EC527: High Performance Programming with Multicore and GPUs

Programming Assignment 7

Part1

After setting the environment for the scc1 and load the cuda9.2, the codes can be compiled and ran successfully. And here is the output.

```
-bash-4.2$      nvcc cuda_test_lab7.cu -o cuda_test_lab7
-bash-4.2$ ./cuda_test_lab7
Length of the array = 50000

Initializing the arrays ...      ... done


GPU time: 0.225248 (msec)

TEST PASSED: All results matched
```

Part 2: A simplified SOR

2a:

SOR implemented. The patch each is 16*16.

2b:

I write the function `COMPARE_RESULTS()` to compare the result between the CPU and GPU calculation, and as shown in the output below, most of them are different. For a matrix with 4,000,000 elements, the two SOR versions produce a different output of no errors since I set the compare to be around 95% and 105% range.

```
-bash-4.2$ ./SOR_cuda
init all done

Initializing the arrays ...      ... done
allocate all done

GPU time: 50591.203125 (msec)
GPU calculation ended

CPU time: 108147.992188(msec)
CPU calculation ended
results check finished
Found 0 errors
```

2c:

The time difference is shown above, and we can see that when using the GPU it is much faster than using the CPU, as the GPU takes about 50591 mesc and the CPU takes 108147 mesc.

2d:

For the 1D strip per thread, I simply set a condition loop to set the x to stay constant,

and make y circulate through the whole matrix, and the result still turns out to be ok, but it takes a lot of time.

```
for (int k = 0; k < SOR_ITERATIONS; k++)
    {
        if (local_tid_x == local_tid_y)
        {
            for (i = local_tid_x; i < MATRIX_SIZE; i += block_row)
            {
                for (j = 0; j < MATRIX_SIZE; j++)
                {
                    if (i > 0 && i < MATRIX_SIZE - 1 && j > 0 && j <
MATRIX_SIZE - 1)
                    {
                        change = arr[i * MATRIX_SIZE + j] - 0.25 *
(arr[(i - 1) * MATRIX_SIZE + j] + arr[(i + 1) * MATRIX_SIZE + j] +
arr[i * MATRIX_SIZE + (j - 1)] + arr[i * MATRIX_SIZE + (j + 1)]);
                        // __syncthreads();
                        arr[i * MATRIX_SIZE + j] -= (change * OMEGA);
                        // __syncthreads();
                    }
                }
            }
        }
    }
```

And here is the result. We can see that it takes a lot of time, but the result is still ok.

```
-bash-4.2$ ./SOR_cuda
init all done

Initializing the arrays ...     ... done
allocate all done

GPU time: 75368.476562 (msec)
GPU calculation ended

CPU time: 101599.242188(msec)
CPU calculation ended
results check finished
Found 0 errors
```

And for the 2D tile per thread, I wrote the function as this:

```
for (int k = 0; k < SOR_ITERATIONS; k++)
    {
        if((local_tid_x == 0 && local_tid_y == 0) or (local_tid_x == 0
&& local_tid_y == 8) or (local_tid_x == 8 && local_tid_y == 0) or
(local_tid_x == 8 && local_tid_y == 8))
        {
            for(int i = local_tid_x; i < MATRIX_SIZE; i += block_col)
```

```
            {
                for(int ii = i; ii < i + 8; ii++)
                {
                    for( int j = local_tid_y; j < MATRIX_SIZE; j +=
block_row)
                    {
                        for(int jj = j; jj < j + 8; jj++)
                        {
                            if (ii > 0 && ii < MATRIX_SIZE - 1 && jj > 0
&& jj < MATRIX_SIZE - 1)
                            {
                                change = arr[ii * MATRIX_SIZE + jj] -
0.25 * (arr[(ii - 1) * MATRIX_SIZE + jj] + arr[(ii + 1) * MATRIX_SIZE +
jj] + arr[ii * MATRIX_SIZE + (jj - 1)] + arr[ii * MATRIX_SIZE + (jj +
1)]);
                                arr[ii * MATRIX_SIZE + jj] -= (change *
OMEGA);
                            }

                        }
                    }

                }
            }
        }
    }
```

And here is the result and the output. We can see that using the 2D tile, even with the GPU, it is much slower than the CPU. Although the result is ok, it is not very efficient.

```
-bash-4.2$ ./SOR_cuda
init all done

Initializing the arrays ...     ... done
allocate all done

GPU time: 306042.250000 (msec)
GPU calculation ended

CPU time: 102025.617188(msec)
CPU calculation ended
results check finished
Found 0 errors
```

Part 3: Multiple blocks

For letting each thread operate on a single output array element, I set the block_num to be 2000/16 = 125 and modify the x and y index.

And the reason that the iteration should be operate on the launching the kernel is because that when doing the iteration on the kernel, as there are multiple blocks, the GPU only chooses several blocks to operate on, and thus if the iteration is on the kernel, some of the blocks will be operated too much time and some will not be even operated.

And here is the result. We can see that it is much faster than the original SOR and the result is still correct.