Parallel Cellular Automata Cloud Simulation on the GPU

Daniel Hua (dhua)

1 Summary

I implemented a parallel cloud simulation based on cellular automata and improved its performance up to 10 times its original performance on a NVIDIA GTX 960M GPU.

2 Background Info

The cloud simulation is based on cellular automata in a 3D voxel grid. Each cell in the voxel grid represents a location in space and contains 3 bits of information:

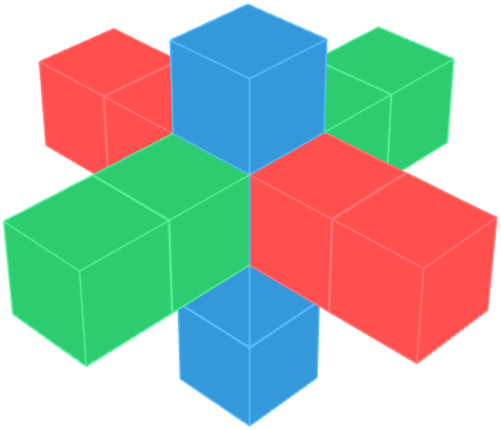
1. HUM (H): The humidity level of the cell
2. ACT (A): The “activation factor” of the cell
3. CLD (C): Does this cell have a cloud or not?

The goal of the simulation is to update the voxel grid so that its information can be passed to a renderer to render the clouds.

The function used to update each cell is as follows:

$$
\begin{aligned}
&hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i) \wedge \neg act(i,j,k,t_i)\\
&cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i) \vee act(i,j,k,t_i)\\
&act(i,j,k,t_{i+1}) = \neg act(i,j,k,t_i) \wedge hum(i,j,k,t_i) \wedge f_{act}(i,j,k)\\
&f_{act}(i,j,k) = act(i+1,j,k,t_i) \vee act(i,j+1,k,t_i)\\
&\quad \vee act(i,j,k+1,t_i) \vee act(i-1,j,k,t_i) \vee act(i,j-1,k,t_i)\\
&\quad \vee act(i,j,k-1,t_i) \vee act(i-2,j,k,t_i) \vee act(i+2,j,k,t_i)\\
&\quad \vee act(i,j-2,k,t_i) \vee act(i,j+2,k,t_i) \vee act(i,j,k-2,t_i)
\end{aligned}
$$

Here is a 3D visualization of the neighboring cells that each cell needs to access in the formula above:



In the above image, each “block” represents a cell in the voxel grid that needs to be accessed to update the cell in the middle (occluded). The red cells are in the x direction, the green cells are in the y direction, and the blue cells are in the z direction. Note that including the occluded block in the middle, a total of 12 memory accesses are required to update the simulation for each cell.

A more detailed description of the algorithm can be found in the original paper, here.

2.1 Potential Areas for Optimization

One of the optimizations that can be done depends on the fact that each cell only requires 3 bits to store, and the update function involves only simple bitwise operations. We can take advantage of this and use one bitwise operation to update multiple cells.

Another fact that I originally thought I could take advantage of is that cellular automata problems generally tend to be bandwidth bound - meaning that I can improve performance by reducing the number of memory operations required by the algorithm. However, it turns out that for this application, this is not the case.

3 Approaches and Results

The first decision I made is whether to perform the simulation on a Intel i7-4720HQ processor or a NVIDIA GTX 960M graphics card. I ended up going with the GPU because of the following reasoning: since the entire purpose of performing this simulation is to render the clouds, it would make more sense to perform the simulation on the GPU. That way, we can avoid having to send a 3D buffer from the CPU to the GPU for rendering, which is a common bottleneck. This is especially true for this application because 3D buffers tend to grow very large very quickly.

Since I did not have access to any starter code or reference implementation of the algorithm, I had to implement a naive version first, so that I can perform comparisons. I started out using one of the CUDA samples, and modifying it to perform the cloud simulation.

