Real-time particle-based fluid simulation

Peter Lef

ThinkPad P53
i7-9750H @ 2.60 GHz
Quadro T1000 (4 GB)
2x8 GB DDR4

Videogames

- Need to:
  - Process user input
  - Update world state
  - Draw result on screen
- Want to do this fast.
  - I’m aiming for at least 60 FPS: 16.6 ms per frame
  - Missing frame presentation deadlines causes input lag and stutter
  - < 33 ms (30 FPS) is often noticeably slow
  - < 66 ms (15 FPS) is often unplayable

I got distracted

- This game is supposed to have fluid physics
- My fluid “simulation” was too slow
- Made it faster
  - And faster
    - And faster
      - and forgot about the rest of the game
- This talk is about accelerating a fluid sim

The fluid sim

- Particle-based
  - Particles exert force on other particles near them
- Not based on real physics
  - In games, we can get away with “believable” or “artistic”
  - But the same broad optimization principles apply
- Old demo: https://youtu.be/Fpx_WgC9fJI

Too slow

- Feb 24:
  - 1k particles: ~5 ms (200 FPS)
  - 10k particles: ~190 ms (5 FPS)

The algorithm is stupid

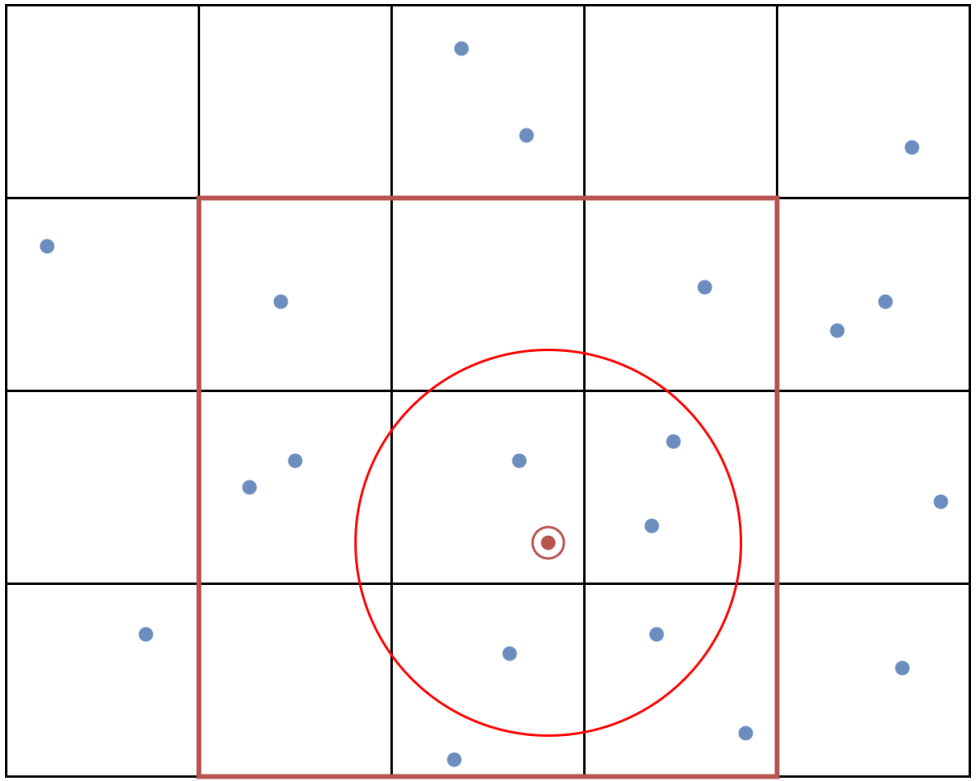
```
for p_a in particles:
   for p_b in particles:
      if distance(p_a, p_b) < interaction_radius:
         compute force on p_a by p_b
```

Most particles will not be within the interaction radius of a given particle.
So checking every particle is a waste of processing time.

Let’s use a data structure that allows us to quickly get a list of nearby particles, and only check those.

Spatial partitioning (1)

Paper: “Multi-Level Memory Structures for Simulating and Rendering Smoothed Particle Hydrodynamics” by Winchenbach and Kolb (DOI: 10.1111/cgf.14090)



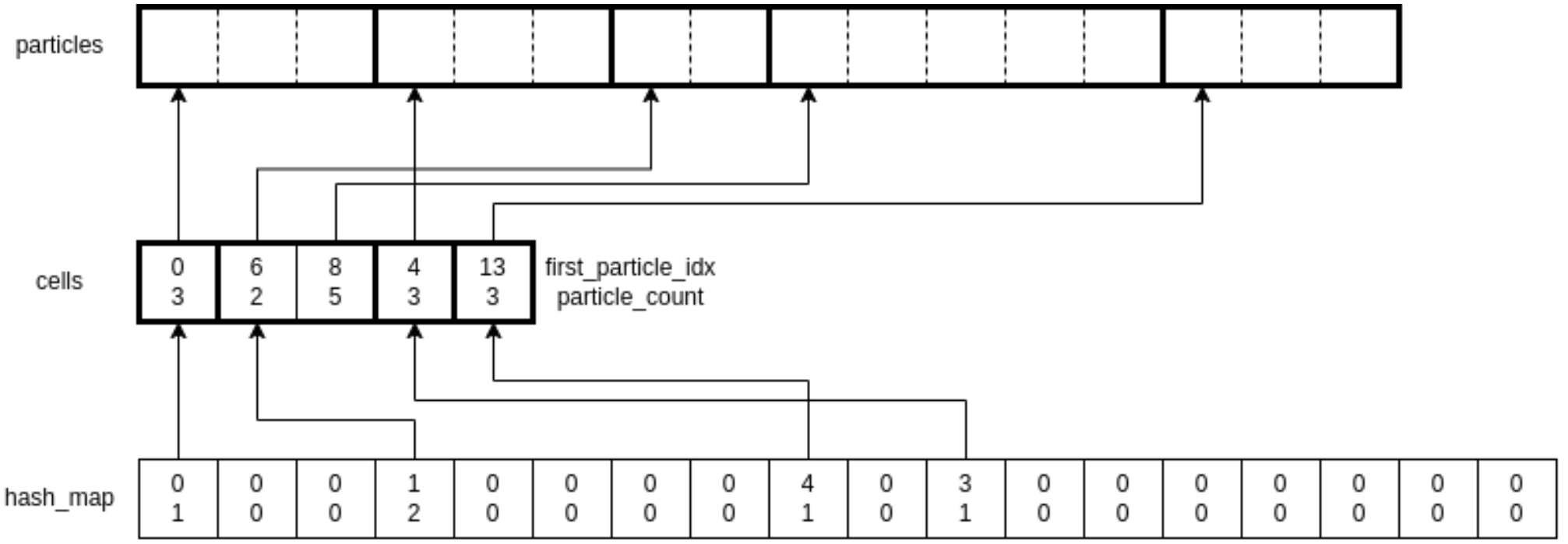
Split space into cells.
cell_width := particle_interaction_radius

For each particle, only check particles in the same cell and adjacent cells.

