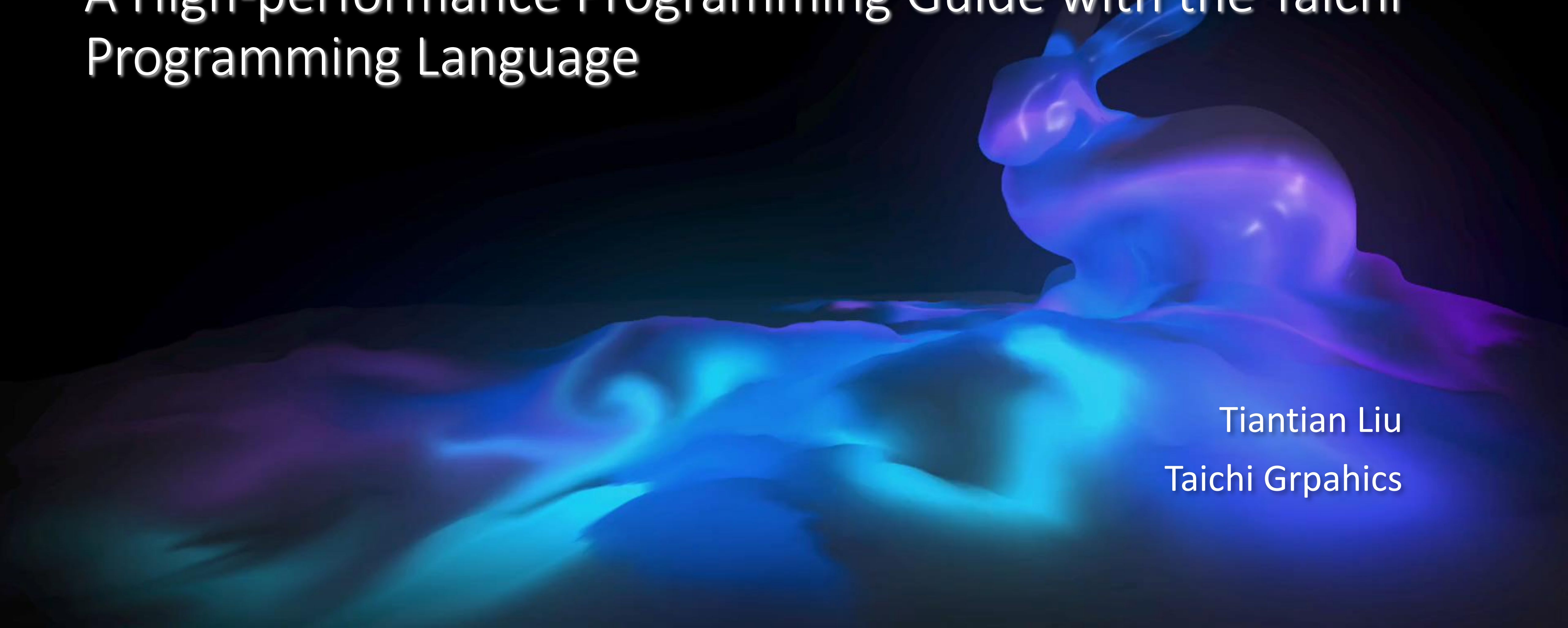
Playing with Your Data:

A High-performance Programming Guide with the Taichi Programming Language

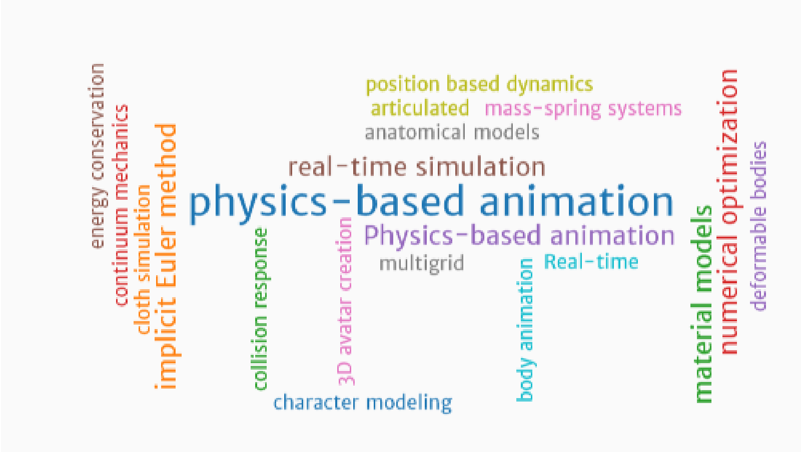


Tiantian Liu

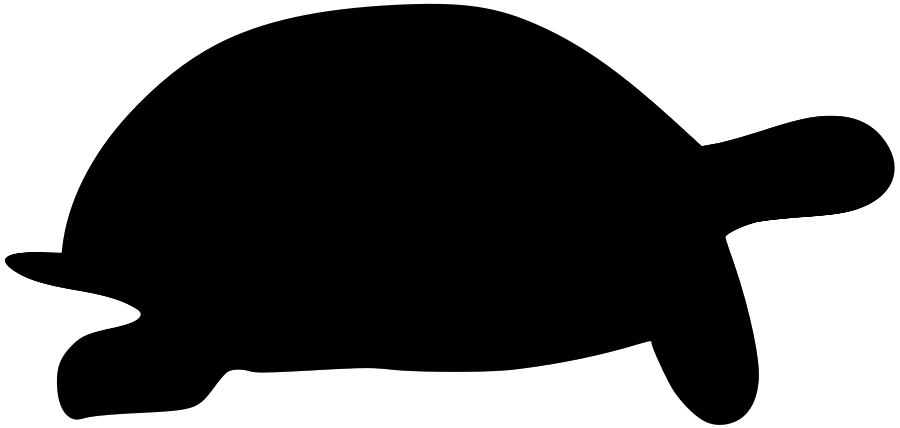
Taichi Grpahics

About me

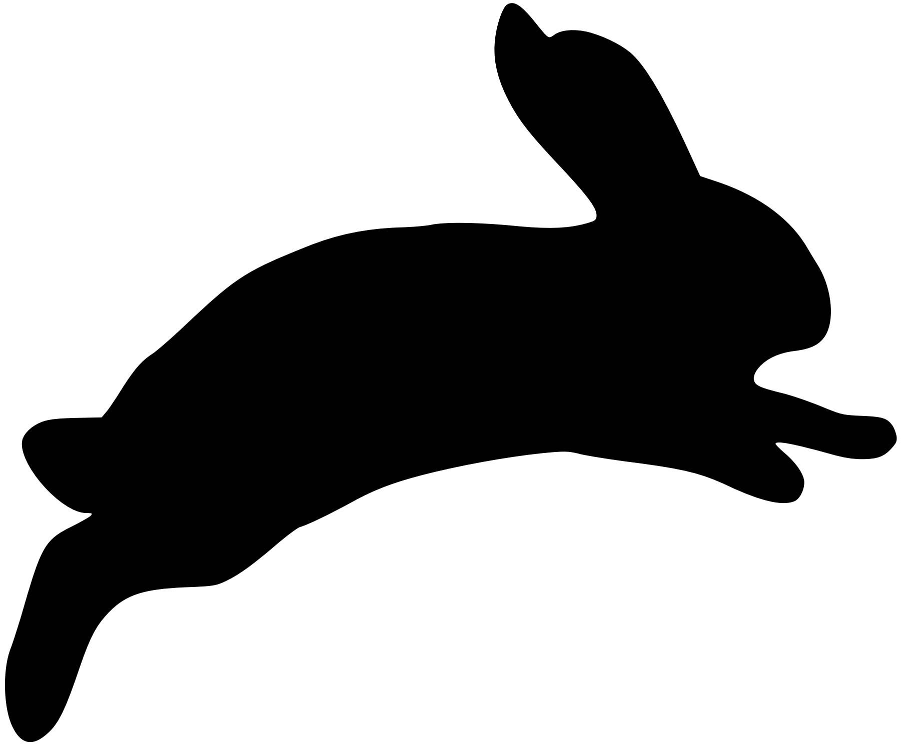
- UPenn -> MSRA -> Taichi Graphics
- Research Interests:
  - Physically based simulation
  - Numerical methods / Nonlinear optimization
  - Parallel computing



The Goal of this Course



Optimizing
Data
Access



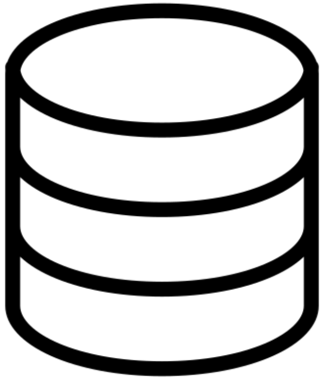
Unoptimized Code

High-performance Code
2-10x faster

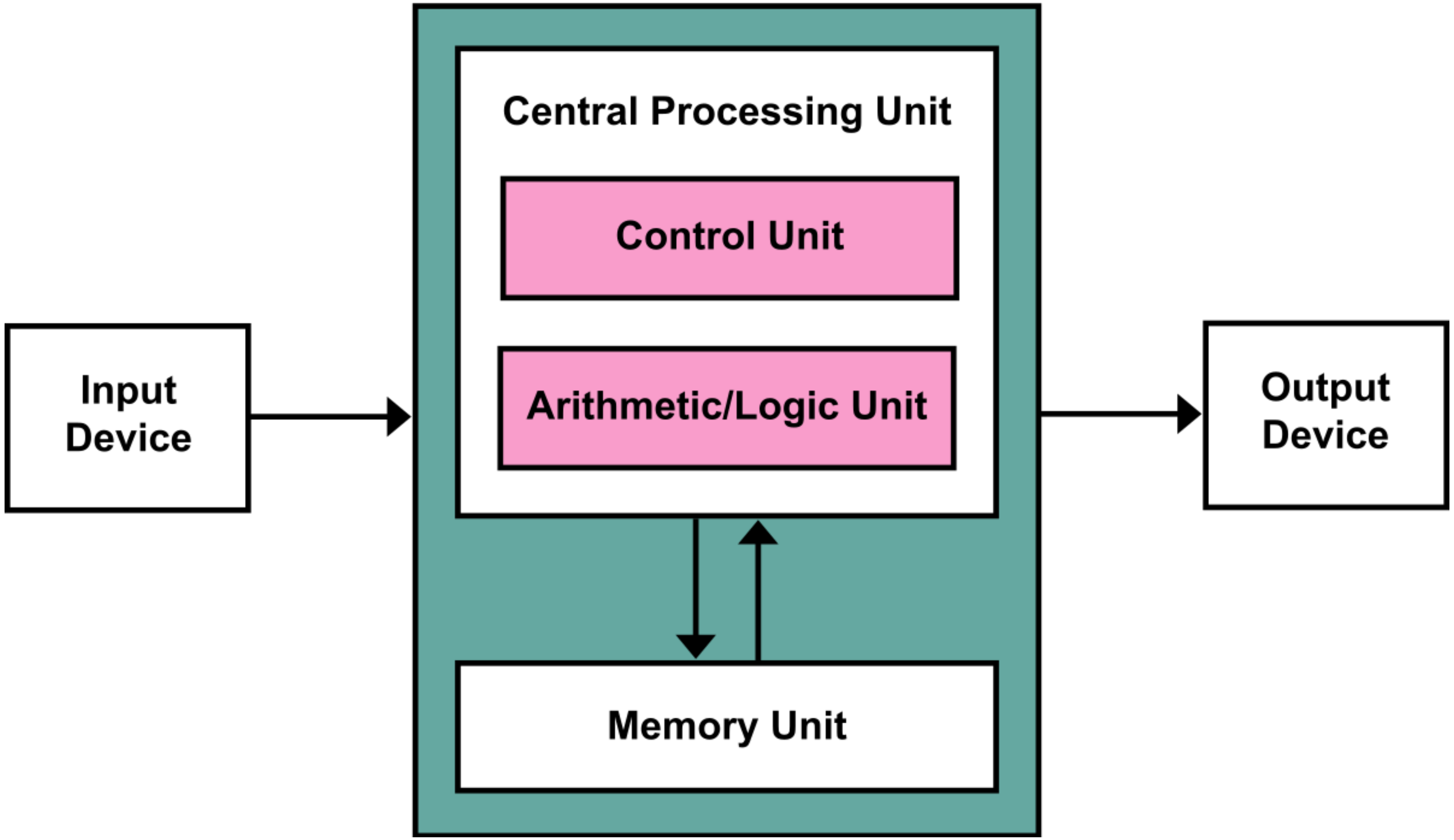
Outline

- The data access matters (10 mins)
- A quick recap of the Taichi programming language (10 mins)
- Efficient dense data layouts (25 mins)
- Spatially sparse data structures (20 mins)
- Mesh-based data access (10 mins)
- Optimization hints for the Taichi compiler (5 mins)
- Quantized data types (15 mins)

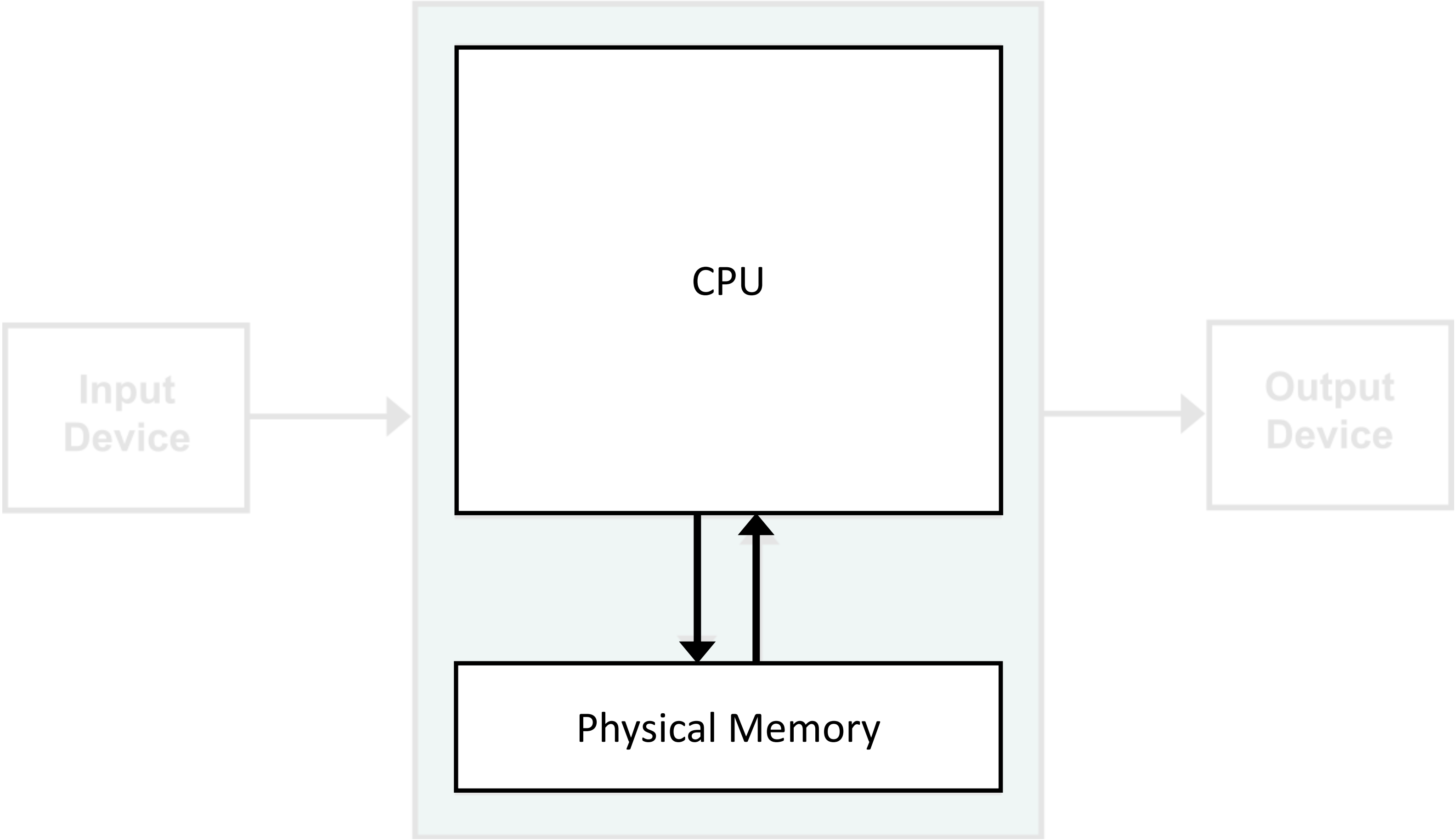
The Data Access Matters



The Von Neumann architecture (1945)

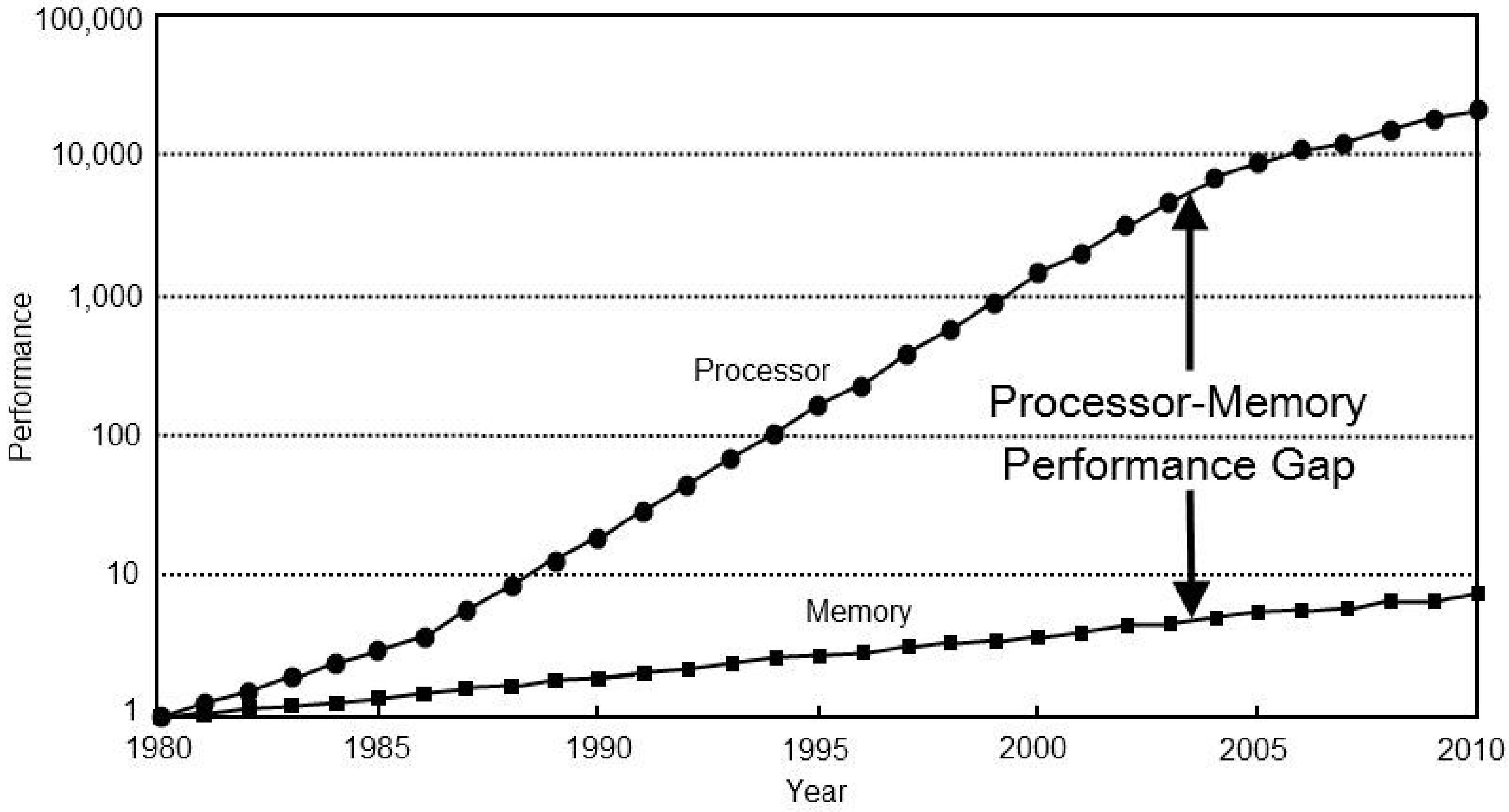


Fast CPU + Fast Data Access = Good Performance



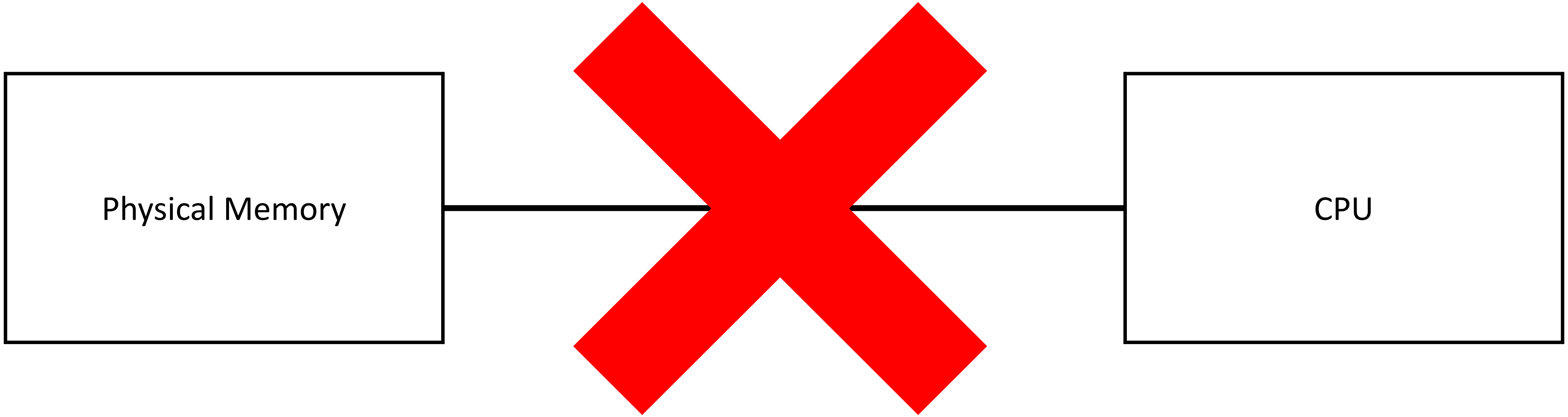
However, our processors run **much** faster than memory

“The era of slow memory”

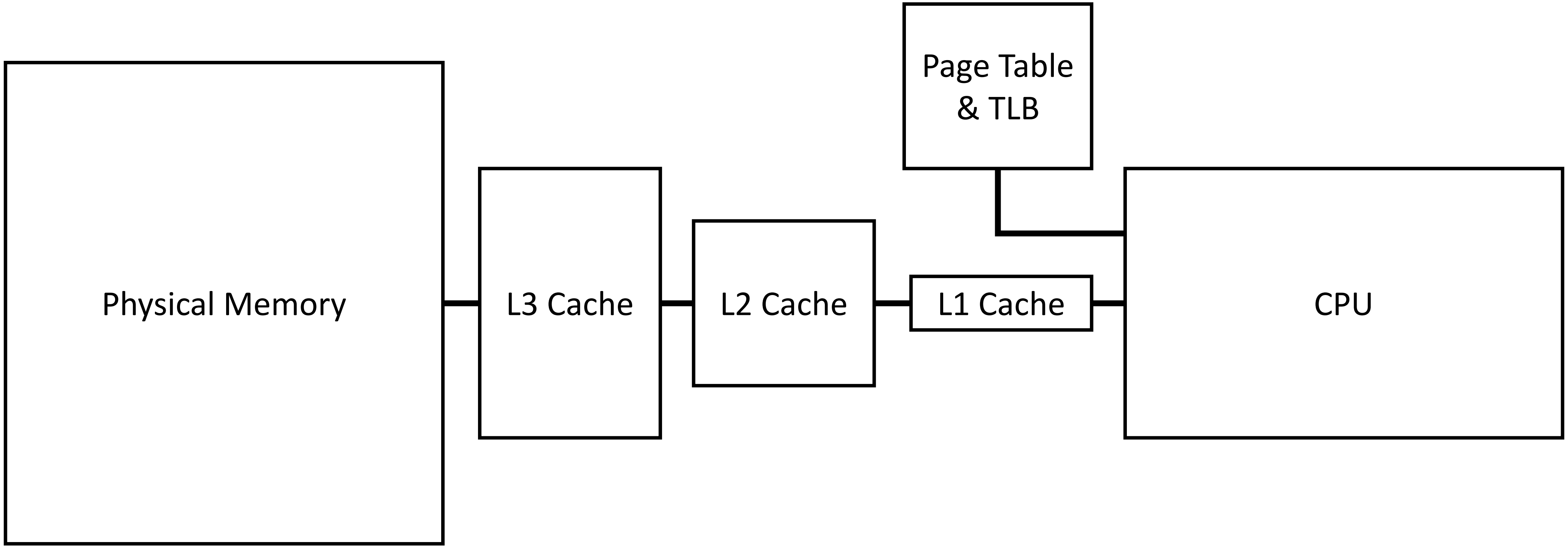


[Hennessy and Patterson 2011]

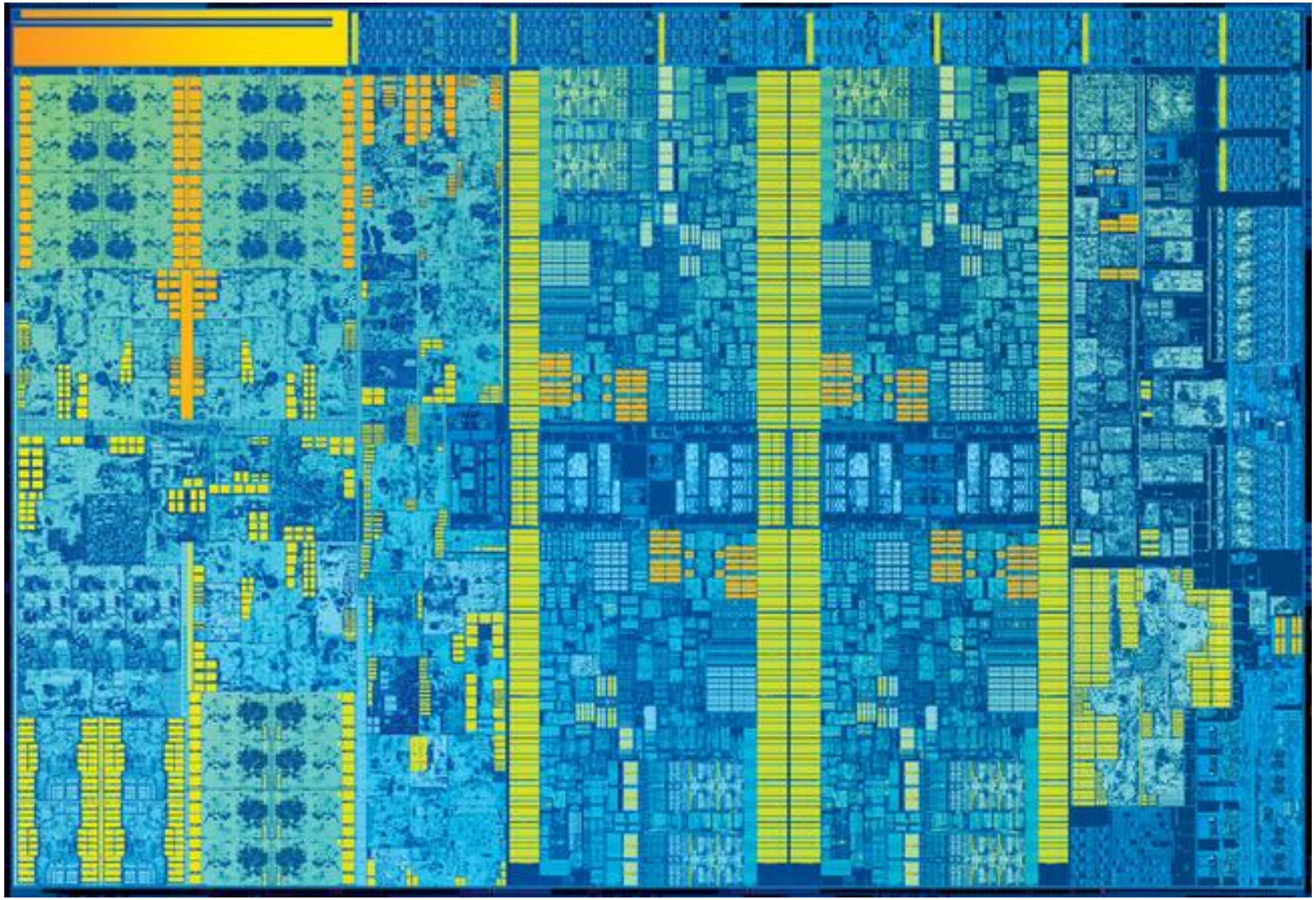
The memory hierarchy



The memory hierarchy

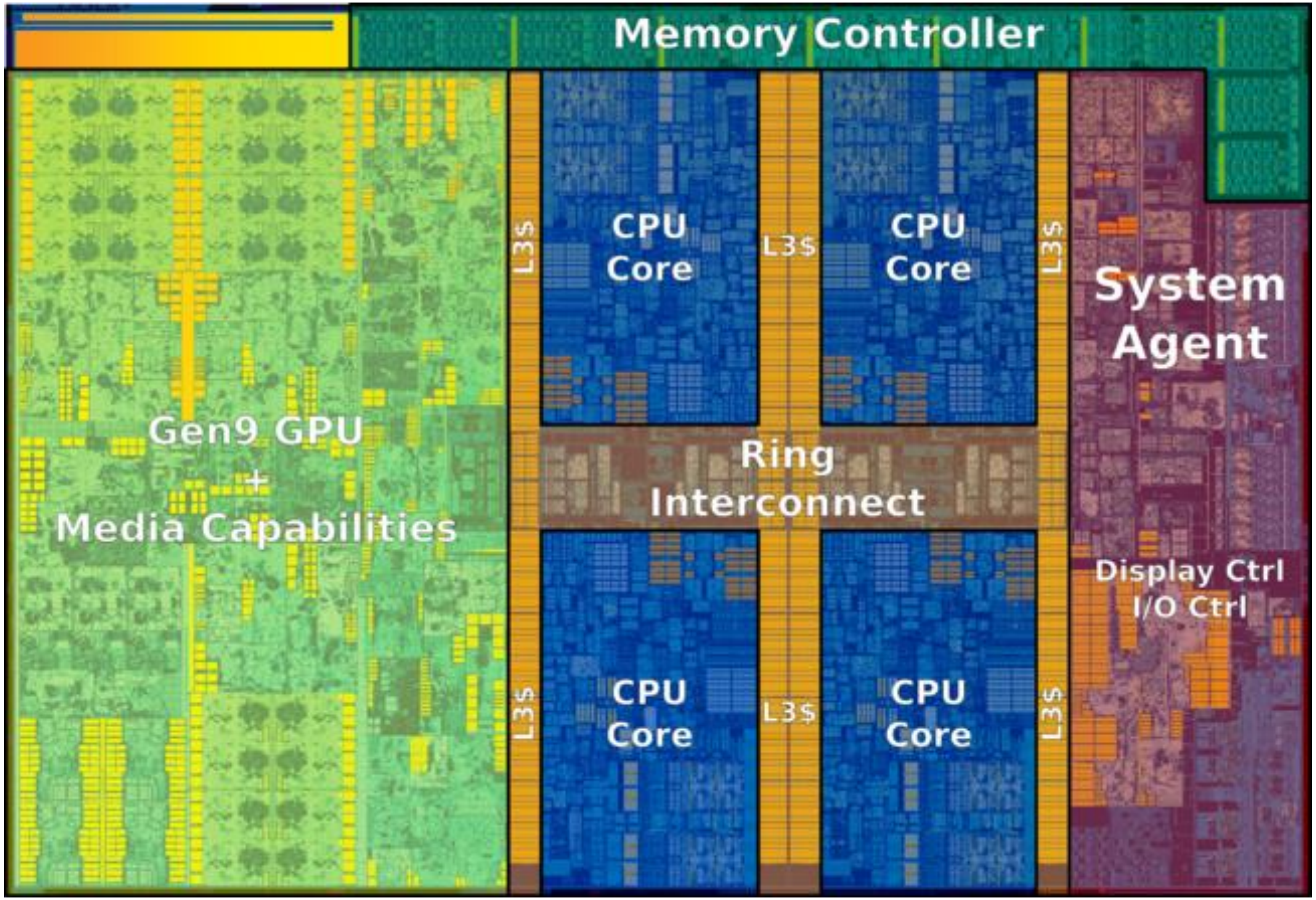


The Intel Skylake i7 (Die)



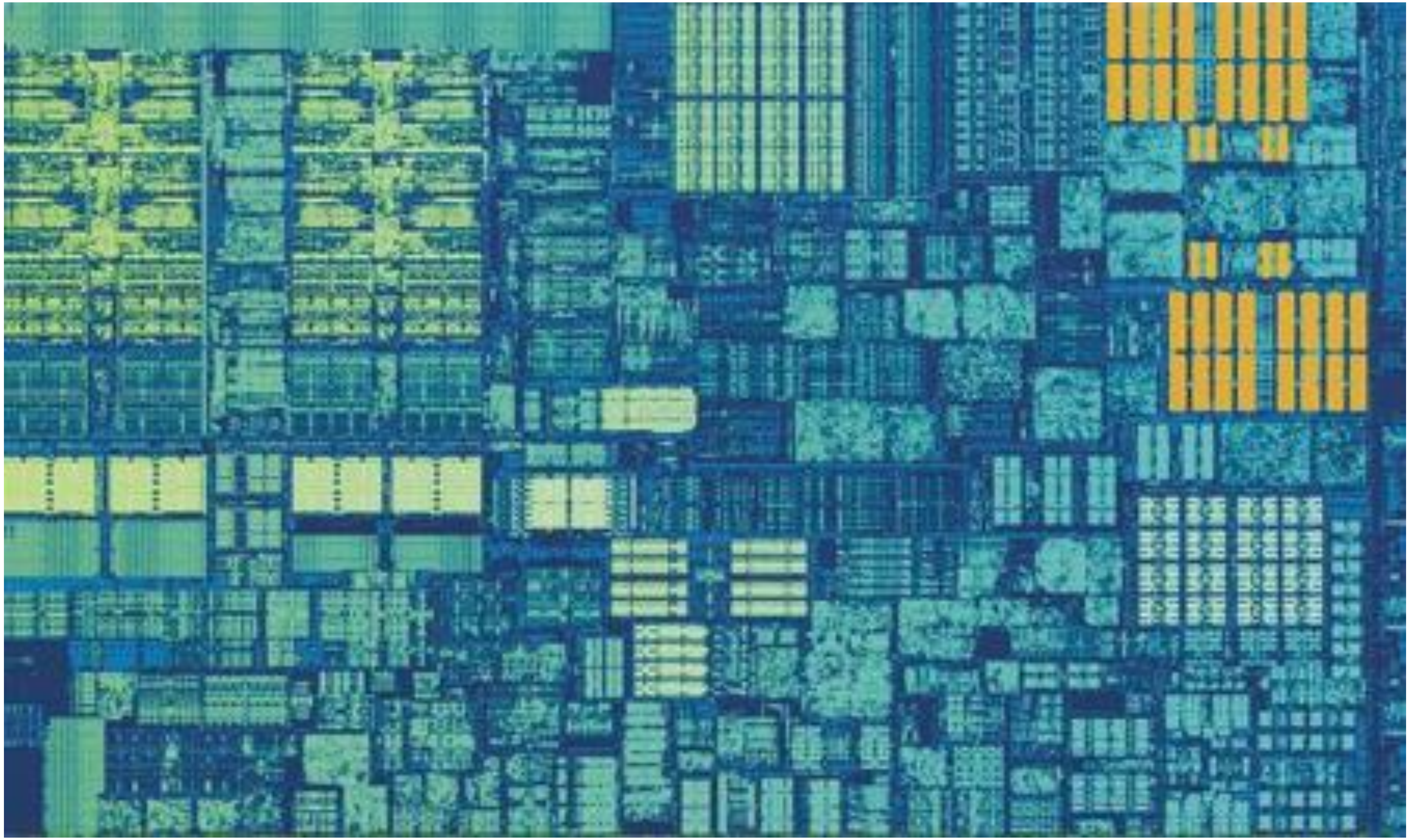
https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

