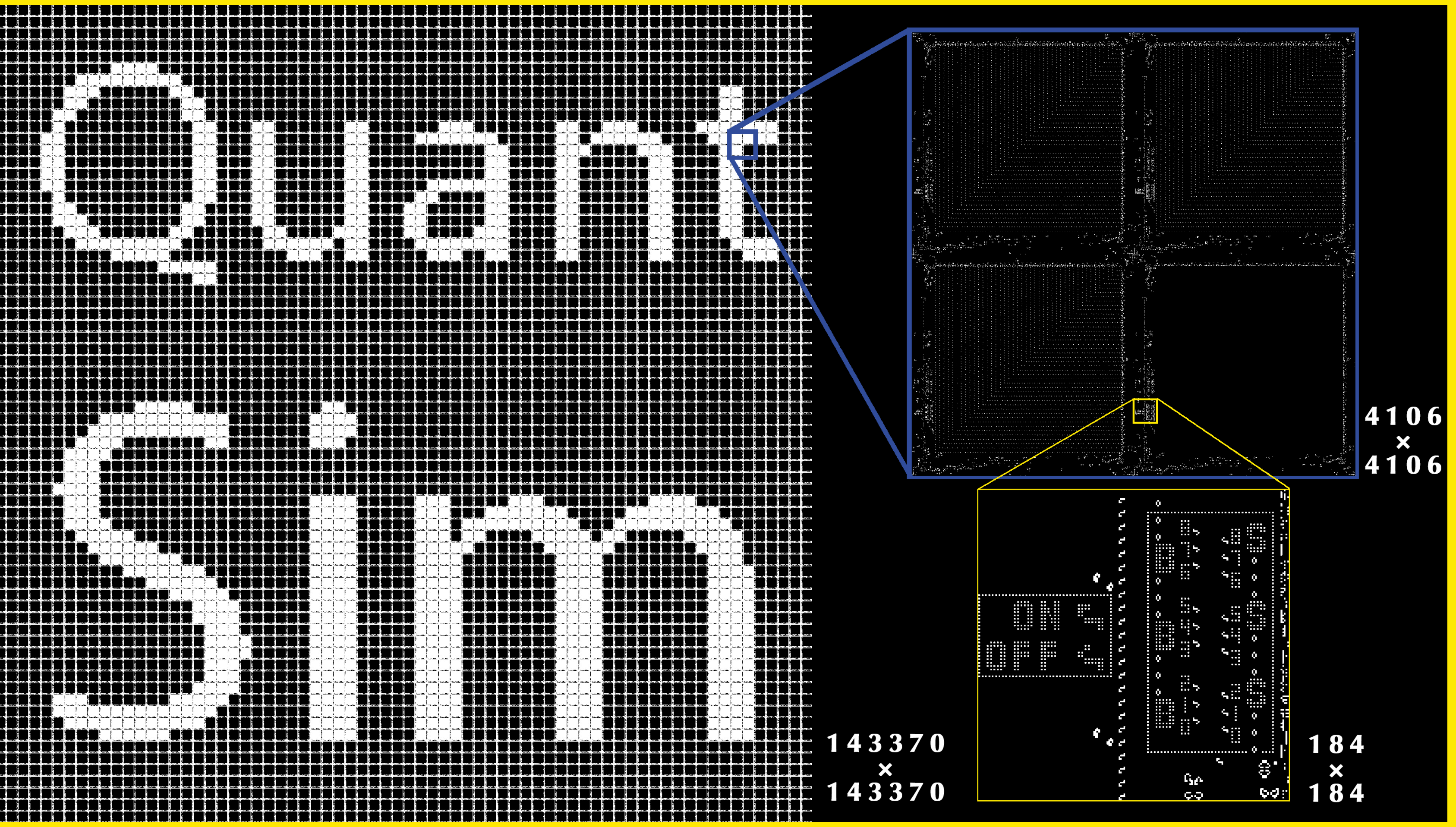
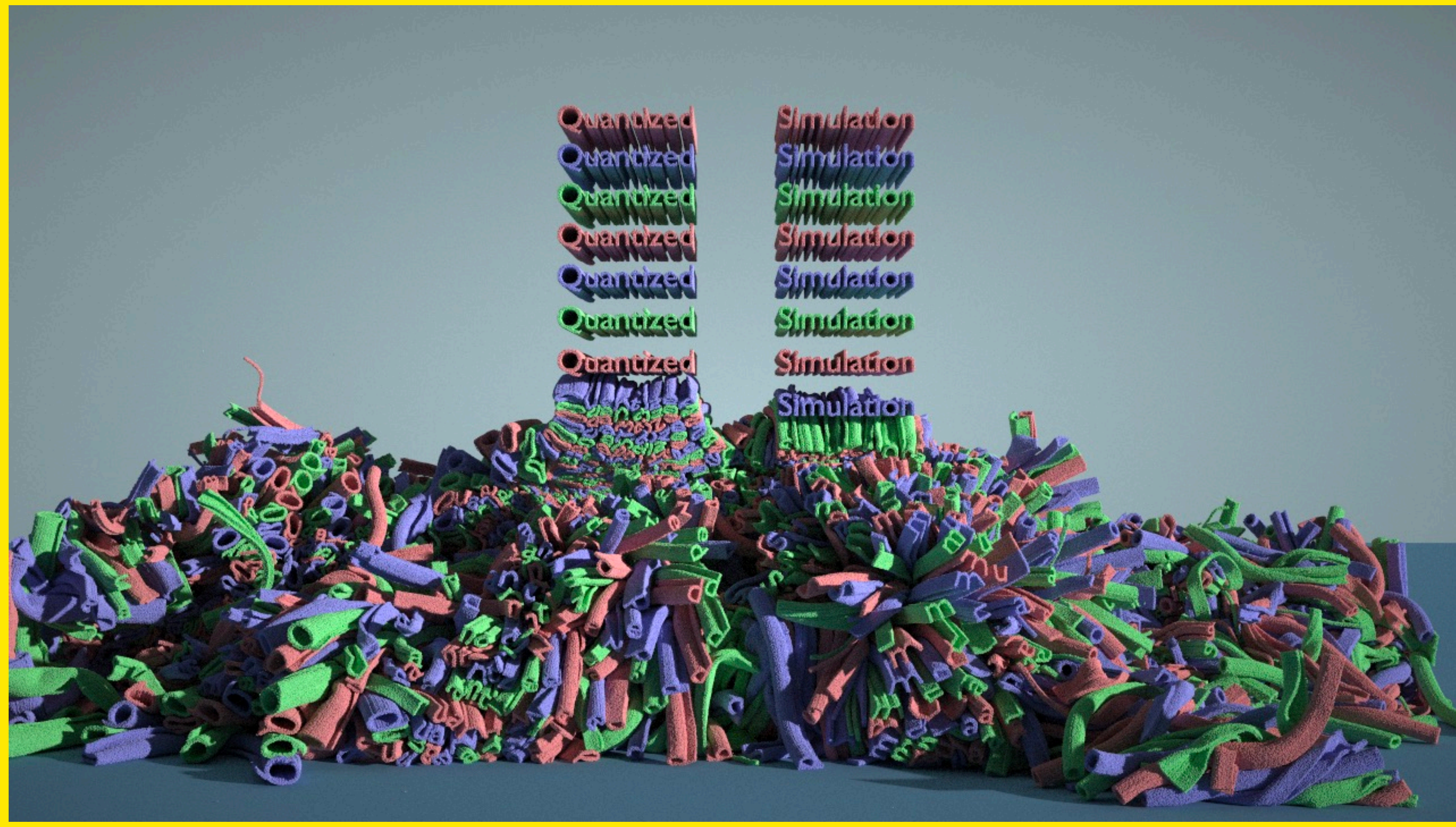
QuanTaichi: A Compiler for Quantized Simulations

"Simulate more with less memory."





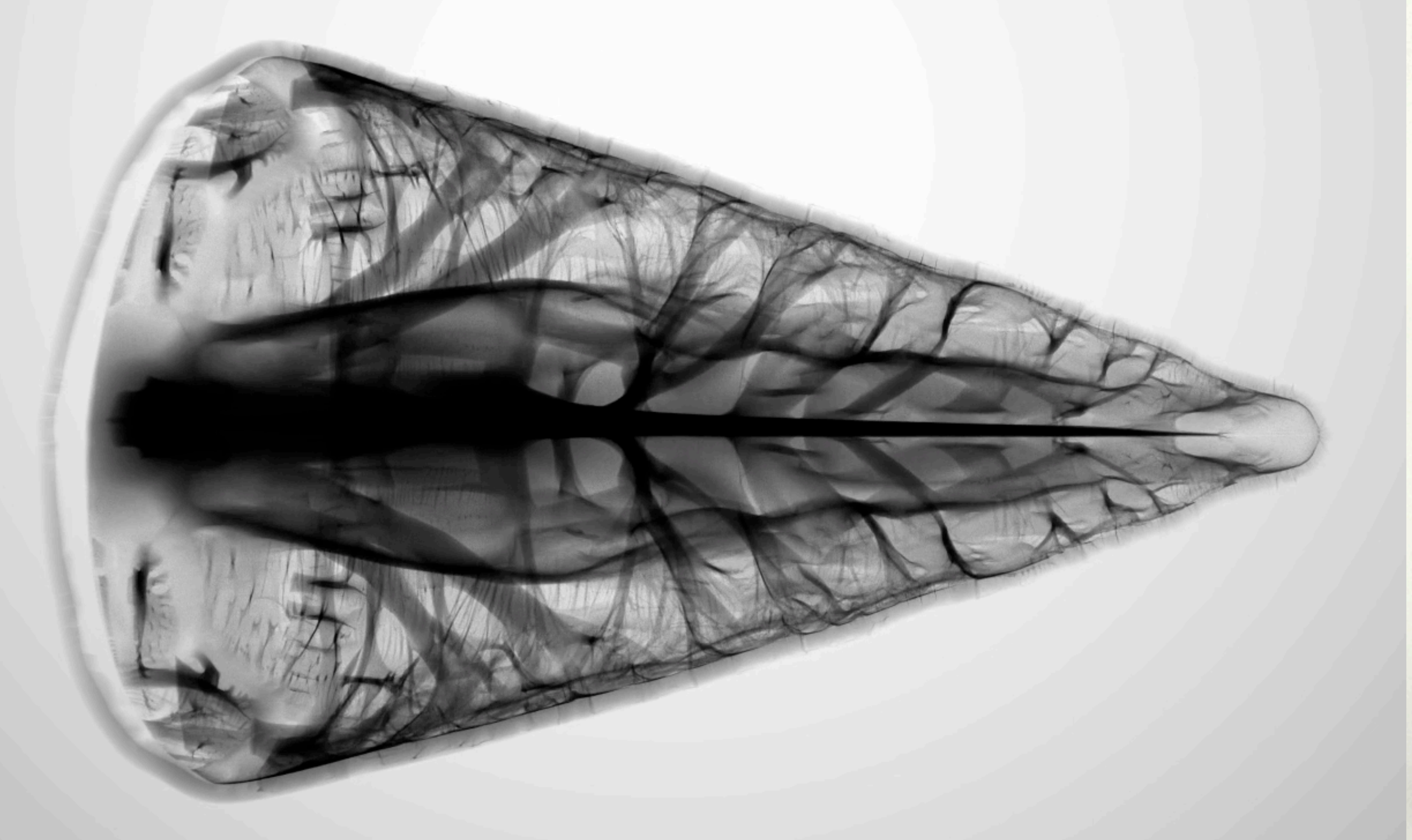
SIGGRAPH 2021

Yuanming Hu[1,2] Jiafeng Liu[3] Xuanda Yang[5] Mingkuan Xu[1,4] Ye Kuang[1]