3x3x3 = 27 cells to check.

Only store cells that contain particles, so cell_count <= particle_count.

Spatial partitioning (2)



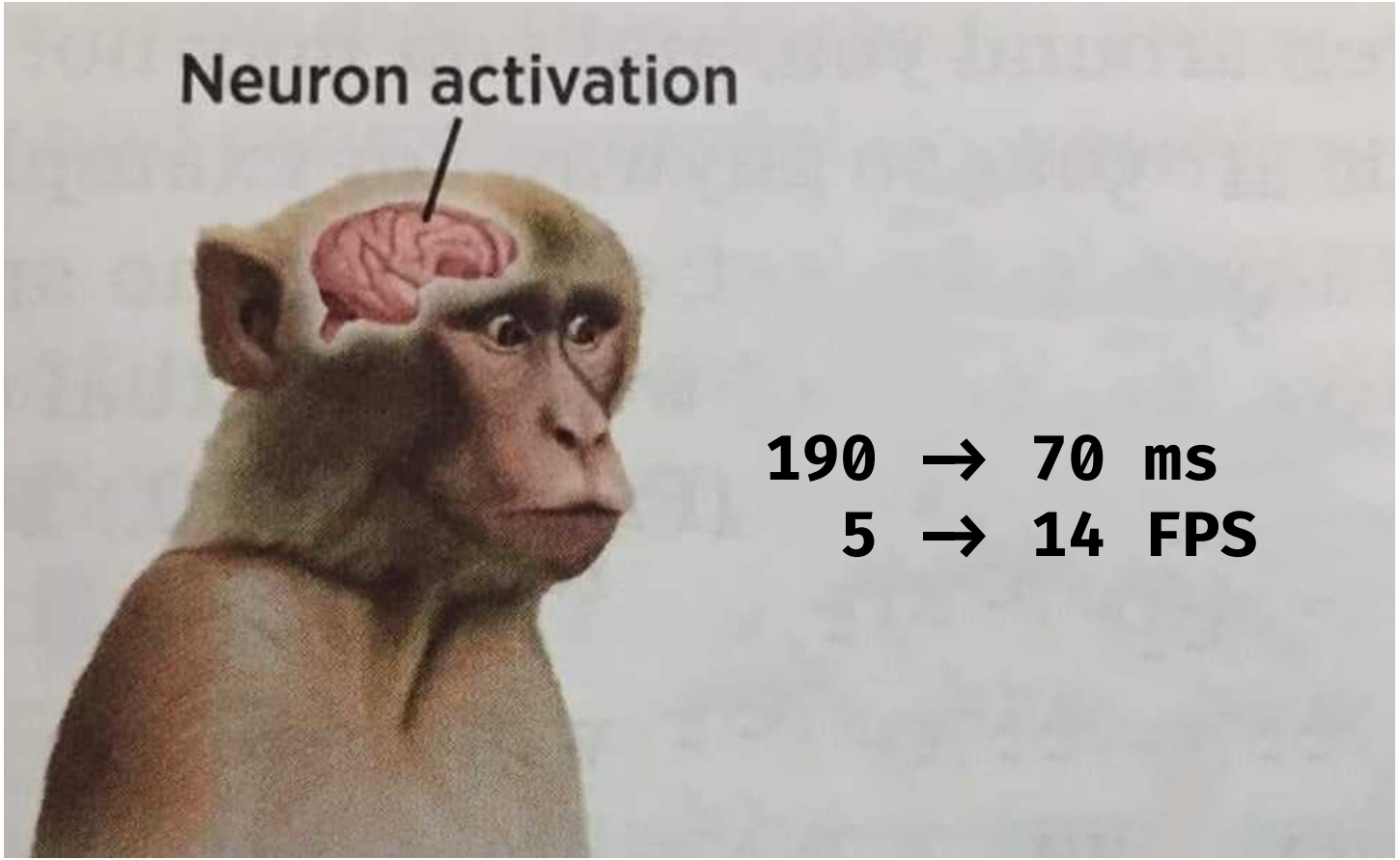
Hash function: hash_map[cell_morton_code % n],
where n := first prime number that is >= particle_count.

cell_morton_code can be computed from a cell's (or particle's) 3D position

Spatial partitioning (3)

- Result:
  - 1k particles: no change
  - 10k particles: 190 ms -> 70 ms (5 FPS -> 14 FPS)

Number goes up



commit 0f1982ff1f6dfc38ccf1c5432d4482e5fbb794f1

Use the GPU (1)

- GPUs are great for running parallelizable computations like this.
  - E.g. we can compute the acceleration for every particle in parallel.*

* parallelizable upto number of available execution units, latency-hiding opportunities, etc

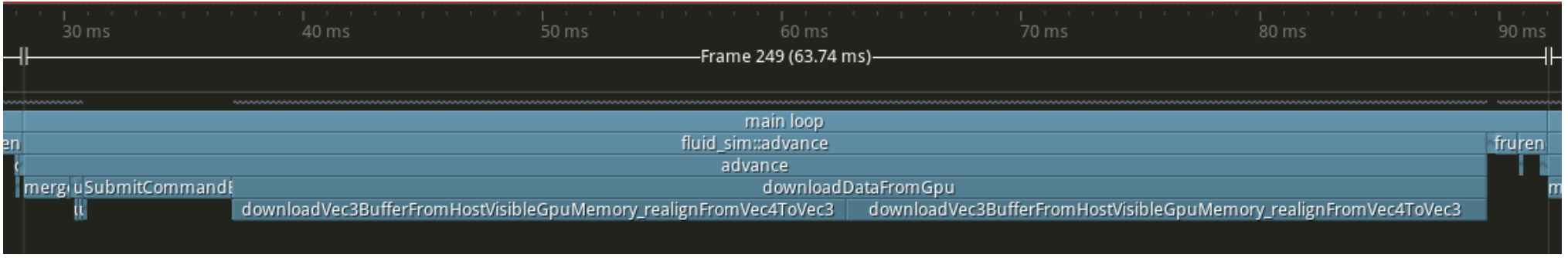
Use the GPU (2)

1. build hashmap on CPU
2. upload particles and hashmap to GPU
3. update particles on GPU
4. download particles to CPU

... no speedup

- 70 ms -> 70 ms
- ... wtf? GPU version is just as slow as single-core CPU version?

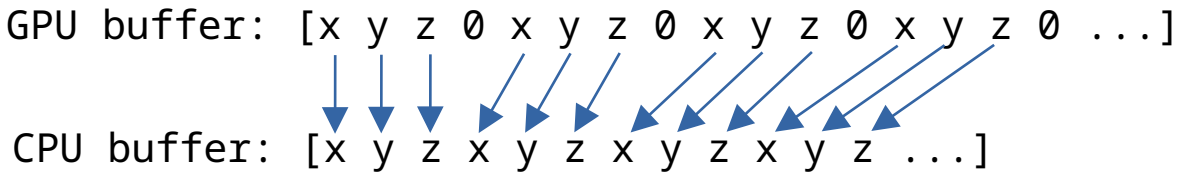
Slow data download



Most of the time is spent downloading the particle data.

