HW3-Report

CS445

Shengyuan Wang

Sequential Version Investigation

First, I investigate the sequential version program by running the command lines below.

```
make gol_seq_prof
./gol_seq_prof -i 500 -v
```

The output will be

```
0, 1146561
50, 500606
100, 396624
150, 349394
200, 313307
250, 290900
300, 272903
350, 258330
400, 245678
450, 235231
Total Alive: 227784
total time: 36.566981 s
```

When we want to generate a profile to see which function take most time and place it into a file called profile.out by running the following command line

```
gprof ./gol_seq_prof gmon.out > profile.out
```

We will find that the profile.out will be

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 83.00     30.32    30.32      500    60.64    60.64  apply_rules
 16.90     36.50     6.18      500    12.35    12.35  update_grid
  0.11     36.54     0.04        1    40.10    40.10  init_grid
  0.08     36.57     0.03      500     0.06     0.06  init_ghosts
  0.03     36.58     0.01                             main
  0.00     36.58     0.00      500     0.00    73.05  gol
  0.00     36.58     0.00        1     0.00     0.00  getArguments
  0.00     36.58     0.00        1     0.00     0.00  isNumber
```

From the table above, we can find that we will choose `apply_rules` and `update_grid` to parallelize since it take most time and we decide to make modifications to these two functions in each separate parallel version.

We need to make a baseline for comparison with other parallel versions with the default problem size of 2048x2048, so we will use the sequential version using gcc without fast optimizations to compare.

```
make gol_seq
./gol_seq -i 500 -v
```

```
0, 1146561
50, 500606
100, 396624
150, 349394
200, 313307
250, 290900
300, 272903
350, 258330
400, 245678
450, 235231
Total Alive: 227784
total time: 34.125455 s
```

OpenMP Version

We first parallel one function to see the improvement for the program, and we decide to first parallel the function `apply_rules`.

```
int apply_rules(int *grid, int *newGrid, int dim) {
    int i,j;
    int num_alive = 0;
    int id, numNeighbors;

    // iterate over the grid

    #pragma omp parallel for default(none) \
    private(i, j, id, numNeighbors)\
    shared(dim, grid, newGrid)\
    reduction(+:num_alive)
    for (i = 1; i <= dim; i++) {
        for (j = 1; j <= dim; j++) {
            id = i*(dim+2) + j;

            numNeighbors =
                grid[id+(dim+2)] + grid[id-(dim+2)]   // lower + upper
                + grid[id+1] + grid[id-1]             // right + left
                + grid[id+(dim+3)] + grid[id-(dim+3)] // diagonal lower + upper right
                + grid[id-(dim+1)] + grid[id+(dim+1)];// diagonal lower + upper left

            // the game rules
            if (grid[id] == 1 && numNeighbors < 2) {
                newGrid[id] = 0;
            } else if (grid[id] == 1 && (numNeighbors == 2 || numNeighbors == 3)) {
                newGrid[id] = 1;
```

```
                num_alive++;
            } else if (grid[id] == 1 && numNeighbors > 3) {
                newGrid[id] = 0;
            } else if (grid[id] == 0 && numNeighbors == 3) {
                newGrid[id] = 1;
                num_alive++;
            } else {
                newGrid[id] = grid[id];
                if (grid[id] == 1) num_alive++;
            }
        }
    }
    return num_alive;
}
```

And then we using the following command to find out the result with different threads numbers(1, 2, 4, 8, 16), and same problem size of 2048x2048.

```
./gol_omp -i 500 -t 1 -v
./gol_omp -i 500 -t 2 -v
./gol_omp -i 500 -t 4 -v
./gol_omp -i 500 -t 8 -v
./gol_omp -i 500 -t 16 -v
```

And we can see a significant improvement on this program after parallelized by OpenMP.

```
numThreads = 1
    0, 1146561
   50, 500606
  100, 396624
  150, 349394
  200, 313307
  250, 290900
  300, 272903
  350, 258330
  400, 245678
  450, 235231
Total Alive: 227784
total time: 35.632020 s

numThreads = 2
    0, 1146561
   50, 500606
  100, 396624
  150, 349394
  200, 313307
  250, 290900
  300, 272903
  350, 258330
  400, 245678
  450, 235231
Total Alive: 227784
total time: 23.781640 s

numThreads = 4
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
```

```
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 16.273275 s

numThreads = 8
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 13.544054 s

numThreads = 16
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 11.928751 s
```

After make modification on another function `update_grid`.