Weiwei Xu[3] Qiang Dai[6] William T. Freeman[2] Frédo Durand[2]

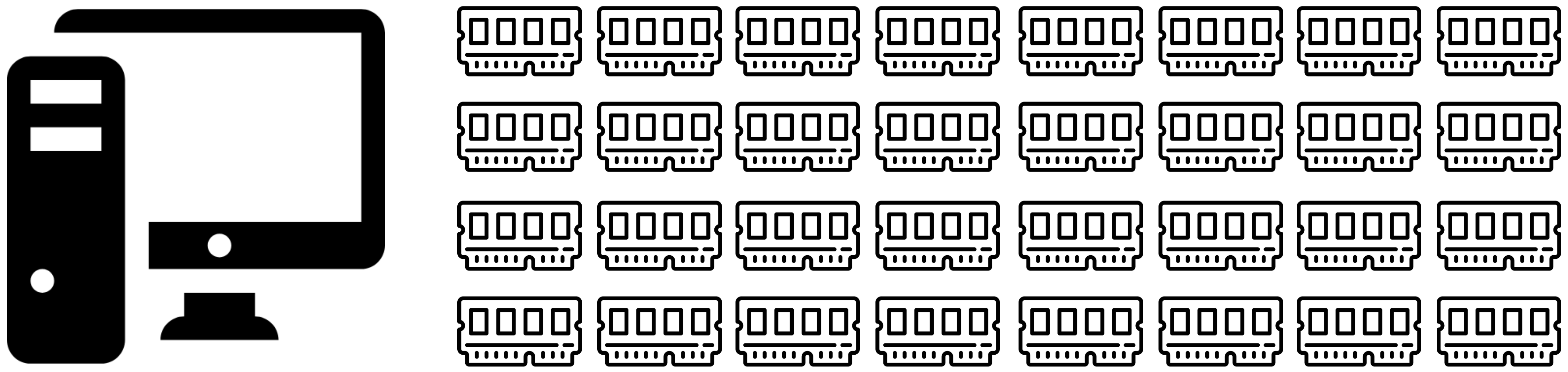
[1]Taichi Graphics [2]MIT CSAIL [3]State Key Laboratory of CAD&CG, Zhejiang University

[4]Tsinghua University [5]Zhejiang University [6]Kuaishou Technology

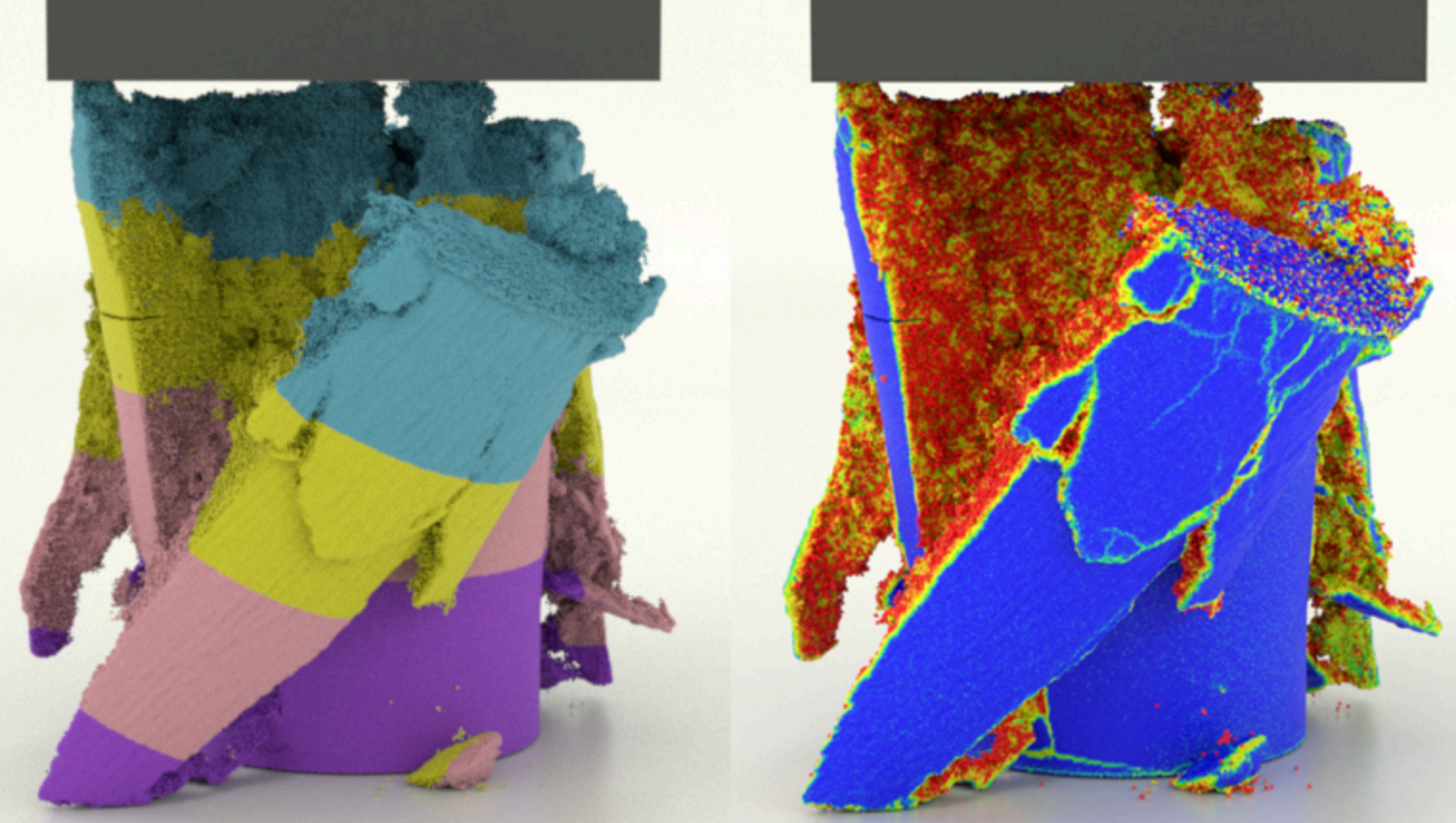


Liu et al. SIGGRAPH Asia 2018: *Narrow-Band Topology Optimization on a Sparsely Populated Grid*

1 billion voxels



Total **512 GB** CPU memory



Wang et al. SIGGRAPH 2020: *A Massively Parallel and Scalable Multi-GPU Material Point Method*

93.8 million particles

4x NVIDIA Quadro P6000

Total **96GB** GPU memory

Saving memory on simulation states

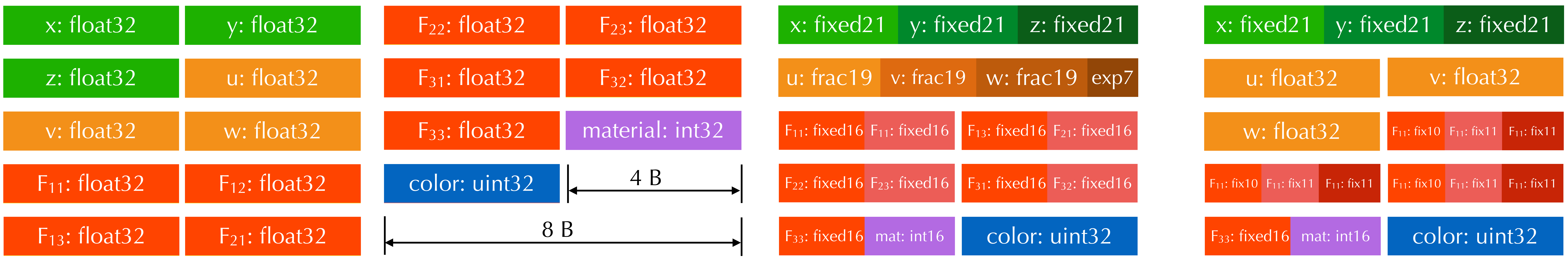
On a single particle:

- Position (x, y, z)
- Velocity (u, v, w)
- Deformation gradient F_{3x3}
- ....

Full-precision: **68** B

Quantization 1: **40** B

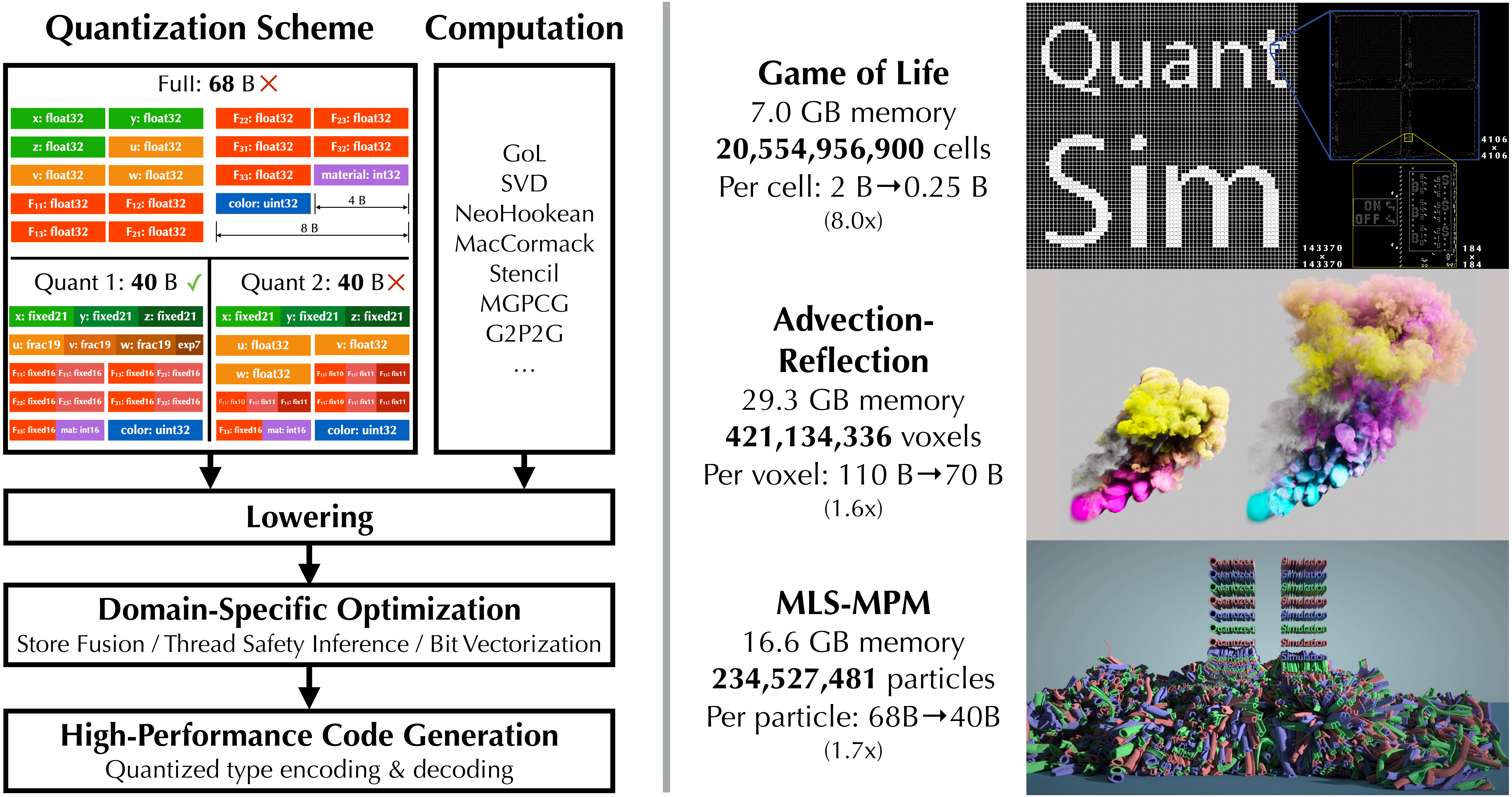
Quantization scheme 2: **40** B



Programming Quantized Simulation

	Manual Engineering & Low-level optimization
	`f = int(x & 32767)) * (1 / 32767.0f)`
		Quantization library
		`template <int exp, int frac>`
		`class Float {…}`
			Language & Compiler (ours)
			`ti.quant.float(exp=4, frac=4)`
			changing only **3%** LoC