The Intel Skylake i7 (Die)



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

The Intel Skylake i7 (Core)



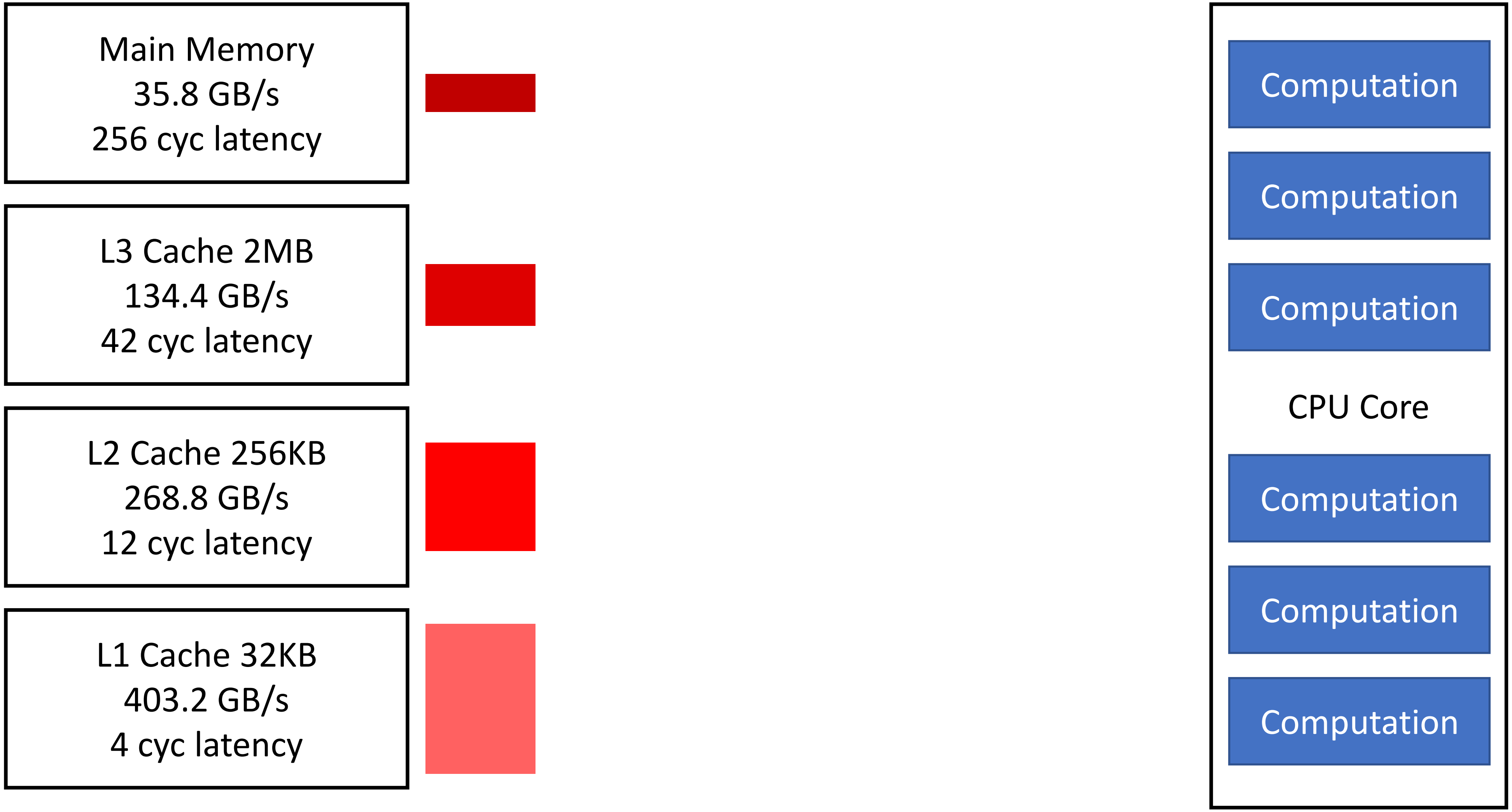
https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

The Intel Skylake i7 (Core)

SIGGRAPH ASIA 2022 DAEGU

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

Caches come to the rescue



Cache lines

```
constexpr int n = 256 * 1024 * 1024;
int a[n];

void benchmark() {
    auto t = get_time();
    for (int i = 0; i < n; i += stride) {
        a[i] = i * 2;
    }
    printf("%f\n", get_time()-t);
}
```

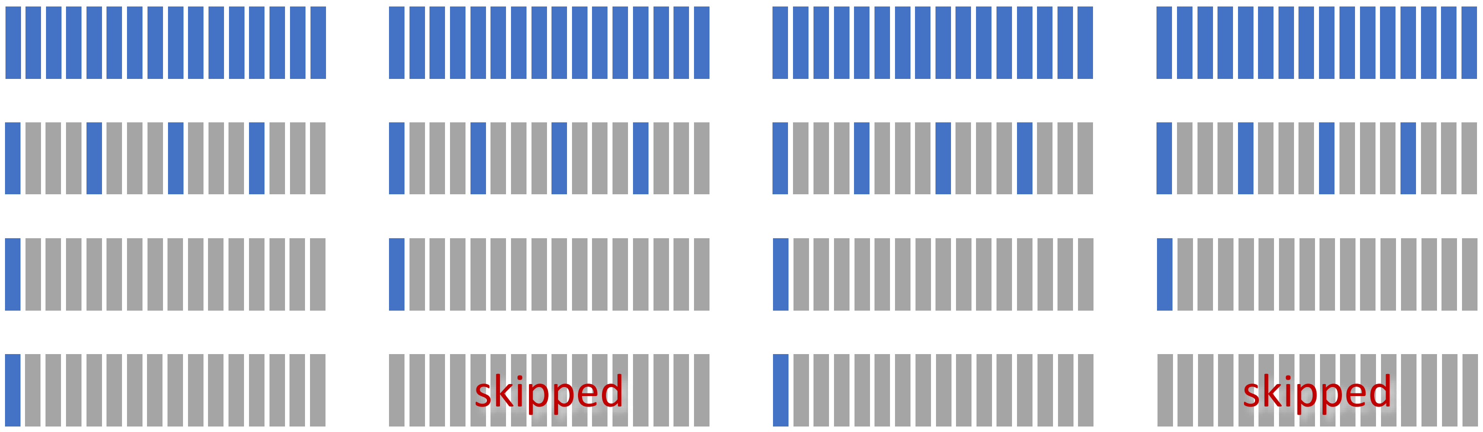
```
stride = 1:  0.13 s
stride = 2:  0.13 s
stride = 4:  0.13 s
stride = 8:  0.13 s
stride = 16: 0.13 s
stride = 32: 0.096 s
stride = 64: 0.069 s
```

Cache lines

```cpp
constexpr int n = 256 * 1024 * 1024;
int a[n];

void benchmark() {
    auto t = get_time();
    for (int i = 0; i < n; i += stride) {
        a[i] = i * 2;
    }
    printf("%f\n", get_time()-t);
}
```

```
stride = 1:  0.13 s
stride = 2:  0.13 s
stride = 4:  0.13 s
stride = 8:  0.13 s
stride = 16: 0.13 s
stride = 32: 0.096 s
stride = 64: 0.069 s
```

```
stride = 1:  0.13 s
stride = 4:  0.13 s
stride = 16: 0.13 s
stride = 32: 0.096 s
```



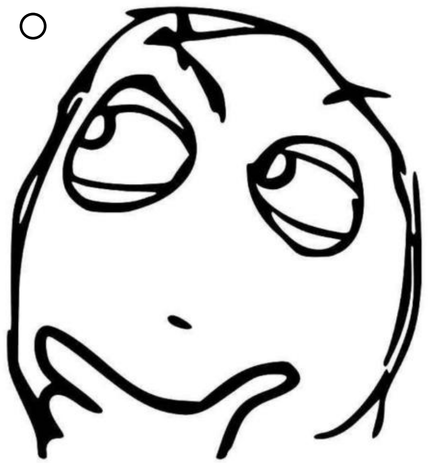
Remark

- Memory accesses are usually much slower than computations in modern computer architectures
- The slow memory problems can be even worse for GPUs
- Optimizing data access can help accelerate our code by a lot
  - Improving the cache hit rate
  - Making use of prefetched cache lines

Remark

- Memory accesses ar… …an computations in modern computer ar…
- The slow memory prob… …worse for GPUs
- Optimizing data access can help accelerate our …ode by a lot
  - Improving the cache hit rate
  - Making use of prefetched cache lines

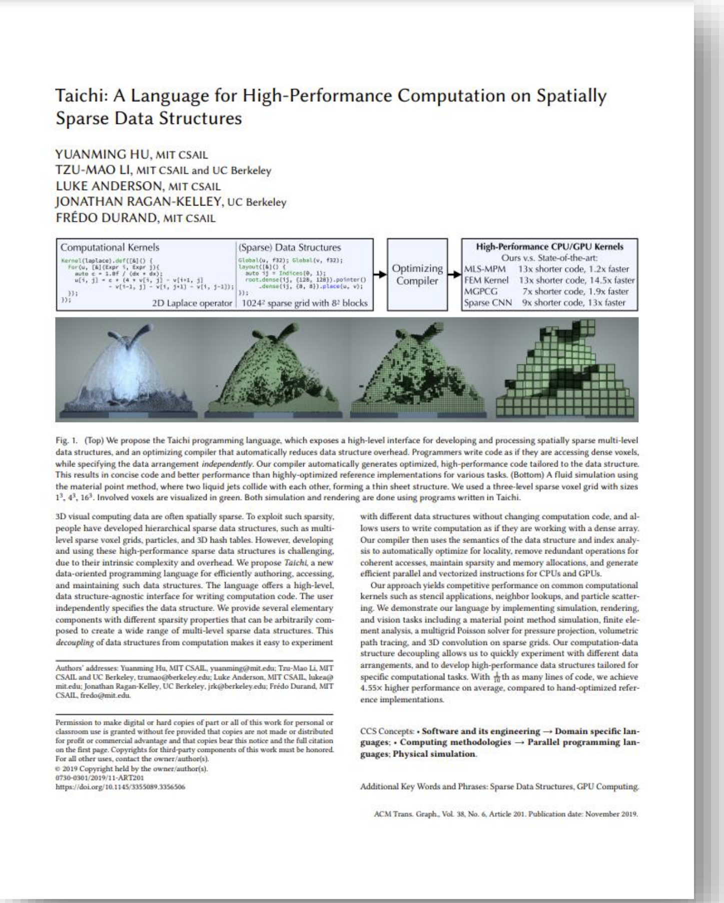
Now I know why.
But how?



A Quick Recap of Taichi

Taichi Lang

Taichi-Lang was born in the graphics community



Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures

YUANMING HU, MIT CSAIL
TZU-MAO LI, MIT CSAIL and UC Berkeley
LUKE ANDERSON, MIT CSAIL
JONATHAN RAGAN-KELLEY, UC Berkeley
FRÉDO DURAND, MIT CSAIL

Fig. 1. (Top) We propose the Taichi programming language, which exposes a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method, where two liquid jets collide with each other, forming a thin sheet structure. We used a three-level sparse voxel grid with sizes 1^3, 4^3, 16^3. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

3D visual computing data are often spatially sparse. To exploit such sparsity, people have developed hierarchical sparse data structures, such as multi-level sparse voxel grids, particles, and 3D hash tables. However, developing and using these high-performance sparse data structures is challenging, due to their intrinsic complexity and overhead. We propose *Taichi*, a new data-oriented programming language for efficiently authoring, accessing, and maintaining such data structures. The language offers a high-level, data structure-agnostic interface for writing computation code. The user independently specifies the data structure. We provide several elementary components with different sparsity properties that can be arbitrarily composed to create a wide range of multi-level sparse data structures. This *decoupling* of data structures from computation makes it easy to experiment with different data structures without changing computation code, and allows users to write computation as if they are working with a dense array. Our compiler then uses the semantics of the data structure and index analysis to automatically optimize for locality, remove redundant operations for coherent accesses, maintain sparsity and memory allocations, and generate efficient parallel and vectorized instructions for CPUs and GPUs.

Our approach yields competitive performance on common computational kernels such as stencil applications, neighbor lookups, and particle scattering. We demonstrate our language by implementing simulation, rendering, and vision tasks including a material point method simulation, finite element analysis, a multigrid Poisson solver for pressure projection, volumetric path tracing, and 3D convolution on sparse grids. Our computation-data structure decoupling allows us to quickly experiment with different data arrangements, and to develop high-performance data structures tailored for specific computational tasks. With $\frac{1}{10}$th as many lines of code, we achieve 4.55× higher performance on average, compared to hand-optimized reference implementations.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Computing methodologies → Parallel programming languages**; **Physical simulation**.

Additional Key Words and Phrases: Sparse Data Structures, GPU Computing.

Authors' addresses: Yuanming Hu, MIT CSAIL, yuanming@mit.edu; Tzu-Mao Li, MIT CSAIL and UC Berkeley, tzumao@berkeley.edu; Luke Anderson, MIT CSAIL, lukea@mit.edu; Jonathan Ragan-Kelley, UC Berkeley, jrk@berkeley.edu; Frédo Durand, MIT CSAIL, fredo@mit.edu.

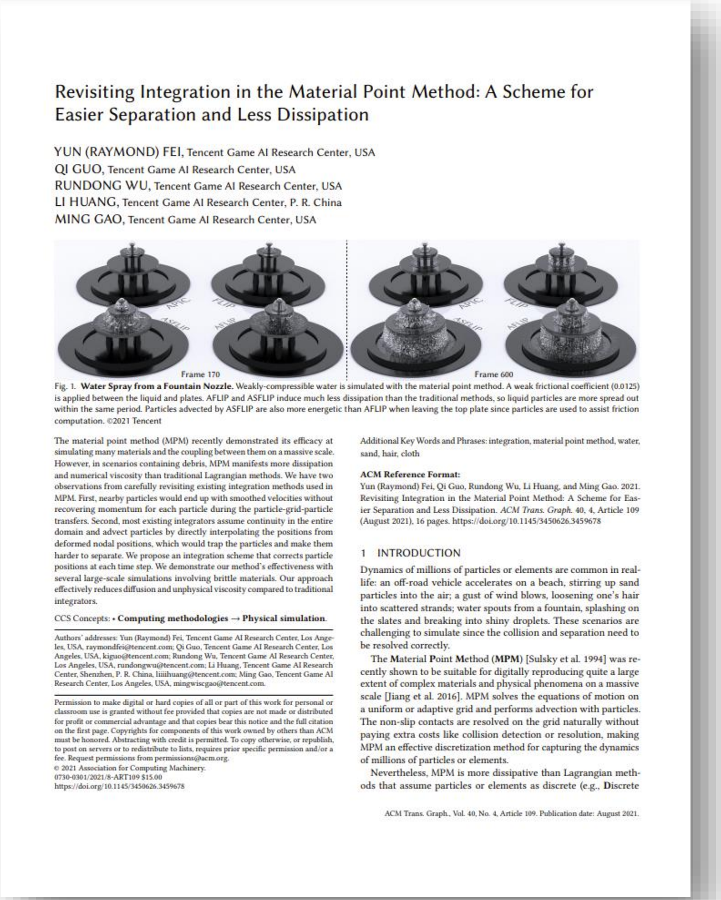
0730-0301/2019/11-ART201
https://doi.org/10.1145/3355089.3356506

ACM Trans. Graph., Vol. 38, No. 6, Article 201. Publication date: November 2019.

[Hu et al. 2019]

[Hu 2020]

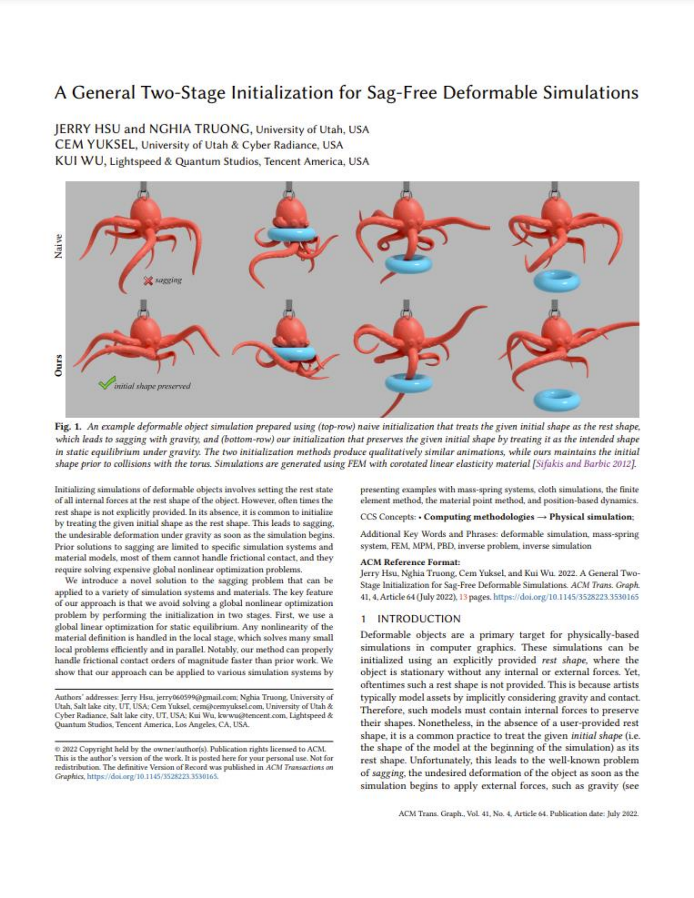
Taichi-Lang serves graphics applications



Revisiting Integration in the Material Point Method: A Scheme for Easier Separation and Less Dissipation

YUN (RAYMOND) FEI, Tencent Game AI Research Center, USA
QI GUO, Tencent Game AI Research Center, USA
RUNDONG WU, Tencent Game AI Research Center, USA
LI HUANG, Tencent Game AI Research Center, P. R. China
MING GAO, Tencent Game AI Research Center, USA

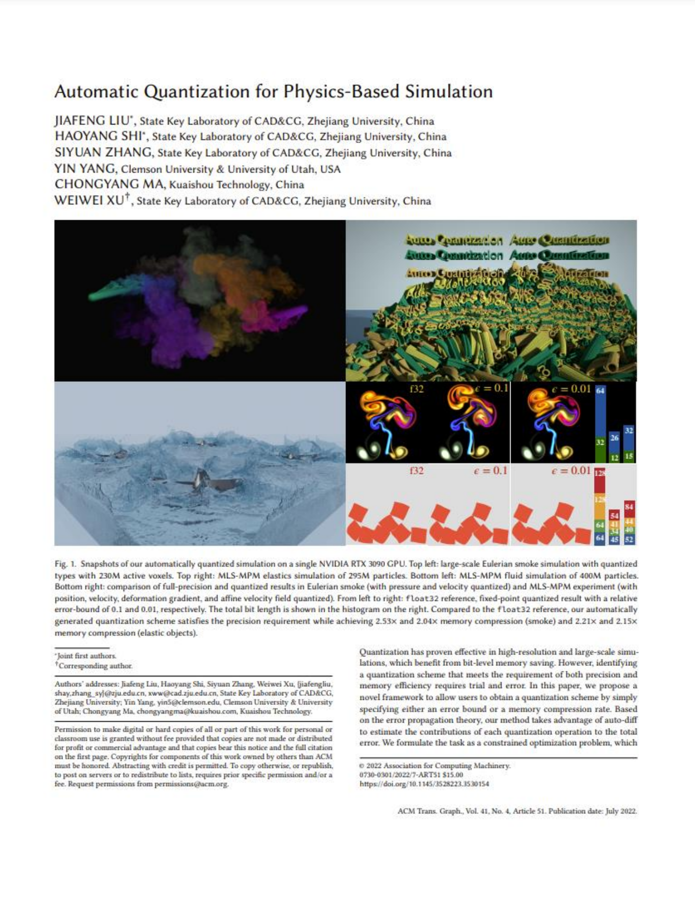
[Fei et al. 2021]



A General Two-Stage Initialization for Sag-Free Deformable Simulations

JERRY HSU and NGHIA TRUONG, University of Utah, USA
CEM YUKSEL, University of Utah & Cyber Radiance, USA
KUI WU, Lightspeed & Quantum Studios, Tencent America, USA

[Hsu et al. 2022]



Automatic Quantization for Physics-Based Simulation

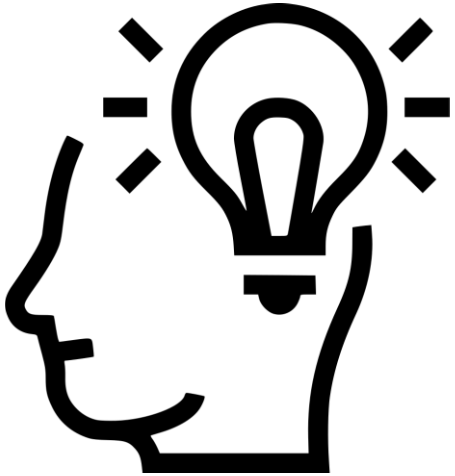
JIAFENG LIU*, State Key Laboratory of CAD&CG, Zhejiang University, China
HAOYANG SHI*, State Key Laboratory of CAD&CG, Zhejiang University, China
SIYUAN ZHANG, State Key Laboratory of CAD&CG, Zhejiang University, China
YIN YANG, Clemson University & University of Utah, USA
CHONGYANG MA, Kuaishou Technology, China
WEIWEI XU†, State Key Laboratory of CAD&CG, Zhejiang University, China

[Liu et al. 2022]

The Taichi programming language

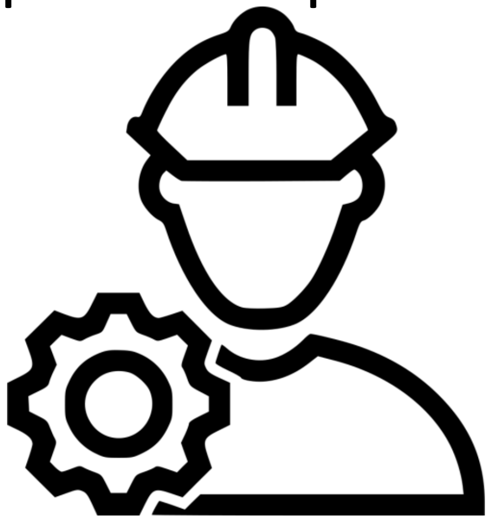
SIGGRAPH ASIA 2022 DAEGU

- A domain specific language (DSL) for computer graphics and parallel computing. The Taichi-lang…
  - has a Python frontend;
  - is Optimized for parallel computing;
  - can be deployed everywhere on CPUs and GPUs;
  - thrives through open-source development.



The Taichi programming language

- A domain specific language (DSL) for computer graphics and parallel computing. The Taichi-lang…
  - has a Python frontend;
  - is Optimized for parallel computing;
  - can be deployed everywhere on CPUs and GPUs;
  - thrives through open-source development.



The Taichi programming language

- A domain specific language (DSL) for computer graphics and parallel computing. The Taichi-lang…
  - has a Python frontend;
  - is Optimized for parallel computing;
  - can be deployed everywhere on CPUs and GPUs;
  - thrives through open-source development.

The Taichi programming language

SIGGRAPH ASIA 2022 DAEGU

- A domain specific language (DSL) for computer graphics and parallel computing. The Taichi-lang…
  - has a Python frontend;
  - is Optimized for parallel computing;
  - can be deployed everywhere on CPUs and GPUs;
  - thrives through open-source development.

Taichi

Installation

- python3 -m **pip install taichi** -U

- Until today (10/30/2022)
  - Latest version: 1.2.0
  - Taichi supports Python 3.6/3.7/3.8/3.9/3.10 (64-bit)
  - Taichi supports Windows, Linux, and OS X.

Quick start

- Use python3 –m taichi or simply ti to start Taichi's CLI

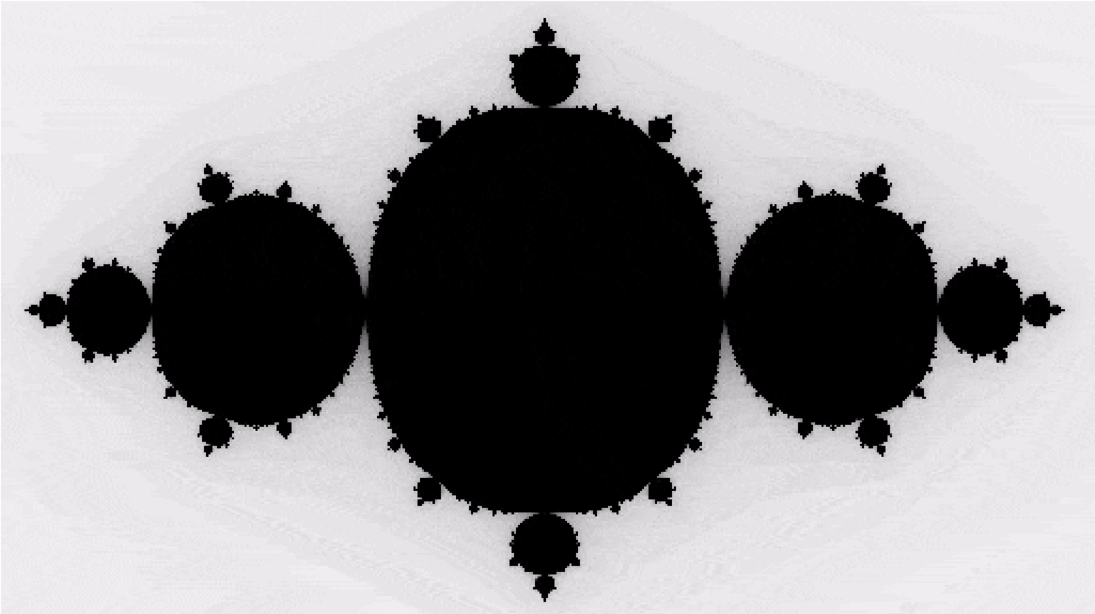
- For example:
  - ti gallery: list featured examples / ti example: list all provided examples

```
TAICHI EXAMPLES

ad_gravity               keyboard                 patterns
comet                    laplace                  pbf2d
cornell_box              mandelbrot_zoom          physarum
diff_sph                 marching_squares         print_offset
euler                    mass_spring_3d_ggui      rasterizer
explicit_activation      mass_spring_game         regression
export_mesh              mass_spring_game_ggui    sdf_renderer
export_ply               mciso_advanced           simple_derivative
export_videos            mgpcg                    simple_texture
fem128                   mgpcg_advanced           simple_uv
fem128_ggui              minimal                  stable_fluid
fem99                    minimization             stable_fluid_ggui
fractal                  mpm128                   stable_fluid_graph
fractal3d_ggui           mpm128_ggui              taichi_bitmasked
fullscreen               mpm3d                    taichi_dynamic
game_of_life             mpm3d_ggui               taichi_logo
gui_image_io             mpm88                    taichi_sparse
gui_widgets              mpm88_graph              tutorial
implicit_fem             mpm99                    vortex_rings
implicit_mass_spring     mpm_lagrangian_forces    waterwave
inital_value_problem     nbody
initial_value_problem    odop_solar

usage: ti example [-h] [-p] [-P] [-s] name
```

Let’s try: a simple Julia set program

```
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

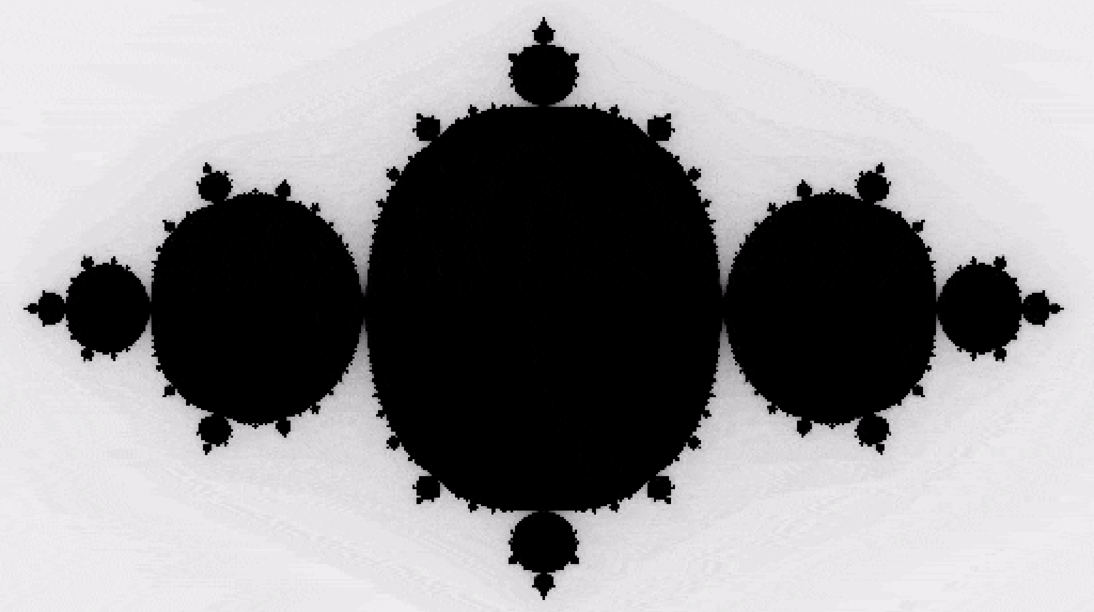
- $z = z^2 + c$



“import taichi as ti”

```python
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default
n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

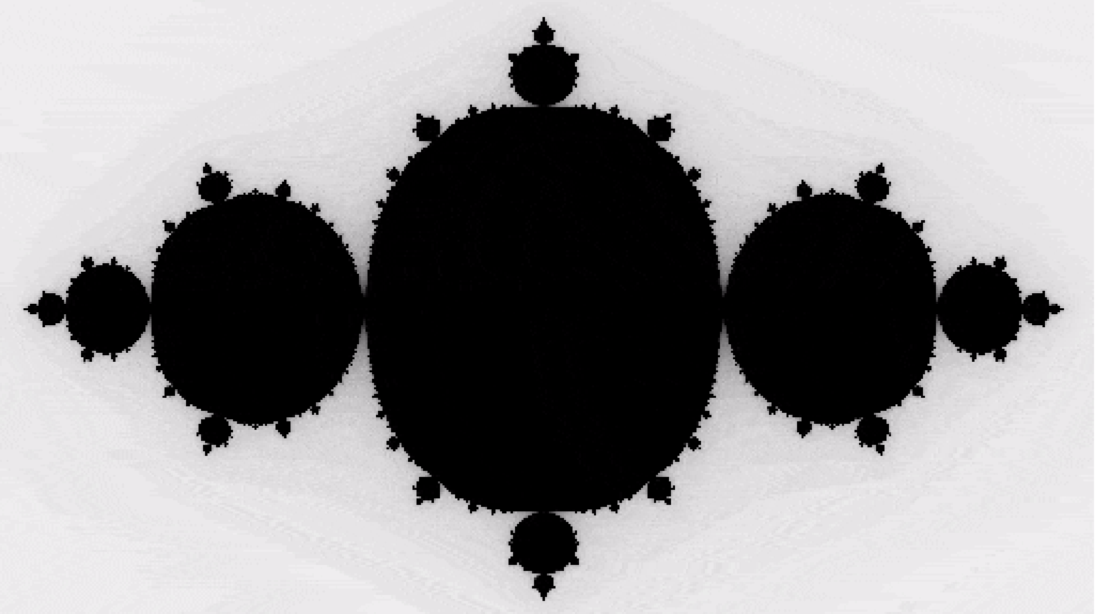
- ti.init()
  - The entry point of every Taichi program



The most useful data container: ti.field

```python
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

- multi-dimensional array of elements
- like an ndarray in *NumPy* or a tensor in *PyTorch*



ti.field

- “a global N-d array of elements”

```
heat_field = ti.field(dtype=ti.f32, shape=(256, 256))
```

ti.field

- "a global N-d array of elements"
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, ...)
  - elements: scalar, vector, matrix, struct

ti.field

- "a global N-d array of elements"
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct

ti.field

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct
  - access elements in a field using [i,j,k,…] indexing

```python
import taichi as ti
ti.init()

pixels = ti.field(dtype=float, shape=(16, 8))

pixels[1, 2] = 42.0
```

```python
import taichi as ti
ti.init()

vf = ti.Vector.field(3, ti.f32, shape=4)

@ti.kernel
def foo():
    v = ti.Vector([1, 2, 3])
    vf[0] = v
```

ti.field

SIGGRAPH ASIA 2022 DAEGU

- “a global N-d array of elements”
  - global: can be read/written from both the Taichi-scope and the Python-scope
  - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, …)
  - elements: scalar, vector, matrix, struct
  - access elements in a field using [i,j,k,…] indexing
    - One special case, access a zero-d field using [None]

```
zero_d_scalar = ti.field(ti.f32, shape=())
zero_d_scalar[None] = 1.0

zero_d_vec = ti.Vector.field(2, ti.f32, shape=())
zero_d_vec[None] = ti.Vector([2.0, 2.5])
```

ti.field examples

- "3D gravitational field in a 256x256x128 room"

```
gravitational_field = ti.Vector.field(n = 3,dtype=ti.f32,shape=(256,256,128))
```

- "2D strain-tensor field in a 64x64 grid"

```
strain_tensor_field = ti.Matrix.field(n = 2,m = 2, dtype=ti.f32, shape=(64,64))
```

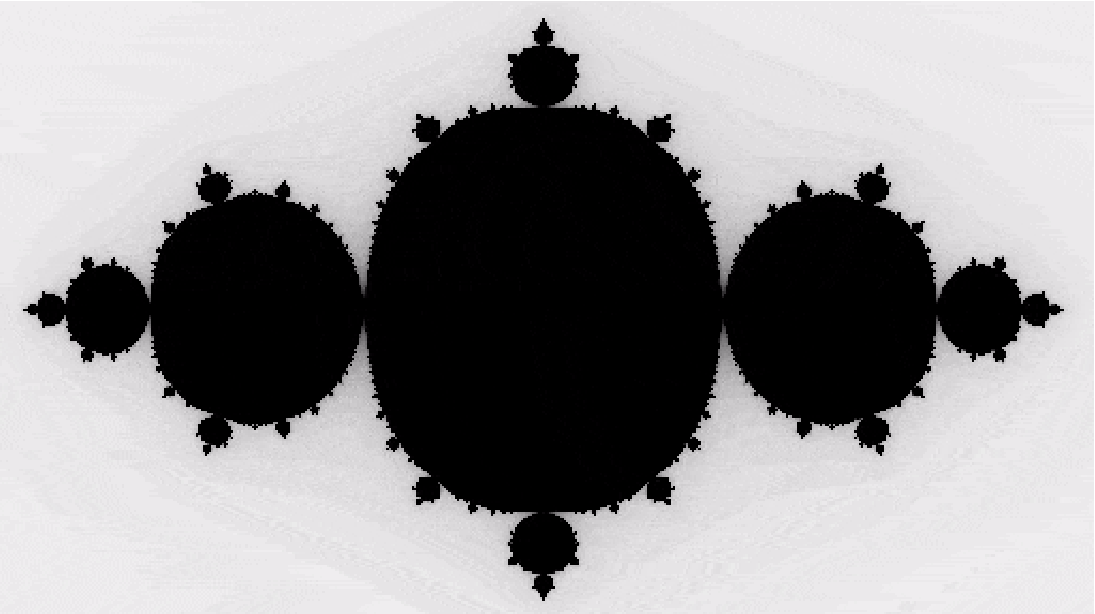
- "a global scalar that I want to access in a Taichi kernel"

