Adilet Segizekov and Eli Cohen

15-418 Parallel Computing

Parallel Hexagonal Cellular Automata Final Writeup

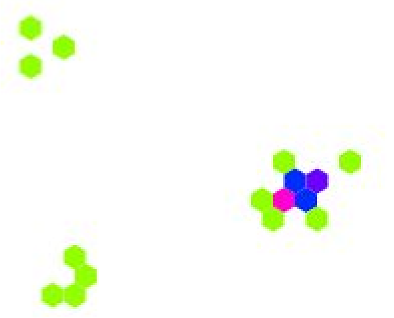


Image courtesy of: https://en.wikipedia.org/wiki/File:Oscillator.gif

Summary:

Cellular automata are discrete models used to simulate various processes often inspired by nature. It usually consists of a grid of cells where each cell is in one of a finite number of states. For every time step the state of each cell is updated based on a predefined rule with respect to its neighboring cells. For example in the famous Game of Life each cell in the “live” state becomes “dead” if there are fewer than 2 live neighboring cells or more than 3 live cells. Whereas any dead cell can become live again if it is surrounded by exactly 3 live cells.

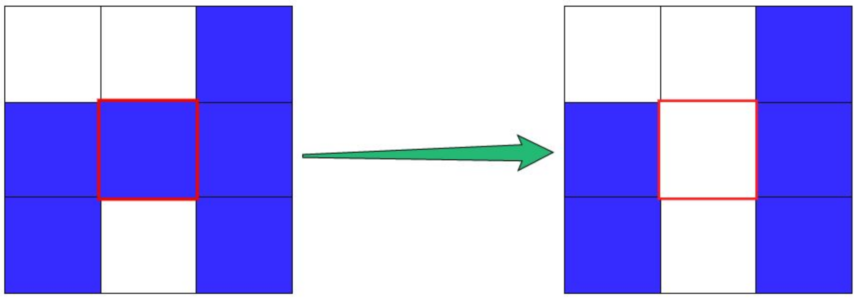


Fig 1. An alive cell becomes dead in the next iteration

These simple rules can create complex simulations, which can be used for a number of applications. Examples include fluid simulators, traffic simulators, and random number generators.

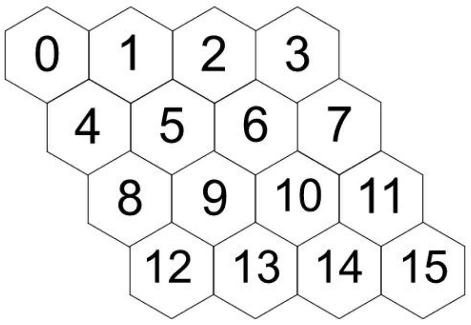
Our project idea was to create a cellular automaton for a hexagonal grid rather than the traditional square one. In addition to this, we decided to let the rules for the automata be customizable. Due to their specializations for working with large arrays, we decided that it would be best to create our program for GPUs, and code it using the CUDA framework, which we already had experience with.

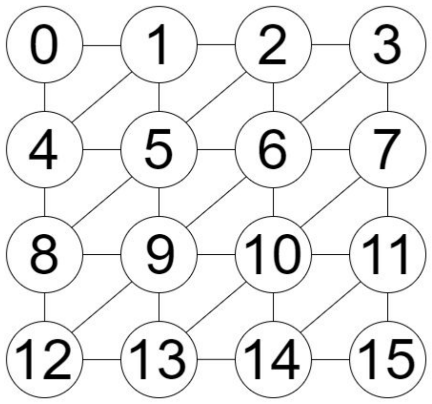
We an easy time at the beginning of our project creating both a serial version of the automata with no parallelism, and a simple CUDA version without any non-trivial optimizations. However we had difficult time improving the speed of our program beyond the naive cuda version, despite using several different strategies. One of our most involved attempts involved maintaining a list of all the cells which could possibly change state in a given cycle, but despite all the time we invested in this approach our final version of it was still slower than the naive implementation.