Performance	✓	✗	✓
Productivity	✗	✓	✓✓



Quantization Scheme
Computation
Full: 68 B
x: float32
y: float32
z: float32
u: float32
v: float32
w: float32
F11: float32
F12: float32
F13: float32
F21: float32
F22: float32
F23: float32
F31: float32
F32: float32
F33: float32
material: int32
color: uint32
4 B
8 B
Quant 1: 40 B
x: fixed21
y: fixed21
z: fixed21
u: frac19
v: frac19
w: frac19
exp7
F11: fixed16
F11: fixed16
F13: fixed16
F21: fixed16
F22: fixed16
F23: fixed16
F31: fixed16
F32: fixed16
F33: fixed16
mat: int16
color: uint32
Quant 2: 40 B
x: fixed21
y: fixed21
z: fixed21
u: float32
v: float32
w: float32
F33: fixed16
mat: int16
color: uint32
GoL
SVD
NeoHookean
MacCormack
Stencil
MGPCG
G2P2G
...
Lowering
Domain-Specific Optimization
Store Fusion / Thread Safety Inference / Bit Vectorization
High-Performance Code Generation
Quantized type encoding & decoding
Game of Life
7.0 GB memory
20,554,956,900 cells
Per cell: 2 B→0.25 B
(8.0x)
Quant
Sim
4106 × 4106
143370 × 143370
184 × 184
ON
OFF
Advection-Reflection
29.3 GB memory
421,134,336 voxels
Per voxel: 110 B→70 B
(1.6x)
MLS-MPM
16.6 GB memory
234,527,481 particles
Per particle: 68B→40B
(1.7x)
Quantized
Simulation

Taichi: DSL embedded in Python
Data-oriented, imperative, sparse, and parallel

```python
@ti.kernel
def saxpy(a: ti.f32):
    for i in x:
        # Parallel for loop over
        # active indices of x
        z[i] = a * x[i] + y[i]

@ti.kernel
def conditional_stencil():
    for i, j in y: # 2D parallel for loop
        if y[i, j] < 0:
            y[i, j] = x[i-1, j] - 2*x[i, j] + x[i+1, j]
```

Type System

Customized Data Types

Integers

```
i5 = ti.quant.int(bits=5)
u19 = ti.quant.int(bits=19, signed=False)
```

Fixed-point

```
fixed17 = ti.quant.fixed(frac=17, range=3.14)
# Range = [-3.14, 3.14)

ufixed5 = ti.quant.fixed(frac=5, signed=False, range=2)
# Range = [0, 2)
```

Floating-point

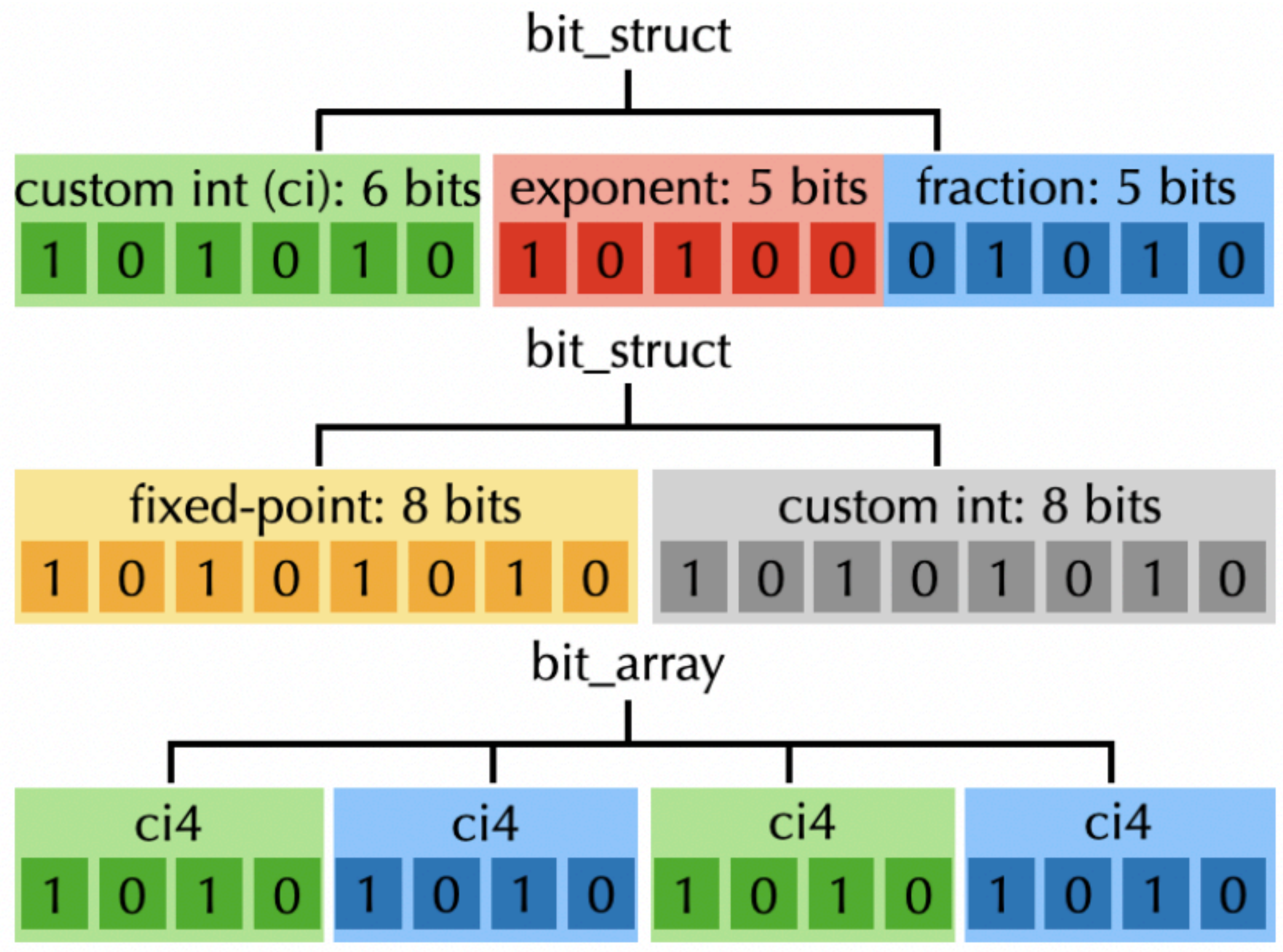
```
f18  = ti.quant.float(exp=4, frac=14)
uf22 = ti.quant.float(exp=6, frac=16, signed=False)
```

"Compute type"

```
i21 = ti.quant.int(bit=21, compute=ti.i64)
bfloat16 = ti.quant.float(exp=8, frac=8, compute=ti.f32)
```

Tree-based Type System

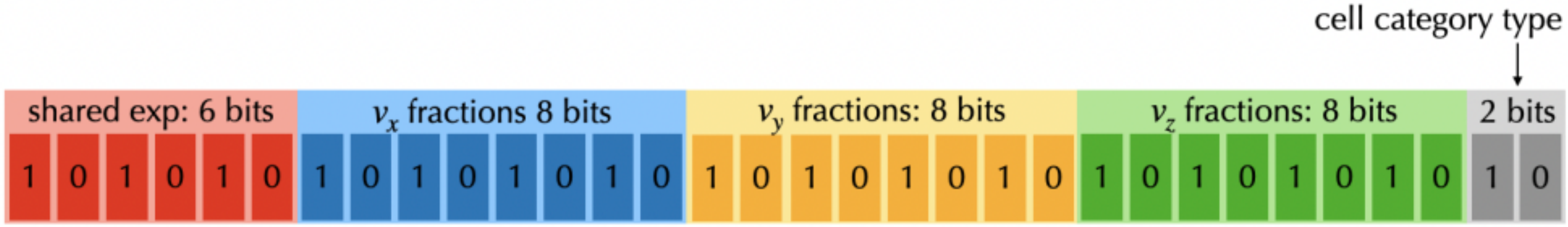
- Two new families of types
  - Bit struct
  - Bit array
- Extends Taichi's hierarchical SNode system
- Decompose hardware-native data types



Bit struct

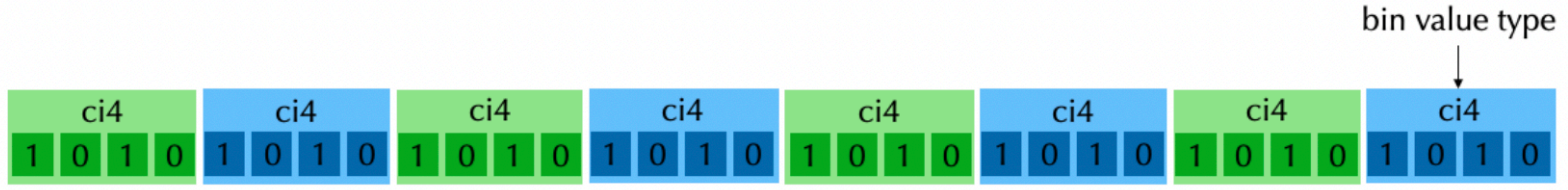
```
velocity_component_type =
   ti.quant.float(exp=6, frac=8, compute=ti.f32)
velocity = ti.Vector(3, dtype=velocity_component_type)

# Since there are only three cell categories,
# 2 bits are enough
cell_category_type =
   ti.quant.int(bits=2, signed=False, compute=ti.i32)
cell_category = ti.field(dtype=cell_category_type)

# The bit struct for 512x512x256 voxels
voxel = ti.root.dense(ti.ijk, (512, 512, 256))
           .bit_struct(num_bits=32)

# Place three components of velocity into the voxel,
# and let them share the components.
voxel.place(velocity, shared_exponent=True)
# Place the 2-bit cell category
voxel.place(cell_category)
```

