

Project - Parallel Wave Function Collapse Terrain Generation using CUDA and MPI

COMP 4901Q: High Performance Computing (HPC) Spring 2022

CHEUK NAM TSANG

HKUST

cntsangaa@connect.ust.hk

1. Introduction

Procedural content generation means methods of generating content based on set rules. Commonly used in video games such as Minecraft and Crypt of the NecroDancer, procedural generation of game environment can allow game developers to design a set of rules that can generate unique permutations of environment for each player without manually designing all of the environment features.

Wave Function Collapse Algorithm is one of the tools for generative design of video game level environment such as terrains [Moreau 2020]. The algorithm is similar to a game of Sudoku. In an empty 9x9 board, all tiles can be all of the 9 numbers. When one number is placed, tiles on the same row, same column or same 3x3 square can no longer be assigned the same number.

Following are the procedures for the algorithm:

1. Initialize the input arena of size $N \times N$ individual cells
2. Create an array w to list all valid states
3. Initialize a boolean array for each cells with has the same size as array w the that indicates whether the state is valid or forbidden.
4. Pick an arbitrary cell and randomly set it to a valid state. Usually, cells with low number of possible states are prioritized to prevent conflicts.
5. Recalculate valid states for all cells.
6. Repeat from step 4 until all cells has defined state.

2. Related works

In 2019, a team of students attempted to parallelize the algorithm using different approaches [Orlowski and Lee 2019]. They used OpenMP and P-Threads in combination with Queues. Their results showed that the algorithm can be parallelized with relatively high scaling efficiency. However, it also revealed that an observer-based approach with sequential queue for execution shows the most speedup when compared to parallelizing the work on all cells.

In this project, I aim to investigate parallelizing the wave function collapse algorithm using CUDA since CUDA has many more cores compared to multiprocessing on CPU, the brute-force approach on CUDA may still be beneficial over CPU with a faster algorithm.

Since the next iteration is based on changes from adjacent cells, this is very similar to Conway's game of life [Games 1970], where the livelihood of cells in the next iteration are entirely based on the cells surrounding it in the current iteration. Implementation of this game using image convolution and

utilizing GPU shader processing units has been proven possible [Ru 2017]. This project also tries to utilize multiple GPU by using MPI, which can further speed up the process.

3. Target Platform

This project targets a multi-GPU cluster, and the problem size will need to be big in order to benefit from parallelization. The cluster is expected to be able to communicate with each other using MPI and each servers should have CUDA capable GPU so that the CUDA program can be executed.

4. Application Objectives

This program will not have a graphical user interface or graphical representation of the results other than being able to export to bitmap.

For simplicity, all cells can be allocated an integer from 0 to 9 and only one rule will apply. Adjacent cells can have a difference of at most 1, meaning cells surrounding a cell with state 8 can only has the value 7, 8 and 9. This also makes propagating the states faster since Manhattan distance can be used to limit the range of propagation.

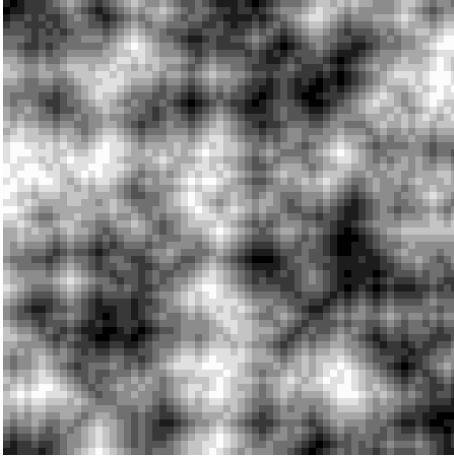


Figure 1. Generated height map.

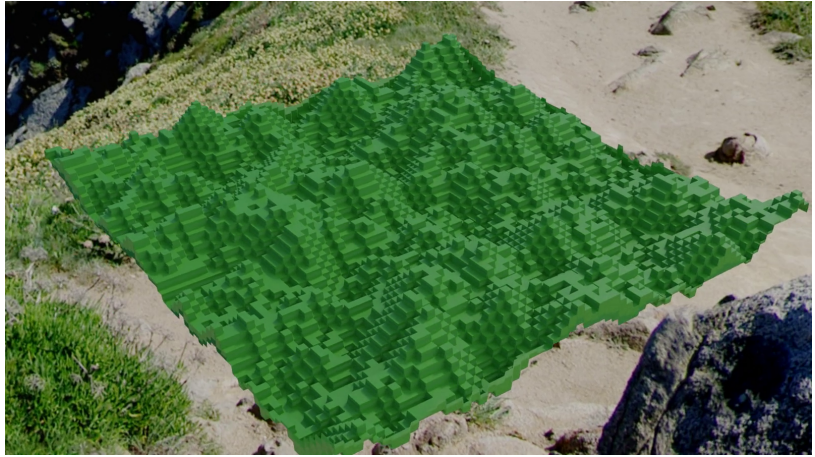


Figure 2. Generated terrain rendered in a 3D environment.

