

CSC-213: PROF. CURTSINGER

FINAL PROJECT REPORT

ANNA BLINDERMAN, DAVID KRAEMER, ZACHARY SEGALL

Project Overview (<400 words)

- introduce Game of Life
- describe GUI
- describe GPU stencil update
- describe listener CPU threads
- describe evaluation strategies
- summarize evaluation results

We implemented a variation of Conway's Game of Life – a cellular automaton simulation created by the mathematician John Conway in order to explore the role of various types of optimization within it. Scientific computing and large-scale simulations are incredibly useful across a variety of fields. However, many of these programs are prohibitively slow. Therefore the algorithms and design of these programs is crucial.

[TRANSITION] Life is simply a series of iterations of a rendering of a grid of cells, all of which are either dead or alive. Each cell's state is determined by the states of its direct neighbors. Life provides an interesting basic model for population growth and is an ideal demonstration of complex systems arising from simple rules.

[friend can you please write the paragraph on the ui don't you read through libraries in your spare time or something]

For each iteration, we use Conway's algorithm to determine the state (dead or alive) of every cell in our grid. Because this algorithm takes as input a cell along with the states of each of its eight immediate neighbors, this algorithm naturally lends itself to an "embarrassingly parallel" stencil computation on the GPU.

We also have two threads running on the CPU – one for recording and acting on user input from the mouse; the other from the keyboard.

To evaluate our system, we first varied the number of threads per block in our stencil function calls. Then, we implemented a feature that allowed us to find regions with no live cells in order to skip their calls to the update functions.

[LOL SUMMARY OF RESULTS WHAT HAPPENED?!?!?] [friend]

Design and Implementation (~2 pages) (Zachary)

- summary of overall implementation structure
- Component 1: GUI
 - responsibilities/how fits into entire structure
 - rationale for choice
 - data structures/algorithms/libraries/other details
- Component 2: mouse/keyboard input
 - responsibilities/how fits into entire structure
 - rationale for choice
 - data structures/algorithms/libraries/other details
- Component 3: GPU stencil update
 - responsibilities/how fits into entire structure
 - rationale for choice
 - data structures/algorithms/libraries/other details
- Butler Lampson's Hints for Computer System Design (integrate?)
 - don't reinvent the wheel (+ how it did/didn't help)
 - be prepared to throw an entire thing out (+ how it did/didn't help)
- figures if appropriate

The rules to Conway's Game of Life are simple. There exists a grid of cells, where every cell is either dead or alive. At each iteration of the simulation, Conway's algorithm is applied to every cell in order to determine its next state. The algorithm is defined as follows:

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

[CITE WIKIPEDIA, ALSO INSERT PICTURE] With these baseline rules, we implement a system where the user was able to click cells to switch their state, pause and run the simulation automatically, or step through various iterations. We also have features that go beyond the basic simulation, including reading boards from files, adding gliders to the board, and having cells change color as the age. The exact specifications for adjustable features are as follows:

- GUI (game board) display size
- cell size (thus adjusting the number of cells)
- delay between iterations while running simulation
- colors, including linear interpolation between start and end values
- can randomize a board
- can load board from file

During the simulation, the user also has the ability to:

- left-click (or drag) to bring dead cells to life
- right-click (or drag) to kill live cells
- press **Ctrl-Q** to quit the simulation (close the GUI)
- press **Ctrl-C** to clear the board (set all cells to dead)
- press **Ctrl-P** to pause the simulation (set the global **running** variable to false)
- press **Ctrl-Space** to advance the iteration one step (run the update function just once)
- press **Ctrl-G** to add a "glider" shape to the board [DAVIS HOW U DO DIS]

Given that the update algorithm is itself simple to implement, most of our difficulties were centered around coordinating between the GUI display, user input, and the update algorithm.

We start development with the GUI, which is responsible for displaying the board and updates with reasonable response time. We use the bitmap class from the Galaxies lab for the GUI because it offers high resolution for the display and easily adjustable parameters. Reusing the bitmap also follows Lampson's principle of "reuse good ideas" - we already have a class built for exactly what we want to do, so there's no need to reinvent the wheel. The

GUI itself provided no difficulties with the implementation.

The Game of Life takes place on a grid, We intentionally separate the representation of the takes place on a grid, so we need the GUI Regarding the more general mechanics of our project, we will use an `SDL_surface` to create a board as in the Galaxy lab. The cells on the board will be initially populated from either a random starting configuration, one of some set of preset starting configurations, or by the user clicking squares on the board. We will have a struct to represent the state of the board and will use a bitmap for the visualization of the board. Because our internal representation will be separate from our visualization, we will need to translate from the internal representation (the struct) to the visualization (the bitmap) at each interval of time. Further, this separation will allow us to more easily manipulate and update the representation and then simply update the visualization afterwards using Conway's original algorithm.

