Group Partners: Amy Lee (alee3) & Jan Orlowski (jorlowsk)

Parallel Wave Function Collapse

Team: Jan Orlowski (jorlowsk) and Amy Lee (alee3)

Website link: https://amylh.github.io/WaveCollapseGen/

Summary

We are going to parallelize a procedural image generation algorithm called Wave Function Collapse. We aim to implement the parallelization using a multi-core CPU platform and potentially using CUDA on GPUs.

Background

Wave Function Collapse is an algorithm for generating large bitmap images that are *locally similar* to a small reference bitmap image. The bitmaps are *N*x*N locally similar* if each *N*x*N* pattern of pixels occurring in the output occurs at least once in the input, possibly rotated or flipped. Below is an example of the Wave Function Collapse algorithm's output:

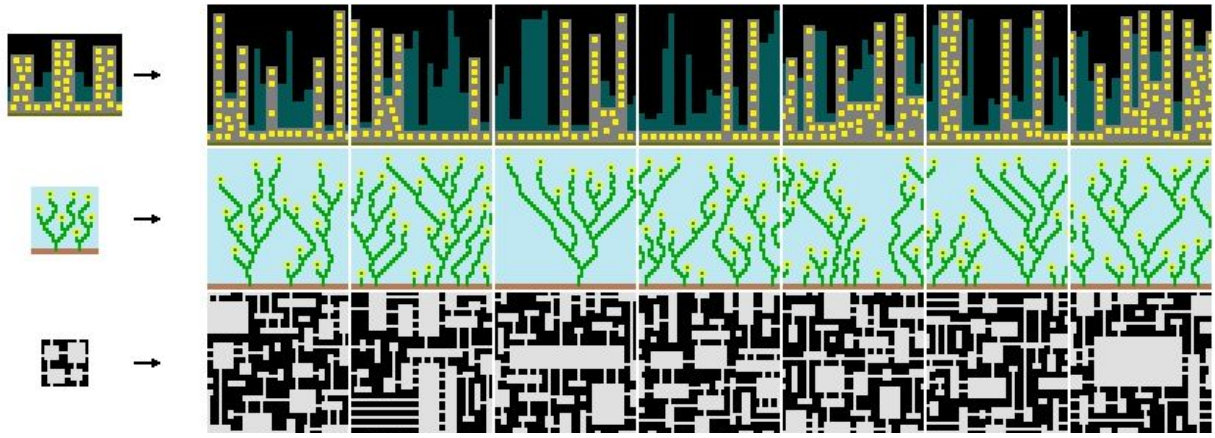
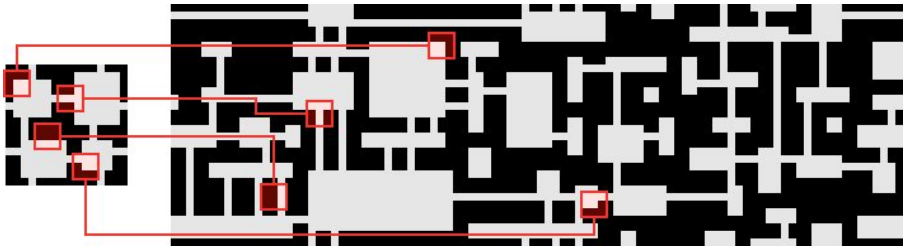


Diagram that shows mapping between local output and input patterns:



(Images taken from https://github.com/mxgmn/WaveFunctionCollapse)

The algorithm can also be extended to work in multiple dimensions and work with additional constraints.

Below is a high-level description of the (sequential) algorithm, adapted from the writeup at https://github.com/mxgmn/WaveFunctionCollapse:

1. Read the input bitmap. Identify and count NxN patterns.
    1. For simplicity, we may format the input as a discrete set of NxN pattern patches.
2. Create an array called `wave` with the dimensions of the output. Each element of `wave` represents the *state* of an NxN region, or "cell", in the output. The *state* of a cell encodes boolean coefficients that store information about which NxN patterns are forbidden (false) or not yet forbidden (true) for that cell. For each cell, we want to "collapse" the list of possible patterns until only 1 possibility is left. In the final output, the cell will be assigned that 1 pattern.
3. Initialize `wave` in the pristine state, i.e. with all the boolean coefficients being true for every cell.
4. Repeat until all cells have 1 possibility:
    1. Pick cell(s) randomly (or using a heuristic) and randomly set one of its patterns to true.
    2. Reduce the possibilities of its neighboring cell depending on their *compatibility* with its pattern. These *compatibilities* are defined in the input.
5. By now all the cell must be either in a completely observed state (all the coefficients except one being zero) or in the contradictory state (all the coefficients being zero). In the first case return the output. In the second case we exit without returning anything.

