太极图形课

第03讲 Advanced Data Layouts

taichi

太极图形课

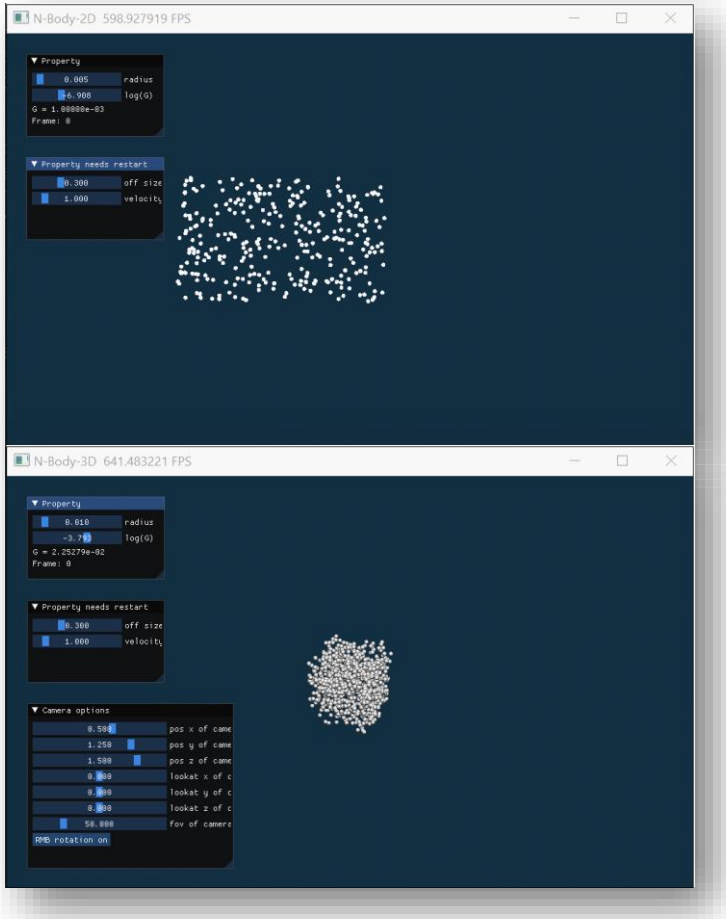
第03讲 Advanced Data Layouts

taichi

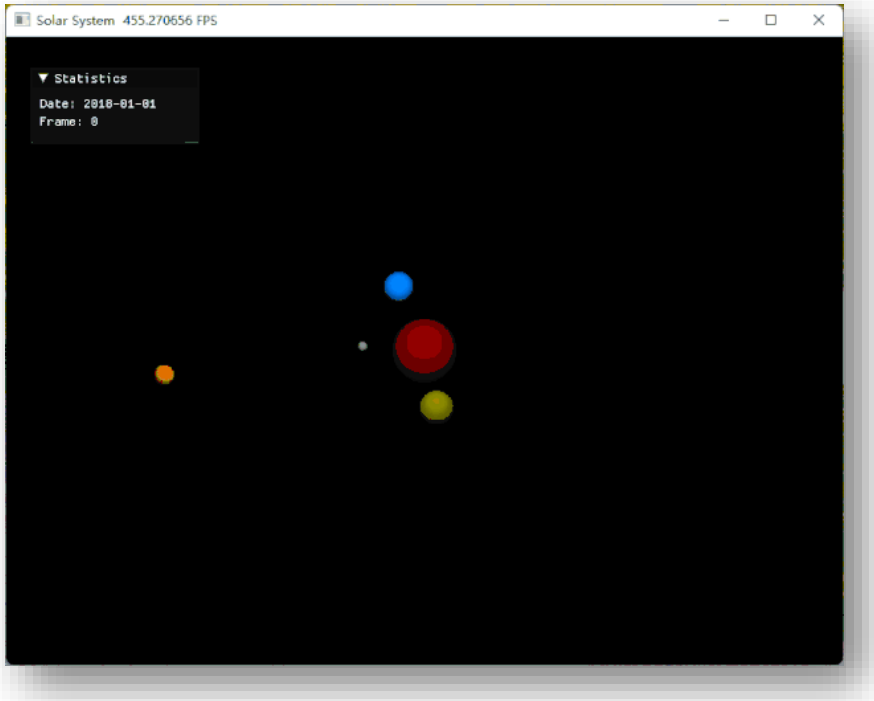
Recap

- Metaprogramming
- Object-oriented programming

N-body systems

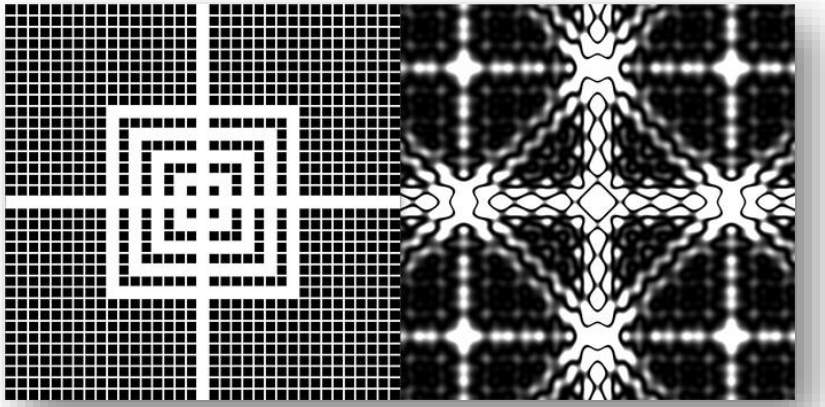


2/3D N-Body Dynamics
@Rabmelon



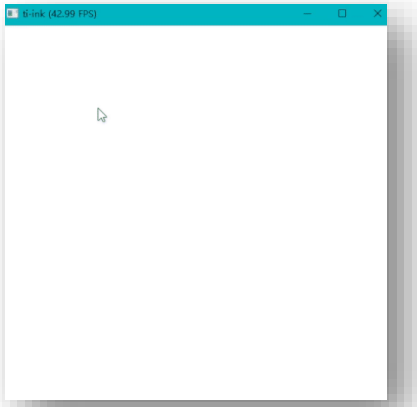
Solar System
@0xzhang

ODOP examples:



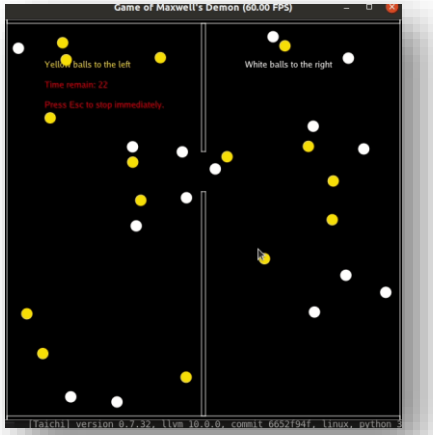
Diffraction @Y-jx007

Bezier @Zydiii

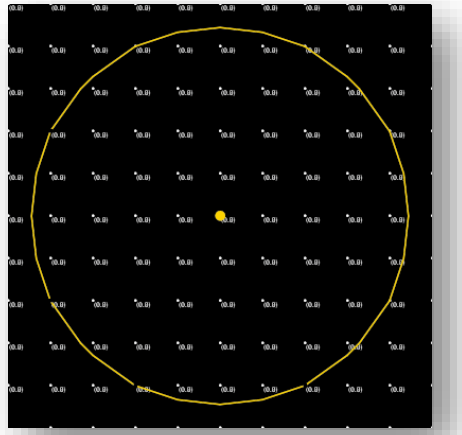


Moxi (墨戏) @Vineyo

Ant Colony
@theAfish

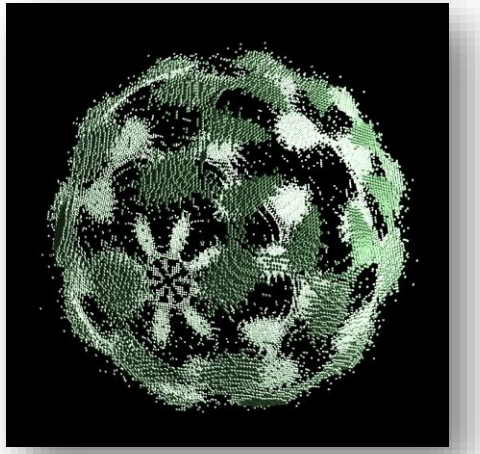


Maxwell's Demon game
@507C

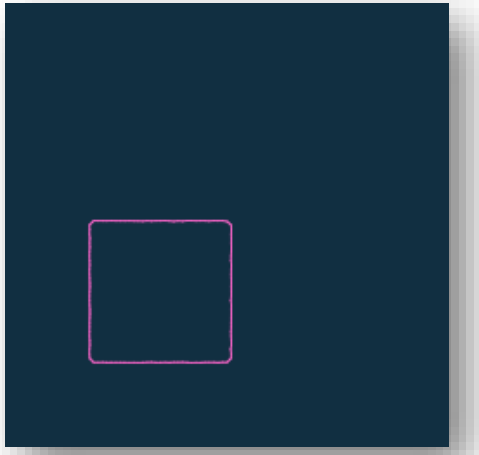


Marching Squares
@AlbertLiDesign

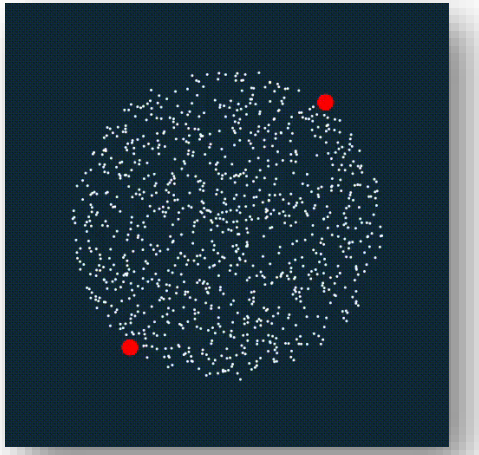
Other HW assignments are welcome as well!



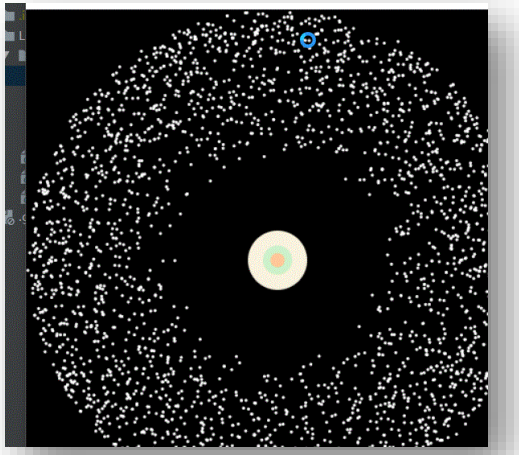
Mandelbulb
@rockeyshao



MPM88 + March Squares
@wangfeng70117



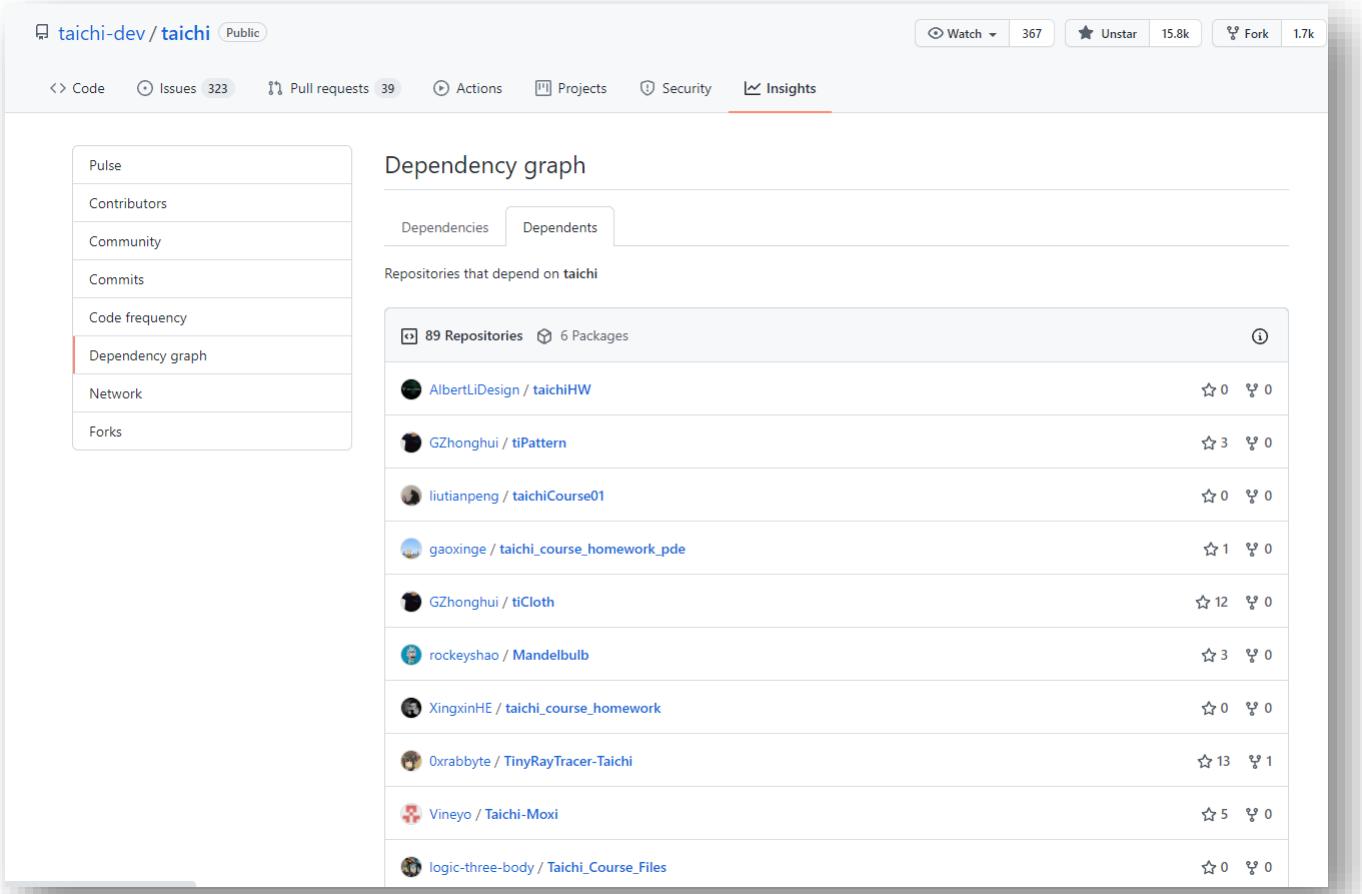
N-body with black hole(s)
@szl2



N-body with black hole(s)
@logic-three-body

Gifts for the gifted

- Check your Github issues ☺



Outline Today

- Advanced dense data layouts
- Sparse data layouts

Outline Today

- Advanced dense data layouts
- Sparse data layouts

Advanced dense data layouts

Taichi

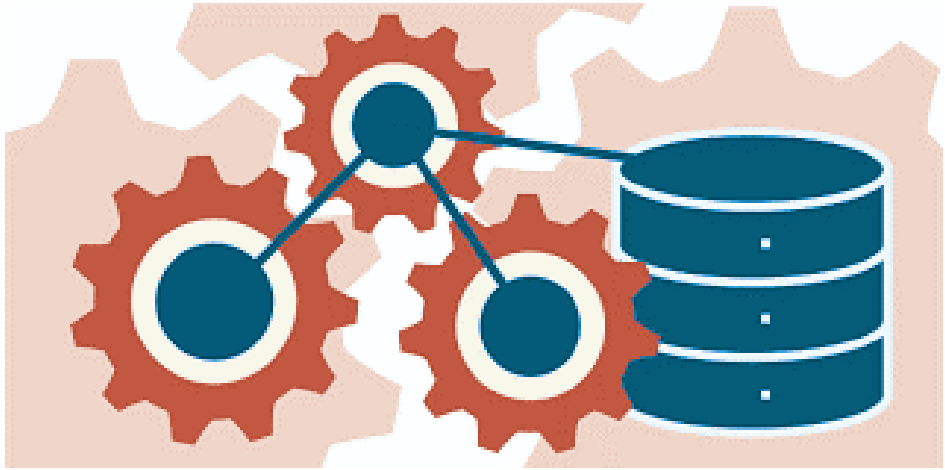
- ti.field()
- @ti.kernel
  - Optimized for ti.field()
- OOP
  - @data_oriented

Taichi: A data-oriented programming language

- ti.**field**()

- @ti.kernel
  - Optimized for ti.**field**()

- OOP
  - @**data_oriented**



Init

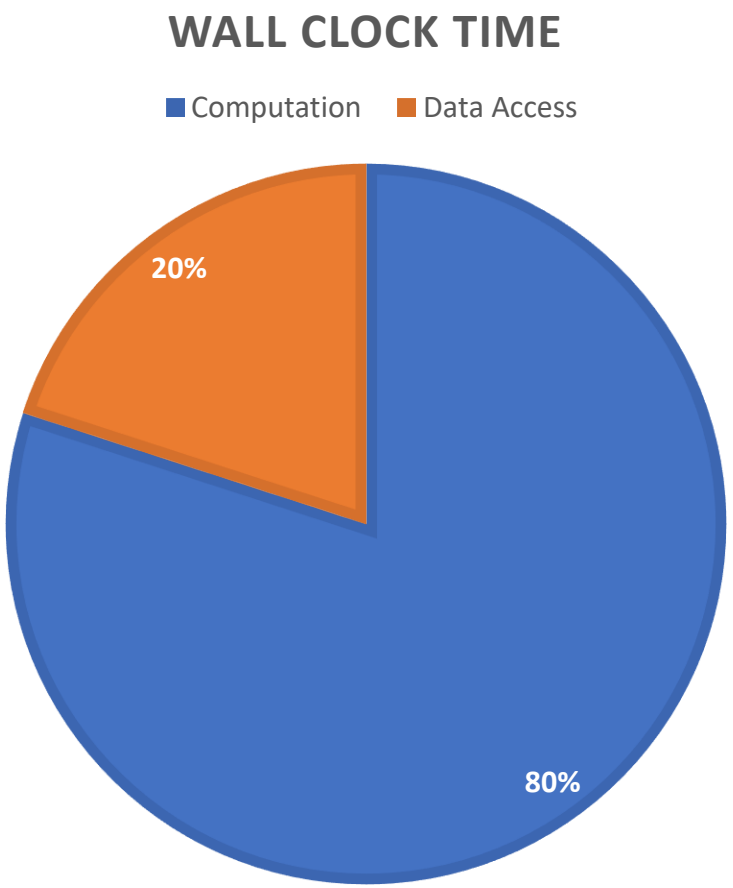
Data

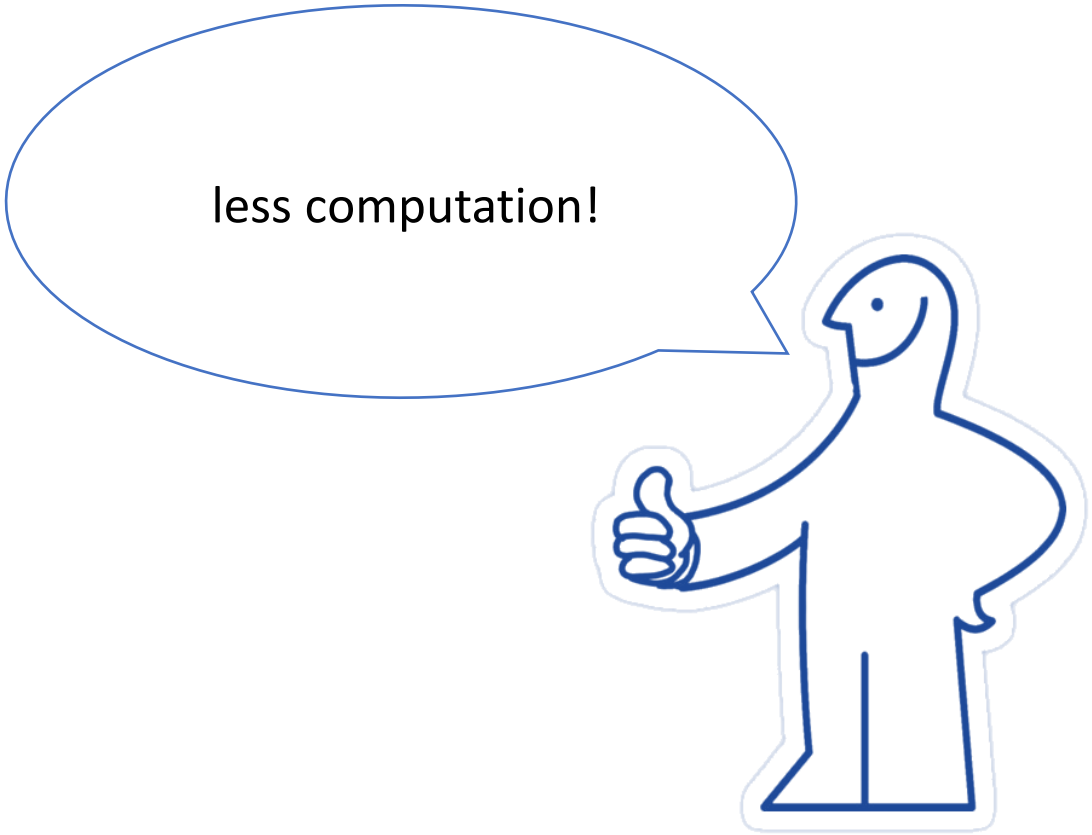
Computation

Visualization

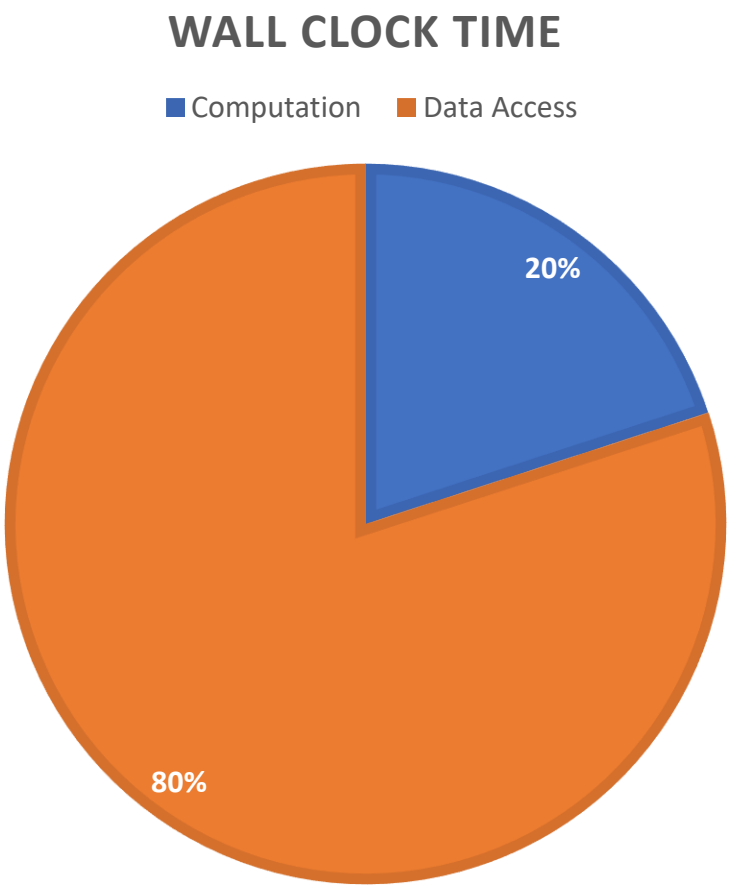
```python
import taichi as ti

ti.init(ti.gpu)

# gravitational constant 6.67408e-11, using 1 for simplicity
G = 1
PI = 3.141592653

# number of planets
N = 300
# unit mass
m = 5
# galaxy size
galaxy_size = 0.4
# planet radius (for rendering)
planet_radius = 2
# init vel
init_vel = 120

# time-step size
h = 1e-5
# substepping
substepping = 10

# pos, vel and force of the planets
# Nx2 vectors
pos = ti.Vector.field(2, ti.f32, N)
vel = ti.Vector.field(2, ti.f32, N)
force = ti.Vector.field(2, ti.f32, N)

@ti.kernel
def initialize():
    center = ti.Vector([0.5, 0.5])
    for i in range(N):
        theta = ti.random() * 4 * PI
        r = (ti.sqrt(ti.random()) * 0.7 + 0.3) * galaxy_size
        offset = r * ti.Vector([ti.cos(theta), ti.sin(theta)])
        pos[i] = center+offset
        vel[i] = [-offset.y, offset.x]
        vel[i] *= init_vel

@ti.kernel
def compute_force():
    # clear force
    for i in range(N):
        force[i] = ti.Vector([0.0, 0.0])

    # compute gravitational force
    for i in range(N):
        p = pos[i]
        for j in range(N):
            if i != j: # double the computation for a better memory footprint and load balance
                diff = p-pos[j]
                r = diff.norm(1e-5)

                # gravitational force -(GMm / r^2) * (diff/r) for i
                f = -G * m * m * (1.0/r)**3 * diff

                # assign to each particle
                force[i] += f

@ti.kernel
def update():
    dt = h/substepping
    for i in range(N):
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]

gui = ti.GUI('N-body problem', (512, 512))

initialize()
while gui.running:

    for i in range(substepping):
        compute_force()
        update()

    ...

    gui.clear(0x112F41)
    gui.circles(pos.to_numpy(), color=0xffffff, radius=planet_radius)
    gui.show()
```