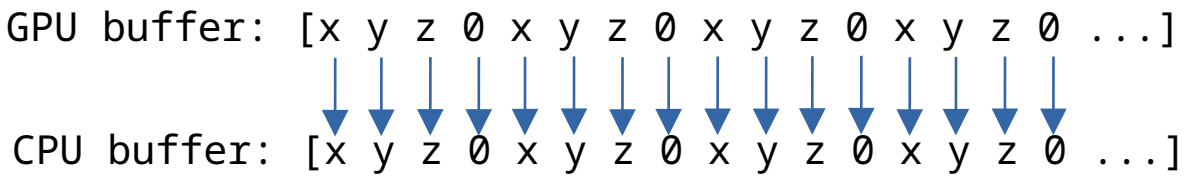
Data alignment

Problem: we are realigning data after the download.



(float3 aligned as float4 on GPU)

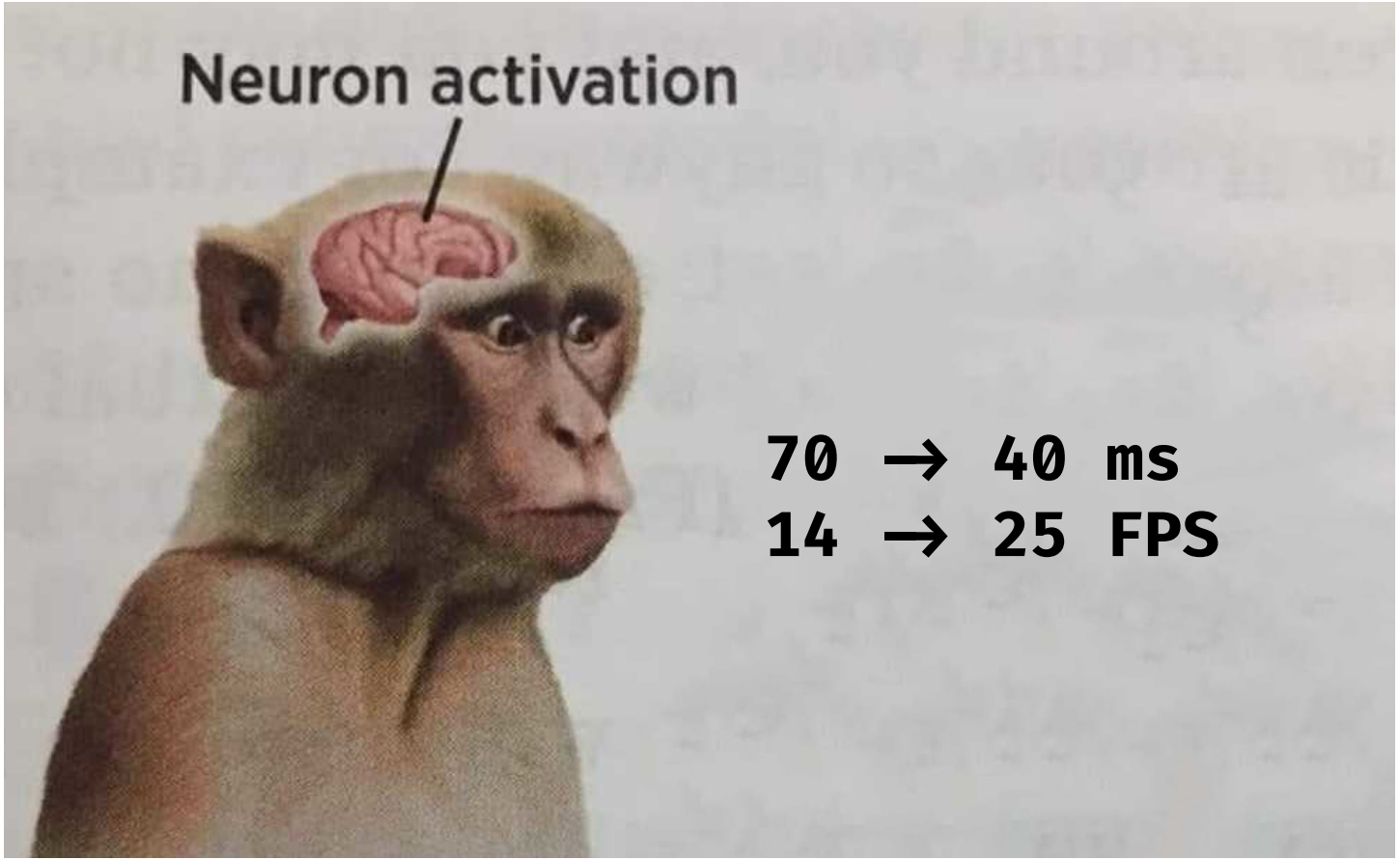
Solution: use float4 alignment on CPU



Now we can simply `memcpy()` the buffers, taking advantage of SIMD.

```
(gdb) bt
#0  0x00007ffff777cc07 in __memmove_avx_unaligned_erms () from /lib64/libc.so.6
#1  0x00007fffda9c0653 in fluid_sim::downloadBufferFromHostVisibleGpuMemory (dst
```