Eventually we created a method which sped up our program significantly. This method involved creating a lookup table to compute the next state of a group of 8 cells, whose next states would all be computed together by the same CUDA thread. This algorithm gave us an execution time of up to 80 times that of the original serial implementation.

Design process:

The first challenge we came across was figuring out how to represent a hexagonal grid on a square array. This ended up having a fairly simple solution where we took the grid of connections for a traditional square automation but removed the upper-right and lower-left connections, as shown in the following graphs:





While every cell is arranged in rectangular grid, the fact that each cell only has connections to 6 instead 8 other ones means that the cells are computed on as if they were arranged hexagonally.

Writing the naive parallel algorithm was pretty straightforward. We assign one cell in the grid to each cuda thread, which computes the live neighbors around the cell and updates the cell correspondingly.

```
__global__ void naive_kernel_single_iteration(grid_elem* curr_grid, grid_elem* next_grid) {

  // cells at border are not modified
  int image_x = blockIdx.x * blockDim.x + threadIdx.x + 1;
  int image_y = blockIdx.y * blockDim.y + threadIdx.y + 1;

  int width = constant_parameters.grid_width;
  int height = constant_parameters.grid_height;
  // index in the grid of this thread
  int grid_index = image_y*width + image_x;

  // cells at border are not modified
  if (image_x < width - 1 && image_y < height - 1) {

    uint8_t live_neighbors = 0;

    // compute the number of live_neighbors
    int neighbors[] = {grid_index - width, grid_index - width + 1, grid_index + 1,
                        grid_index + width, grid_index + width - 1, grid_index - 1};

    for (int i = 0; i < 6; i++) {
      live_neighbors += curr_grid[neighbors[i]];
    }

    //grid_elem curr_value = constant_parameters.curr_grid[grid_index];
    grid_elem curr_value = curr_grid[grid_index];
    // value for the next iteration from supplied rule
    grid_elem next_value = const_parameters.rule->next_state[live_neighbors + curr_value*7];

    //constant_parameters.next_grid[grid_index] = next_value;
    next_grid[grid_index] = next_value;
  }
}
```

While the naive implementation didn’t take a significant amount of effort to create, improving on it was a long process of trial and error. We spent time working on a number of approaches to improve the algorithm, and for a long while we had no success, with none of our approaches having a speedup greater than that of the naive implementation. Early on we realized that this would be a difficult task due to the simple nature of the automaton algorithm, where the majority of time is spent being held up by memory reads rather than actually doing computations. Because of this we mainly tried to focus on decreasing memory accesses with our attempts. Some of these unsuccessful attempts include:

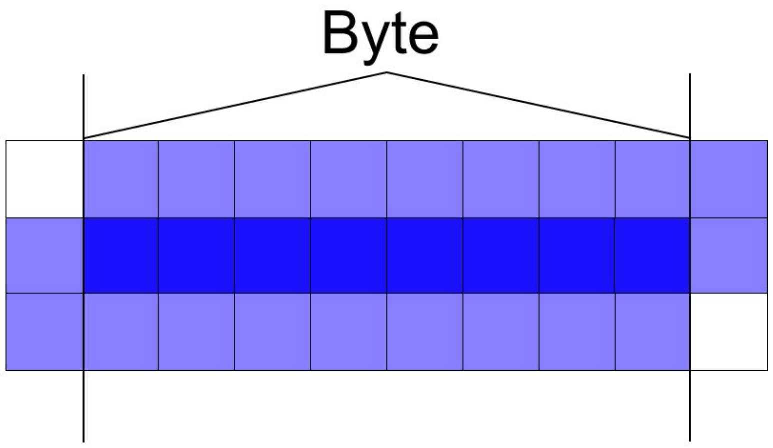
- **Shared Memory:** We had each block create an array in its shared memory, which would be filled in with the current values in the region in the grid that its threads would modify. Instead of accessing the global grid for reads, threads would access this grid. We believe that the failure for this approach to give us a speedup is due to the cells not being accessed enough (only a maximum of 7 reads - and no writes - per cell) for shared memory to be more efficient than the GPU's internal tricks. In addition, to get information from the border of cells surrounding the block's area, which is needed in the computation, dummy threads whose only purpose was to load the values of the border cells into shared memory had to be created, increasing the method's overhead.
- **Multiple threads per cell:** We simply had single threads compute the next state of 2 threads instead of one. While this was unsuccessful at first, we ended up revisiting the idea in a later approach.
- **Skipping dead blocks:** We created a new array in memory with an index for every block of threads in the kernel. The purpose of this array was to record whether a block had or was adjacent to any live cells. Every thread of a block