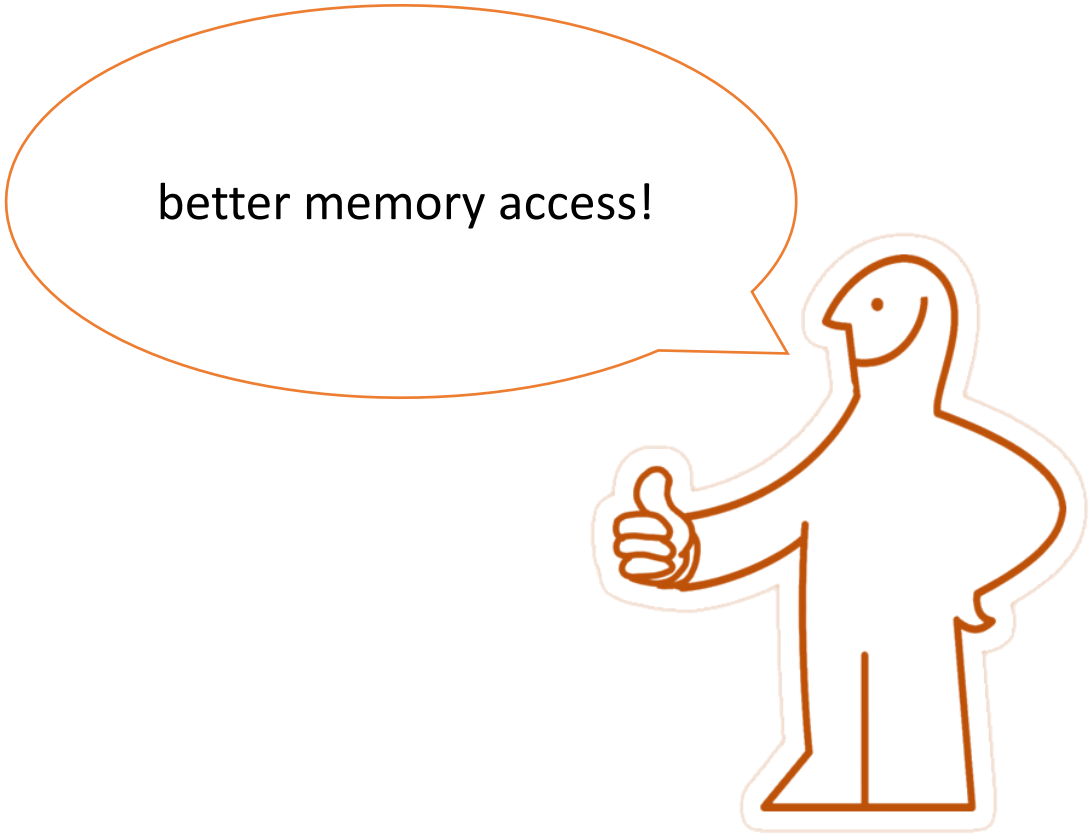
Performance @CPU…





Performance @GPU…

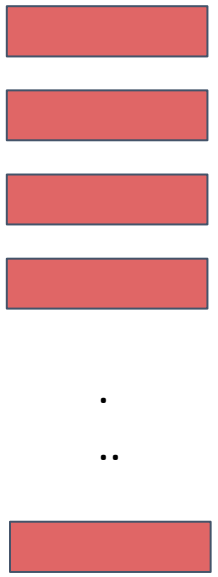


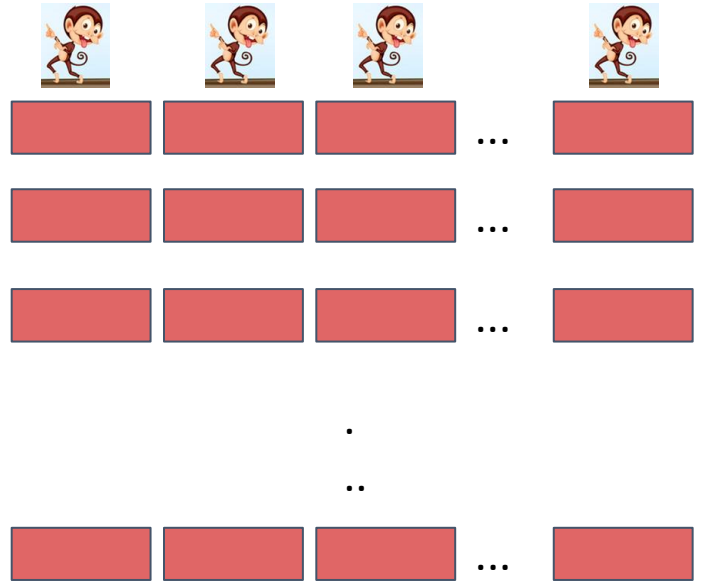


搬砖 Example （a slide from @禹鹏）

```python
def banzhuan():
    for i in range(len(zhuan)):
      move(zhuan[i])
```

```python
@ti.kernel
def banzhuan():
    for i in zhuan:
        move(zhuan[i])
```





Before we go: packed mode

- Initialized in *ti.init()*
- Decides whether to pad the data to the power of two
  - Default choice: *packed=False*, will do the padding
  - We assume *packed=True* in this class for simplicity

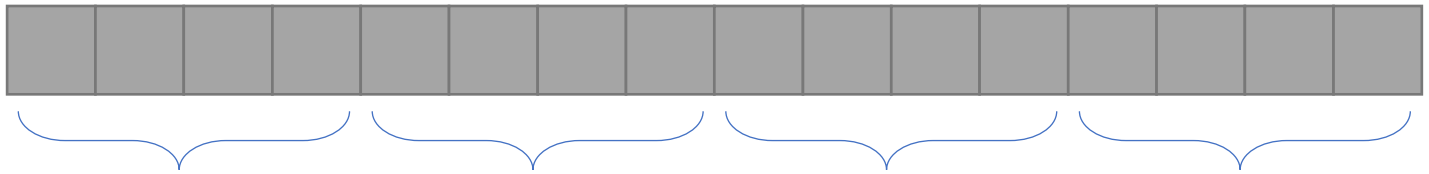
```
ti.init()  # default: packed=False
a = ti.field(ti.i32, shape=(18, 65))  # padded to (32, 128)
```

```
ti.init(packed=True)
a = ti.field(ti.i32, shape=(18, 65))  # no padding
```

Taichi: optimized for data-access

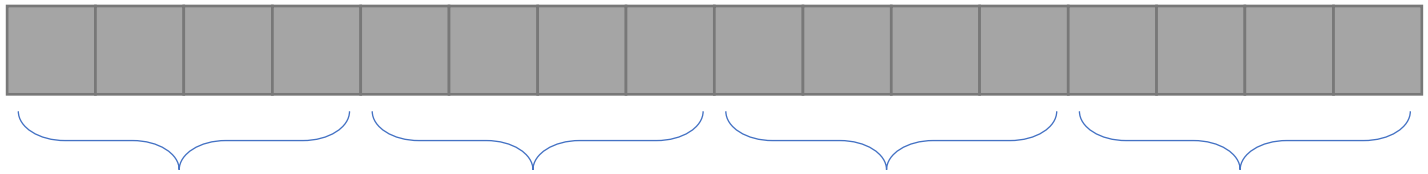
```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

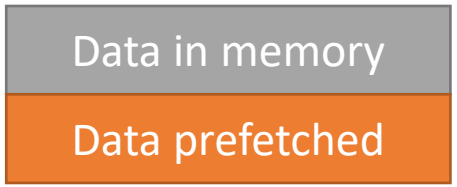
Taichi: optimized for data-access



```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

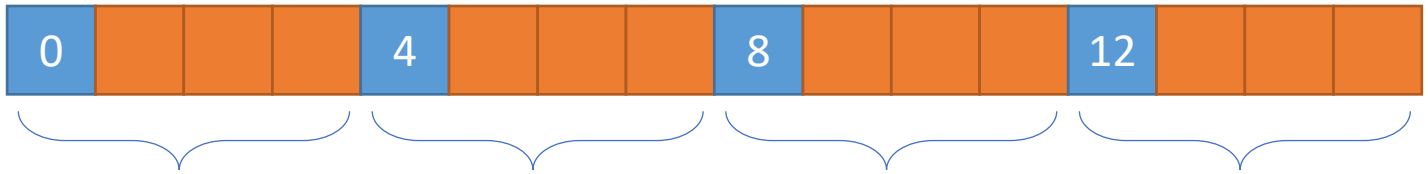
Taichi: optimized for data-access

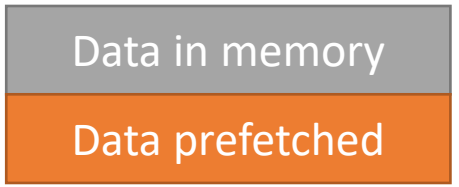





```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

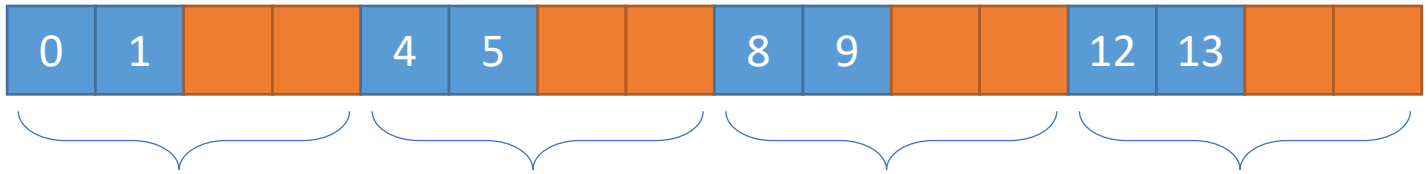
Taichi: optimized for data-access






```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

Taichi: A data-oriented programming language

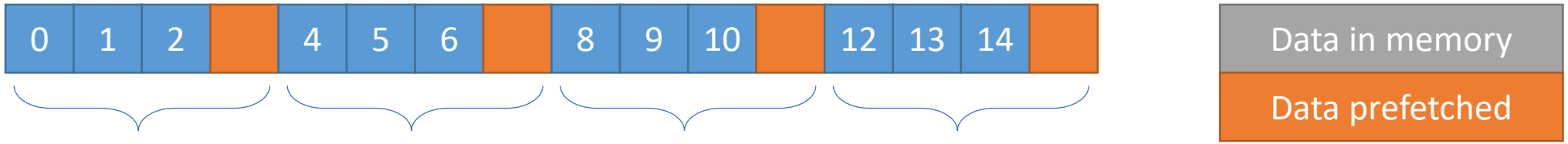


Data in memory

Data prefetched

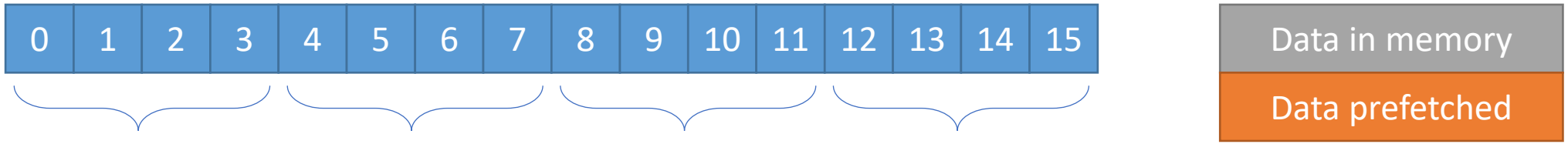
```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

Taichi: optimized for data-access



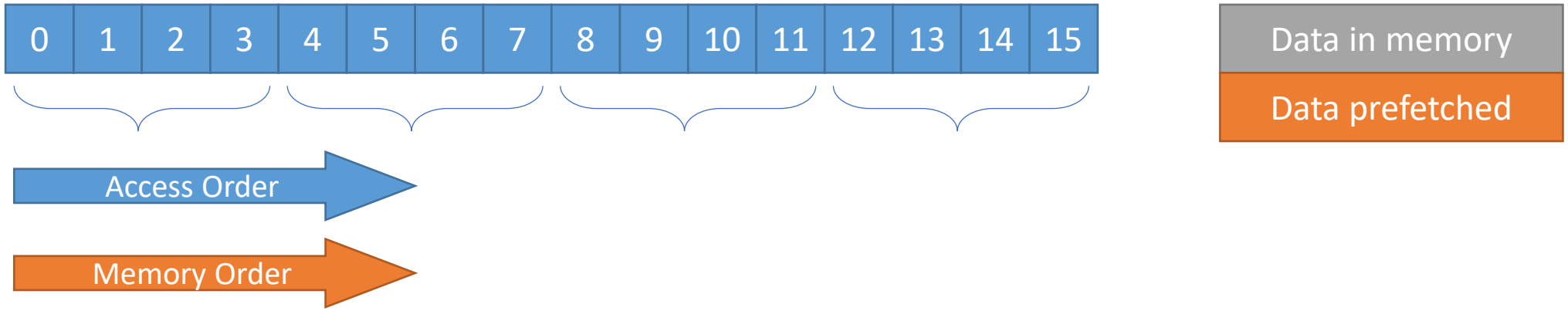

```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

Taichi: optimized for data-access



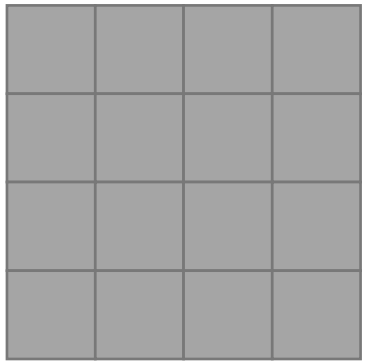

```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

Taichi: optimized for data-access



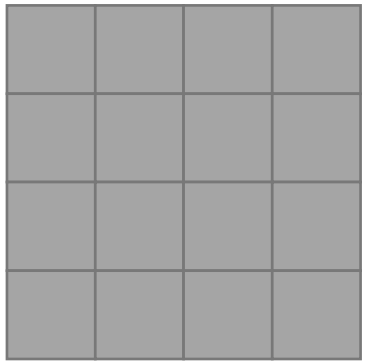

```
x = ti.field(ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        x[i] = i

fill()
```

How about multi-dimensional fields?



```
x = ti.field(ti.i32, shape = (4, 4))

@ti.kernel
def fill():
    for i,j in x:
        x[i,j] = 10*i + j

fill()
```

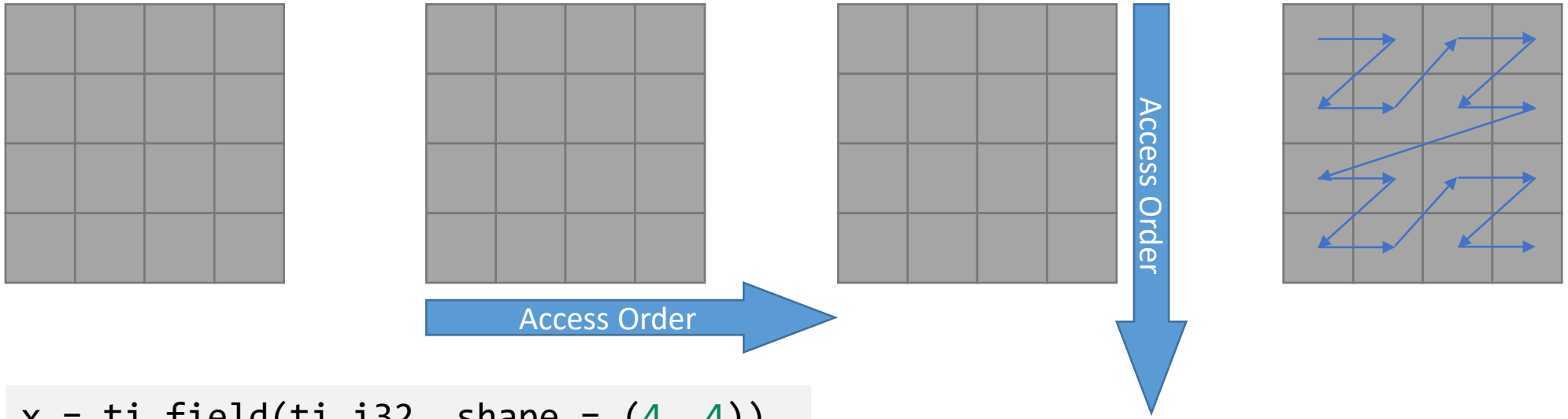
N-D fields are stored in our 1-D memory…



An N-D field
we think

An N-D field
we store

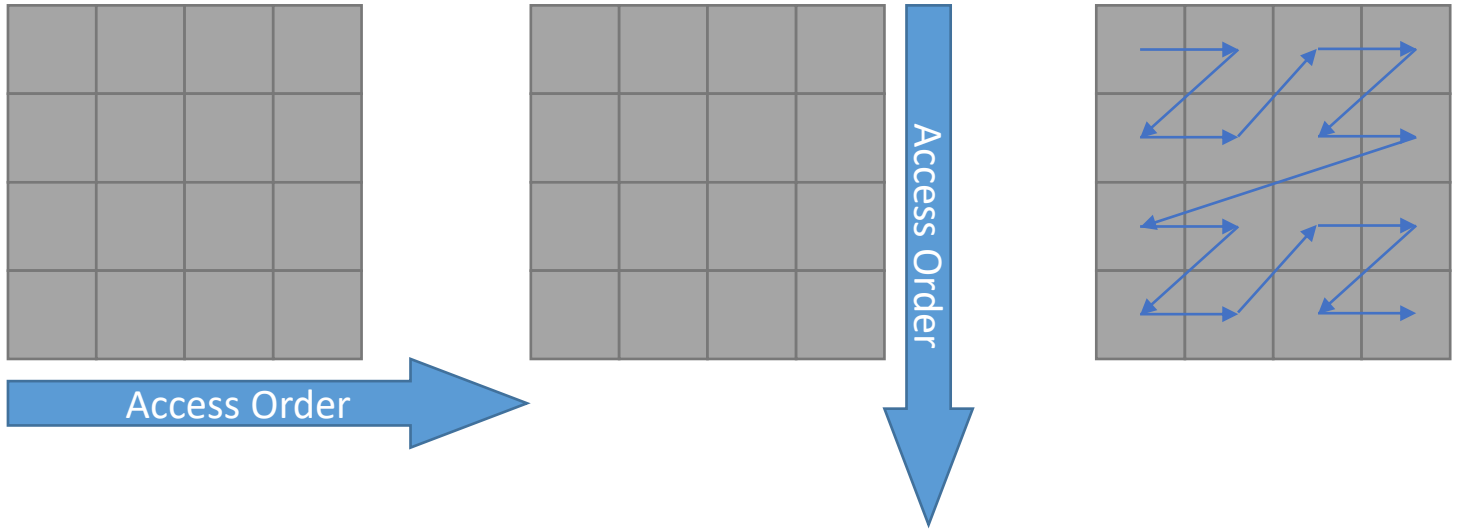
However the access pattern is not determined…



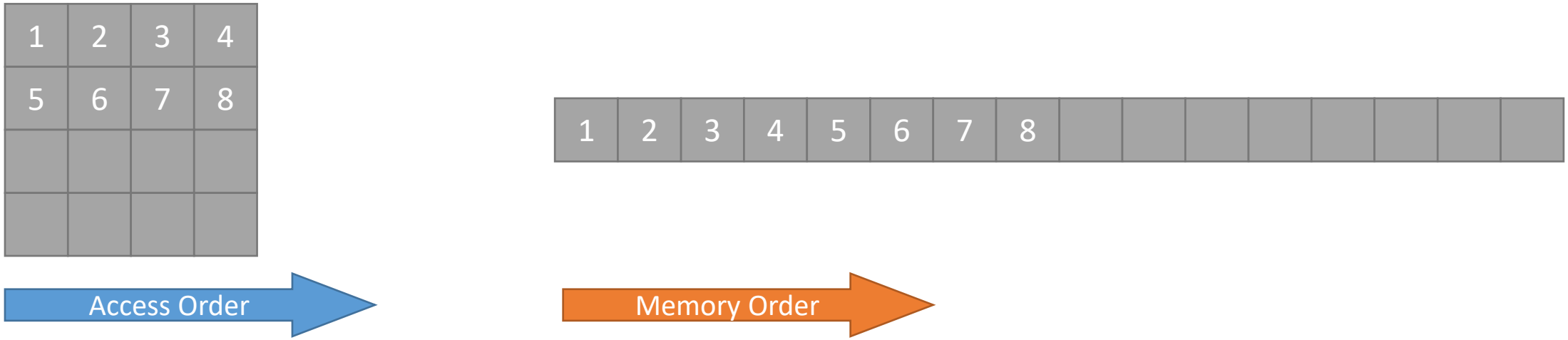

```
x = ti.field(ti.i32, shape = (4, 4))