Divide hardware-native types (e.g., i32) into smaller pieces

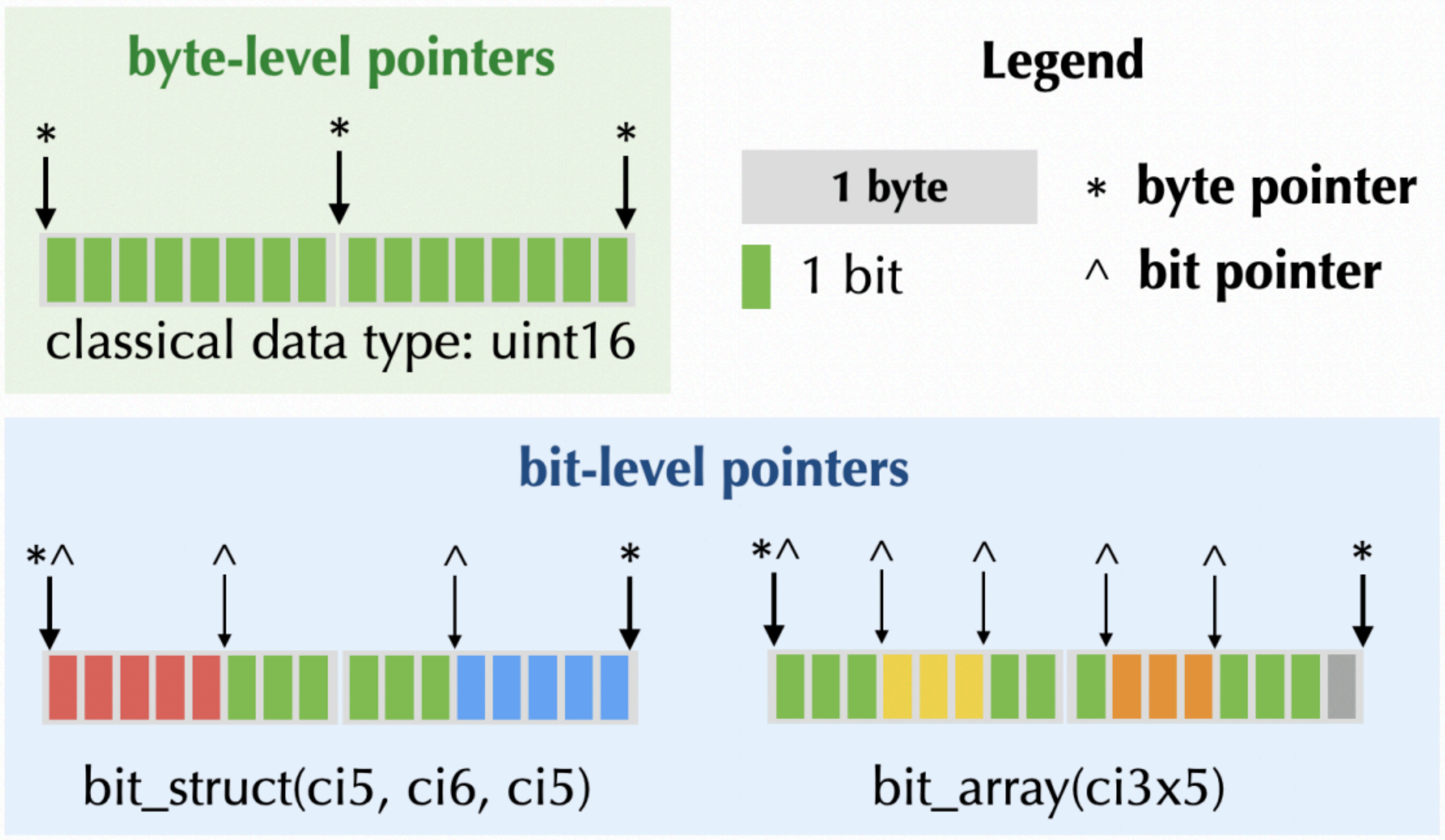


Bit array

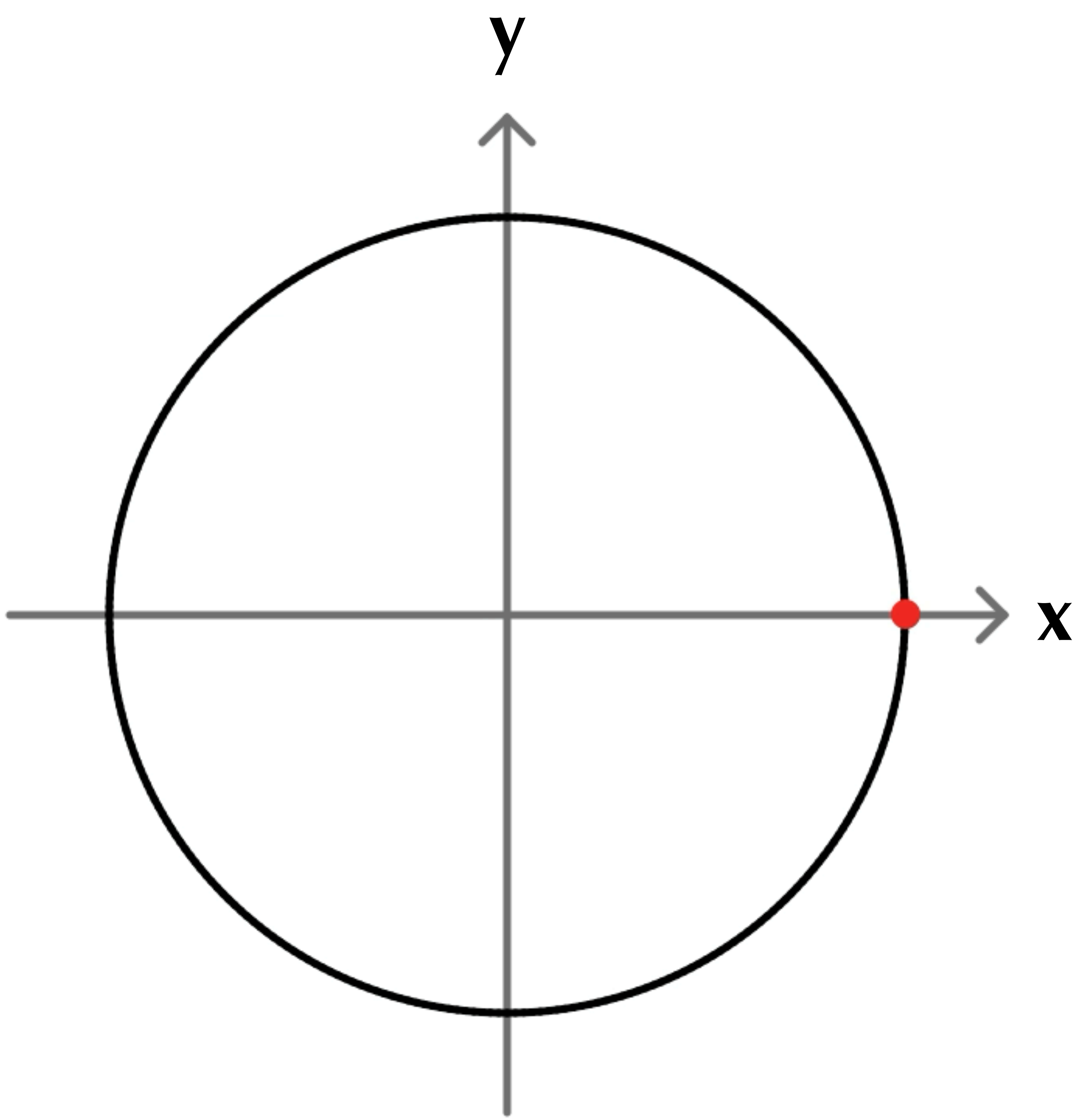
```
bin_value_type = ti.quant.int(num_bits=4, signed=False)

# The bit array for 512x512 bin values
array = ti.root.dense(ti.ij, (512, 64))
            .bit_array(ti.i, 8, num_bits=32)

# Place the unsigned 4-bit bin value into the array
array.place(bin_value_type)
```



Bit pointers: addressing bits



Real Number Types



	x	y
	1.0000	0.0000

x (IEEE 754 "float"): $sign_x$ 0 | exp_x 01111111 | $frac_x$ 00000000000000000000000

y (IEEE 754 "float"): $sign_y$ 0 | exp_y 00000000 | $frac_y$ 00000000000000000000000

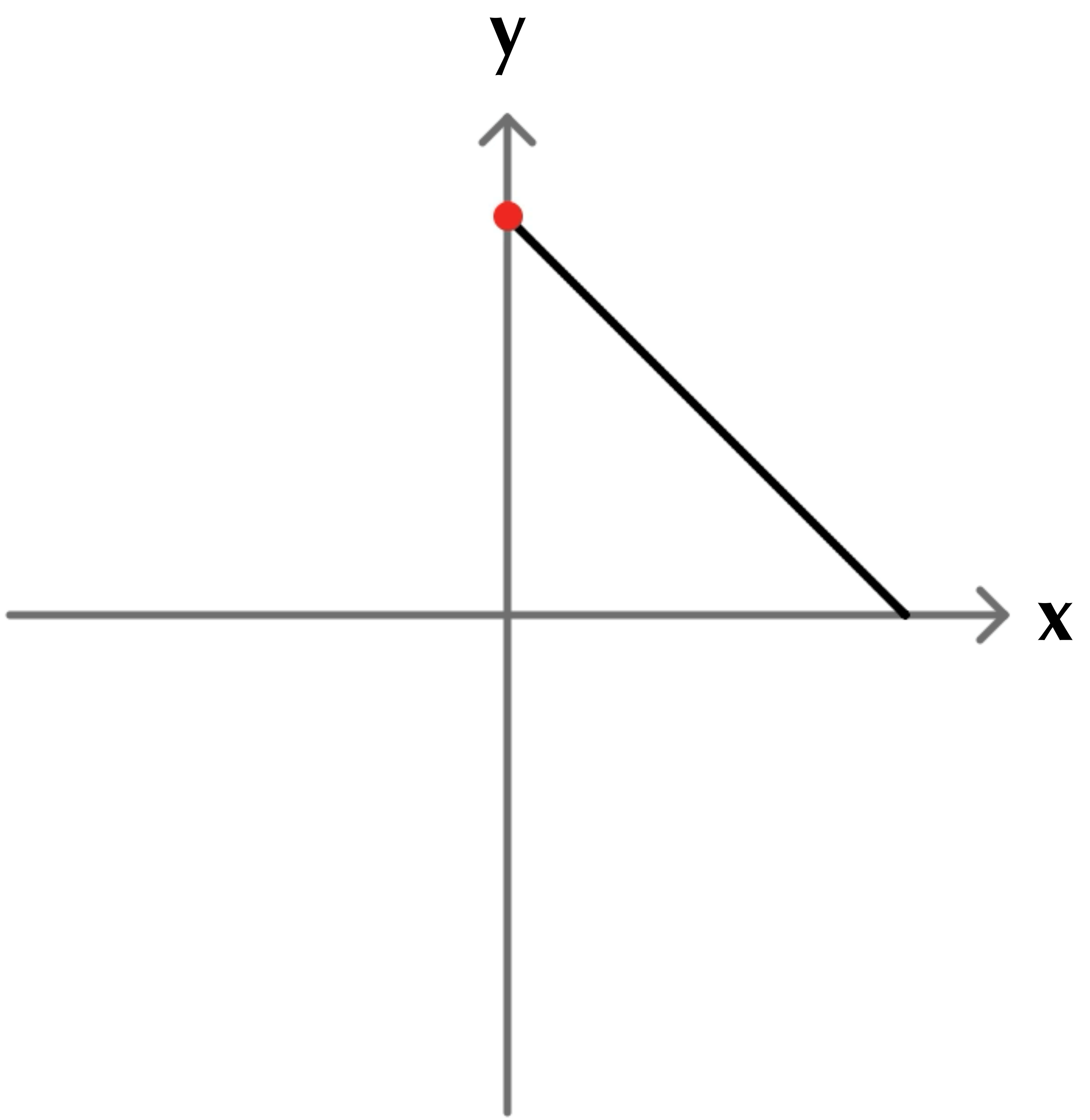
---

x, y: ti.quant.float (**exp**=6, **fraction**=13) shared exponent: exp_{xy} 011111 | $sign_x$ 0 | $frac_x$ 1000000000000 | $sign_y$ 0 | $frac_y$ 0000000000000

x, y: ti.quant.float (**exp**=5, **fraction**=11): exp_x 01111 | $sign_x$ 0 | $frac_x$ 00000000000 | exp_y 00000 | $sign_y$ 0 | $frac_y$ 00000000000

x, y: ti.quant.fixed (**fraction**=16, **range**=2.0): $sign_x$ 0 | $frac_x$ 1000000000000000 | $sign_y$ 0 | $frac_y$ 0000000000000000

