_dim, block_dim>>>(matrix, result, N);
    verify_result(matrix, h_mask, result, N);
    printf("COMPLETED SUCCESSFULLY!");
    cudaFree(matrix);
    cudaFree(result);
    cudaFree(h_mask);
    return 0;
}
```