@ti.kernel
def fill():
    for i,j in x:
        x[i,j] = 10*i + j

fill()
```

What we want:

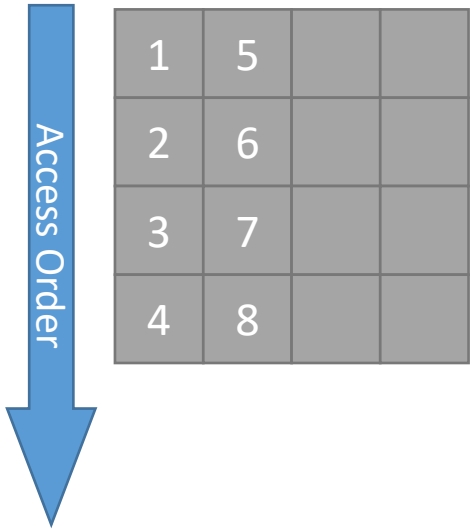
- Store our data in a memory-access-friendly way.

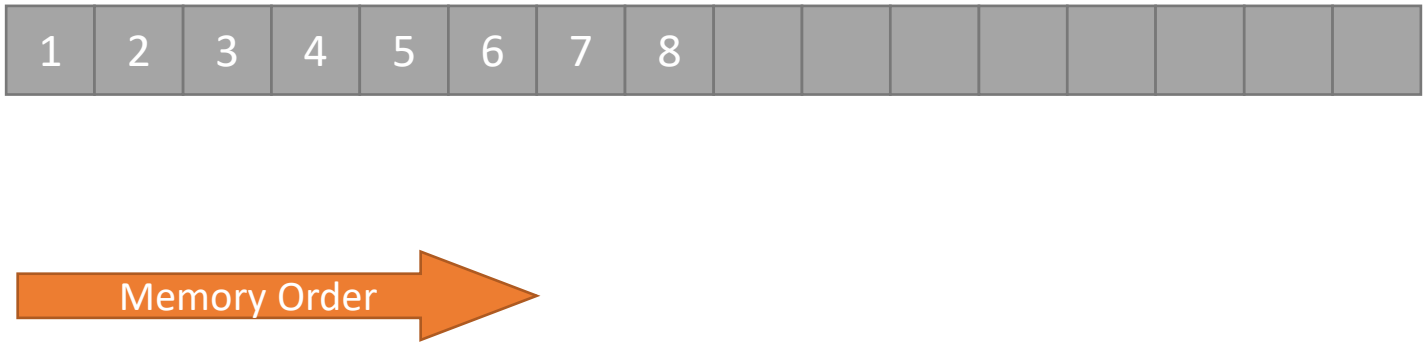


Ideal memory layout of an N-D field:

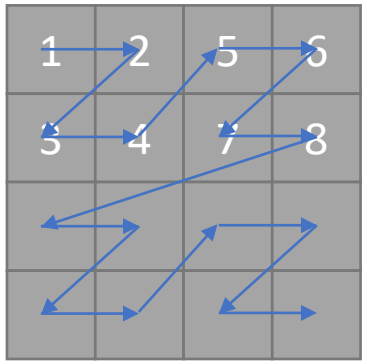


Ideal memory layout of an N-D field:



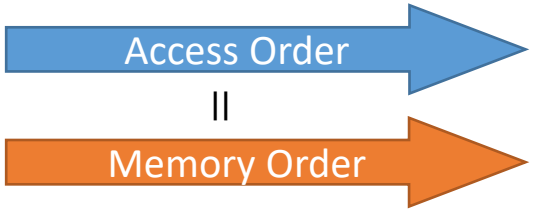


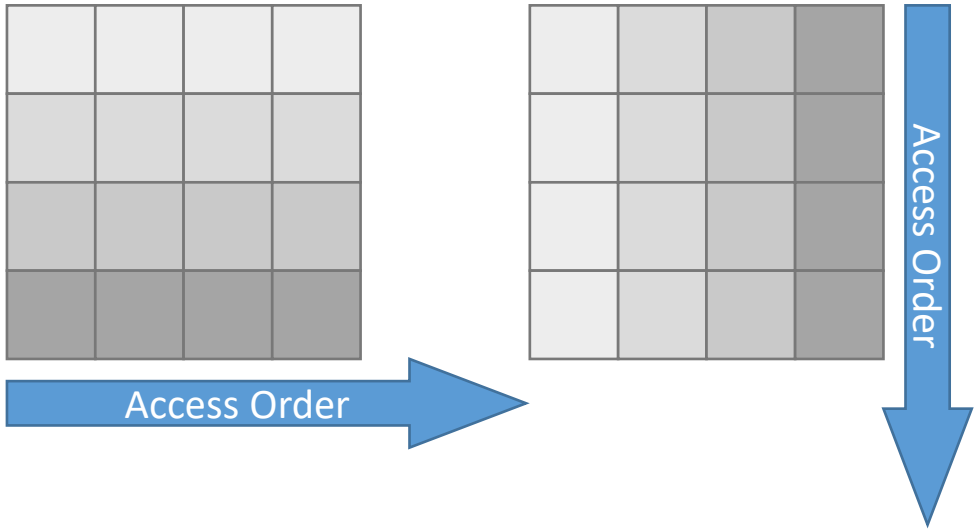
Ideal memory layout of an N-D field:



What we want:

- Store our data in a memory-access-friendly way.

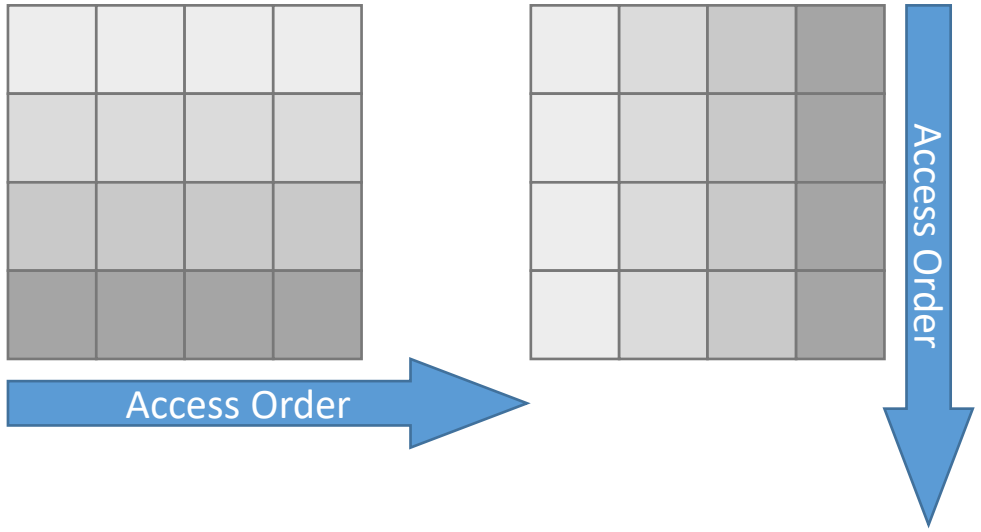


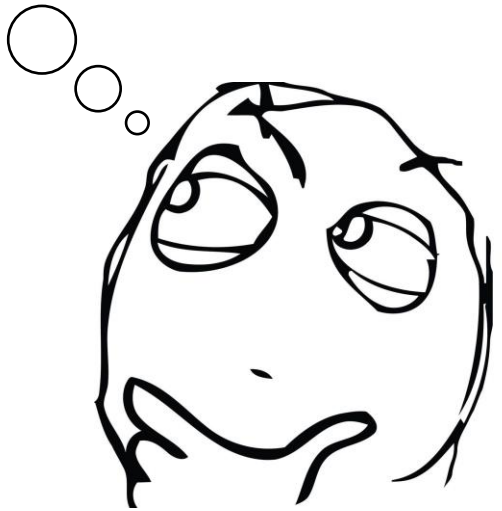


What we want:

- memory-access-friendly?
- using ti.field()?

```
x = ti.field(ti.i32, shape = (4, 4))
```

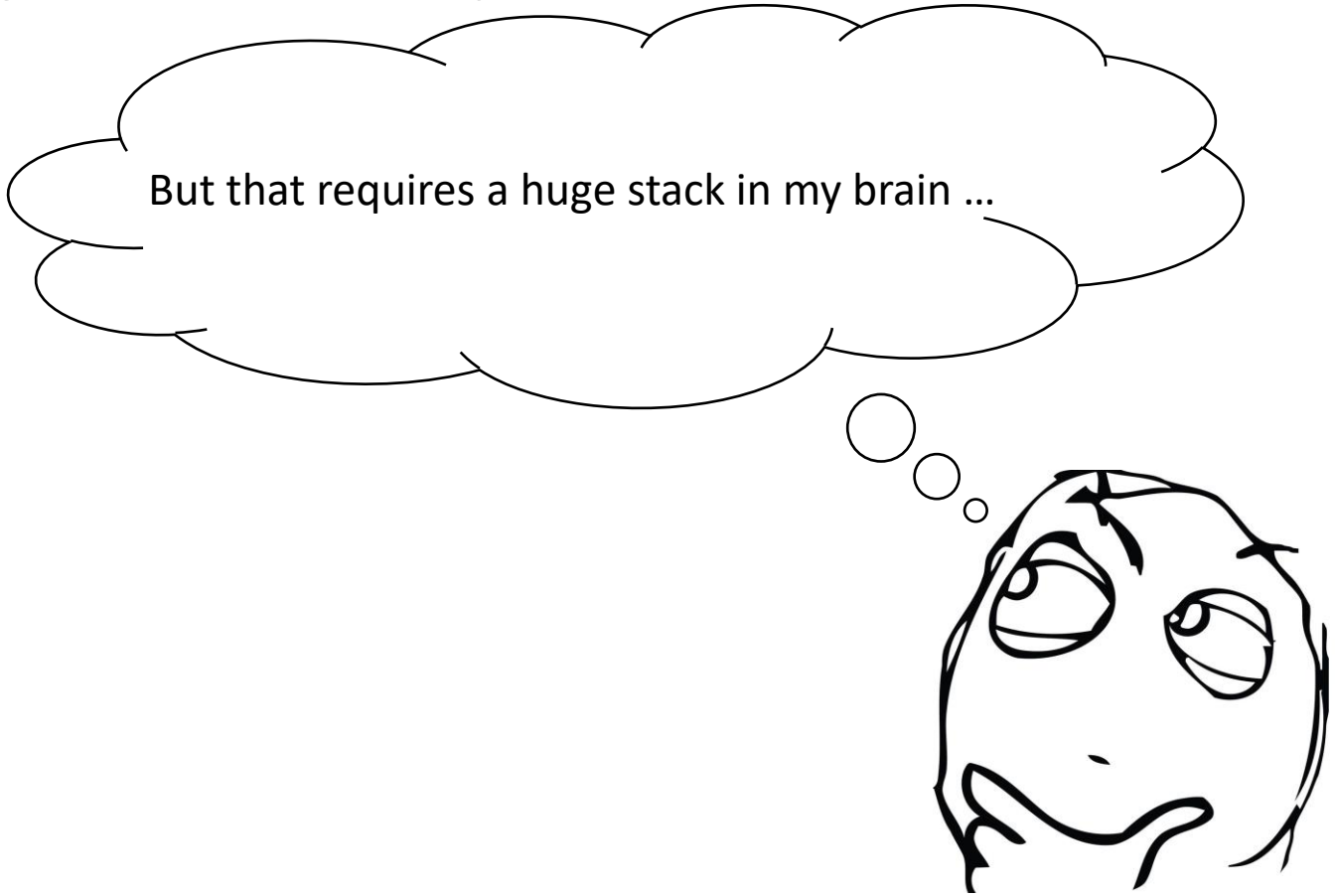




Access row/col-majored arrays/fields in C/C++

```c
int x[3][2];  // row-major
int y[2][3];  // column-major


foo(){
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            do_something(x[i][j]);
        }
    }

    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < 3; i++) {
            do_something(y[j][i]);
        }
    }
}
```

C/C++



Upgrade your ti.field()

Layout 101: from *shape* to *ti.root*

```
x = ti.Vector.field(3, ti.f32, shape = 16)
```

```
x = ti.Vector.field(3, ti.f32)
ti.root.dense(ti.i, 16).place(x)
```

ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*

Layout 101: from *shape* to *ti.root*

```
x = ti.Vector.field(3, ti.f32, shape = 16)
```

```
x = ti.Vector.field(3, ti.f32)
ti.root.dense(ti.i, 16).place(x)
```

Root

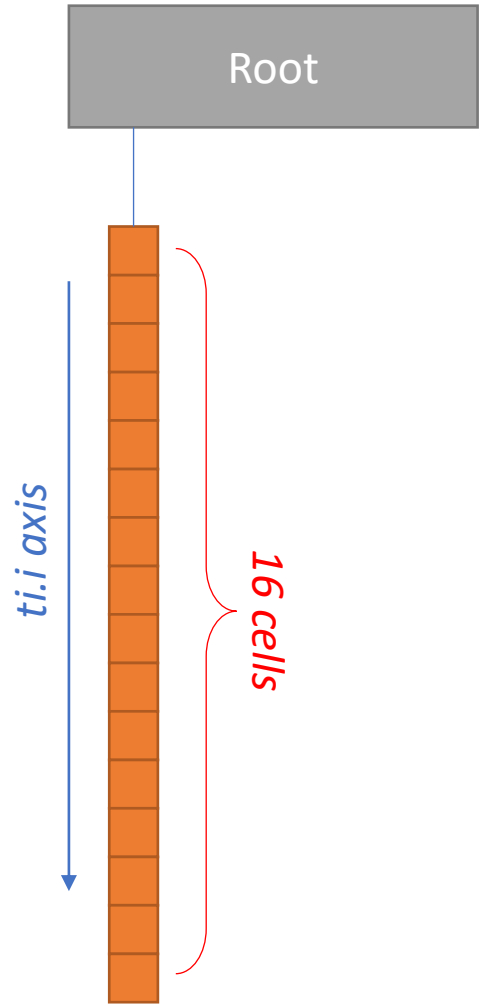
ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*

Layout 101: from *shape* to *ti.root*

```
x = ti.Vector.field(3, ti.f32, shape = 16)
```

```
x = ti.Vector.field(3, ti.f32)
ti.root.dense(ti.i, 16).place(x)
```

ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*

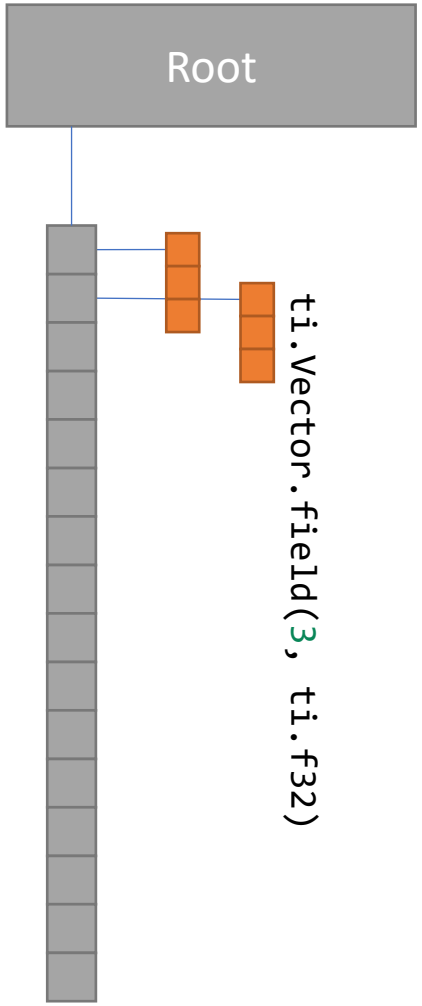


Layout 101: from *shape* to *ti.root*

```
x = ti.Vector.field(3, ti.f32, shape = 16)
```

```
x = ti.Vector.field(3, ti.f32)
ti.root.dense(ti.i, 16).place(x)
```

ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*



ti.root: more examples:

```
x = ti.field(ti.f32, shape=())
```

```
x = ti.field(ti.f32)
ti.root.place(x)
```

```
x = ti.field(ti.f32, shape=3)
```

```
x = ti.field(ti.f32)
ti.root.dense(ti.i, 3).place(x)
```

```
x = ti.field(ti.f32, shape=(3, 4))
```

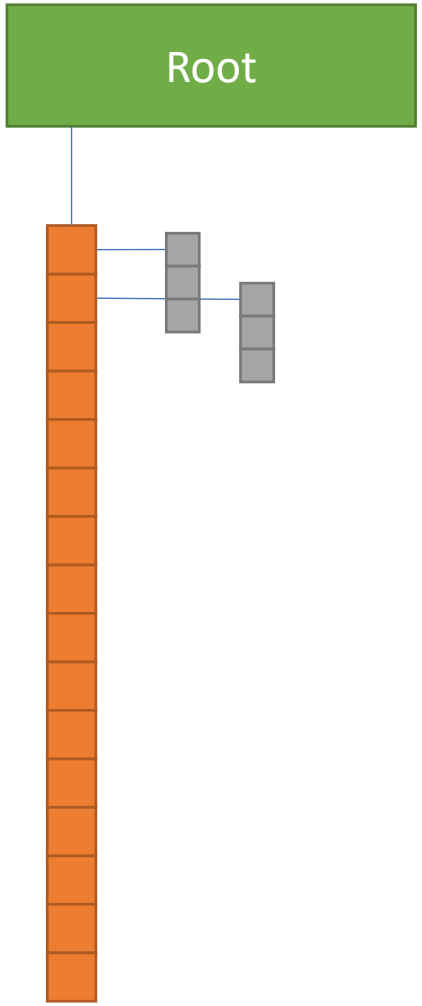
```
x = ti.field(ti.f32)
ti.root.dense(ti.ij, (3, 4)).place(x)
```

```
x = ti.Matrix.field(2, 2, ti.f32, shape=5)
```

```
x = ti.Matrix.field(2, 2, ti.f32)
ti.root.dense(ti.i, 5).place(x)
```

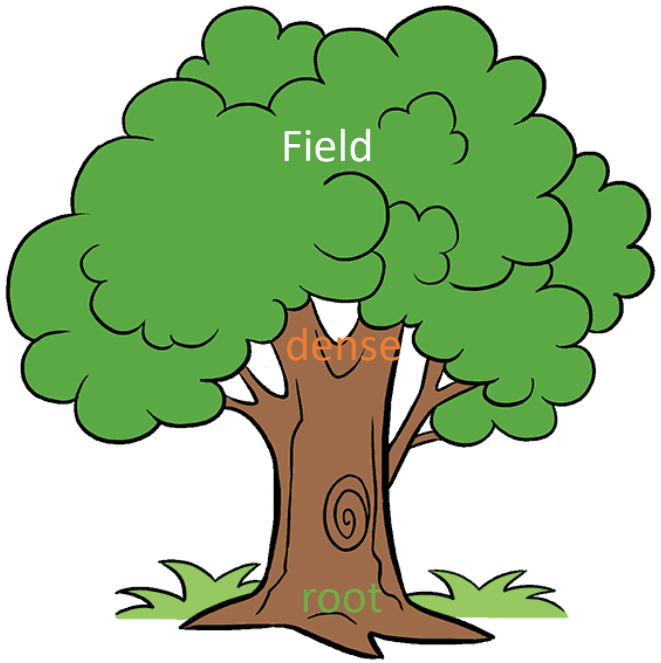
ti.root: the root of a *SNode-tree*

- SNode: Structural Node
- An SNode tree:
  - ti.root ← the root of the SNode-tree
    - .dense() ← a dense container describing shape
      - .place(ti.field()) ← a field describing cell data
    - …



ti.root: the root of a *SNode-tree*

- SNode: Structural Node
- An SNode tree:
  - ti.root ← the root of the SNode-tree
    - .dense() ← a dense container describing shape
      - .place(ti.field()) ← a field describing cell data
    - …

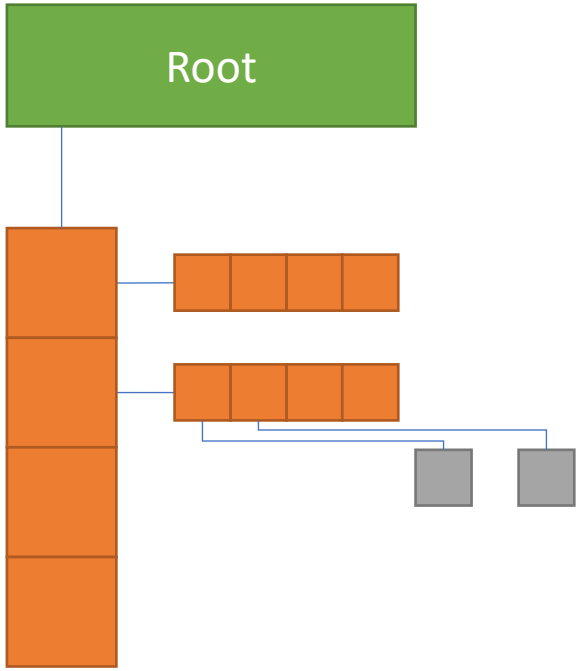


The SNode-tree

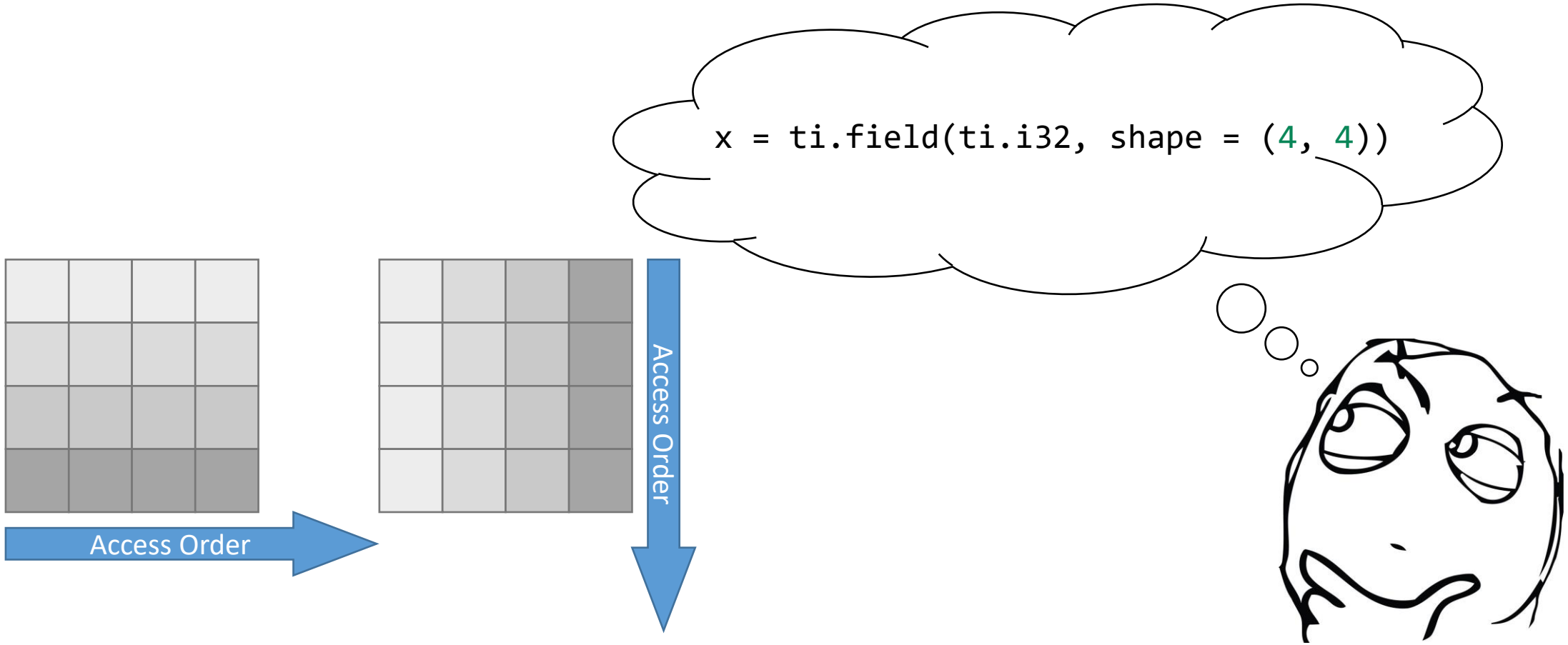
```
x = ti.field(ti.i32, shape = (4, 4))
```

```
x = ti.field(ti.i32)
ti.root.dense(ti.ij, (4, 4)).place(x)
```

```
x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).dense(ti.j, 4).place(x)
```

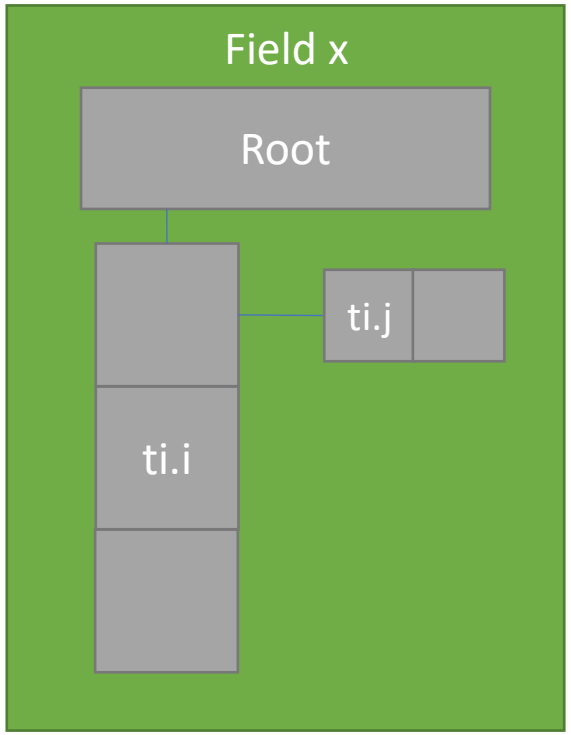


Row-major v.s. column-major

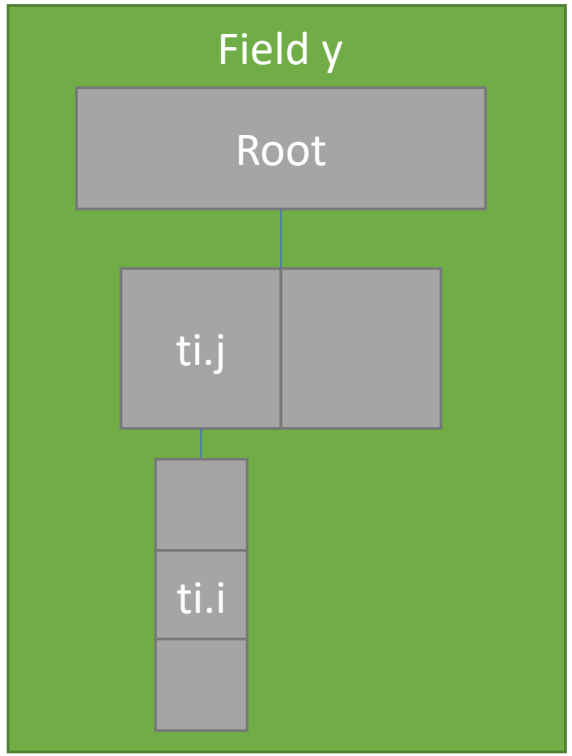


Row-major v.s. column-major

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 3).dense(ti.j, 2).place(x)   # row-major
ti.root.dense(ti.j, 2).dense(ti.i, 3).place(y)   # column-major
```