We believe step 4.1 may be parallelized, i.e. we choose multiple starting cells to propagate updates in parallel. And we believe step 4.2 may be parallelized, i.e. we update neighbors in parallel.

Challenges

For large images (and even more so for higher dimensional grids), it can take a long time to generate the output due to the large amount of cells in the image. But there are several challenges in parallelizing the algorithm to make it faster:

- **Cell Neighbor Dependencies**: Whenever a cell is collapsed, all of its neighbors must also be updated. This may lead to contention if two neighbors are both collapsed in parallel.
- **Square-in-Square Access:** When we access a cell, we also access all the pixels in it. We have to be careful to arrange the pixels in memory in a way that does not lead to false sharing.

- **Non-uniform Access Patterns**: Since starting cells and patterns are chosen randomly, and the propagation of updates may occur in random directions, the array accesses will also be random and could lead to bad caching behavior. However, we believe there may be some degree of locality since updates are propagated through neighbors.
- **Contradictions:** In the event that we run into a contradiction while generating an image (i.e. a cell cannot be collapsed in any way without violating a constraint/local similarity), we need a way to stop execution of all cores and backtrack/reset the generation process fast.

Resources

We will use the write up by Maxim Gumin at https://github.com/mxgmn/WaveFunctionCollapse to understand the algorithm on a theoretical level. We have also found multiple sequential C++ implementations of the Wave Function Collapse algorithm by:

- Emil Ernerfeld – https://github.com/emilk/wfc
- Mathieu Fehr and Nathanaël Courant – https://github.com/math-fehr/fast-wfc

We will use these sequential implementations as a starting point for our parallel code and as benchmarks for measuring our parallel speedup.

We may also reference a 2009 paper by Paul C. Merrell, which was the basis for Gumin's algorithm, at http://graphics.stanford.edu/~pmerrell/thesis.pdf.

Goals & Deliverables

Plan to Achieve

- Parallel WFC (multi-core CPU) for 2D bitmap images

Hope to Achieve

- Parallel WFC (CUDA on GPU) for 2D bitmap images
- Parallel WFC (multi-core CPU) for 3D grids
- Parallel WFC (CUDA on GPU) for 3D grids

Planned Demo

For the poster session demo, we hope to show:

- Side-by-side comparisons of the sequential WFC algorithm and our parallel WFC implementations, working on images of various patterns and sizes in real time.

- Graphs displaying the relative speedup of our implementations for different numbers of cores, input sizes, and output sizes.

Platform Choice

First, we plan to parallelize the WFC algorithm in C++ using multi-core CPU code. We are using C++ because much of the starter code we plan to use is written in C++, and the language is relatively fast and familiar to us. We also believe using multi-core CPU is suitable because active, non-adjacent cells can perform work independently on separate cores.

We also hope to parallelize the WFC with CUDA on a GPU. We believe this may be effective in improving locality and caching behavior for adjacent cells that are performing work in parallel, since CUDA has features that take advantage of blocking parallel threads.

Schedule

Date	Goal	Description
Nov 2	Research + Literature Review	Get familiar with the algorithm and the sequential implementations described above.
Nov 8	(2D) Tackle Non-uniform Access Patterns	Improve locality / caching for cell / neighbor accesses, using multi-core CPU
Nov 14	(2D) Tackle Cell Neighbor Dependencies	Handle contention between dependent cells, using multi-core CPU
Nov 18	Checkpoint	Formal progress report
Nov 24	(2D) Tackle Non-uniform Access Patterns	Improve locality / caching for cell / neighbor accesses, using CUDA
Nov 30	(2D) Tackle Cell Neighbor Dependencies	Handle contention between dependent cells, using CUDA
Dec 9	Final Report	Clean up code, finalize tuning parameters, prepare poster
Dec 10	Presentation	Present!