Jeongbin Son

CS 481 High Performance Computing

HW 5 – GPU Game of Life Implementation

Nov 20th, 2024

Introduction

This report is about Conway's Game of Life and the multiple implementations possible for the game, specifically for a GPU based architecture. Previously, for this CS 481 course, I worked on CPU based versions of GoL using OpenMP and MPI, which taught me of the importance of parallelization in the performance of a program. HW 5 extends upon what we have learned throughout the semester and introduces a GPU based version of GoL which takes advantage of the asax cluster's GPU architecture for parallelism. The rest of the paper will explore the GPU based implementation, the testing involved, and the comparison and performance evaluations of the GPU version against the CPU versions.

Previous Game of Life Implementations

This course explored implementing GoL with CPU based architectures before this homework, utilizing techniques like OpenMP as well as MPI. This section will present the methods used for the two mentioned CPU versions of GoL as well as illustrate the data collected during the previous projects.

OpenMP:

The OpenMP version of the Game of Life program used multiple threading techniques to distribute the computation across multiple CPU cores, thus improving performance for medium to larger sized boards of GoL. This program initialized the board with a fixed random seed (srand(12345)) which would allow reproducibility as well as allow for comparisons between this implementation with other versions (MPI and GPU and GPU+) by checking that the final board state is the same. OpenMP's #pgrama omp parallel for directive parallelizes the initialization loop which allows the multiple threads to fill the grid simultaneously.

The primary computations were performed within the nextGeneration function of the code, which calculated the state of each cell based on basic GoL rules. Additionally, even counting the neighbors were parallelized in this version, where threads would independently process cells in their assigned regions. Because the threads operated independently, this ensured a higher level of efficiency overall. The program was also coded to terminate when it reached the maximum value for generations/iterations (given by the command line argument) or if two generations side by side produced no differences within the two boards, indicating that the two generations were identical using isBoardSame. The final board state was also represented in the output file as '*' for alive and '.' for dead.

By using OpenMP, this version of GoL showed a scalable solution that could solve medium to large sized boards fairly well (as compared to a serialized version). However, the shared memory model that OpenMP has limits it for very large board sizes / higher number of max generations. Below is the data collected for the OpenMP version:

Threads	Board Size	Max Generation	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Average Time (ms)	Relative to 1 Thread	Efficiency (%)
1	5000	5000	5464663	5456814	5454084	5,458,520.33	1.00	100%
2	5000	5000	2726971	2725436	2724320	2,725,575.67	2.00	100%
4	5000	5000	1360945	1363962	1362538	1,362,481.67	4.01	100%
8	5000	5000	681703	682079	681328	681,703.33	0.80	10%
10	5000	5000	872813	858387	858219	863,139.67	0.63	6.3%
16	5000	5000	682654	683173	685276	683,701.00	0.80	5%
20	5000	5000	764618	746965	747271	752,951.33	0.72	3.6%

The data collection for OpenMP shows that there is a definitive decrease in execution time as the number of threads increased, demonstrating the scalability of the CPU implemention for the GoL. With one thread, the average execution time was 5,458,520.33 ms, which is halved when using two threads and further reduced down to 1,362,481.67 ms with four threads. However, as the thread count increased beyond eight, the efficiency is shown to drop by a lot, showcasing that OpenMP suffers from diminishing returns potentially due to some overheads happening. The data shows that the OpenMP version I had had the best performance at four threads, as shown by the 100% efficiency, while higher thread counts such as 20 suffered lower efficiency. I can conclude from my data collection that OpenMP's shared memory doesn't really scale well past a certain number of threads used. Thus, it's important to balance out the thread count to gain the best possible performance boost from using OpenMP.

MPI:

The next implementation for GoL utilized MPI, where the work was distributed across multiple processes instead of threads this time. This version leveraged a row wise partitioning for the board as well as inter process communication to achieve parallelism for the program. Here, each process handled a subset of rows from the game board, with additional ghost rows, facilitating communication between the neighboring processes. The ghost rows were to ensure correctness of neighbor calculations for the edge cells.

To update the board, the MPI version used MPI_Sendrecv, where the bottom row of one process was sent to the top ghost row of the next, and vice versa. The nextGeneration function from OpenMP was used here as well, which calculated the state of each cell independently for its assigned rows, excluding ghost rows. Additionally, the code detected early termination by combining local convergence results by using MPI_Allreduce, which checked that all processes agreed on when the board has reached a stable state (two consecutive boards did not have ANY change, therefore the program can now stop). The final results were also output to the file, but the results were gathered by using MPI_Gatherv, which assembled and wrote the complete board out.