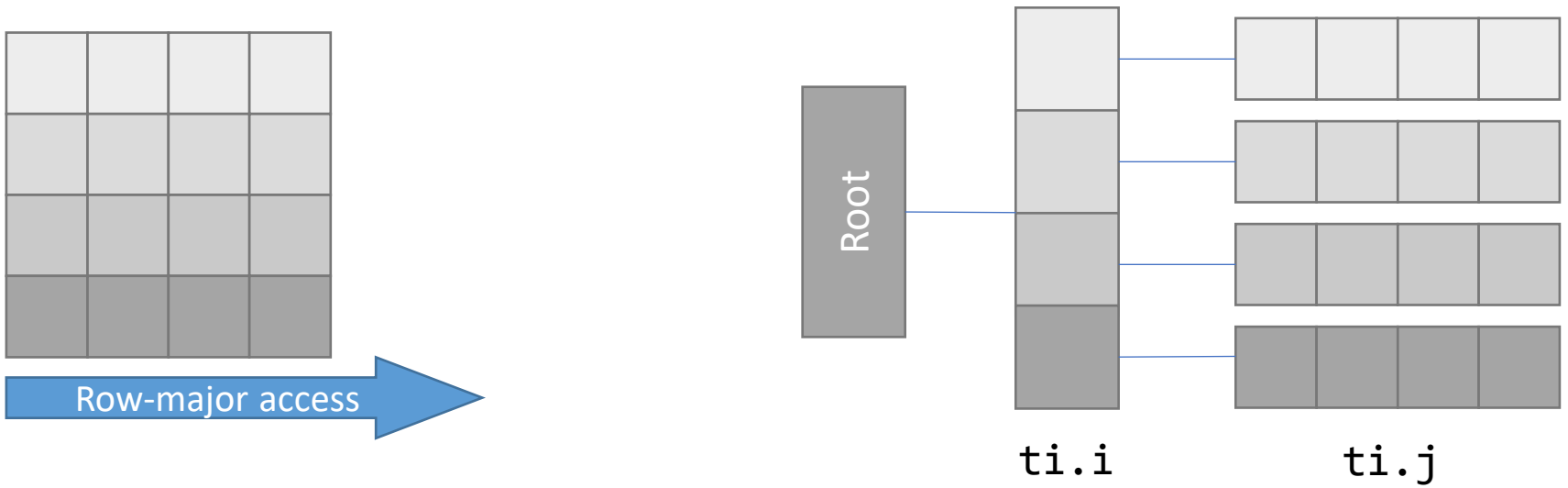


 # address: low ..... High
 # x: x[0, 0] x[0, 1] x[1, 0] x[1, 1] x[2, 0] x[2, 1]
 # y: y[0, 0] y[1, 0] y[2, 0] y[0, 1] y[1, 1] y[2, 1]



Row-major access

```
x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).dense(ti.j, 4).place(x)   # row-major
```



Access row/col-majored fields

```
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

x = ti.field(ti.i32)
ti.root.dense(ti.i, 3).dense(ti.j, 2).place(x)
# row-major

@ti.kernel
def fill():
    for i,j in x:
        x[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in x:
        print("x[",i,",",j,"]=",x[i,j],sep='', en
d=' ')

fill()
print_field()
```

```
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

x = ti.field(ti.i32)
ti.root.dense(ti.j, 2).dense(ti.i, 3).place(x)
# column-major

@ti.kernel
def fill():
    for i,j in x:
        x[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in x:
        print("x[",i,",",j,"]=",x[i,j],sep='', en
d=' ')

fill()
print_field()
```

Access row/col-majored fields

```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

x = ti.field(ti.i32)
ti.root.dense(ti.i, 3).dense(ti.j, 2).place(x)
# row-major

@ti.kernel
def fill():
    for i,j in x:
        x[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in x:
        print("x[",i,",",j,"]=",x[i,j],sep='', en
d=' ')

fill()
print_field()
```

Loop over ***ti.j*** first

```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

x = ti.field(ti.i32)
ti.root.dense(ti.j, 2).dense(ti.i, 3).place(x)
# column-major

@ti.kernel
def fill():
    for i,j in x:
        x[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in x:
        print("x[",i,",",j,"]=",x[i,j],sep='', en
d=' ')

fill()
print_field()
```

Loop over ***ti.i*** first

Access row/col-majored arrays/fields in C/C++ v.s. in Taichi

```cpp
int x[3][2];  // row-major
int y[2][3];  // column-major


foo(){
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
            do_something(x[i][j]);
        }
    }

    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < 3; i++) {
            do_something(y[j][i]);
        }
    }
}
```

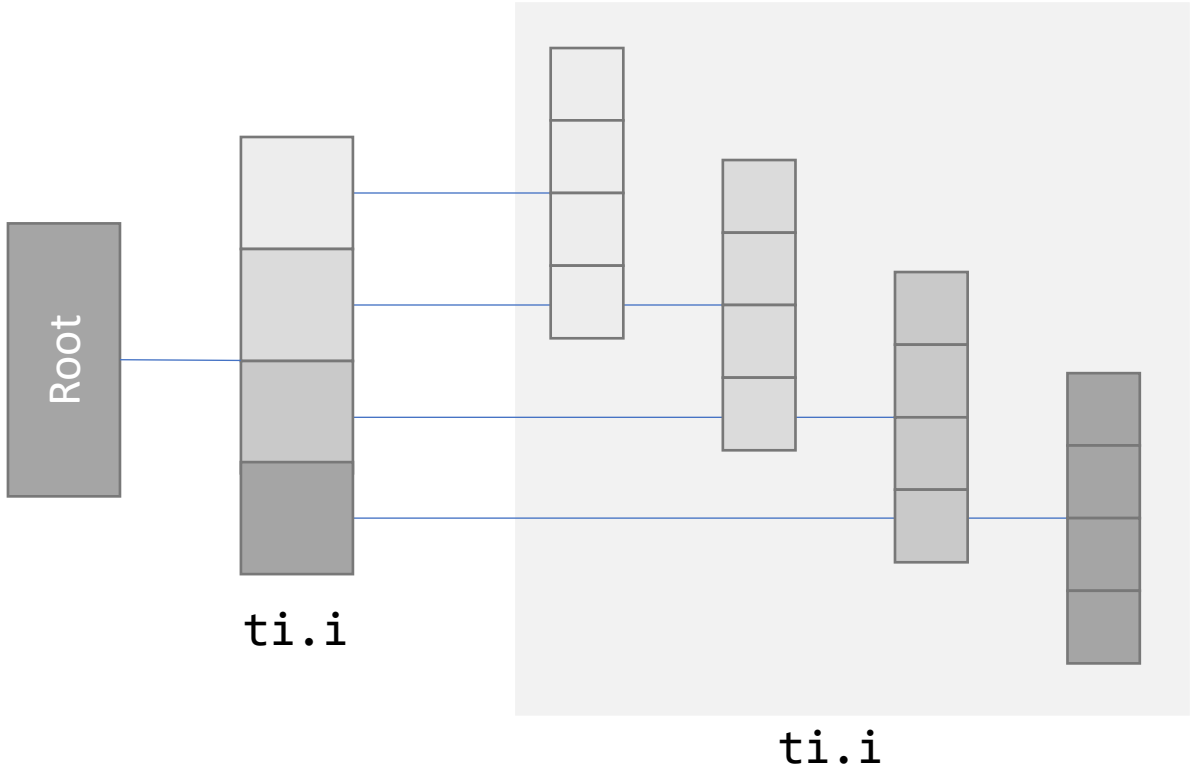
C/C++

```python
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 3).dense(ti.j, 2).place(x)   # row-major
ti.root.dense(ti.j, 2).dense(ti.i, 3).place(y)   # column-major

@ti.kernel
def foo():
    for i,j in x:
        do_something(x[i, j])

    for i,j in y:
        do_something(y[i, j])
```

Taichi (Python)

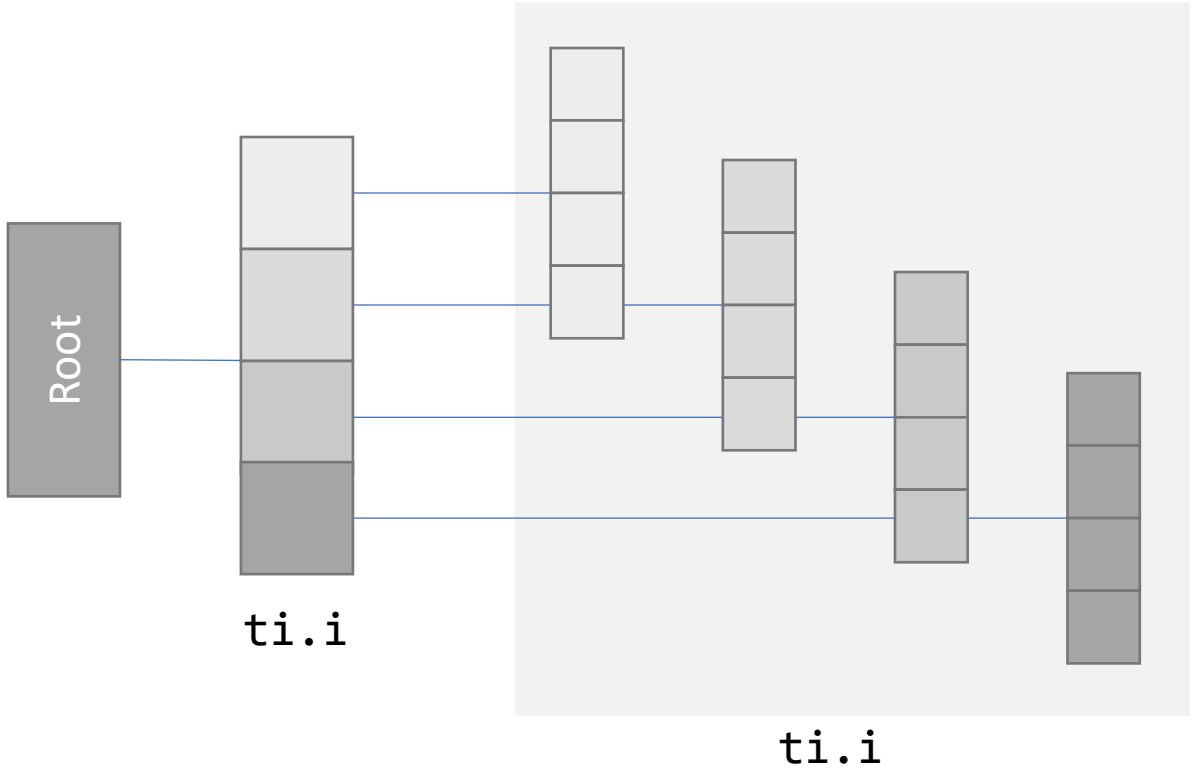
A special case:

```
x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).dense(ti.i, 4).place(x)   # what is this?
```



Hierarchical layouts

```
x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).dense(ti.i, 4).place(x)   # A hierarchical 1-D field
```



Access a hierarchical 1-D field

```
import taichi as ti
ti.init(arch = ti.cpu)

x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).dense(ti.i, 4).place(x)
# A hierarchical 1-D field

@ti.kernel
def print_id():
    for i in x:
        print(i, end = ' ')

print_id()
```

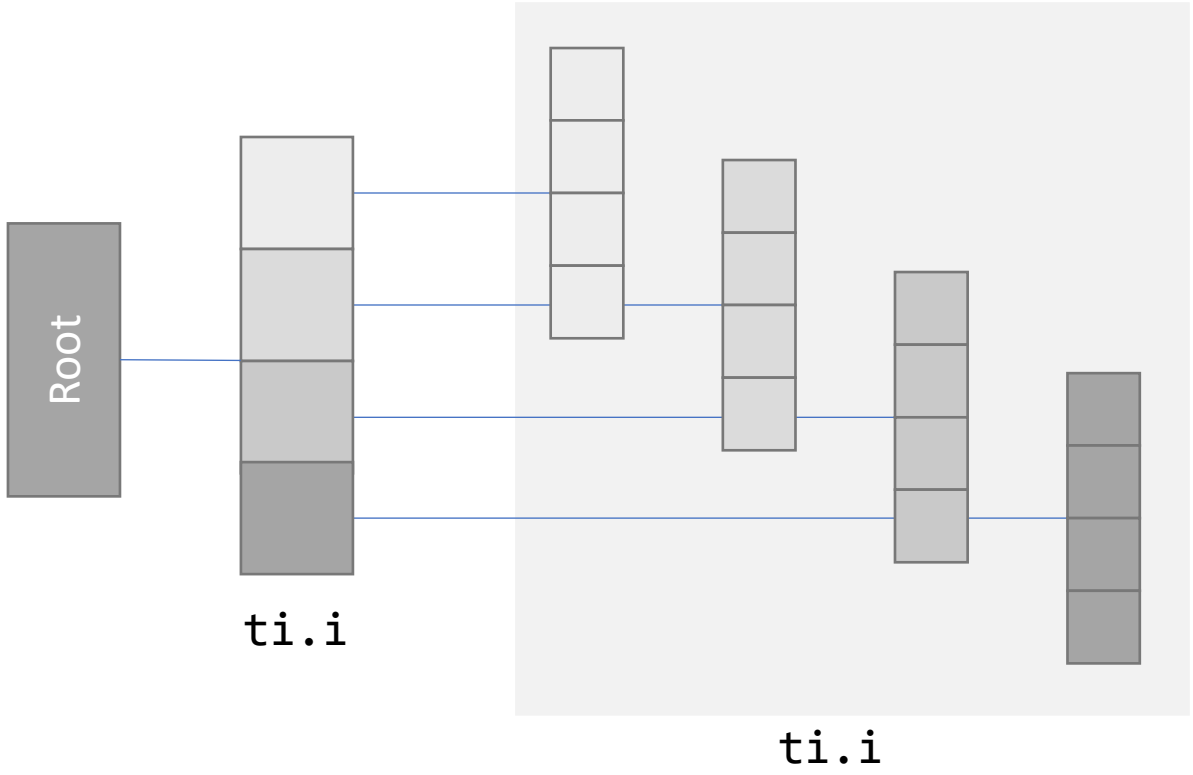
My first execution:
[Taichi] Starting on arch=x64
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

My second execution:
[Taichi] Starting on arch=x64
0 1 2 3 4 5 6 7 12 13 14 15 8 9 10 11

My third execution:
[Taichi] Starting on arch=x64
4 5 6 7 12 13 14 15 0 1 2 3 8 9 10 11

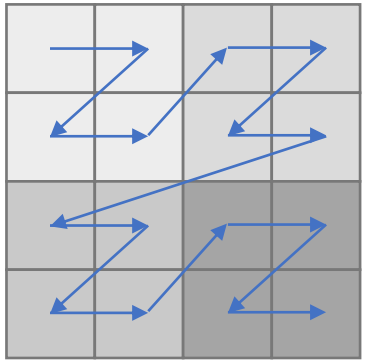
Hierarchical layouts

```
x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).dense(ti.i, 4).place(x)    # A hierarchical 1-D field
```

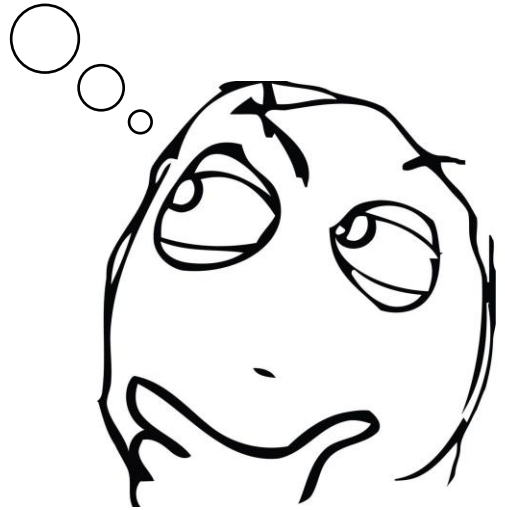


- Access like a 1-D field
- Store like a 2-D field (in blocks)

Block-majored access?

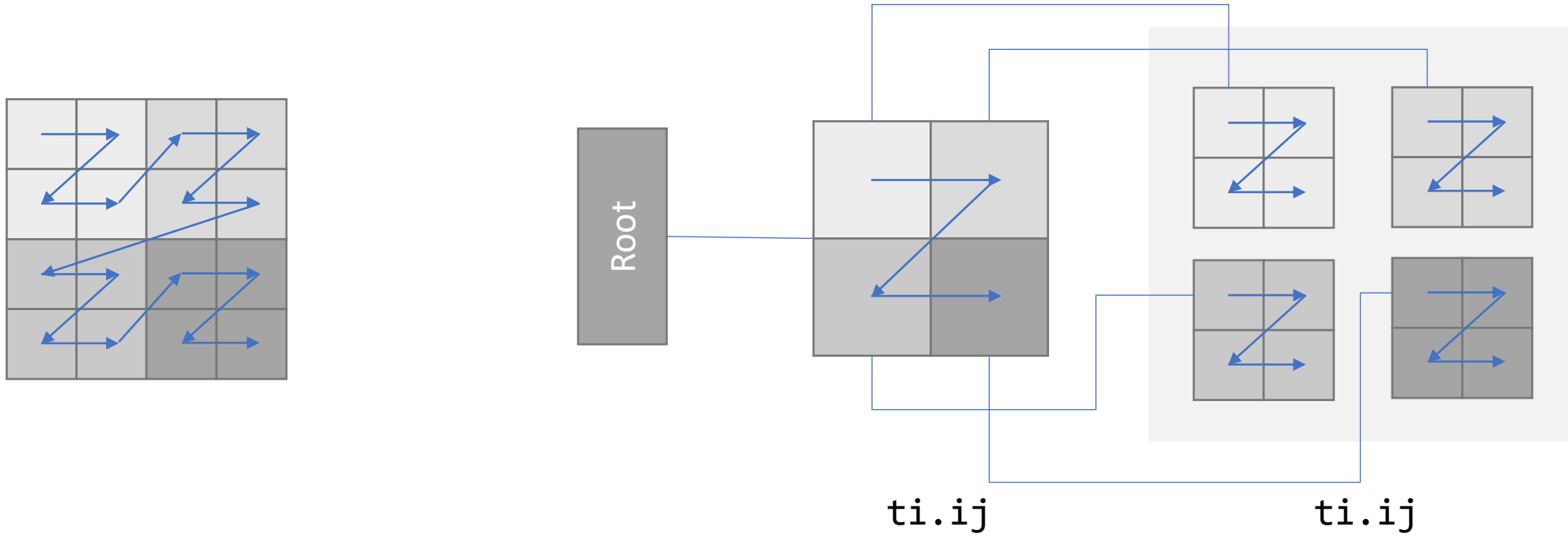


```
x = ti.field(ti.i32, shape = (4, 4))
```



Block-majored access using hierarchical fields

```
x = ti.field(ti.i32)
ti.root.dense(ti.ij, (2,2)).dense(ti.ij, (2,2)).place(x)    # block-major
```



Flat layouts v.s. hierarchical layouts

```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

z = ti.field(ti.i32, shape=(4,4))
# a row-majored flat layout, size = 4x4


@ti.kernel
def fill():
    for i,j in z:
        z[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in z:
        print("z[",i,",",j,"]=",z[i,j],sep='', end=' ')

fill()
print_field()
```

First loop over ***ti.j***, then ***ti.i***

```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

z = ti.field(ti.i32)
ti.root.dense(ti.ij, (2,2)).dense(ti.ij, (2,2)).place(z)
# a block-majored hierarchical layout, size = 4x4

@ti.kernel
def fill():
    for i,j in z:
        z[i, j] = i*10 + j

@ti.kernel
def print_field():
    for i,j in z:
        print("z[",i,",",j,"]=",z[i,j],sep='', end=' ')

fill()
print_field()
```

First loop over ***ti.j***, then ***ti.i,***
in 2x2 blocks

Why do we need block-majored access?

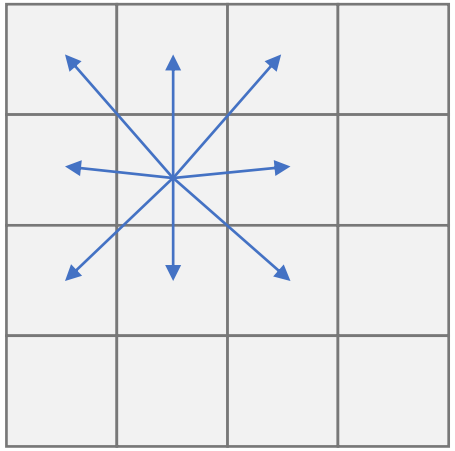

```
@ti.kernel
def update_flow(self):
    for P in ti.grouped(self.Flow):
        # self.Flow[P]=self.FlowNext[P]
        for i in ti.static(range(9)):
            prePos = P-self.e[i]
            self.Flow[P][i] = self.kar_avg[P][i] * \
                (self.FlowNext[P][self.k[i]] - self.FlowNext[prePos][i]) + \
                self.FlowNext[prePos][i]
```

```
self.e = ti.Vector.field(2, dtype=int, shape=9)
self.e[0] = ti.Vector([0, 0])
self.e[1] = ti.Vector([0, 1])
self.e[2] = ti.Vector([-1, 0])
self.e[3] = ti.Vector([0, -1])
self.e[4] = ti.Vector([1, 0])
self.e[5] = ti.Vector([1, 1])
self.e[6] = ti.Vector([-1, 1])
self.e[7] = ti.Vector([-1, -1])
self.e[8] = ti.Vector([1, -1])
```

Moxi (墨戏) @Vineyo

Why do we need block-majored access?

Moxi (墨戏) @Vineyo



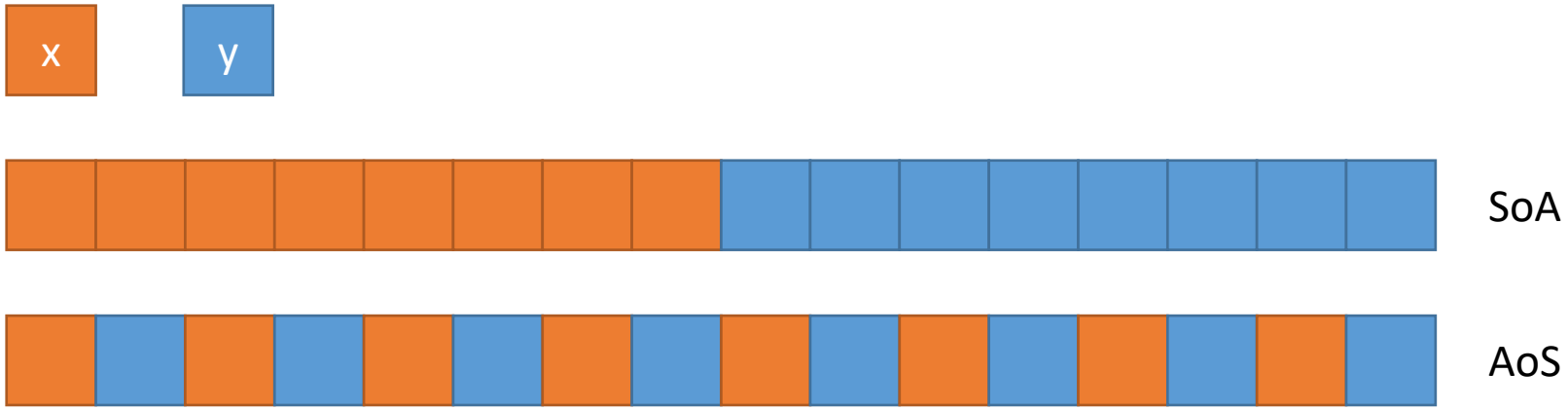
9-point stencil

```
self.e = ti.Vector.field(2, dtype=int, shape=9)
self.e[0] = ti.Vector([0, 0])
self.e[1] = ti.Vector([0, 1])
self.e[2] = ti.Vector([-1, 0])
self.e[3] = ti.Vector([0, -1])
self.e[4] = ti.Vector([1, 0])
self.e[5] = ti.Vector([1, 1])
self.e[6] = ti.Vector([-1, 1])
self.e[7] = ti.Vector([-1, -1])
self.e[8] = ti.Vector([1, -1])
```

Array of structures (AoS) v.s. structure of arrays (SoA) in C/C++

```
struct S1
{
    int x[8];
    int y[8];
}
S1 soa;
```

```
struct S2
{
    int x;
    int y;
}
S2 aos[8];
```

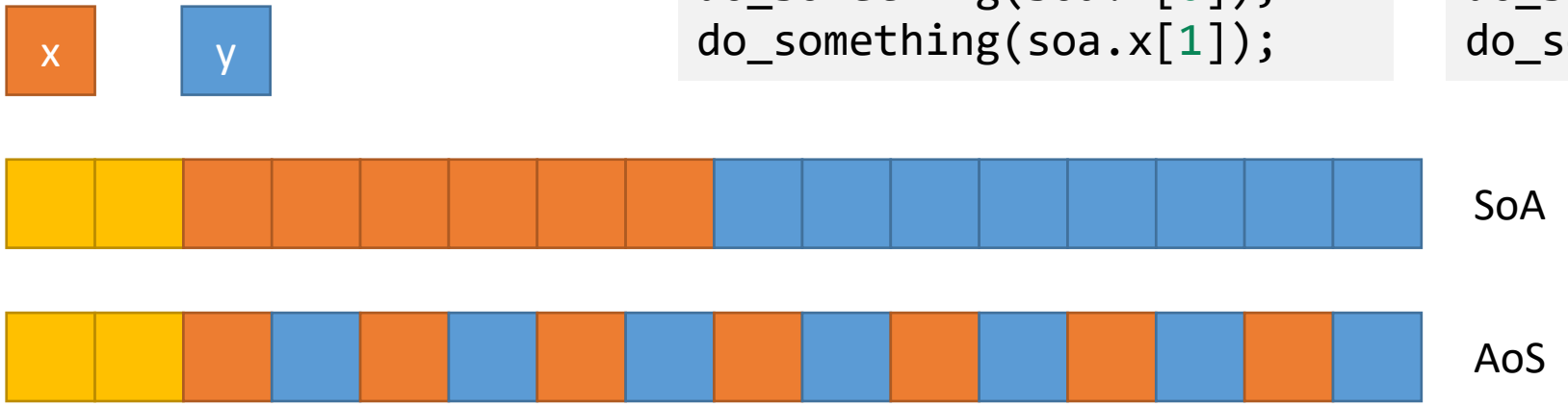


AoS v.s. SoA, which one is better?