```
global_scalar = ti.field(dtype=ti.f32, shape=())
```

Computations in Taichi


```python
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

- Decorated with @ti.kernel or @ti.func
- The outermost for loop is automatically parallelized



For-loops in a @ti.kernel

SIGGRAPH ASIA 2022 DAEGU

- For loops at the **outermost scope** in a Taichi kernel is **automatically parallelized**

```
@ti.kernel
def fill():
    for i in range(10): # Parallelized
        x[i] += i

        s = 0
        for j in range(5): # Serialized in each parallel thread
            s += j

        y[i] = s

    for k in rang(20): # Parallelized
        z[k] = k
```

For-loops in a @ti.kernel

- Outermost scope ?

```
import taichi as ti
ti.init()

@ti.kernel
def foo(k: ti.i32):
    for i in range(10): # Parallelized :-)
        if k > 42:
            ...

@ti.kernel
def bar(k: ti.i32):
    if k > 42:
        for i in range(10): # Serial :-(
            ...
```

Types of for-loops in Taichi

- range-for: loops over a **range**, identical to Python range-for
- struct-for: loops over a **ti.field**, only lives at the outermost scope

```python
import taichi as ti
ti.init()

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
  for i in range(N):
      x[i] = i

foo()
```

```python
import taichi as ti
ti.init()

N = 10
x = ti.Vector.field(2,dtype=ti.i32, shape=(N,N))

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

foo()
```

Types of for-loops in Taichi

- range-for: loops over a **range**, identical to Python range-for
- struct-for: loops over a **ti.field**, only lives at the outermost scope

```python
import taichi as ti
ti.init()

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
  for i in range(N):
      x[i] = i

foo()
```

```python
import taichi as ti
ti.init()

N = 10
x = ti.Vector.field(2,dtype=ti.i32, shape=(N,N))

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

foo()
```

Race condition

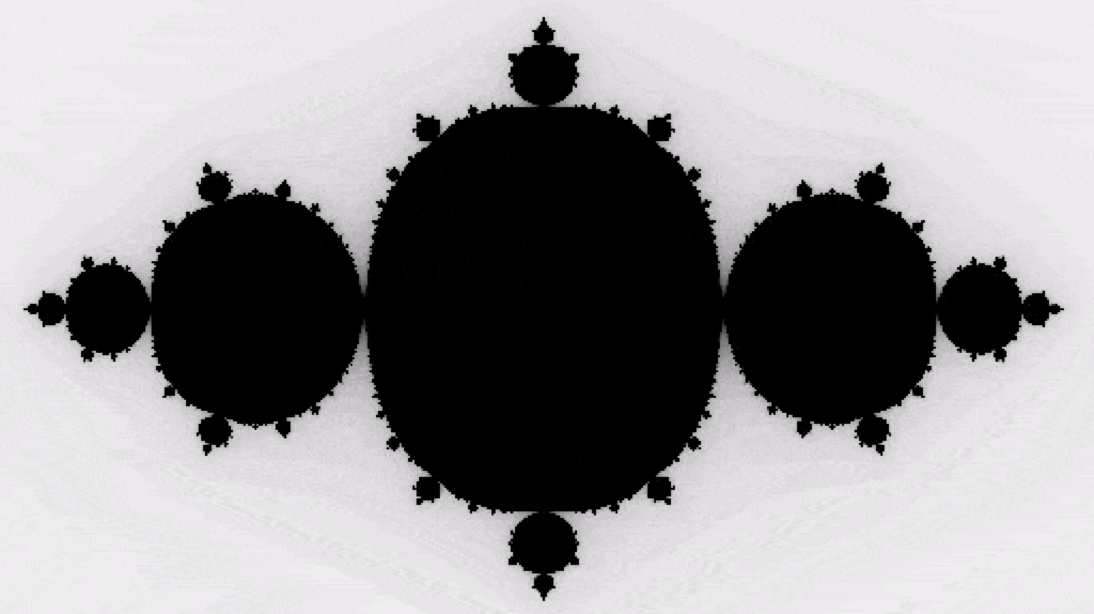
- Taichi uses += as an atomic add
- The compiler optimizes for unnecessary atomic operations

```python
@ti.kernel
def sum():
    for i in range(10):
        # 1. OK
        total[None] += x[i]

        # 2. OK
        ti.atomic_add(total[None], x[i])

        # 3. data race
        total[None] = total[None] + x[i]
```

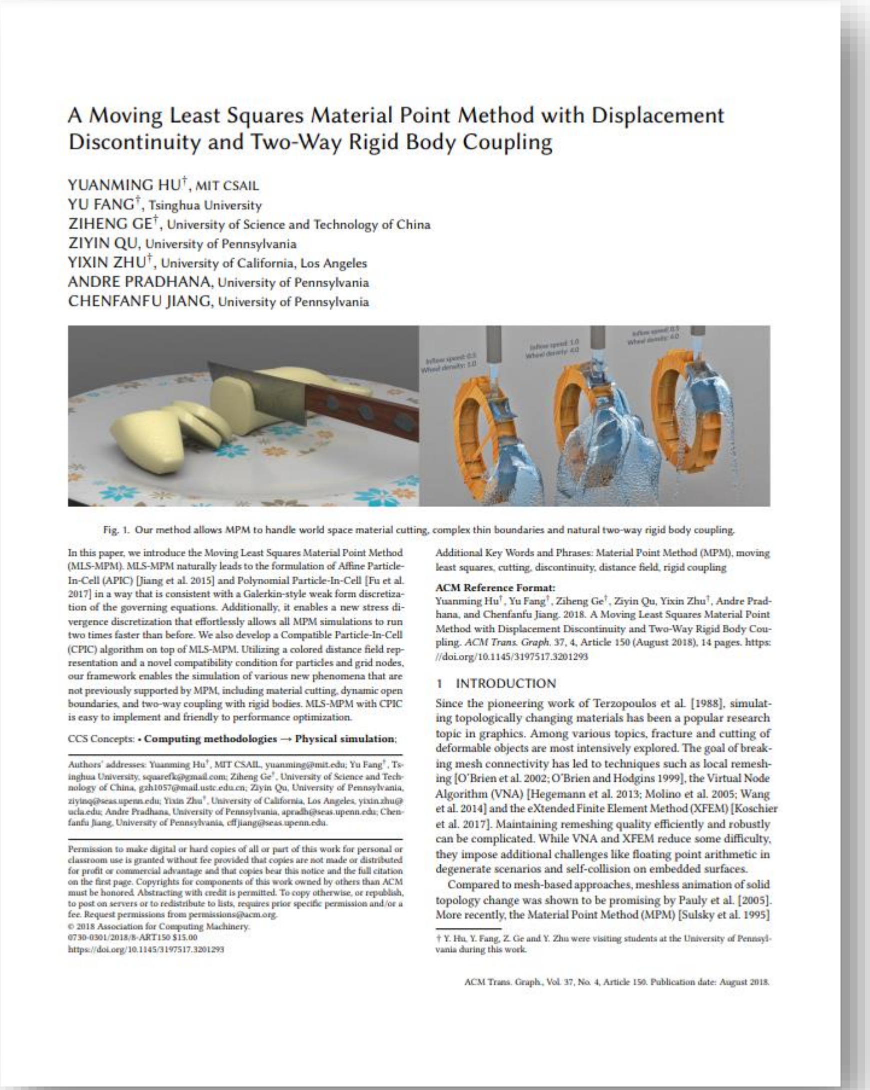
Visualization in Taichi

```
import taichi as ti

ti.init(arch=ti.gpu) # Run on GPU by default

n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

- Using the GUI or GGUI system to visualize your results



"ti example mpm128" [Hu et al. 2018]



A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling

YUANMING HU[†], MIT CSAIL
YU FANG[†], Tsinghua University
ZIHENG GE[†], University of Science and Technology of China
ZIYIN QU, University of Pennsylvania
YIXIN ZHU[†], University of California, Los Angeles
ANDRE PRADHANA, University of Pennsylvania
CHENFANFU JIANG, University of Pennsylvania

Fig. 1. Our method allows MPM to handle world space material cutting, complex thin boundaries and natural two-way rigid body coupling.

In this paper, we introduce the Moving Least Squares Material Point Method (MLS-MPM). MLS-MPM naturally leads to the formulation of Affine Particle-In-Cell (APIC) [Jiang et al. 2015] and Polynomial Particle-In-Cell [Fu et al. 2017] in a way that is consistent with a Galerkin-style weak form discretization of the governing equations. Additionally, it enables a new stress divergence discretization that effortlessly allows all MPM simulations to run two times faster than before. We also develop a Compatible Particle-In-Cell (CPIC) algorithm on top of MLS-MPM. Utilizing a colored distance field representation and a novel compatibility condition for particles and grid nodes, our framework enables the simulation of various new phenomena that are not previously supported by MPM, including material cutting, dynamic open boundaries, and two-way coupling with rigid bodies. MLS-MPM with CPIC is easy to implement and friendly to performance optimization.

CCS Concepts: • **Computing methodologies → Physical simulation**;

Additional Key Words and Phrases: Material Point Method (MPM), moving least squares, cutting, discontinuity, distance field, rigid coupling

ACM Reference Format:
Yuanming Hu[†], Yu Fang[†], Ziheng Ge[†], Ziyin Qu, Yixin Zhu[†], Andre Pradhana, and Chenfanfu Jiang. 2018. A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling. *ACM Trans. Graph.* 37, 4, Article 150 (August 2018), 14 pages. https://doi.org/10.1145/3197517.3201293

1 INTRODUCTION

Since the pioneering work of Terzopoulos et al. [1988], simulating topologically changing materials has been a popular research topic in graphics. Among various topics, fracture and cutting of deformable objects are most intensively explored. The goal of breaking mesh connectivity has led to techniques such as local remeshing [O'Brien et al. 2002; O'Brien and Hodgins 1999], the Virtual Node Algorithm (VNA) [Hegemann et al. 2013; Molino et al. 2005; Wang et al. 2014] and the eXtended Finite Element Method (XFEM) [Koschier et al. 2017]. Maintaining remeshing quality efficiently and robustly can be complicated. While VNA and XFEM reduce some difficulty, they impose additional challenges like floating point arithmetic in degenerate scenarios and self-collision on embedded surfaces.

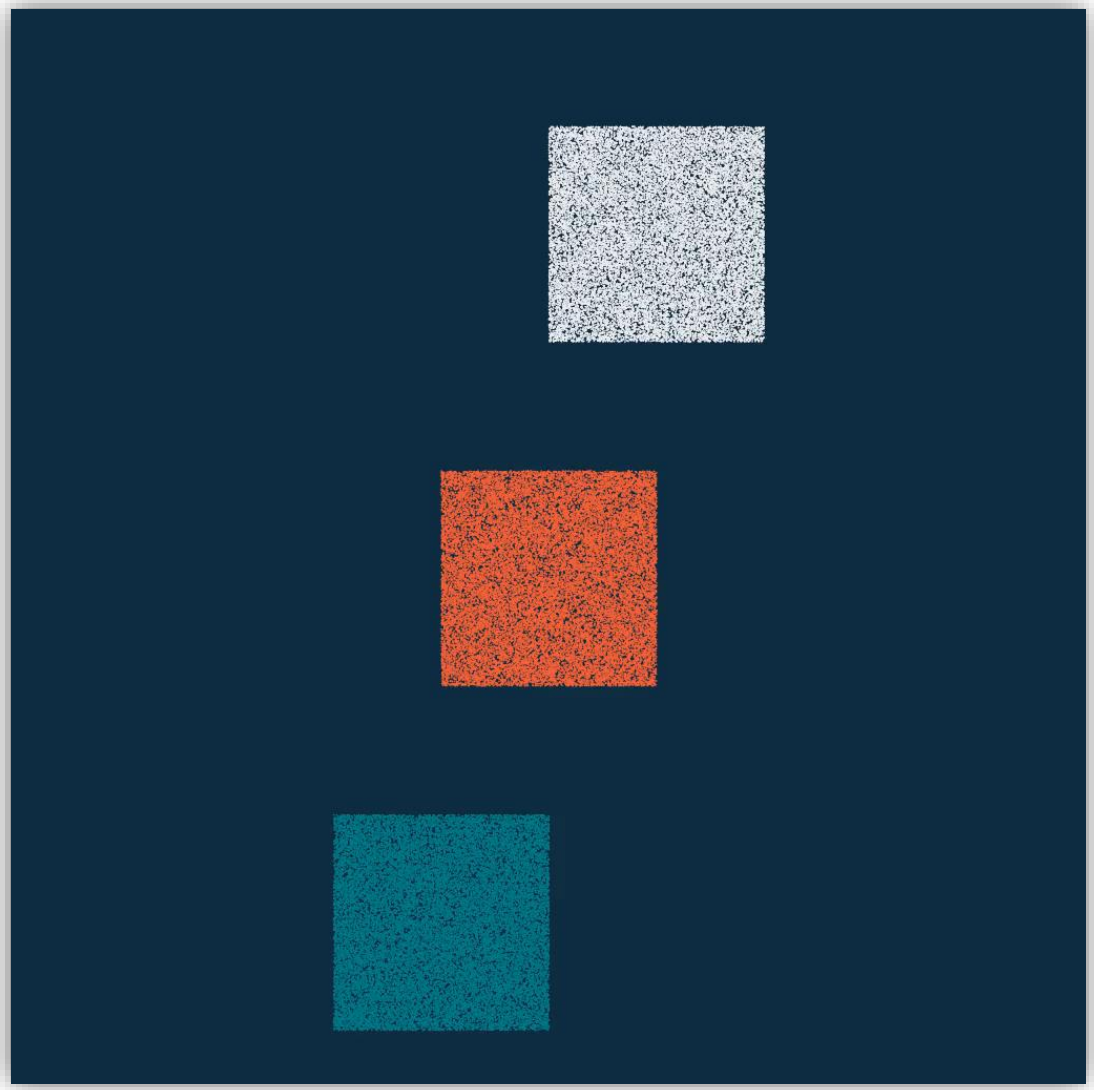
Compared to mesh-based approaches, meshless animation of solid topology change was shown to be promising by Pauly et al. [2005]. More recently, the Material Point Method (MPM) [Sulsky et al. 1995]

Authors' addresses: Yuanming Hu[†], MIT CSAIL, yuanming@mit.edu; Yu Fang[†], Tsinghua University, squarefk@gmail.com; Ziheng Ge[†], University of Science and Technology of China, gzh1057@mail.ustc.edu.cn; Ziyin Qu, University of Pennsylvania, ziyinq@seas.upenn.edu; Yixin Zhu[†], University of California, Los Angeles, yixin.zhu@ucla.edu; Andre Pradhana, University of Pennsylvania, apradh@seas.upenn.edu; Chenfanfu Jiang, University of Pennsylvania, cffjiang@seas.upenn.edu.

0730-0301/2018/8-ART150 $15.00
https://doi.org/10.1145/3197517.3201293

† Y. Hu, Y. Fang, Z. Ge and Y. Zhu were visiting students at the University of Pennsylvania during this work.

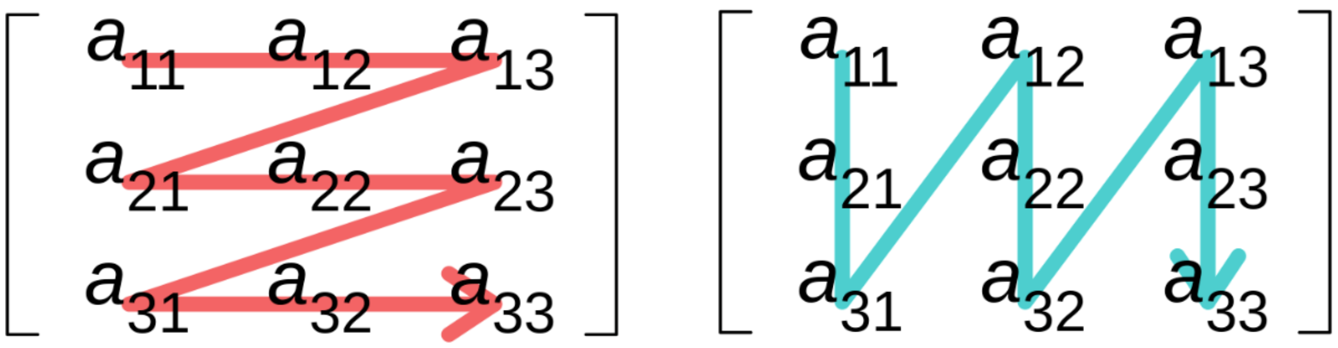
ACM Trans. Graph., Vol. 37, No. 4, Article 150. Publication date: August 2018.



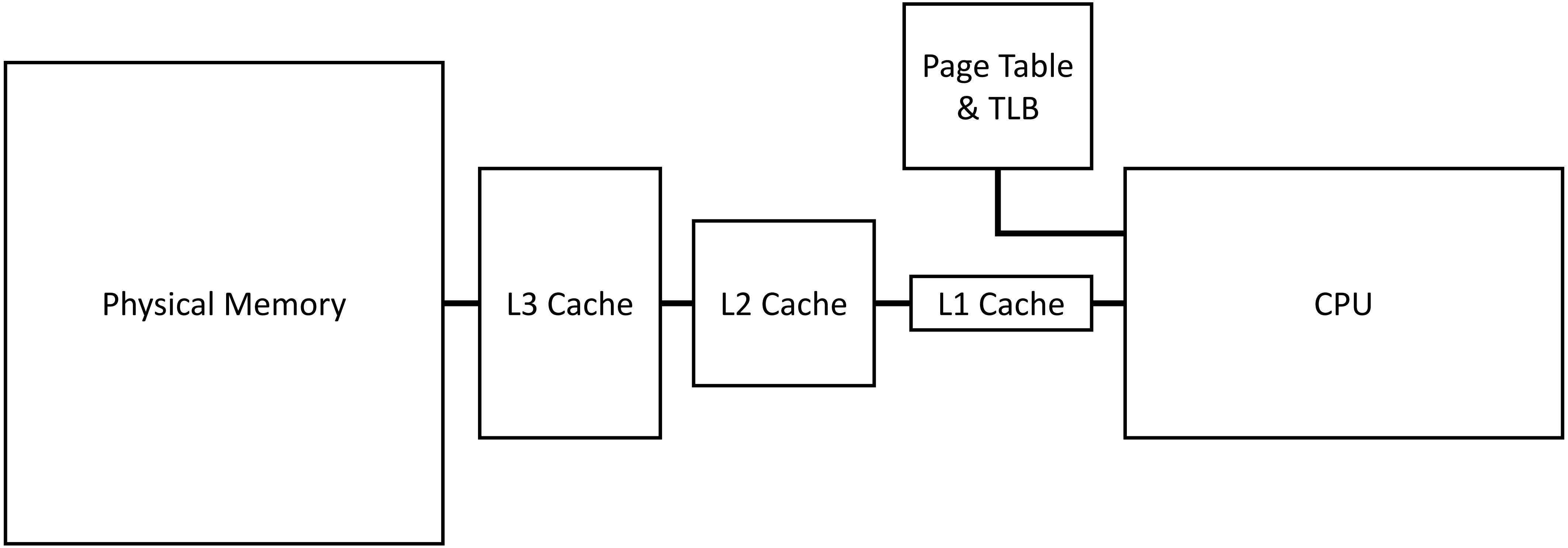
Remark

- The most useful data container:
  - ti.field
- Functions taken over by the Taichi compiler:
  - decorated with @ti.kernel or @ti.func
- For-loops
  - The outer-most for-loop in a @ti.kernel is parallelized
  - Types of for-loops: range-for and struct-for

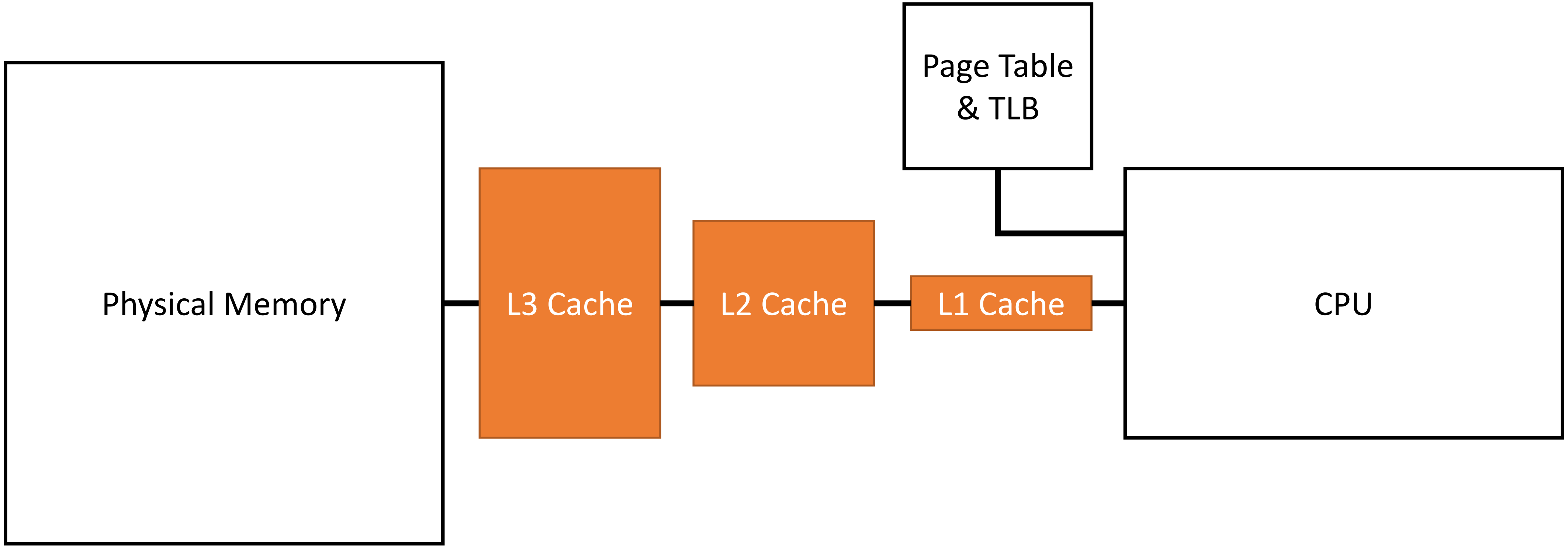
Efficient Dense Data Layouts



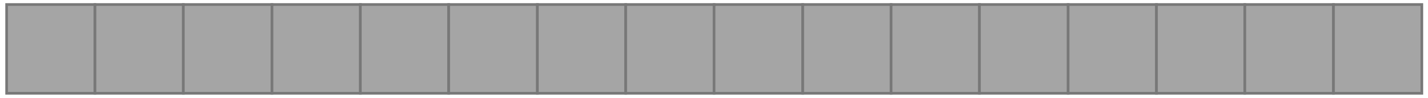
Still remember the memory hierarchy?



We want to keep our data cached

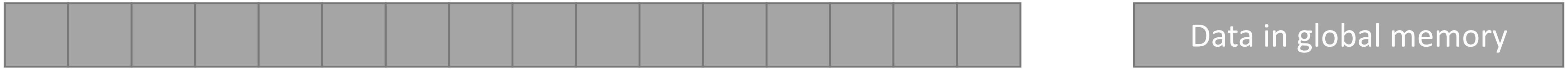


Accessing a 1-D array



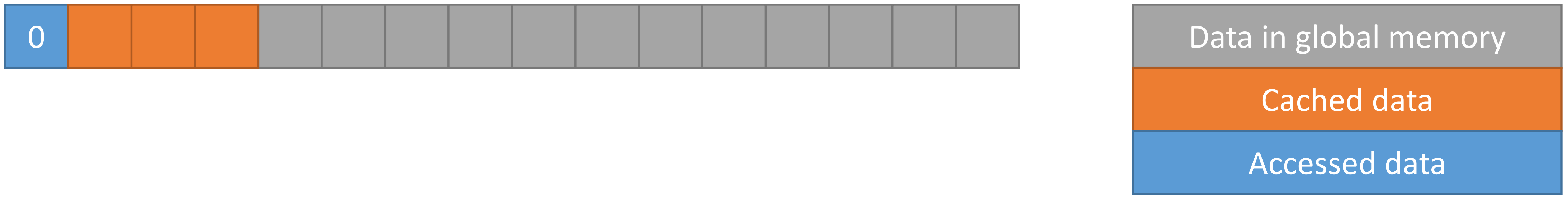
```
x = ti.field(ti.i32, shape=16)

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

fill()
```

Accessing a 1-D array




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

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

fill()
```

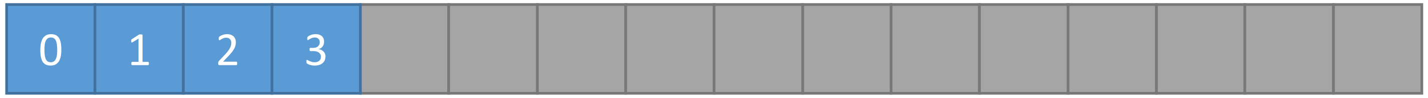
Accessing a 1-D array




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

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

fill()
```

(Assuming each the cache line is 16 Bytes here)

Accessing a 1-D array



Data in global memory

Cached data

Accessed data

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

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

fill()
```

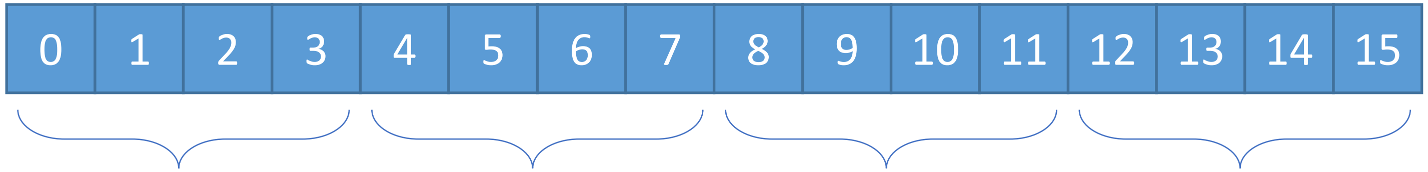
(Assuming each the cache line is 16 Bytes here)

Accessing a 1-D array


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

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

fill()
```

(Assuming each the cache line is 16 Bytes here)

Accessing a 1-D array



4 coalesced global memory access

Data in global memory

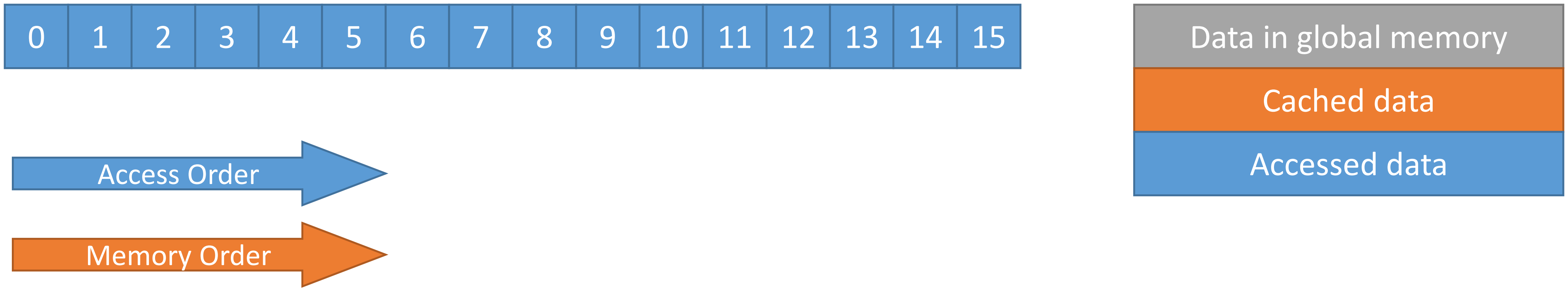
Cached data

Accessed data

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

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

fill()
```

(Assuming each the cache line is 16 Bytes here)

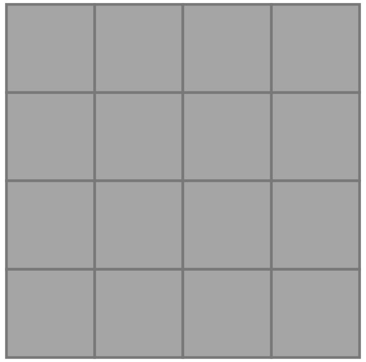
The access order aligns with the memory order ☺




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

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

fill()
```

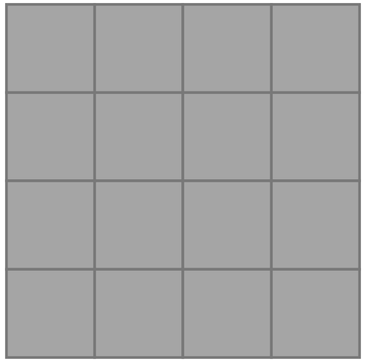
(Assuming each the cache line is 16 Bytes here)

How about multi-dimensional arrays?

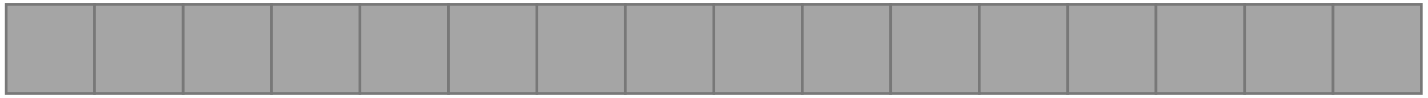


```
x = ti.field(ti.i32, shape = (4, 4))

