Final Project: Game of Life

Logan Hunt

December 8, 2021

1 Description

From Wolfram MathWorld:

> A cellular automaton is a collection of "colored" cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied iteratively for as many time steps as desired.

Conway's Game of Life is one such automaton. In the Game of Life, the rules for each cell are as follows (from Wikipedia):

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

To help visualize this automaton I created a script to go through the output of my Game of Life simulation and compile a video with ffmpeg. As an example, I've uploaded the output of a simulation with a 1920x1080 grid of cells with 1000 iterations to YouTube. Each cell that is white is alive and each black cell is dead.

There are four implementations of Conway's Game of Life in this project; a serial implementation, a distributed memory implementation (in OpenMPI), a shared memory implementation (in OpenMP), and a GPU implementation (in Cuda).

A timing study is performed on each implementation by calculating the elapsed time of the program given varying sizes of initial grids and, in the shared and distributed memory versions, a different number of cores. In each, both the time it takes to compute the next iteration and the total wall clock time are measured.

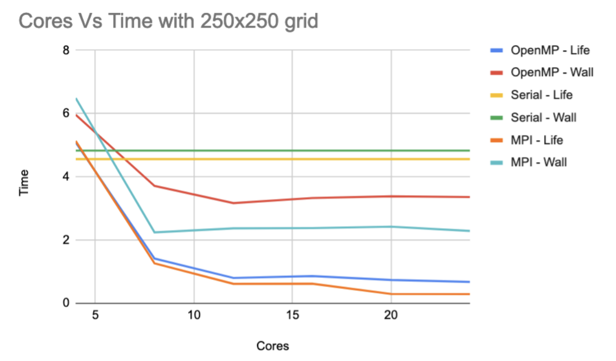
2 Performance analysis (of Game of Life iteration time)

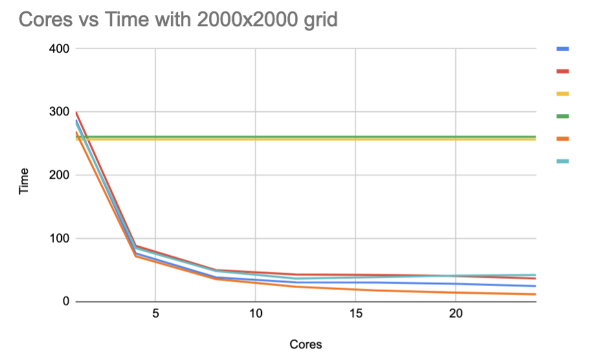
Results can be found on a Google Sheet

2.1 Shared memory vs distributed memory

2.1.1 Runtime

In runtime, both implementations have the same property of decreasing over an increasing number of cores in all problem sizes (as one would certainly hope). As the problem size increases, the overall differences in the runtimes of each implementation also decreases; meaning they follow the same trends. This can be shown in the runtimes for both implmentations running on a small grid and a large grid:

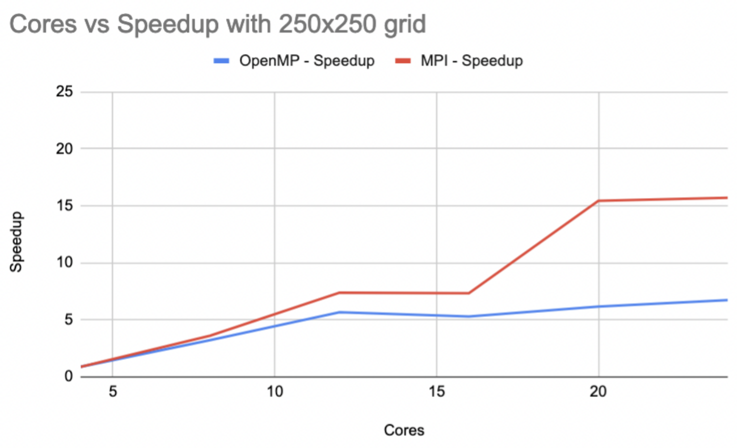


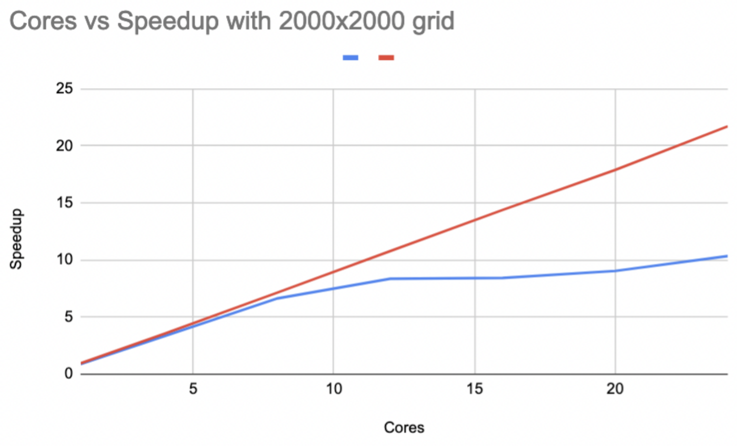


Both seem to converge to some rational function. Using an online regression calculator it was found that the MPI Life Computation (iteration computation time only) runtime follows the function $t(p) = \frac{274.449}{p^{0.985}}$ with a correlation coefficient of $r = -0.999892441$. Since t is very close to being a rational function of p, we know that the runtime fits to what could be expected: $T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}$.

