GPU Programming Assignment Report

Don Wijesinghe

May 21 2024

Implementation

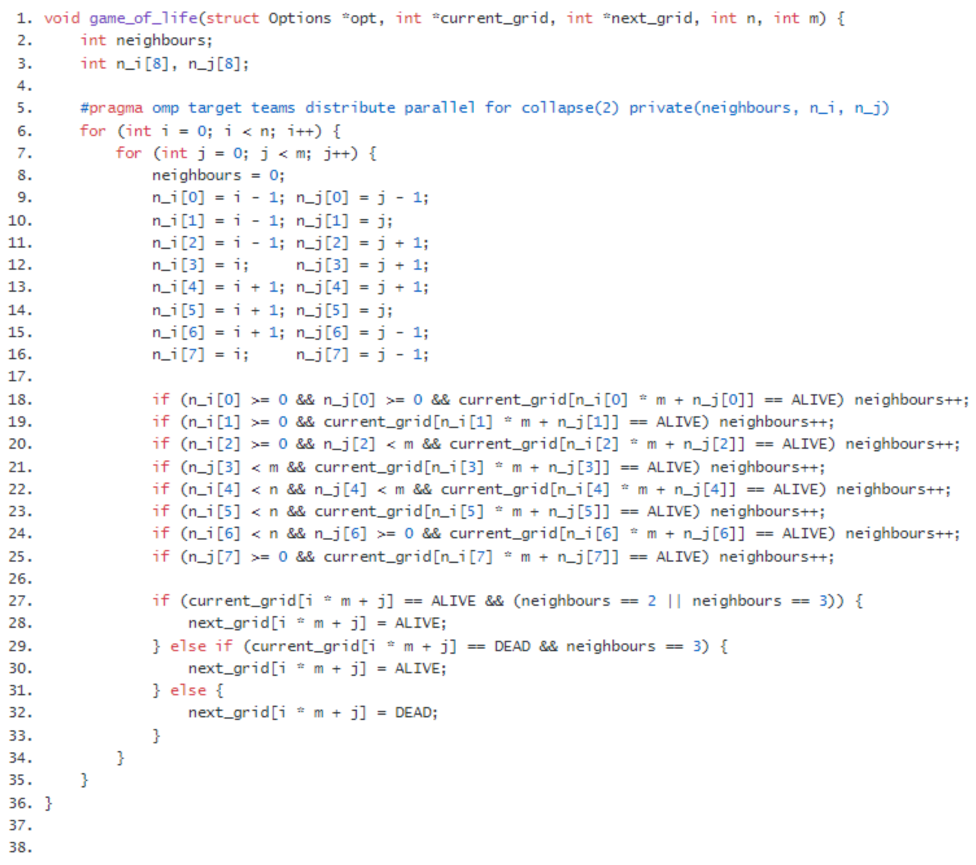
In this section we discuss two different GPU implementations of the Game of Life code from the previous assignment using HIP and OpenMP. The traditional serial implementation of the Game of Life updates the state of each cell based on the states of its eight neighbours. For large grids and time steps, this process can be computationally intensive, which makes it a suitable candidate for parallelization using a GPU.

Serial code

The serial implementation of the Game of Life is straightforward. It iterates through each cell of the grid, counts its alive neighbors, and determines its next state based on the game's rules. The *game_of_life-/cpu_game_of_life_step* functions are the most computationally expensive confirmed in the previous assignment. This approach is simple and effective for small grids but becomes inefficient for large grids due to the sequential processing of each cell as we have a time complexity expressed as O(n^2) where n is the size of one dimension of the grid. The *game_of_life_stats* function was also computationally expensive however for this assignment it is placed outside the while loop, meaning is ran only once in the entire program.