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

fill()
```

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

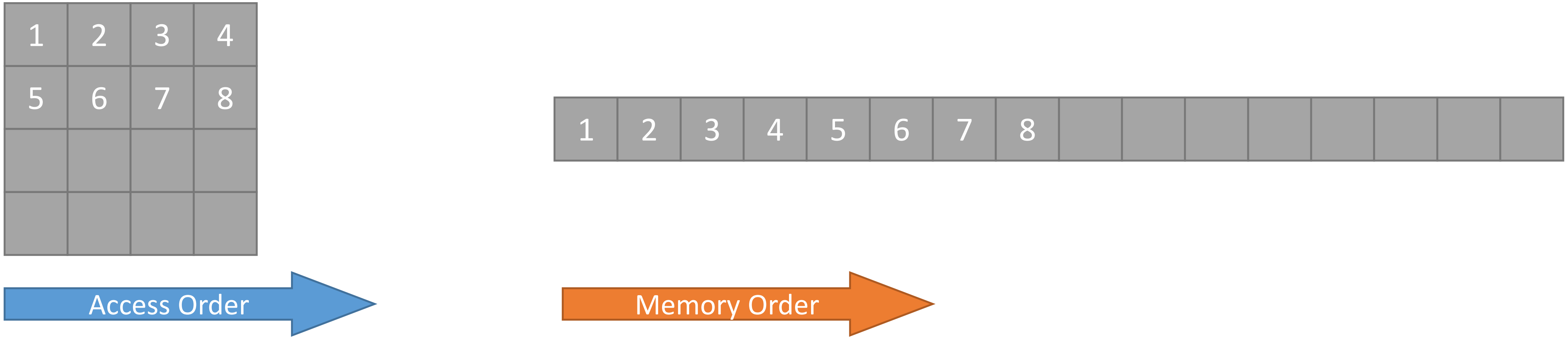


An N-D array
we think

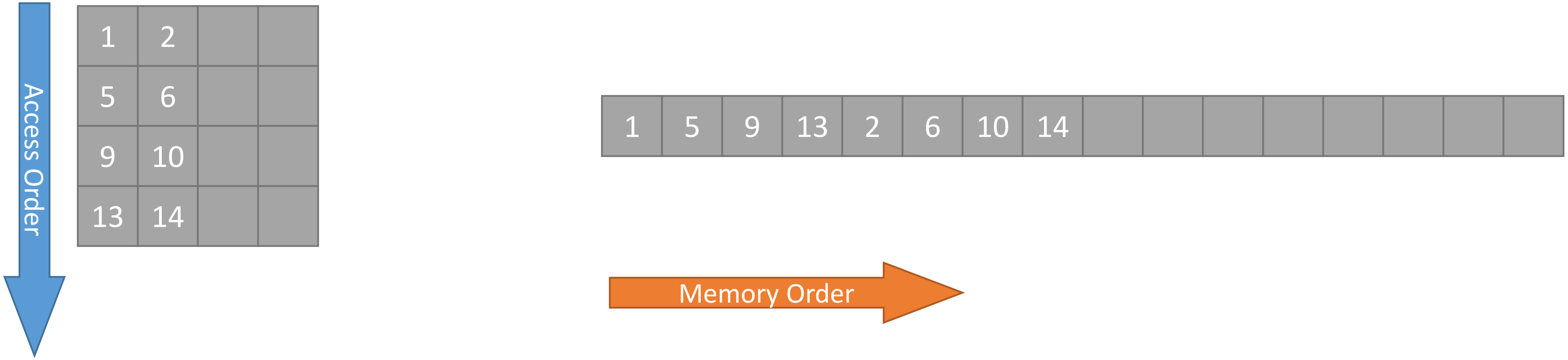


An N-D array
we store

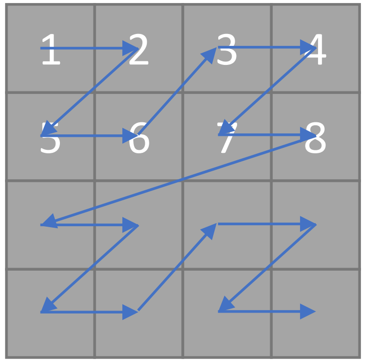
Ideal memory layout of an N-D field:



Ideal memory layout of an N-D field:

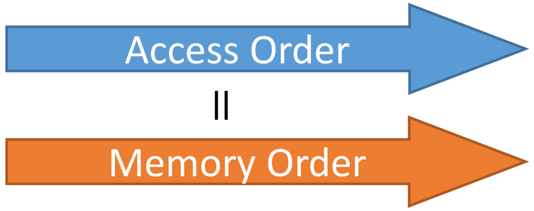


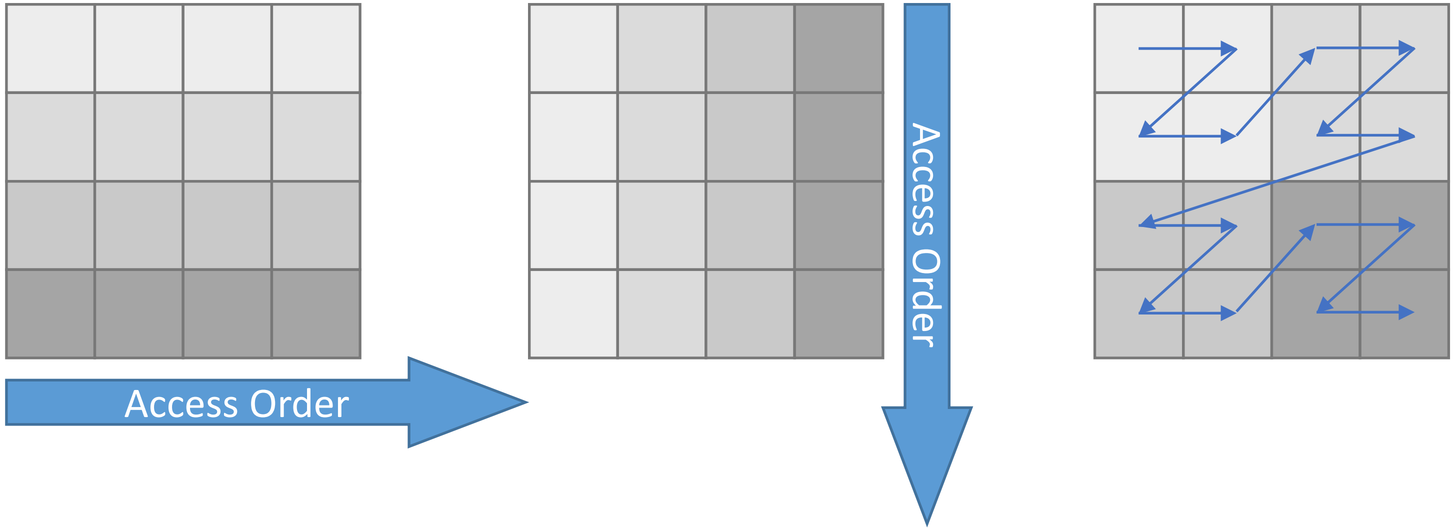
Ideal memory layout of an N-D field:



What we want:

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

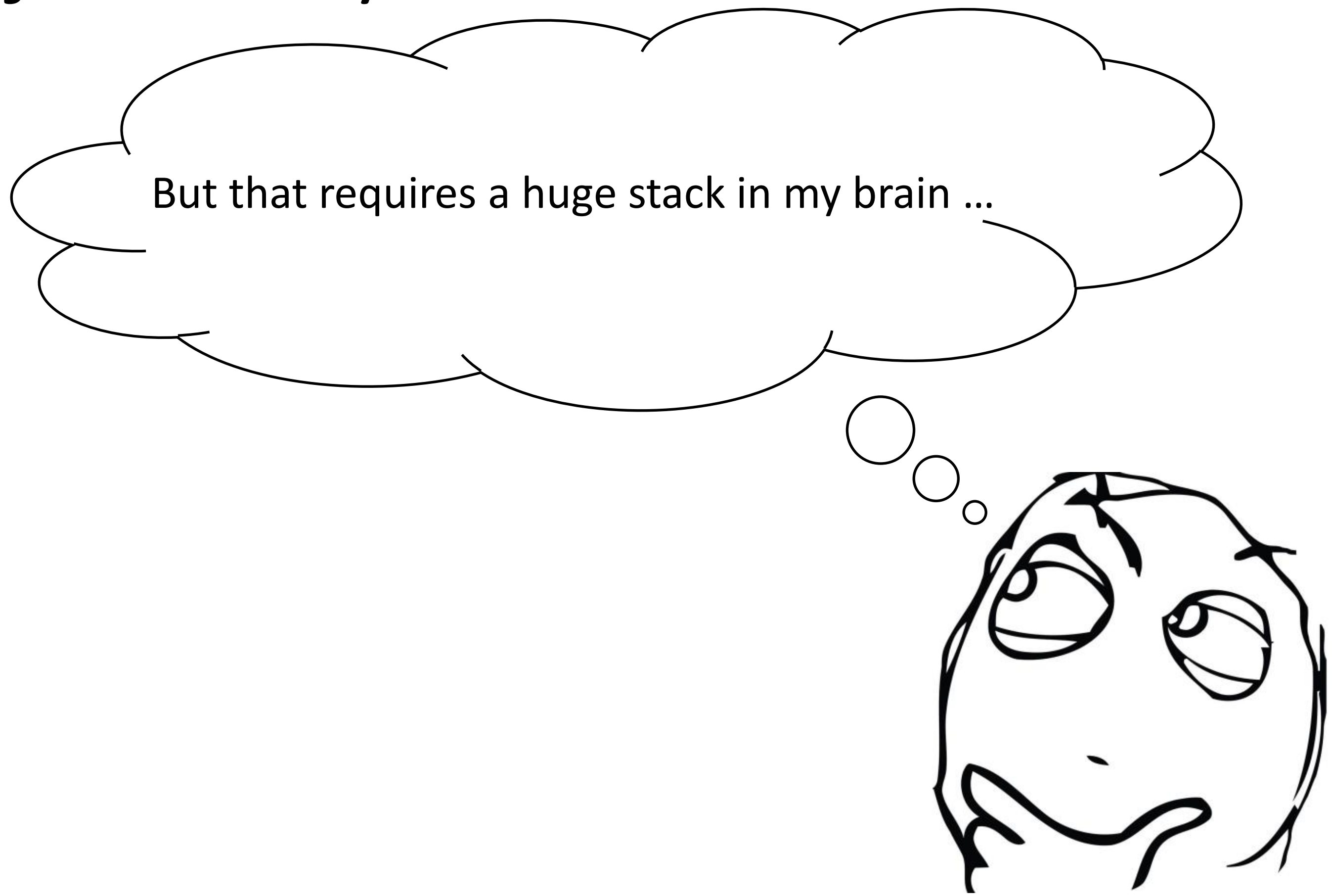




Access row/col-major arrays in C/C++

```
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)
```

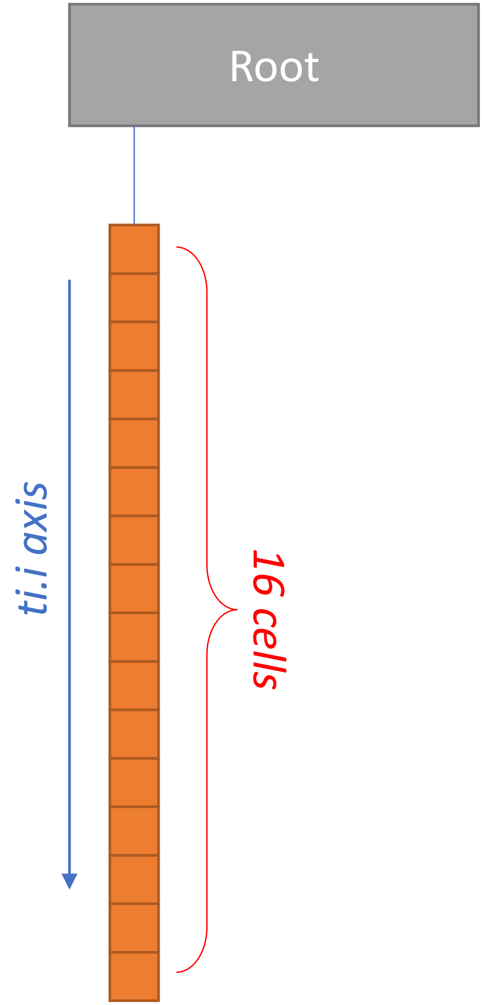
ti.root In English:
Each cell of *root* has a *dense* container with *16 cells* along the *ti.i axis*. Each cell of a *dense* container has *field x*

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*

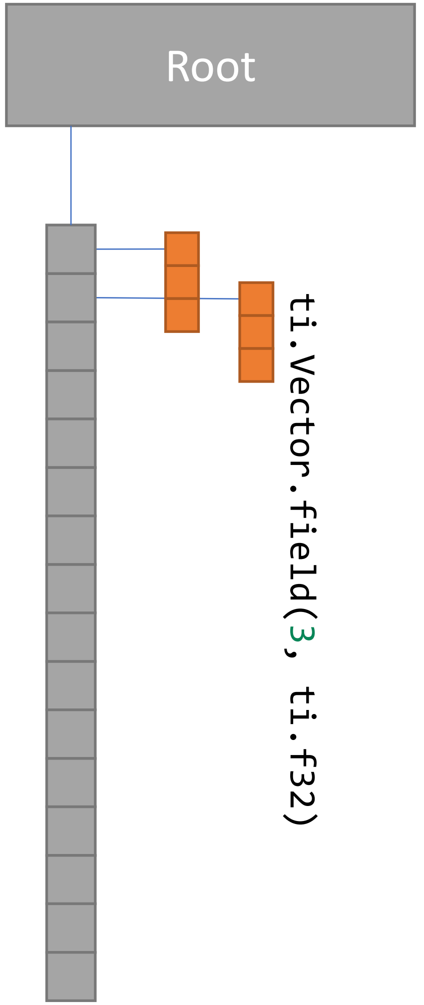


Layout 101: from *shape* to *ti.root*

```
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))
```

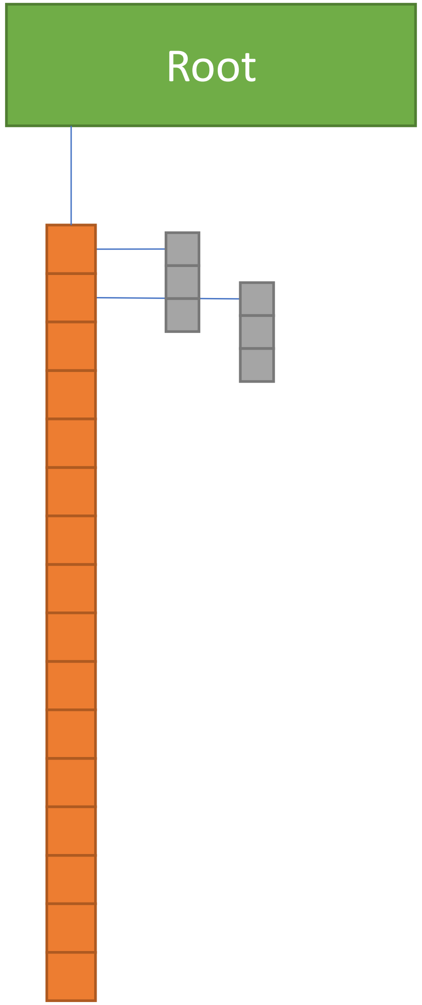
```
x = ti.field(ti.f32)
ti.root.dense(ti.ij, (3, 4)).place(x)
```

```
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)
```

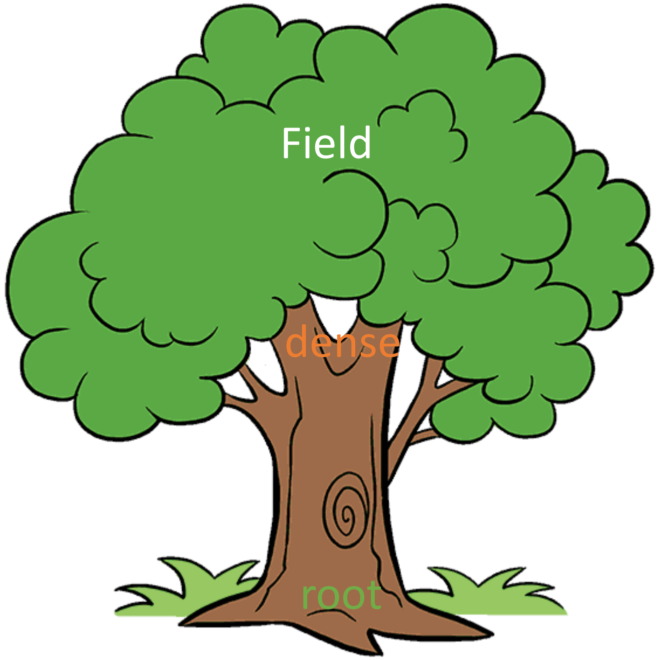
ti.root: the root of a *SNode-tree*

- 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
    - …

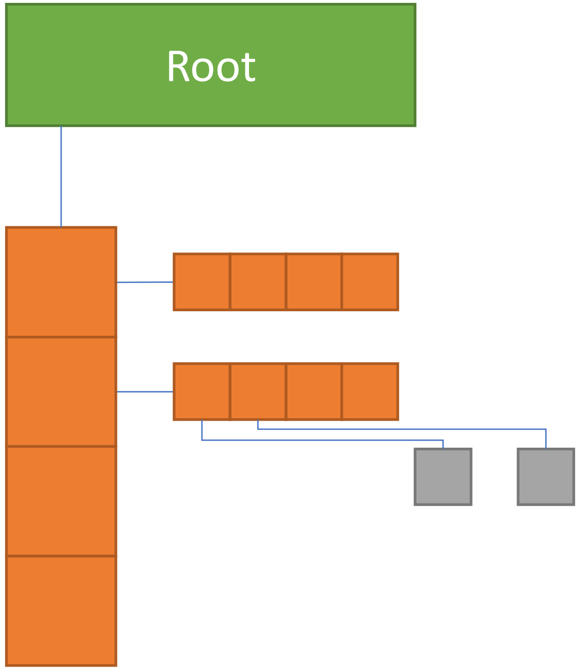


The SNode-tree

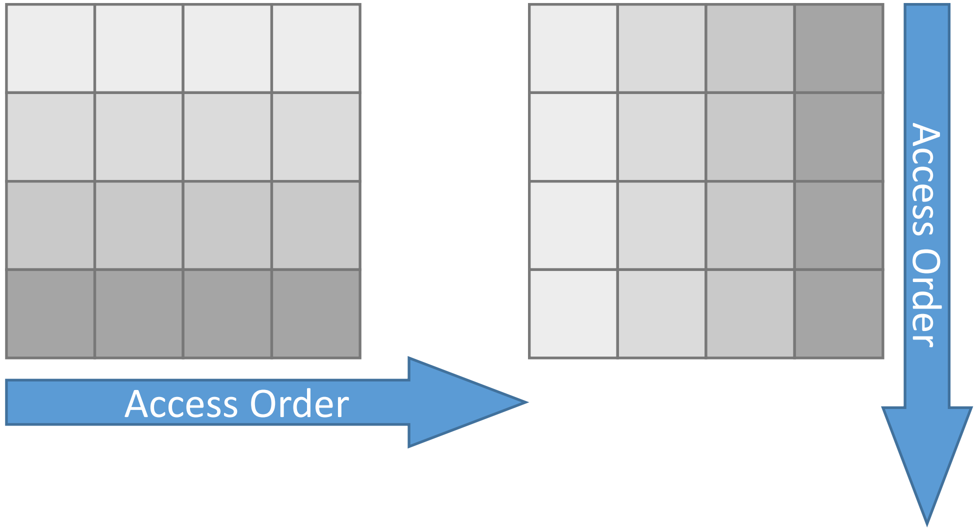
```
x = ti.field(ti.i32, shape = (4, 4))
```

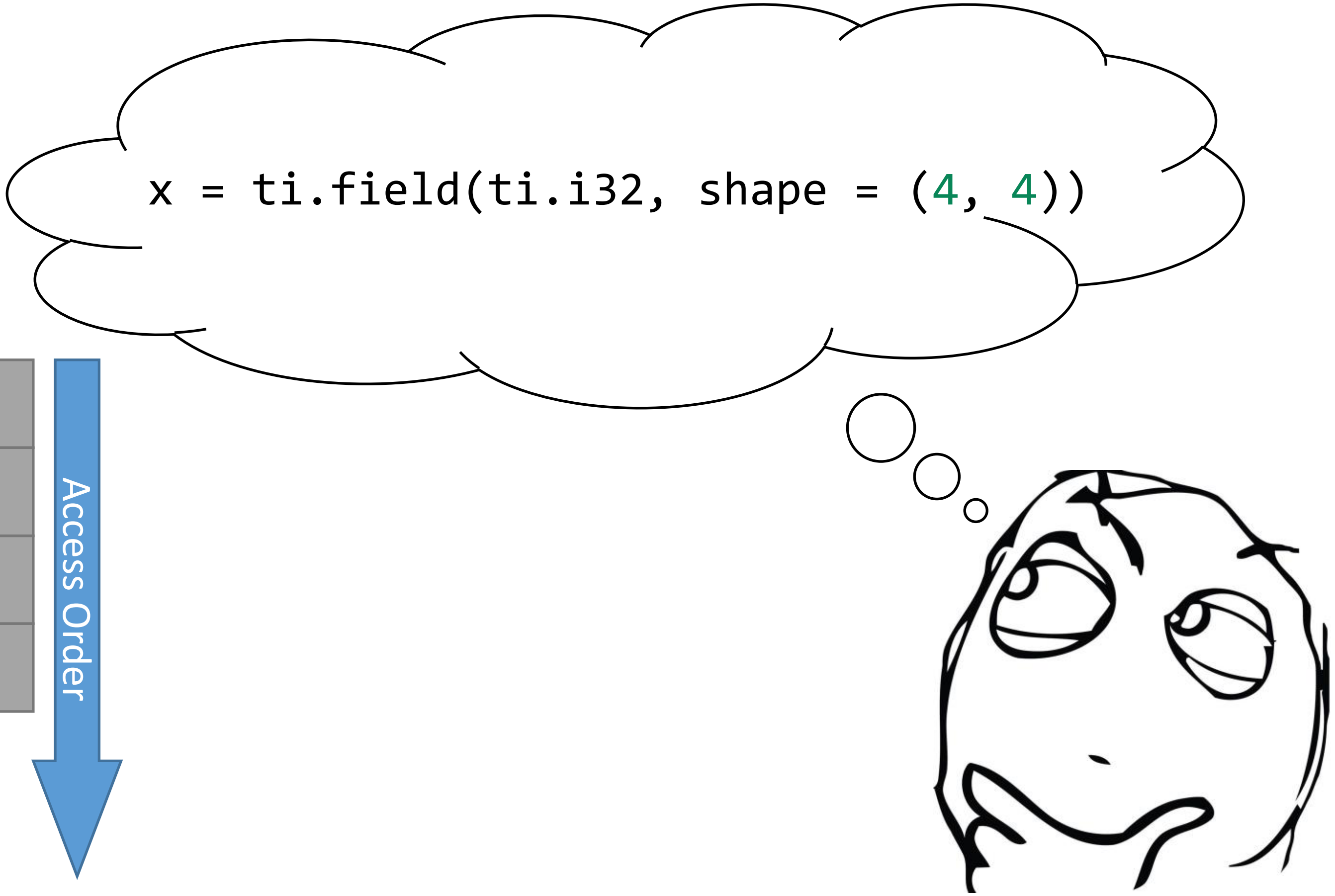
```
x = ti.field(ti.i32)
ti.root.dense(ti.ij, (4, 4)).place(x)
```

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



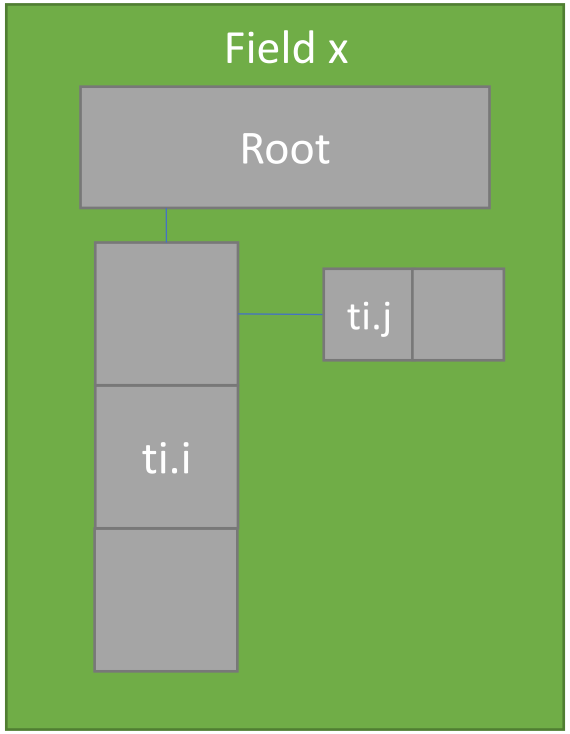
Row-major v.s. column-major



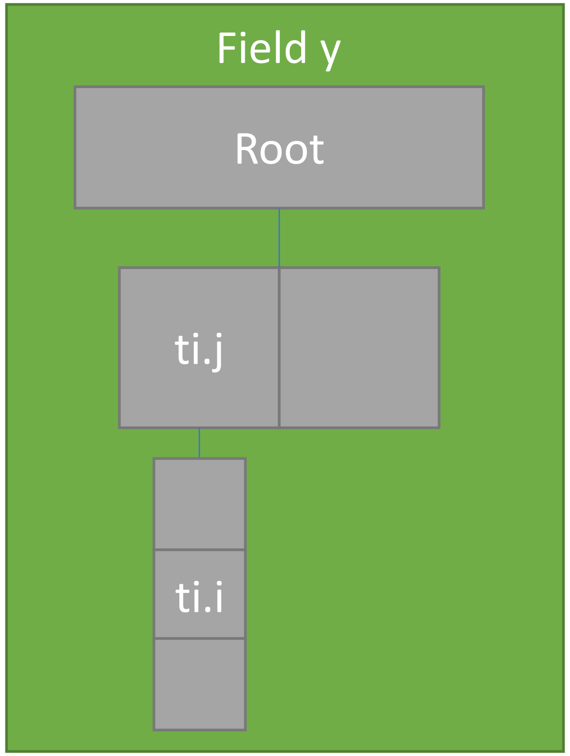


Row-major v.s. column-major

```
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]



Access row/col-major arrays

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

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

@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='', end=' ')

fill()
print_field()
```

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

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

@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='', end=' ')

fill()
print_field()
```

Access row/col-major arrays

SIGGRAPH ASIA 2022 DAEGU

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

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

@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='', end=' ')

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

@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='', end=' ')

fill()
print_field()
```

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

Access row/col-major arrays 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]);
        }
    }
}
```

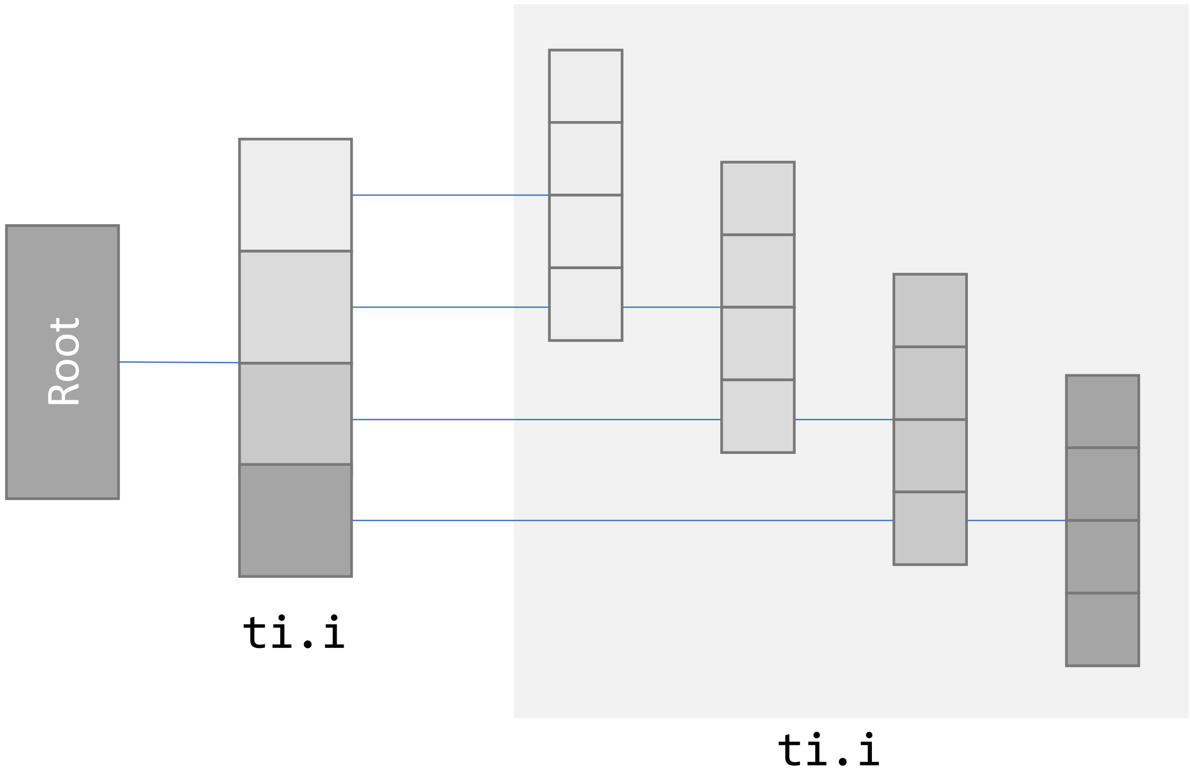
C/C++

```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

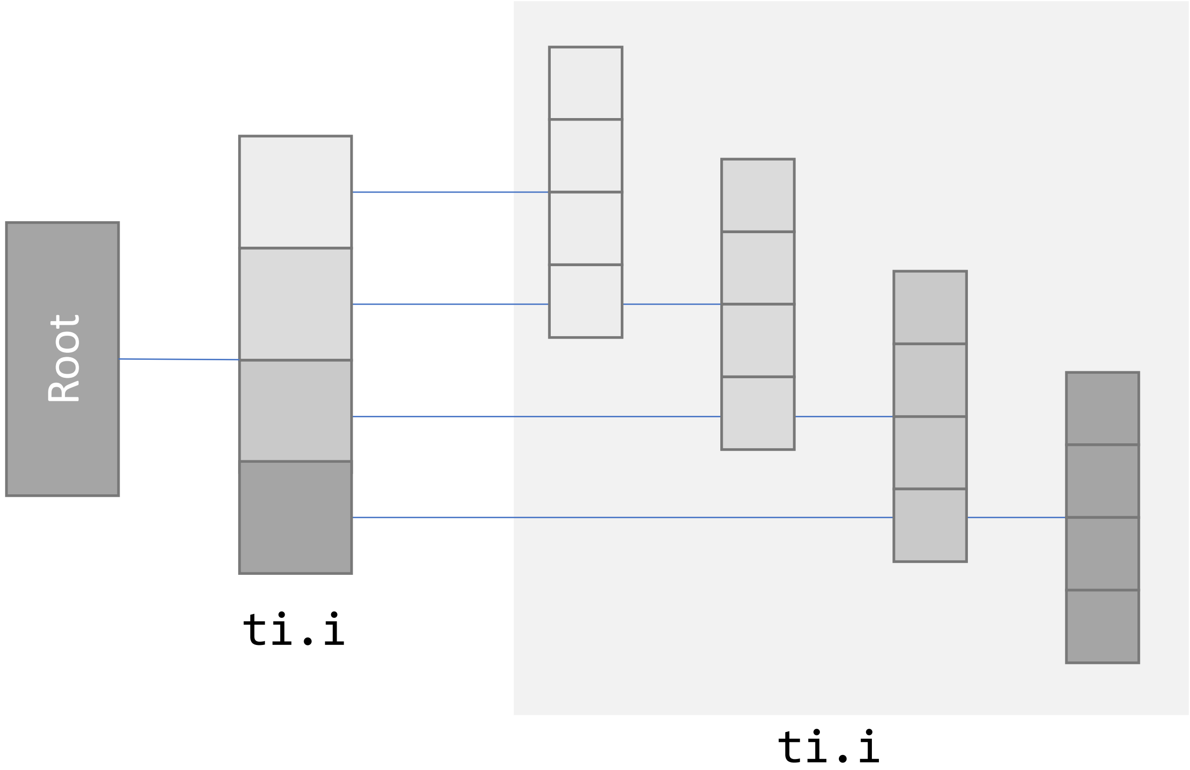
A special case:

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



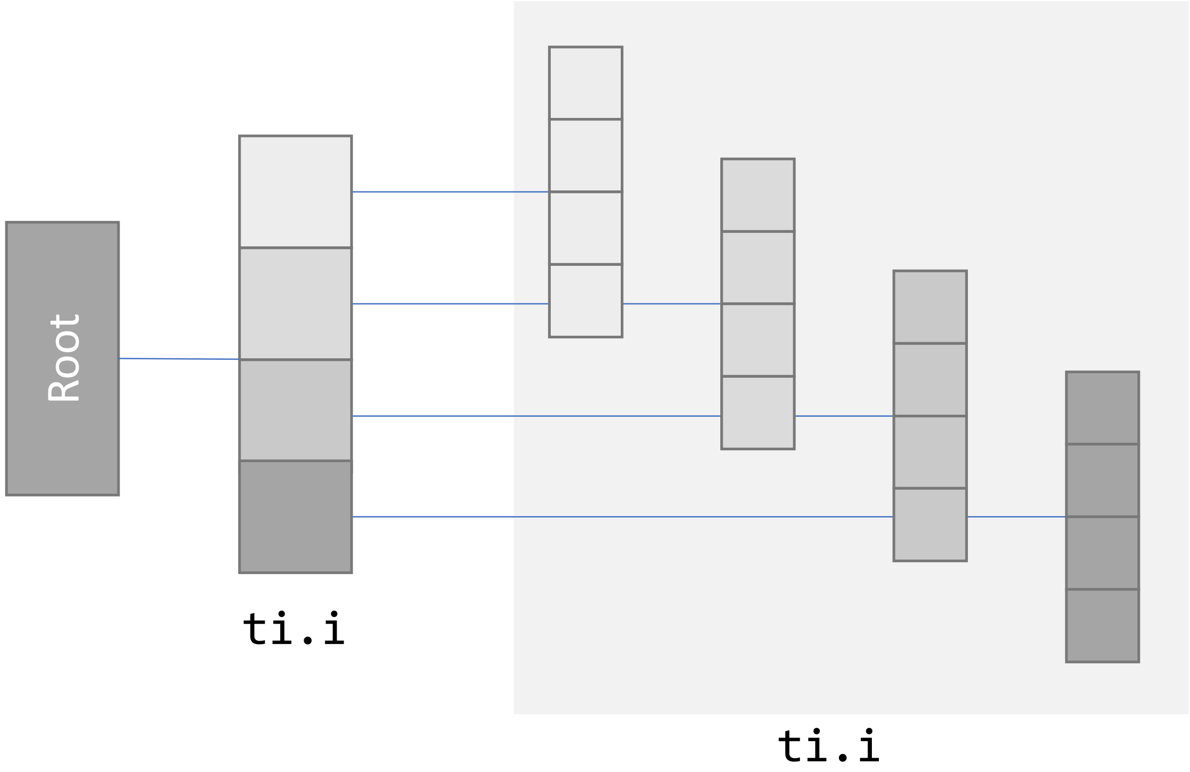
Hierarchical layouts

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



Hierarchical layouts

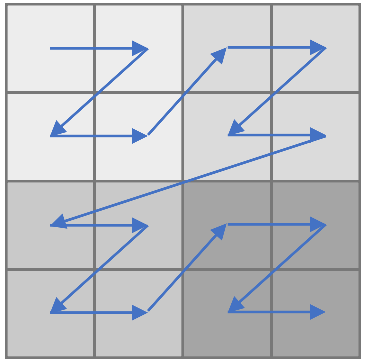
```
x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).dense(ti.i, 4).place(x)    # A hierarchical 1-D field
```

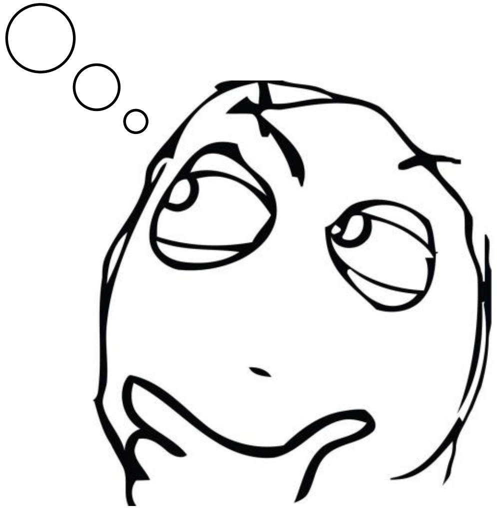


- Access like a 1-D field
- Store like a 2-D field (in blocks)

Z-order (block-major) access?

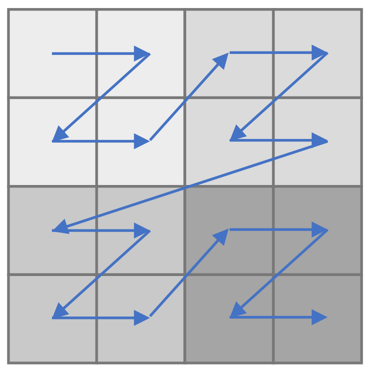
```
x = ti.field(ti.i32, shape = (4, 4))
```

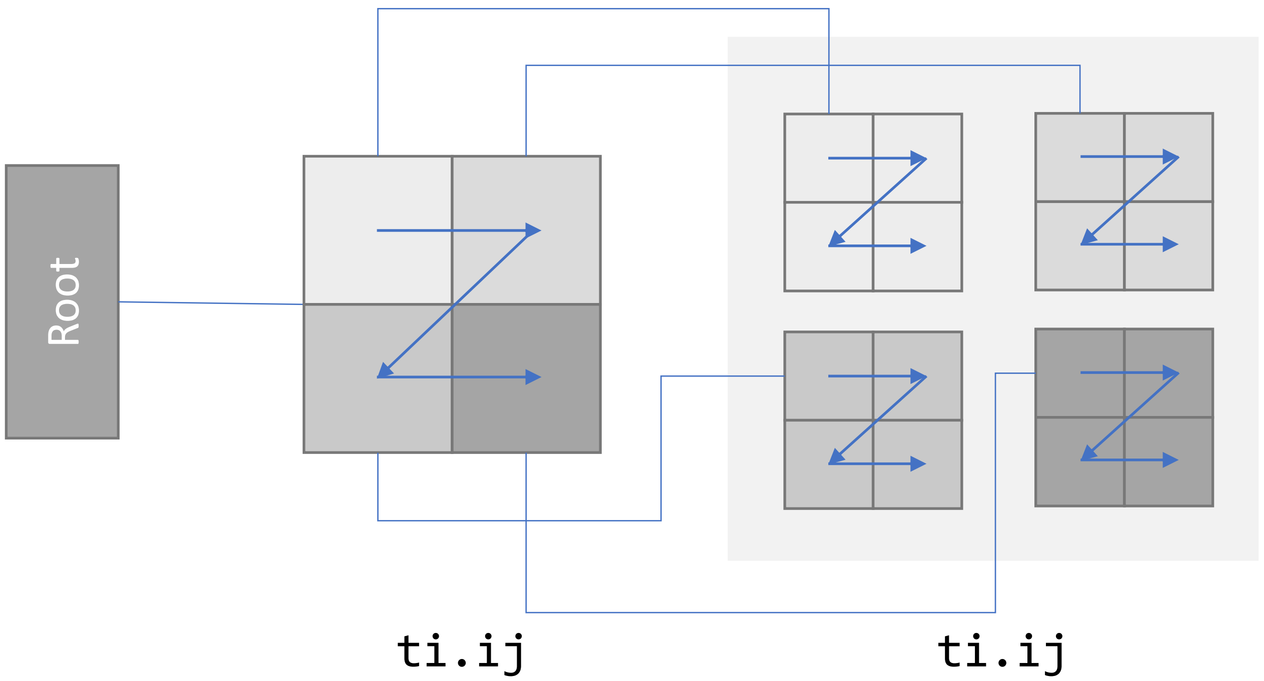




Z-order access using hierarchical layouts

