CS3210 Assignment 2

Wang Chenyu & Cai Xuemeng

1. **Program's design and implementation assumptions**

Basically the program's design is: We partition the whole array to each of the threads and at every generation, let each thread of GPU calculate whether the items that it is in charge of is dead or not. We use a __device__ int death[MAX_THREAD_NUM] array to store the death number of each thread. All other computations, like file I/O, generate the initial world and move it to CUDA are all done by CPU as a single-thread workflow, after all the iterations are done, CPU collect the death number of each thread and add them together to get the output. Basically the program model we used are more or less *Parbegin-Parend* in which the paralleled *Parbegin-Parend Construct* part is done by GPU. At the beginning of every generation, the CPU launch the *Parbegin-Parend Construct* by calling the global function __global__ void execute() and use cudaDeviceSynchronize() to wait for them to finish before starting the next generation.

The implementation assumption we made is: The maximum CUDA thread number (that is, *GRID_X * GRID_Y * GRID_Z * BLOCK_X * BLOCK_Y * BLOCK_Z*) that runs the program is smaller than MAX_THREAD_NUM because the death[MAX_THREAD_NUM] array is pre-allocated before we know the exact thread number. This will not influence the performance much because it is just an int array.

2. Parallel strategy used in CUDA implementation synchronisation, work distribution, memory usage and layout, etc.

At first, we try minimising the code modification and just keep the double for loop for iterating all the rowIndex and colIndex, inside the double for loop, we try calculate whether this item is the current thread's work, if yes, compute it and store the death number. However, it turned out that because we have branching inside the double for loop, it is super slow because CUDA hates branching that can only be run one-by one inside a wrap and the double for loop makes branching part to be super large.

Therefore we modified the code to calculate the number of items that one thread needs to execute and only use one loop to iterate it, in each loop the thread locate an item and calculate that. The target rowIndex and colIndex is calculated as follows:

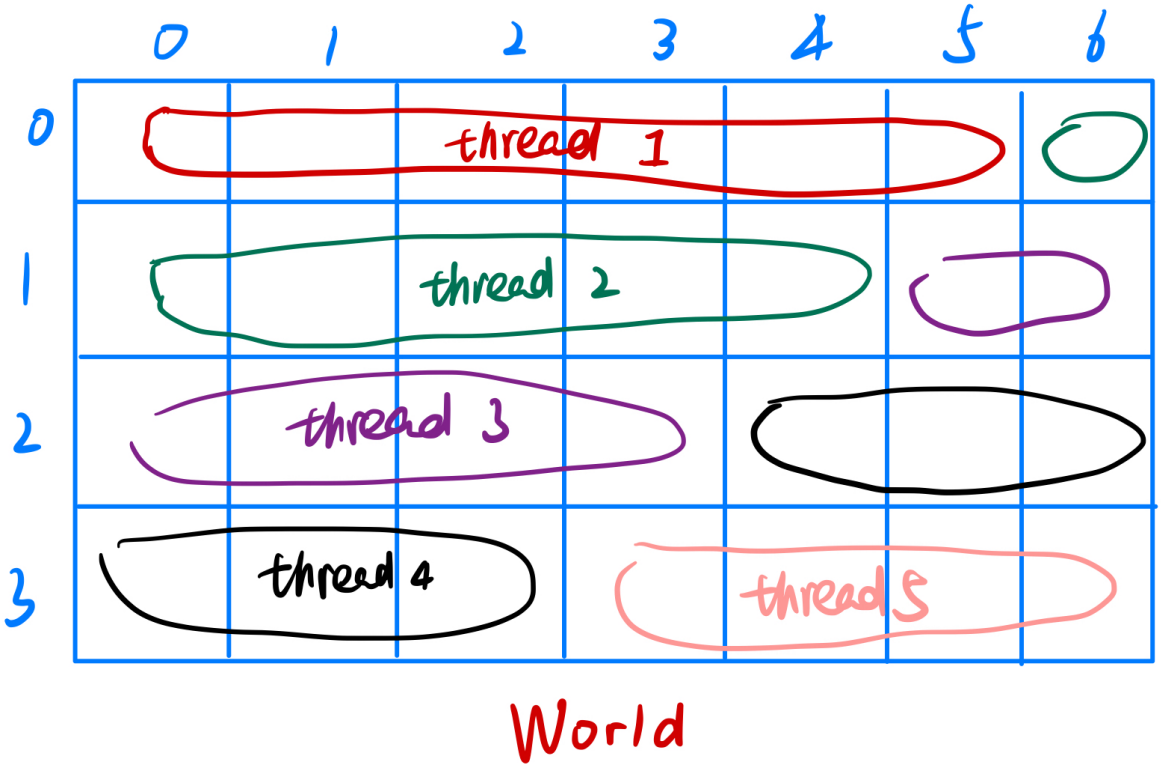
Before for loop:

```
int eachThreadWork = (nRows * nCols - 1) / num + 1;
int numTasksFinished = threadId * eachThreadWork;
int currentRowIndex = numTasksFinished / nCols; //<-Init Row Index
int currentColumnIndex = numTasksFinished % nCols; //<-Init Col Index
```

