Cellular Automata Cloud Simulation

Daniel Hua

Why is this Problem Interesting?

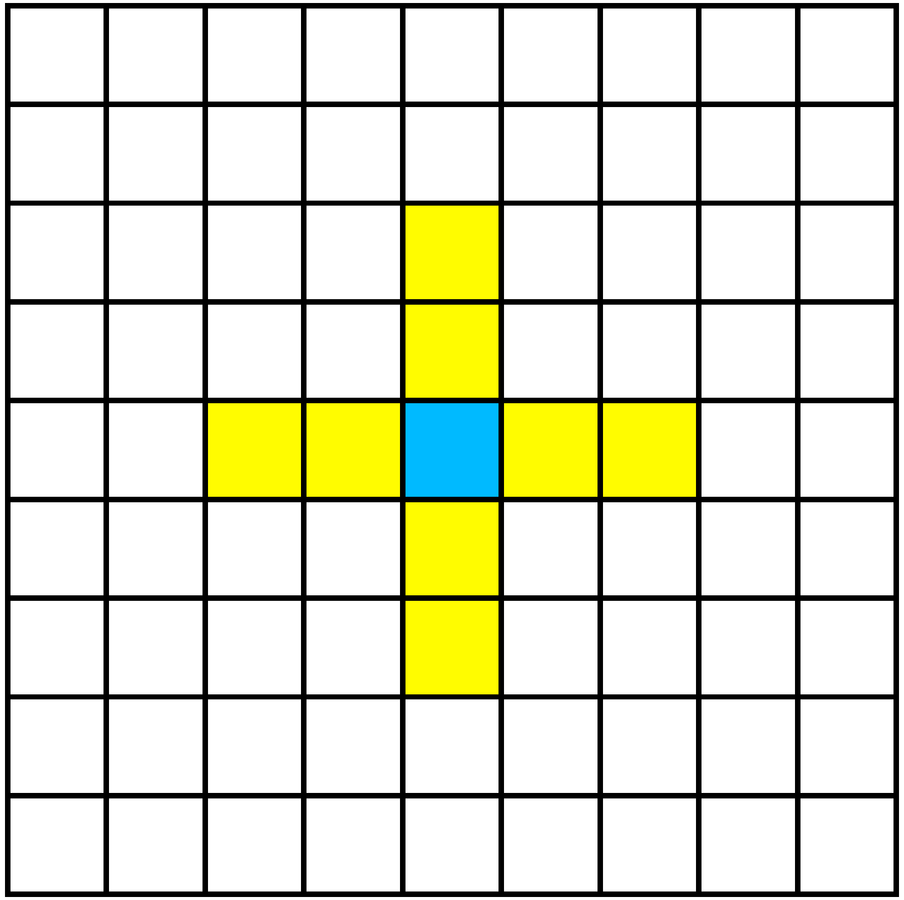
- 3D Cellular Automata – Bandwidth Bound?

- Slightly “Irregular” Memory Accesses for Each Cell

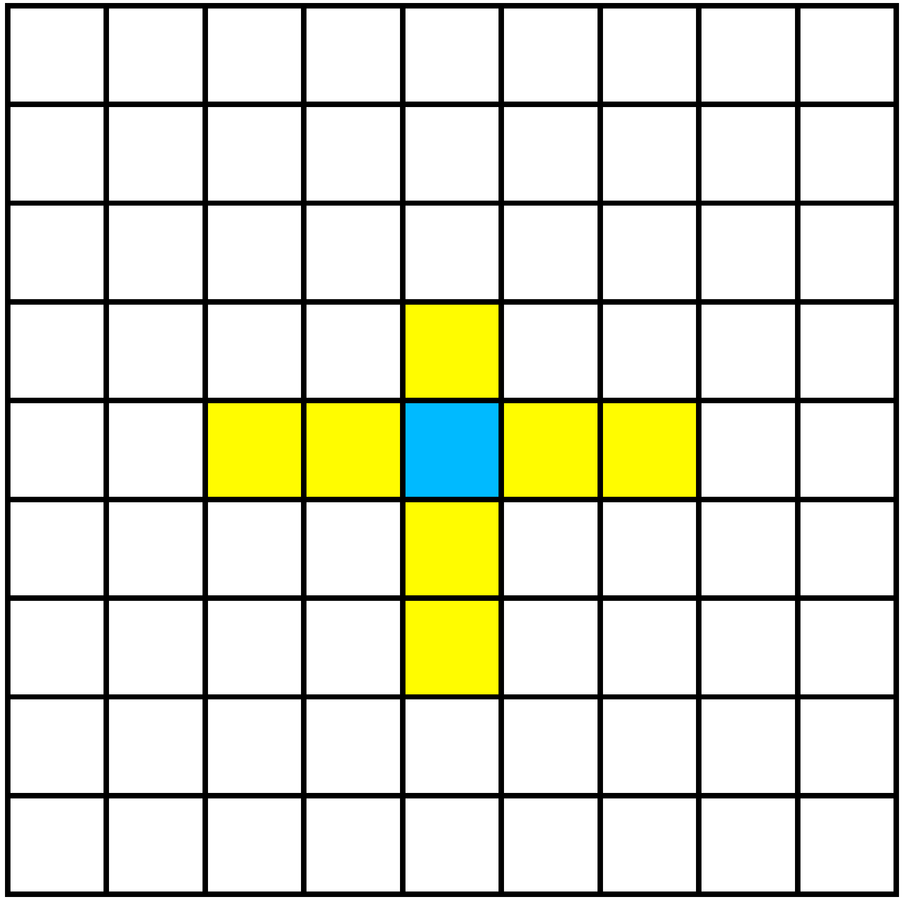
- Real Time Cloud Simulation can be used in Games