OpenMP GPU

The GPU implementation uses the parallel processing power of GPUs to significantly accelerate the computation. This is achieved by distributing the workload across multiple GPU threads. The following sections detail the key implementation decisions and strategies used. The core computation of the Game of Life is implemented in a GPU kernel. Each thread on the GPU is responsible for updating the state of a single cell in the grid. This allows the computation to be massively parallelised. The core computation of *game_of_life* is offloaded to the GPU using OpenMP target directives. Each thread handles one cell's state update.



```c
void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m) {
    int neighbours;
    int n_i[8], n_j[8];

    #pragma omp target teams distribute parallel for collapse(2) private(neighbours, n_i, n_j)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            neighbours = 0;
            n_i[0] = i - 1; n_j[0] = j - 1;
            n_i[1] = i - 1; n_j[1] = j;
            n_i[2] = i - 1; n_j[2] = j + 1;
            n_i[3] = i;     n_j[3] = j + 1;
            n_i[4] = i + 1; n_j[4] = j + 1;
            n_i[5] = i + 1; n_j[5] = j;
            n_i[6] = i + 1; n_j[6] = j - 1;
            n_i[7] = i;     n_j[7] = j - 1;

            if (n_i[0] >= 0 && n_j[0] >= 0 && current_grid[n_i[0] * m + n_j[0]] == ALIVE) neighbours++;
            if (n_i[1] >= 0 && current_grid[n_i[1] * m + n_j[1]] == ALIVE) neighbours++;
            if (n_i[2] >= 0 && n_j[2] < m && current_grid[n_i[2] * m + n_j[2]] == ALIVE) neighbours++;
            if (n_j[3] < m && current_grid[n_i[3] * m + n_j[3]] == ALIVE) neighbours++;
            if (n_i[4] < n && n_j[4] < m && current_grid[n_i[4] * m + n_j[4]] == ALIVE) neighbours++;
            if (n_i[5] < n && current_grid[n_i[5] * m + n_j[5]] == ALIVE) neighbours++;
            if (n_i[6] < n && n_j[6] >= 0 && current_grid[n_i[6] * m + n_j[6]] == ALIVE) neighbours++;
            if (n_j[7] >= 0 && current_grid[n_i[7] * m + n_j[7]] == ALIVE) neighbours++;

            if (current_grid[i * m + j] == ALIVE && (neighbours == 2 || neighbours == 3)) {
                next_grid[i * m + j] = ALIVE;
            } else if (current_grid[i * m + j] == DEAD && neighbours == 3) {
                next_grid[i * m + j] = ALIVE;
            } else {
                next_grid[i * m + j] = DEAD;
            }
        }
    }
}


```

Key directives in line 5:

target: Offloads the enclosed code block to the GPU.

teams: Creates a league of teams where each team executes a portion of the parallel loop. Each team consists of multiple threads.

distribute parallel for: Distributes the iterations of the loop across the teams, and within each team, the iterations are executed in parallel.

collapse(2): Combines the two nested loops (i and j) into a single loop, allowing better utilisation of the GPU's parallelism by creating a larger iteration space.

private(neighbours, n_i, n_j): Specifies that each thread should have its own private copy of these variables. This prevents data races and ensures correct computation since each thread works on independent cells.

```c
void game_of_life_stats(struct Options *opt, int step, int *current_grid){
    unsigned long long num_in_state[NUMSTATES];
    int m = opt->m, n = opt->n;

    for(int i = 0; i < NUMSTATES; i++) num_in_state[i] = 0;
    #pragma omp target teams distribute parallel for collapse(2) map(num_in_state) map(to:current_grid[0:n*m]) reduction(+:num_in_state)
    for(int j = 0; j < m; j++){
        for(int i = 0; i < n; i++){
            num_in_state[current_grid[i*m + j]]++;
        }
    }

```

We do a similar implementation to the *game_of_life_stats* function where we count the states of each cell in the final grid. Here we map the *num_in_states* to and from the gpu, map *current_grid* to the gpu for counting and then we perform a reduction operation to sum up the values in *num_in_state* across all threads. This function is parallelised separately to the while loop, as you can only count the states after the final grid is transferred back to the host, since we don't do any updates inside the parallel region of the while loop as seen in the next block of code

Efficient memory management is crucial for performance in GPU programming. The data needs to be transferred between the host and the device in a manner that minimises overhead.

```c
#pragma omp target enter data map(to: grid[0:n*m]) map(alloc: updated_grid[0:n*m])
    {
        while(current_step != nsteps){
                steptime = init_time();


                game_of_life(opt, grid, updated_grid, n, m);

                // swap current and updated grid only
                tmp = grid;
                grid = updated_grid;
                updated_grid = tmp;

                current_step++;
                get_elapsed_time(steptime);
        }
    }
    #pragma omp target exit data map(from: grid[0:n*m]) map(delete:updated_grid[0:n*m])

    game_of_life_stats(opt, current_step, grid);
    visualise_ascii(current_step, grid,n,m);