checked this value at the beginning of the kernel, and if they didn’t have any live cells they could simply stop computation there, as a block with all dead cells would have no way of gaining live cells unless there were live cells on its perimeter. Additional work had to be done to update the new grid every cycle. The problem with this approach was that even when a block of threads all stopped their computation early, they still contributed to a significant amount to the total runtime due to the time cost of actually starting the threads up in the GPU. In addition to this, accesing the value in the new grid was very costly, especially since hundreds of threads would be trying to access the same value at once, leading to contention.

- **List of active cells:** The approach we spent the most time working on took the previous approaches’ same basic concept of skipping the computation of the next state of dead cells. As the biggest problem with that approach was the still significant amount of time that threads for cells contributed, we thought that a good way to improve performance would be for there to not even be threads for dead cells in the first place. To do this, each cycle we created an array of all the cells that are alive or have alive neighbors, and only create threads for cells on this list. This dramatically reduced the number of threads being created per cycle, therefore significantly reducing the time spent in the kernel. The catch to this was the fact that the time required to create the list every cycle was extremely large, initially increasing the runtime by over 100 times. We were eventually able to make the algorithm for creating the list more efficient, reducing most of its work to a single exclusive scan. However, this scan still took up over 95% of the execution time.

Final Implementation:

For our final implementation we decided to use a lookup table to store precomputed cells based on certain cell configurations. In this approach we first store the state of each cell as a single bit and we assign our grid element size to be a single byte, thus each grid element contains 8 cells.



The diagram above shows 1 grid element (8 cells) that is to be computed, and the purple cells are the neighboring cells required for performing the next state computation. Each cuda thread receives one grid element that needs to be updated.
The thread then computes the index for the left half of the grid element using bitwise operations which are very fast. This is repeated for the right half of the grid element.