This rule set simulates generation of mountain terrain in video games where no steep cliffs are present and the map has gradually changing attitudes. As can be seen in the images above, this algorithm aims to programmatically generate organic-seeming terrains.

5. Baseline Implementation

The serial program will first initialize the map with user specified size, and then start the wave function collapse algorithm. After collapsing state of each individual cell, All neighbors with more than one valid states will need to be updated with the following formula:

$$lowerBound = \max(lowerBound, collapsed.value - ManhattanDistance(collapsed, this))$$

$$upperBound = \min(upperBound, collapsed.value + ManhattanDistance(collapsed, this))$$

Since there are only 10 states, only cells with a Manhattan distance of 9 or below will need to have their states updated. This propagation optimization reduces the complexity but still consumes majority

of the runtime in previous studies. A cell is randomly chosen to has its state collapsed until all cells has a definite state.

Once the execution has completed, the solution will be verified against the rule set. In addition, user can specify that the generated result to be saved to a bitmap image file so that it can be read by other software for visualization, as can be seen in figure 1 and 2.

6. Parallel Implementation

6.1 Theoretical Background

The specific set of rules implemented in this project which limits the side effects to neighboring cells. This rule set is common in game terrain generation but it is also popular because of its locality benefits. By locality, it means that non-determinate cells encapsulated by determinate cells will not be restricted further by cells outside of the walls. It also means that a map can be divided into smaller sections and allow each processes to handle their local area without depending their calculation on results from other processes. This project will try to take advantage of this task division method to spread the task to all processes in the MPI-connected system.

Moreover, the locality also limits the number of cells to update since their restrictions will be based on cells with Manhattan distance of 9 or below, which means that all cells with the newly defined cell can be updated simultaneously without depending on calculations from the previous updates. This project takes advantage of this and process all cell updates using parallel CUDA threads. In initial testings, using a single block with threads to speed up propagation can achieve some performance gains over CPU but the speedup is limited. Thus, the idea of locality used in the MPI is also adopted in the CUDA implementation to enable the benefit of scheduling CUDA warps;

6.2 Implementation Details

First, the map is divided into subsections equal to the number of MPI processes. A minimum height of 9 is set to avoid constraint conflicts. The top horizontal line of each strip are independent from the top horizontal line of other strips because the Manhattan distance is above the limit of the propagation in this rule set. Thus, the first row of each strip are independently initialized on different processes in the MPI system. Moreover, collapsing cells in a straight line can delay propagation of non-forbidden states in neighboring cells to after the entire edge is initialized.

Then, each process will propagate their top edge to the process responsible for the strip above. This allows for each process to have a localized copy of the problem with a smaller problem size, yet all of them are compatible since the connecting edges are synchronized. Each process have to propagate the constraints from both top and bottom edges before starting to fill in the remaining cells accordingly.

Each MPI processes perform calculation on CUDA-enabled GPU for parallel constraints propagation. This can be done because constraint propagation in this project is similar to convolution which is suited for SIMD architecture. Continuing the locality idea, each subsection divided by MPI can be further divided into 19 by 19 grids. Given the same local coordinates inside the block, propagation will not overlap with adjacent blocks.

Lastly, the master rank in the MPI system will collect the strips and combine them into one image which can then be verified or optionally saved to a bitmap file.

The feature to export the resultant map into a 2D image allowed visual confirmation of the correctness of the program on top of the verification implemented that only checks whether the map complies with the rules. This is an important feature not only to allow the computation result to be

saved, but also debug versions of the program where a matrix of all zero was returned which can pass the verification but is not the desired result.

7. Performance Scaling

With both MPI and CUDA independently implemented, the project can test and verify the performance scaling on various configurations. The three testable configurations are CUDA, MPI and MPI+CUDA.