Memory Accesses for Each Cell

Top View

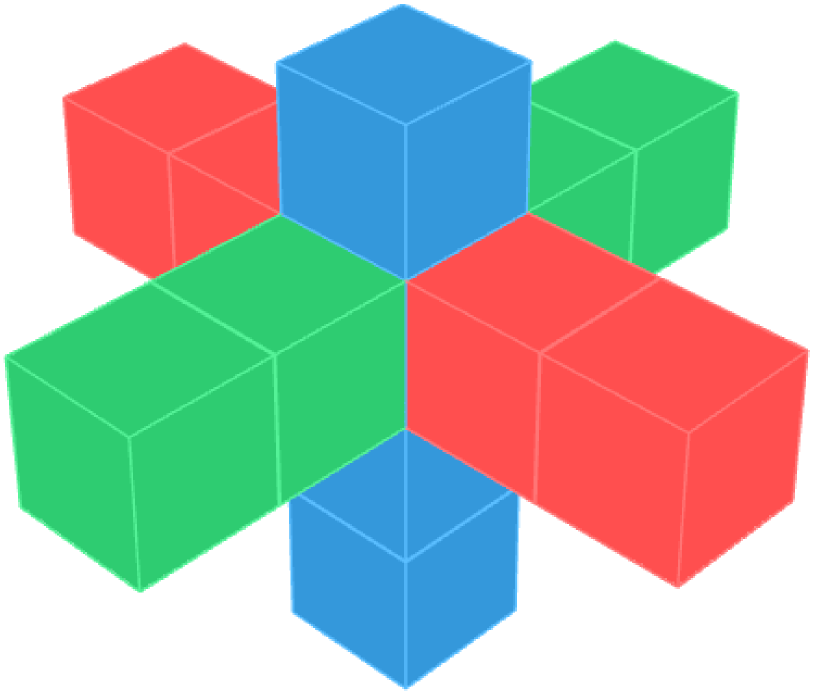


Side View



Memory Accesses for Each Cell

3D View



The Algorithm

- **Each Cell has 3 Bits of Information:**

- **<u>HUM (H)</u>: Is this cell humid enough to form clouds?**

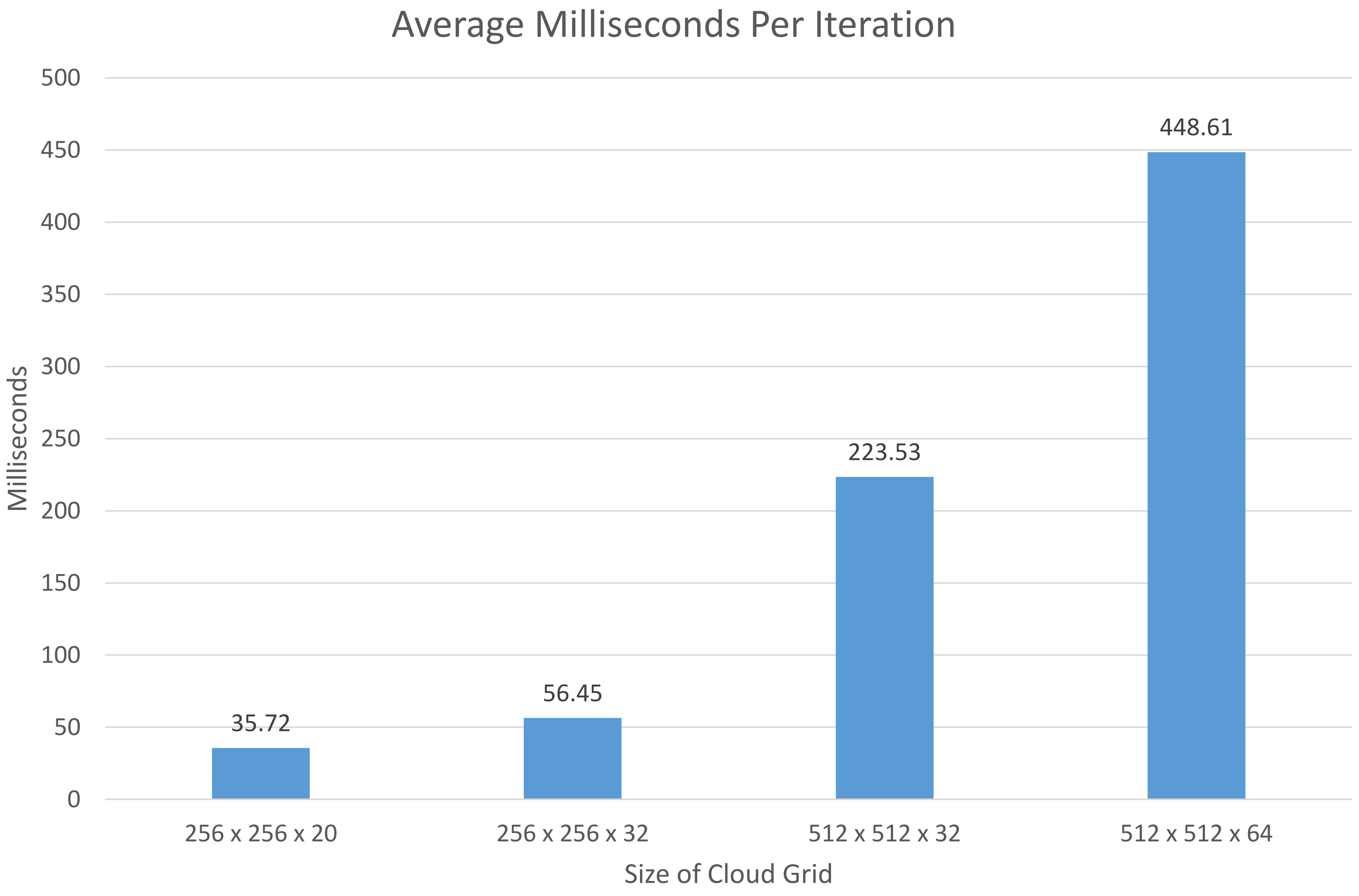