Real Number Types



	x	y
	0.0000	1.0000

x (IEEE 754 "float")
$sign_x$ exp_x $frac_x$
0 00000000 00000000000000000000000

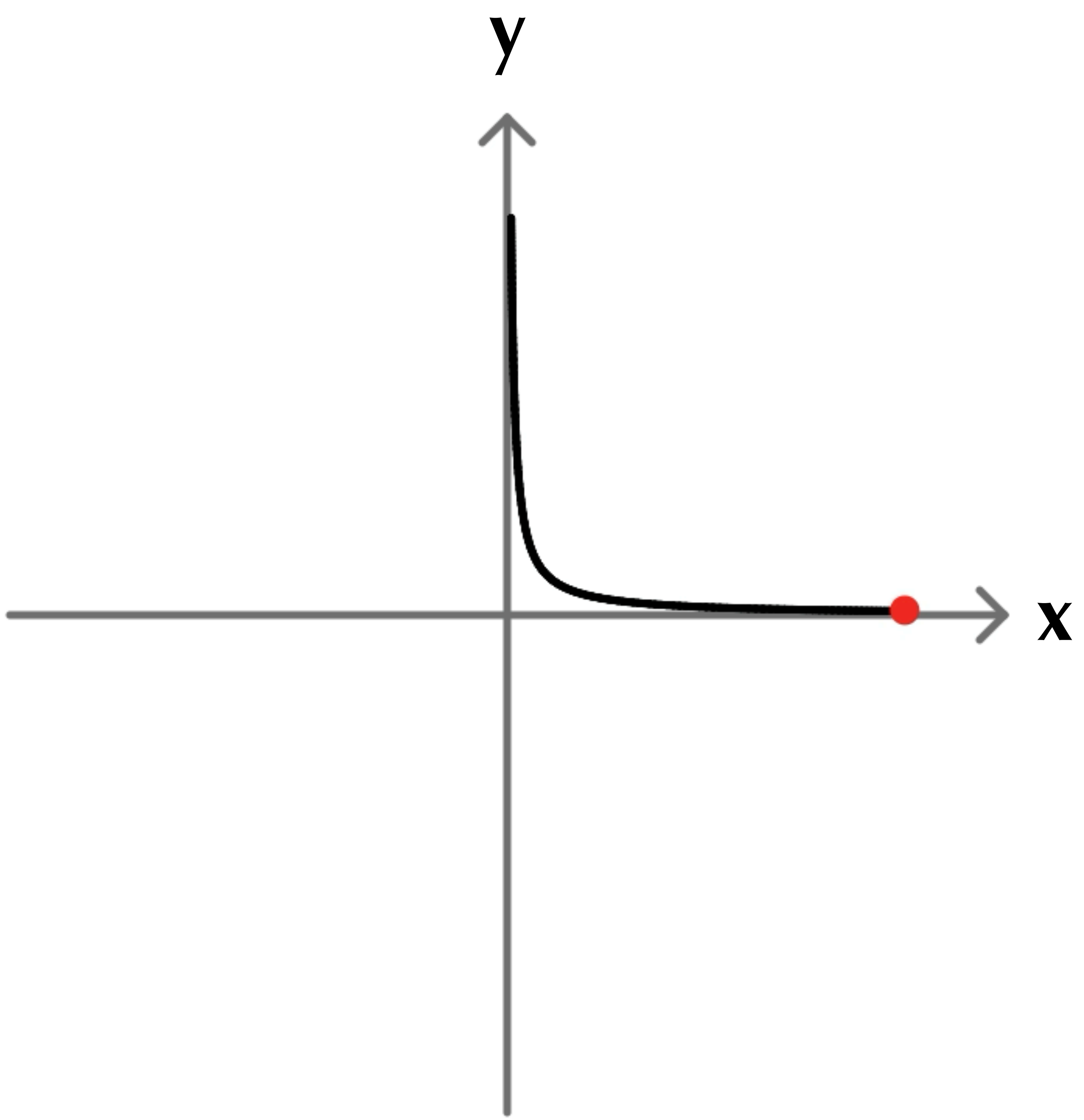
y (IEEE 754 "float")
$sign_y$ exp_y $frac_y$
0 01111111 00000000000000000000000

x, y: ti.quant.float
(**exp**=6, **fraction**=13)
shared exponent
exp_{xy} $sign_x$ $frac_x$ $sign_y$ $frac_y$
011111 0 000000000000 0 100000000000

x, y: ti.quant.float
(**exp**=5, **fraction**=11)
exp_x $sign_x$ $frac_x$ exp_y $sign_y$ $frac_y$
00000 0 0000000000 01111 0 0000000000

x, y: ti.quant.fixed
(**fraction**=16, **range**=2.0)
$sign_x$ $frac_x$ $sign_y$ $frac_y$
0 000000000000000 0 100000000000000

Real Number Types



	x	y
	0.9974	0.0100

x (IEEE 754 “float”)

$sign_x$	exp_x	$frac_x$
0	01111110	11111110101011011001101

y (IEEE 754 “float”)

$sign_y$	exp_y	$frac_y$
0	01111000	01001000100001110011011

---

x, y: ti.quant.float
(**exp**=6, **fraction**=13)
shared exponent

$\mathbf{exp_{xy}}$	$sign_x$	$frac_x$	$sign_y$	$frac_y$
011110	0	1111111110101	0	0000000101001

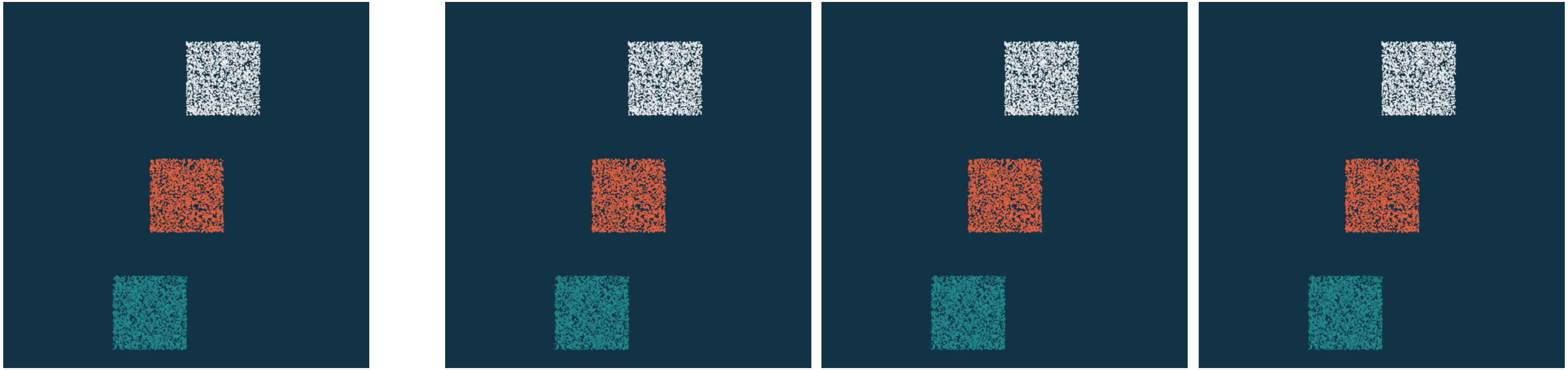
x, y: ti.quant.float
(**exp**=5, **fraction**=11)

exp_x	$sign_x$	$frac_x$	exp_y	$sign_y$	$frac_y$
01110	0	11111111011	01000	0	01001000010

x, y: ti.quant.fixed
(**fraction**=16, **range**=2.0)