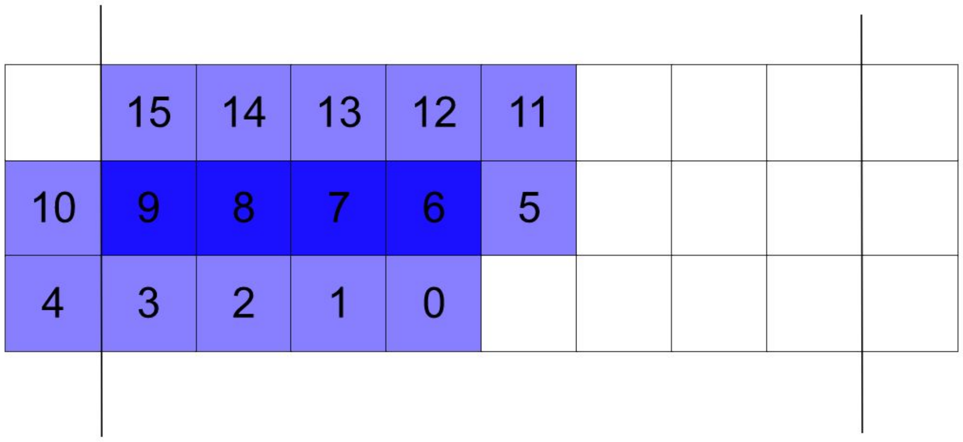


Fig 4. Left half index calculation

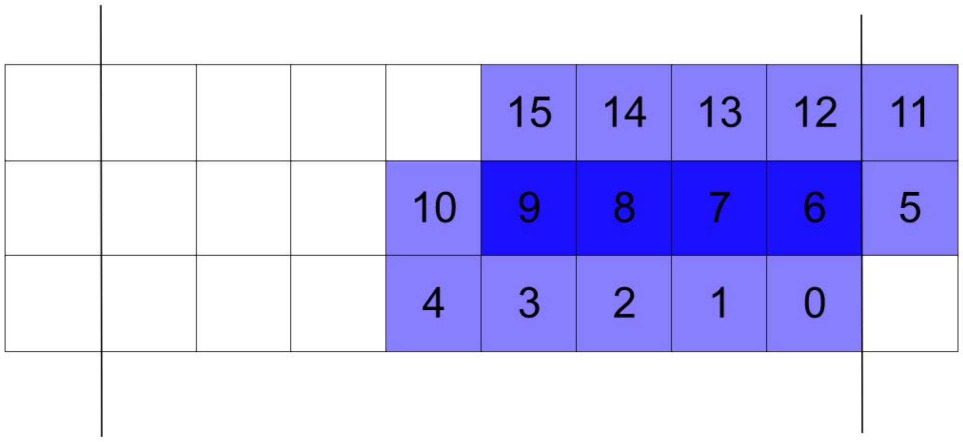


Fig 5. Right half index calculation

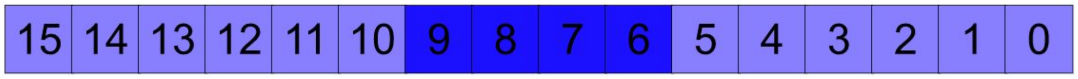


Fig 6. Cell configuration as a 16 bit index number

The resulting 16 bit number is used as an index into the lookup table which returns a 4 bit number, representing the next states of the 4 cells. Since a 16 bit number is used as an index, the lookup table contains 2^{16} elements.

```
// kernel_single_iteration (CUDA device code)
//
// compute a single iteration on the grid, putting the results in next_grid
__global__ void kernel_single_iteration(grid_elem* curr_grid, grid_elem* next_grid) {
  // cells at border are not modified
  int x = blockIdx.x * blockDim.x + threadIdx.x + 1;
  int y = blockIdx.y * blockDim.y + threadIdx.y + 1;

  int width = const_params.grid_width;
  int height = const_params.grid_height;
  int cols = const_params.num_cols;
  // index in the grid of this thread
  int grid_index = y*cols + x;

  // cells at border are not modified
  if (x >= cols - 1 || y >= height - 1)
      return;
  int left_col  = x - 1;
  int right_col = x + 1;
  int y_above   = y - 1 ;
  int y_below   = y + 1;

  //grab cells from the left column
  int buffer_top = 0;
  int buffer_mid = curr_grid[left_col + cols*y] << 16;
  int buffer_bot= curr_grid[left_col + cols*y_below] << 16;

  //grab cells from the middle column
  buffer_top  = curr_grid[x + cols*y_above] << 8;
  buffer_mid |= curr_grid[x + cols*y] << 8;
  buffer_bot |= curr_grid[x + cols*y_below] << 8;

  //grab cells from the right column
  buffer_top |= curr_grid[right_col + cols*y_above];
  buffer_mid |= curr_grid[right_col + cols*y];

  int left_half_index = (buffer_top & 0xf800) | ((buffer_mid & 0x1f800) >> 6) | ((buffer_bot
& 0x1f000) >> 12);
  int right_half_index = ((buffer_top & 0xf80) << 4) | ((buffer_mid) & 0x1f80) >> 2 |
((buffer_bot & 0x1f00) >> 8);

  //left val and right val each contain 4 cells/bits
  grid_elem new_val = 0;
  grid_elem left_val = const_params.lookup_table[left_half_index];
  grid_elem right_val = const_params.lookup_table[right_half_index];
  new_val = (left_val << 4) | right_val;
  next_grid[grid_index] = new_val;
}
```