- **<u>ACT (A)</u>: Activation factor.**

- **<u>CLD (C)</u>: Is there a cloud in this cell?**

Naïve Implementation (Kernel Pseudocode)

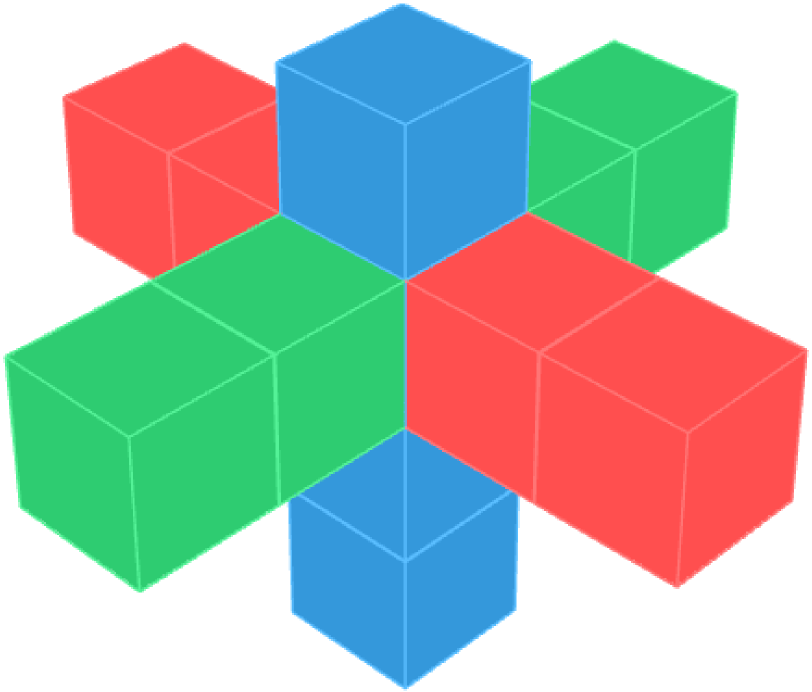
```
updateCellNaive(char* src_buffer, char* dst_buffer){
    char cell = // read cell from src_buffer


    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
```