By distributing to different processes & overlapping communication, the MPI version of GoL also worked to solve many different board sizes as well as higher number of generations/iterations like OpenMP, but with different advantages and disadvantages. However, the extra communication is an overhead which the data shown below explains how efficiency reduces for smaller board sizes or when the number of processes is too high relative to the amount of work needed. MPI version also taught me of the importance of balancing between the tradeoffs for getting the most efficient program. Below is the data collected for the MPI version:

Processes	Board Size	Max Iterations	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Average Time (ms)	Relative to 1 Process	Efficiency (%)
1	5000	5000	5562330	5559815	5561768	5561304.33	1.0	100%
2	5000	5000	2788387	2790778	2782445	2787203.33	1.99	99.8%
4	5000	5000	1393548	1418454	1403913	1405305	3.96	98.9%
8	5000	5000	696107	696691	697422	696740	7.98	99.8%
10	5000	5000	556488	556975	555983	556482	9.99	99.9%
16	5000	5000	348894	349249	349463	349202	15.93	99.5%
20	5000	5000	279484	279589	279355	279476	19.90	99.5%

The data collection for the MPI version of GoL shows a consistent decrease in the execution times as the number of processes increased, demonstrating the scalability of the distributed memory implementation. With one process, the average execution time was 5,561,304.33 ms, which was nearly halved when using two processes and further reduced to 1,405,305 ms with four processes. Unlike OpenMP, the efficiency for MPI remains very high, with values above 99% even as the number of processes increases to 20, showcasing the effectiveness of MPI in handling larger board sizes / iterations. This indicates that MPI's distributed memory model can greatly minimize overheads that OpenMP suffered and communication delays, even after pushing a high number of threads at once.

OpenMP V.S. MPI Version:

The data collection shows how OpenMP and MPI perform relatively well for smaller threads and number of processes. However, MPI outperformed OpenMP as the level of parallelism increased (with higher threads/processes). MPI shows a linearized speedup up to the 8 processes edge case, highlighting MPI's ability to parallelize without much diminishing returns as compared to the drop in efficiency for OpenMP at 8 threads. Overall, MPI is shown to be a great solution for large tasks that demand such parallelization.

HW 5 GPU Implementation

The GPU version of the Game of Life leverages CUDA for Nvidia's GPU to perform computations in parallel across many threads. This implementation assigns one thread to handle each cell of the game board. The core functionality is within the CUDA kernel, which for our program is gameOfLifeKernel, which is set to update the state of the board by calculating the next state for each cell based on their neighbors.

Each thread in the kernel is in charge of processing a single "cell", where its position is determined by the thread and block indices. The gameOfLifeKernel computes the number of neighbors that are alive for the cell by iterating over the 8 surrounding cells, while ensuring that there aren't issues with accessing memory out of bounds. Of course, the same GoL rules as the other projects are applied for this project as well. Lastly, the results are written onto a final board output like the previous projects.

To initialize the board fairly, the program also generates the random state using the same fixed seed as previous projects (srand(12345)). The CPU will allocate memory for the board, initialize it, and then copies the data from the CPU to the GPU by using cudaMemcpy. Then the board is processed with blocks of 16 by 16 threads, where the grid dimensions are calculated based on the board size (from command line) to ensure no data is left behind. The kernel executes for the value of max generations or until the boards become stable for this project as well. After all generations are processed, the final board state is output to a file, to ensure that we can confirm if the program was outputting the same boards as the other programs. Likewise, * (alive) and '.' for dead cells are still in use for this implementation.

Performance Results of GPU Version

CPU / GPU	Board Size	Max Generations	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Average Time (ms)
GPU	5000	5000	1660	1650	1661	1657.00
GPU	10000	5000	6360	6350	6347	6352.33
GPU	15000	15000	46072	46078	46076	46,075.33
GPU	25000	10000	85854	85855	85852	85,853.67

The data collected for the GPU version shows a consistent increase in execution time as the board size grew but still demonstrated that the GPU can handle large computations effectively. For the 5000x5000 board with 5000 max generations use case, the average execution time came out to be 1657 ms, which is significantly faster than OpenMP and MPI versions. I checked the validity of my GPU version by comparing the final output board generated between the GPU version to the OpenMP / MPI versions. As the board size increased to 10,000x10,000 with 5000 max generations, we can see that it took much longer for this use case compared to the first use case mentioned. However, 6352.33 ms is still very fast for such a large task. In comparison, OpenMP took 752,951.33 ms to compute 5000x5000 for 5000 generations while MPI took 279476 ms even with the lowest average time recorded (20 processes concurrently). This data showcases the GPU's ability in processing large scale problems extremely fast compared to the CPU architecture implementations. But I would note that this is definitely limited to the cluster's memory limits and GPU utilized.