```

Line 1:

We map the *grid* array to the GPU, transferring its contents and allocates memory for *updated_grid* on the GPU without initialising it, outside the while loop. The reason why we don't map *current_grid* and *next_grid* in the *game_of_life* function is because, this would mean we're mapping these arrays to and from the gpu each iteration, which is unnecessary and creates communication overhead. Furthermore, we do not update the *updated_grid* array from the GPU to the host within this while loop for subsequent steps, as we only require the final state of the grid when the max iterations is reached.

Lines 10-12:

These steps are required for the compiler to know that the device also needs to swap their *updated_grid* and *grid* pointers so that the next calculation can be performed on the updated grid.

Line 18:

Once max iterations has reached, we retrieve the final updated grid back to the host to run the stats function (or any visualisation if its enabled), and delete the *updated_grid* in the device.

HIP

```
1. __global__ void gpu_game_of_life_step(int *current_grid, int *next_grid, int n, int m){
2.     int neighbours;
3.     int n_i[8], n_j[8];
4.     unsigned int idx_in = blockDim.x * blockIdx.x + threadIdx.x;
5.     if (idx_in < n*m){
6.             unsigned int i = idx_in / m;
7.             unsigned int j = idx_in % m;
8.
9.             // count the number of neighbours, clockwise around the current cell.
10.            neighbours = 0;
11.            n_i[0] = i - 1; n_j[0] = j - 1;
12.            n_i[1] = i - 1; n_j[1] = j;
13.            n_i[2] = i - 1; n_j[2] = j + 1;
14.            n_i[3] = i;     n_j[3] = j + 1;
15.            n_i[4] = i + 1; n_j[4] = j + 1;
16.            n_i[5] = i + 1; n_j[5] = j;
17.            n_i[6] = i + 1; n_j[6] = j - 1;
18.            n_i[7] = i;     n_j[7] = j - 1;
19.
20.            if(n_i[0] >= 0 && n_j[0] >= 0 && current_grid[n_i[0] * m + n_j[0]] == ALIVE) neighbours++;
21.            if(n_i[1] >= 0 && current_grid[n_i[1] * m + n_j[1]] == ALIVE) neighbours++;
22.            if(n_i[2] >= 0 && n_j[2] < m && current_grid[n_i[2] * m + n_j[2]] == ALIVE) neighbours++;
23.            if(n_j[3] < m && current_grid[n_i[3] * m + n_j[3]] == ALIVE) neighbours++;
24.            if(n_i[4] < n && n_j[4] < m && current_grid[n_i[4] * m + n_j[4]] == ALIVE) neighbours++;
25.            if(n_i[5] < n && current_grid[n_i[5] * m + n_j[5]] == ALIVE) neighbours++;
26.            if(n_i[6] < n && n_j[6] >= 0 && current_grid[n_i[6] * m + n_j[6]] == ALIVE) neighbours++;
27.            if(n_j[7] >= 0 && current_grid[n_i[7] * m + n_j[7]] == ALIVE) neighbours++;
28.
29.            if(current_grid[i*m + j] == ALIVE && (neighbours == 2 || neighbours == 3)){
30.                next_grid[i*m + j] = ALIVE;
31.            } else if(current_grid[i*m + j] == DEAD && neighbours == 3){
32.                next_grid[i*m + j] = ALIVE;
33.            }else{
34.                next_grid[i*m + j] = DEAD;
35.            }
36.        }
37.
38. }
```

Similar to before, the core computation is offloaded to a GPU kernel, with each thread responsible for updating the state of a single cell. In this kernel, each thread calculates its unique index (*idx_in*) to determine the specific cell it will update. This index is used to compute the row and column of the cell in the grid, allowing each thread to work independently on different cells. The *__global__* keyword denotes that this function is a kernel that runs on the GPU but is called from the host. The GPU architecture executes kernel functions across a grid of threads. This grid is divided into blocks, and each block contains multiple threads.