Performance of Naïve Implementation



Optimization Attempt 1: Shared Memory

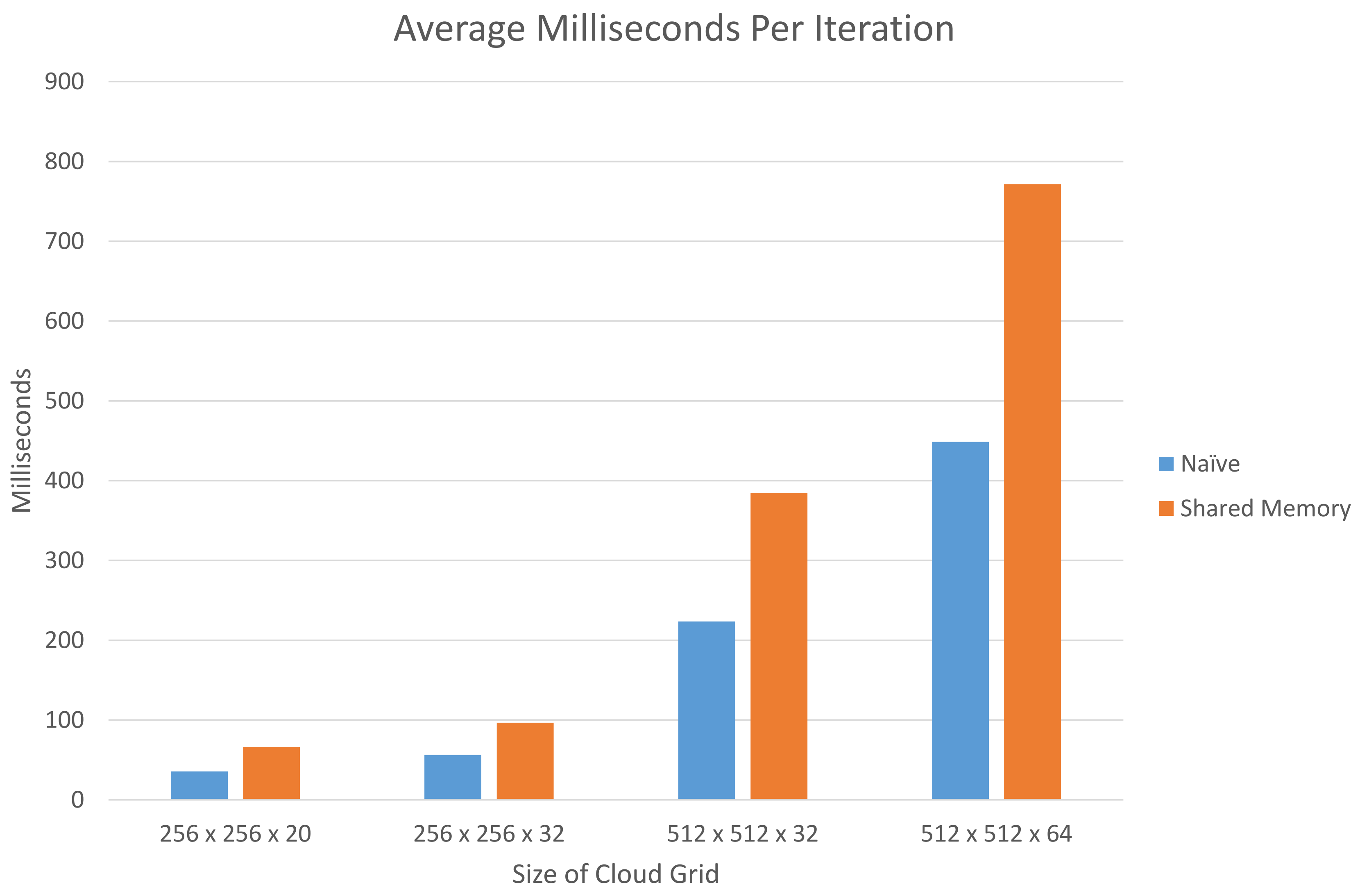
- Each cell requires accessing global memory 12 times for neighboring cells and itself.
- Using shared memory, we would only need to load each cell 1 time from global memory.