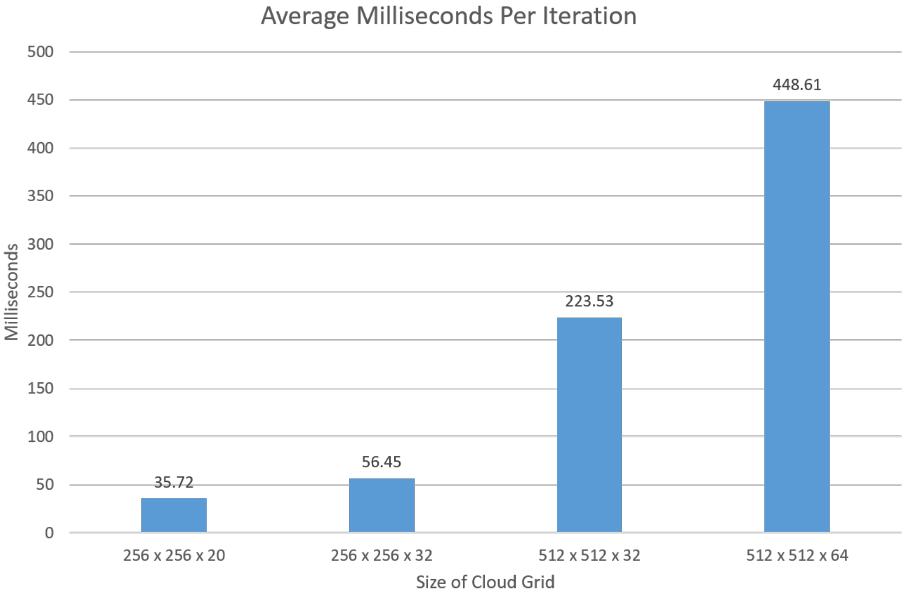
3.1 Naive Implementation

The basic pseudocode for the naive implementation is as follows:

```
// Kernel Function
updateCellNaive(char* src_buffer, char* dst_buffer){
    char cell = // read cell from src_buffer

    char new_cell = // update cell using formula (requires reading additional cells)

    // write new_cell to dst_buffer
}

// swap src and dst buffers after each iteration
```

3.2 Naive Implementation Results

The performance of the naive implementation is in the following chart:



Note that the 256x256x20 size is based on the size of the cloud grid in the original paper. For this size, the naive implementation completes one iteration in 35.72 milliseconds on average. This is just around 30 frames per second, but that does not account for rendering time. In order to be capable of realtime (60 fps) rendering, it is necessary to improve the performance of the simulation.

For the larger grid sizes, we see a roughly linear increase in run time.

3.3 Implementation Using Shared Memory

Cellular automata simulations generally have extremely simple update rules that need to be applied over large amounts of cells. Because of this, they usually tend to be bandiwidth bound. Thus the first attempt at optimizing the performance of the simulation is to use shared memory on the GPU to reduce the number of global memory accesses.

As mentioned previously, to update each cell, we need to access it and 11 of its neighbors, resulting in a total of 12 global memory accesses in the naive version. Equivalently, each cell will be accessed a total of 12 times: once when updating itself, and 11 more times when its neighbors are updating. By using shared memory, we can turn 11 of these global memory accesses into shared memory accesses, which greatly reduces the bandwidth that the simulation uses.

3.4 First Attempt

Here is the pseudocode for the first attempt at using shared memory.

```
// Inside of Kernel

// Note that this is hard-coded for a 32x1x32 block size, so I need to load a 36x4x36 block into shared memory

// Also, in this version, "y" is the vertical direction, but in future versions, "z" is the vertical direction

for (int i = 0; i < 4; i++) { // For each "y" layer
  if (y >= 0 && y < dimY) { // Boundary conditions
   if (x1 >= 0 && z1 >= 0 && x1 < dimX && z1 < dimZ) { // Boundary conditions
    idx1 = // compute correct global index
    idx2 = // compute correct shared memory index
    sharedMem[idx2] = src_buf[idx1];
   }

   int x2 = x1 + 32; // Coordinate for loading additional cells
   int z2 = z1 + 32; // Coordinate for loading additional cells

   if (threadIdx.x < 4 && threadIdx.z < 4) { // Load an additional element
    if (x2 >= 0 && z2 >= 0 && x2 <= dimX && z2 <= dimZ) { // Boundary conditions
     idx1 = // compute correct global index
     idx2 = // compute correct shared memory index

     sharedMem[idx2] = src_buf[idx1];
    }
   }

   if (threadIdx.z < 4) { // Load an additional element
```

```
        if (x1 >= 0 && z2 >= 0 && x1 <= dimX && z2 <= dimZ) { // Boundary conditions
          idx1 = // compute correct global index
          idx2 = // compute correct shared memory index

          sharedMem[idx2] = src_buf[idx1];
        }
      }

      if (threadIdx.x < 4) { // Load an additional element
        if (x2 >= 0 && z1 >= 0 && x2 <= dimX && z2 <= dimZ) { // Boundary conditions
          idx1 = // compute correct global index
          idx2 = // compute correct shared memory index

          sharedMem[idx2] = src_buf[idx1];
        }
      }
  }
}

__syncthreads(); // Barrier

// Do simulation using shared memory instead of src_buffer
```