- *blockDim.x*: This represents the number of threads in a block along the x-dimension. In the context of a 1D block, this is simply the total number of threads per block.
- *blockIdx.x*: This is the block index along the x-dimension. It identifies the position of the block within the grid of blocks.
- *threadIdx.x*: This is the thread index within the block along the x-dimension. It identifies the position of the thread within its block.

The global index idx_in is calculated to uniquely identify each thread across the entire grid of blocks and threads. This index is essential for ensuring that each thread processes a unique portion of the data. This calculation is done in line 4. Once we have the global thread index idx_in, we need to map this index to the corresponding cell in the 2D grid. This is achieved in lines 6 and 7 using the modular method, since our 2D grid is in a 1D array. The if condition ensures that the thread index does not exceed the total number of cells in the grid. This is necessary because the number of threads launched can sometimes exceed the number of cells, especially when the grid dimensions (n * m) are not perfectly divisible by the number of threads per block.

```
1. int* gpu_game_of_life(const int *initial_state, int n, int m, int nsteps){
2.     int *grid = (int *) malloc(sizeof(int) * n * m);
3.    // int *updated_grid = (int *) malloc(sizeof(int) * n * m);
4.
5.     //allocate memory in gpu
6.     int* dev_grid, *dev_updated_grid;
7.     HIP_CHECK_ERROR(hipMalloc(&dev_grid, sizeof(int) * n * m));
8.     HIP_CHECK_ERROR(hipMalloc(&dev_updated_grid, sizeof(int) * n * m));
9.
10.    if(!grid){ //!updated_grid){
11.        printf("Error while allocating memory.\n");
12.        exit(1);
13.    }
14.    int current_step = 0;
15.    int *tmp = NULL;
16.    memcpy(grid, initial_state, sizeof(int) * n * m);
17.
18.    //copy initial state grid to device
19.    HIP_CHECK_ERROR(hipMemcpy(dev_grid,grid, sizeof(int) * n * m, hipMemcpyHostToDevice));
20.    //calculate nblocks
21.    unsigned int nBlocks = (n*m + NTHREADS -1) / NTHREADS;
22.
23.    float total=0;
24.    while(current_step != nsteps){
25.        current_step++;
26.        // Uncomment the following line if you want to print the state at every step
27.        //visualise(VISUAL_ASCII, current_step, grid, n, m);
28.        hipEvent_t start, stop;
29.        HIP_CHECK_ERROR(hipEventCreate(&start));
30.        HIP_CHECK_ERROR(hipEventCreate(&stop));
31.        HIP_CHECK_ERROR(hipEventRecord(start));
32.        //run gpu kernal
33.        gpu_game_of_life_step<<<nBlocks, NTHREADS>>>(dev_grid, dev_updated_grid, n, m);
34.
35.        HIP_CHECK_ERROR(hipGetLastError());
36.        HIP_CHECK_ERROR(hipEventRecord(stop));
37.        HIP_CHECK_ERROR(hipDeviceSynchronize());
38.
39.        float elapsed;
40.        HIP_CHECK_ERROR(hipEventElapsedTime(&elapsed, start, stop));
41.        printf("'game_of_life' kernel execution time (ms): %.4f\n", elapsed);
42.
43.        total += elapsed;
44.        // swap device pointers
45.        tmp = dev_grid;
46.        dev_grid = dev_updated_grid;
47.        dev_updated_grid = tmp;
48.
49.
50.        HIP_CHECK_ERROR(hipEventDestroy(start));
51.        HIP_CHECK_ERROR(hipEventDestroy(stop));
52.    }
53.
54.    HIP_CHECK_ERROR(hipMemcpy(grid, dev_updated_grid, sizeof(int) * n * m, hipMemcpyDeviceToHost));
55.    HIP_CHECK_ERROR(hipDeviceSynchronize());
56.
57.   //visualise(VISUAL_ASCII, current_step, grid, n, m);
58.
59.    printf("Total Kernal time (ms): %.4f\n", total);
60.    HIP_CHECK_ERROR(hipFree(dev_updated_grid));
61.    HIP_CHECK_ERROR(hipFree(dev_grid));
62.    //free(updated_grid);
63.    return grid;
64. }
```