2.1.2 Speedup

In speedup, both implementations tend to increase over an increasing number of cores in all problem sizes. However, it doesn't strictly increase. With some numbers of cores in the shared memory implementation, the speedup actually decreases from its predecessor.





2.1.3 Efficiency

Efficiency is the ratio of speedup to p processors ($E = \frac{S}{p}$), so it can be thought of as the derivative of the speedup. Thus efficiency can be measured without plotting it explicitly.

By definition, a program is "strongly scalable" if it can keep its efficiency constant over a varying input size. In the results, it can be seen that the slope of the Distributed Memory Life Computation Time line tends to be constant, meaning that the efficiency is also constant. Thus, the MPI version is strongly scalable.

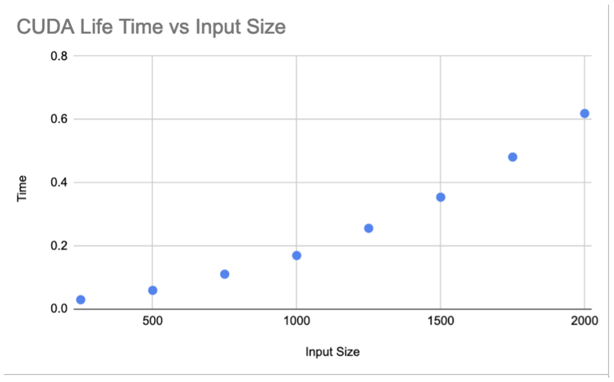
However, the shared memory (OpenMP) implementation does not seem to be perfectly strongly scalable. As the problem size varies, the speedup does not follow a constant slope. Instead, it tends to match the efficiency of the MPI implemenation until some point where the slope drops off.

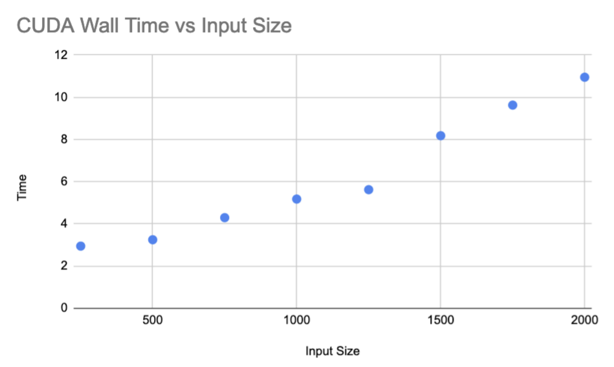
Theoretically, the OpenMP implementation should be just as strongly scalable as the MPI implementation. One reason overhead could be present is in thread scheduling.

2.2 CUDA Implementation

2.2.1 Runtime

For the CUDA implementation, different grid sizes are used to measure the iteration time as well as the wall time. Again, 1000 iterations are used for the timing study.





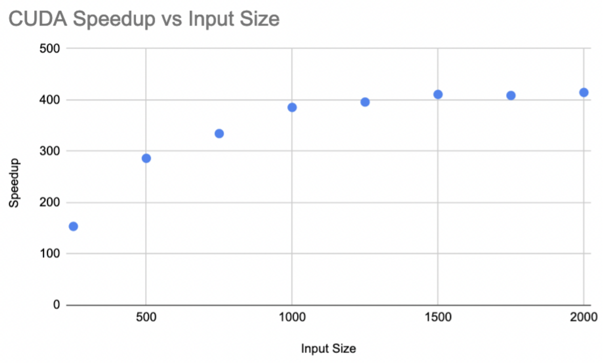
Using an online regression calculator again, it was found that the runtime as a function of input size can be expressed with by $t(n) = (1.486)(10^{-7})n^2 + (1.328)(10^{-6})n + 0.02151$ with a correlation coefficient $r = 0.9999278678$.

Since the number of cores is constant, we would hope to see a quadratic increase in the runtime as the input size grows. This is because the number of cells increases with (input size)^2.

Indeed, this is what we see.

2.2.2 Speedup

The speedup of the cuda implementation as input size increases tends to follow a logarithmic curve, plateuing after around $n = 1000$. While I am not entirely sure why it follows this trend, I guess it might have to do with the warp scheduling.



2.2.3 Efficiency

Since the core count on the K80 is constant (4992 CUDA cores), the efficiency can be calculated by $E = \frac{S}{4992}$. As the efficiency is just a constant multiplied by the speedup, the efficiency graph will just be a scaled version of the speedup graph. As such the efficiency will not be constant over different input sizes since the speedup isn't, and thus the CUDA implementation is not strongly scalable.