- It really depends…

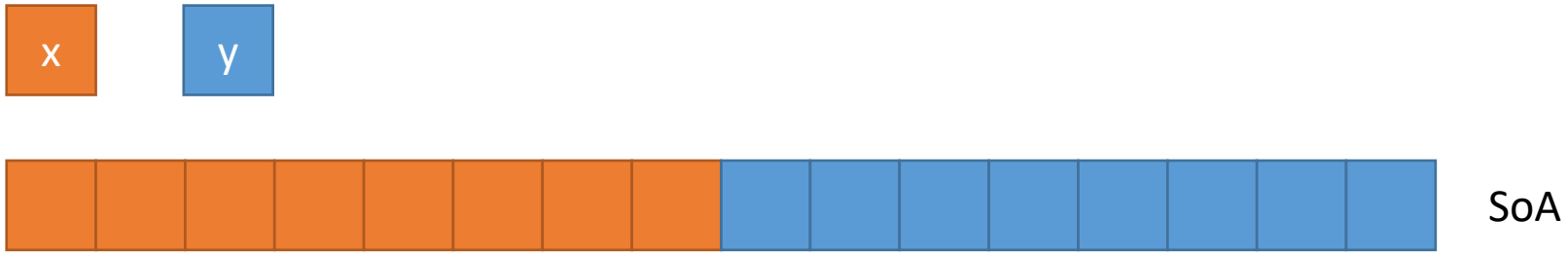
```
struct S1
{
    int x[8];
    int y[8];
}
S1 soa;

do_something(soa.x[0]);
do_something(soa.x[1]);
```

```
struct S2
{
    int x;
    int y;
}
S2 aos[8];

do_something(aos[0].x);
do_something(aos[0].y);
```



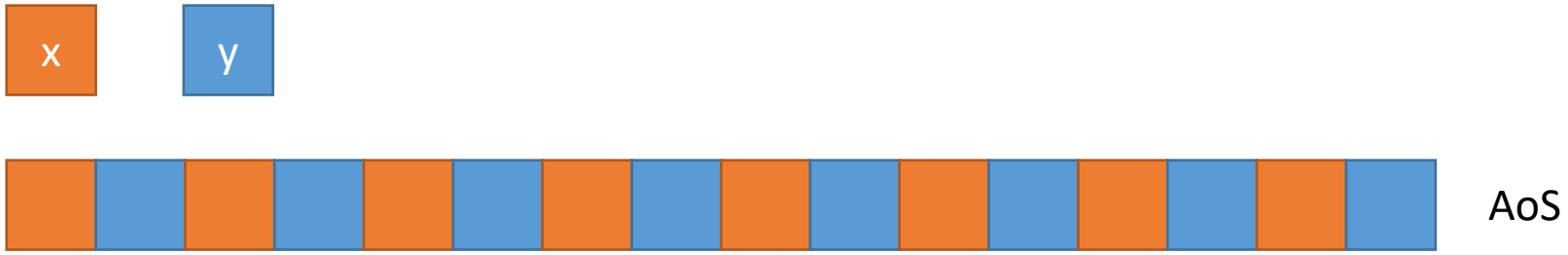
SoA in Taichi

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 8).place(x)
ti.root.dense(ti.i, 8).place(y)
#  address: low ........................... high
#           x[0]  x[1] … x[7]  y[0]  y[1] … y[7]
```



AoS in Taichi

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 8).place(x, y)
#  address: low ........................... high
#           x[0]  y[0]  x[1]  y[1] … x[7]  y[7]
```



AoS in Taichi

- Only same-shaped fields can be placed in AoS fashion

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 8).place(x)
ti.root.dense(ti.i, 16).place(y)
# different-shaped fields x and y can not be placed in AoS fashion
```

  - Shapes are determined by SNodes

```
x = ti.field(ti.i32)
y = ti.Vector.field(2,ti.i32)
ti.root.dense(ti.i, 8).dense(ti.j, 8).place(x, y)
# scalar field x and vector field y can be placed in AoS fashion
```

Switching between AoS and SoA in Taichi

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 8).place(x, y)
```

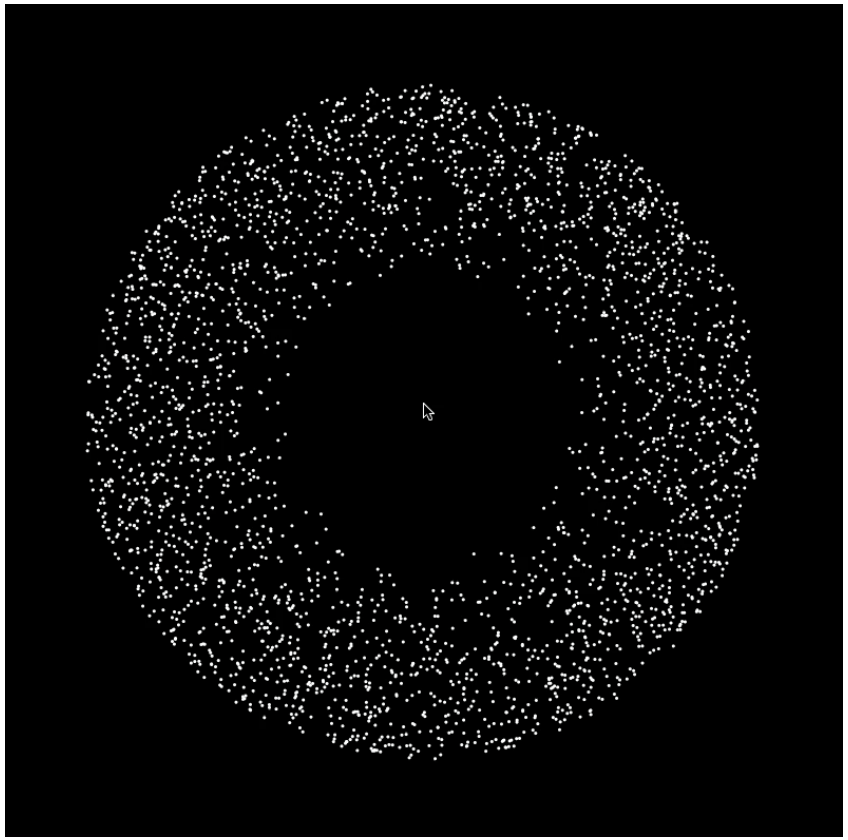
```
@ti.kernel
def foo():
    for i in x:
        do_something(x[i])

    for i in y:
        do_something(y[i])
```

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
ti.root.dense(ti.i, 8).place(x)
ti.root.dense(ti.i, 8).place(y)
```

```
@ti.kernel
def foo():
    for i in x:
        do_something(x[i])

    for i in y:
        do_something(y[i])
```

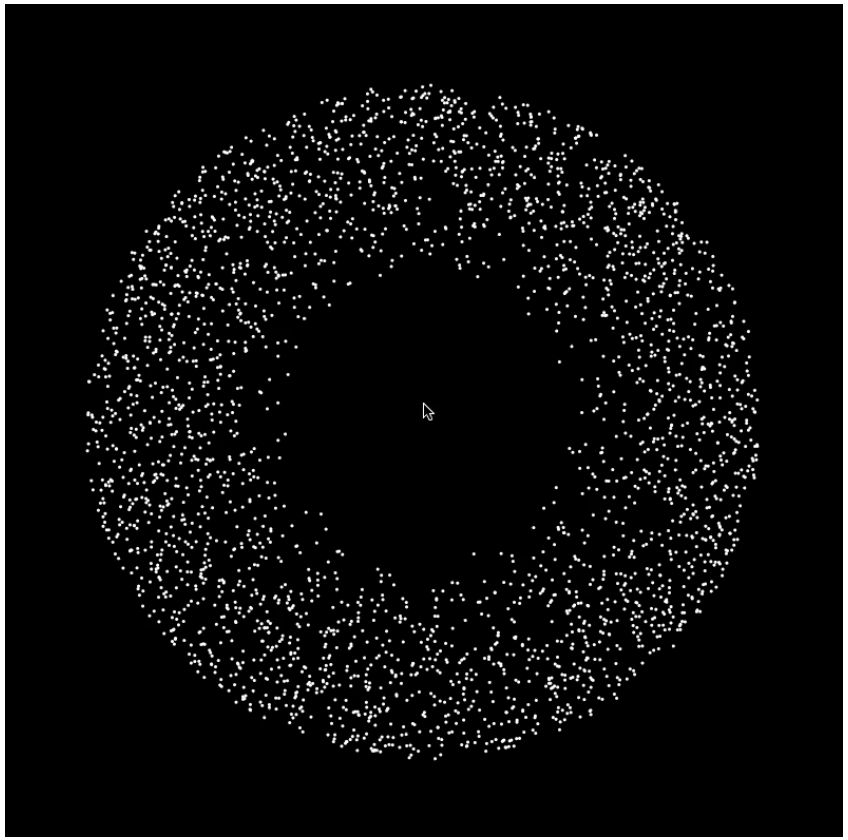
SoA Example, N-body:

```
pos = ti.Vector.field(2, ti.f32, N)
vel = ti.Vector.field(2, ti.f32, N)
force = ti.Vector.field(2, ti.f32, N)

...

@ti.kernel
def update():
    dt = h/substepping
    for i in range(N):
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]
```



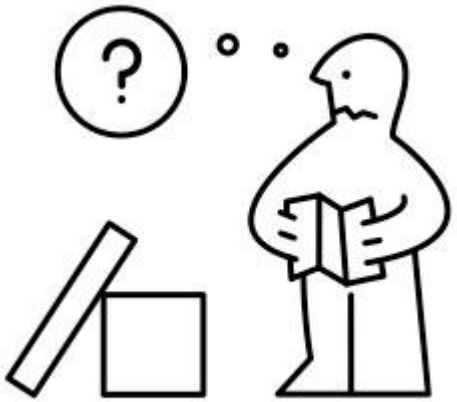
AoS Example, N-body:

```
pos = ti.Vector.field(2, ti.f32)
vel = ti.Vector.field(2, ti.f32)
force = ti.Vector.field(2, ti.f32)
ti.root.dense(ti.i, N).place(pos, vel, force)

...