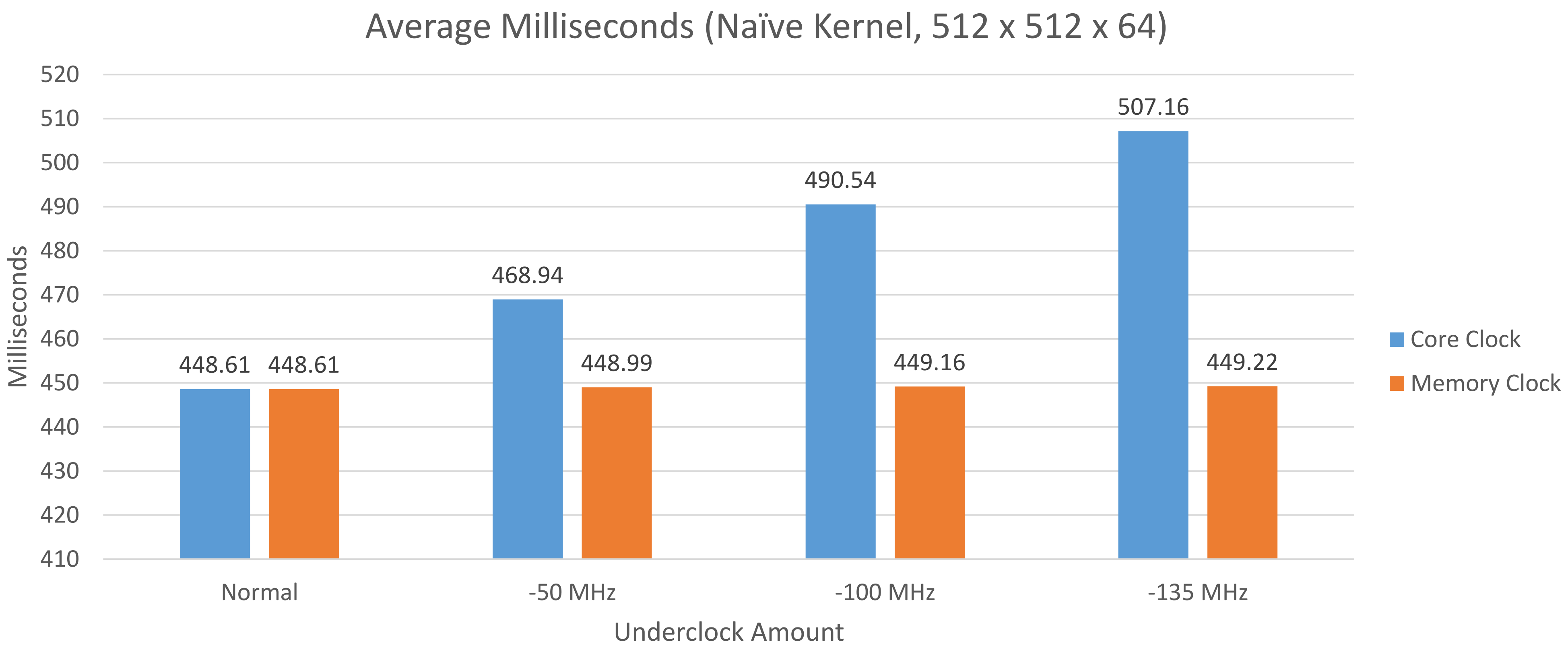
Shared Memory Pseudocode (First Attempt)

```
updateCellNaive_SharedMem(char* src_buffer, char* dst_buffer){
    __shared__ sharedMem[]; // statically allocated
                            // (hard coded for block size)

    // load elements into shared memory

    __syncthreads();


    // update as before, except read from shared memory instead

    // write new_cell to dst_buffer
}

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

Shared Memory Makes the Performance Worse?



What Are We Bound By?



- Less Bandwidth → Same Performance
- Less Compute → Lower Performance
- We are <u>compute bound</u>

Why are We Compute Bound?

- The simulation involves quite a bit of operations

- Computing the correct indices for using shared memory is even more work

$$hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i) \wedge \neg act(i,j,k,t_i)$$

$$cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i) \vee act(i,j,k,t_i)$$