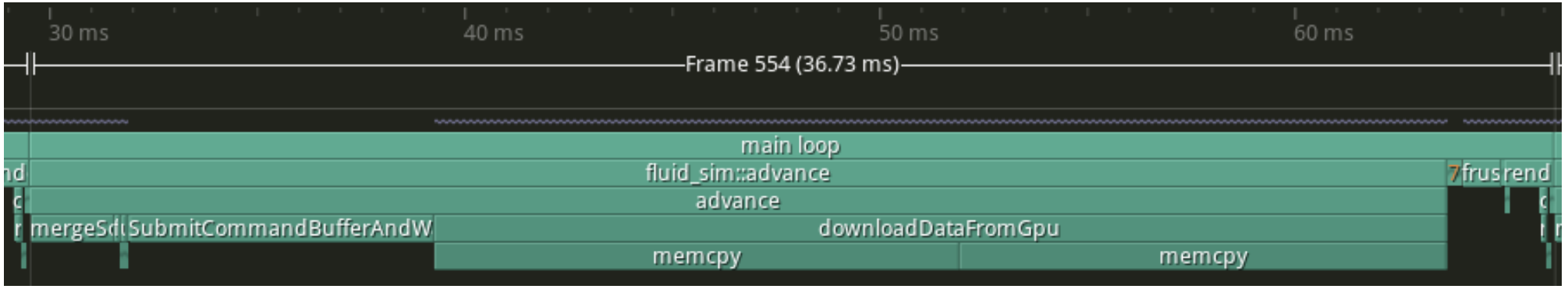
Number goes up



commit ff4f672e8977b713dbf6d71d29e8d102379475a7

Download still slow

- Most of the time is spent in the `memcpy()`



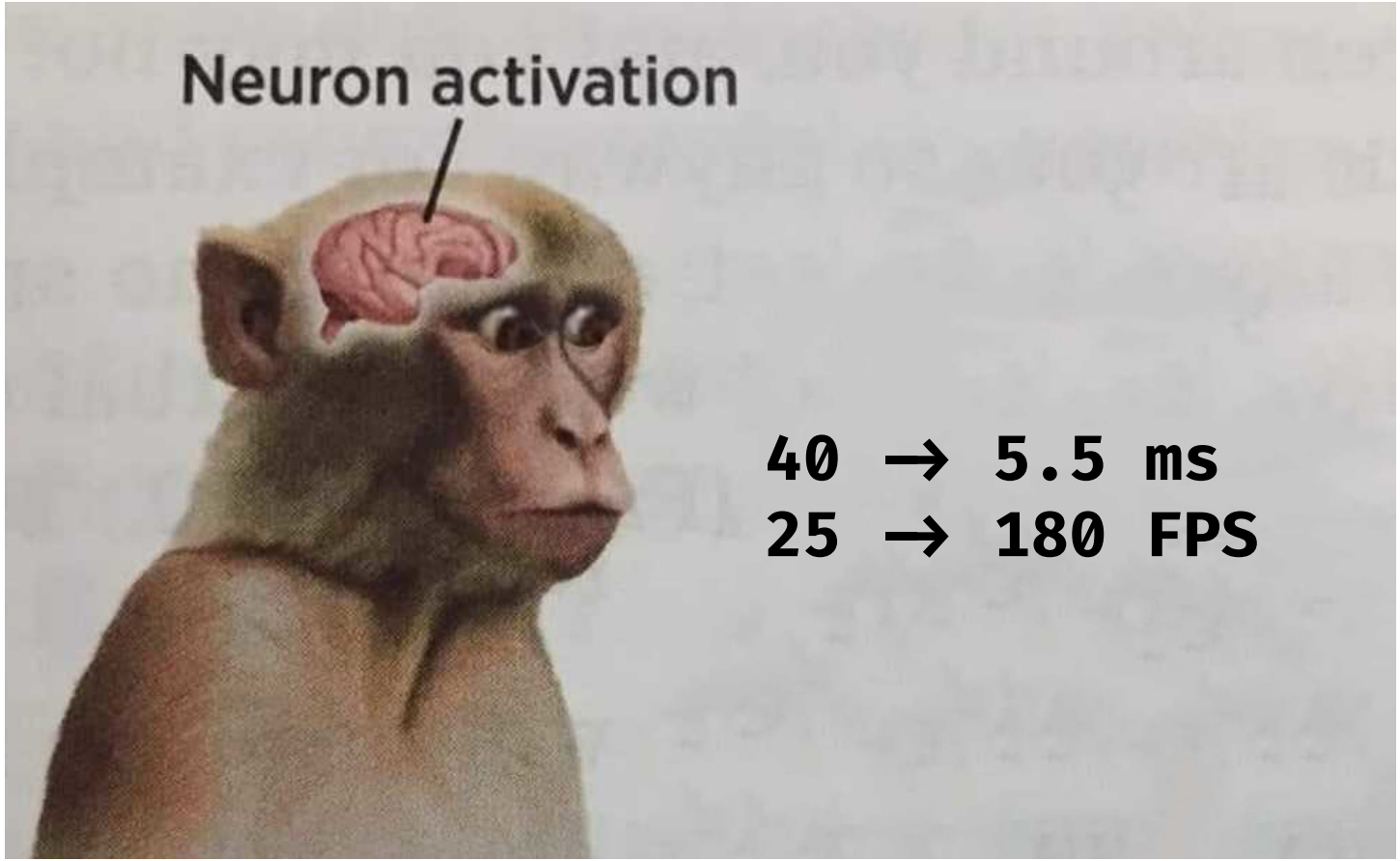
“GPU” buffer

- The GPU buffer is actually on the GPU.
- Reads by CPU happen over a slow data bus.
- Instead of copying the buffer from GPU to CPU, maybe let shaders write directly to a CPU buffer?

```
VmaAllocationCreateInfo buffer_alloc_info {
    .flags = VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL
    .usage = VMA_MEMORY_USAGE_AUTO,
    .requiredFlags = VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT,
    .usage = VMA_MEMORY_USAGE_AUTO_PREFER_HOST,
    .requiredFlags = VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT,
};
```

Old: VMA created a host-visible device-local buffer.
New: host-visible, allocated on host.

Number goes up



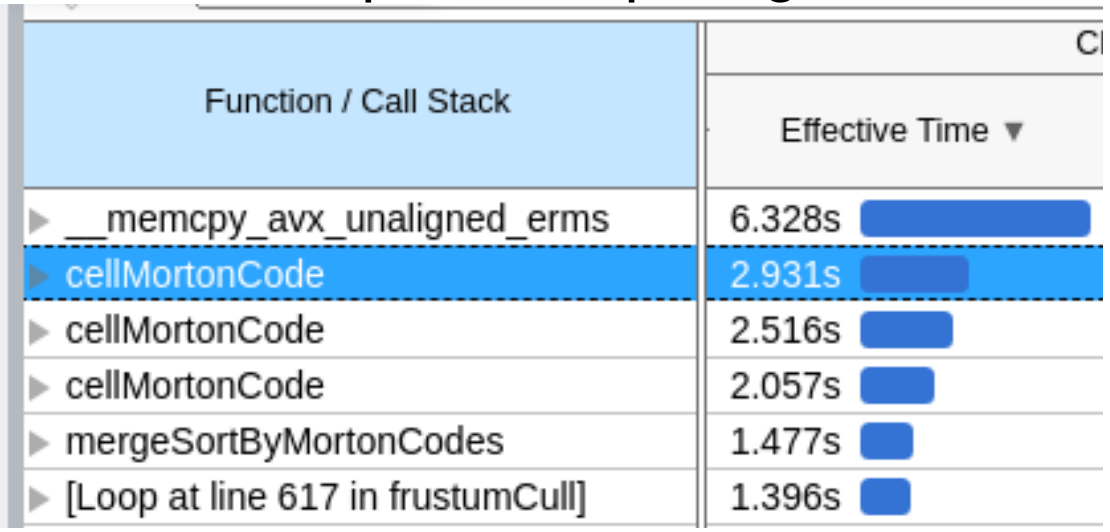
commit 04ad04009bbb64cbed9ea548fd1d23773479f88e

More particles

- 10k: 5.5 ms (180 FPS)
- 100k: 50 ms (20 FPS)

Surprise

Lot of time spent computing Morton codes



Function / Call Stack	Effective Time ▼
__memcpy_avx_unaligned_erms	6.328s
cellMortonCode	2.931s
cellMortonCode	2.516s
cellMortonCode	2.057s
mergeSortByMortonCodes	1.477s
[Loop at line 617 in frustumCull]	1.396s

Unexpected because Morton code computation is short

- But performed very frequently!