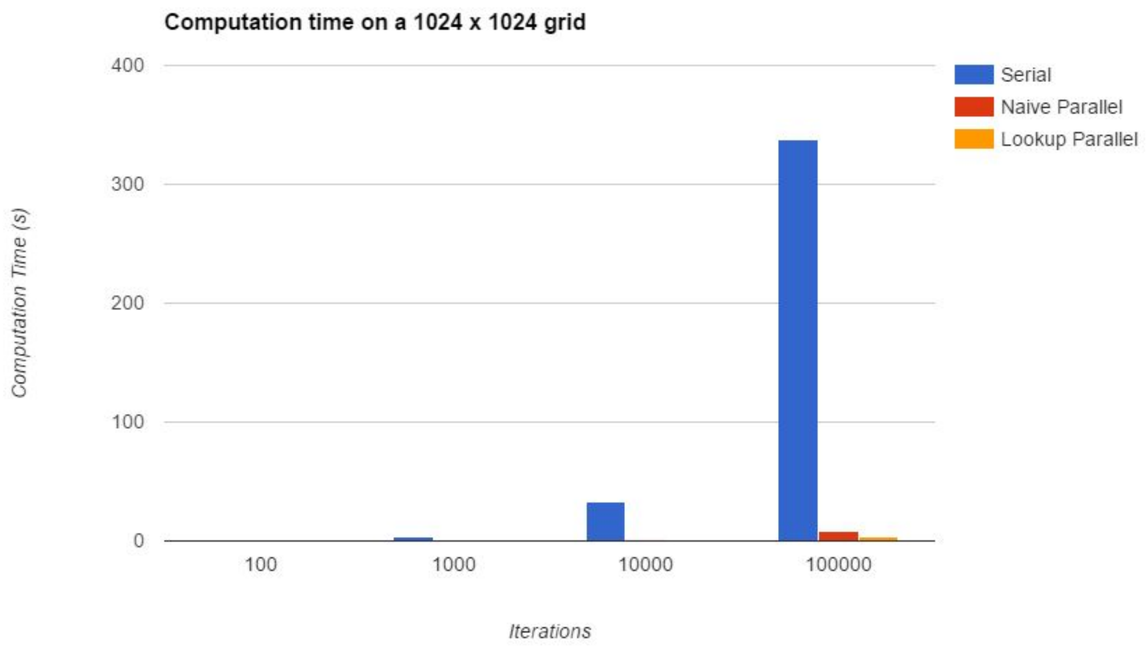
The lookup table is constructed in much the same way except in reverse. It iterates through all numbers 0 to 2^{16} - 1, treats the 16 bit number as a cell configuration and computes the next states using bitwise operations.

```
//computing lookup table for a given index
//@param next_state: array representation of automata rule
grid_elem* create_lookup_table(grid_elem* next_state) {
  grid_elem* table = (grid_elem*) malloc(sizeof(grid_elem) * (1<<16));
  int max = 1<<16;
  for (int num = 0; num < max; num++) {
    int i = num;
    grid_elem res = 0;
    for (int j = 0; j < 4; j++) {
        int center = ((i >> 6) & 1);
        int live_neighbors  = (i & 1) + ((i >> 1) & 1) + ((i >> 5) & 1) + ((i >> 7) & 1) +
((i >> 11) & 1) + ((i >> 12) & 1);
        int alive = next_state[live_neighbors + center*7]; //alive or not based on rule
        res |= alive << j;
        i >>= 1;
    }
    table[num] = res;
  }
  return table;
}
```