$sign_x$	$frac_x$	$sign_y$	$frac_y$
0	0111111111010110	0	0000000010100100

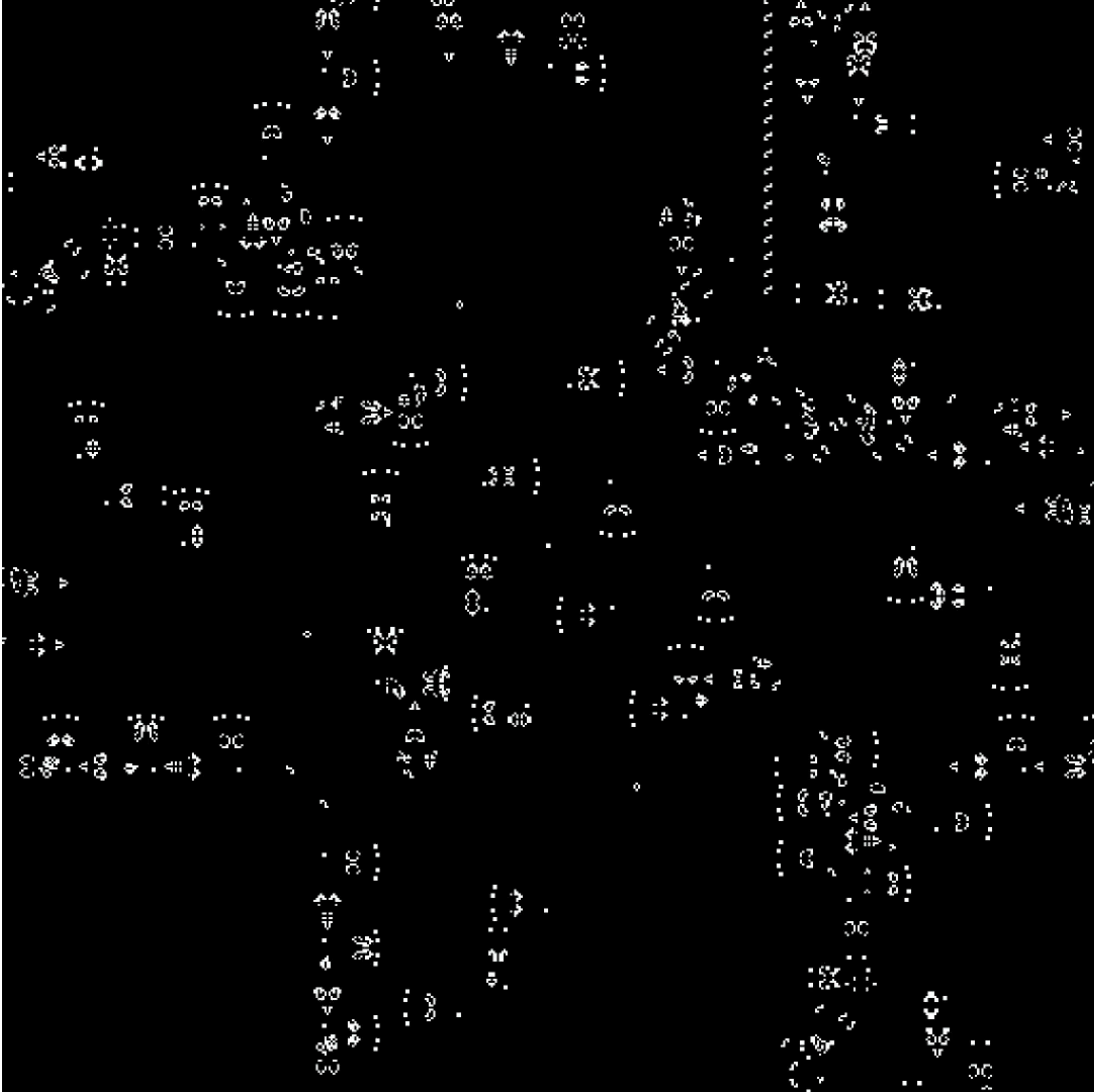
The impact of rounding scheme



float32 reference Round towards zero Rounding up Round to nearest

Large-Scale Demos

All run on a single GPU with at most 32 GB memory



Game of Life

with 2048x2048 OTCA meta cells

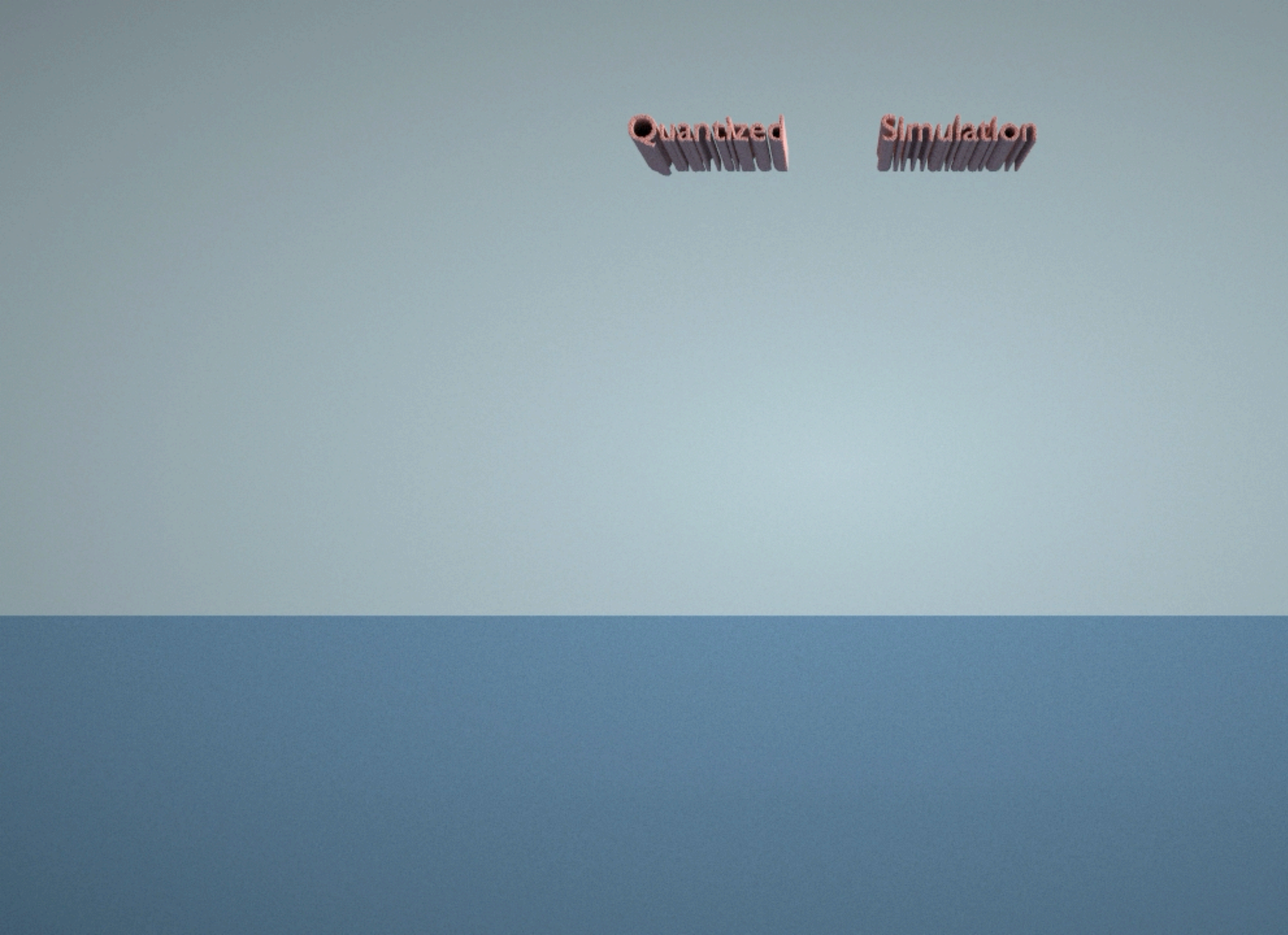
20,554,956,900 cells

Per cell 2 B→0.25 B

8x compressed storage

53.0s / frame

7.0 GB memory allocated



MLS-MPM

234,527,481 particles

Per particle 68B→40B

76.2s / frame

16.6 GB memory allocated

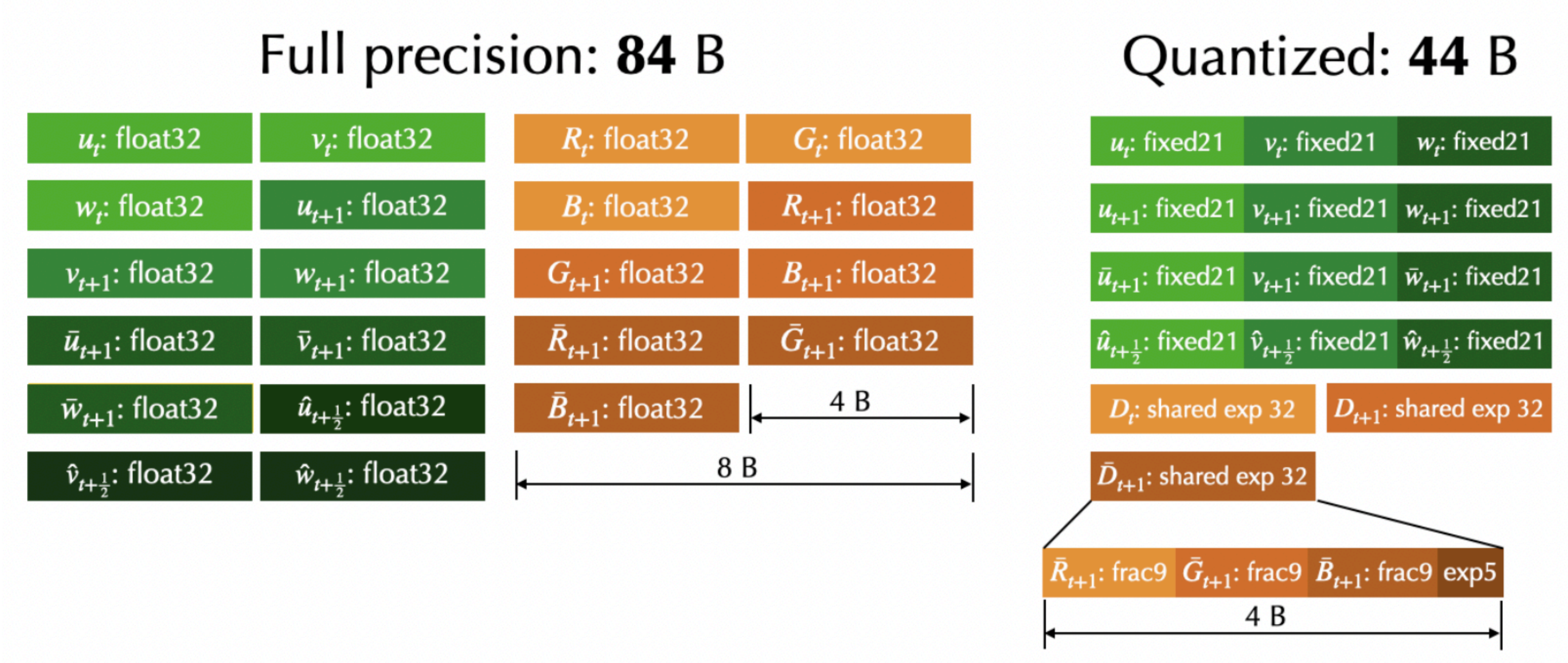
x: fixed21	y: fixed21	z: fixed21	
u: frac19	v: frac19	w: frac19	exp7
F_{11}: fixed16	F_{11}: fixed16	F_{13}: fixed16	F_{21}: fixed16
F_{22}: fixed16	F_{23}: fixed16	F_{31}: fixed16	F_{32}: fixed16
F_{33}: fixed16	mat: int16	color: uint32	