Only noticed because of sampling profiler (VTune)

Morton codes

- Maps a 3D cell index to an integer
- Computed by interspersing bits

```
3D cell index: {
    u32 x;
    u32 y;
    u32 z;
}
Morton code: u32 [ ... z2 y2 x2 z1 y1 x1 z0 y0 x0 ]
```

* Only uses 10 bits from each dimension. Top 2 bits are 0.

There’s an instruction for that

```
static inline u32 cellMortonCode(uvec3 cell_index) {
    return
        ((cell_index.x &    1) <<  0) |
        ((cell_index.y &    1) <<  1) |
        ((cell_index.z &    1) <<  2) |
        ((cell_index.x &    2) <<  2) |
        ((cell_index.y &    2) <<  3) |
        ((cell_index.z &    2) <<  4) |
        ((cell_index.x &    4) <<  4) |
        ((cell_index.y &    4) <<  5) |
        ((cell_index.z &    4) <<  6) |
        ((cell_index.x &    8) <<  6) |
        ((cell_index.y &    8) <<  7) |
        ((cell_index.z &    8) <<  8) |
        ((cell_index.x &   16) <<  8) |
        ((cell_index.y &   16) <<  9) |
        ((cell_index.z &   16) << 10) |
        ((cell_index.x &   64) << 10) |
        ((cell_index.y &   64) << 11) |
        ((cell_index.z &   64) << 12) |
        ((cell_index.x &  128) << 12) |
        ((cell_index.y &  128) << 13) |
        ((cell_index.z &  128) << 14) |
        ((cell_index.x &  256) << 14) |
        ((cell_index.y &  256) << 15) |
        ((cell_index.z &  256) << 16) |
        ((cell_index.x &  512) << 16) |
        ((cell_index.y &  512) << 17) |
        ((cell_index.z &  512) << 18) |
        ((cell_index.x & 1024) << 18) |
        ((cell_index.y & 1024) << 19) |
        ((cell_index.z & 1024) << 20) ;
}
```

<- naive version

using PDEP instruction

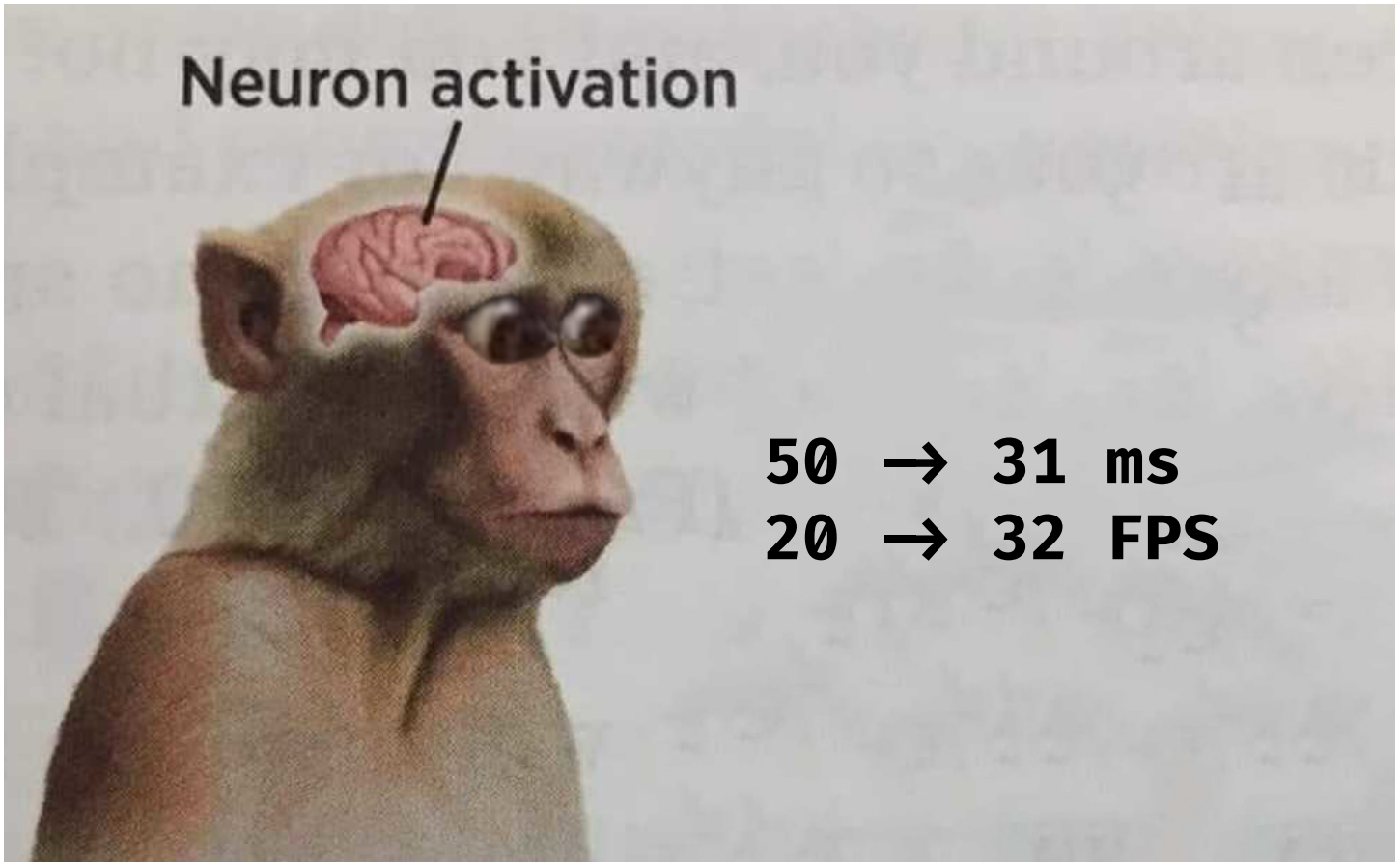
```
static inline u32 cellMortonCode(uvec3 cell_index) {

    u32 result = 0;
    result |= _pdep_u32(cell_index.x, 0b00'001001001001001001001001001001);
    result |= _pdep_u32(cell_index.y, 0b00'010010010010010010010010010010);
    result |= _pdep_u32(cell_index.z, 0b00'100100100100100100100100100100);
    return result;
}
```

Caveats:

- There is a faster-than-naive method without intrinsics: https://fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/
- PDEP only exists on x86 CPUs
- PDEP only exists on x86 CPUs with the BMI2 extension
- PDEP is slow as shit on AMD Zen 1 and Zen 2 (but fast on newer CPUs)
- We can probably use 1 fewer register by writing inline assembly instead of doing the OR operation.