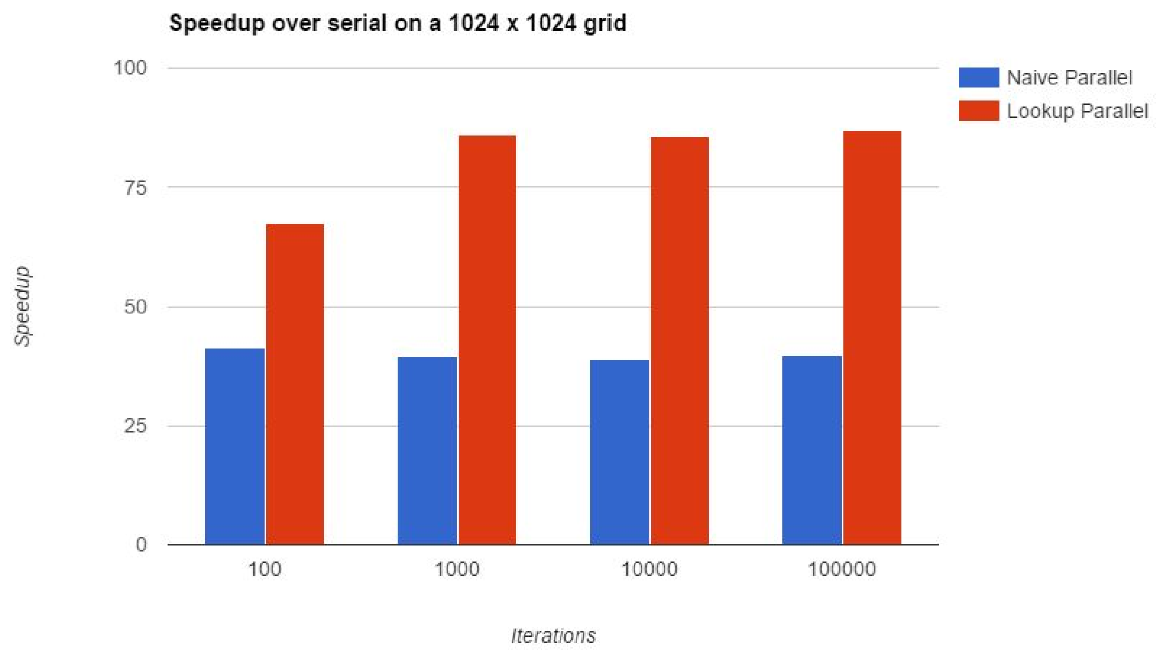
We have also tried to have the lookup table store the next state of 8 cells at once, thus each thread would only require 1 memory access instead of 2 as in our implementation. However this approach turned out to be worse since the number of elements now becomes 2^{28}. This exponentially larger lookup table decreases the number of cache hits and thus each array access is more likely to be slower.

Performance:

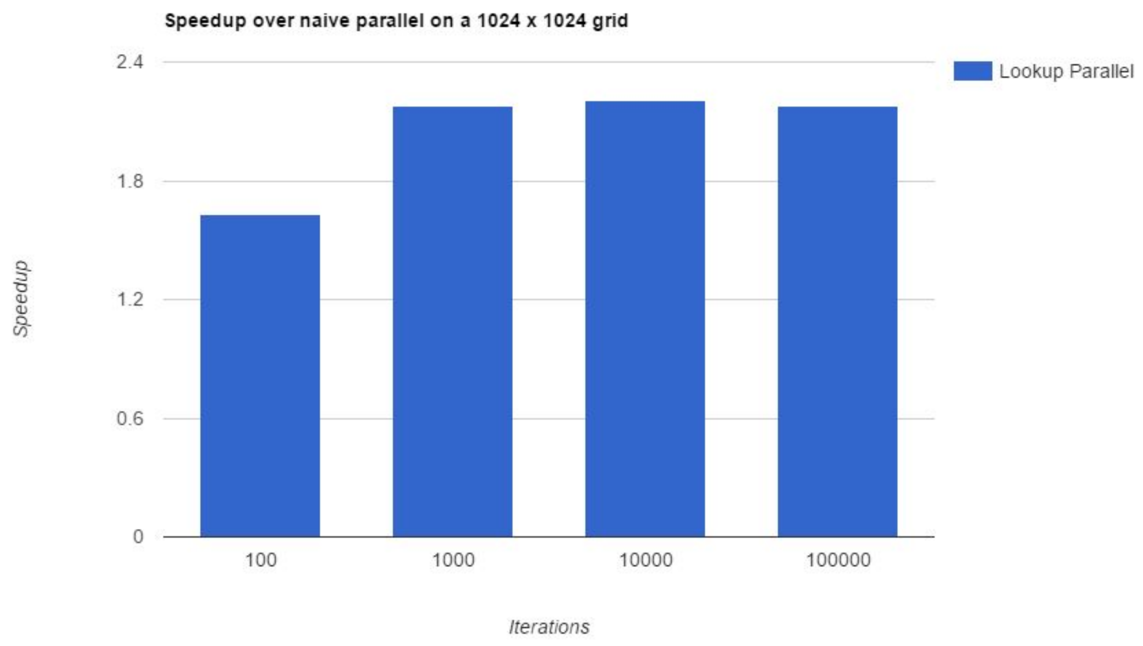
All performance results were tested on the latedays machines using the Nvidia K40 GPU. The tests used a 34/2 rule where an alive cell stays alive if it has 3 or 4 live neighbors and a dead cell becomes alive if there are exactly 2 live neighbors.