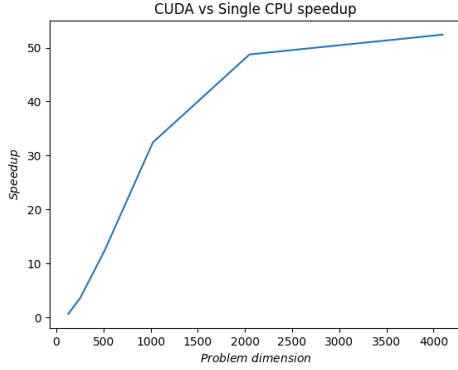


Figure 3. Speedup of using CUDA on different problem sizes.

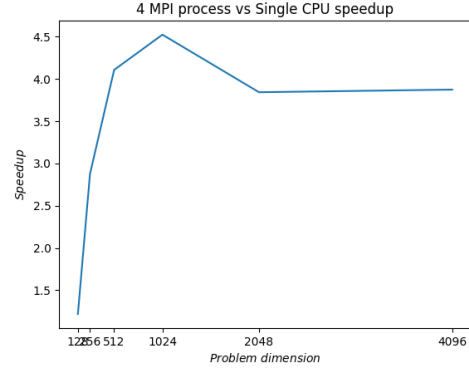


Figure 4. Speedup of using 4 MPI processes on different problem sizes.

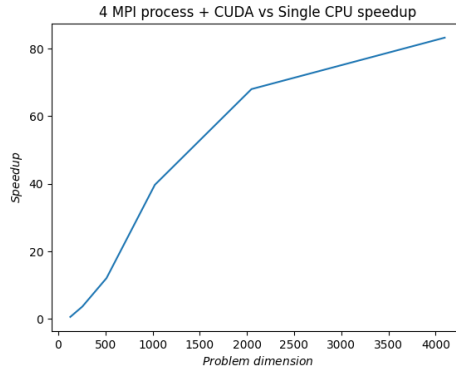


Figure 5. Speedup of using 4 MPI processes with CUDA on different problem sizes.

As can be seen from the graphs above, the speedup of CUDA has a positive relation with the problem size. This is a result of optimal warp scheduling since more blocks can be allocated to the same problem and allowing the warp scheduler to keep GPU utilization high. Calculation dependency requirements are limited to local regions and it allows a higher number of blocks to be allocated to the problem for propagation.

Early implementation of the algorithm only used one single block and threading to allow parallel propagation without loops, but the speedup was limited the speedup to around 42% but enabling more than one block showed the most prominent speedup.

On the MPI performance, it is apparent that the overhead of collecting the final map to the CPU is significant in smaller problem sizes. Curiously, at problem size of around 1024, the performance gain is greater than the number of processes allocated for the task. This could be the result of splitting the map into sections that limited divergence and increased the rate of wave function collapse which can reduce the number of calls to the random number generator.

When combined, the speedup efficiency of CUDA diminishes as the problem size per GPU decreases and communication overhead from MPI increases. However, MPI+CUDA still shows some performance improvement over single GPU implementation albeit at a lower scaling efficiency.

On performance increase as the number of processors increases, the CPU-only implementation is compared against single CPU implementation, and the MPI+CUDA implementation is compared against single GPU implementation. Both the MPI+CPU version and the MPI+CUDA version are tested with a constant problem size of 8192.

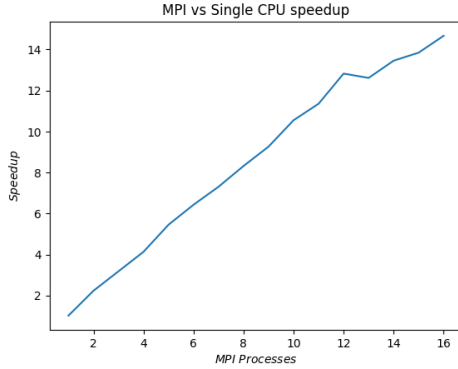


Figure 6. Speedup of MPI on different number of processes.

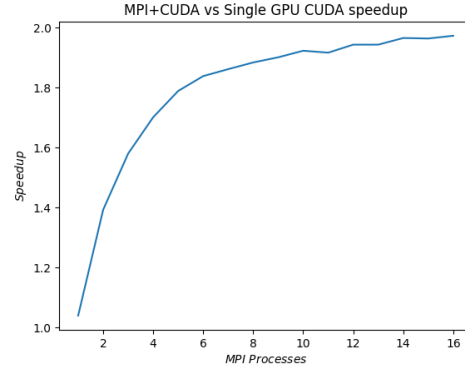


Figure 7. Speedup of MPI+CUDA on different number of processes.