Advection-reflection fluid simulation

421,134,336 voxels

Per voxel 110 B→70 B

68.8s / frame

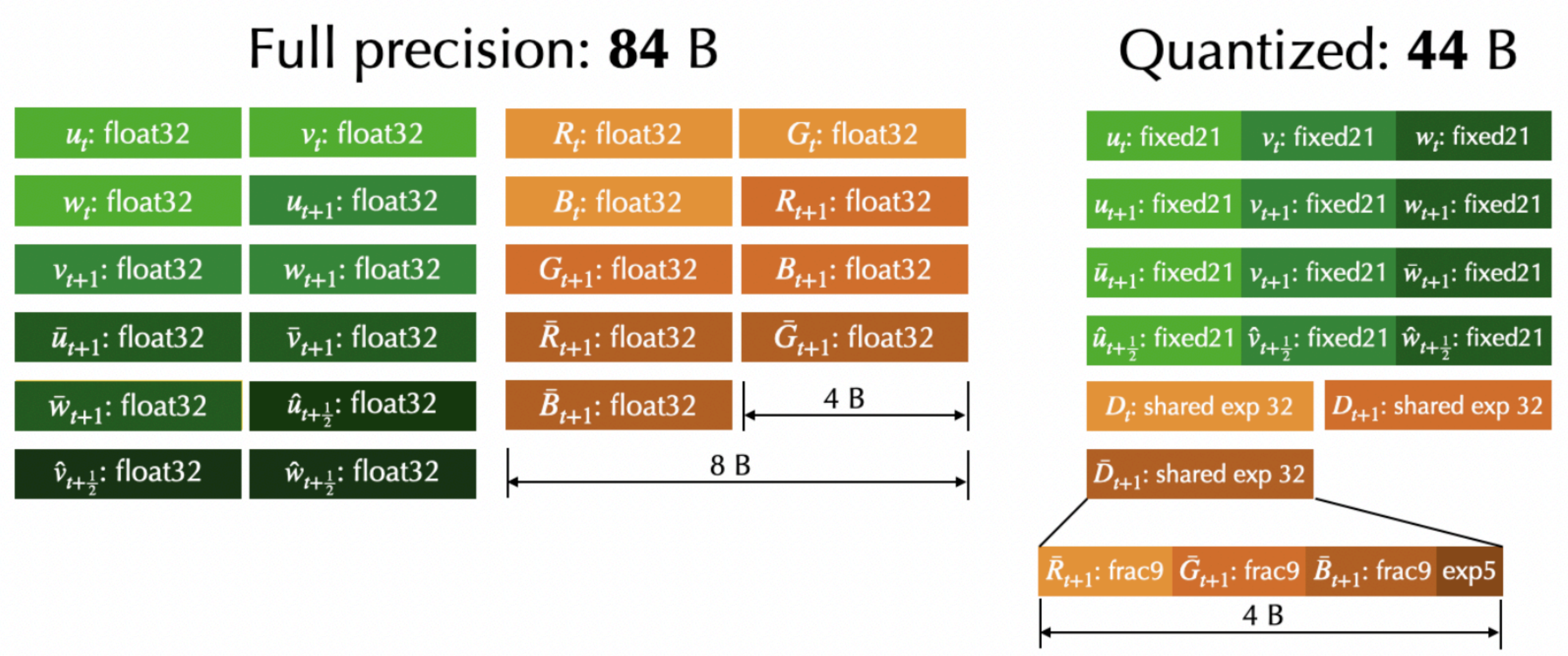


Advection-reflection fluid simulation

421,134,336 voxels

Per voxel 110 B→70 B

58.3s / frame

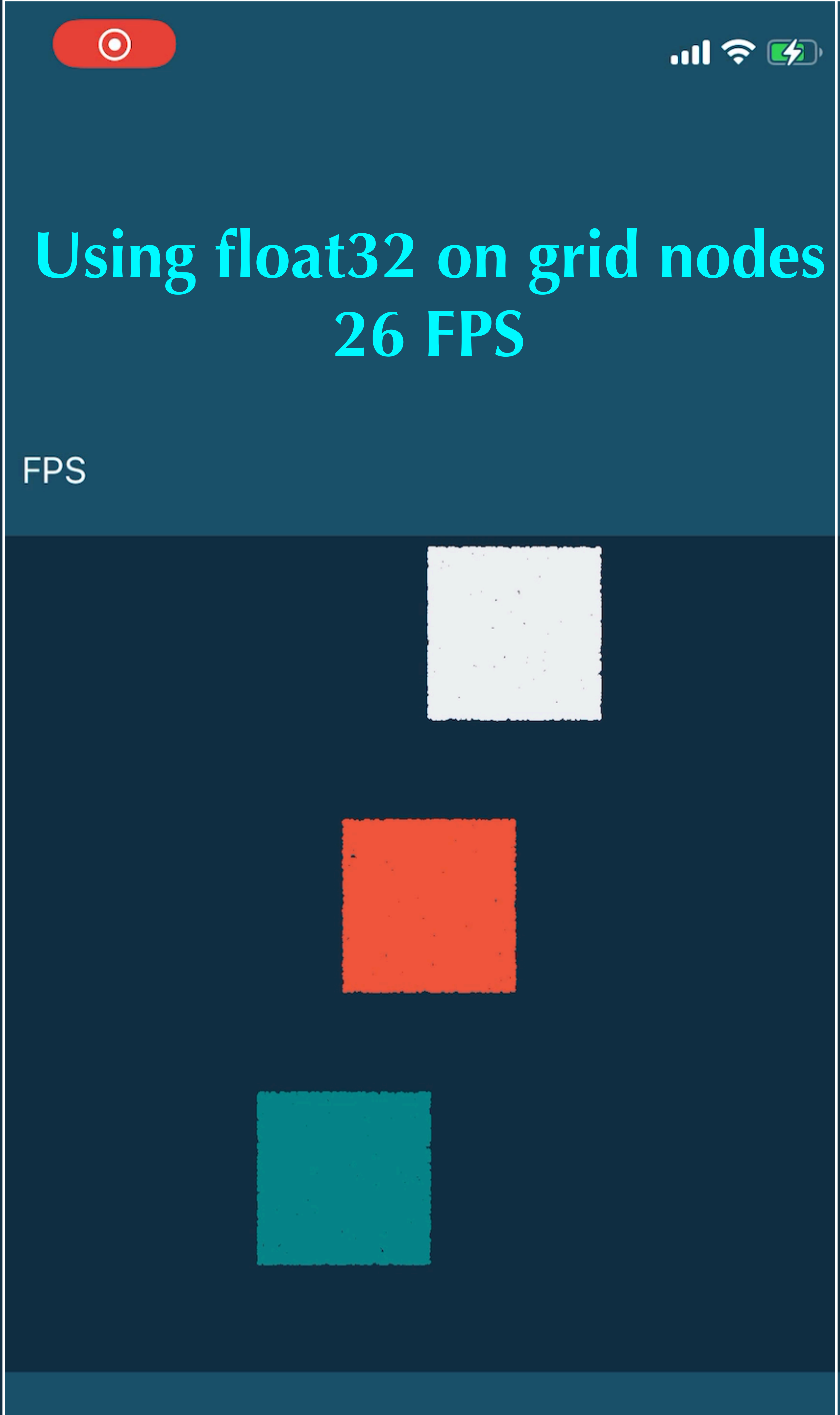


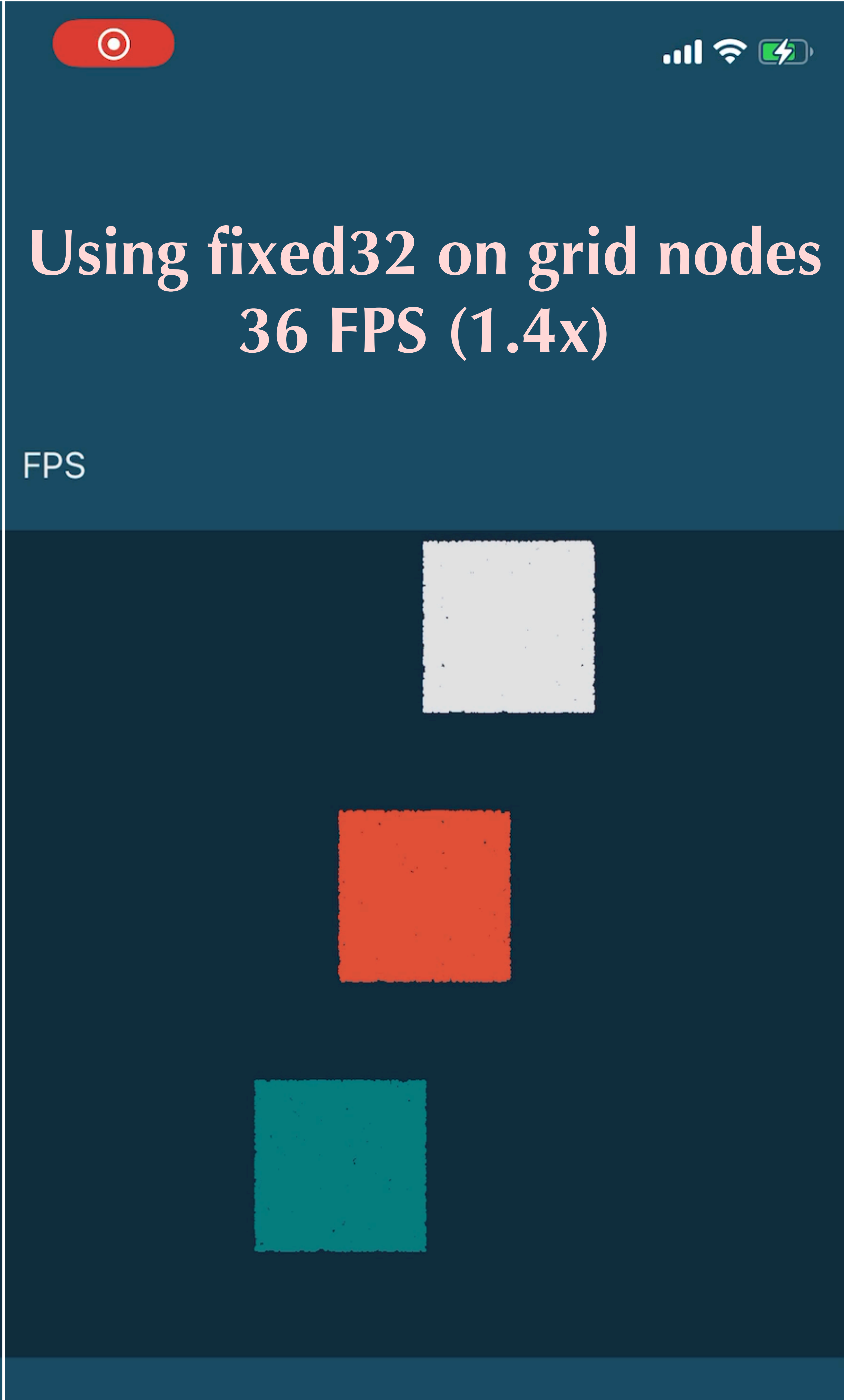
MLS-MPM on iPhone XS

36K particles
256x256 grid

1.4x speed up

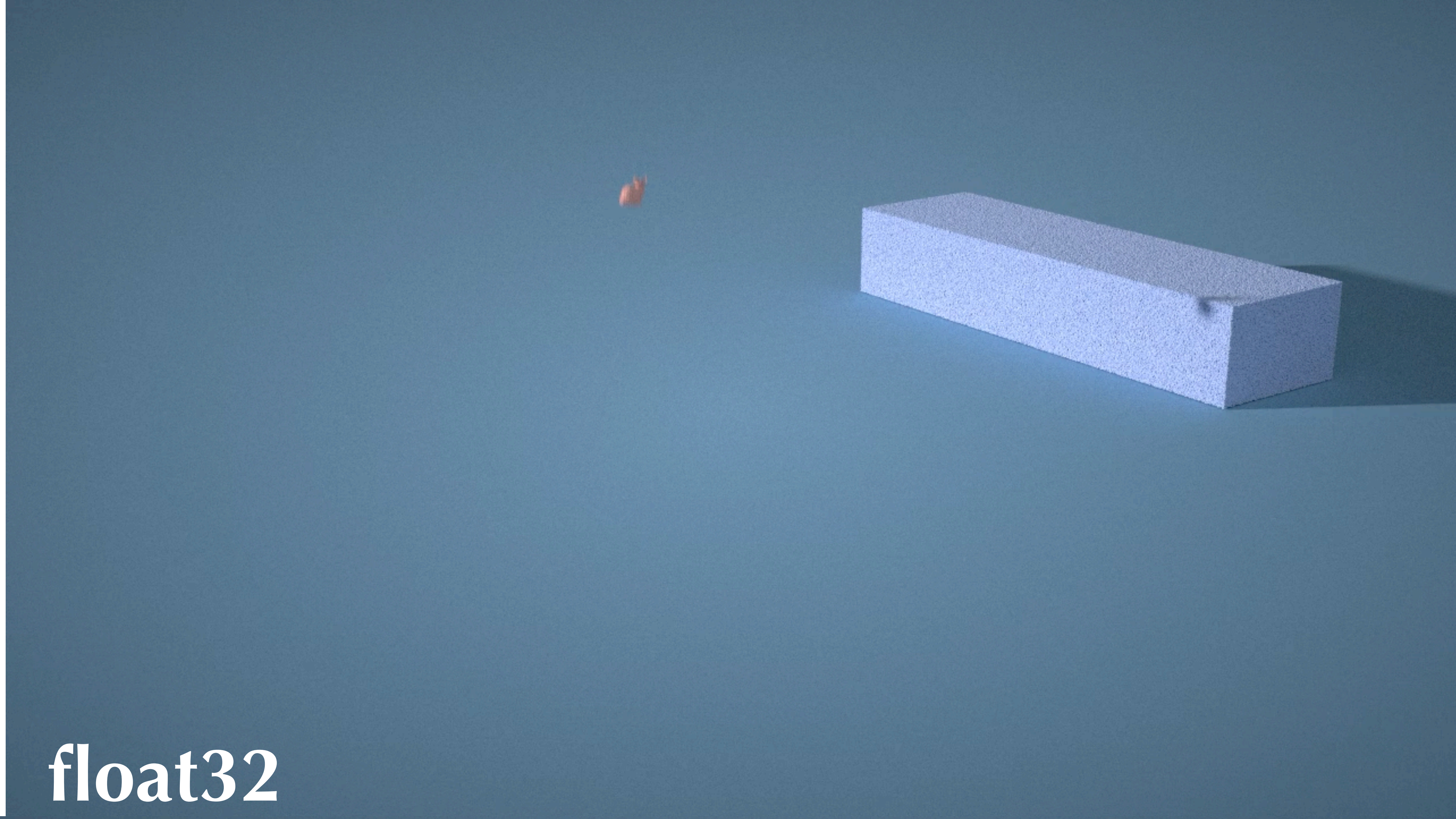
float32 atomics is slower than fixed32 (essentially int32) atomics during P2G.