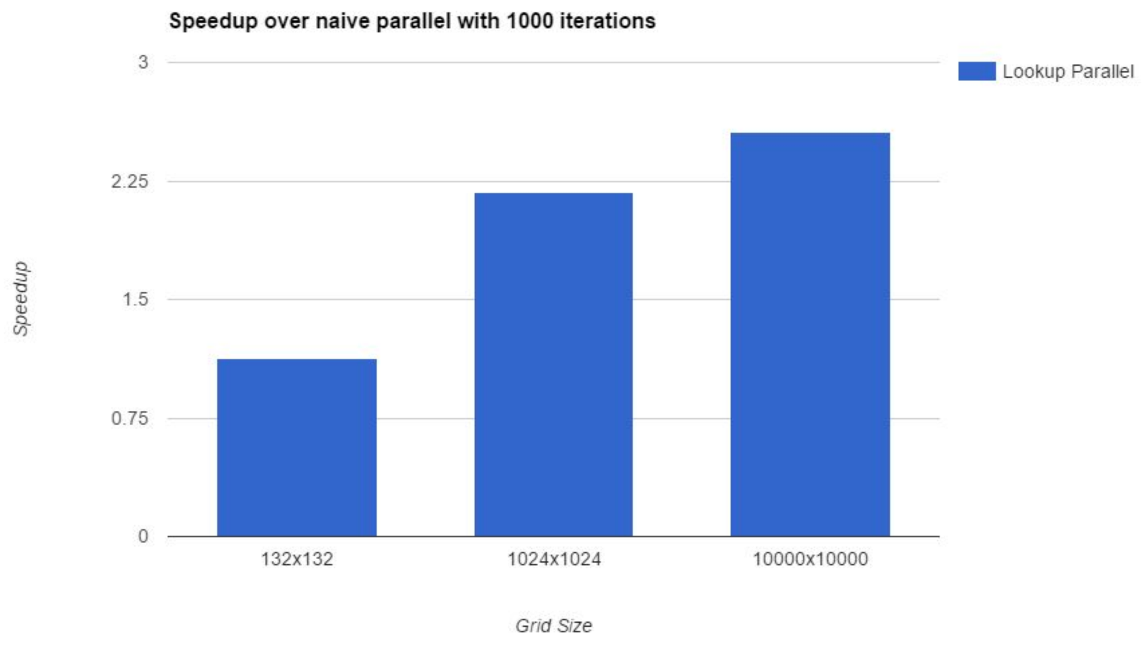
The above graph illustrates how both parallel implementations run much faster than the serial implementation for a large number of iterations.



The speedup graph relative to the serial implementation is quite interesting. The naive parallel approach provides a constant speedup of 40x regardless of the number of iterations, whereas the lookup parallel approach speedup jumps from 67x to 85x at 1000 iterations and then stays constant. We hypothesize this is because by this point most of the grid is homogenous, and thus reads to the the lookup table hit the cache more often, giving significant speedups.