To handle user input, we intended to use a scheduler (as in the Worm lab) in order to split up listening for input and updating across processes. We quickly ran into problems with this approach, as the listeners would sometimes fail to register input if they weren't being executed by the schedule at the exact moment the user clicked or pressed a button. Thus, we realized we needed complete concurrency on the CPU and switched to an implementation with threads: one for mouse input, one for keyboard input, and the main one to run updates. At this point, we applied Lampson's lesson of "Throw one away" and gave up on our scheduler based approach.

We used SDL to register input... [DAVIS PLEASE TALK ABOUT SDL CAUSE LOL WHAT IS THAT] [AND ALSO BITMAP THX UR THE BEST]

In order for the threads to properly take in and act on user input, we have a struct which contains the information necessary for the various actions in the program to be taken – the location of the cell from which the user input was recorded, the state of the mouse (clicked or unclicked), and the `SDL_event` which indicates [IS THE THE KEY DAVIS PLS HELP HELP PLS]. The mouse thread is then responsible for executing the first two of the above bullet points when appropriate; the keyboard thread is responsible for the other five. The main thread calls our update function. In any case, this struct of arguments is passed to the functions so that they may act appropriately.

For the simulation itself to run, we must update all cells in our board according to Conway's original algorithm. As described above, this algorithm naturally lends itself to an "embarrassingly parallel" implementation. The update function progresses in two steps: first, we used a stencil pattern since a cell's next state depends on the states of its eight immediate neighbors, then use a map pattern to process whether or not a cell survives. We actually have two grids in the program: one that stores counts each cell's neighbors and one that stores the actual age of each cell. The steps are parallelizable because the state of a cell depends only on the states of the cells in the past iteration (i.e. not on any of the states of the other updated cells). Between each call of the update function, we copy the grid between the GPU and the CPU in order to account or user input.

Evaluation

- describe experimental setup
 - hardware (???)
 - software (libraries) (friend?)
 - ~~data-gathering methods (Anna)~~
- ~~explain what trying to measure (speedup) (Anna)~~
 - ~~vary block size~~
 - ~~implement thing that only updates regions with living things?~~
 - ~~implement serial version?~~
- include at least one graph with at least 8 data points [friend]
- discuss interpretation of results [friend]

[HEY DAVIS CAN YOU

In addition to ensuring that our implementation properly reflected Conway’s rules, we verify that the concurrent components of our system function properly. For our listener threads on the CPU, we notice from running many tests that the program’s response to user input is both immediate and correct.

However, the speed of our program is our main focus in our evaluations. As a benchmark, we implement a serial version of each of our GPU kernel functions involved in the advancement of our simulation from one iteration to the next. In order to test the efficiency of our parallelized program against that of the serial version, we create a file and use `time_ms` to measure the running time of our update function and write that information to the file.

For our serial implementation, we loop over every cell in our grid to count the number of its neighbors that are alive. Then we apply Conway’s algorithm to decide what its state will be at the next iteration. Finally, we loop over each pixel in our bitmap in order to correctly render these updates to the screen. Our parallel implementation is similar, although we count neighbors and update our cell grid using blocks of threads.

We ran 1000 iterations of our serially-implemented simulation and collected the running time of each of the 1000 updates in a file. We did five similar runs for our parallel implementation, testing threads per block values of 16, 32, 64, 128, and 256.

While running our simulations, we noticed a possible way to speed up our code further: because there are oftentimes large regions of the board in which there are no live cells, we can know that there is no need to check for updates in such regions. Thus, running the kernel function on those cells is computationally wasteful. In order to reduce the effects of this inefficiency, we overlay yet another grid onto our board. This “regions” grid, which is of lower dimension than our grid of cells, holds the a count of the number of live cells within its boundaries. We then modify our GPU kernels so that they only run the main body on

cells that are in regions with at least one live cell. We again run 1000 iterations of this program, testing all ten combinations of the five threads per block values from above and a “region” dimension of either 5 or 50. We note that we ignore edge cases (within a region, we must actually check the entirety of the region as well as another layer of cells surrounding it in order to ensure no cell within the region will be updated), so these tests report a lower runtime for our optimized runs than they would if our optimization were implemented perfectly (properly? but like lol we wouldn’t wanna sound improper now would we)