Number goes up



commit 6114fd209a1d3605b15f8c5351db11c8f92d5a08

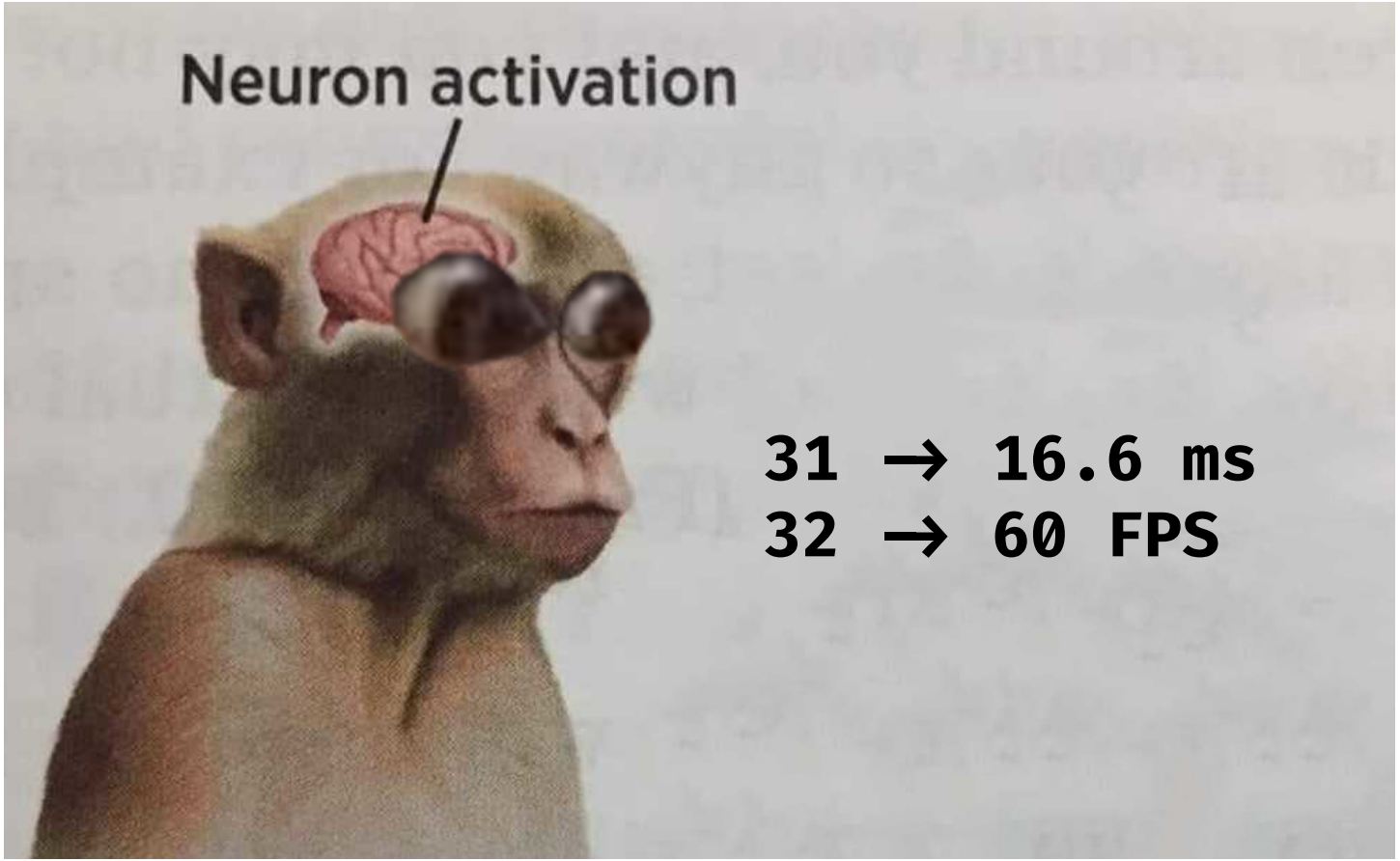
Host caching

- Currently, GPU writes to the CPU particle buffers skip the CPU cache.
- We can enable caching.

```
VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT
-> VMA_ALLOCATION_CREATE_HOST_ACCESS_RANDOM_BIT
```

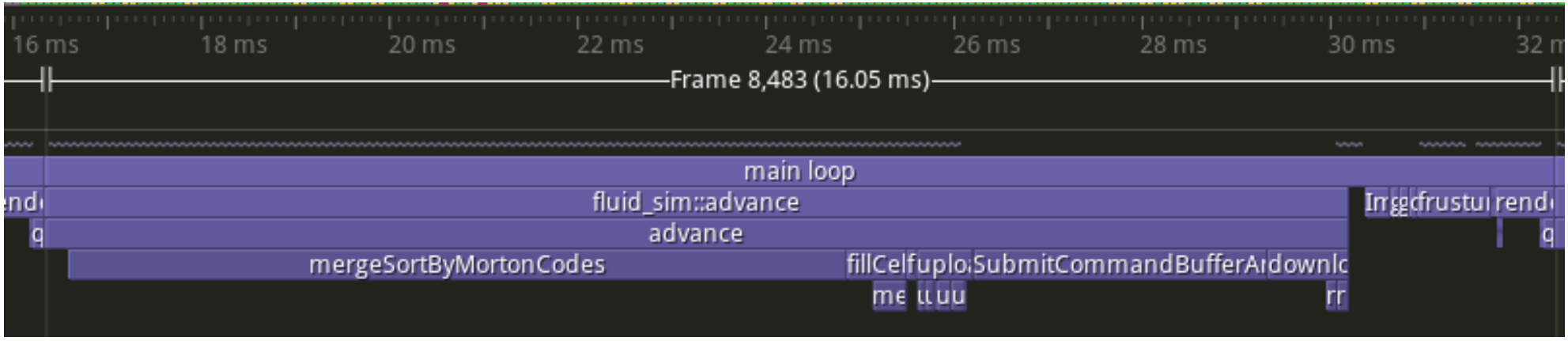
* I also changed the particle buffers to be device-local, + staging buffers, but iirc that didn’t account for most of the speedup.