$$act(i,j,k,t_{i+1}) = \neg act(i,j,k,t_i) \wedge hum(i,j,k,t_i) \wedge f_{act}(i,j,k)$$

$$\begin{aligned} f_{act}(i,j,k) = {} & act(i+1,j,k,t_i) \vee act(i,j+1,k,t_i) \\ & \vee act(i,j,k+1,t_i) \vee act(i-1,j,k,t_i) \vee act(i,j-1,k,t_i) \\ & \vee act(i,j,k-1,t_i) \vee act(i-2,j,k,t_i) \vee act(i+2,j,k,t_i) \\ & \vee act(i,j-2,k,t_i) \vee act(i,j+2,k,t_i) \vee act(i,j,k-2,t_i) \end{aligned}$$

Source: http://evasion.imag.fr/~Antoine.Bouthors/research/dea/sig00_cloud.pdf

So What Now?

- Let’s try to reduce the overall computation needed.

Optimization Attempt 2: Compact Storage

- Each cell only requires 3 bits to store

- The naïve kernel uses 1 byte to store each cell. That's wasting 5 bits per cell!
  - Current bit layout (1 byte): 00000CAH

- We can easily store 2 cells in each byte (still wastes 2 bits):
  - $00C_1A_1H_1C_2A_2H_2$

There's a Better Solution