```
void update_grid(int *grid, int *newGrid, int dim) {
    int i,j;
    int id;
    // copy new grid over, as pointers cannot be switched on the device
    #pragma omp parallel for default(none) \
    private(i, j, id)\
    shared(dim, grid, newGrid)
    for(i = 1; i <= dim; i++) {
        for(j = 1; j <= dim; j++) {
            id = i*(dim+2) + j;
            grid[id] = newGrid[id];
        }
    }
}
```

We can test the whole program after parallelizing the two most time-consuming functions.

```
./gol_omp -i 500 -t 1 -v
./gol_omp -i 500 -t 2 -v
./gol_omp -i 500 -t 4 -v
./gol_omp -i 500 -t 8 -v
./gol_omp -i 500 -t 16 -v
```

And the result turn out to be

```
numThreads = 1
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 35.478160 s

numThreads = 2
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 17.909855 s

numThreads = 4
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 9.800947 s

numThreads = 8
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 5.575188 s

numThreads = 16
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
```

```
   450, 235231
Total Alive: 227784
total time: 2.895421 s
```

From the result, we can find that after parallelizing the second function `update_grid`, the program improved better with more threads utilized.

OpenACC-Multicore

We first parallel one function to see the improvement for the program, and we decide to first parallel the function `apply_rules`.

```
int apply_rules(int *grid, int *newGrid, int dim) {
    int i,j;
    int num_alive = 0;
    int id, numNeighbors;

    // iterate over the grid
    #pragma acc parallel loop \
    private(i, j, id, numNeighbors)\
    reduction(+:num_alive)
    for (i = 1; i <= dim; i++) {
        for (j = 1; j <= dim; j++) {
            id = i*(dim+2) + j;

            numNeighbors =
                grid[id+(dim+2)] + grid[id-(dim+2)]   // lower + upper
                + grid[id+1] + grid[id-1]             // right + left
                + grid[id+(dim+3)] + grid[id-(dim+3)] // diagonal lower + upper right
                + grid[id-(dim+1)] + grid[id+(dim+1)];// diagonal lower + upper left

            // the game rules
            if (grid[id] == 1 && numNeighbors < 2) {
                newGrid[id] = 0;
            } else if (grid[id] == 1 && (numNeighbors == 2 || numNeighbors == 3)) {
                newGrid[id] = 1;
                num_alive++;
            } else if (grid[id] == 1 && numNeighbors > 3) {
                newGrid[id] = 0;
            } else if (grid[id] == 0 && numNeighbors == 3) {
                newGrid[id] = 1;
                num_alive++;
            } else {
                newGrid[id] = grid[id];
                if (grid[id] == 1) num_alive++;
            }
        }
    }
    return num_alive;
}
```

And then we using the following command to find out the result with different threads numbers(1, 2, 4, 8, 16), and same default problem size of 2048x2048.

```
./gol_mc -i 500 -t 1 -v
./gol_mc -i 500 -t 2 -v
./gol_mc -i 500 -t 4 -v
./gol_mc -i 500 -t 8 -v
./gol_mc -i 500 -t 16 -v
```

And we can see a significant improvement on this program after parallelized by Multi-core.

```
numThreads = 1
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 15.761788 s

numThreads = 2
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 9.104639 s

numThreads = 4
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 6.355166 s

numThreads = 8
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 4.484128 s

numThreads = 16
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
```

```
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 4.032817 s
```

After make modification on another function `update_grid`.

```
void update_grid(int *grid, int *newGrid, int dim) {
    int i,j;
    int id;

    // copy new grid over, as pointers cannot be switched on the device
    #pragma acc parallel loop \
    private(i, j, id)
    for(i = 1; i <= dim; i++) {
        for(j = 1; j <= dim; j++) {
            id = i*(dim+2) + j;
            grid[id] = newGrid[id];
        }
    }
}
```

We can test the whole program after parallelizing the two most time-consuming functions.

```
./gol_mc -i 500 -t 1 -v
./gol_mc -i 500 -t 2 -v
./gol_mc -i 500 -t 4 -v
./gol_mc -i 500 -t 8 -v
./gol_mc -i 500 -t 16 -v
```

And the result turn out to be

```
numThreads = 1
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 17.517594 s

numThreads = 2
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 9.081060 s
```

```
numThreads = 4
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 4.977396 s

numThreads = 8
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 2.954982 s

numThreads = 16
     0, 1146561
    50, 500606
   100, 396624
   150, 349394
   200, 313307
   250, 290900
   300, 272903
   350, 258330
   400, 245678
   450, 235231
Total Alive: 227784
total time: 1.524011 s
```

From the result, we can find that after parallelizing the second function `update_grid`, the program improved better with more threads utilized.

OpenACC-GPU

We first parallel one function to see the improvement for the program, and we decide to first parallel the function `apply_rules`.