Number goes up



commit 1aff7d388758ce0a7f5daf841db5a0132d4a7c42

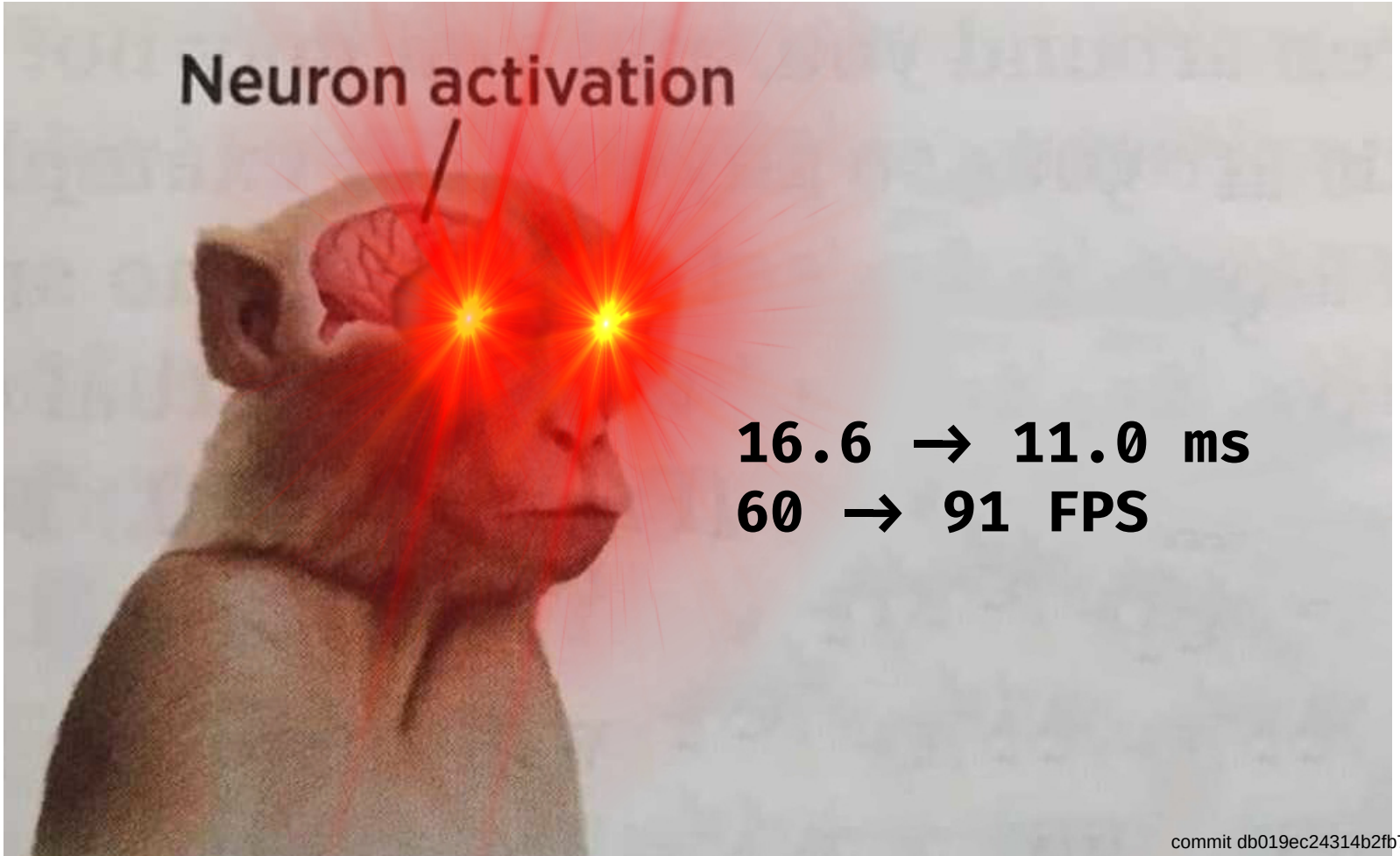
Sorting is slow



Moving too much data

- We sort the particles by their Morton codes
- Currently sorting the particle data directly
  - Each particle has a float4 position, float4 velocity
  - That is a lot of data movement
- Instead, sort only the Morton codes and track the permutation.
  - Then use the generated permutation to sort the particles data in one pass.

Number goes up



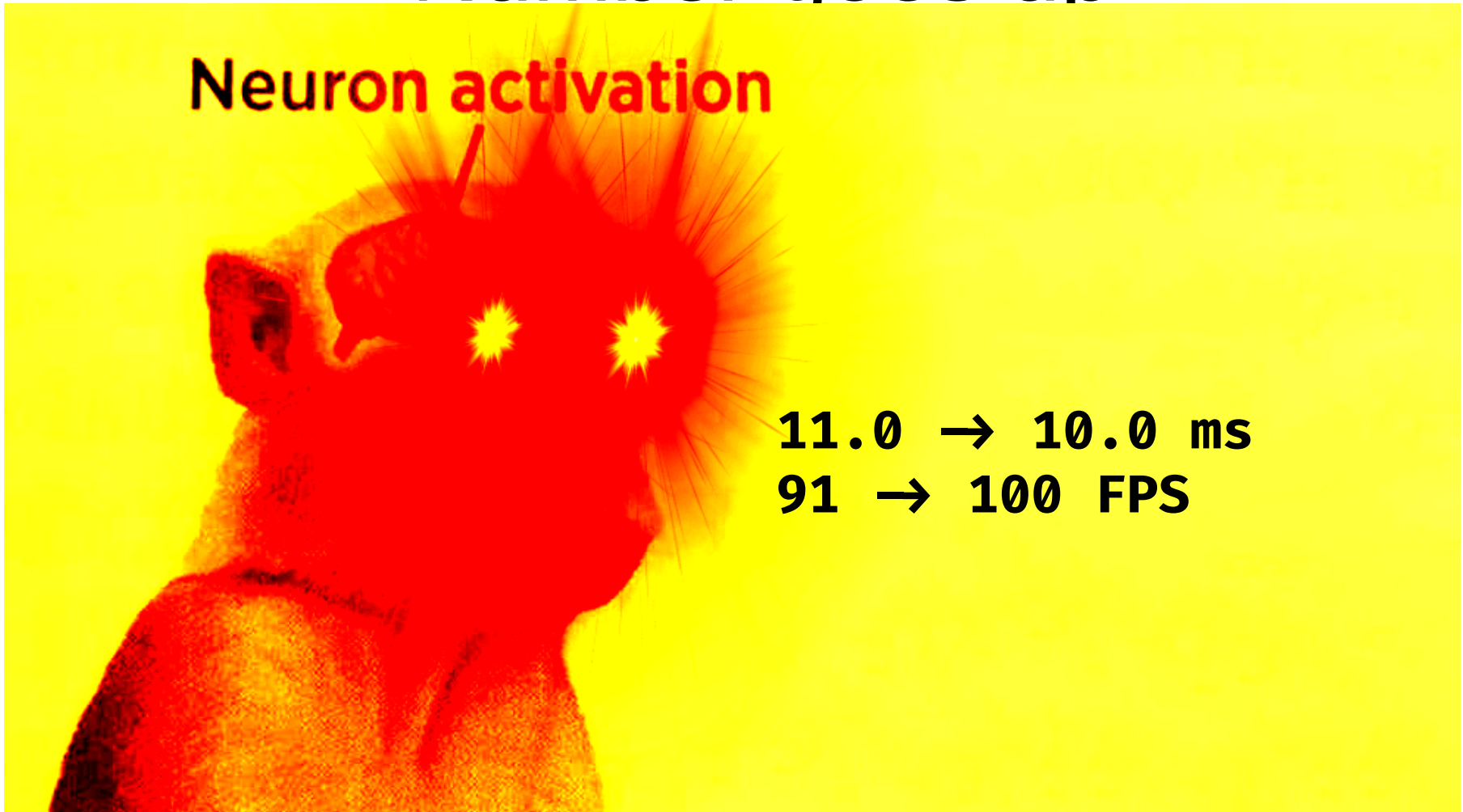
commit db019ec24314b2fb7493f94ae12c4e38532ded4a