3.5 Result and Analysis of First Attempt at Shared Memory

This version of code did not run very well. In fact, it ran almost twice as slow as the naive version. After analyzing this kernel, I have came up with two reasons for why this might be slow:

1. The block size is 32x1x32, which requires loading 36x4x36 cells. This is slightly over 5 times the number of cells that we are actually updating. Switching to a block size of 8x8x8 may be better, since that would only require loading 12x11x12 cells, which is slightly over 3 times the number of cells.

2. In the above pseudocode, certain threads will have to load 16 memory addresses. Because of the barrier at the bottom, this means that the effective latency of this operation is 16 reads, which is greater than the 12 reads that the naive version needs to do.

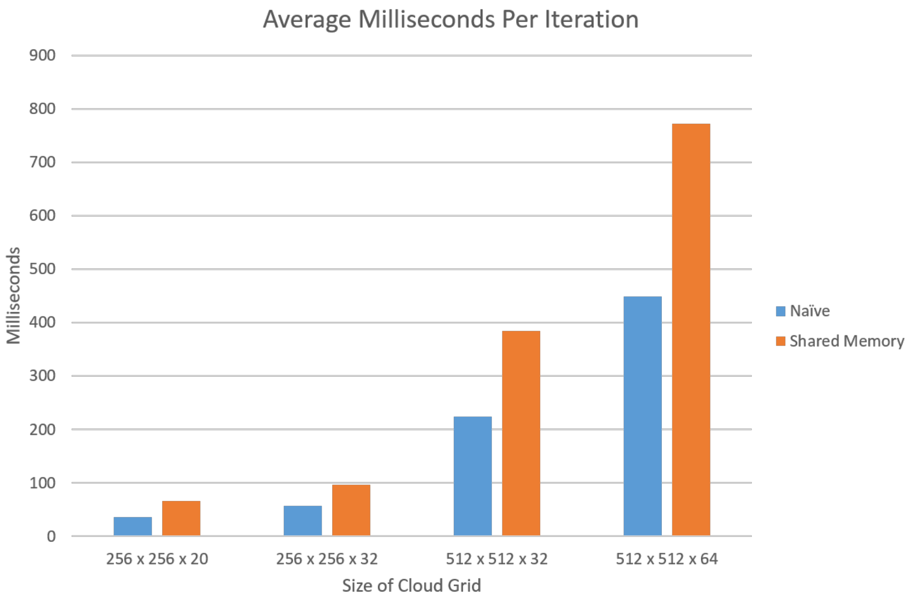
3.6 Second Attempt

Here is some pseudocode that addresses the problems with the first implementation.

```
// Using 8x8x8 block size
for (int iter = 0; iter < 4; iter++) {
    // load the iter-th group of 512 elements:

    if (idx < 1584) { // Shared memory is only 1584 (12 x 12 x 11) elements
        // Calculate approiate indices and load into shared memory
    }
}

__syncthreads();

// Do simulation using shared memory
```

This code is much better at distributing the work evenly, since each thread loads 3 or 4 cells each. Then the code should be faster now, right?