Inside the for loop, before next iteration:

```
if (currentColumnIndex == nCols - 1) {
    currentRowIndex++;
    currentColumnIndex = 0;
}else{
    currentColumnIndex++;
}
```

At the beginning of each for loop the thread will check whether the current index is valid to make the last thread stops ASAP when the work cannot be perfectly divided.



This will cause the overall partition to be like this:

And each misses will be minimised if each time a block of adjacent data are fetched because the item being processed in adjacent loops will also be adjacent in memory most of the time and therefore can be fetched in-one-go.

The only synchronisation needed is in each generation iteration, after CPU launch the kernel to calculate the death number in this generation, cudaDeviceSynchronize() will be called by CPU to wait for all the threads to finish execution before starting next iteration. Otherwise the world of next generation are not completed yet, which will cause error in next generation's calculation.

We put the initialise the world in CPU's address space and copy it into GPU, after that, the current world and next generation's world are all allocated and being processed in GPU's address space, which maximumly decrease the memory copy overhead (between CPU and GPU), thus achieves a very good performance.

3. Special consideration or implementation detail

N/A, I explained all of the important details in other parts already.

4. Details on how to reproduce your results

After cd into the project directory:

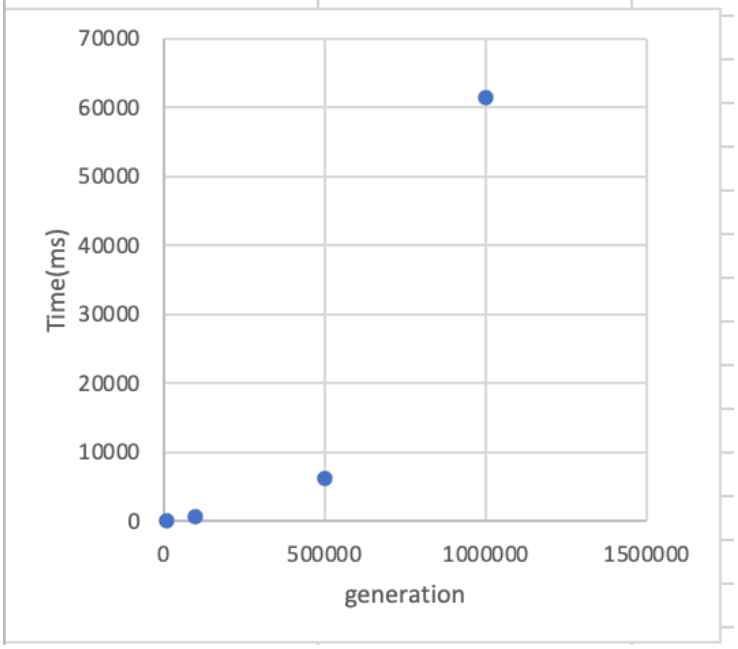
```
make
./goi_cuda.out <Path to the input file> output.out 10 10 10 8 8 8
```

Then check the output.out file to verify the result.