```
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

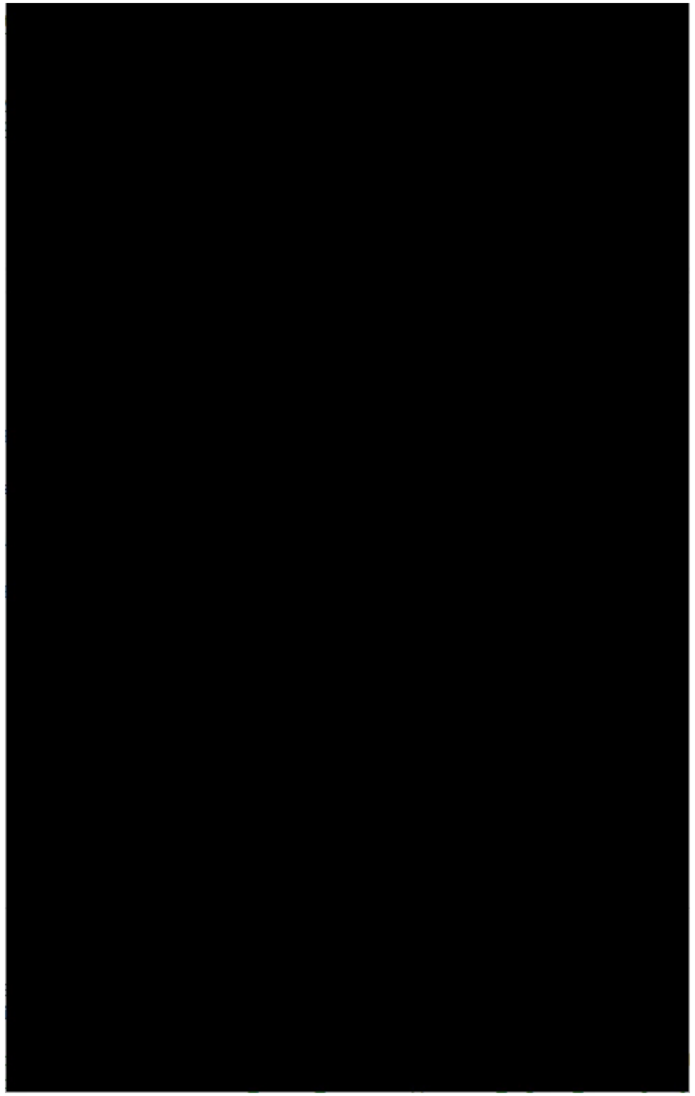
```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

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

@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.i***, then ***ti.j***

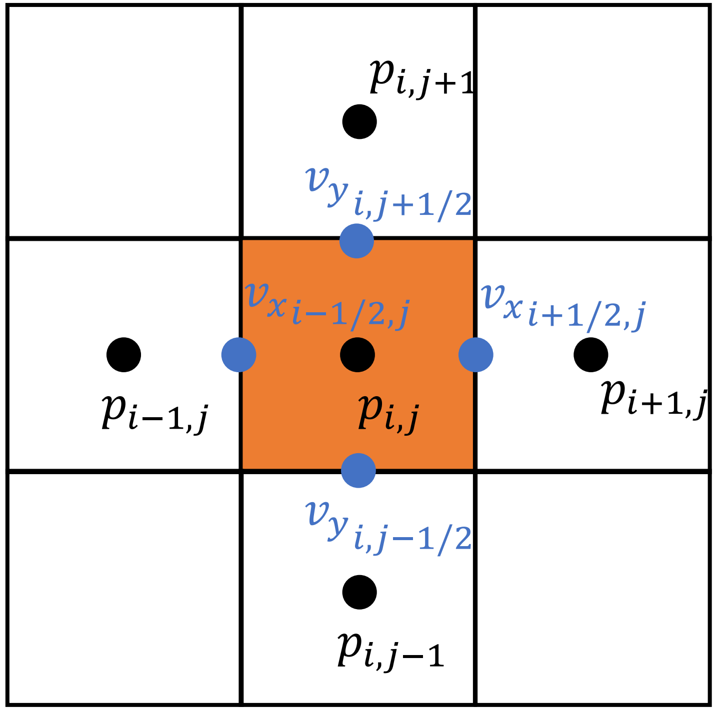
```python
import taichi as ti
ti.init(arch = ti.cpu, cpu_max_num_threads=1)

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

@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.ij***, then ***ti.ij,***
in 2x2 blocks

Example: a stable fluid simulation



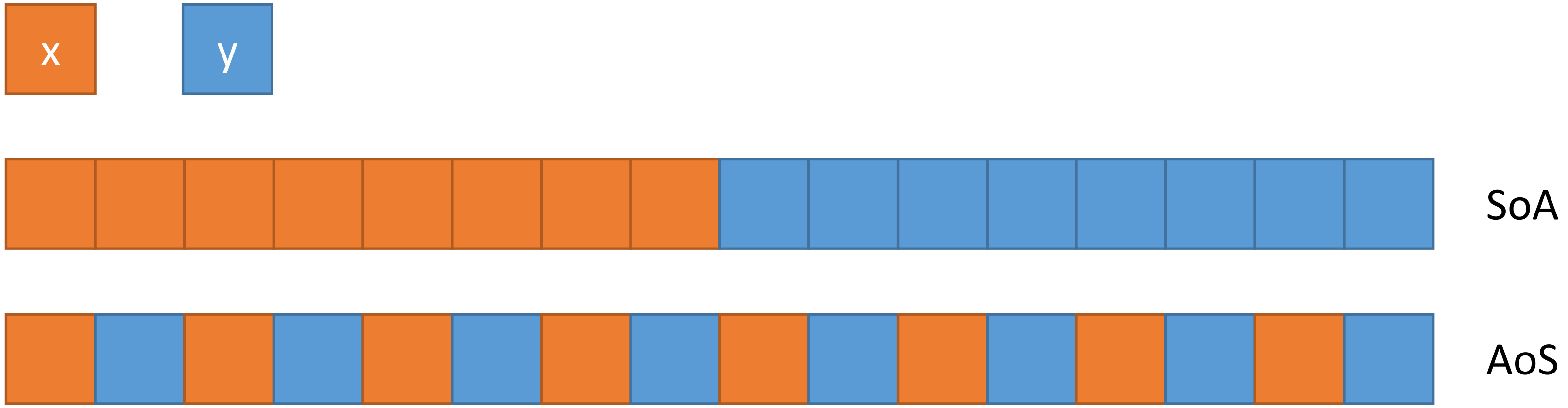
$$\frac{\Delta t}{\rho}\frac{4p_{i,j}-p_{i+1,j}-p_{i-1,j}-p_{i,j+1}-p_{i,j-1}}{\Delta x^2}$$
$$=-\frac{{v_x}^n_{i+1/2,j}-{v_x}^n_{i-1/2,j}+{v_y}^n_{i,j-1/2}-{v_y}^n_{i,j-1/2}}{\Delta x}$$



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?

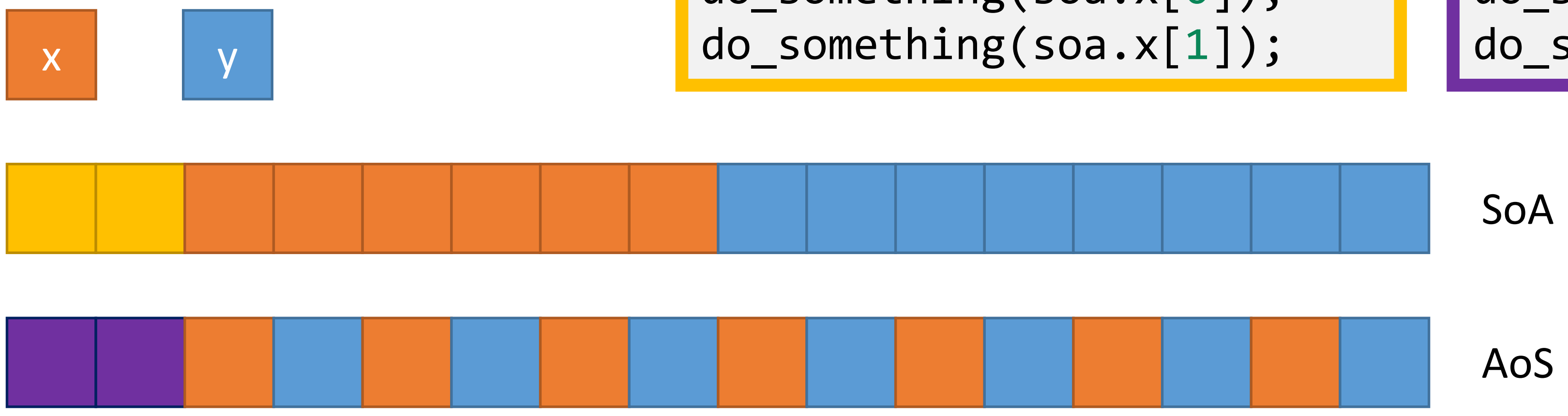
- It really depends…

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

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

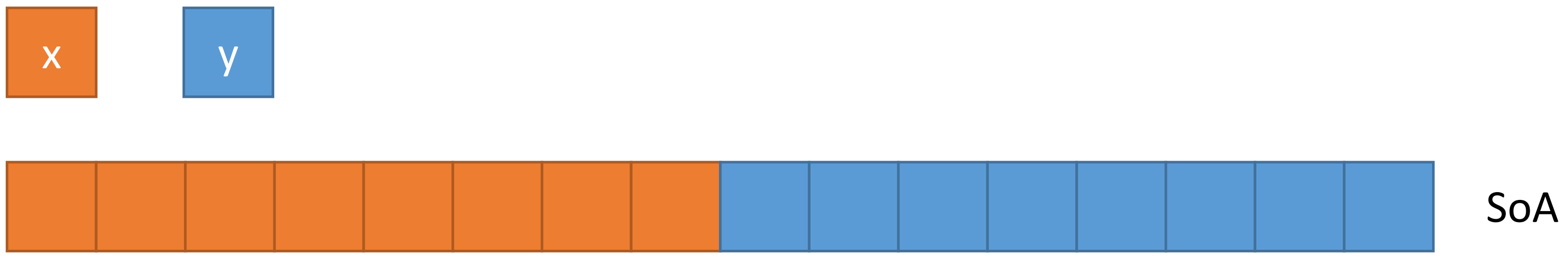
```
struct S2
{
    int x;
    int y;
}
S2 aos[8];
```

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



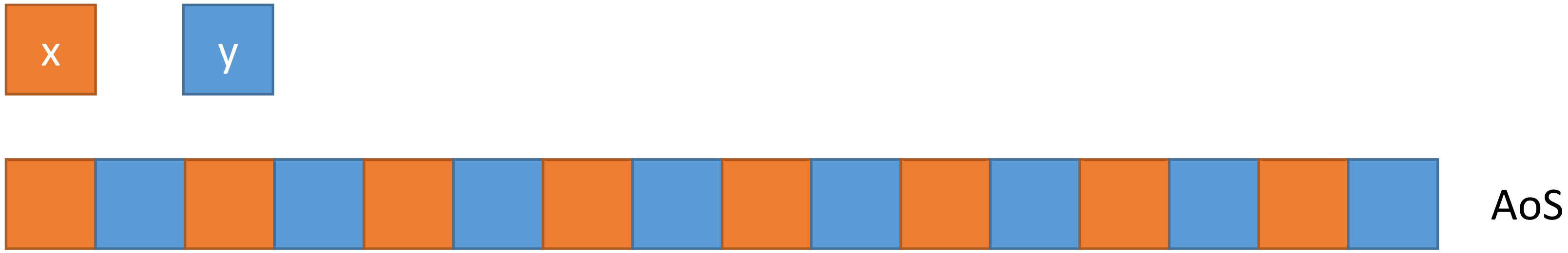
SoA in Taichi

```
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

```python
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]
```



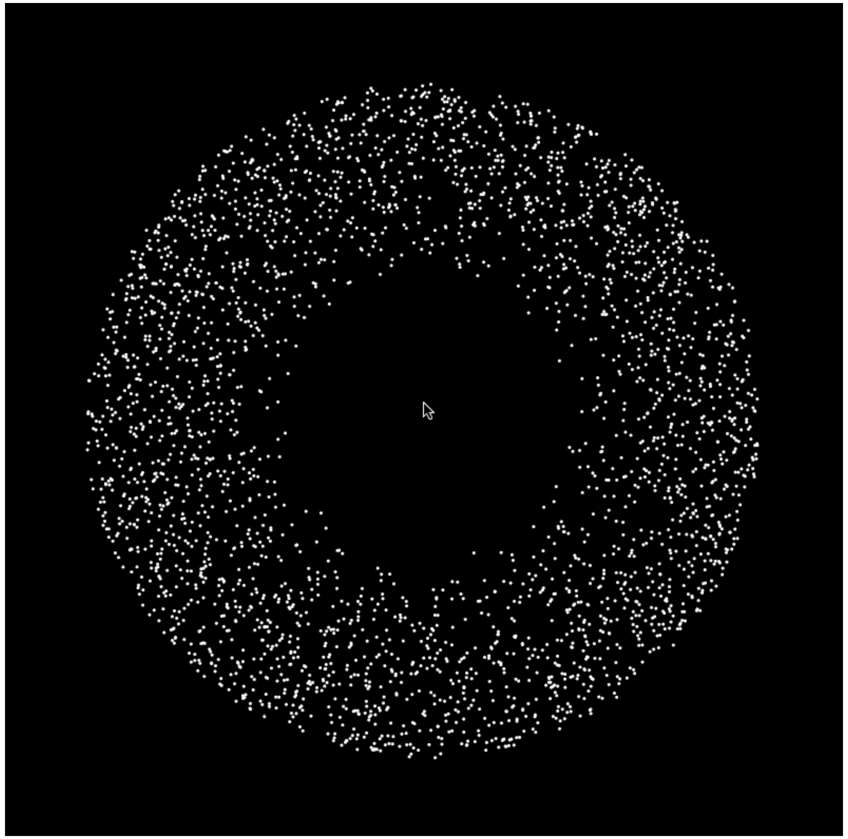
Switching between AoS and SoA in Taichi

```
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])
```

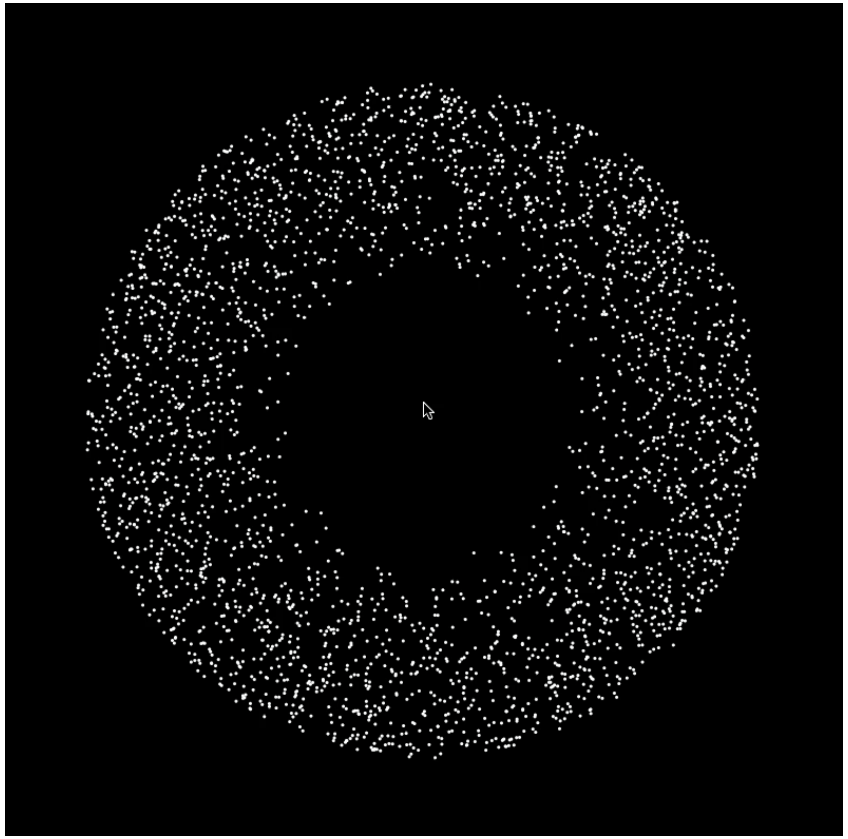
SoA Example, N-body:

```
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 pos:
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]
```



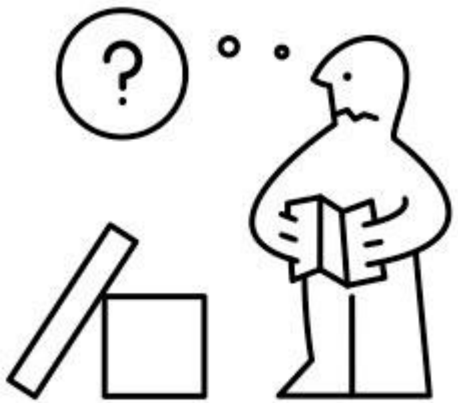
AoS Example, N-body:

```
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 pos:
        #symplectic euler
        vel[i] += dt*force[i]/m
        pos[i] += dt*vel[i]
```



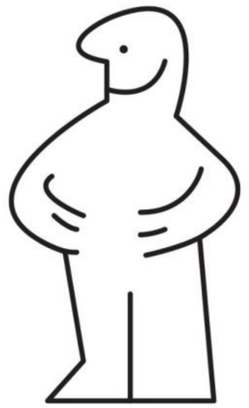
Tips for accessing advanced data layouts

- Tips?



Tips for accessing advanced data layouts

- No Tips!
  - You can access your advanced data layouts using struct-for(s) as if they were your old friend *ti.field()* defined with *shape*.



Remark: dense data layouts

- It is always preferred to align the memory order with access orders
- Taichi fields are Tree-structured: we call them SNode-trees
  - A SNode stands for "Structural Node"
- We 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)`
- We do not need to worry about the access of our data layouts
  - The Taichi struct-for handles it for us

Remark: dense data layouts

- It is always preferred to align the memory order with access orders
- Taichi fields are Tree-structured: we call them SNode-trees
  - A SNode stands for "Structural Node"
- We 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)`
- We do not need to worry about the access of our data layouts
  - The Taichi struct-for handles it for us

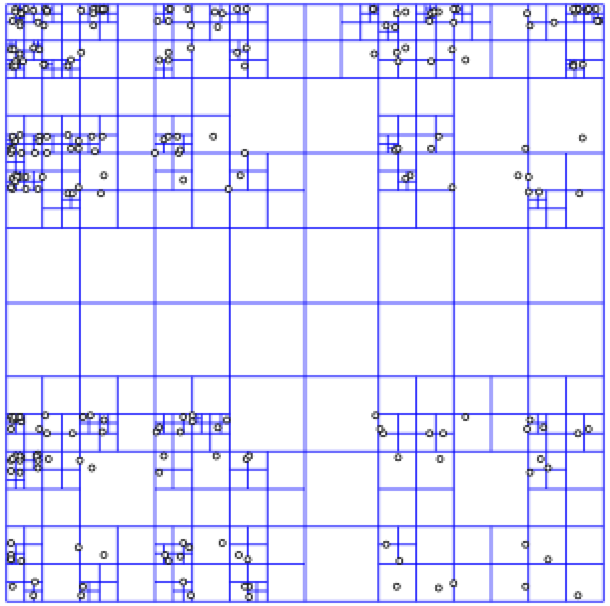
Remark: dense data layouts

- It is always preferred to align the memory order with access orders
- Taichi fields are Tree-structured: we call them SNode-trees
  - A SNode stands for “Structural Node”
- We 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)`
- We do not need to worry about the access of our data layouts
  - The Taichi struct-for handles it for us

Remark: dense data layouts

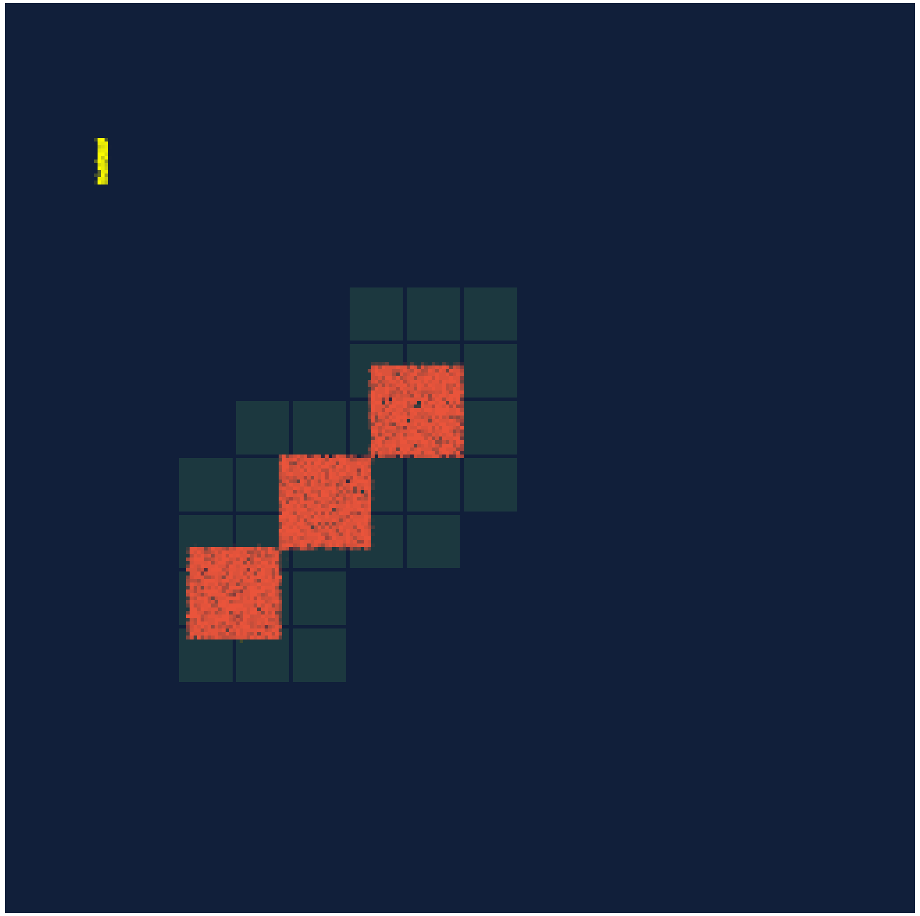
- It is always preferred to align the memory order with access orders
- Taichi fields are Tree-structured: we call them SNode-trees
  - A SNode stands for “Structural Node”
- We can append (multiple) dense cells to other dense cells
  - Row/col-major: `ti.root.dense(ti.i, N).dense(ti.j, M)`
  - Hierarchical layouts: `ti.root.dense(ti.i, N).dense(ti.i, M)`
  - SoA/AoS: `ti.root.dense(ti.i, N).place(x, y, z)`
- We do not need to worry about the access of our data layouts
  - The Taichi struct-for handles it for us

Spatially Sparse Data Structures



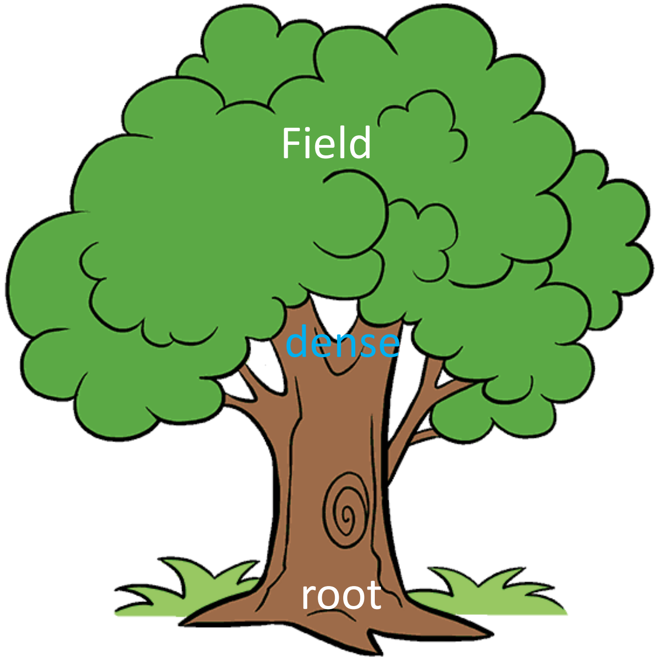
Sparse computation! but why?

- 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



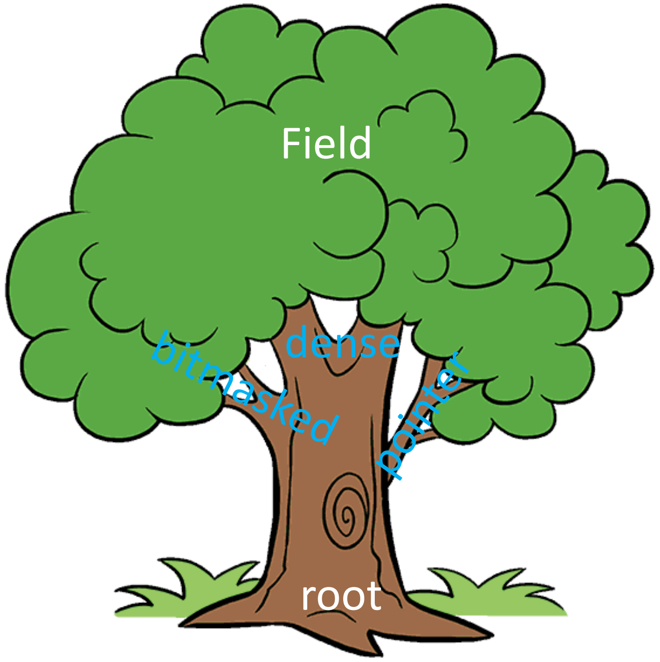
A SNode-tree

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



A SNode-tree

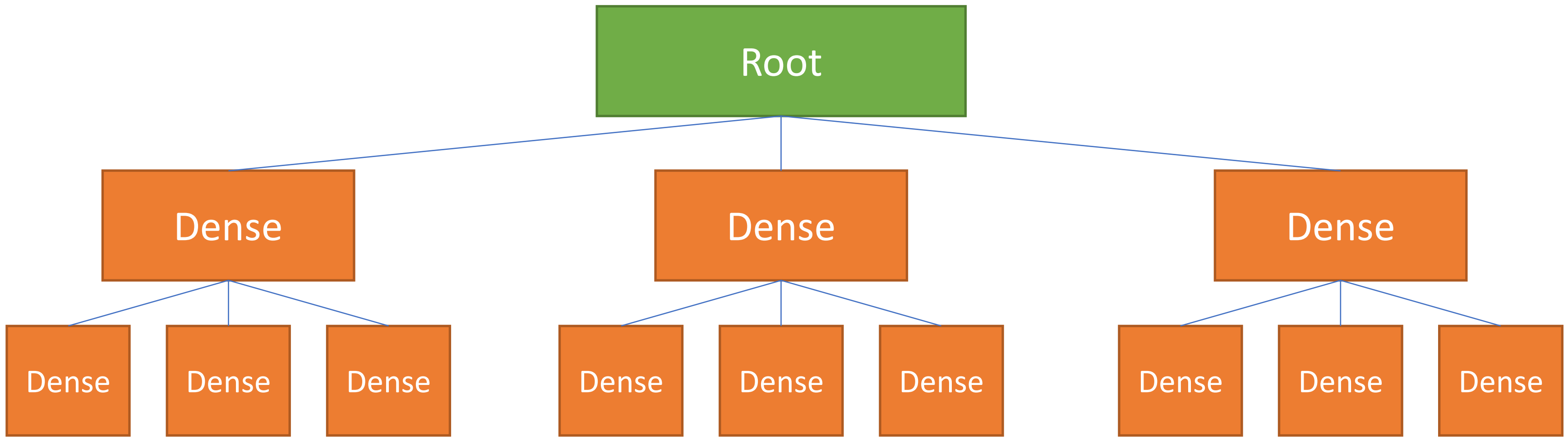
- root: the root of the data structure
- dense: a fixed-length contiguous array.
- bitmasked: similar to dense, but it also uses a mask to maintain ***sparsity*** information, one bit per child.
- pointer: stores pointers instead of the whole structure to save memory and maintain ***sparsity***



Let's start with a dense SNode tree

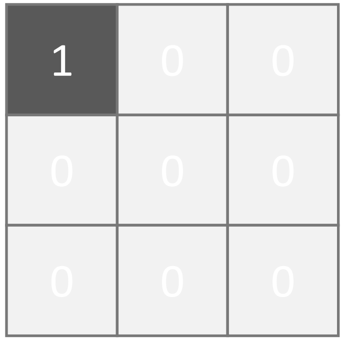
- A dense SNode-tree:

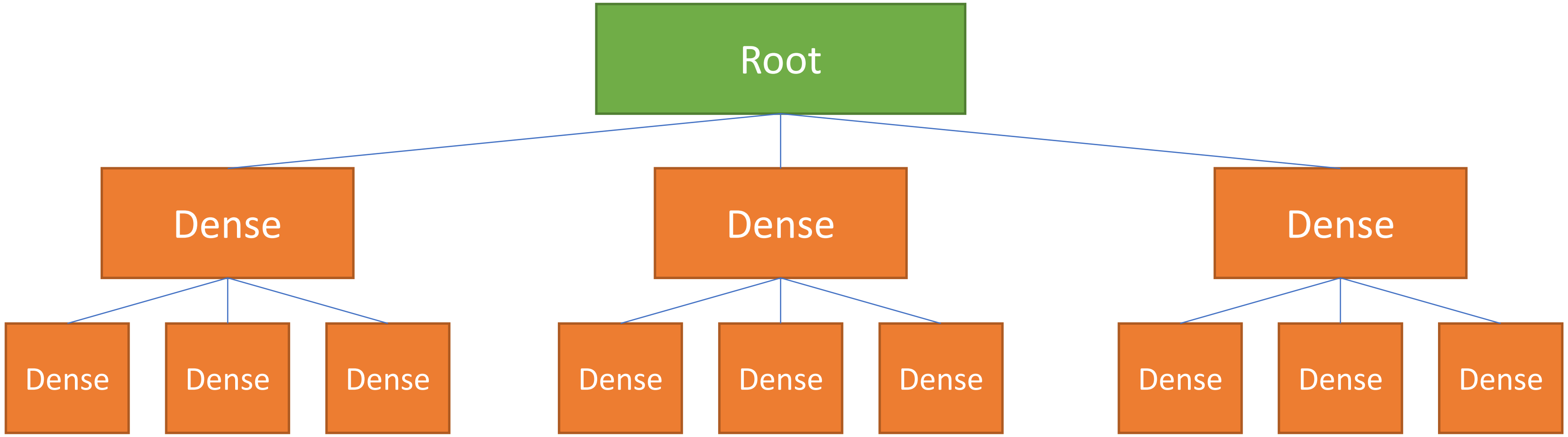
```
x = ti.field(ti.i32)
block1 = ti.root.dense(ti.i, 3)
block2 = block1.dense(ti.j, 3)
block2.place(x)
# equivalent to ti.root.dense(ti.i,3)
.dense(ti.j,3).place(x)
```



… and fill it with a single non-zero element

- A dense SNode-tree:

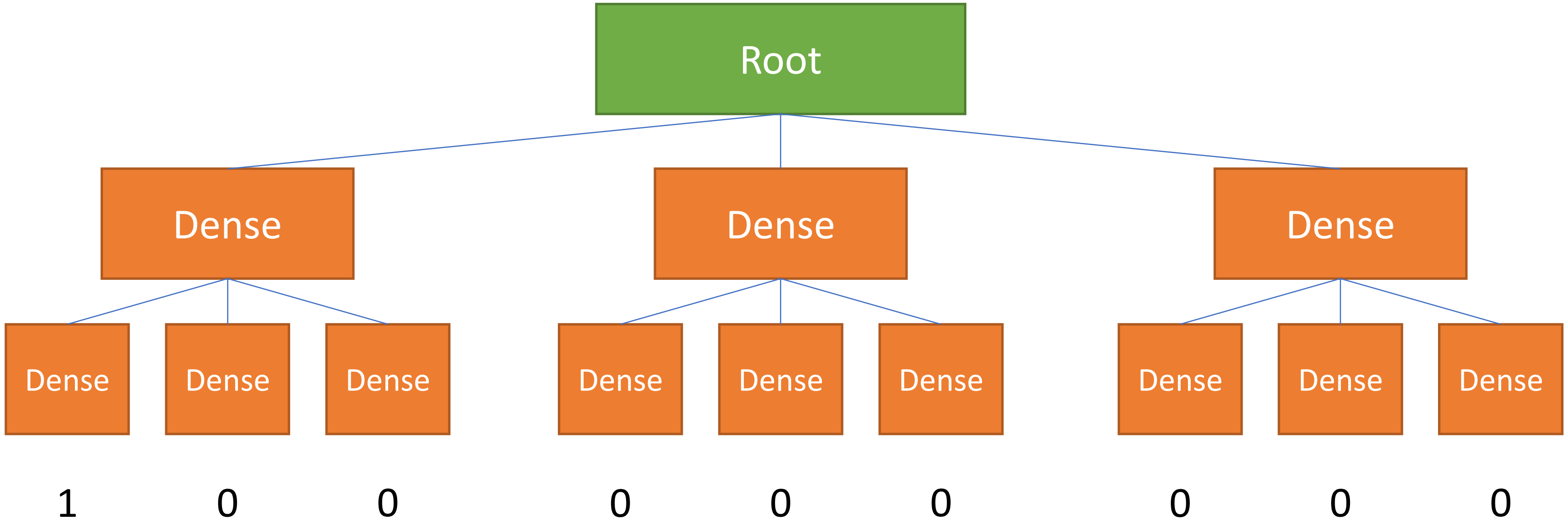




… and fill it with a single non-zero element

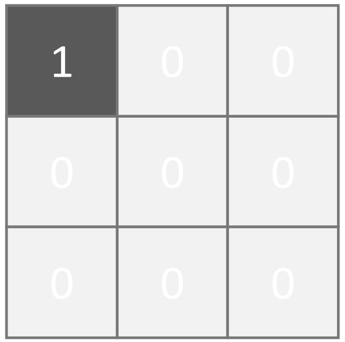
- A dense SNode-tree:

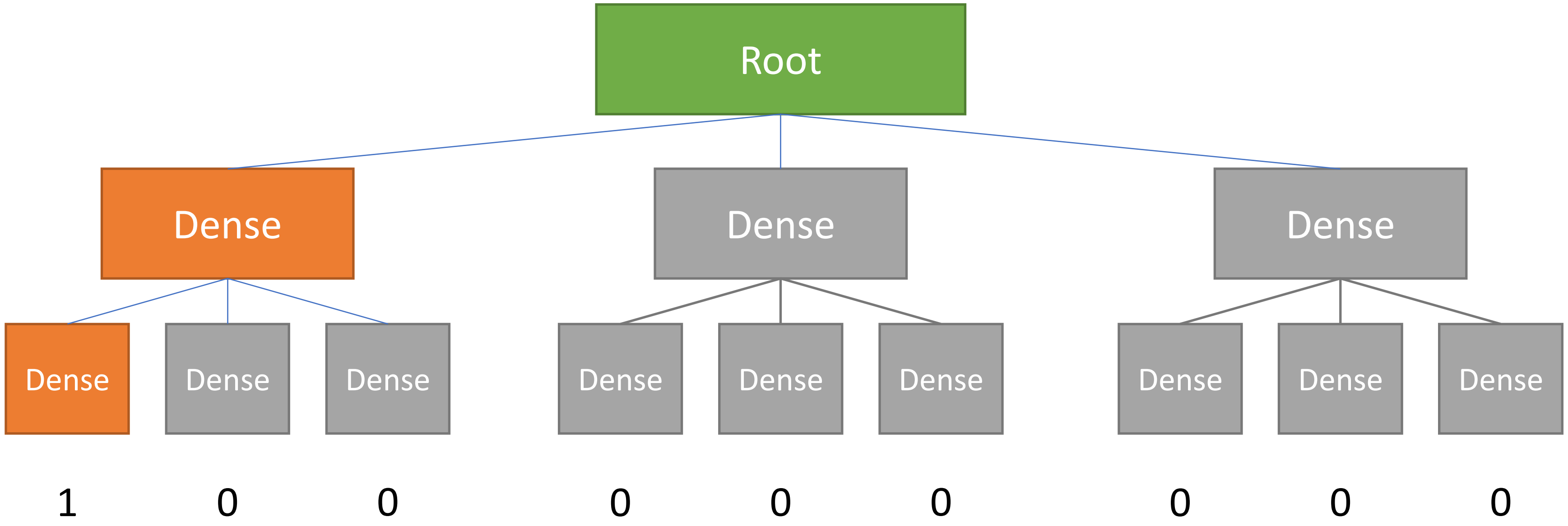
1	0	0
0	0	0
0	0	0



It is a huge waste

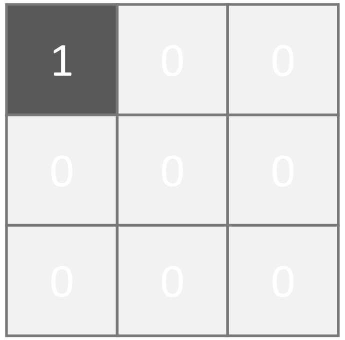
- A dense SNode-tree:

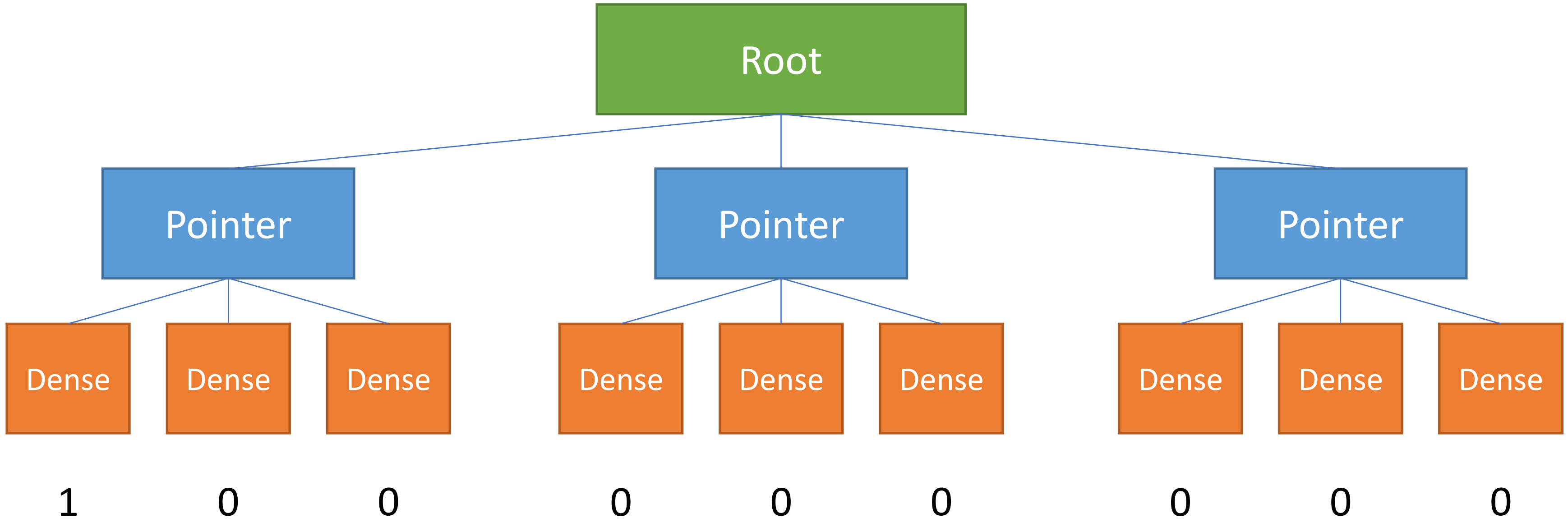




From .dense() to .pointer()

- A sparse SNode-tree:

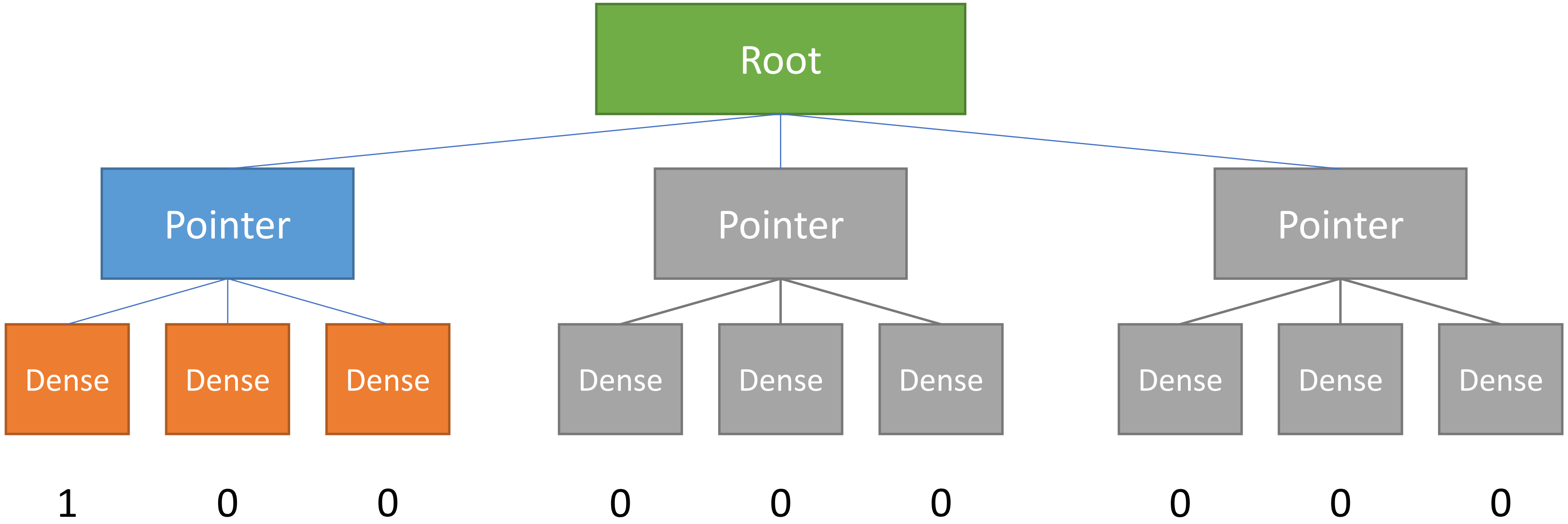




From .dense() to .pointer()

- A sparse SNode-tree:

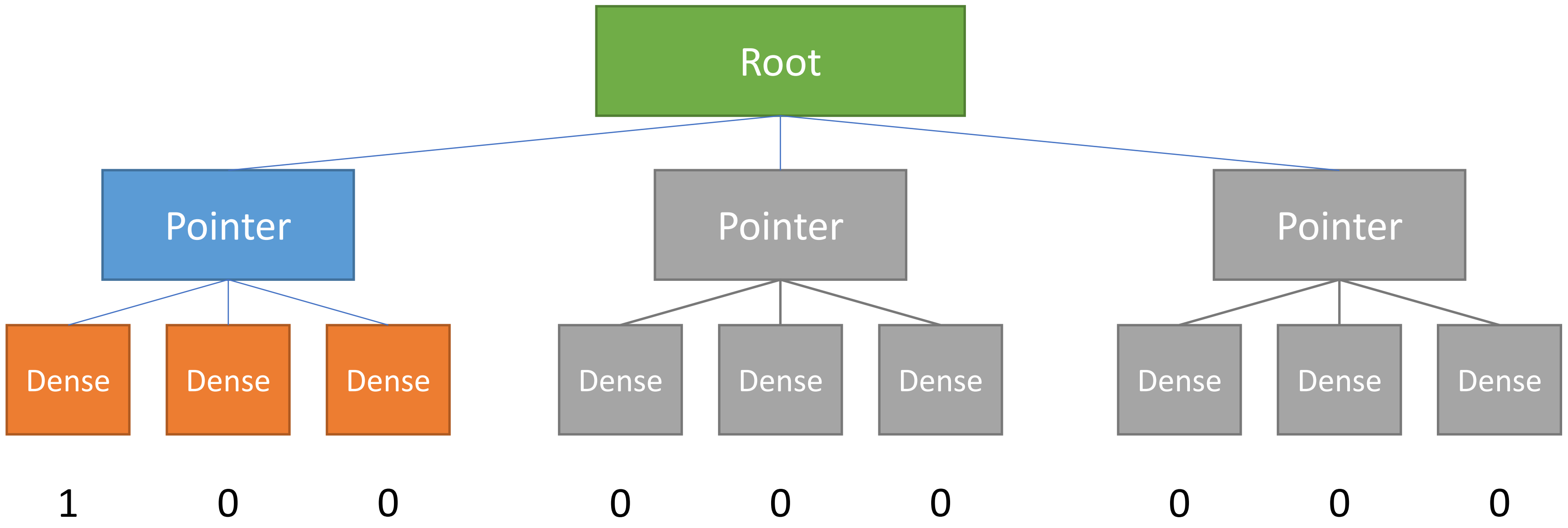
1	0	0
0	0	0
0	0	0



From .dense() to .pointer()

- A sparse SNode-tree:

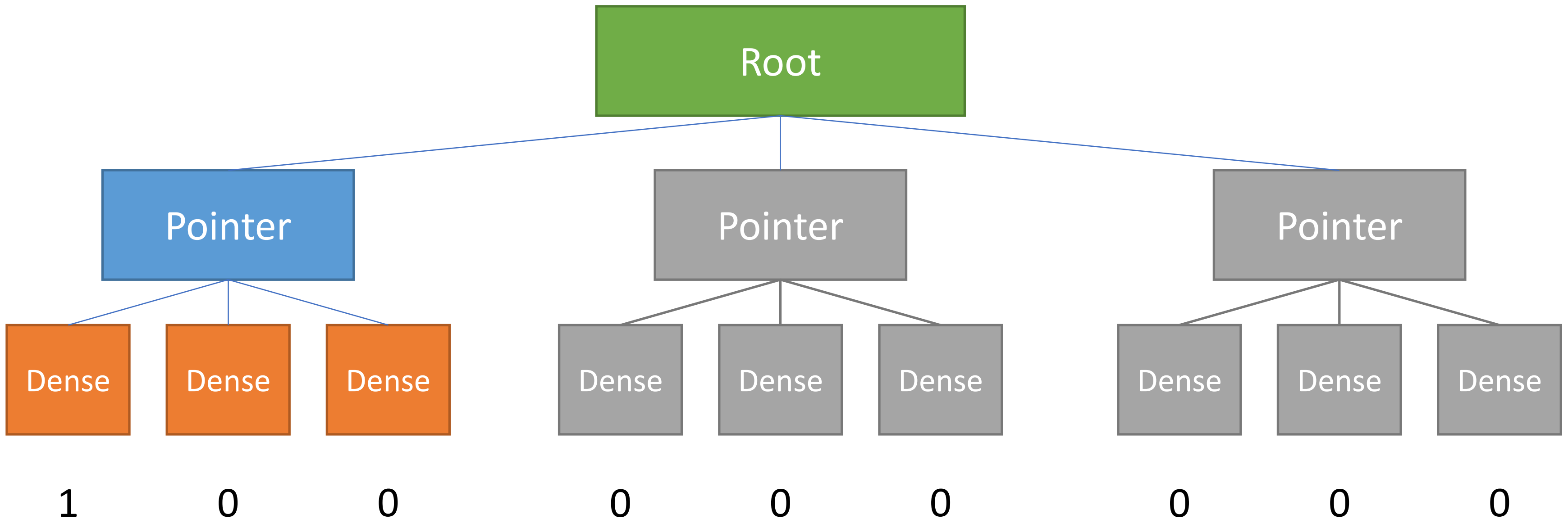
```
x = ti.field(ti.i32)