The *gpu_game_of_life* function is where we manage the memory transfers for both GPU and the host. The *hipMalloc* function is used to allocate memory on the GPU for both the current grid (*dev_grid*) and the next grid (*dev_updated_grid*). This ensures that the GPU has its own memory space to work with, independent of the host's memory. This is done in line 7 and 8. The initial state is then transferred from the host to the device using *hipMemcpy*. We then calculate the number of blocks using the max number of threads per block (1024) and the problem space size(n * m). The Game of Life iterations then begins, and the kernel is launched with a specified number of blocks and threads per block in line 33. This parallelises the computation across many GPU threads and updates the state of the grid from *dev_grid* to *dev_updated_grid*. *hipGetLastError()* checks if there were any errors during the kernel launch. *hipDeviceSynchronize()* ensures that all previous operations on the device have completed before moving on. This is crucial to make sure that the kernel has finished executing before measuring the elapsed time. Hip events are used for timing the kernel execution time and also the total time spent on kernel calculations is also calculated. Again, we do not copy back the updated grid to the host due to unnecessary overhead caused by communication, and so we do the entire calculation inside the gpu. The device pointers *dev_grid* and *dev_updated_grid* are swapped to prepare for the next kernel launch. This ensures that the next kernel launch will read from the updated grid. Once the the number of steps are reached, we copy the final state back to the host by performing another *hipMemcpy* and then return the final grid.

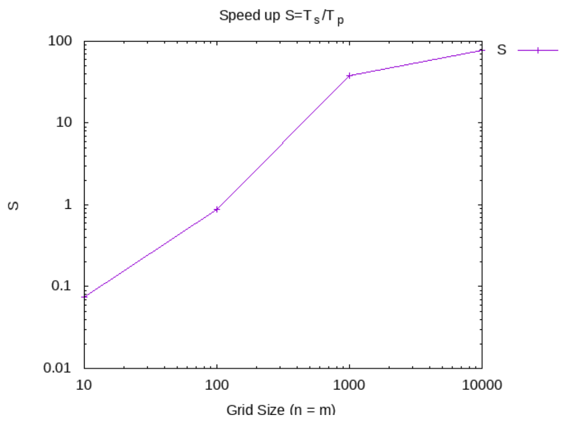
Correctness of results

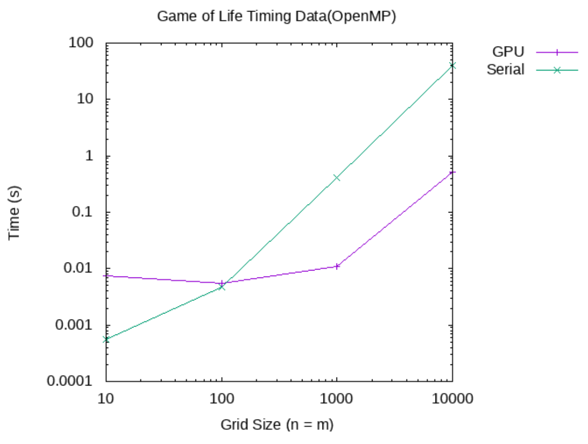
Ensuring the correctness of results between the GPU and serial implementations of the Game of Life involves verifying that both versions produce the same output for the same initial conditions over a given number of iterations. This was done by enabling the *visualisation* function and comparing the outputs for serial and gpu versions. The dimensions were also changed such that $n \neq m$, to test outputs. For higher chunk sizes where visual inspections cannot be done, the *diff* command was used to compare the output files.

Performance

OpenMP

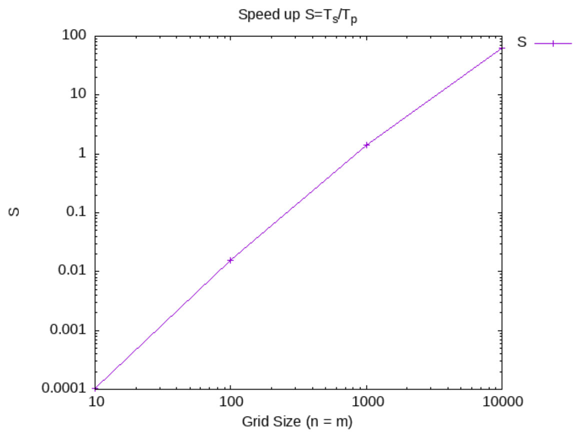
Grid size	T_s(s)	T_p(s)	S
n=m=10	0.000559	0.007462	0.074913
n=m=100	0.004781	0.005470	0.87404
n=m=1000	0.408753	0.010881	37.56576
n=m= 10000	40.603225	0.517145	78.5142
n=1000, n=10000	4.056893	0.056992	71.18355