As shown above, the scaling efficiency of this MPI algorithm on CPU is high with almost linear scaling when problem size is large.

The CUDA-aware version shows that the MPI implementation can lower the speedup in CUDA implementation by lowering the problem size per GPU. Although this indicates that the speedup of using MPI+CUDA is far less prominent than CUDA alone, MPI is still a powerful addition to this algorithm since CUDA performance will drop sharply if the map size exceeds the GPU memory.

In general, using a small number of CUDA-aware MPI processes are favorable in terms of scaling efficiency and reducing overhead.

8. Limitations

This implementation of the Wave Function Collapse algorithm has two main limitations. First is the restrictive rules. Since the algorithm is designed to optimize for the specific set of rules, it cannot be easily applied to speed up other rule sets. In other words, the proposed solutions are not general to all Wave Function Collapse algorithms. However, the concept of boundary setting and independent local wave function collapse can still be useful in future implementation.

The second weakness of this implementation is that the CUDA version requires that the entire map and wave information to be in GPU memory during execution. In low VRAM hardware such as those with 1GB or 2GB of GPU memory, large problem size may result in errors. In this implementation, the remedy is using the CUDA implementation in conjunction with MPI.

9. Algorithm Tuning

Initial version of the program attempted to collapse the wave of cells in sequential order, each row will be collapsed from left to right, then the next row. Although the algorithm can return a correct solution, this poses the problem of the resultant image showing diagonal patterns since the constraints of states consistently start from one direction to another.

To enable pseudo random ordering without repeatedly check on cells or keep track of all cells in a queue, modulo addition of prime number is used. As suggested on similar discussion on Stack Overflow [Lindley 2013], modulo addition of a prime number larger than dimension of the array can jump the indices without repeating or missing any in a full set of iterations.

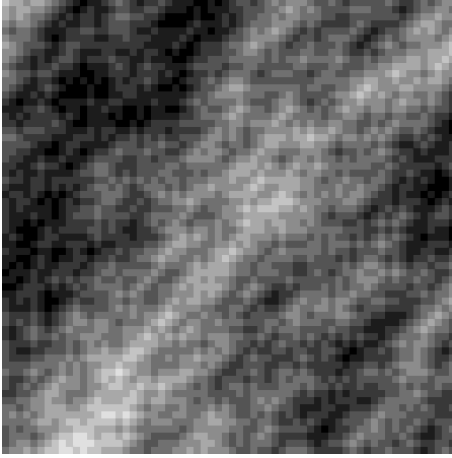


Figure 8. Collapsed in sequential order

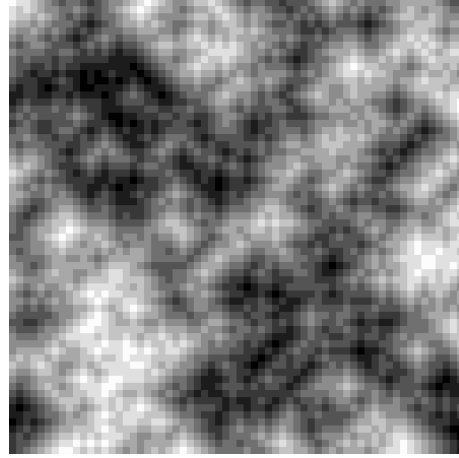


Figure 9. Collapsed in modulo addition order

My algorithm implemented a two-dimensional version of this modulo addition indexing which allows for much more organic results as can be seen in the figures above with minimal computational cost.

References

- M. Games. The fantastic combinations of john conway’s new solitaire game “life” by martin gardner. *Scientific American*, 223:120–123, 1970.
- B. Lindley. Iterating over the array in random order [duplicate]. <https://stackoverflow.com/a/18994414>, Sept 2013.
- S. Moreau. Generative design with the wave function collapse algorithm, Sep 2020. URL <https://www.bim42.com/2020/09/generative-design-with-the-wave-function-collapse-algorithm>.
- J. Orłowski and A. Lee. Parallel wave function collapse. <https://github.com/amy1h/WaveCollapseGen>, 2019.
- N. Ru. Convolutions and the game of life, Dec 2017. URL <https://nicholasrui.com/2017/12/18/convolutions-and-the-game-of-life/>.