CSC512 Project:4 Report

1.Performance Issues:

In the given program norm.cu, each thread is calculating sum of particular set of elements or numbers.
Threads in the same warp should execute same execution trajectory while in this program all threads are not executing the same instructions at the same time. As a result thread divergence is occuring. Second issue is all threads are calculating the sum instead a single thread can perform the task.

In the if- else block of the program, the THEN and ELSE part instead of executing in parallel are executed in serial, due to this the program fails to perform better.

The innermost for loop executes *start + i * width + j* , the part *start + i * width* is independent of the variable j. So in any iteration its value won't change so there is no need to calculate it in a loop. Loop unrolling can be performed to compensate this performance issue.

The on chip memory is not used for data transfers between host and device in the following two parts of code.

```
sum += in[start + i * width + j] * mul[j];
```

```
out[tx * width + ty] = 2.0 * in[tx * width + ty]/sum;
```

2.Optimizations:

1. Optimization 1 : of type **operations a thread needs to do.**
   In this optimization, a single thread has performed the computation. As in the program all the threads were performing the calculation, the execution time possible increases because of that. If a single thread is performing the sum then the execution time can be drastically reduced. As per expectation the execution time was reduced to 0.006581s seconds on 480.

```
if(threadIdx.x==1 && threadIdx.y==1){
for(int i = 0; i < BLOCK_SIZE; i++){
        for(int j = 0; j < BLOCK_SIZE; j++){
                sum += in[start + i * width + j] * mul[j];
        }
}
}
__syncthreads();
```

2. Optimization 2: of type **Minimize thread divergences**
   Another issue that is observed in the program is the thread divergence. In the original program the thread is calculating and storing the normalized value at the same time simultaneously. If we calculate it first and store it to a temporary variable and then store it to the memory, optimization can be achieved. For that I stored the calculation to temporary variable param first and after each calculation stored it to the memory. The performance expected was reduction in almost 100 ms but it reduced to 0.012196s seconds.(480)

```
int location=tx * width + ty;
    float param;

if(tx % 2 == 0 && ty % 2 == 0)
            param= 2.0 * in[location]/sum;
    else if(tx % 2 == 1 && ty % 2 == 0)
            param= in[location]/sum;
    else if(tx % 2 == 1 && ty % 2 == 1)
            param= (-1.0) * in[location]/sum;
    else
            param = 0.0f;

    out[location]=param;
```

3. Optimization 3: of typ **Memory optimizations**

__constant__ float mul[BLOCK_SIZE];

In this optimization, constant memory for data is used that will not change over the course of kernel execution. The mul is declared as constant. The performance is optimized as warp of thread reads from same location. The following instruction is used in order to copy the values to the kernel.

cudaMemcpyToSymbol(mul, hB_in, BLOCK_SIZE * sizeof(float));

This optimization is used because in some situations using constant memory will reduce required memory bandwidth. Gain of almost 68.8 % of original execution time on both hardware.

4. All optimization combined:

In this experiment all the optimizations are combined of the type **operations a thread needs to do, Minimize thread divergences, using better data placement on the memory systems.** By combination of this optimizations the performance issues like thread divergence are solved. The execution time is reduced lot by this combination.

3. Performance:

ON 480!

TYPE	EXECUTION TIME
original	0.012488s
Optimization 1	0.006581s
Optimization 2	0.012196s
Optimization 3	0.011736s
Optimization ALL	0.005019s

ON 780!

TYPE	EXECUTION TIME
original	0.016304s
Optimization 1	0.009405s
Optimization 2	0.015881s
Optimization 3	0.014591s
Optimization ALL	0.007469s

4.Experience on tool:

In the era of fast computing and parallel computing using the GPU can be very beneficial and a lot of knowledge about gpu and cuda programming is obtained. It was useful to know how the kernel works with the dimensions of grid size and block size. The brief information about how all threads in a grid execute same kernel function. It was interesting to know how large data sets can use cuda programming to speed up the computations as reduction in execution time of kernel can be seen by applying optimizations which are provided on the cuda programming guide website on the given normal program in project. Cuda programming can be effectively helpful in future to me whenever optimization is to be done specially when to scale parallelism to many core GPUs.