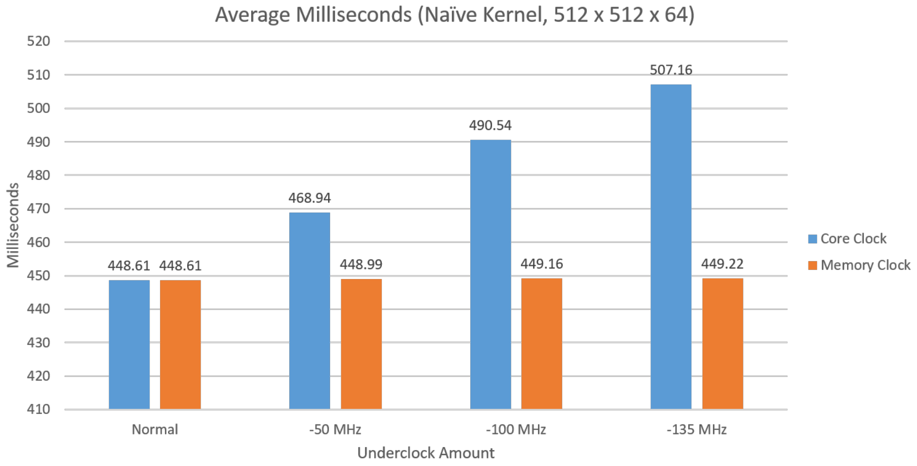
3.7 Result of Second Attempt



Well, the result is still not faster than the naive version. In fact, the performance is still roughly the same. At this point, it might be reasonable to consider the possiblity of the simulation being compute bound instead of bandwidth bound.

3.8 Testing Whether the Kernel is Compute or Bandwidth Bound

In order to confirm that the kernel is compute bound, I measured the performance of the naive implementation while using MSI Afterburner to underclock my GPU's clock rate and memory rate[1]. Here are the results:



In the above graph, I measured the time it took for the naive kernel to finish 1 iteration of a 512x512x64 cloud grid under two conditions:

1. Decreasing processor rate while keeping memory rate normal (Blue)
2. Decreasing memory rate while keeping processor rate normal (Orange)

[1] Unfortunately, due to some driver problems, the CUDA profiler could not analyze the kernels in enough detail.

From this graph, we can see that the performance of the kernel remained the same when we decreased the memory rate, and the performance decreased when we decreased the processor rate. This is strong evidence that the kernel is compute bound, since if it were bandwidth bound, decreasing the memory rate should have a negative impact on performance and decreasing processor rate should have no effect on performance.

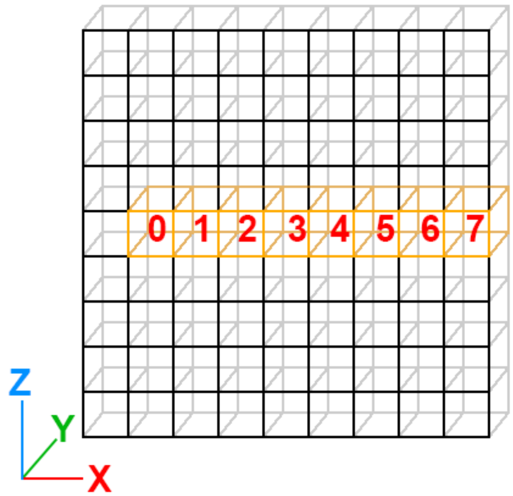
So because the kernel is compute bound, optimizing memory operations using shared memory is useless! In fact, because computing the correct indices for the elements in shared memory requires a significant amount of work, using shared memory actually hurts performance.

4 Optimization Attempt 2: Compact Cells

This optimization takes advantage of the fact that each cell only requires 3 bits of information. In the naive version, one byte is used to store each cell, but that leaves 5 bits unused per byte. It is easy to see how we can store two cells contiguously in one byte:

$$\underbrace{00000}_{\substack{\text{Unused}\\\text{bits}}}\underbrace{c_1a_1h_1}_{\text{Cell 1}} \Longrightarrow \underbrace{00}_{\substack{\text{Unused}\\\text{bits}}}\ \underbrace{c_1a_1h_1}_{\text{Cell 1}}\underbrace{c_2a_2h_2}_{\text{Cell 2}}$$

Using this method, we still have 2 unused bits per byte. But if we take a closer look at the update formula, we realize that during the simulation, the only information that each cell needs from its neighbors is their ACT bits. In addition, when we want to render the clouds, we only care about the CLD bit in each cell. Therefore, it makes sense to store these pieces of information in separate buffers. If we can do this, we can store the information for 8 cells in 1 byte (though we'll need 1 byte for each piece of information).