5. Present and explain graphs showing the execution time and speedup (y-axis) variation with world size, and grid size (x-axis) (fixed input size). Show measurements with graphs showing how the block size/grid size (task granularity) impact on the execution time and speedup:

Number of generations vs Time taken (ms) :

num of generations	Time(ms)
10000	74.374
100000	630.35
500000	6122.04
1000000	61536.5

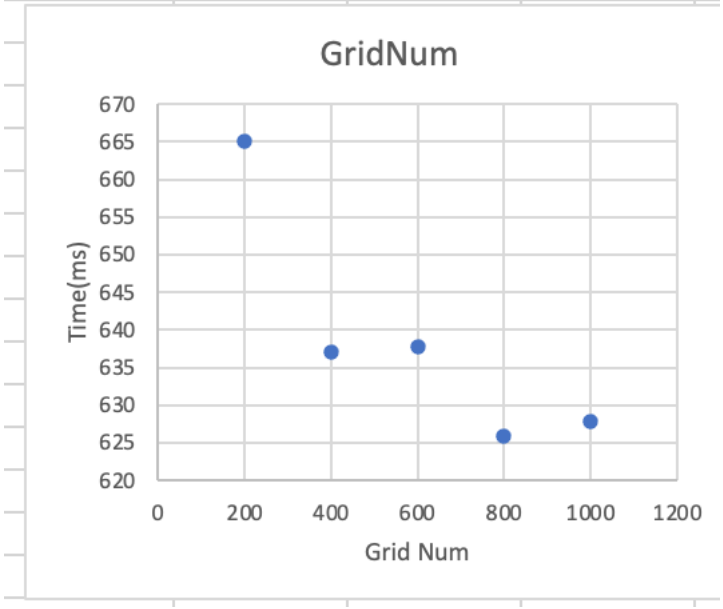


This graph shows as number of generations increases, the time taken to run the program grows exponentially. This observation makes sense because as

number of generations increases, the number of iterations will increase so that the running time will increase.

Number of grids vs Time taken(ms):

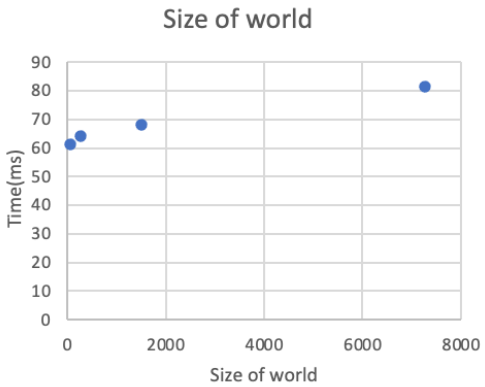
	gridNum	Time(ms)		S
	200	665.09		
	400	636.98		
	600	637.73		
	800	625.83		
	1000	627.78		



The time taken decreases as number of grids increases in general but for sometimes the time taken using small number of grids may be shorter than the time taken using large number of grids.

Size of world vs Time taken (ms)

Size	Time(ms)		
49	61.416		
266	64.146		
1500	68.083		
7272	81.537		



The time taken increases as size of world increases in general but the increases is not big.