@ti.kernel
def update():
    dt = h/substepping
    for i in range(N):
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]
```



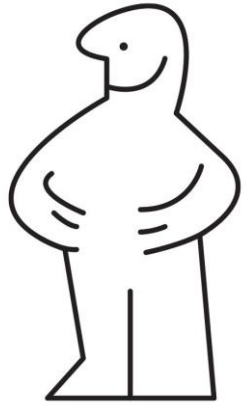
Loop over advanced data layouts

- Note



Loop over advanced data layouts

- Note = None
  - You can access your advanced data layouts using struct-for(s) as if they were your old friend *ti.field()* defined with *shape*.



Before moving to the next topic…

- Generates advanced dense data layouts using *ti.root*
  - Tree-structured: *SNode-trees.*
    - The SNode stands for “Structural Nodes”
  - All fields in Taichi are built using SNode-trees
    - ti.root is actuall “the root” of an SNode-tree
    - x=ti.field(ti.f32, shape=N) <==> x=ti.field(ti.f32) + ti.root.dense(ti.i, N).place(x)
    - ti.root.dense(ti.ij, (N, M)) <==> ti.root.dense(ti.i, N).dense(ti.j, M)
  - You can append (multiple) dense cells to other dense cells
    - Row/col-major: ti.root.dense(ti.i, N).dense(ti.j, M)
    - Hierarchical layouts: ti.root.dense(ti.i, N).dense(ti.i, M)
    - SoA/AoS: ti.root.dense(ti.i, N).place(x, y, z)
  - You do not need to worry about the access of your data layouts
    - The Taichi struct-for handles it for you

Before moving to the next topic…

- Generates advanced dense data layouts using *ti.root*
  - Tree-structured: *SNode-trees.*
    - The SNode stands for “Structural Nodes”
  - All fields in Taichi are built using SNode-trees
    - ti.root is actuall “the root” of an SNode-tree
    - x=ti.field(ti.f32, shape=N) <==> x=ti.field(ti.f32) + ti.root.dense(ti.i, N).place(x)
    - ti.root.dense(ti.ij, (N, M)) <==> ti.root.dense(ti.i, N).dense(ti.j, M)
  - You can append (multiple) dense cells to other dense cells
    - Row/col-major: ti.root.dense(ti.i, N).dense(ti.j, M)
    - Hierarchical layouts: ti.root.dense(ti.i, N).dense(ti.i, M)
    - SoA/AoS: ti.root.dense(ti.i, N).place(x, y, z)
  - You do not need to worry about the access of your data layouts
    - The Taichi struct-for handles it for you

Before moving to the next topic…

- Generates advanced dense data layouts using *ti.root*
  - Tree-structured: *SNode-trees.*
    - The SNode stands for “Structural Nodes”
  - All fields in Taichi are built using SNode-trees
    - ti.root is actuall “the root” of an SNode-tree
    - x=ti.field(ti.f32, shape=N) <==> x=ti.field(ti.f32) + ti.root.dense(ti.i, N).place(x)
    - ti.root.dense(ti.ij, (N, M)) <==> ti.root.dense(ti.i, N).dense(ti.j, M)
  - You can append (multiple) dense cells to other dense cells
    - Row/col-major: ti.root.dense(ti.i, N).dense(ti.j, M)
    - Hierarchical layouts: ti.root.dense(ti.i, N).dense(ti.i, M)
    - SoA/AoS: ti.root.dense(ti.i, N).place(x, y, z)
  - You do not need to worry about the access of your data layouts
    - The Taichi struct-for handles it for you

Before moving to the next topic…

- Generates advanced dense data layouts using *ti.root*
  - Tree-structured: *SNode-trees.*
    - The SNode stands for “Structural Nodes”
  - All fields in Taichi are built using SNode-trees
    - ti.root is actuall “the root” of an SNode-tree
    - x=ti.field(ti.f32, shape=N) <==> x=ti.field(ti.f32) + ti.root.dense(ti.i, N).place(x)
    - ti.root.dense(ti.ij, (N, M)) <==> ti.root.dense(ti.i, N).dense(ti.j, M)
  - You can append (multiple) dense cells to other dense cells
    - Row/col-major: ti.root.dense(ti.i, N).dense(ti.j, M)
    - Hierarchical layouts: ti.root.dense(ti.i, N).dense(ti.i, M)
    - SoA/AoS: ti.root.dense(ti.i, N).place(x, y, z)
  - You do not need to worry about the access of your data layouts
    - The Taichi struct-for handles it for you

Before moving to the next topic…

- Generates advanced dense data layouts using *ti.root*
  - ti.i, ti.j, ti.k, ti.l <==> ti.axes(0), ti.axes(1), ti.axes(2), ti.axes(3)
    - Currently Taichi supports at most 8 axes to ti.axes(7)
      - ti.root.dense(ti.axes(0), 1).dense(ti.axes(1), 2).dense(ti.axes(2), 3).dense(ti.axes(3), 4).dense(ti.axes(4), 5).dense(ti.axes(5), 6).dense(ti.axes(6), 7).dense(ti.axes(7), 8).place(x)
  - Get your Taichi updated to get the correct behavior for row/col-majored fields
    - We have a new release today (10/12/2021)

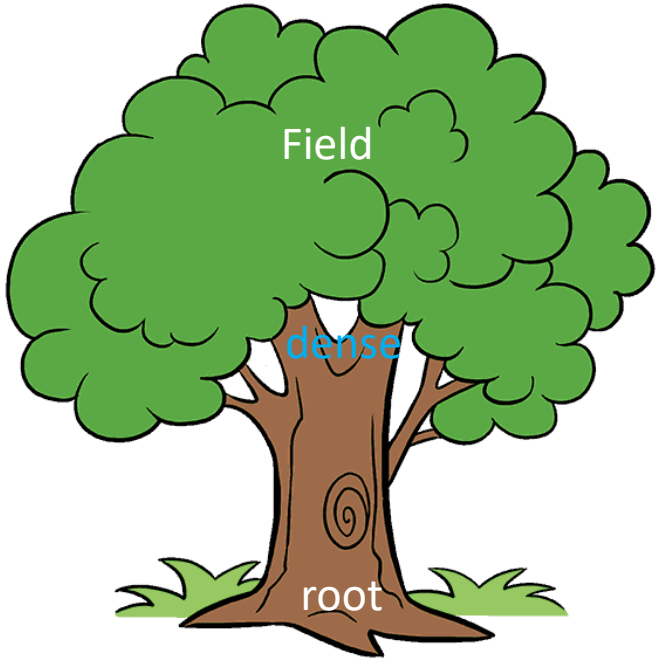
Before moving to the next topic…

- Generates advanced dense data layouts using *ti.root*
  - ti.i, ti.j, ti.k, ti.l <==> ti.axes(0), ti.axes(1), ti.axes(2), ti.axes(3)
    - Currently Taichi supports at most 8 axes to ti.axes(7)
      - ti.root.dense(ti.axes(0), 1).dense(ti.axes(1), 2).dense(ti.axes(2), 3).dense(ti.axes(3), 4).dense(ti.axes(4), 5).dense(ti.axes(5), 6).dense(ti.axes(6), 7).dense(ti.axes(7), 8).place(x)
  - Get your Taichi updated to get the correct behavior for row/col-majored fields
    - We have a new release (ver. 0.8.3) today (10/12/2021)

Sparse data layouts

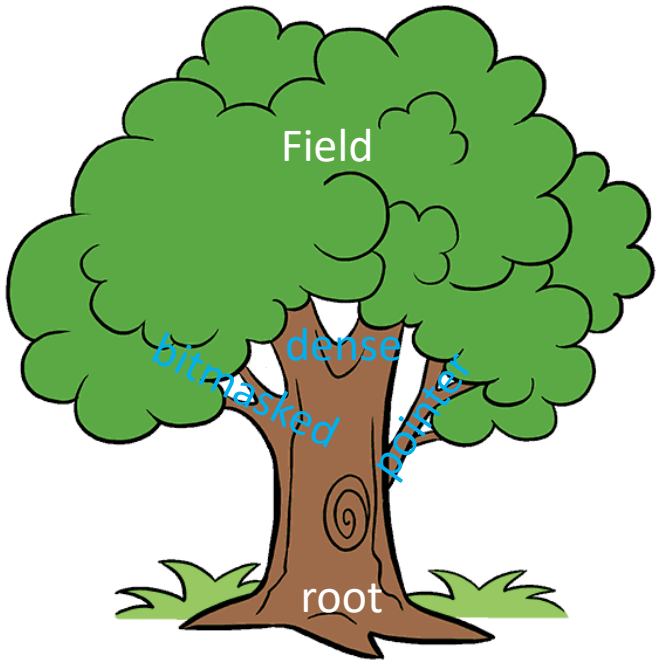
The SNode-tree

- root: the root of the data structure
- dense: a fixed-length contiguous array.



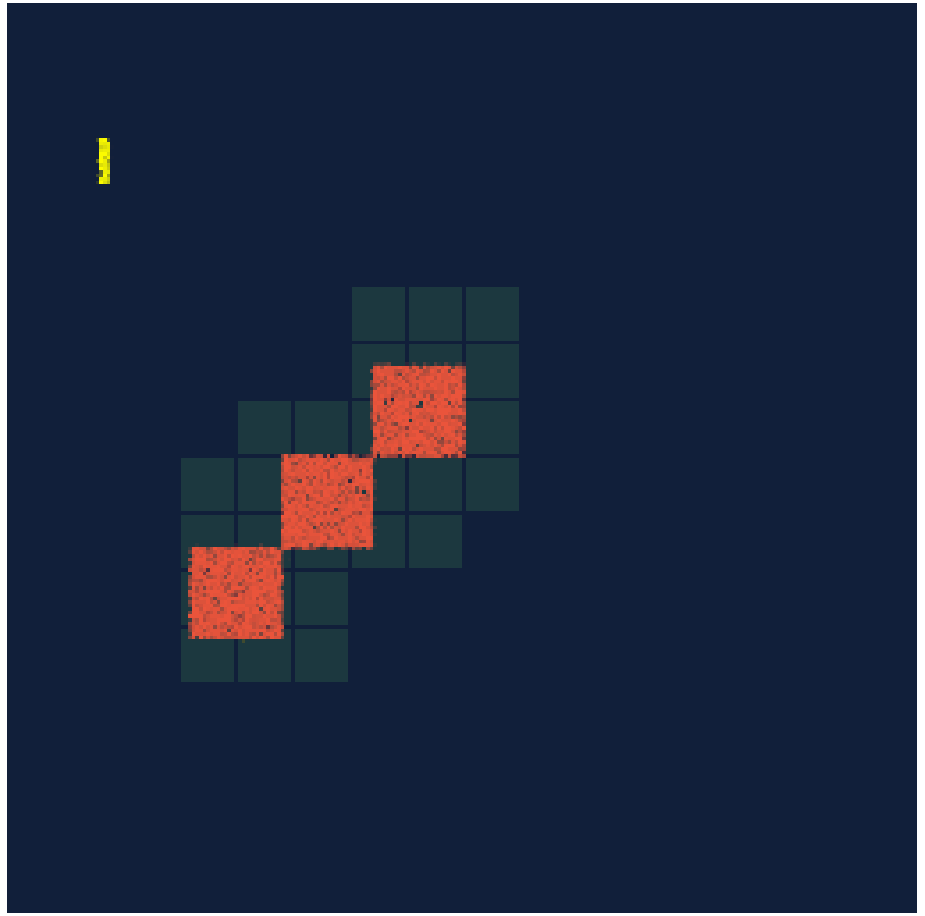
The SNode-tree

- root: the root of the data structure
- dense: a fixed-length contiguous array.
- bitmasked: similar to dense, but it also uses a mask to maintain ***sparsity*** information, one bit per child.
- pointer: stores pointers instead of the whole structure to save memory and maintain ***sparsity***



Sparse computation! but why?

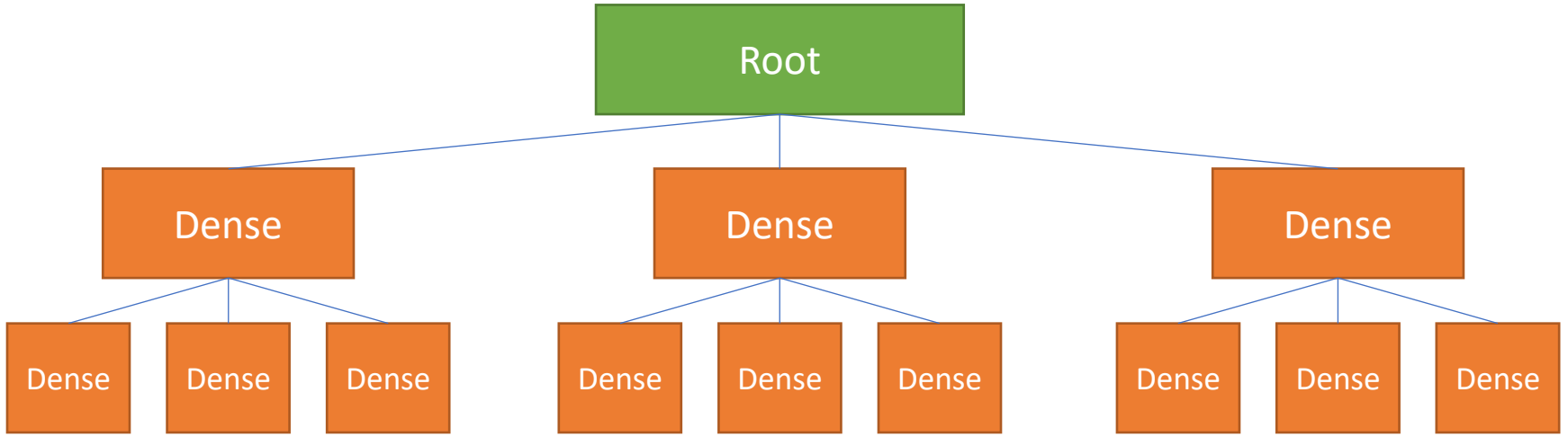
- MPM simulation ➔➔
  - 256x256 grid cells in total
  - Subdivided to 16x16 blocks
    - Each block has 16x16 grid cells
  - Allocating memory for the total 256x256 grid cells is a waste.
    - The dark blocks are filled with zeros anyway



Sparse computation! Then how?

- A dense SNode-tree:

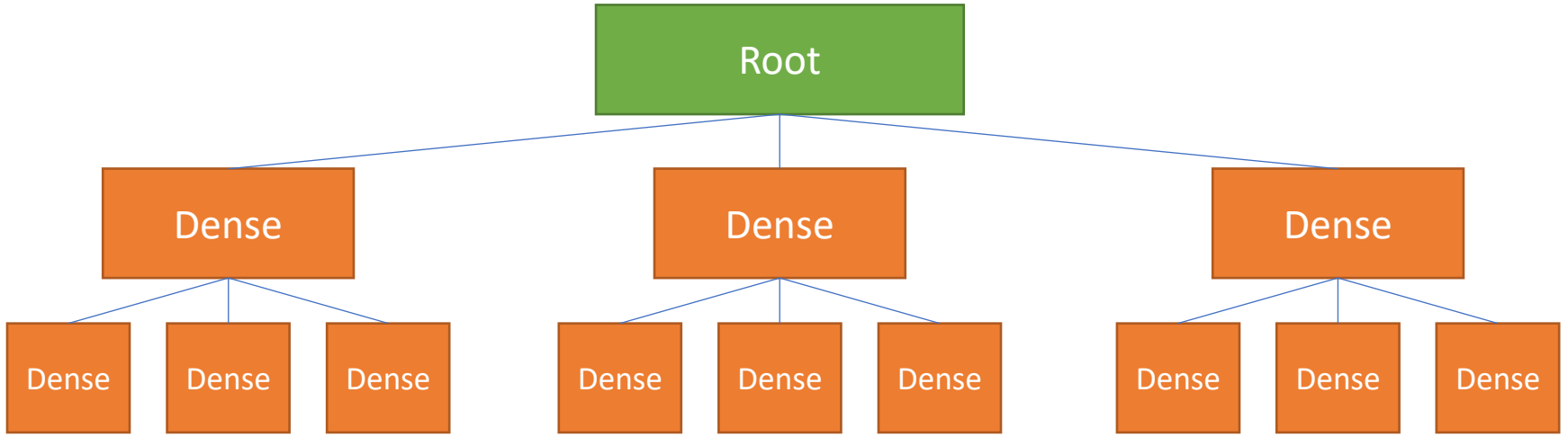
```
x = ti.field(ti.i32)
block1 = ti.root.dense(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.dense(ti.i,3)
.dense(ti.j,3).place(x)
```



Sparse computation! Then how?

- A dense SNode-tree:

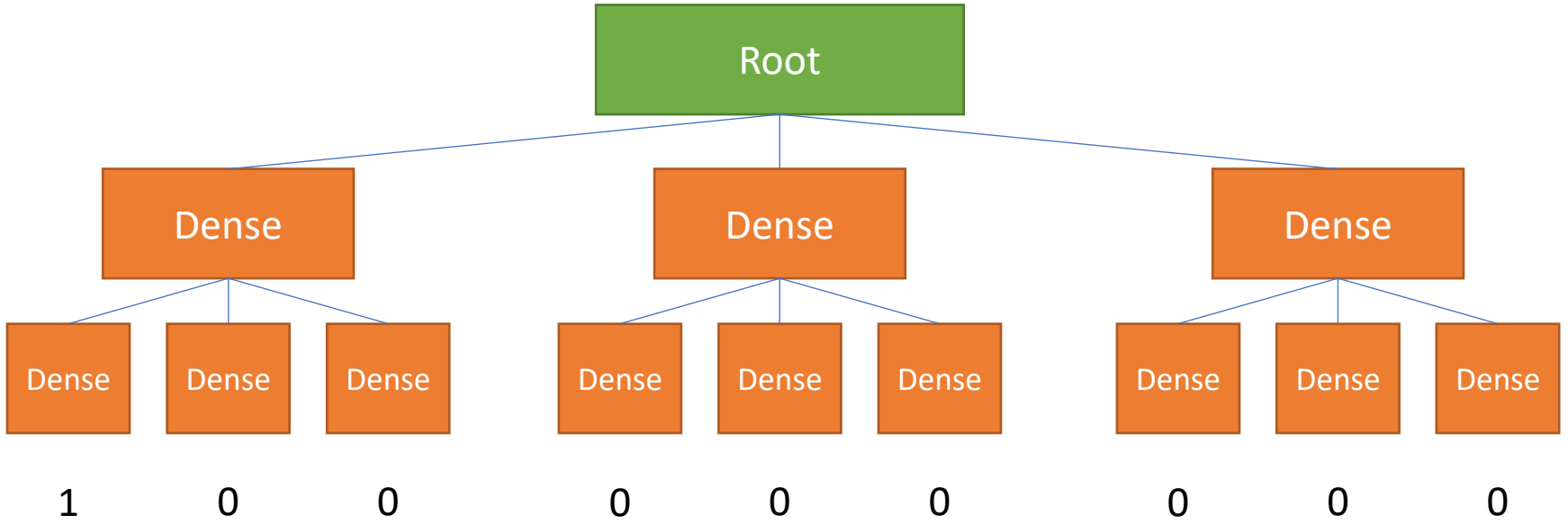
1	0	0
0	0	0
0	0	0



Sparse computation! Then how?

- A dense SNode-tree:

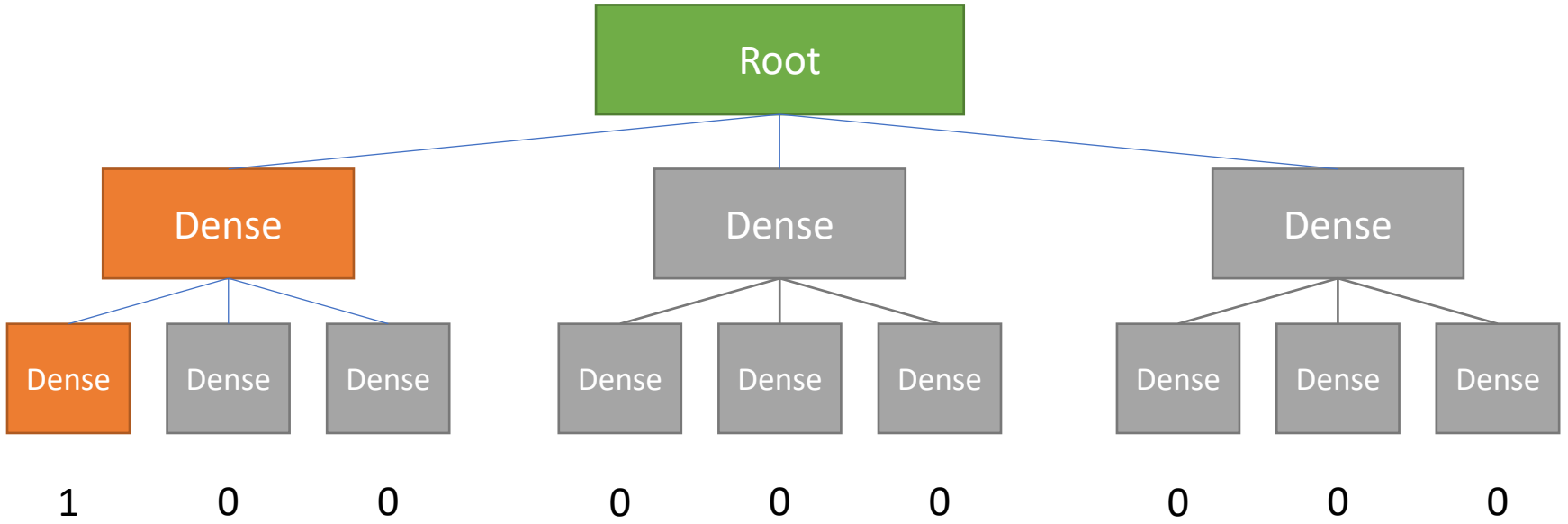
1	0	0
0	0	0
0	0	0



Sparse computation! Then how?

- A dense SNode-tree:

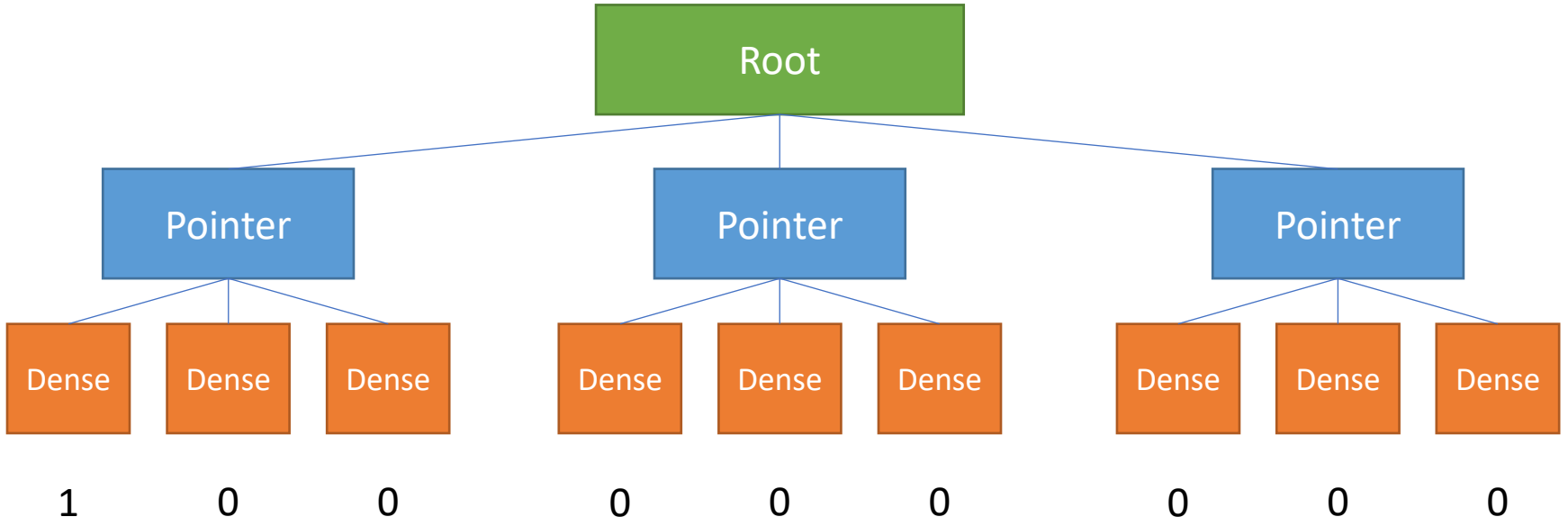
1	0	0
0	0	0
0	0	0



From .dense() to .pointer()

- A sparse SNode-tree:

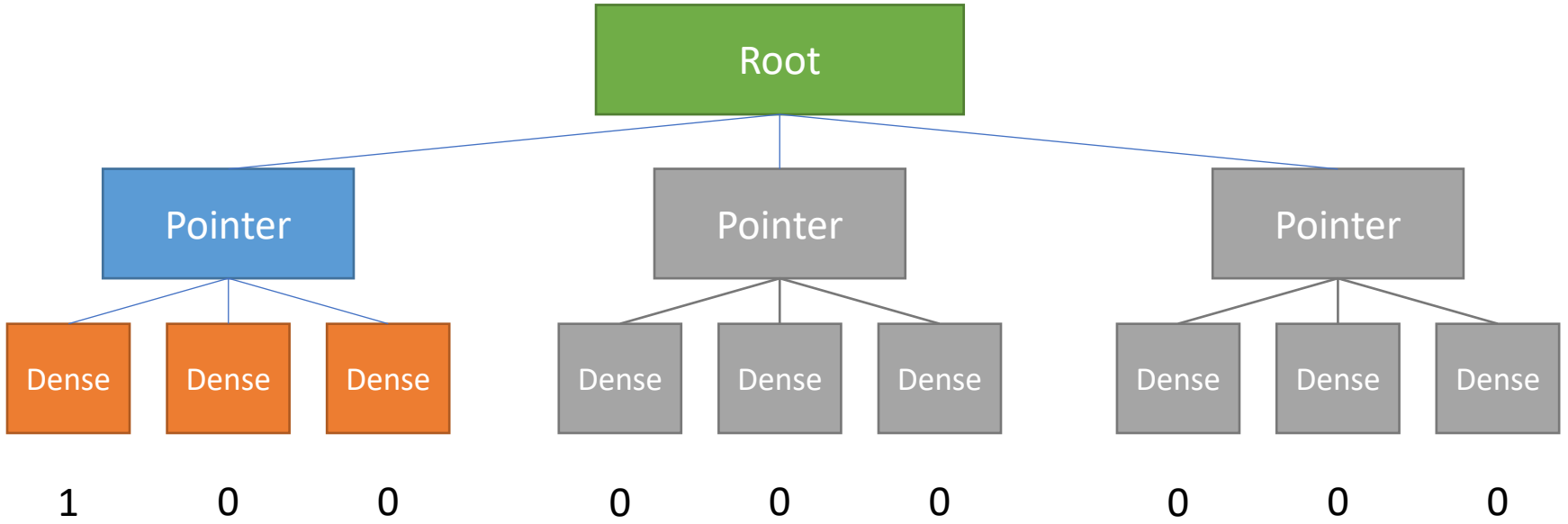
1	0	0
0	0	0
0	0	0



From .dense() to .pointer()

- A sparse SNode-tree:

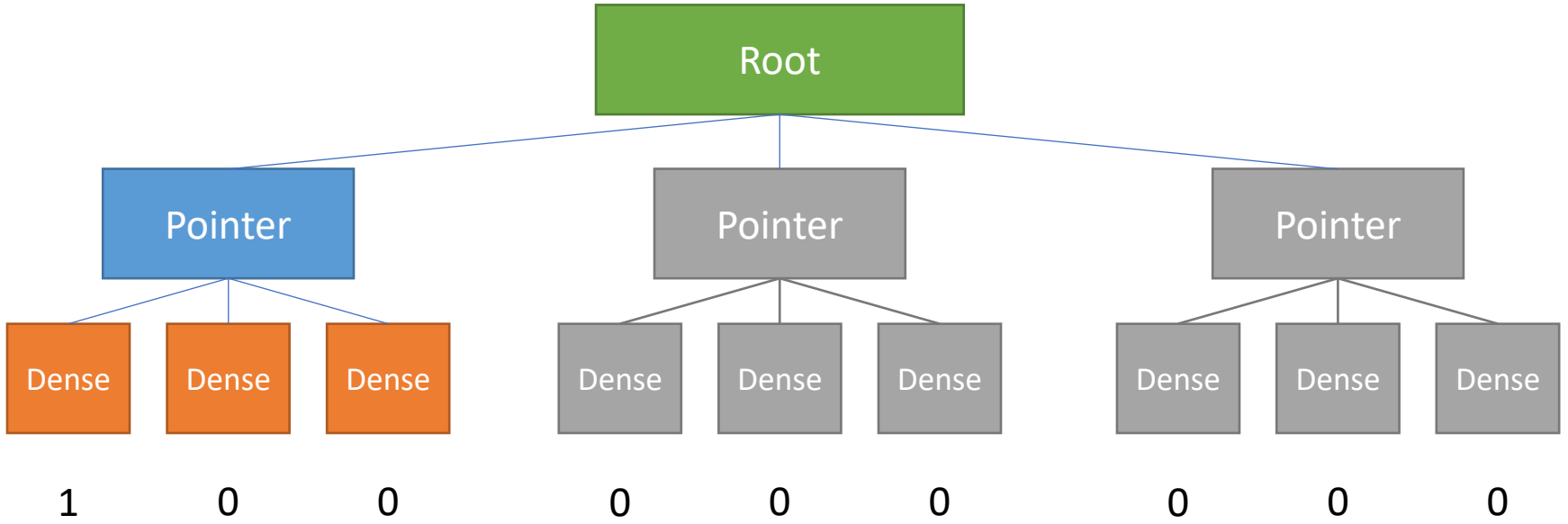
1	0	0
0	0	0
0	0	0



From .dense() to .pointer()

- A sparse SNode-tree:

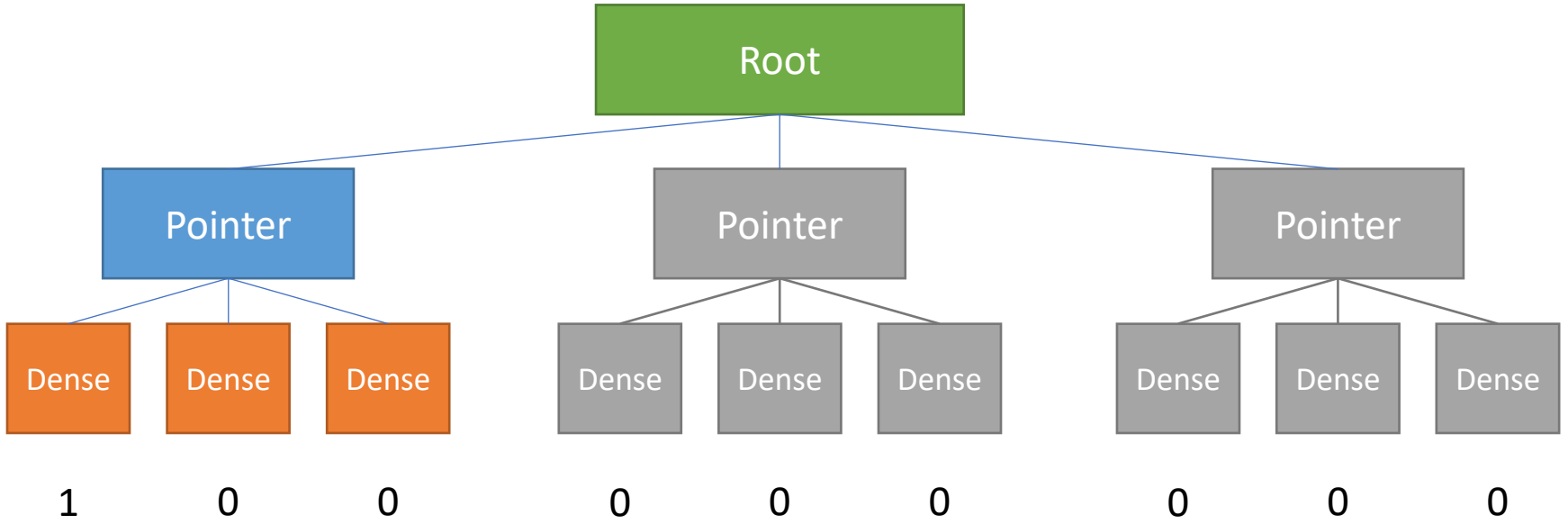
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```



From .dense() to .pointer()

- A sparse SNode-tree:

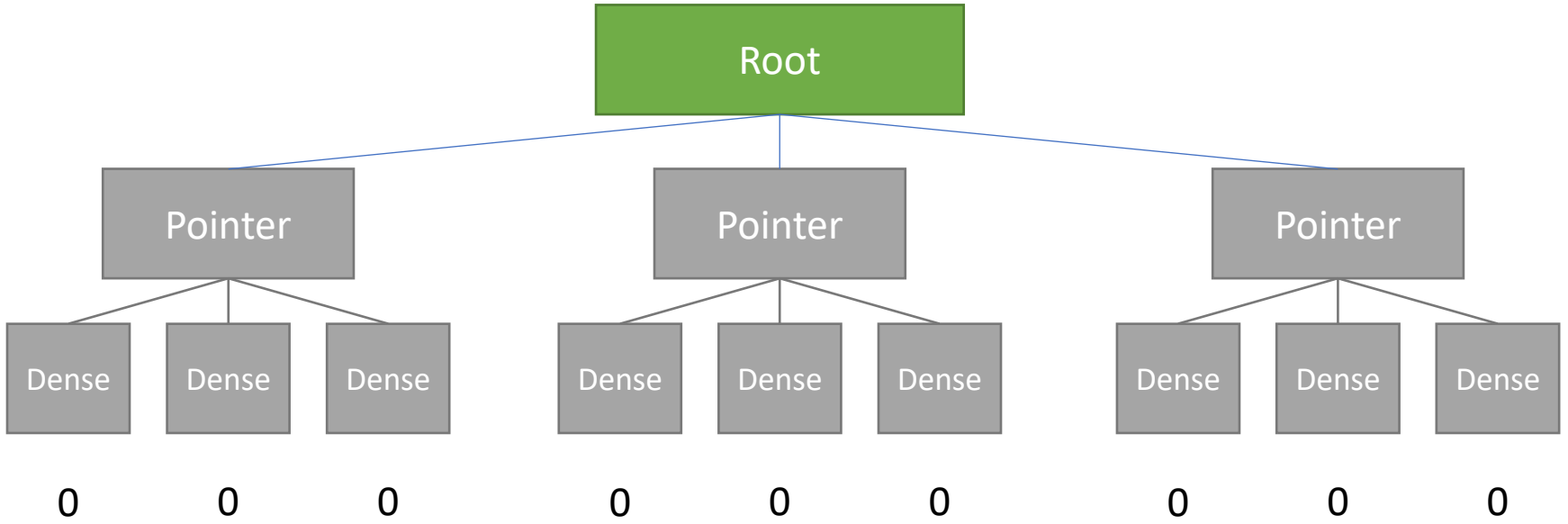
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```



Activation

- A sparse SNode-tree born empty:

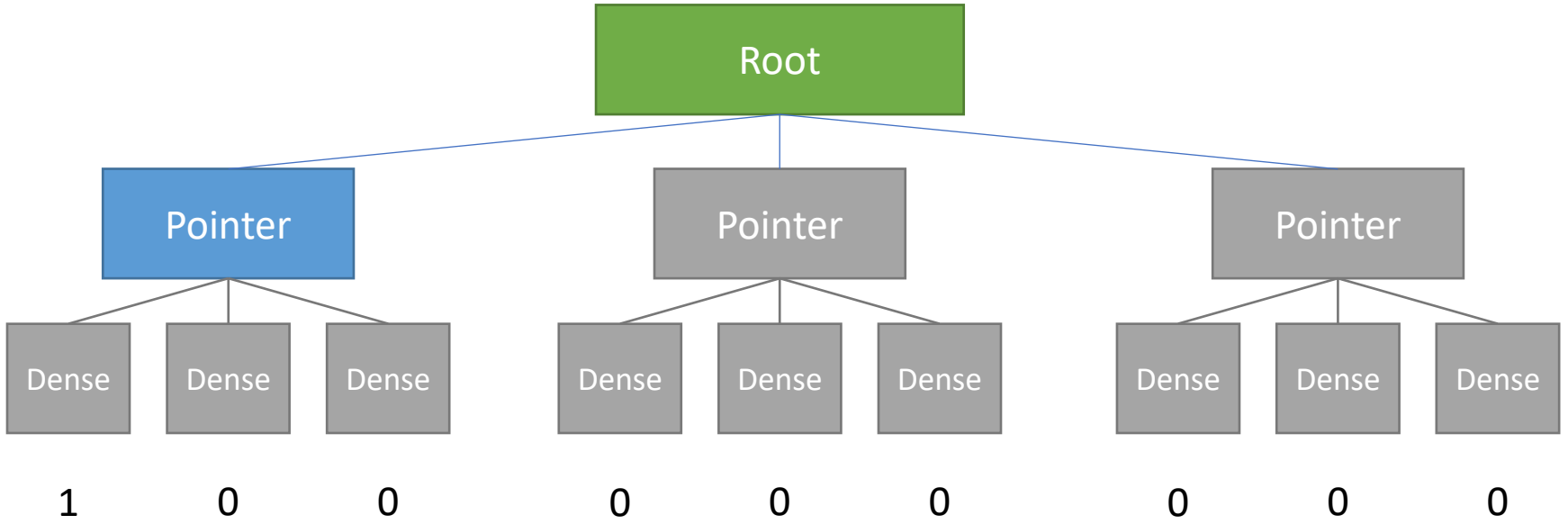
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```



Activation

- Once ***writing*** an inactive cell:

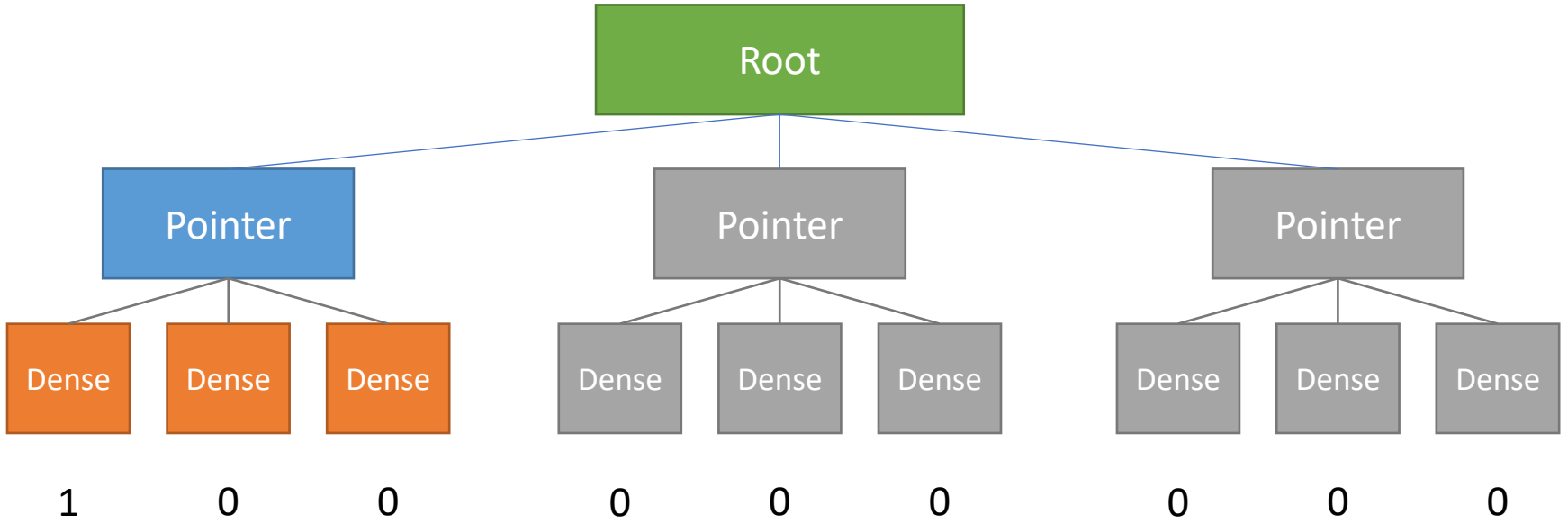
```
x[0,0] = 1
# activates block1[0]
```



Activation

- Once ***writing*** an inactive cell:

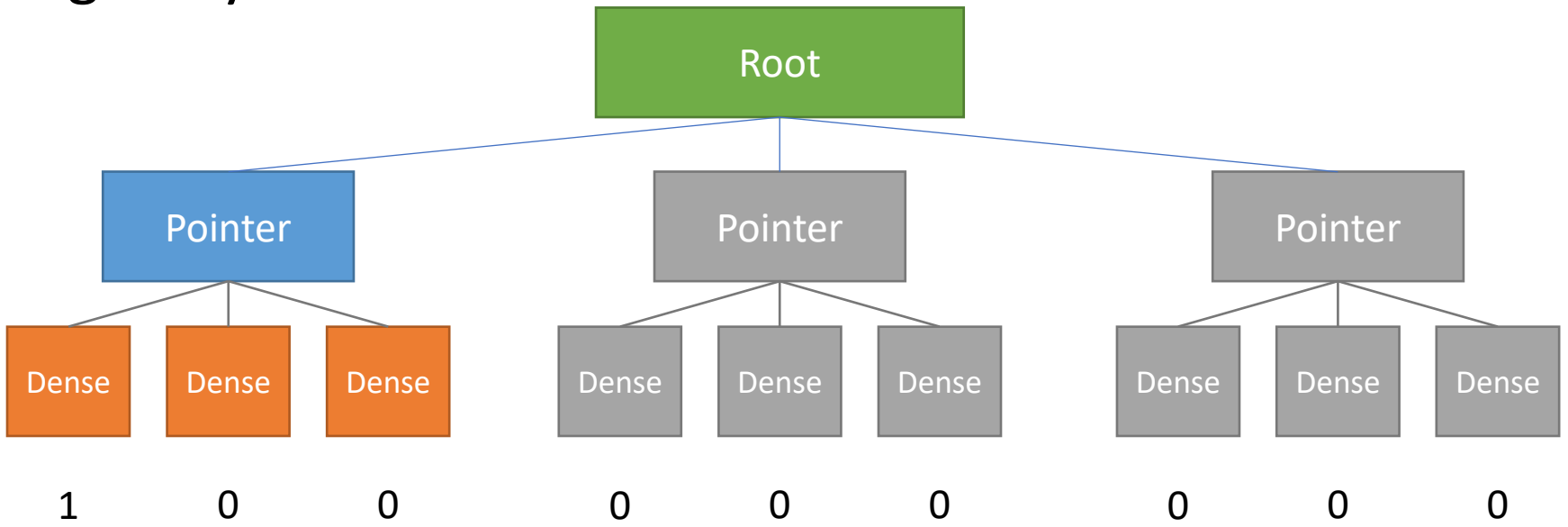
```
x[0,0] = 1
# activates block1[0] and thereby block2[0],
 block2[1] and block2[2]
```



Data access in a sparse field (a sparse SNode-tree)

- Use Taichi struct-for to access a sparse field
  - Inactive pointers are skipped
- Manually accessing inactive data gives you a zero

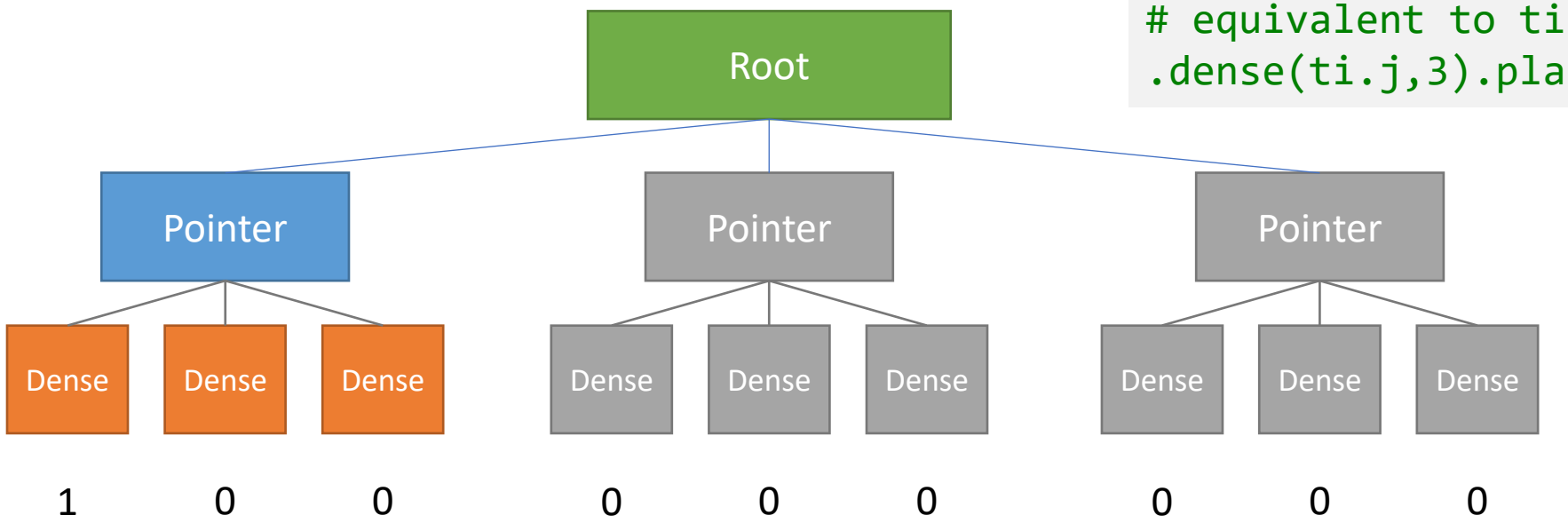
```
@ti.kernel
def access_all():
    for i,j in x:
        print(x[i, j]) # 1, 0, 0