- The only information that you need to render a cloud is the CLOUD bit of each cell

- The only information you need from adjacent cells is the ACTIVE bit

- Use separate HUM, ACT, and CLD buffers.

$$hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i) \wedge \neg act(i,j,k,t_i)$$

$$cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i) \vee act(i,j,k,t_i)$$

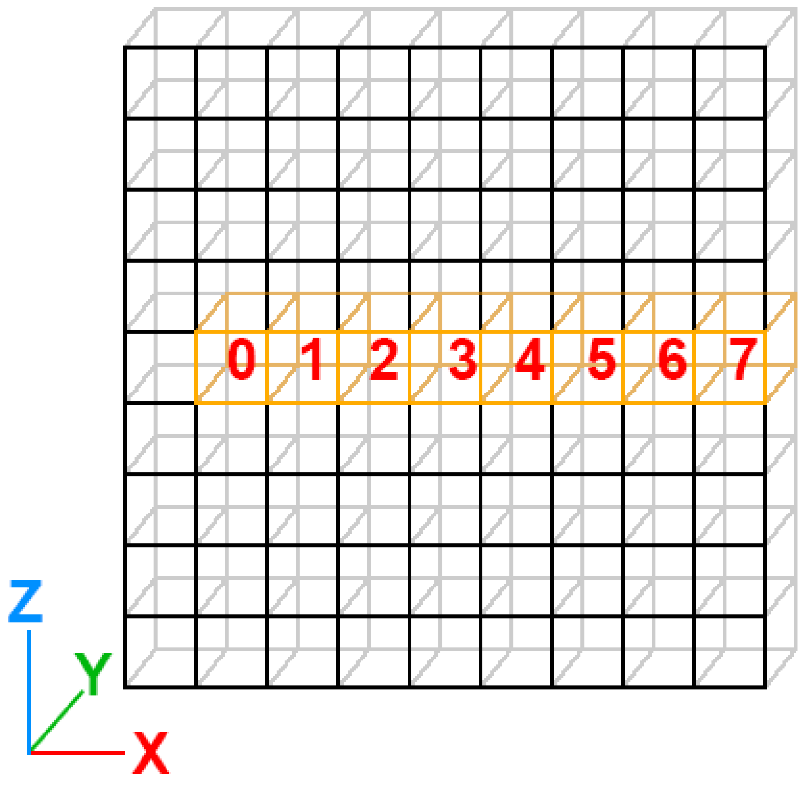
$$act(i,j,k,t_{i+1}) = \neg act(i,j,k,t_i) \wedge hum(i,j,k,t_i) \wedge f_{act}(i,j,k)$$

$$\boxed{\begin{aligned} f_{act}(i,j,k) = {} & act(i+1,j,k,t_i) \vee act(i,j+1,k,t_i) \\ & \vee act(i,j,k+1,t_i) \vee act(i-1,j,k,t_i) \vee act(i,j-1,k,t_i) \\ & \vee act(i,j,k-1,t_i) \vee act(i-2,j,k,t_i) \vee act(i+2,j,k,t_i) \\ & \vee act(i,j-2,k,t_i) \vee act(i,j+2,k,t_i) \vee act(i,j,k-2,t_i) \end{aligned}}$$

Separate Buffers Attempt 1 (8x1x1 Blocks)

- Layout of a byte in the buffers:
  - $C_0C_1C_2C_3C_4C_5C_6C_7$
  - $H_0H_1H_2H_3H_4H_5H_6H_7$
  - $A_0A_1A_2A_3A_4A_5A_6A_7$