GPU+ Optimized Code

This project required optimizing the initial GPU version, so I chose to call this GPU+. GPU+ builds upon the final version of GPU code written but introduces 'tiling' optimizations to hopefully enhance the performance of the program, especially for larger board sizes and iterations. GPU+ uses a TILE_SIZE x TILE_SIZE method within each thread block as I had hoped that this method would improve memory access and thus reduce the overhead and latency of the GPU version.

For GPU+, CUDA kernel processes the board in tiles now rather than individual cells as opposed to GPU. Each thread within a block is assigned a tile, and it goes through its cells using nested loops. By doing so, the number of global memory accesses are reduced. The same GoL rules are applied for GPU+ still, of course. The same file output is utilized as well.

The ‘tiling’ optimization is used in the grid / block configurations of the GPU code. The number of threads per block (by dim3 threadsPerBlock(16,16)) remains the same, but now the effective computation area per thread is expanded by another factor of TILE_SIZE. The grid dimensions are adjusted to ensure complete coverage of the board by accounting for the tile size in the calculation of blocksPerGrid. This reduces the kernel overhead for GPU+ as fewer threads are now needed to cover the same board size from the GPU version.

The GPU+ version over the GPU version will show that its performance is slightly better than GPU with the data collection for GPU+ below. It’s not much, but with an extremely large board, the difference between the two versions begins to show. Unfortunately, I could not get a better optimization for GPU version but still showed some improvements to execution time. Later, this paper will discuss possible further optimizations that could be made to the code.

Performance Results of GPU+ Version

CPU / GPU	Board Size	Max Generations	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Average Time (ms)
GPU+	5000	5000	1708	1663	1658	1,676.33
GPU+	10000	5000	6364	6378	6373	6,371.67
GPU+	15000	15000	45699	45681	45663	45,681.00
GPU+	25000	10000	84880	84806	84812	84,832.67

GPU+ data showcases that it’s similar to the GPU data but still slightly better average execution times than the GPU version by using the tiling optimizations. For GPU+’s 5000x5000 board with 5000 generations, the average time taken was 1,676.33 ms compared to 1657.00 ms for GPU, which is a negligible difference. However, increasing the board size and generations to 15,000x15,000 with 15,000 generations shows that there starts to be a difference between the GPU and GPU+ versions in terms of execution speeds, with GPU+ having a shorter time to compute. As the board size and generations continue to increase for the last case, it’s clear that the GPU+ version is consistently better than GPU, considering we took multiple test cases and averaged the numbers for reliability.

GPU vs GPU+ vs OpenMP vs MPI for GoL

The data that was collected throughout multiple homework's has shown that there are definite differences between the execution times and the scalability of the OpenMP, MPI, GPU, and GPU+ implementations of the GoL. Each method has its own advantages and disadvantages, depending on the size of the computation and other parameters.

Overall, the GPU and GPU+ versions did outperform the OpenMP and MPI versions by a large margin for average execution times. While the GPU versions utilized CUDA to parallelize the number of cells, OpenMP / MPI suffered overheads when trying to manage the different threads and processes simultaneously. Additionally, GPU+ went further and had tiling optimizations done as well, which further improved its performance for large board cases. MPI did however demonstrate that it was efficient in scaling to a higher number of processes, maintaining over 99% efficiency across all test cases at least up to 20 processes. But OpenMP suffered with diminishing returns as we saw in the data collection starting beyond 8 threads. OpenMP / MPI relied on distributed memory, which might have caused overhead with communicating to processes and threads, whilst the GPU version relied on global memory. As the data showed, the GPU version was very efficient with its memory that outperformed the CPU based implementations.

Overall, OpenMP was suitable for smaller tasks with moderate amounts of parallelism, and MPI shined for larger workloads while keeping a linear scalability throughout the different number of processes. The GPU versions were the best out of all the implementations that were done to handle extremely large-scale computations due to the parallelism that the GPU can handle. The potential future improvements to the GPU and GPU+ implementations I would suggest would be looking into bitmap usage for better memory efficiency.

Conclusion

In the end, this report has talked about the previous implementations for the GoL for CS 481, High Performance Computing as well as the newest version utilizing GPU architecture. After going through the data collection and analysis, as well as explaining the implementations/methods used for CPU based OpenMP and MPI as well as GPU based CUDA version, this paper can conclude that GPU programming has a great performance for higher degrees of parallelism. This paper also discussed potential improvements, but those improvements were not tested (for after GPU+).