```
int apply_rules(int *grid, int *newGrid, int dim)
// void apply_rules(int *grid, int *newGrid, int dim)
{
    int i,j;
    int num_alive = 0;
    int id, numNeighbors;

    // iterate over the grid
    #pragma acc kernels
    #pragma acc loop independent reduction(+:num_alive)
```

```
    for (i = 1; i <= dim; i++) {
        #pragma acc loop independent
        for (j = 1; j <= dim; j++) {
            id = i*(dim+2) + j;

            numNeighbors =
                grid[id+(dim+2)] + grid[id-(dim+2)]   // lower + upper
                + grid[id+1] + grid[id-1]             // right + left
                + grid[id+(dim+3)] + grid[id-(dim+3)] // diagonal lower + upper right
                + grid[id-(dim+1)] + grid[id+(dim+1)];// diagonal lower + upper left

            // the game rules
            if (grid[id] == 1 && numNeighbors < 2) {
                newGrid[id] = 0;
            } else if (grid[id] == 1 && (numNeighbors == 2 || numNeighbors == 3)) {
                newGrid[id] = 1;
                num_alive++;
            } else if (grid[id] == 1 && numNeighbors > 3) {
                newGrid[id] = 0;
            } else if (grid[id] == 0 && numNeighbors == 3) {
                newGrid[id] = 1;
                num_alive++;
            } else {
                newGrid[id] = grid[id];
                if (grid[id] == 1) num_alive++;
            }
        }
    }
    return num_alive;
}
```

And then we using the following command to find out the result in the experiment with the default problem size of 2048x2048.

```
./gol_acc -i 500 -v
```

And we can see a significant improvement on this program after parallelized by Multi-core.

```
0, 1146561
50, 500606
100, 396624
150, 349394
200, 313307
250, 290900
300, 272903
350, 258330
400, 245678
450, 235231
Total Alive: 227784
total time: 10.100377 s
```

After make modification on another function `update_grid`.

```
void update_grid(int *grid, int *newGrid, int dim) {
    int i,j;
    int id;
```

```
    // copy new grid over, as pointers cannot be switched on the device
    // #pragma acc parallel loop \
    // private (i, j, id, dim)
    #pragma acc kernels
    #pragma acc loop independent
    for(i = 1; i <= dim; i++) {
        #pragma acc loop independent
        for(j = 1; j <= dim; j++) {
            id = i*(dim+2) + j;
            grid[id] = newGrid[id];
        }
    }
}
```

We can test the whole program after parallelizing the two most time-consuming functions.

```
./gol_acc -i 500 -v
```

result:

```
0, 1146561
50, 500606
100, 396624
150, 349394
200, 313307
250, 290900
300, 272903
350, 258330
400, 245678
450, 235231
Total Alive: 227784
total time: 4.881773 s
```

From the result, we can find that after parallelizing the second function `update_grid`, the program improved better with more threads utilized.

Salability test:

In the following scalability tests, we tried the experiment with different problem sizes(2048x2048, 2896x2896,4096x4096) with different parallel methods. The reason why we set the column number as 2048, 2896, and 4096 is that we want to double the problem size in each step. And as we double the problem size, for the gpa version, the running time doubles; for parallel version, given the same threads, the running time doubles; and for the sequential version, the running time also doubles with doubled problem size.

OpenACC-GPU

```
>> ./gol_acc -i 500
Total Alive: 227784
 total time: 4.753431 s
>> ./gol_acc -i 500 -n 2896
```

```
Total Alive: 452723
 total time: 8.740919 s
>> ./gol_acc -i 500 -n 4096
Total Alive: 902642
 total time: 16.806134 s
```

OpenACC-Multicore