- No bits are unused

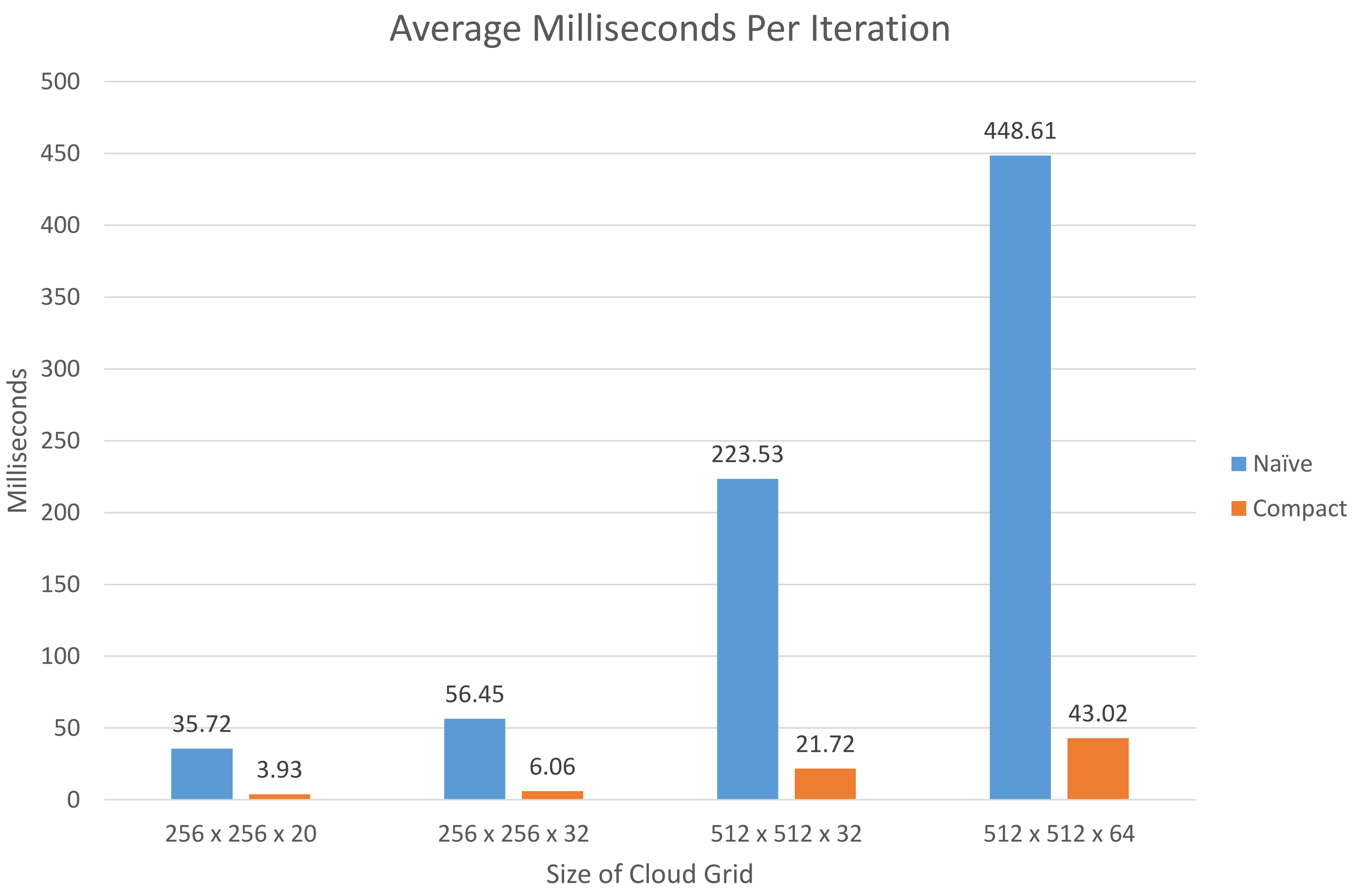
Cloud Grid



Wait, Why are We Reducing Memory?

- Using separate buffers, we can actually store the information from 8 cells in 1 byte
- Then using bitwise operations, we can update 8 cells at a time

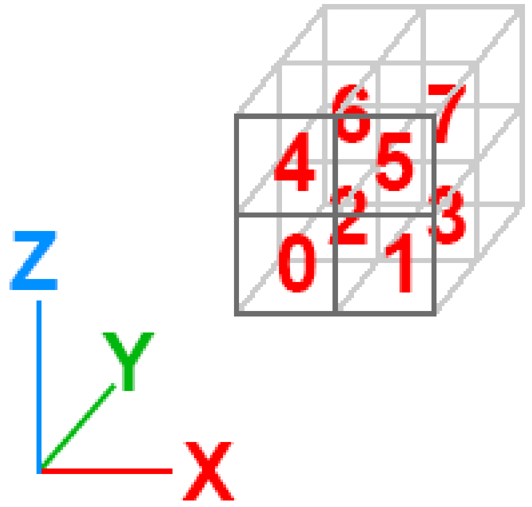
Using 8x1x1 Blocks Results in Speedup



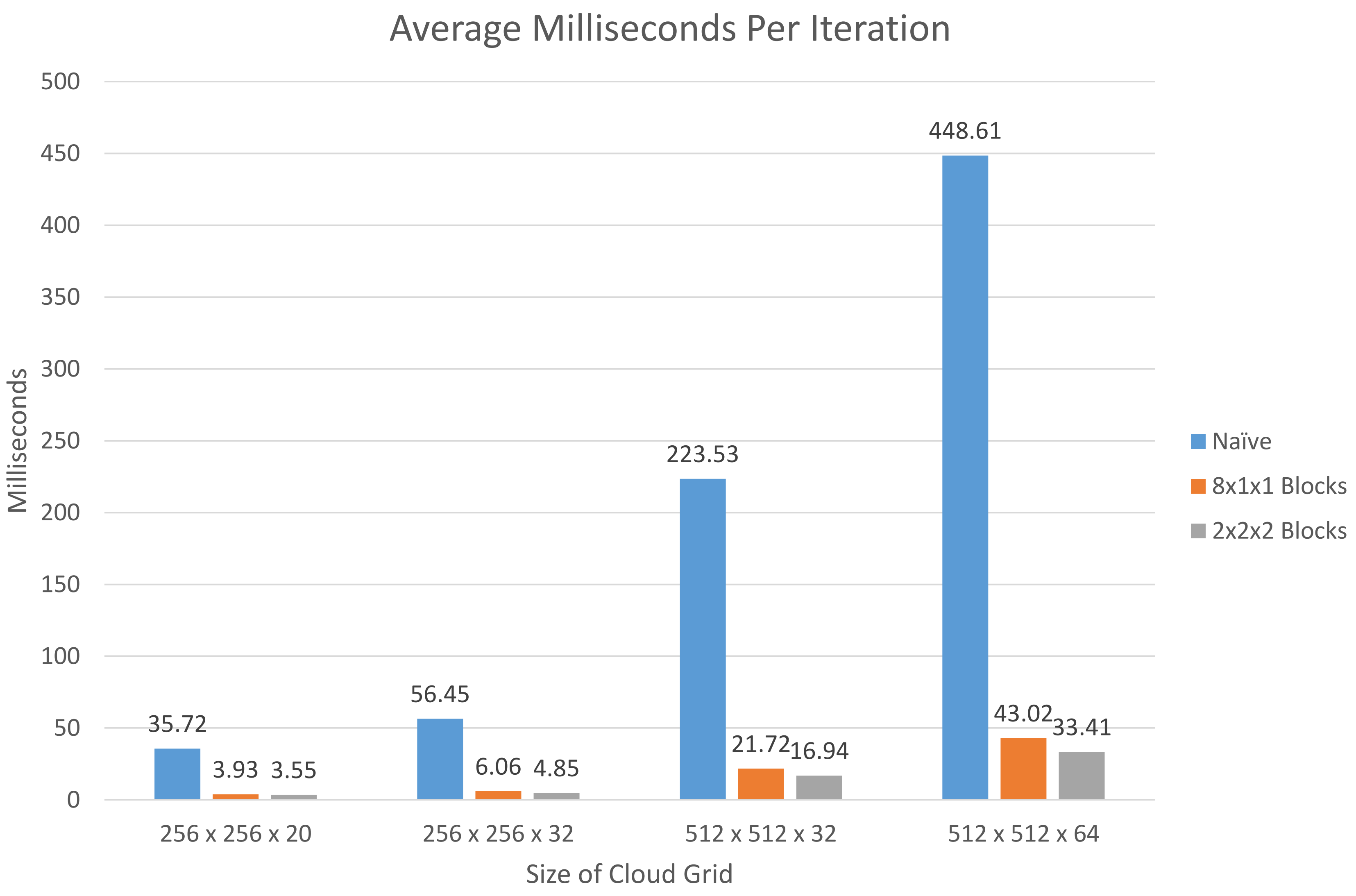
What About a Different Block Shape?

- Layout of a byte in the buffers:
  - $C_0C_1C_2C_3C_4C_5C_6C_7$
  - $H_0H_1H_2H_3H_4H_5H_6H_7$
  - $A_0A_1A_2A_3A_4A_5A_6A_7$

2x2x2 Block



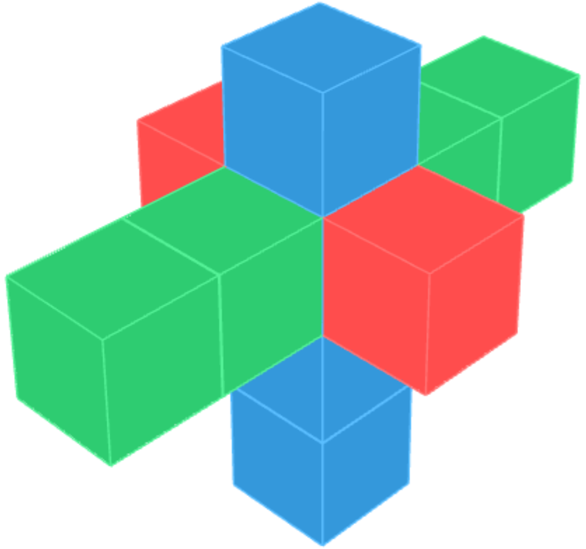
2x2x2 Blocks Perform Better Than 8x1x1 Blocks



Why is 2x2x2 Better?

Less Memory Accesses?

8x1x1 Block



11 Total

2x2x2 Block

7 Total

More Code Divergence for Checking Boundary Conditions

8x1x1 Block

```
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 Block

```
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;
```

Questions?