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:

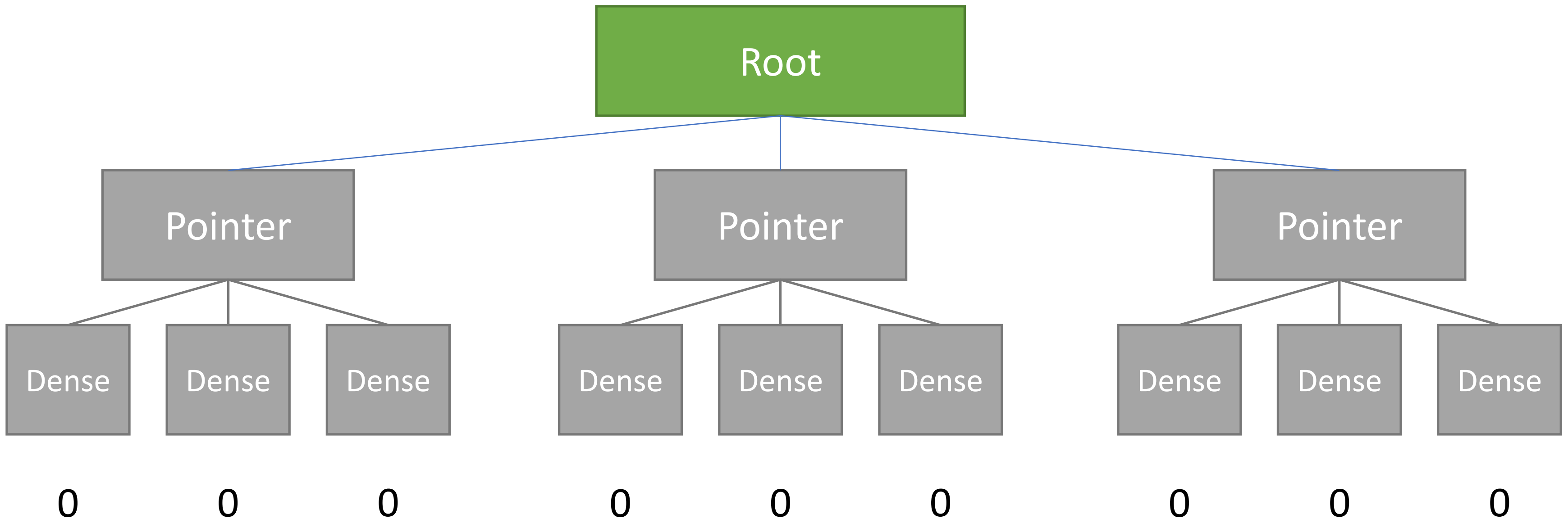
```
x = ti.field(ti.i32)

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:

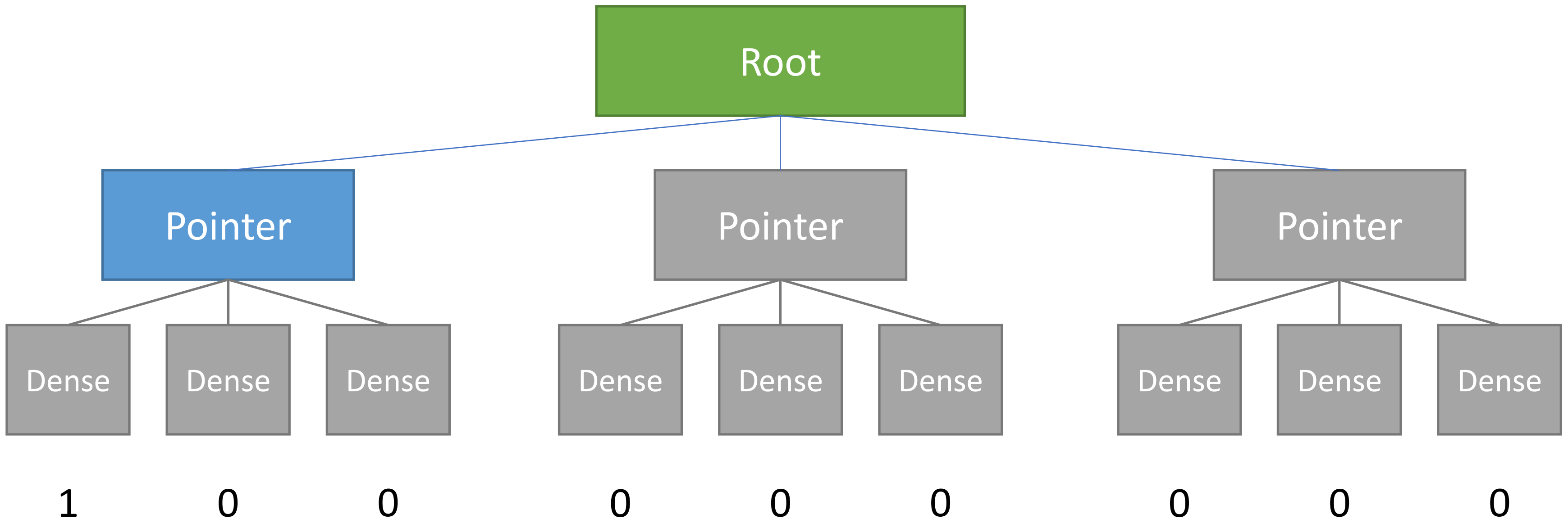
```
x = ti.field(ti.i32)

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:

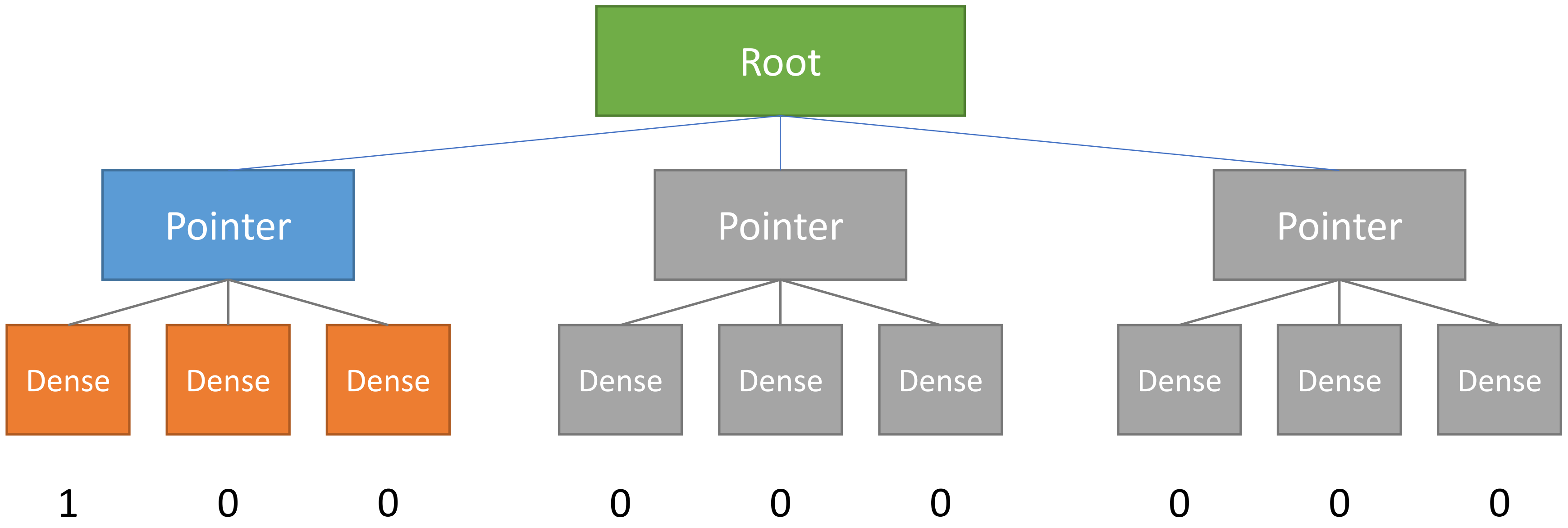
```
x[0,0] = 1
# activates block1[0]
```



Activation

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

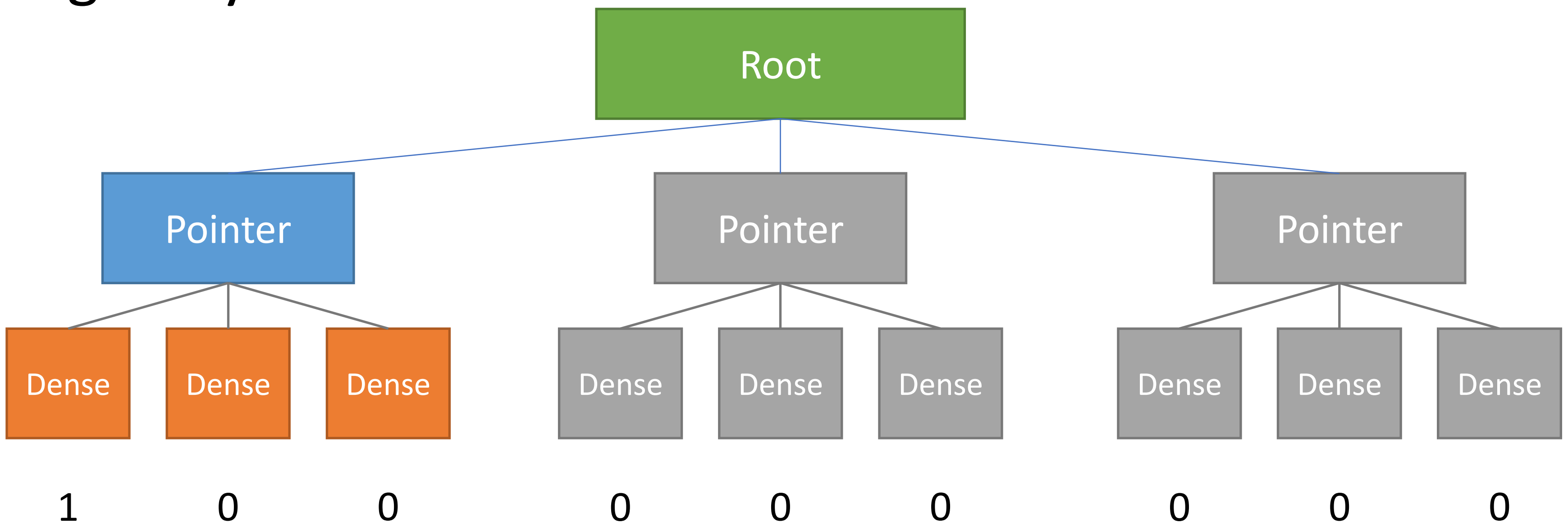
```
x[0,0] = 1
# activates block1[0] and thereby block2[0],
 block2[1] and block2[2]
```



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

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

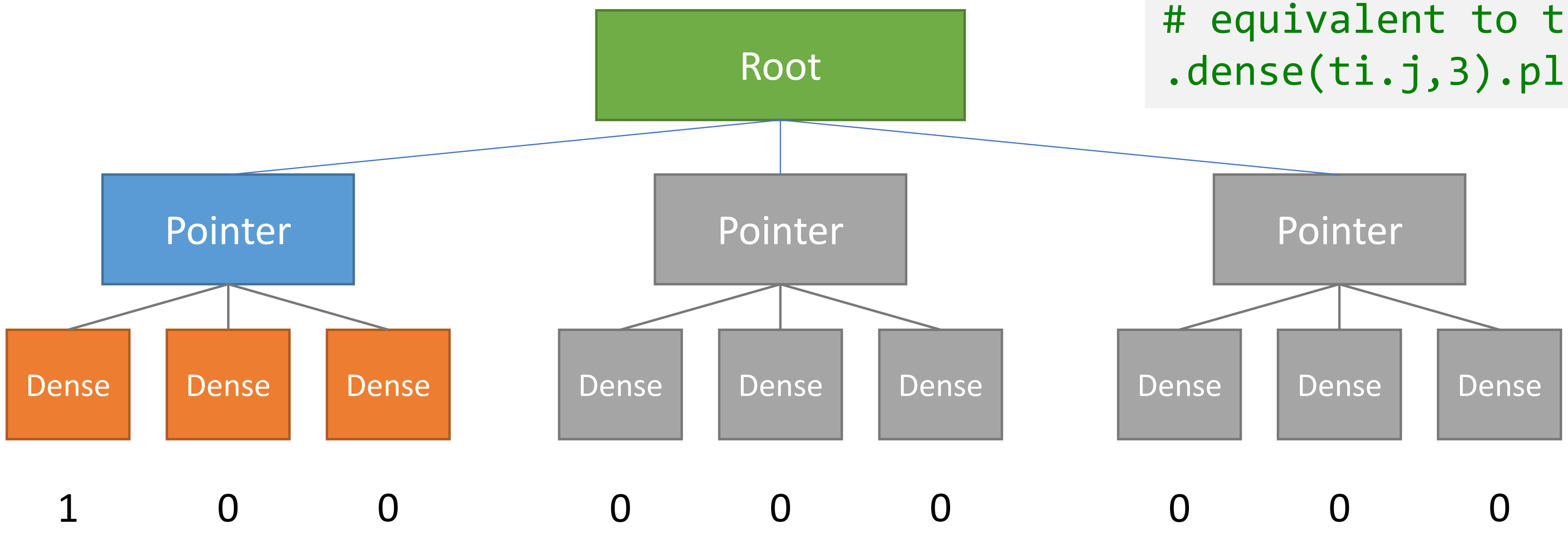
```
@ti.kernel
def access_all():
    for i,j in x:
        print(x[i, j]) # 1, 0, 0

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



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

- Because they belong to the same dense block

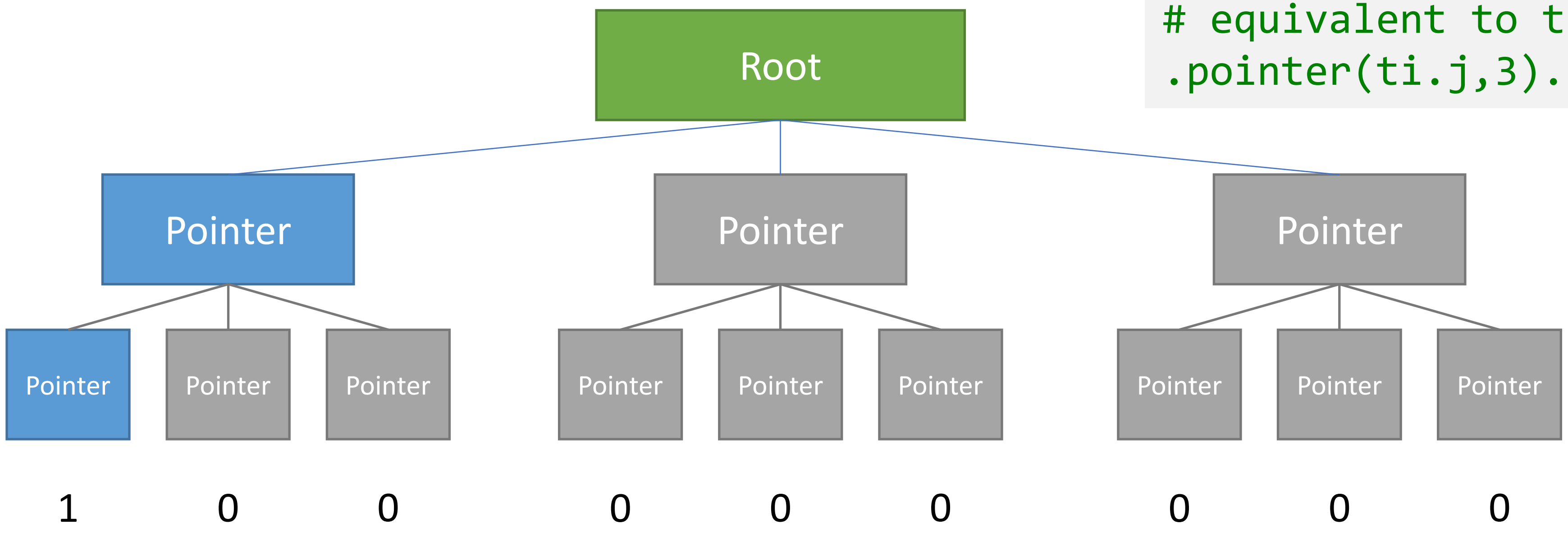
```
x = ti.field(ti.i32)

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.i32 → 32 bits
  - a taichi pointer → 64 bits
  - Dense blocks are faster to visit

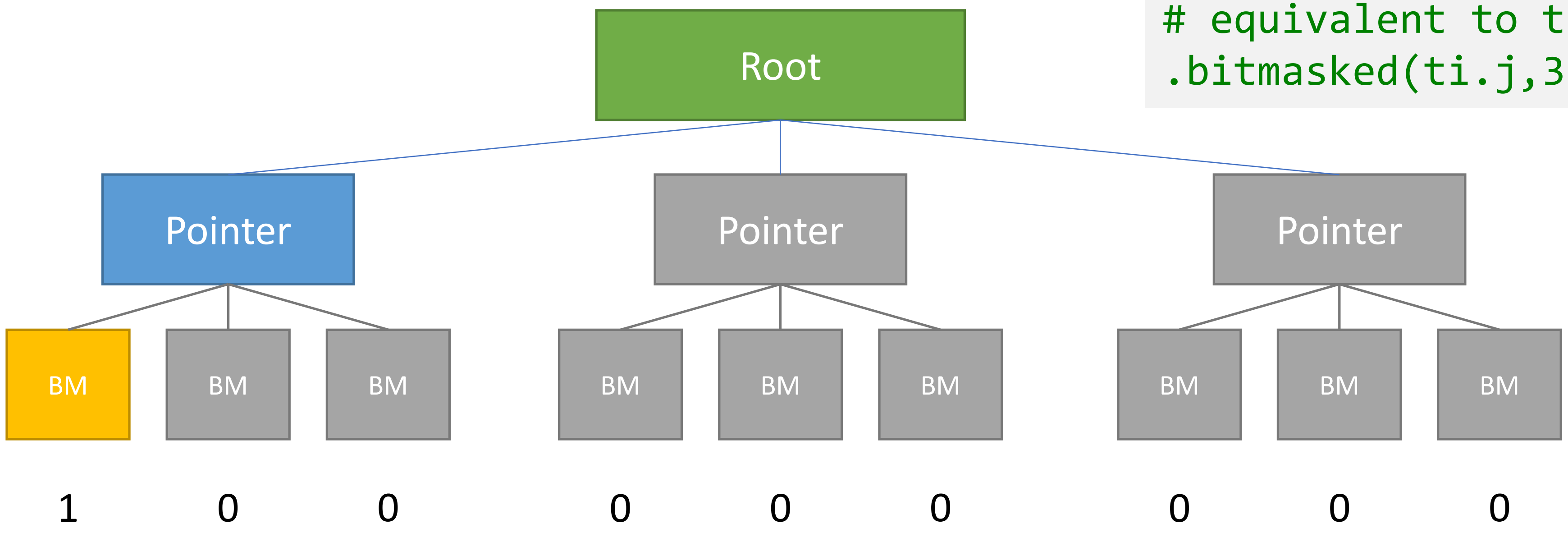
```
x = ti.field(ti.i32)

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

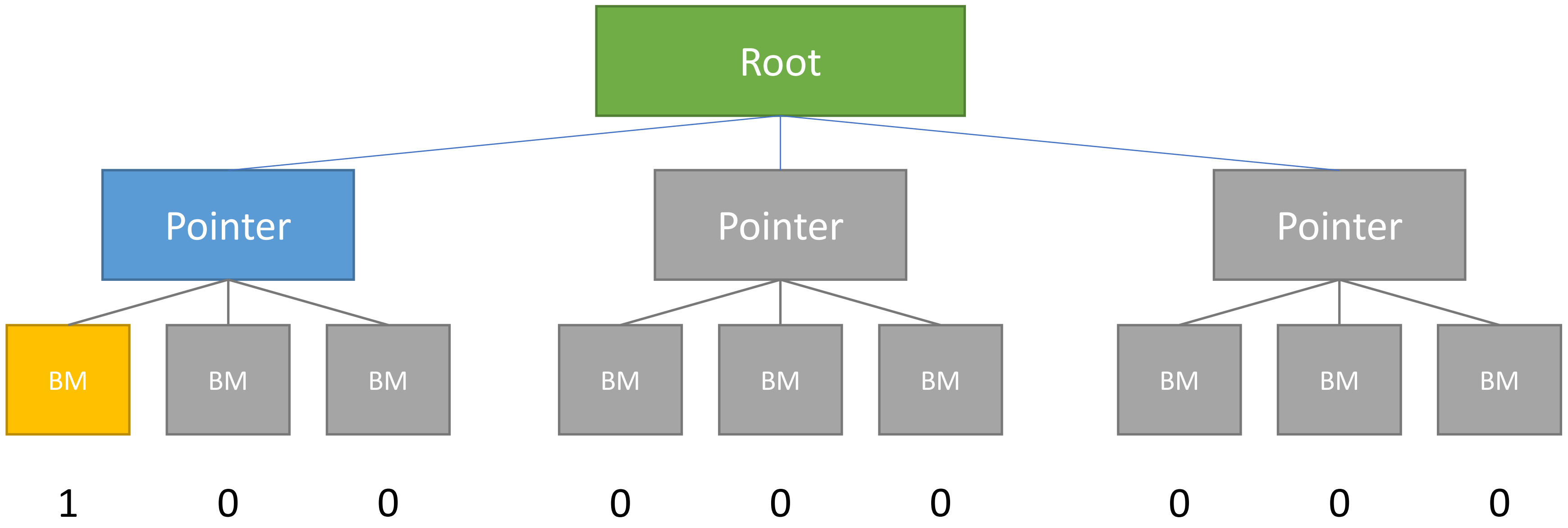
```
x = ti.field(ti.i32)

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...