print(x[2, 2]) # 0
```



Why activating x[0, 1] and x[0, 2] as well?

- Because they belong to the same dense block

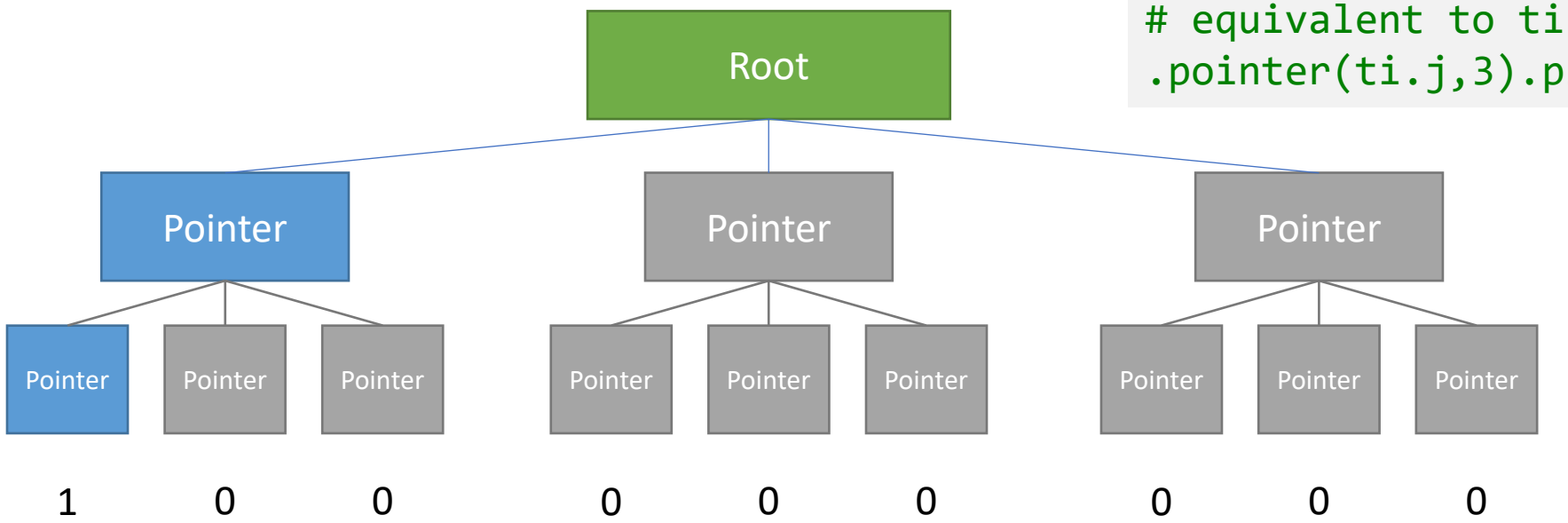
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```



Why not using pointer everywhere?

- Bad design idea:
  - a ti.f32 → 32 bits
  - a taichi pointer → 64 bits

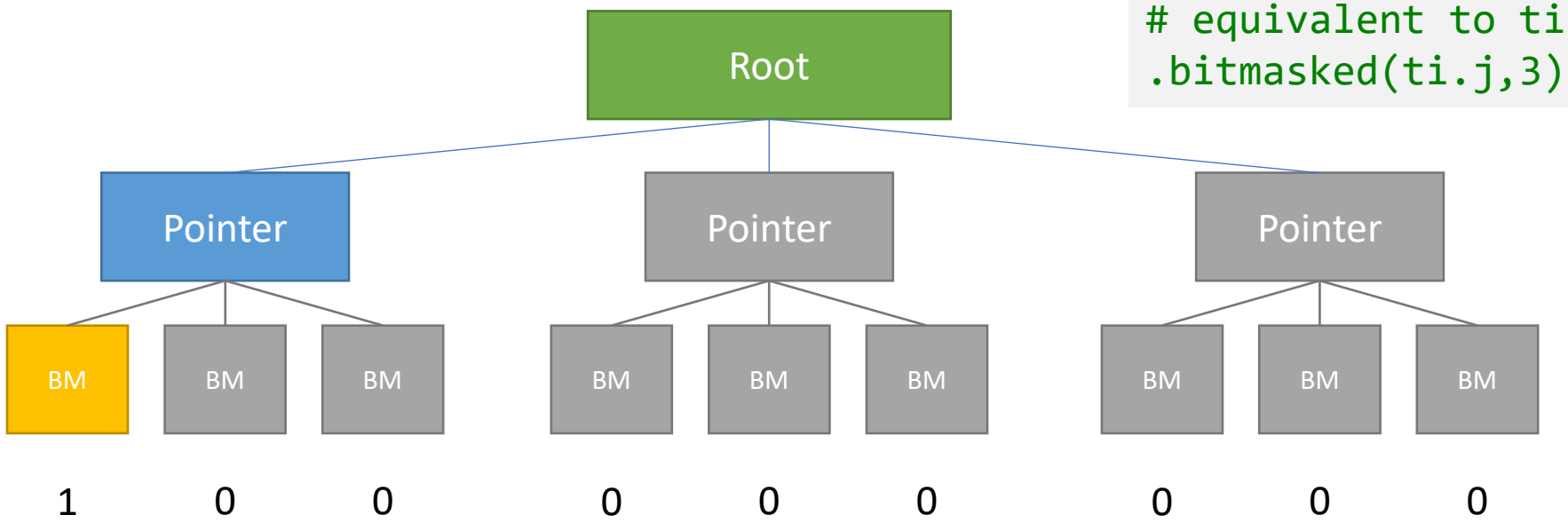
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.pointer(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.pointer(ti.j,3).place(x)
```



Use *bitmasks* if you really want to flag leaf cells one at a time…

- Works for leaf cells only
- Each leaf cell has its own activation flag

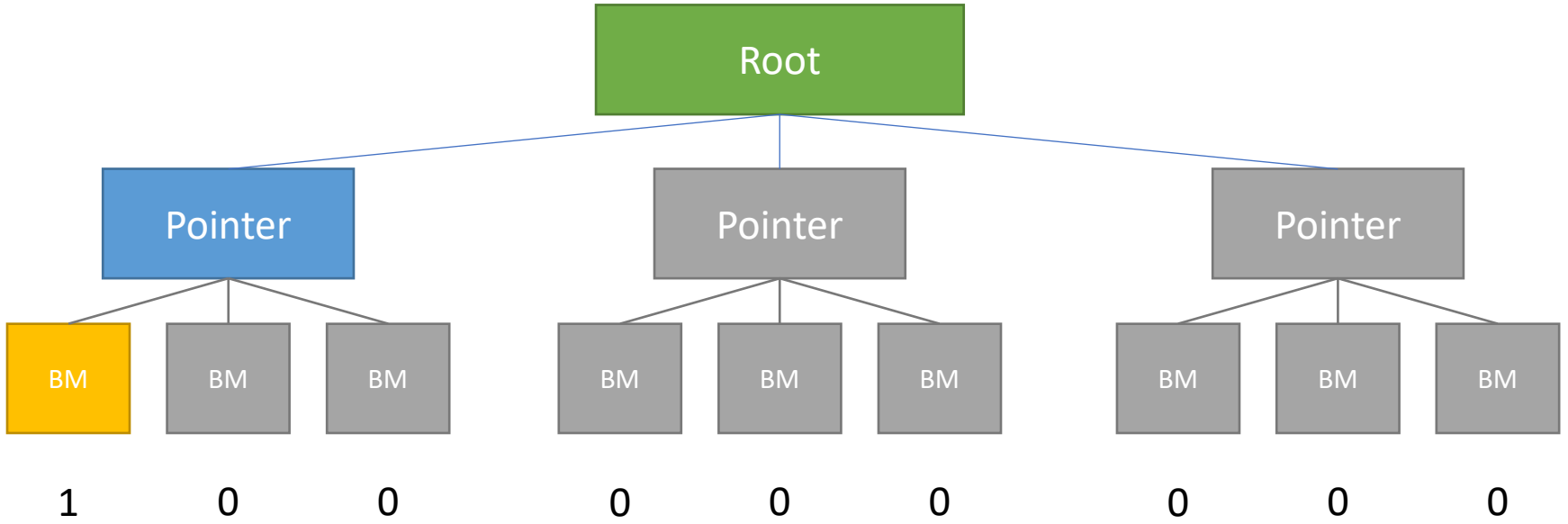
```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.bitmasked(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.bitmasked(ti.j,3).place(x)
```



Use *bitmasks* if you really want to flag leaf cells one at a time…

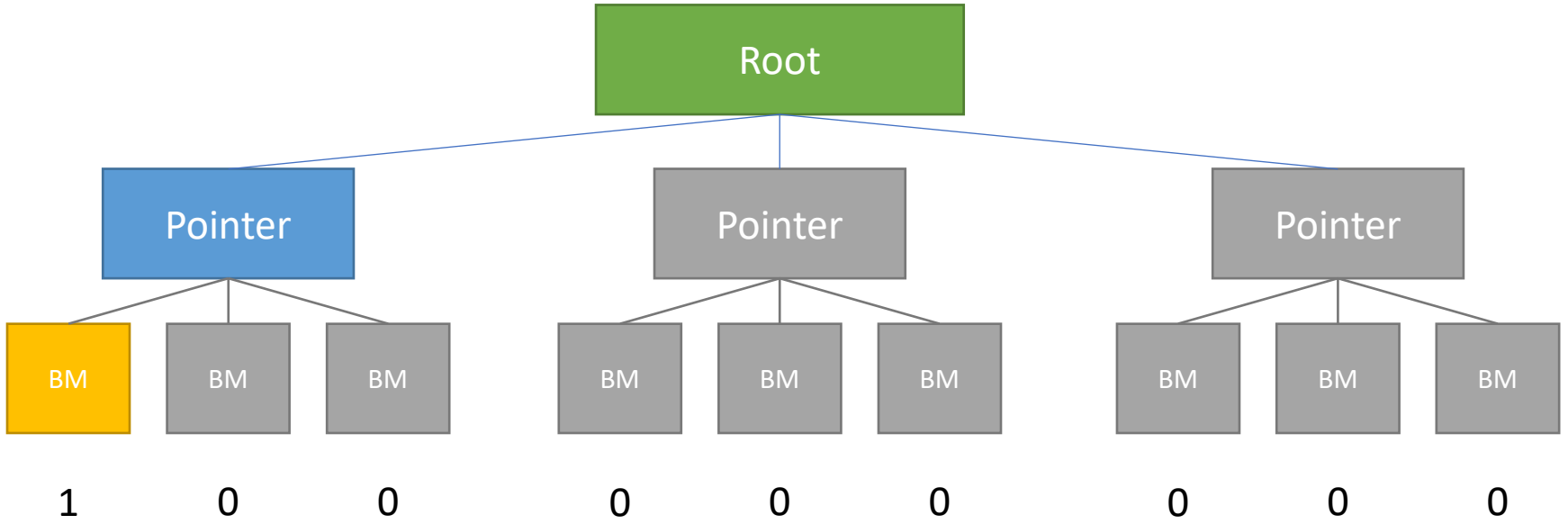
- Works for leaf cells only
- Each leaf cell has its own activation flag

```
@ti.kernel
def access_all():
    for i,j in x:
        print(x[i, j]) # 1
```



Use *bitmasks* if you really want to flag leaf cells one at a time…

- Cost 1-bit-per-cell extra
- Skip struct-for(s) when bitmasked inactive



Manual sparse field manipulation

- API
  - Check activation status:
    - ti.is_active(snode, [i,j,…])
      - for example: ti.is_active(block1, [0]) #=True
  - Activate/deactivate cells:
    - ti.activate/deactivate(snode, [i,j])
  - Deactivate a cell and its children:
    - snode.deactivate_all()
  - Compute the index of ancestor
    - ti.rescale_index(snode/field, ancestor_snode, index)
      - for example: ti.rescale_index(block2, block1, [4]) #=1

```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```

Manual sparse field manipulation

- API
  - Check activation status:
    - ti.is_active(snode, [i,j,...])
      - for example: ti.is_active(block1, [0]) #=True
  - Activate/deactivate cells:
    - ti.activate/deactivate(snode, [i,j])
  - Deactivate a cell and its children:
    - snode.deactivate_all()
  - Compute the index of ancestor
    - ti.rescale_index(snode/field, ancestor_snode, index)
      - for example: ti.rescale_index(block2, block1, [4]) #=1

```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```

Manual sparse field manipulation

- API
  - Check activation status:
    - ti.is_active(snode, [i,j,...])
      - for example: ti.is_active(block1, [0]) #=True
  - Activate/deactivate cells:
    - ti.activate/deactivate(snode, [i,j])
  - Deactivate a cell and its children:
    - snode.deactivate_all()
  - Compute the index of ancestor
    - ti.rescale_index(snode/field, ancestor_snode, index)
      - for example: ti.rescale_index(block2, block1, [4]) #=1

```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```

Manual sparse field manipulation

- API
  - Check activation status:
    - ti.is_active(snode, [i,j,...])
      - for example: ti.is_active(block1, [0]) #=True
  - Activate/deactivate cells:
    - ti.activate/deactivate(snode, [i,j])
  - Deactivate a cell and its children:
    - snode.deactivate_all()
  - Compute the index of ancestor
    - ti.rescale_index(snode/field, ancestor_snode, index)
      - for example: ti.rescale_index(block2, block1, [4]) #=1
        - Do not use 4//3 to compute the index of ancestor