Misc sorting optimizations

- Multithreaded particle-sort (with a thread pool):
  - 11.0 -> 9.9 ms commit ebe1f81ae6d5e10b6939ac648c8c28b1bcdcbfb3
- Use AoS instead of SoA when sorting
  - i.e. intersperse Morton codes and permutation indices
  - 9.9 -> 9.4 ms commit dfabe82b7ec6ac5e87ad9fdbec22bd82f846e52b
- Misc cell-sorting optimizations
  - 9.4 -> 10.0 ms
  - but 26.0 -> 12.3 ms when particles are very spread out

commit c1b06650f452a36427647cbe914039914e2faceb

Number goes up

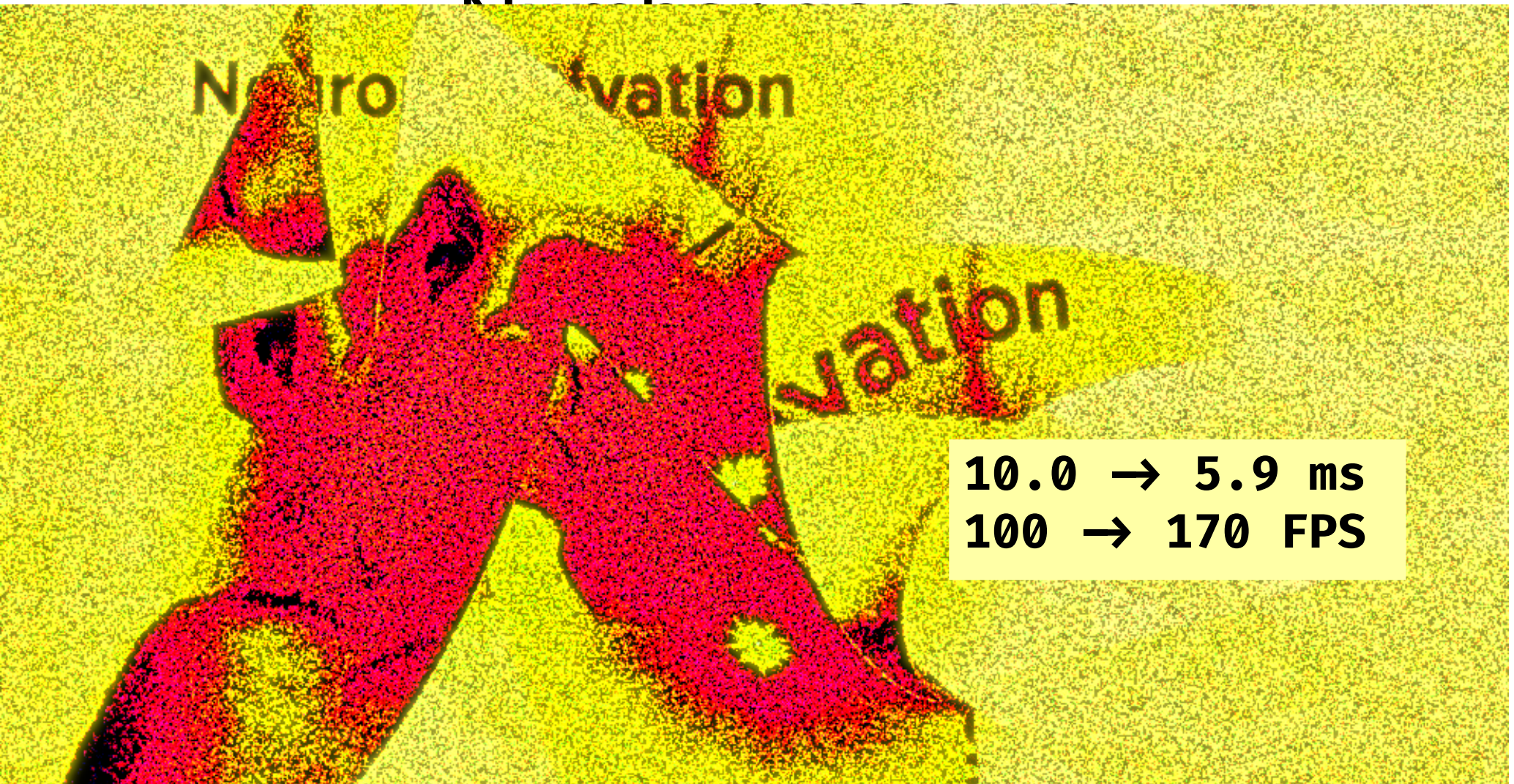


Reduced data movement

- Currently downloading and uploading a lot of particle data
  - positions: float4 [particle_count]
  - velocities: float4 [particle_count]
- CPU only needs the particles' Morton codes
  - Morton codes: u32 [particle_count]
  - Compute Morton codes on GPU and download them
  - 8x less data

Not so easy

- GPU:
  - Update particles
  - Compute domain min (required for Morton codes)
  - Compute Morton codes
- Min computation on GPU is not trivial
  - Implemented GPU reduction
    - Based on https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
    - Lots of optimization left to do


10.0 → 5.9 ms
100 → 170 FPS

Recap

- Feb 24:
  - 1k particles: ~5 ms (200 FPS)
  - 10k particles: ~190 ms (5 FPS)
  - 100k particles: broken
- Apr 12:
  - 100k particles: ~6 ms (165 FPS)
  - 1M particles: ~65 ms (15 FPS)

Caveats

- Computation time depends on simulation state
  - Much slower when particles are densely packed
  - Can partly be improved by tuning sim parameters
- I only reported overall frame times
  - For better understanding, we must look at individual steps
  - Sometimes speedup in one step causes slowdown in another

Acknowledgements

- Clarkson Open Source Institute
  - Providing a place to discuss random CS topics
  - Especially Thomas Johnson for mentorship
- Handmade Network
  - Encouraging deeply technical endeavors
- Graphics Programming (Discord server)
  - Helping improve GPU data download speeds

More details

- **Wed, Apr 24, ~7:45 -- 9pm, COSI (SC 336)**
- So much more to talk about
- Will present a much longer version, with:
  - Automatic code hot-reloading
  - Profilers and debugging tools (including for GPU)
  - How and why these optimizations work
  - Implementation details
  - Unsolved mysteries