The above diagram shows a 8x1x1 block of cloud cells. We can store all of these cells using a total of 3 bytes across 3 separate buffers:

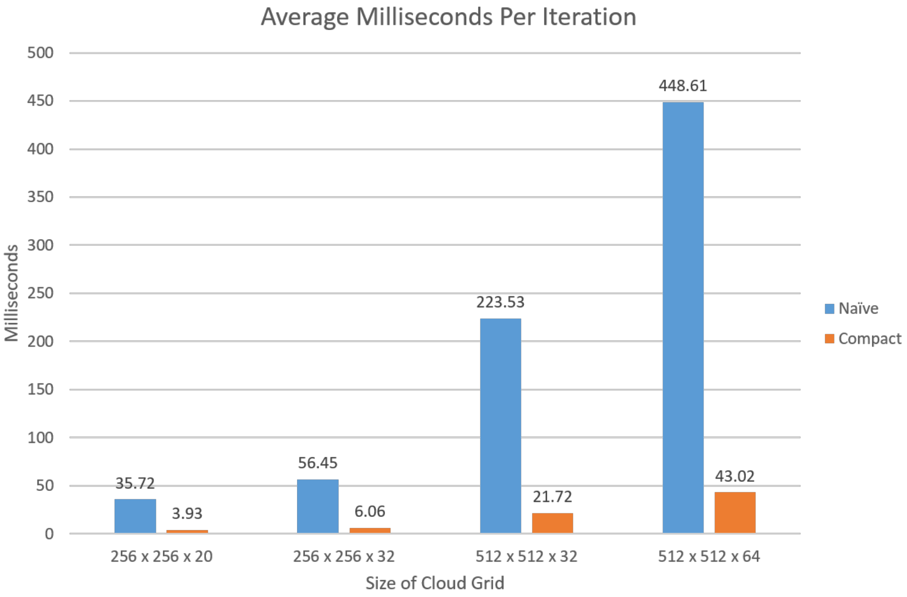
1. 1 byte in the HUM buffer: $h_0h_1h_2h_3h_4h_5h_6h_7$
2. 1 byte in the ACT buffer: $a_0a_1a_2a_3a_4a_5a_6a_7$
3. 1 byte in the CLD buffer: $c_0c_1c_2h_3c_4c_5c_6c_7$

But wait, why are we optimizing memory efficiency? Didn't we just confirm that the kernel is compute bound?

While it's true that this more compact storage improves memory efficiency, because the simulation uses boolean operations, we can use bitwise operators on each byte to update 8 cells at once. This roughly increases the number of computation that we can do by 8 times!

4.1 Results of 8x1x1 Compact Block Method

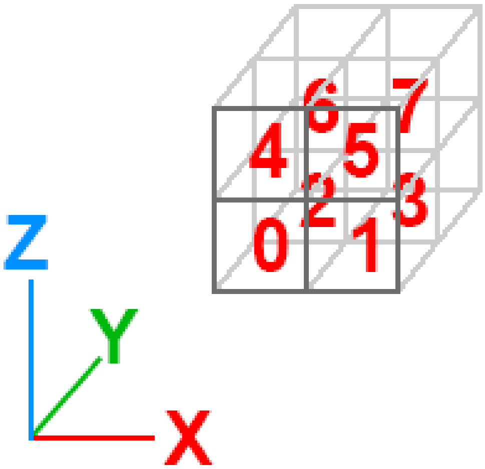
The following graph shows the performance of the compact block method vs the naive method:



As you can see, there is roughly a 10 times performance boost compared to the naive version. More importantly, if we wanted to have a 60 FPS cloud simulation on a 256x256x20 grid, we have over 12 milliseconds to do the rendering now.