This graph illustrates the previous point, except now it displays the relative speedup of the lookup parallel approach to the naive parallel approach. For a low number of iterations the lookup achieves a 1.6x speedup, which increases to 2.2x for 1000 iterations and greater.



Another interesting result can be seen from the graph above: as the grid size increases exponentially, the lookup parallel speedup increases linearly. We theorize that this is mainly due to the lookup table being more memory efficient. As the number of cells becomes greater than the number of cuda threads, each thread in the naive version is tasked with more grid elements to compute. Whereas in our final implementation each thread updates 8 cells at a time,

Conclusion:

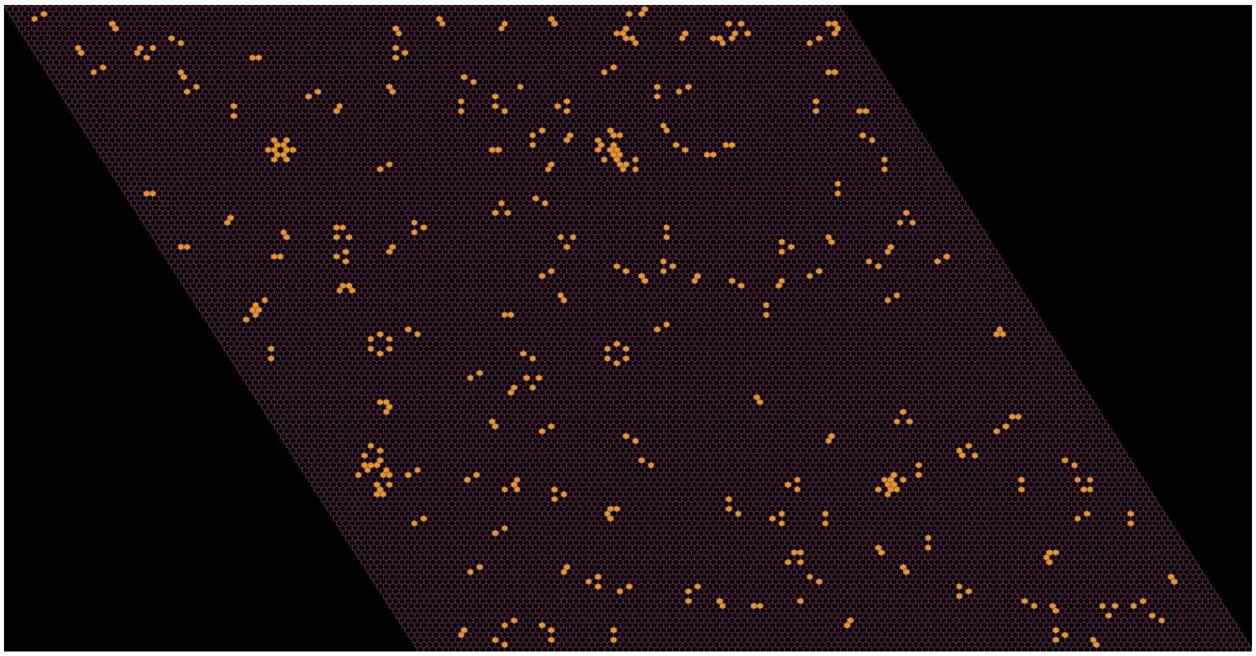


Fig 7. 132 x 132 grid after 100 iterations of rule 34/2

We originally planned to create a framework for hexagonal automata, that could have any number of states and was not limited to the 6 surrounding neighbors for updates. However we had trouble parallelizing such a general framework with lots of variable parameters and thus settled for a simpler binary cellular automata with simple adjustable rules.

In the end we managed to achieve an 85x speedup using our lookup table implementation in CUDA, which we are happy with. However we still feel that this could still be improved upon by perhaps optimizing the number of neighbor memory accesses. We believe that CUDA was the right platform of choice for this project due to the high amount of data level parallelism with a large array.

References:

Game of Life Lookup Table
Assignment 2 as Starter Code