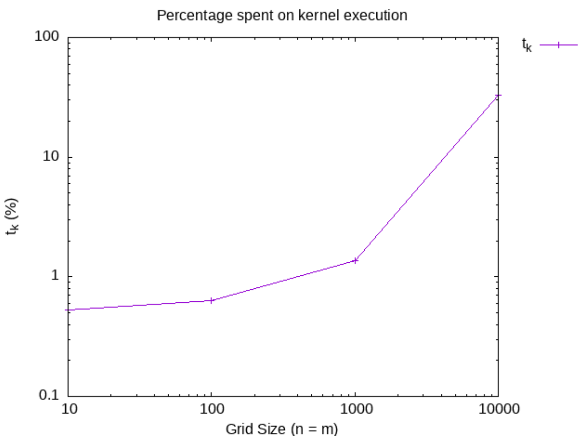


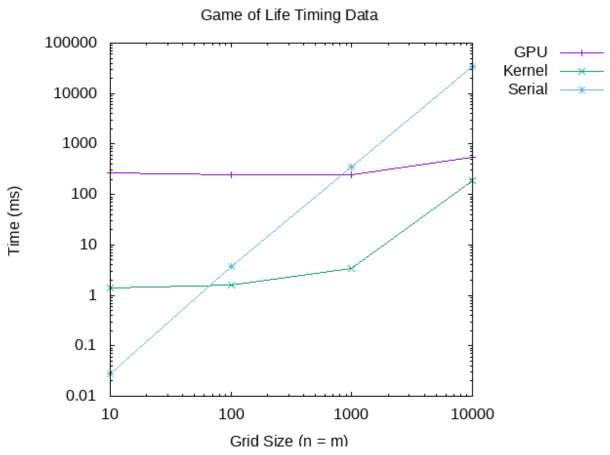


HIP

Grid size	T_s(ms)	T_p(ms)	t_k(%)	S
n=m=10	0.028000	270.072998	0.549	0.000104
n=m=100	3.811000	247.182007	0.641997	0.015418
n=m=1000	350.582001	250.962006	1.378137	1.396952
n=m= 10000	34969.445312	554.445007	33.66164	63.07108
n=1000, n=10000	3489.573975	260.820007	7.651522	13.37924







Both HIP and OpenMP show significant speedups at larger problem sizes compared to serial performance. For small problem sizes (e.g., n=m=10), both implementations perform poorly with HIP showing particularly low speedup due to high overhead in communication. Speed up for OpenMP slows down for higher problem space sizes, although more data points are needed to confirm this. Although, the speedup values for the highest chunk sizes in our data shows a higher speed up value for the OpenMP case with 78 compared to 63 for HIP, the trend in the speed up graphs indicate that it will outperforms OpenMP at very large grid sizes ensuring better scalability. For applications with smaller problem sizes, OpenMP might be preferable due to its lower overhead. We also see that the kernel execution time increasing more significantly at higher chunk sizes. Considering the kernel is the most computationally intensive, other optimisation must be done to the code since percentage of time spent on the *gpu_game_of_life_step* function is relatively low. One notable outcome was that for $n \neq m$, the OpenMP version outperformed the HIP version significantly as speed up values are not comparable.

Conclusion

In this report, we compared the performance of the Game of Life implementation using HIP and OpenMP for parallel execution. Our findings indicate that both OpenMP and HIP have similar performance, however for $n \neq m$ grid sizes, OpenMP outperforms HIP, this is likely due to using collapse(2) in OpenMP being more efficient at parallelising the computation across the GPU threads. For very large grid sizes, HIP shows better scalability, while for lower chunk sizes OpenMP might be better. Therefore, the choice between HIP and OpenMP should be guided by problem size and specific performance requirements, with OpenMP being more suitable for smaller to moderate grids and HIP for very large grids, provided optimization challenges are addressed.