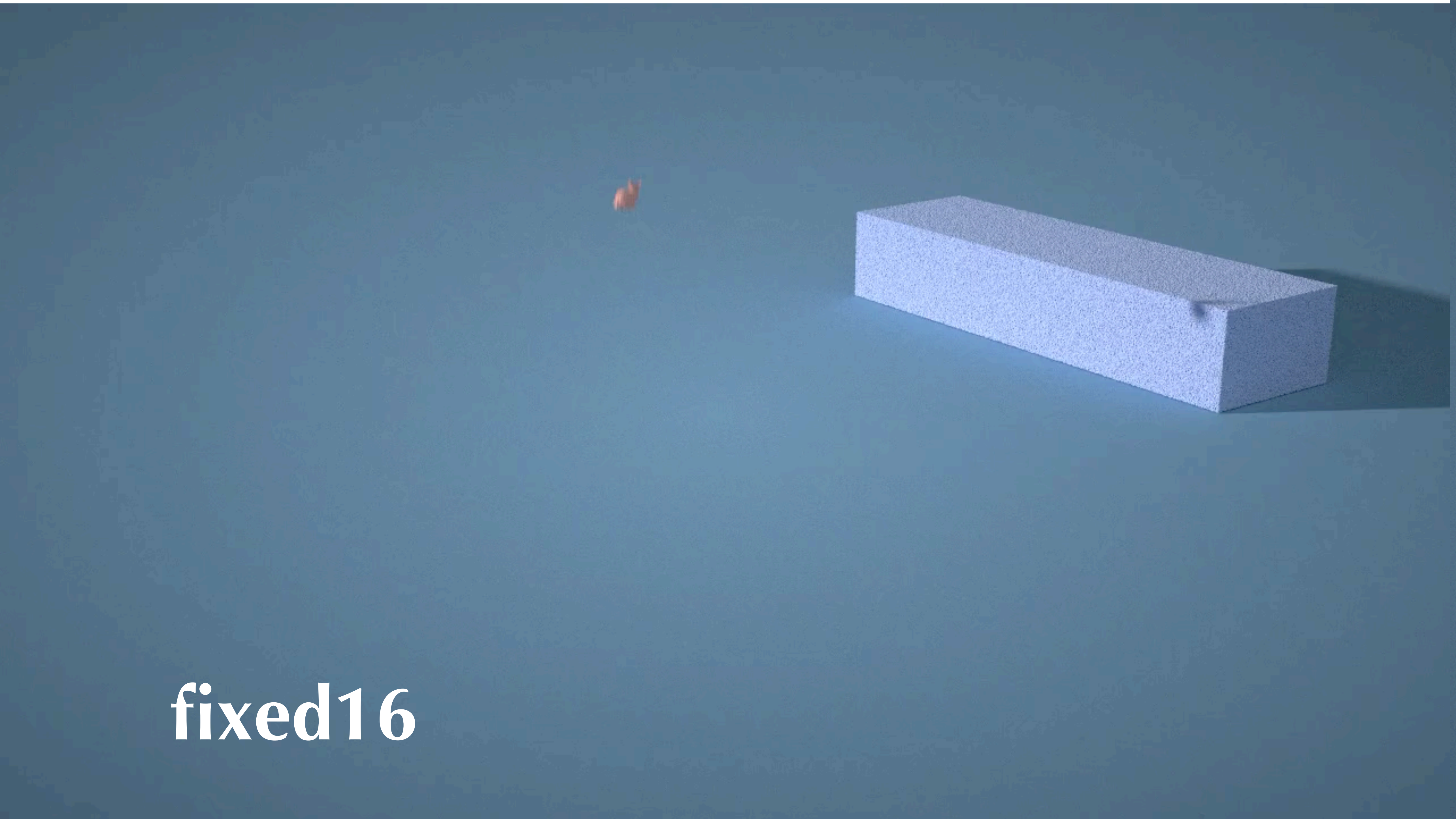


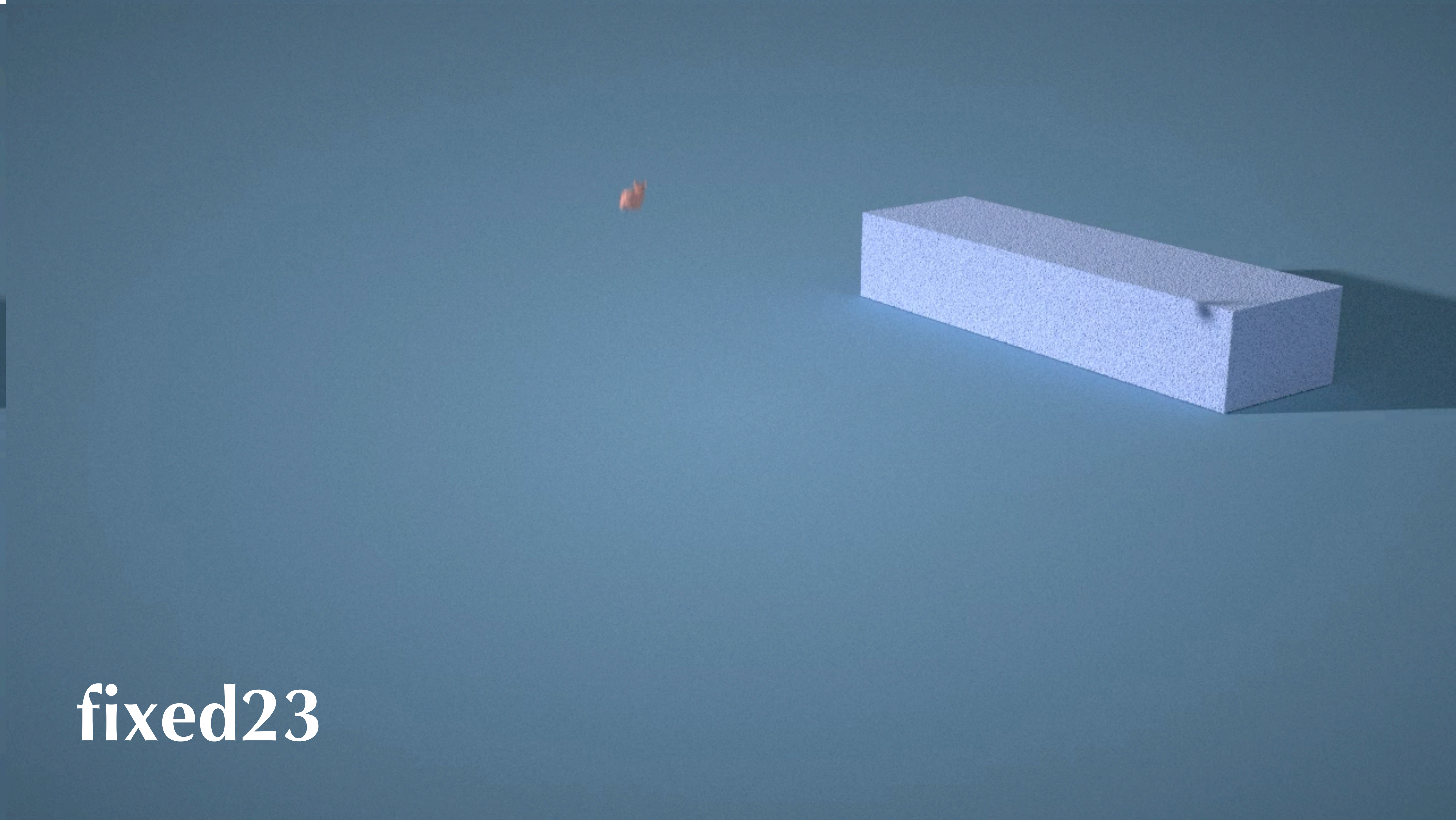


Example failure case

Using fixed16 leads to fluid volume gain
The solution is to simply add more bits.

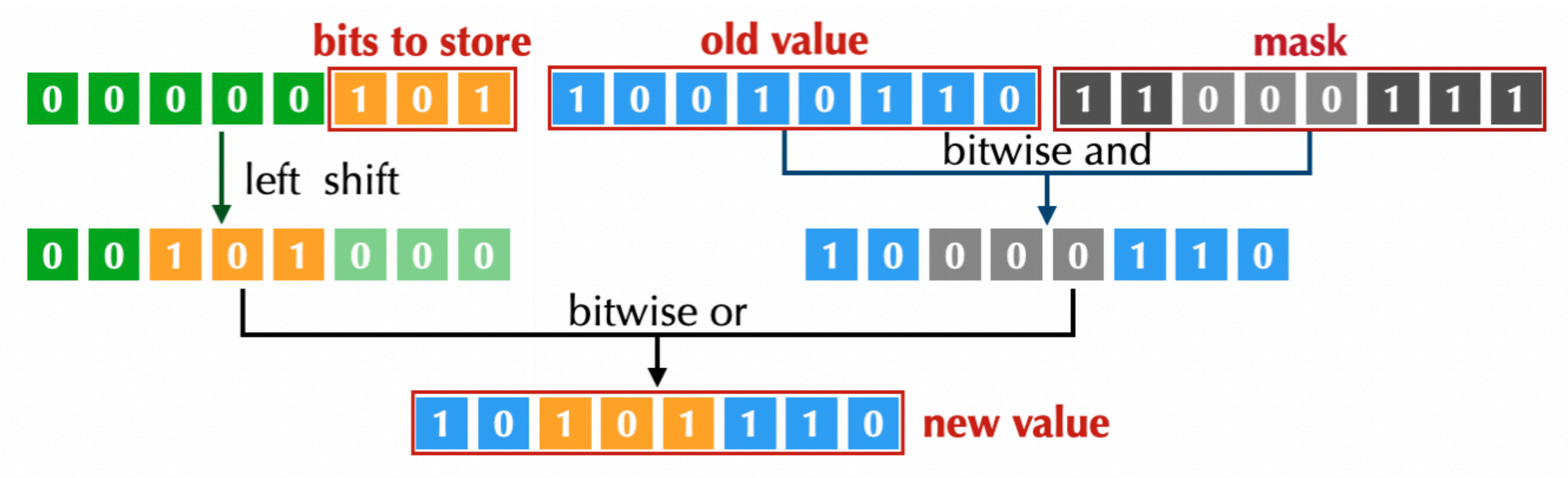






Performance

Partial-Bit Stores:
Modifying part of a hardware native type (e.g., int32)



Note: may need atomicRMW for thread safety!

Domain-Specific Optimizations

- **Bit-struct store fusion (1.91x faster)**
  - Stores to multiple components of a single bit struct can be fused into one
- **Thread-safety inference (2.15x faster)**
  - No need to atomicRMW if we know a store is thread-safe
- **Bit array vectorization (150x faster on GoL)**
  - Let each i32 register represent 32xi1 (32-wide vectorization)

End-to-end Performance

- **Factors**
  - Encoding/decoding takes time (slower)
  - But quantization reduces memory bandwidth consumption (faster)
- ***In reality those two factors fight against each other and one will win***
  - MPM: 1.03-1.14x **faster** with quantization
  - Stable fluids: 1.27x **slower** (shared exponent is slow)

Quantization on Taichi
Conclusions

- **Saves space (1.57~8x)**
- **Easy to use (3% LoC modification)**
  - Flexibly switching between different quantization schemes
- **Good performance (comparable, or even faster than full-precision)**
  - Domain-specific memory access optimizations are important
- **No significant visual quality degradation (more details in paper)**

Thank you!

- **QuanTaichi is now officially part of Taichi**
  - Compiler: https://github.com/taichi-dev/taichi
  - Demos and microbenchmarks: https://github.com/taichi-dev/quantaichi
- **More about Taichi:** https://taichi.graphics

Taichi
Programming Language