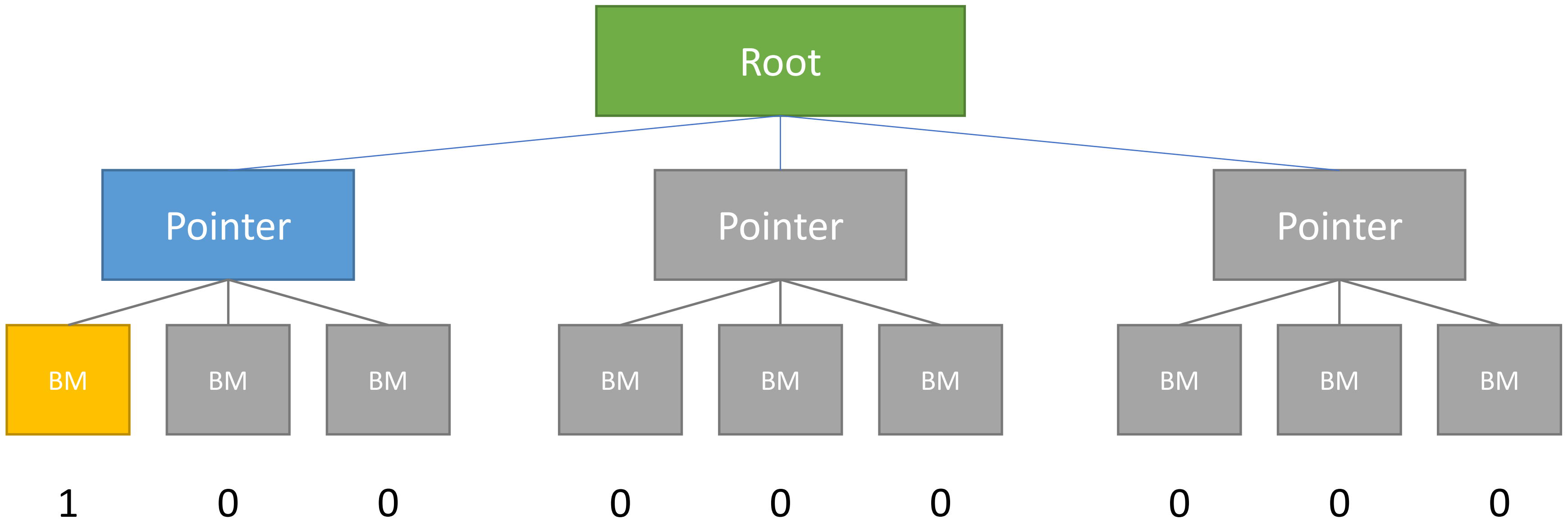
- Works for leaf cells only
- Each leaf cell has its own activation flag

```
@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

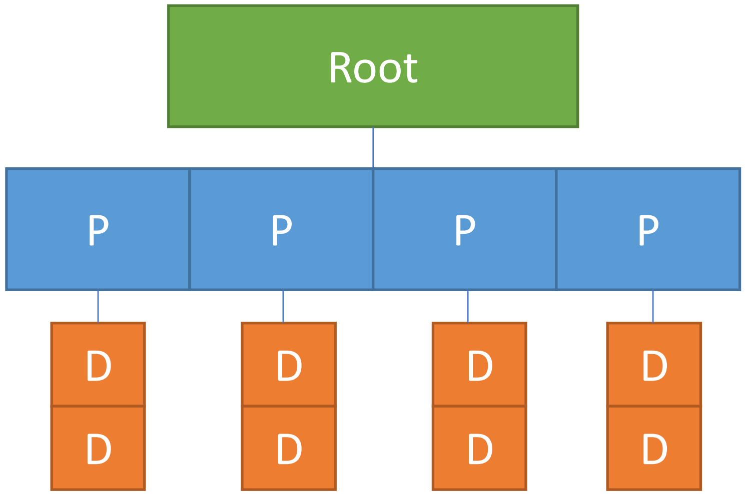
- APIs
  - Check activation status:
    - ti.is_active(snode, [i,j,...])
      - for example: ti.is_active(block1, [0]) #=True
  - Activate/deactivate cells:
    - ti.activate/deactivate(snode, [i,j])
  - Deactivate a cell and its children:
    - snode.deactivate_all()
  - Compute the index of ancestor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

Putting things together

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

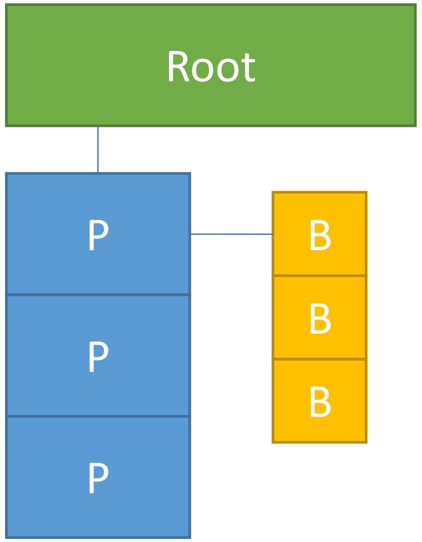
```
x = ti.field(ti.i32)
ti.root.pointer(ti.j,4).dense(ti.i,2).place(x)
```



Putting things together

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

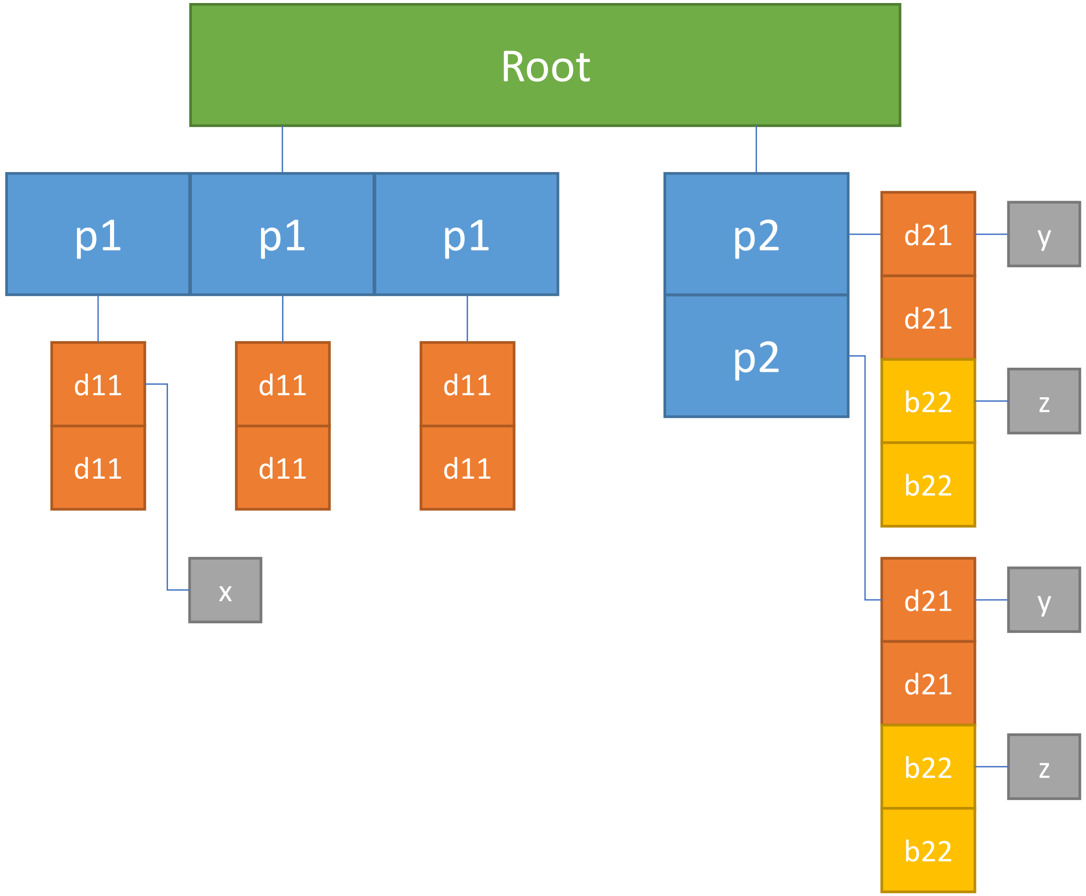
```
x = ti.field(ti.i32)
ti.root.pointer(ti.i,3).bitmasked(ti.i,3).place(x)
```



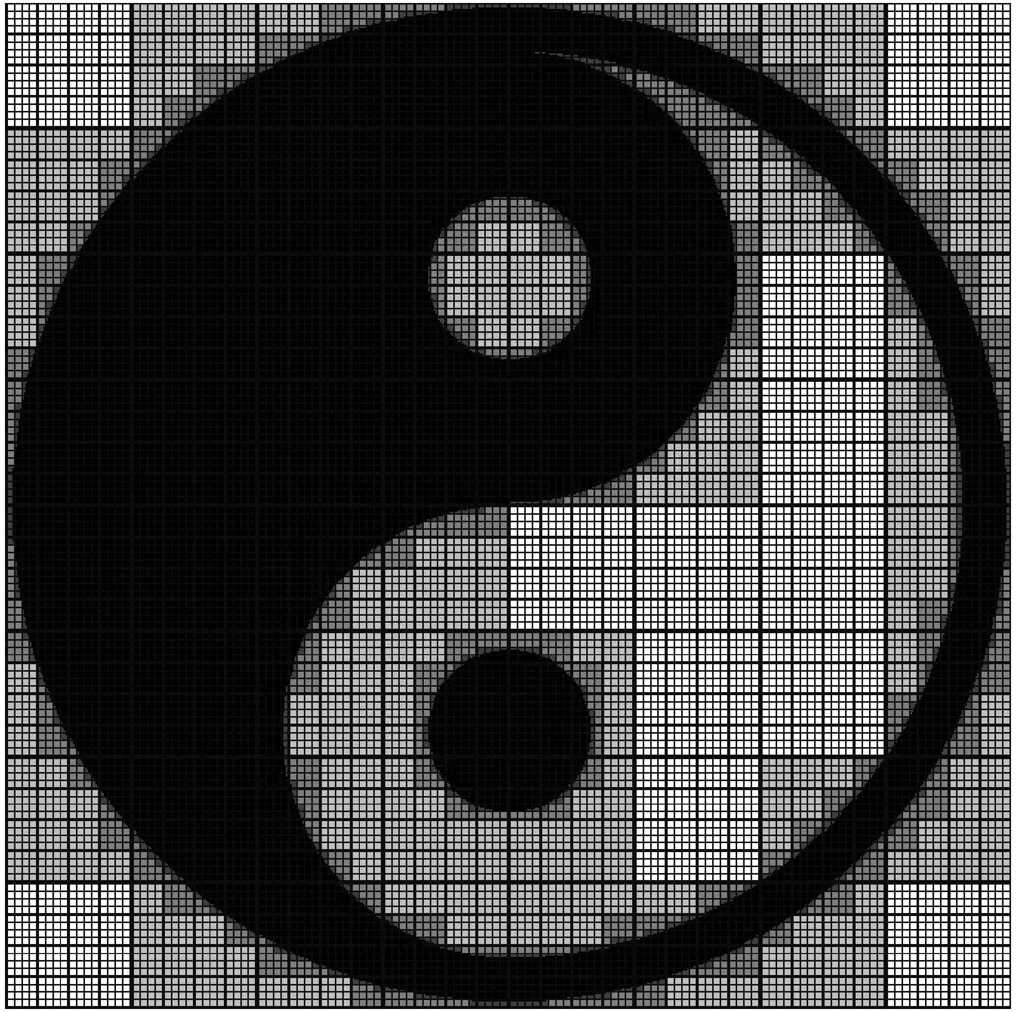
Putting things together

- I wrote this because I could:
  - x: A column-majored 2x3 2D sparse field
  - y/z: hierarchical 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)
```



“ti example taichi_sparse”



```
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].

Remark: Sparse data structures

- 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 ☺

Remark: Sparse data structures

- 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 ☺

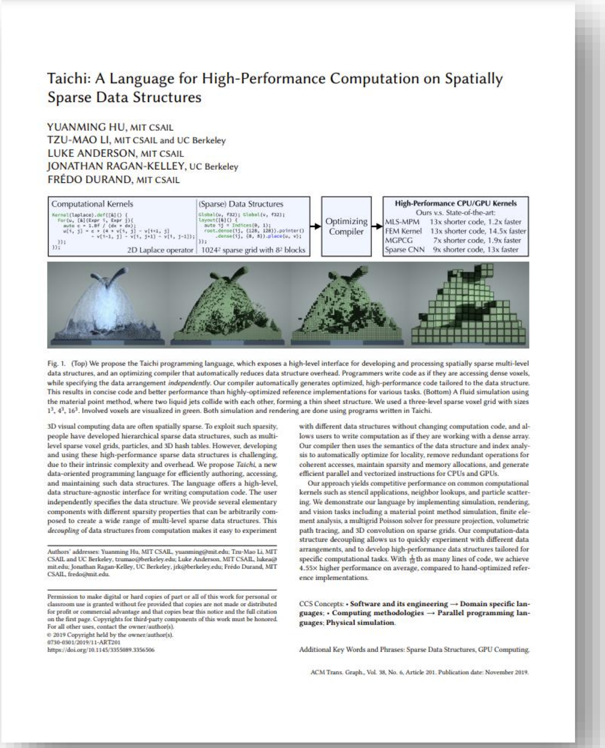
Remark: Sparse data structures

- 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 ☺

Further Readings

- The SNode system (dense and sparse) was one of the main contribution of the original Taichi paper
- Check more demos at the Taichi Elements repo

[Taichi Elements]



Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures

YUANMING HU, MIT CSAIL
TZU-MAO LI, MIT CSAIL and UC Berkeley
LUKE ANDERSON, MIT CSAIL
JONATHAN RAGAN-KELLEY, UC Berkeley
FRÉDO DURAND, MIT CSAIL

Fig. 1. (Top) We propose the Taichi programming language, which exposes a high-level interface for developing and processing spatially sparse multi-level data structures, and an optimizing compiler that automatically reduces data structure overhead. Programmers write code as if they are accessing dense voxels, while specifying the data arrangement *independently*. Our compiler automatically generates optimized, high-performance code tailored to the data structure. This results in concise code and better performance than highly-optimized reference implementations for various tasks. (Bottom) A fluid simulation using the material point method, where two liquid jets collide with each other, forming a thin sheet structure. We used a three-level sparse voxel grid with sizes 1^3, 4^3, 16^3. Involved voxels are visualized in green. Both simulation and rendering are done using programs written in Taichi.

3D visual computing data are often spatially sparse. To exploit such sparsity, people have developed hierarchical sparse data structures, such as multi-level sparse voxel grids, particles, and 3D hash tables. However, developing and using these high-performance sparse data structures is challenging, due to their intrinsic complexity and overhead. We propose *Taichi*, a new data-oriented programming language for efficiently authoring, accessing, and maintaining such data structures. The language offers a high-level, data structure-agnostic interface for writing computation code. The user independently specifies the data structure. We provide several elementary components with different sparsity properties that can be arbitrarily composed to create a wide range of multi-level sparse data structures. This *decoupling* of data structures from computation makes it easy to experiment with different data structures without changing computation code, and allows users to write computation as if they are working with a dense array. Our compiler then uses the semantics of the data structure and index analysis to automatically optimize for locality, remove redundant operations for coherent accesses, maintain sparsity and memory allocations, and generate efficient parallel and vectorized instructions for CPUs and GPUs.

Our approach yields competitive performance on common computational kernels such as stencil applications, neighbor lookups, and particle scattering. We demonstrate our language by implementing simulation, rendering, and vision tasks including a material point method simulation, finite element analysis, a multigrid Poisson solver for pressure projection, volumetric path tracing, and 3D convolution on sparse grids. Our computation-data structure decoupling allows us to quickly experiment with different data arrangements, and to develop high-performance data structures tailored for specific computational tasks. With $\frac{1}{10}$th as many lines of code, we achieve 4.55× higher performance on average, compared to hand-optimized reference implementations.

Authors' addresses: Yuanming Hu, MIT CSAIL, yuanming@mit.edu; Tzu-Mao Li, MIT CSAIL and UC Berkeley, tzumao@berkeley.edu; Luke Anderson, MIT CSAIL, lukea@mit.edu; Jonathan Ragan-Kelley, UC Berkeley, jrk@berkeley.edu; Frédo Durand, MIT CSAIL, fredo@mit.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2019 Copyright held by the owner/author(s).
0730-0301/2019/11-ART201
https://doi.org/10.1145/3355089.3356506

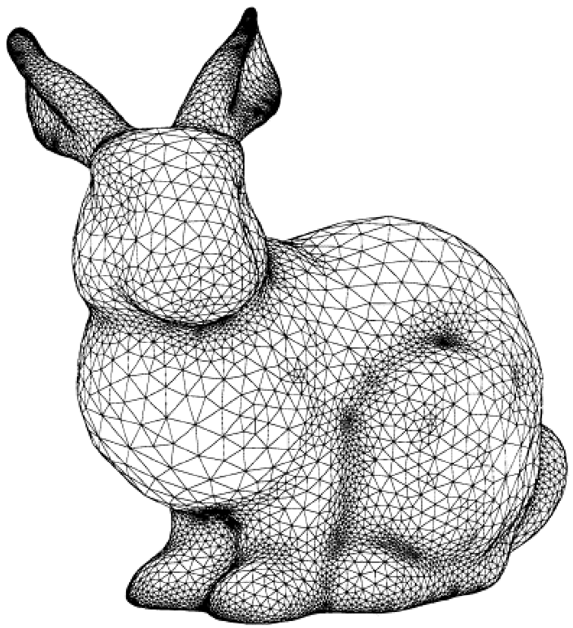
CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Computing methodologies → Parallel programming languages**; **Physical simulation**.

Additional Key Words and Phrases: Sparse Data Structures, GPU Computing.

ACM Trans. Graph., Vol. 38, No. 6, Article 201. Publication date: November 2019.

[Hu et al. 2019]

Mesh-based Data Access

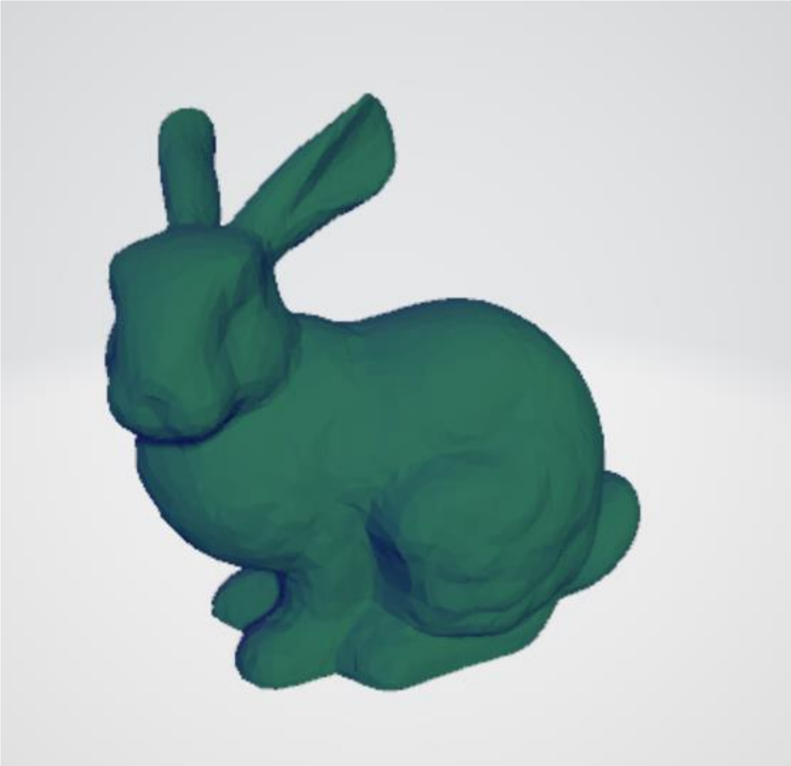


Meshes are represented with explicit relations

```
# OBJ file format with ext .obj
# vertex count = 2503
# face count = 4968
v -3.4101800e-003 1.3031957e-001 2.1754370e-002
v -8.1719160e-002 1.5250145e-001 2.9656090e-002
v -3.0543480e-002 1.2477885e-001 1.0983400e-003
v -2.4901590e-002 1.1211138e-001 3.7560240e-002
v -1.8405680e-002 1.7843055e-001 -2.4219580e-002
v 1.9067940e-002 1.2144925e-001 3.1968440e-002
v 6.0412000e-003 1.2494359e-001 3.2652890e-002
v -1.3469030e-002 1.6299355e-001 -1.2000020e-002
```

```
v -6.8866880e-002 1.4723338e-001 -2.8739870e-002
v -6.0965420e-002 1.7002113e-001 -6.0839390e-002
v -1.3895490e-002 1.6787168e-001 -2.1897230e-002
v -6.9413000e-002 1.5121847e-001 -4.4538540e-002
v -5.5039800e-002 5.7309700e-002 1.6990900e-002
f 1069 1647 1578
f 1058 909 939
f 421 1176 238
f 1055 1101 1042
f 238 1059 1126
f 1254 30 1261
f 1065 1071 1
f 1037 1130 1120
```

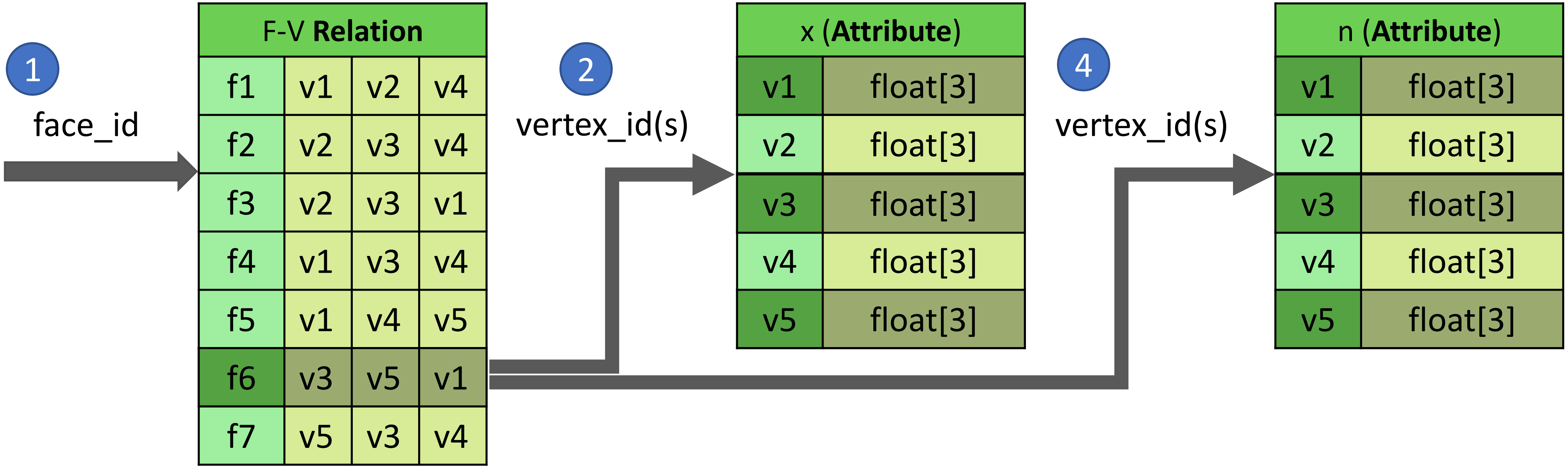
relations



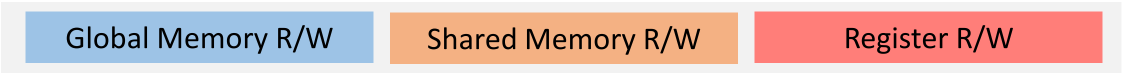
Data accessing for meshes are typically slow

Example: Vertex normal

```
void calc_vertex_normal(int face_id) {
    int v0 = FV[face_id][0], v1 = FV[face_id][1], v2 = FV[face_id][2]; // (1)

    float3 x0 = x[v0], x1 = x[v1], x2 = x[v2]; // (2)

    float3 n0, n1, n2;
    compute_weighted_normal(x0, x1, x2, &n0, &n1, &n2); // (3)

    atomicAdd(&normal[v0], n0); // (4)
    atomicAdd(&normal[v1], n1);
    atomicAdd(&normal[v2], n2);
}
```

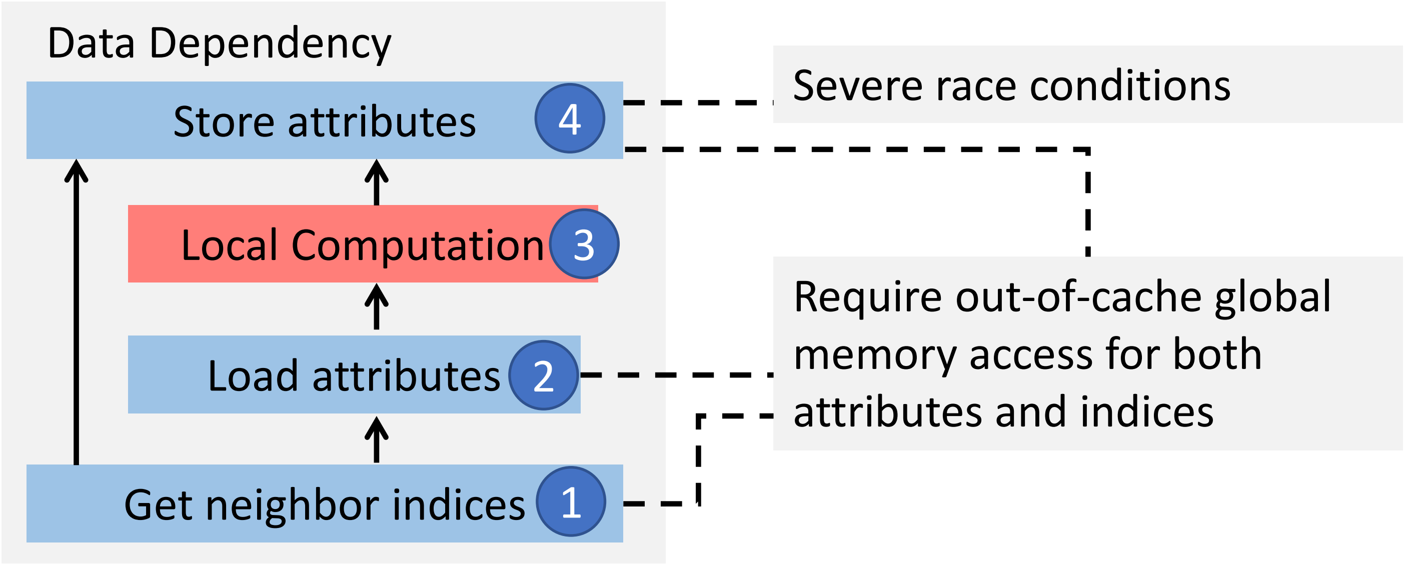


Slow data accessing



Example: Vertex normal

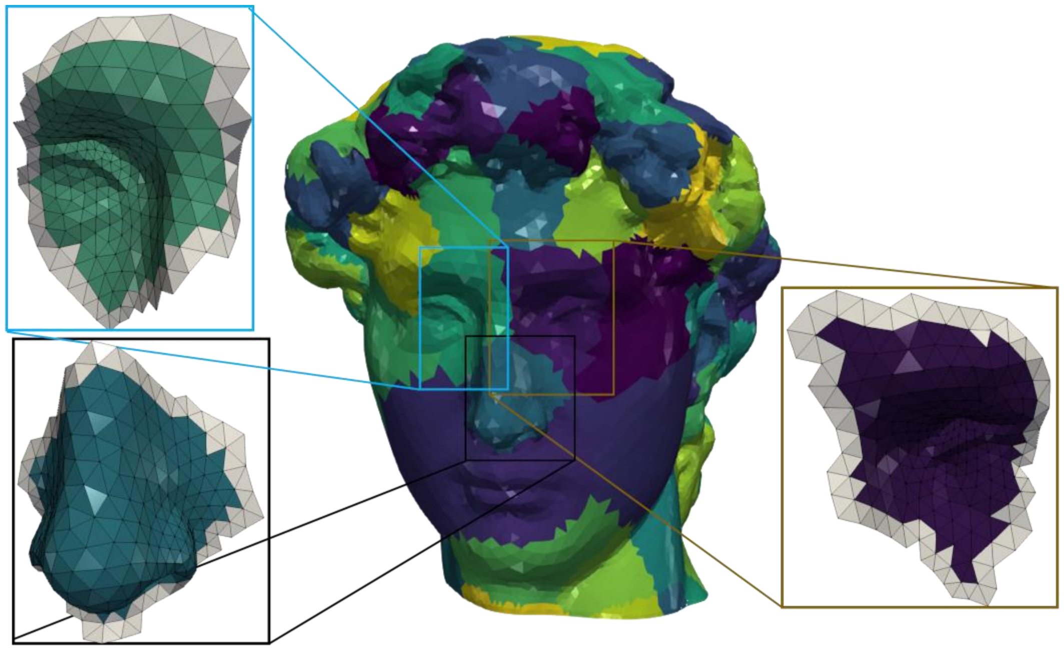
```
void calc_vertex_normal(int face_id) {
    int v0 = FV[face_id][0], v1 = FV[face_id][1], v2 = FV[face_id][2]; // (1)

    float3 x0 = x[v0], x1 = x[v1], x2 = x[v2]; // (2)

    float3 n0, n1, n2;
    compute_weighted_normal(x0, x1, x2, &n0, &n1, &n2); // (3)

    atomicAdd(&normal[v0], n0); // (4)
    atomicAdd(&normal[v1], n1);
    atomicAdd(&normal[v2], n2);
}
```



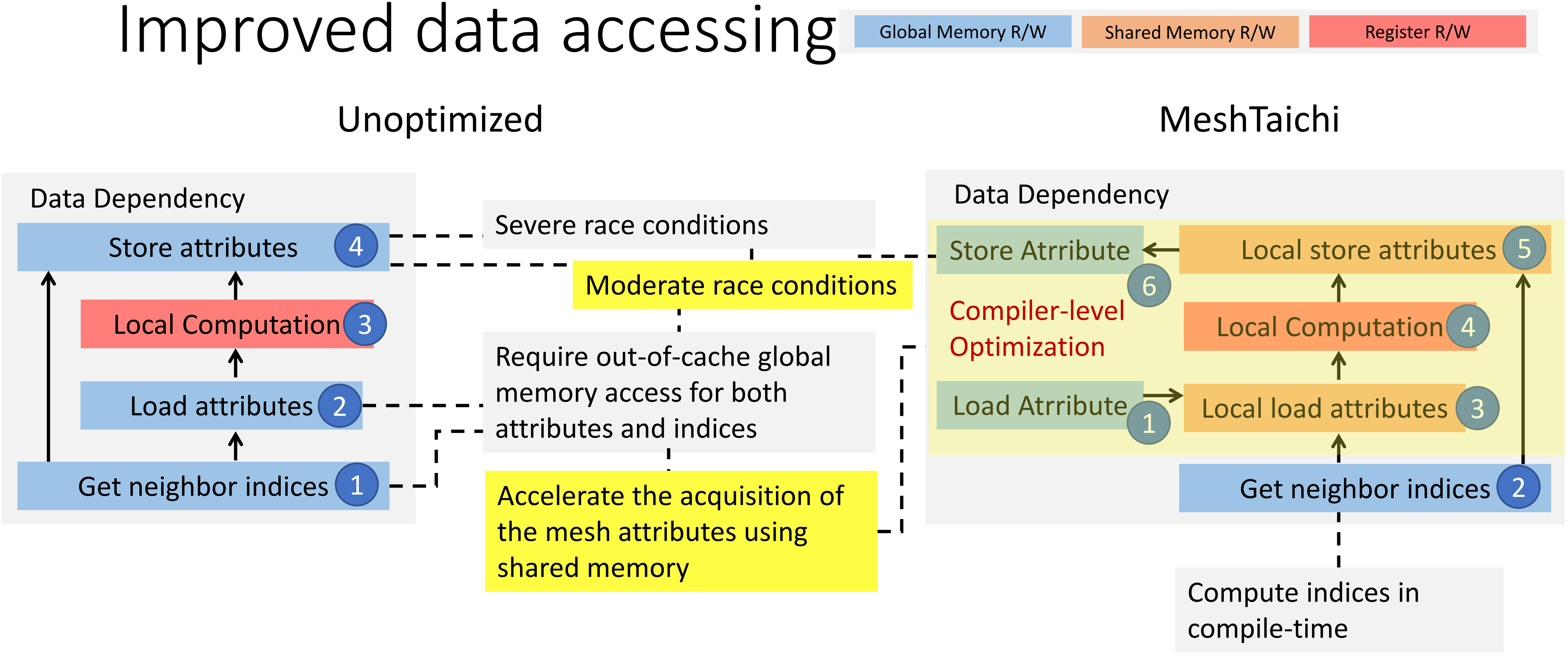
Patch-based data localization and parallization

NVIDIA Turing Mesh Shader

RXMesh [Mahmoud et.al 2021]



Improved data accessing



Instantiating a mesh in MeshTaichi

SIGGRAPH ASIA 2022 DAEGU

1. Load a tri/tet mesh from an external file
2. Tell the compiler your wanted relations
3. Define the attributes for each mesh element

```
mesh = Patcher.load_mesh(“./bunny.mesh”, relations=[“CV”])

mesh.verts.place({'pos': ti.math.vec3,
                  'vel': ti.math.vec3,
                  'force': ti.math.vec3})
mesh.cells.place({‘B’: ti.math.mat3,
                  'w': ti.f32})
```

Looping over a mesh model using mesh-for

- “for every cell in a bunny”:

```
# parallel loop over all mesh cells
for c in bunny.cells:
    ...
```

- “for every vertex in a bunny”:

```
# parallel loop over all mesh vertices
for v in bunny.verts:
    ...
```

- mesh-for-loops are executed in parallel for each patch

Querying relations

- "The velocity of the first vertex of a tetrahedron":

```
for c in bunny.cells:
    v = c.verts[0].vel
```

- "The position of all neighboring vertices of a vertex":

```
for v0 in bunny.verts:
    for v1 in v0.verts:
        diff = v0.pos - v1.pos
```

Caching attributes in advance

SIGGRAPH ASIA 2022 DAEGU

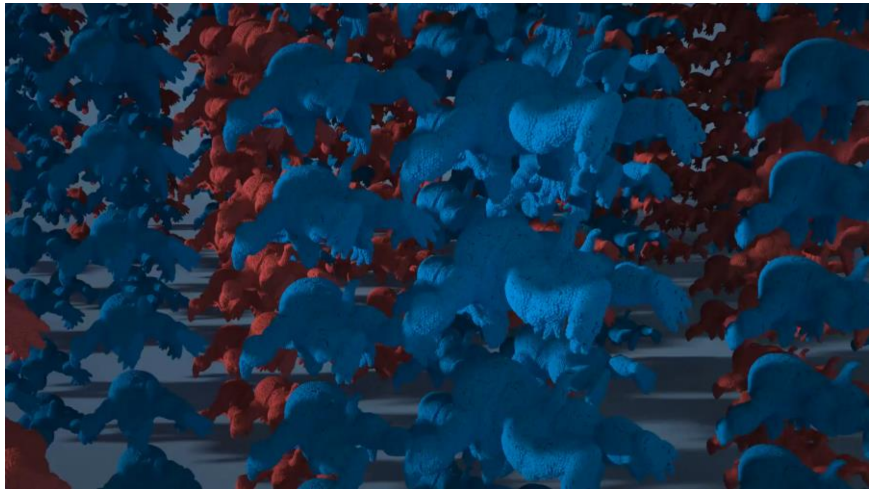
```
@ti.kernel
def substep(model : ti.template()):
    for c in model.cells:
        Ds = ti.Matrix.zero(ti.f32, 3, 3)

        for i in ti.static(range(3)):
            for j in ti.static(range(3)):
                Ds[j, i] = c.verts[i].pos[j] - c.verts[3].pos[j]

        F = Ds @ c.B
        P = PK1(F) # 1st Piola-Krichhoff stress
        H = -c.w * P @ c.B.transpose()
        for i in ti.static(range(3)):
            c.verts[i].force += H[:, i]
            c.verts[3].force += -H[:, i]
```

- Try satisfying lowest occupancy constraint first (determining the maximum size for cached attributes)
- Cache stored attributes first
- Order the other attributes by their load frequency

A sample computation on meshes

Automatic **Parallelization**

```python
@ti.kernel
def substep(model : ti.template()):
    for c in model.cells:
        Ds = ti.Matrix.zero(ti.f32, 3, 3)

        for i in ti.static(range(3)):
            for j in ti.static(range(3)):
                Ds[j, i] = c.verts[i].pos[j] - c.verts[3].pos[j]