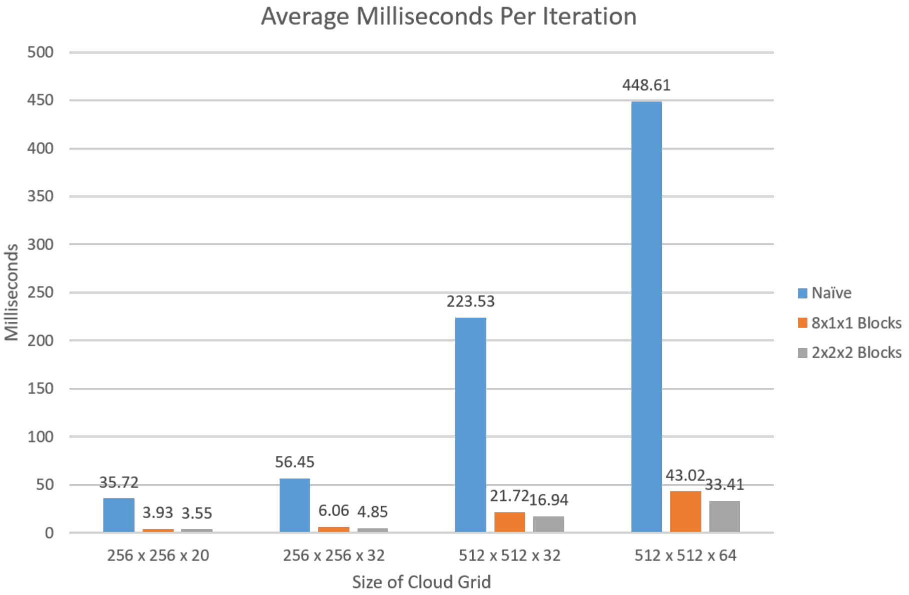
4.2 Using Different Block Shapes

Previously, I used one byte to store the information for an 8x1x1 block of cells. I decided to test out a 2x2x2 block just to see what would happen:



The memory layout of each byte is still the same as before, but the interpretation of the cells that are in each byte changed.

4.3 Results of 2x2x2 Compact Block Method



I was not expecting a difference in performance, but the 2x2x2 block is slightly faster than the 8x1x1 block. Why is this the case?

The first thought that comes to mind is that to update each cell in the 2x2x2 block method, we only have to get one neighboring block in each direction, while in the 8x1x1 method, we need to get two neighboring blocks in a few directions. But this doesn't make too much sense because the simulation is compute bound.

After examining the code a bit further, I noticed the following two pieces of code in the kernels:

8x1x1 Blocks

```
// 8x1x1
if (y - 2 >= 0)
    f |= src_act[xyzToIdx2(x, y - 2, z, d)];
if (y - 1 >= 0)
    f |= src_act[xyzToIdx2(x, y - 1, z, d)];
if (y + 1 < d.y)
    f |= src_act[xyzToIdx2(x, y + 1, z, d)];
if (y + 2 < d.y)

    f |= src_act[xyzToIdx2(x, y + 2, z, d)];
if (z - 2 >= 0)
    f |= src_act[xyzToIdx2(x, y, z - 2, d)];
if (z - 1 >= 0)
    f |= src_act[xyzToIdx2(x, y, z - 1, d)];
if (z + 1 < d.z)
    f |= src_act[xyzToIdx2(x, y, z + 1, d)];
```

2x2x2 Blocks

```
if (y - 1 >= 0) {
    char front = src_act[xyzToIdx2(x, y - 1, z, d)];
    f |= front;
    f |= (front >> 2) & 0x33;
}
f |= (a & 0x33) << 2;

if (y + 1 < d.y) {
    char back = src_act[xyzToIdx2(x, y + 1, z, d)];
    f |= back;
    f |= (back & 0x33) << 2;
}
f |= (a >> 2) & 0x33;

if (z - 1 > 0) {
    char lower = src_act[xyzToIdx2(x, y, z - 1, d)];
    f |= lower;
    f |= (lower >> 4) & 0xF;
}
f |= (a & 0xF) << 4;

if (z + 1 < d.z) {
    char upper = src_act[xyzToIdx2(x, y, z + 1, d)];
    f |= (upper & 0xF) << 4;
}
f |= (a >> 4) & 0xF;
```

The above code is the part that obtains information from neighboring cells in the forward, backward, up, and down directions. As a result of having to perform less memory operations in the second case, we actually save on some boundary condition checks too. The 8x1x1 method needs to perform 7 boundary condition checks, but the 2x2x2 method only needs 4 boundary condition checks. This leads to less code divergence in the 2x2x2 method, which can explain why it performs better.

5 References

5.1 Main References

Original Paper
How to tell if a kernel is memory or compute bound

5.2 General CUDA/GPU Info

CUDA Documentation
CUDA Memory and Cache Architecture

5.3 Specifications

Intel i7-4720HQ
NVIDIA GTX 960M