6. Compare your CUDA implementation performance with your OpenMP implementation performance

(Use a world size of 3000 × 3000 and 10000 steps)

<The thread number / CUDA dimensions are all tuned and the below tests are run under optimal thread number / CUDA dimensions, all tests have been taken 5 times and the below screen shots are the tests that their time-taken are the very close to the average taken of 5 repeat tests.>

```
temp10@soctf-pdc-001:~/PA2-OMP$ perf stat -- ./goi_omp ./sample_inputs/sample7.in result.out 40
<INPUT_PATH>: ./sample_inputs/sample7.in
<OUTPUT_PATH>: result.out
<NUM_THREADS>: 40

 Performance counter stats for './goi_omp ./sample_inputs/sample7.in result.out 40':

      22286154.16 msec task-clock                #   18.334 CPUs utilized
          2106598      context-switches          #    0.095 K/sec
           411551      cpu-migrations            #    0.018 K/sec
         87917746      page-faults               #    0.004 M/sec
   54435773219664      cycles                    #    2.443 GHz
   77819522349397      instructions              #    1.43  insn per cycle
   13375917337532      branches                  #  600.190 M/sec
       4117113817      branch-misses             #    0.03% of all branches

   1215.540618598 seconds time elapsed

  22088.599191000 seconds user
    214.552661000 seconds sys
```

```
e0886595@xgpd5:~/pa2$ nvprof ./goi_cuda.out /home/e/e0886595/pa2/sample_inputs/sample7.in output.out 10 10 10 8 8 8
<INPUT_PATH>: /home/e/e0886595/pa2/sample_inputs/sample7.in
<OUTPUT_PATH>: output.out
==1932295== NVPROF is profiling process 1932295, command: ./goi_cuda.out /home/e/e0886595/pa2/sample_inputs/sample7.in output.out 10 10 10 8 8 8
==1932295== Profiling application: ./goi_cuda.out /home/e/e0886595/pa2/sample_inputs/sample7.in output.out 10 10 10 8 8 8
==1932295== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.99%  88.8226s     10000  8.8823ms  8.6773ms  9.1065ms  execute(int*, int const *, int const *, int, int, int)
                    0.01%  8.2092ms         2  4.1046ms  386.78us  7.8224ms  [CUDA memcpy HtoD]
                    0.00%  711.39us         1  711.39us  711.39us  711.39us  [CUDA memcpy DtoH]
      API calls:   87.04%  88.8470s     10000  8.8847ms  8.1856ms  9.5588ms  cudaDeviceSynchronize
                   10.63%  10.8475s     10001  1.0846ms  217.53us  39.752ms  cudaMalloc
                    1.88%  1.92228s     10001  192.21us  131.76us  24.441ms  cudaFree
                    0.34%  344.35ms         1  344.35ms  344.35ms  344.35ms  cudaMemcpyToSymbol
                    0.10%  105.28ms     10000  10.528us  8.5150us  691.15us  cudaLaunchKernel
                    0.01%  8.0664ms         1  8.0664ms  8.0664ms  8.0664ms  cudaMemcpy
                    0.00%  1.5556ms         1  1.5556ms  1.5556ms  1.5556ms  cudaMemcpyFromSymbol
                    0.00%  660.80us         2  330.40us  330.37us  330.44us  cuDeviceTotalMem
                    0.00%  518.90us       202  2.5680us     128ns  120.49us  cuDeviceGetAttribute
                    0.00%  60.585us         2  30.292us  24.081us  36.504us  cuDeviceGetName
                    0.00%  13.770us         2  6.8850us  3.0720us  10.698us  cuDeviceGetPCIBusId
                    0.00%  1.7340us         4     433ns     149ns  1.2630us  cuDeviceGet
                    0.00%  1.5510us         3     517ns     177ns  1.0790us  cuDeviceGetCount
                    0.00%     514ns         2     257ns     204ns     310ns  cuDeviceGetUuid
```

OpenMP: Average 1221s, using soctf-pdc-001 and we use 40 threads to run it in parallel.
CUDA: Average 88s, using xgpd5 and we set the grid dimension to be <10, 10, 10> and block dimension to be <8, 8, 8>

7. **Description of the modifications made to your code (from your baseline correct CUDA implementation) and an analysis of their impact on performance.**
   It has been described precisely under *[2.Parallel strategy used in CUDA implementation synchronisation, work distribution, memory usage and layout, etc.]*

- END -