```
>> ./gol_mc -i 500
numThreads = 1
Total Alive: 227784
 total time: 18.002406 s
>> ./gol_mc -i 500 -t 8
numThreads = 8
Total Alive: 227784
 total time: 3.004548 s
>> ./gol_mc -i 500 -n 2896 -t 8
numThreads = 8
Total Alive: 452723
 total time: 5.611881 s
>> ./gol_mc -i 500 -n 4096 -t 8
numThreads = 8
Total Alive: 902642
 total time: 11.453195 s
```

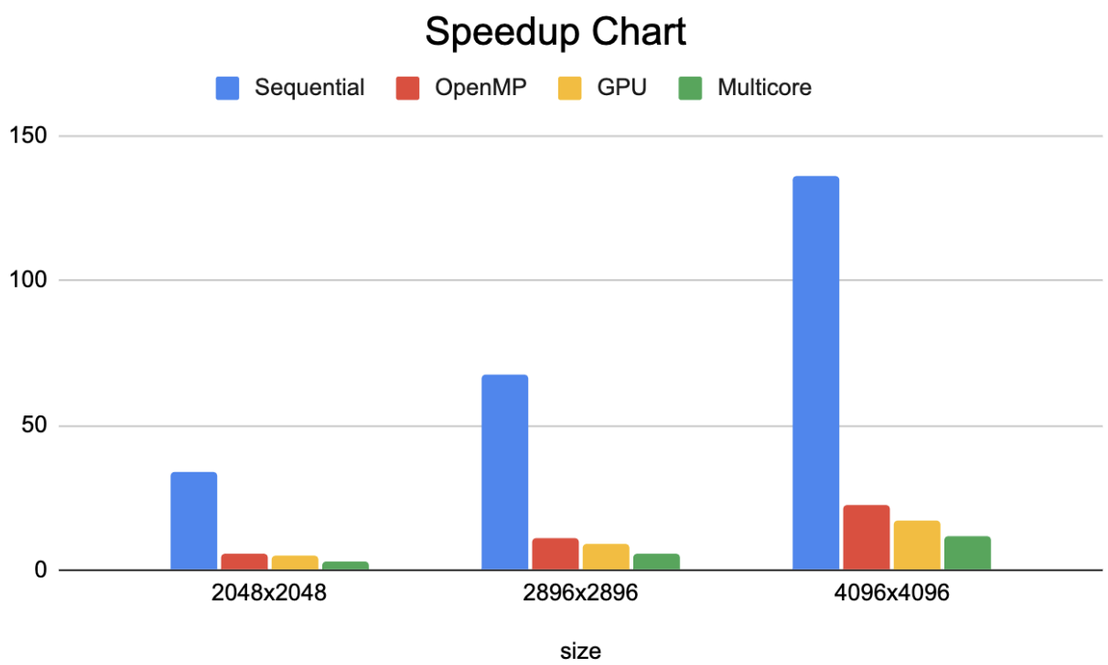
OpenMP

```
>> ./gol_omp -i 500 -t 8
numThreads = 8
Total Alive: 227784
 total time: 5.595418 s
>> ./gol_omp -i 500 -n 2896 -t 8
numThreads = 8
Total Alive: 452723
 total time: 11.127999 s
>> ./gol_omp -i 500 -n 4096 -t 8
numThreads = 8
Total Alive: 902642
 total time: 22.474517 s
```

Sequential

```
>> ./gol_seq -i 500 -n 2048
Total Alive: 227784
total time: 33.977661 s
>> ./gol_seq -i 500 -n 2896
Total Alive: 452723
 total time: 67.782562 s
>> ./gol_seq -i 500 -n 4096
Total Alive: 902642
 total time: 135.820815 s
```

We can conclude the results we get from the command lines above into the Speedup Chart below



From the graph above, we can see that after parallelize the program, the speed of the program increases tremendously. And for each problem size, we can find that the multicore performs better than GPU and OpenMP versions. This is an interesting results. It seems that the GPU card version is not faster than the optimized multicore version. When we investigate the program, we can see that the GPU computation will almost certainly involve moving data, either from the host to the devise and back, or from the device back to the host when a computation is computed. And the data set is larger in this problem, so the moving process will take much time, which finally lead to the result that the GPU version is not faster than the optimized multicore version.

Conclusion

In this report, I parallelize the game of life program using different compilers and optimizations. And the results shows that the speed of the programs: Multicore > GPU > OpenMP > Sequential. And I investigate the difference between GPU and CPU. Both CPU and GPU serve in different domains of computer processing and have different spheres of excellence, as well as limitations. GPUs can process data several orders of magnitude faster than a CPU, while CPUs have large and broad instruction sets. While individual CPU cores are faster and smarter than individual GPU cores, the sheer number of GPU cores and massive amount of parallelism will make up single-core speed difference and limited instruction sets.