```
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.pointer(ti.i,3)
.dense(ti.j,3).place(x)
```

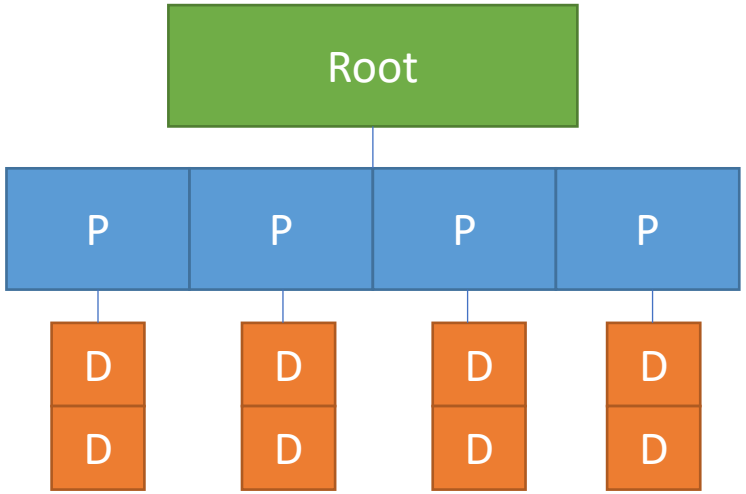
Putting things together

- Previous section:
  - Row-major v.s. col-major, flat v.s. hierarchical layouts
- This section:
  - .dense() v.s. .pointer()/.bitmasked()

Putting things together

- A column-majored 2x4 2D sparse field:

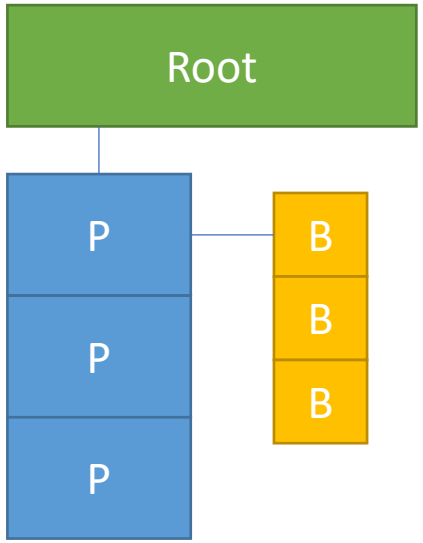
```
x = ti.field(ti.i32)
ti.root.pointer(ti.j,4).dense(ti.i,2).place(x)
```



Putting things together

- A block-majored (block size = 3) 9x1 1D sparse field:

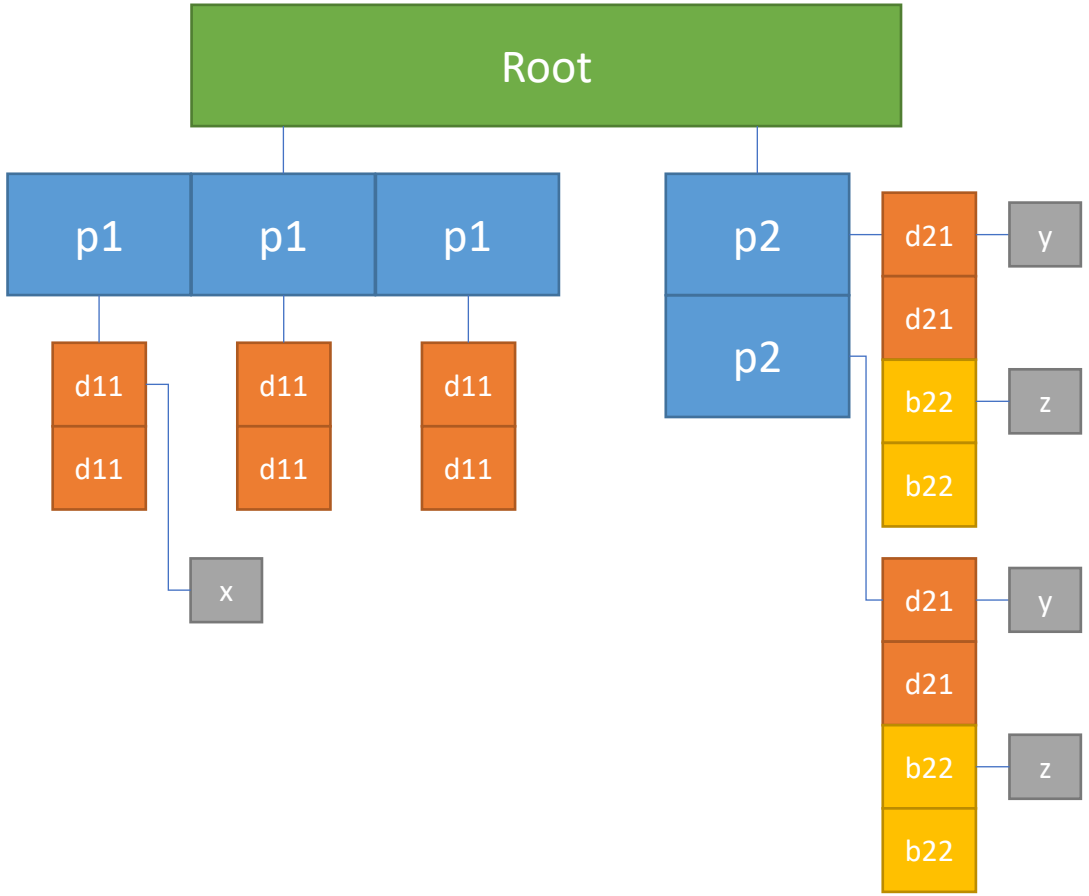
```
x = ti.field(ti.i32)
ti.root.pointer(ti.i,3).bitmasked(ti.i,3).place(x)
```



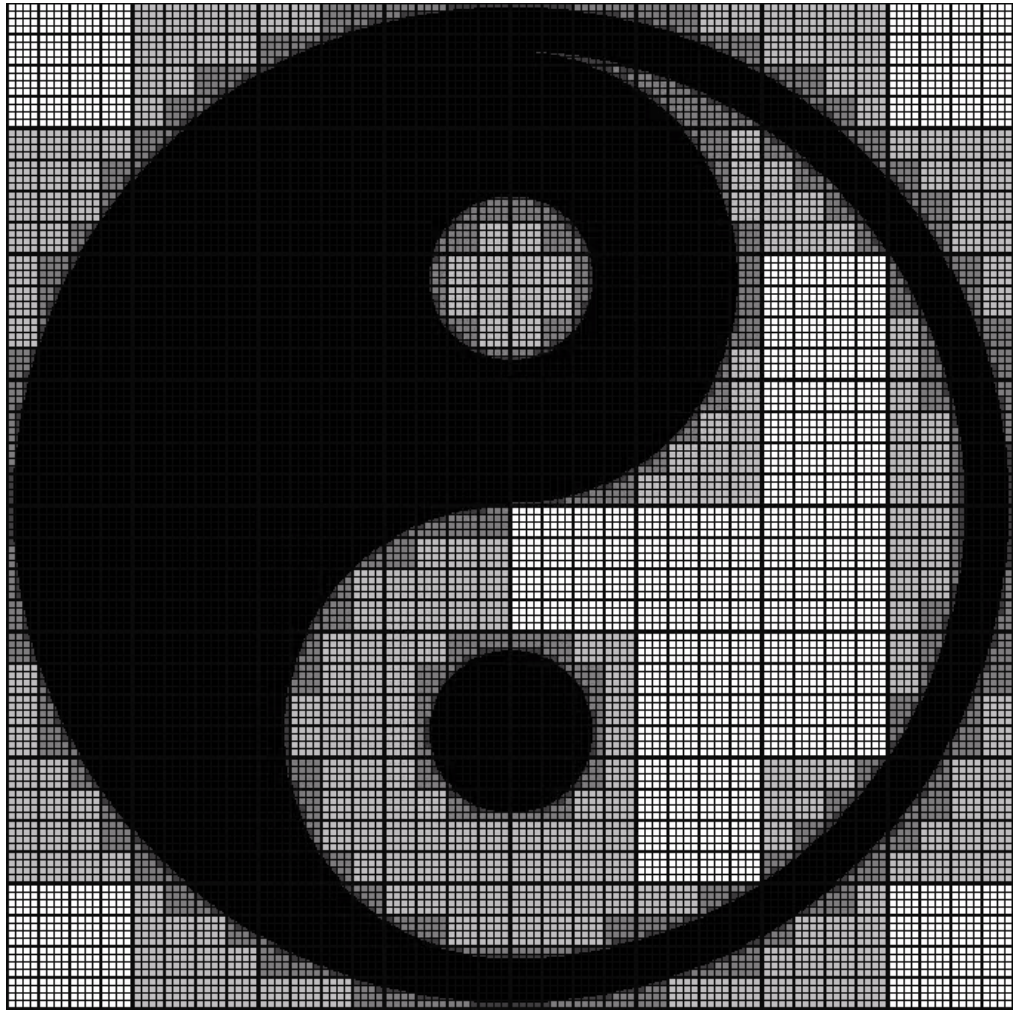
Putting things together

- I wrote this because I could:
  - x: A column-majored 2x3 2D sparse field
  - y/z: block-majored sparse 4x1 1D sparse fields
  - y and z share the same sparsity pattern on p2

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
z = ti.field(ti.i32)
p1 = ti.root.pointer(ti.j,3)
p2 = ti.root.pointer(ti.i,2)
d11 = p1.dense(ti.i, 2)
d21 = p2.dense(ti.i, 2)
b22 = p2.bitmasked(ti.i, 2)
d11.place(x)
d21.place(y)
b22.place(z)
```



A rolling Taichi [Code]



```
n = 512
x = ti.field(ti.i32)

block1 = ti.root.pointer(ti.ij, n // 64)
block2 = block1.pointer(ti.ij, 4)
block3 = block2.pointer(ti.ij, 4)
block3.dense(ti.ij, 4).place(x)
```

The grid is divided into 8x8 *block1* containers;
Each *block1* container has 4x4 *block2* cells;
Each *block2* container has 4x4 *block3* cells;
Each *block3* container has 4x4 *pixel* cells;
Each *pixel* contains an i32 value x[i, j].

Sparse data layouts

- Append more types to your SNode-tree:
  - .pointer() to represent sparse cells
  - .bitmasked() to represent sparse leaf cells
- Activate cells (and its ancestors) by writing
  - x[0,0] = 1
- Use Taichi struct-for(s) to access sparse fields
  - as if they were dense ☺

Sparse data layouts

- Append more types to your SNode-tree:
  - .pointer() to represent sparse cells
  - .bitmasked() to represent sparse leaf cells
- Activate cells (and its ancestors) by writing
  - x[0,0] = 1
- Use Taichi struct-for(s) to access sparse fields
  - as if they were dense ☺

Sparse data layouts

- Append more types to your SNode-tree:
  - .pointer() to represent sparse cells
  - .bitmasked() to represent sparse leaf cells
- Activate cells (and its ancestors) by writing
  - x[0,0] = 1
- Use Taichi struct-for(s) to access sparse fields
  - as if they were dense ☺

Sparse data layouts

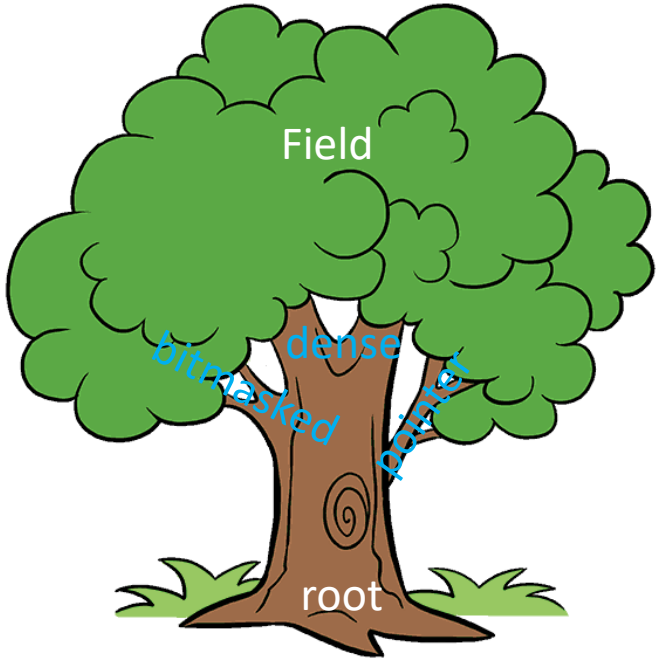
- Limited backend compatibility
  - Supported by CPU/CUDA/Metal backends
- Sparse matrices are usually NOT implemented in Taichi via sparse data layouts.
  - Will cover it next week

Remark

- Advanced Data layouts for
  - Dense data structures:
    - .dense()
    - row-major v.s. col-major, hierarchical v.s. flat, AoS v.s. SoA
  - Sparse data structures:
    - .pointer() / .bitmasked()

A bigger picture

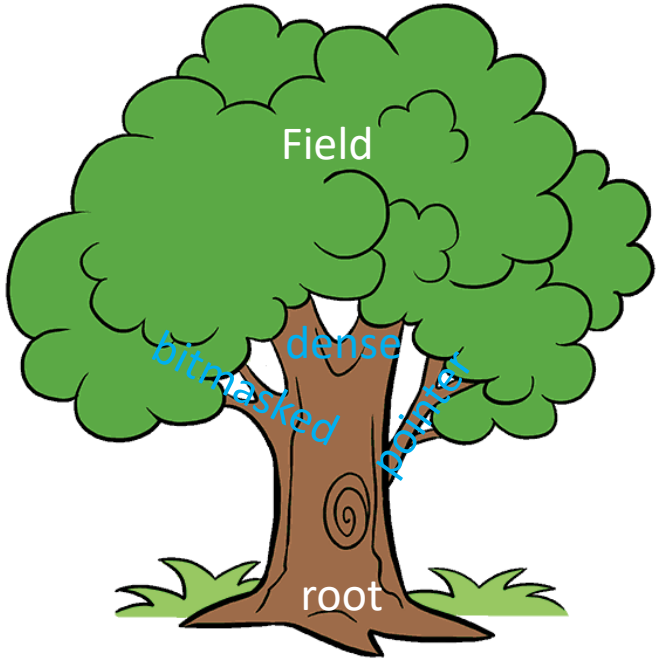
- The SNode-tree
  - root: the *root* of the data structure
  - dense: a *fixed-length* contiguous array
  - bitmasked: similar to dense, but it also uses a mask to maintain *sparsity* information, one bit per child
  - pointer: stores pointers instead of the whole structure to save memory and maintain *sparsity*



A bigger picture

- The SNode-tree
  - root: the *root* of the data structure
  - dense: a *fixed-length* contiguous array
  - bitmasked: similar to dense, but it also uses a mask to maintain *sparsity* information, one bit per child
  - pointer: stores pointers instead of the whole structure to save memory and maintain *sparsity*
  - dynamic: *variable-length* array, with a predefined maximum length

- Check Yuanming's paper for more details

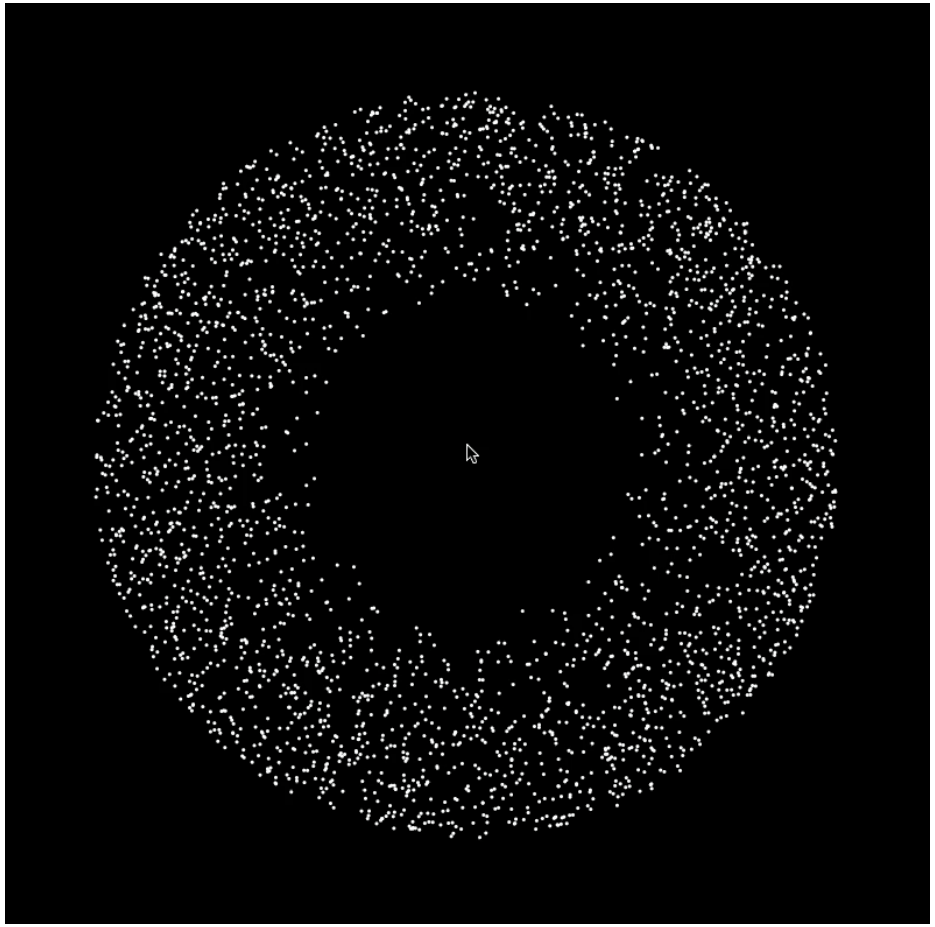


Taichi: a data-oriented programming language

- Focus on data-access
  - Faster data-access ≈ better performance in GPU
- Decouple the data-structures from computation
  - No need to change your code for trying different data layouts

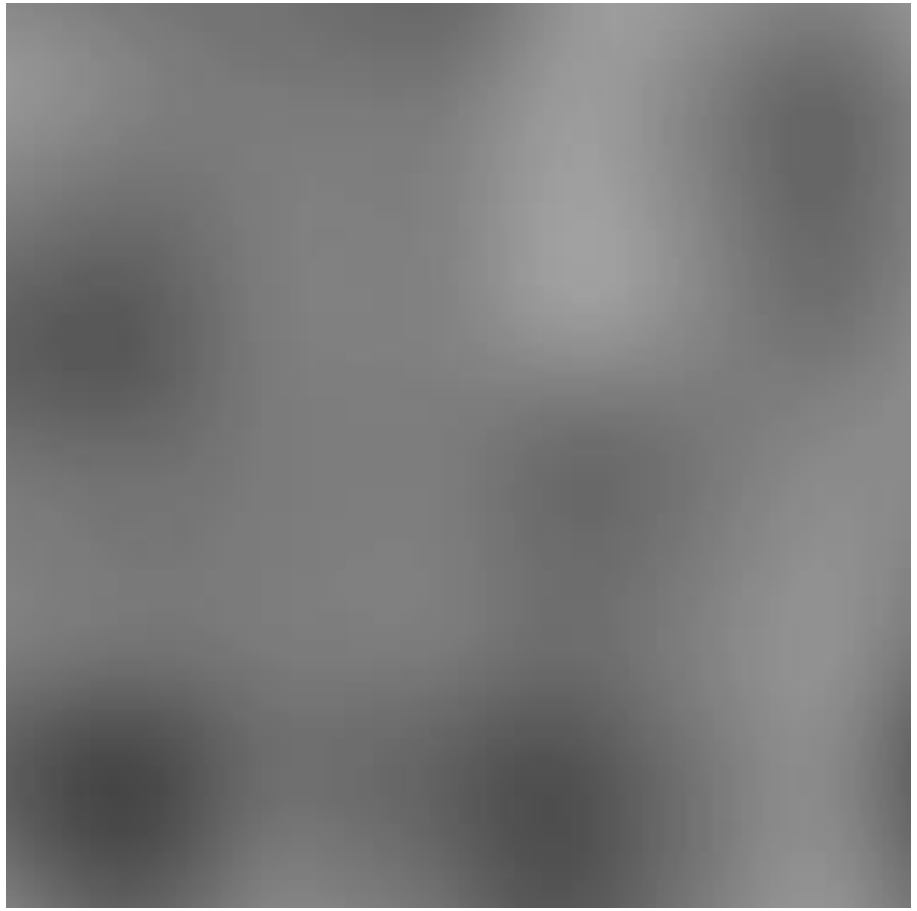
Homework

N-body: [Link]



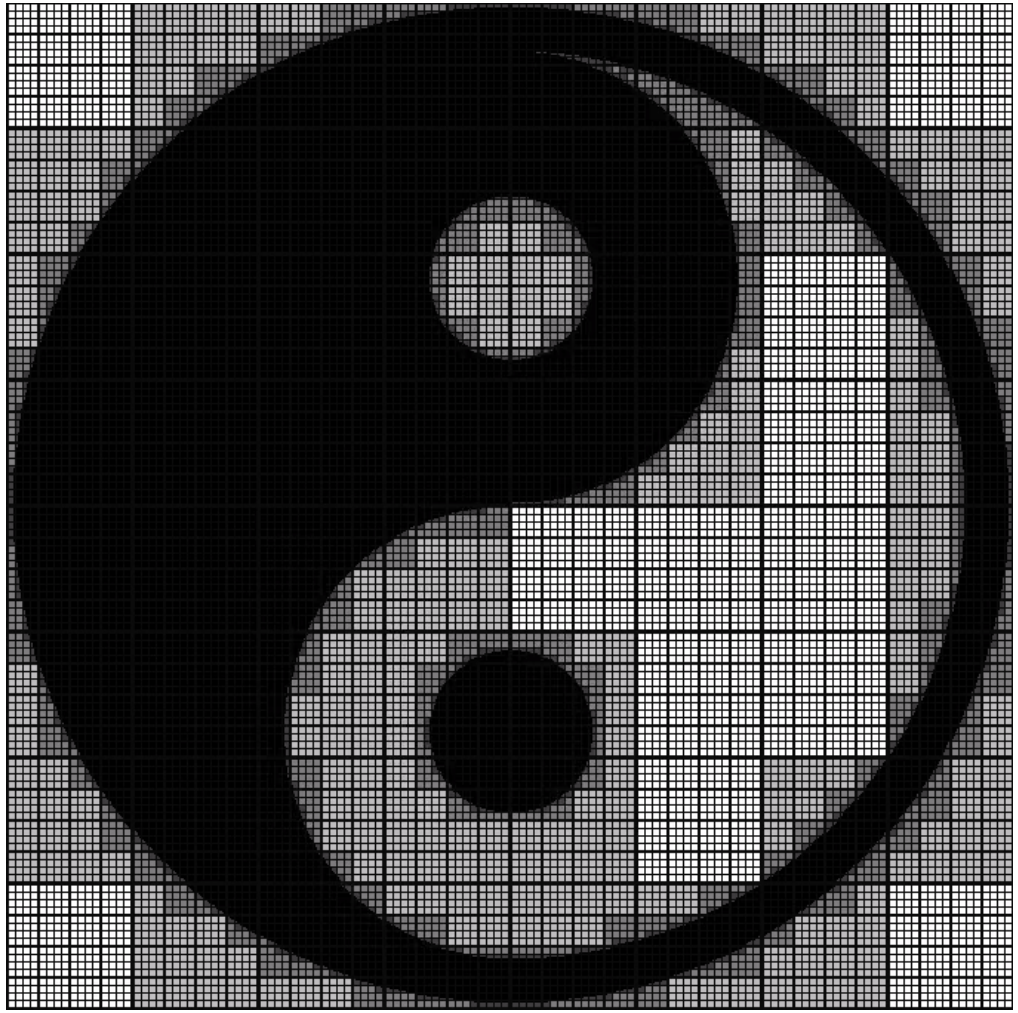
- Check the performance
  - SoA v.s. AoS

Perlin noise: [Link]



- Check the performance
  - Flat layout v.s. hierarchical layout

A rolling Taichi: [Link]



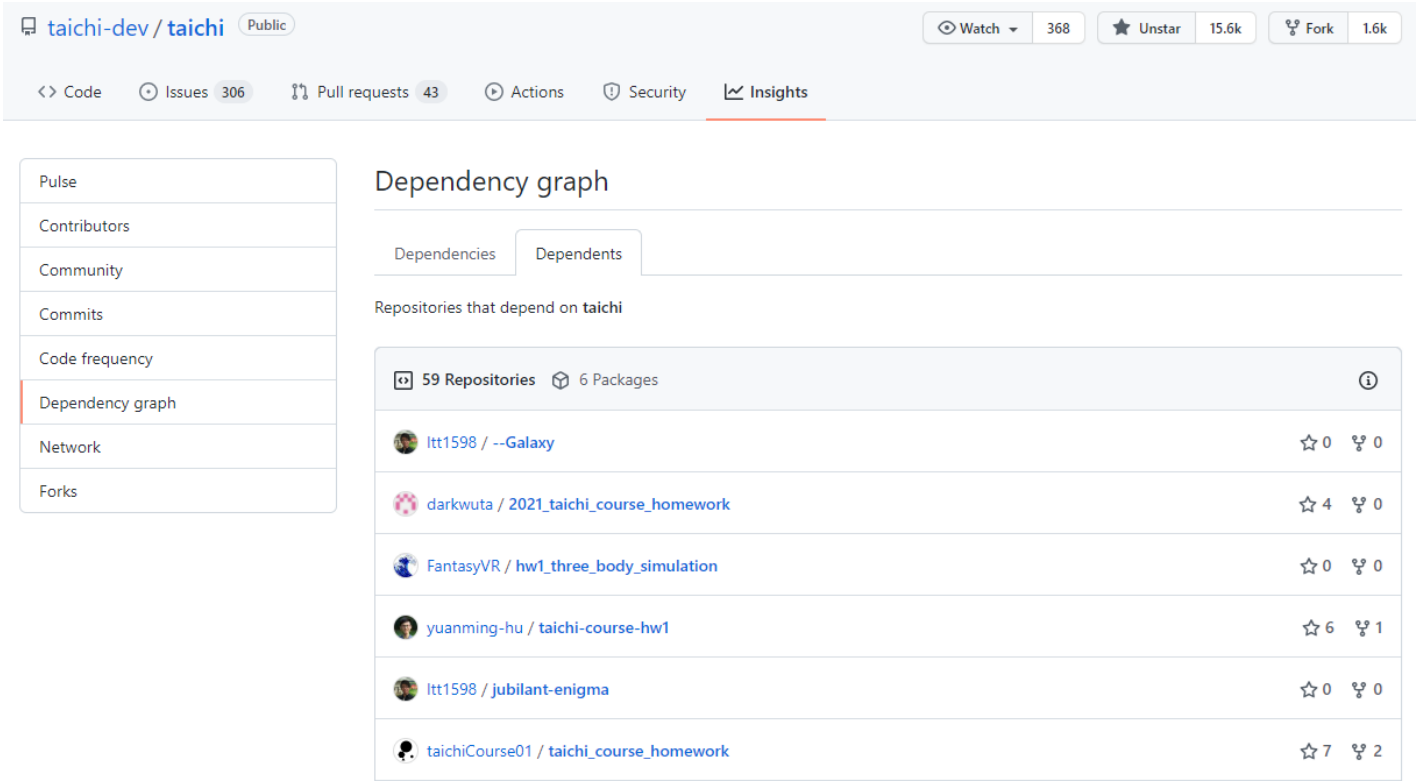
- Check the performance
  - Sparse (.pointer()) layout v.s. dense (.dense()) layout

Share your homework

- Could be ANYTHING you programmed using Taichi
- Help us find your homework by using Template
- Share it with your classmates at forum.taichi.graphics
  - 太极图形课作业区: https://forum.taichi.graphics/c/homework/14
  - Share your Taichi zoo link or your github/gitee link
  - Compile a .gif animation at your will

Gifts for the gifted

- Next check: Nov. 9th 2021



Final tip

- Update your Taichi to 0.8.3 (released today on 10/12/2021)
  - python -m pip install taichi --upgrade

- Taichi is constantly evolving:
  - Raise an issue @https://github.com/taichi-dev/taichi if you think you find a bug

Questions?

本次答疑：10/14
下次直播：10/19
直播回放：Bilibili 搜索「太极图形」
主页&课件：https://github.com/taichiCourse01