        F = Ds @ c.B
        P = PK1(F) # 1st Piola-Krichhoff stress
        H = -c.w * P @ c.B.transpose()
        for i in ti.static(range(3)):
            c.verts[i].force += H[:, i]
            c.verts[3].force += -H[:, i]
```



Auto-managed mesh **relation**

Optimized mesh **attribute** R/W

Changing memory order?

- Sometimes we want to reorder the attributes w.r.t. the patches

```
mesh.verts.place({'pos': ti.math.vec3,
                  'vel': ti.math.vec3,
                  'force': ti.math.vec3})
mesh.cells.place({‘B’: ti.math.mat3,
                  'w': ti.f32})
```

```
mesh.verts.place({'pos': ti.math.vec3,
                  'vel': ti.math.vec3}, reorder = True)
mesh.verts.place({'force': ti.math.vec3}, reorder = False)
mesh.cells.place({'B': ti.math.mat3,
                  'w': ti.f32}, reorder = True)
```

How to access reordered data?

SIGGRAPH ASIA 2022 DAEGU

- “for every cell in a bunny”:

```
# parallel loop over all mesh cells
for c in bunny.cells:
    ...
```

- “for every cell in a reordered bunny”:

```
# parallel loop over all mesh vertices
for c in bunny.cells:
    ...
```

- They are the same ☺
  - Changing memory ordering does not change the computation

Projective Dynamics
0.7 M Tets, 5 PD iterations / frame, 30 CG iterations / linear solve
34 FPS

Remark: Mesh-based operations

SIGGRAPH ASIA 2022 DAEGU

- Operations on meshes are usually complicated and slow
  - Relation information need to be managed explicitly (using lookup tables)
  - Memory access is far less coherent compared to grid data
- MeshTaichi
  - Intuitive syntax to access relations using reference style (no lookups!)
  - Automatic parallelized execution for each patch
  - Efficient attribute access
  - Decouple memory order with computation

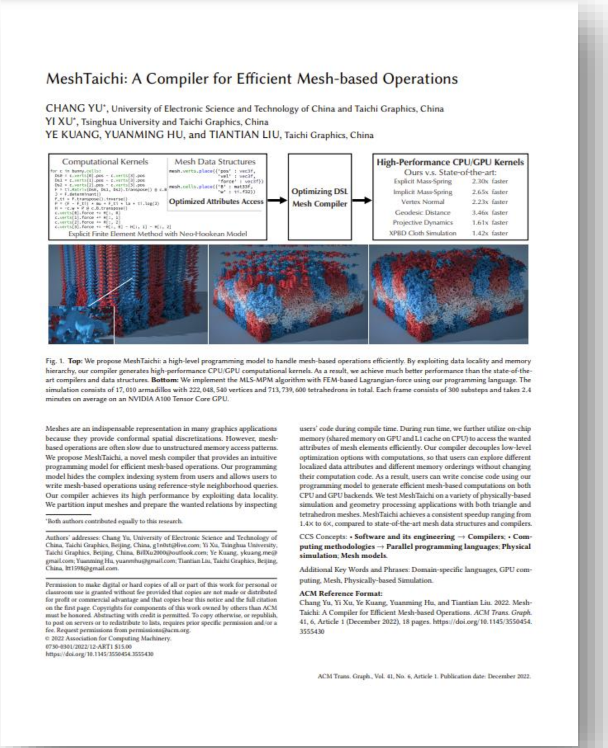
Remark: Mesh-based operations

- Operations on meshes are usually complicated and slow
  - Relation information need to be managed explicitly (using lookup tables)
  - Memory access is far less coherent compared to grid data
- MeshTaichi
  - Intuitive syntax to access relations using reference style (no lookups!)
  - Automatic parallelized execution for each patch
  - Efficient attribute access
  - Decouple memory order with computation

Further Readings

- Technical details described in [Yu et al. 2022]
- Check more examples at the MeshTaichi repo

MeshTaichi



MeshTaichi: A Compiler for Efficient Mesh-based Operations

CHANG YU*, University of Electronic Science and Technology of China and Taichi Graphics, China
YI XU*, Tsinghua University and Taichi Graphics, China
YE KUANG, YUANMING HU, and TIANTIAN LIU, Taichi Graphics, China

Fig. 1. **Top:** We propose MeshTaichi: a high-level programming model to handle mesh-based operations efficiently. By exploiting data locality and memory hierarchy, our compiler generates high-performance CPU/GPU computational kernels. As a result, we achieve much better performance than the state-of-the-art compilers and data structures. **Bottom:** We implement the MLS-MPM algorithm with FEM-based Lagrangian-force using our programming language. The simulation consists of 17, 010 armadillos with 222, 048, 540 vertices and 713, 739, 600 tetrahedrons in total. Each frame consists of 300 substeps and takes 2.4 minutes on average on an NVIDIA A100 Tensor Core GPU.

Meshes are an indispensable representation in many graphics applications because they provide conformal spatial discretizations. However, mesh-based operations are often slow due to unstructured memory access patterns. We propose MeshTaichi, a novel mesh compiler that provides an intuitive programming model for efficient mesh-based operations. Our programming model hides the complex indexing system from users and allows users to write mesh-based operations using reference-style neighborhood queries. Our compiler achieves its high performance by exploiting data locality. We partition input meshes and prepare the wanted relations by inspecting users' code during compile time. During run time, we further utilize on-chip memory (shared memory on GPU and L1 cache on CPU) to access the wanted attributes of mesh elements efficiently. Our compiler decouples low-level optimization options with computations, so that users can explore different localized data attributes and different memory orderings without changing their computation code. As a result, users can write concise code using our programming model to generate efficient mesh-based computations on both CPU and GPU backends. We test MeshTaichi on a variety of physically-based simulation and geometry processing applications with both triangle and tetrahedron meshes. MeshTaichi achieves a consistent speedup ranging from 1.4× to 6×, compared to state-of-the-art mesh data structures and compilers.

*Both authors contributed equally to this research.

Authors' addresses: Chang Yu, University of Electronic Science and Technology of China, Taichi Graphics, Beijing, China, g1n0st@live.com; Yi Xu, Tsinghua University, Taichi Graphics, Beijing, China, BillXu2000@outlook.com; Ye Kuang, ykuang.me@gmail.com; Yuanming Hu, yuanmhu@gmail.com; Tiantian Liu, Taichi Graphics, Beijing, China, ltt1598@gmail.com.

0730-0301/2022/12-ART1 $15.00
https://doi.org/10.1145/3550454.3555430

CCS Concepts: • **Software and its engineering → Compilers**; • **Computing methodologies → Parallel programming languages**; **Physical simulation**; **Mesh models**.

Additional Key Words and Phrases: Domain-specific languages, GPU computing, Mesh, Physically-based Simulation.

ACM Reference Format:
Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. 2022. MeshTaichi: A Compiler for Efficient Mesh-based Operations. *ACM Trans. Graph.* 41, 6, Article 1 (December 2022), 18 pages. https://doi.org/10.1145/3550454.3555430

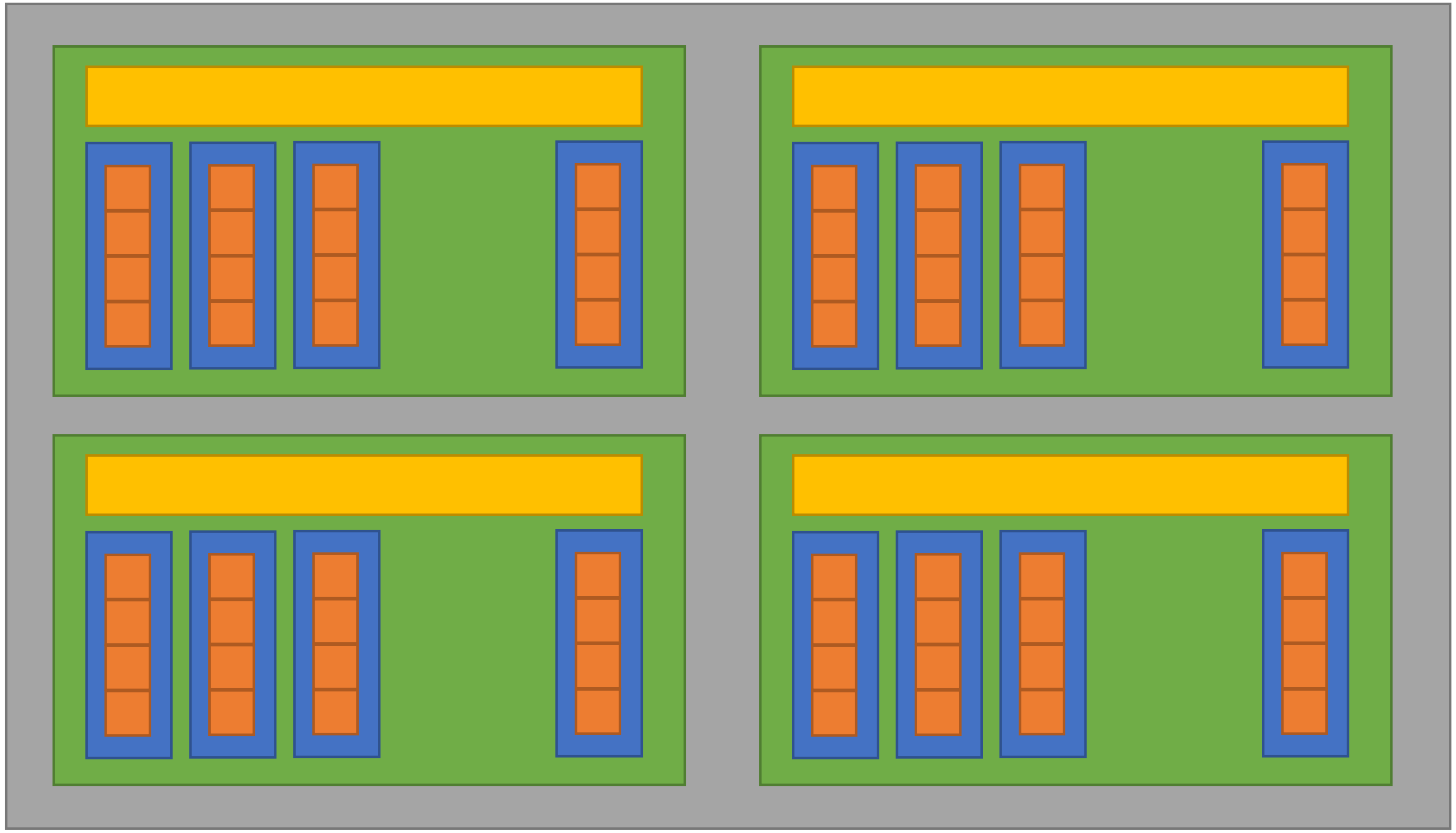
[Yu et al. 2022]

Compiler Hints

Compiler hints

- The cache / shared memory has limited size
- Some data may be more important (more frequently used) than others
- We can tell the compiler to prioritize the important data

Thread hierarchy of Taichi in GPU



Iteration: each iteration in a for-loop

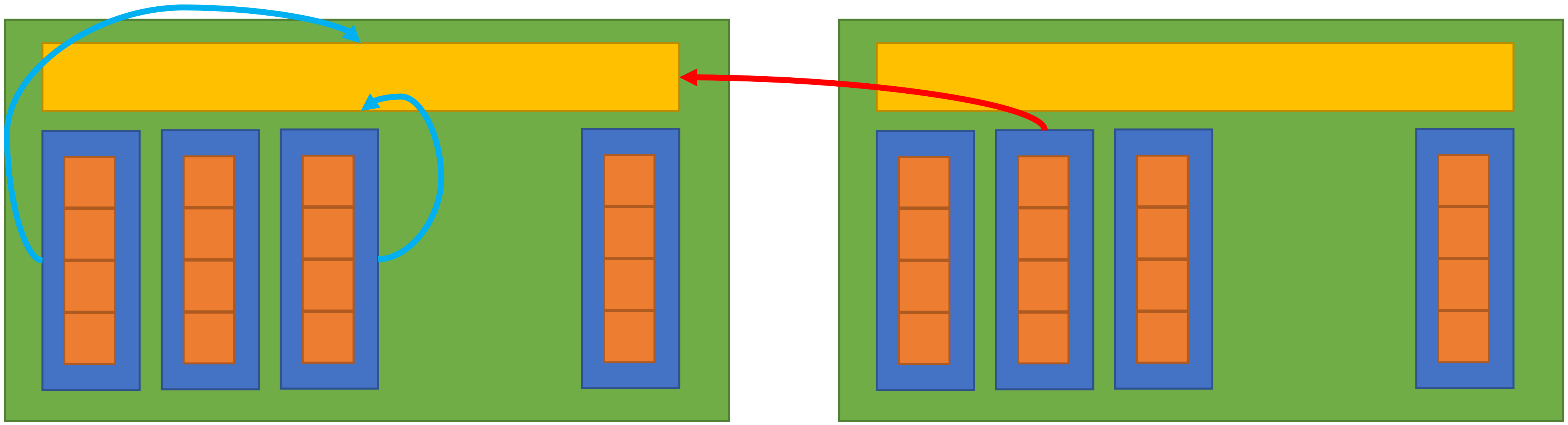
Thread: the minimal parallelizable unit

Block: threads are grouped in blocks with shared block local storage.

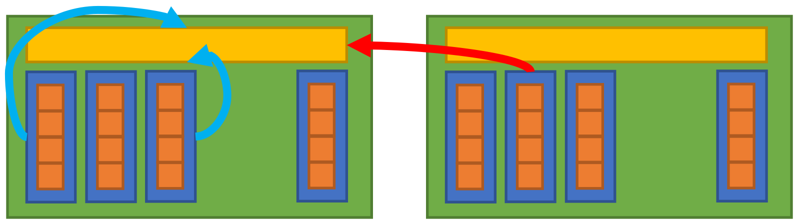
Grid: the minimal unit that being launched from the host

The block local storage (BLS)

- Implemented using shared memory in GPU
- Fast to read/write but small in size

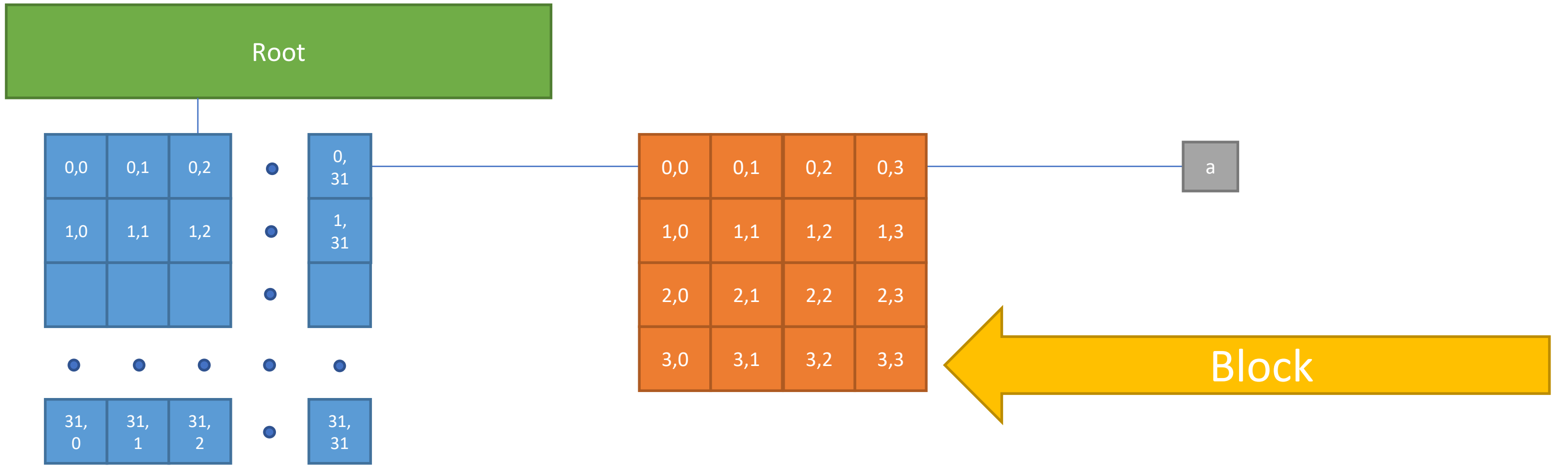


Decide the block size



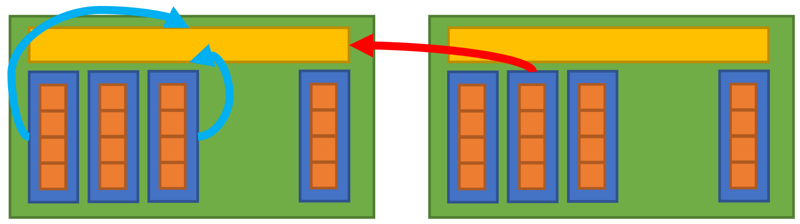
- The ideal block size of an hierarchically defined field (SNode-tree):

```
a = ti.field(ti.f32)
# `a` has a block size of 4x4
ti.root.pointer(ti.ij, 32).dense(ti.ij, 4).place(a)
```



bls_size = 4x4(x4 Bytes)
= 64 Bytes

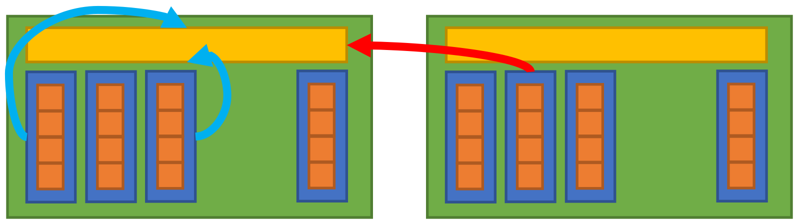
Decide the block size



- Decide the size of the blocks by prepending a *ti.block_dim()* before a parallel for-loop: (default_block_dim = 256 Bytes)

```
@ti.kernel
def func():
    for i in range(8192):  # no decorator, use default settings
        ...

    ti.block_dim(128)      # change the property of next for-loop:
    for i in range(8192):  # will be parallelized with block_dim=128
        ...

    for i in range(8192):  # no decorator, use default settings
        ...
```

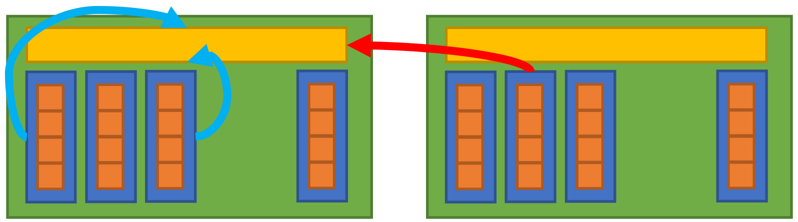
Cache the wanted data



- Cache the most frequently-used data into local storage manually using *ti.block_local()*:

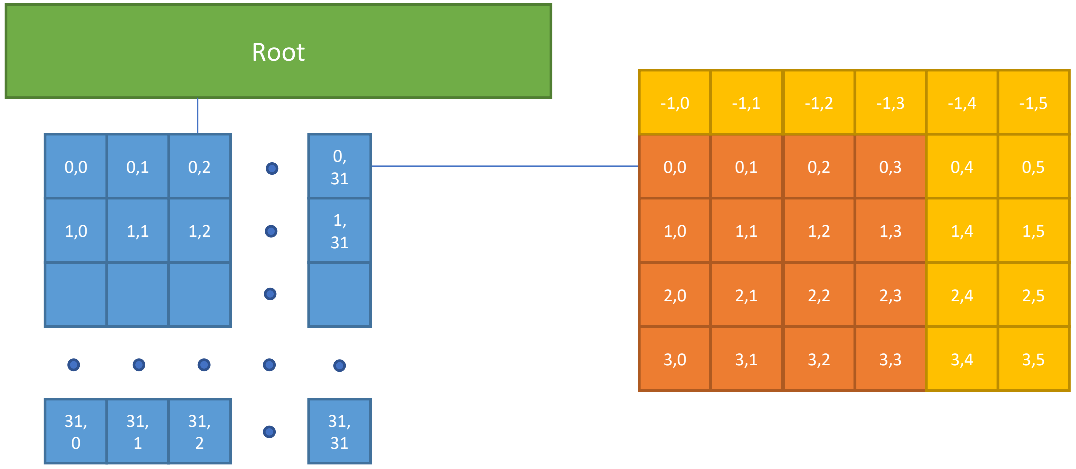
```
a = ti.field(ti.f32)
# `a` has a block size of 4x4
ti.root.pointer(ti.ij, 32).dense(ti.ij, 4).place(a)

@ti.kernel
def foo():
  # Taichi will cache `a` into shared memory first
  ti.block_local(a)
  for i, j in a:
    print(a[i - 1, j], a[i, j + 2])
```

Cache the wanted data



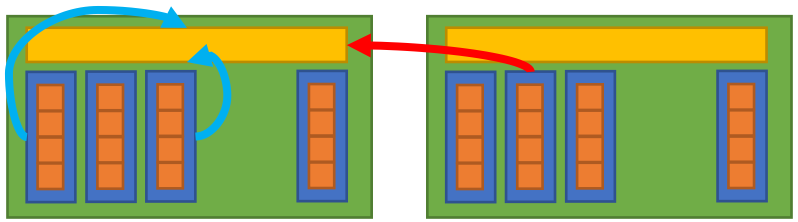
- Cache the most frequently-used data into local storage manually using *ti.block_local()*:

```
ti.block_local(a)
for i, j in a:
  print(a[i - 1, j], a[i, j + 2])
```



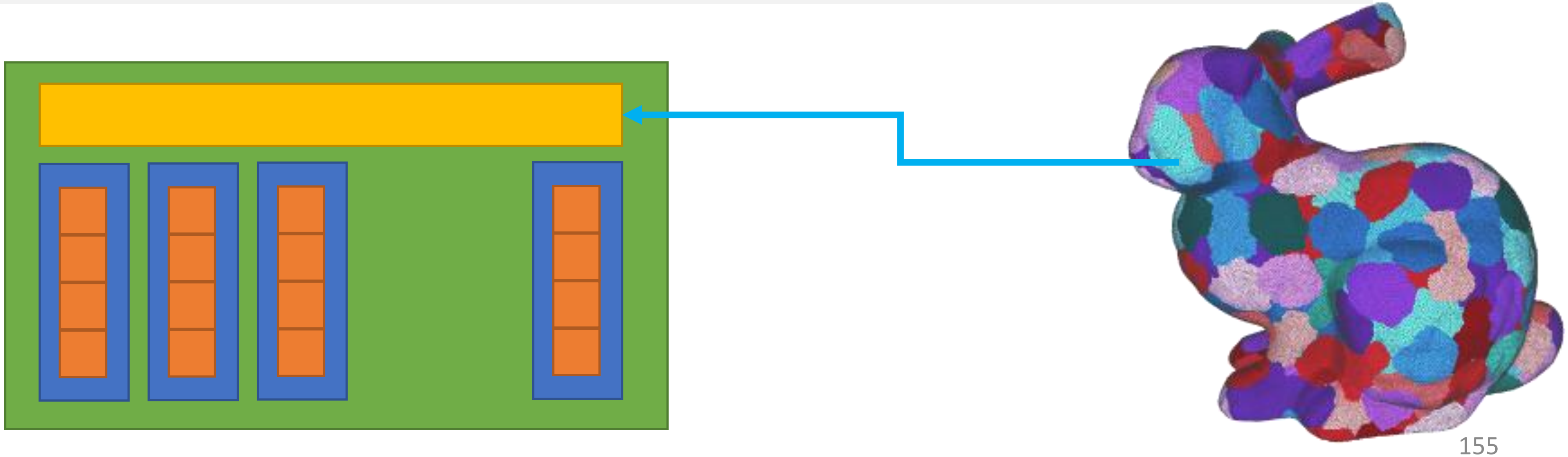
bls_size = 5x6(x4 Bytes)
= 120 Bytes

Cache the wanted mesh data



- Cache the most frequently-used attributes into local storage manually using *ti.mesh_local()*:

```
# put attributes pos and force of vertices into the shared memory
ti.mesh_local(bunny.verts.pos, bunny.verts.force)
for c in bunny.cells:
    ...
```

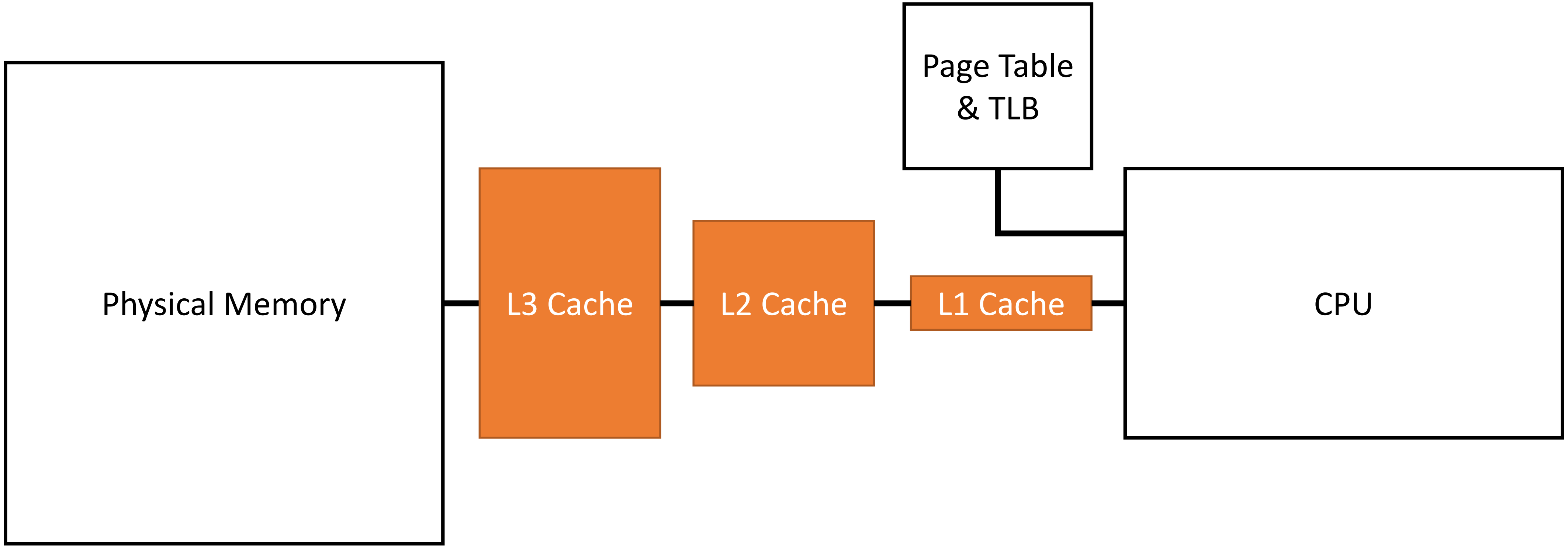


Remark: Compiler hints

- Tell the compiler to cache the most important (frequently-used) data
  - ti.block_local: works for dense SNodes, good for stencil computation
  - ti.mesh_local: works for mesh attributes

Quantized Data Types

Previously in this course: data locality



Previously in this course: data types

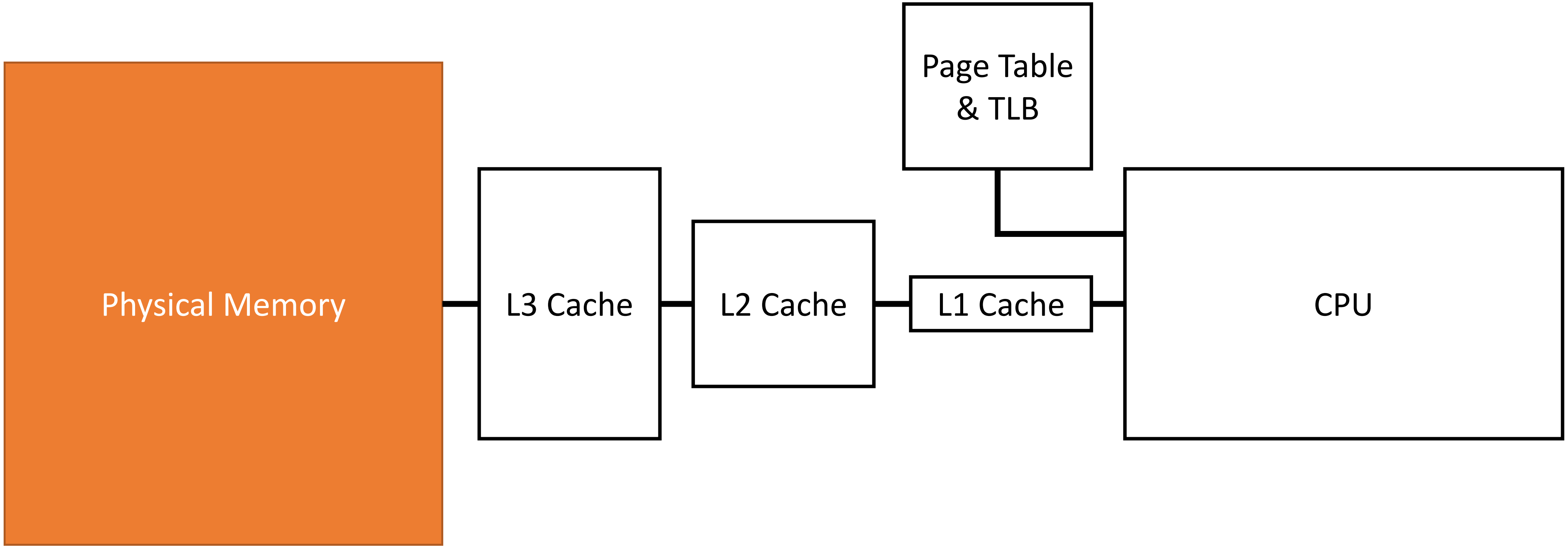
- Primitive data types:
  - signed integers: ti.i8, ti.i16, ti.i32, ti.i64
  - unsigned integers: ti.u8, ti.u16, ti.u32, ti.u64
  - floating points: ti.f32, ti.f64
- Compound data types:

```
vec3f = ti.types.vector(3, ti.f32)
mat2f = ti.types.matrix(2, 2, ti.f32)
ray = ti.types.struct(ro=vec3f, rd=vec3f, l=ti.f32)
```

- Multi-dimensional arrays:

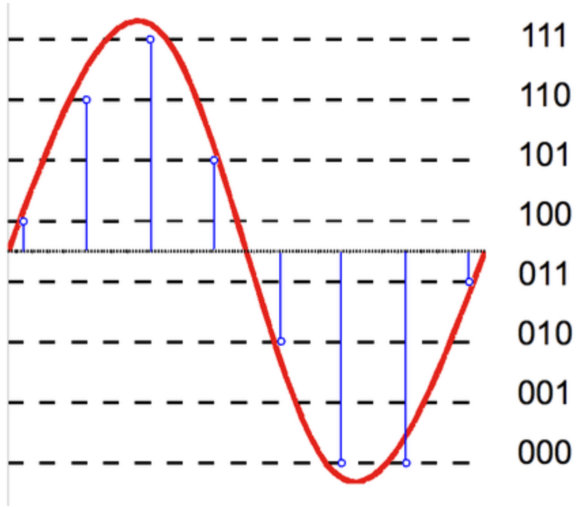
```
gravitational_field = ti.Vector.field(n = 3,dtype=ti.f32,shape=(256,256,128))
strain_tensor_field = ti.Matrix.field(n = 2,m = 2, dtype=ti.f32, shape=(64,64))
```

Now let us focus on total memory consumption



Quantization

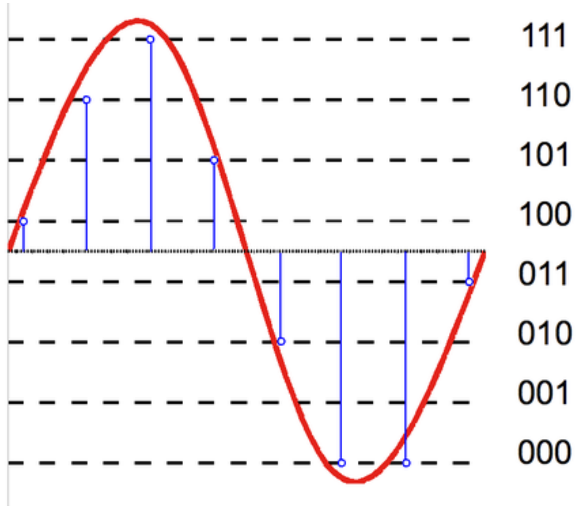
- “Mapping input values from a large set to output values in a smaller set”, for example:
  - (Signal processing) Mapping an analog signal to digital signal
  - (Computer) Mapping high-precision data types to low-precision data types



Quantization

- “Mapping in[…]put values in a smaller set”, for e[…]
  - (Signal p[…]gital signal
  - (Comput[…]es to low-precision data types

We can trade off precision for memory consumption



Quantized data types

SIGGRAPH ASIA 2022 DAEGU

- Quantized integers

```
# if we know the quantity belongs to [0, 32)
u5 = ti.types.quant.int(bits=5, signed=False)
```

- Quantized fixed-point numbers

```
# 10-bit signed (default) fixed-point type within [-20.0, 20.0]
fixed_type_a = ti.types.quant.fixed(bits=10, max_value=20.0)
# 5-bit unsigned fixed-point type within [0.0, 100.0]
fixed_type_b = ti.types.quant.fixed(bits=5, signed=False, max_value=100.0)
# 6-bit unsigned fixed-point type within [0, 64.0]
fixed_type_c = ti.types.quant.fixed(bits=6, signed=False, scale=1.0)
```

- Quantized floating-point numbers

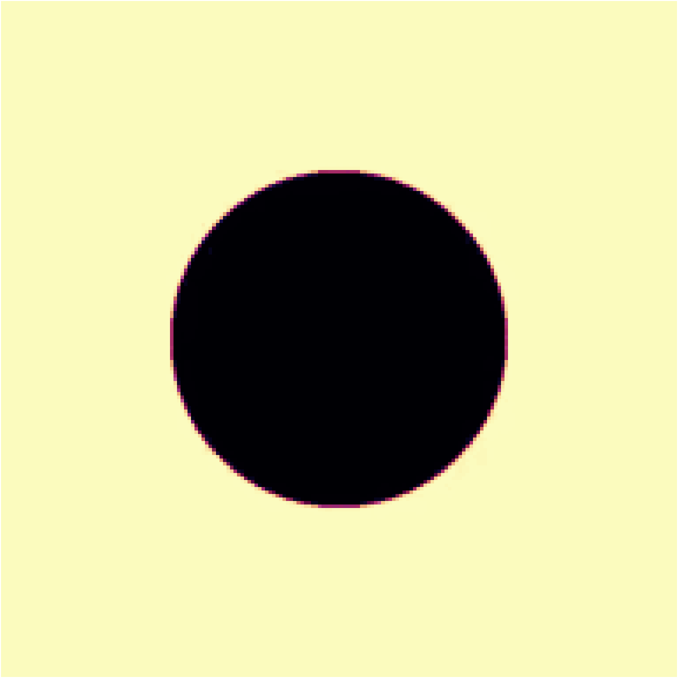
```
# 15-bit signed (default) floating-point type with 5 exponent bits
fp_5_10 = ti.types.quant.float(exp=5, frac=10)
```

“ti example euler”

- `Q = ti.Vector.field(4, dtype=ti.f32, shape=(N, N))`
  - Every vector is sized $4 \times 32 = 128$ bits

- Can we shrink it to 64 bits?
  - Sure:

```
fp_8_8 = ti.types.quant.float(exp=8, frac=8)
```

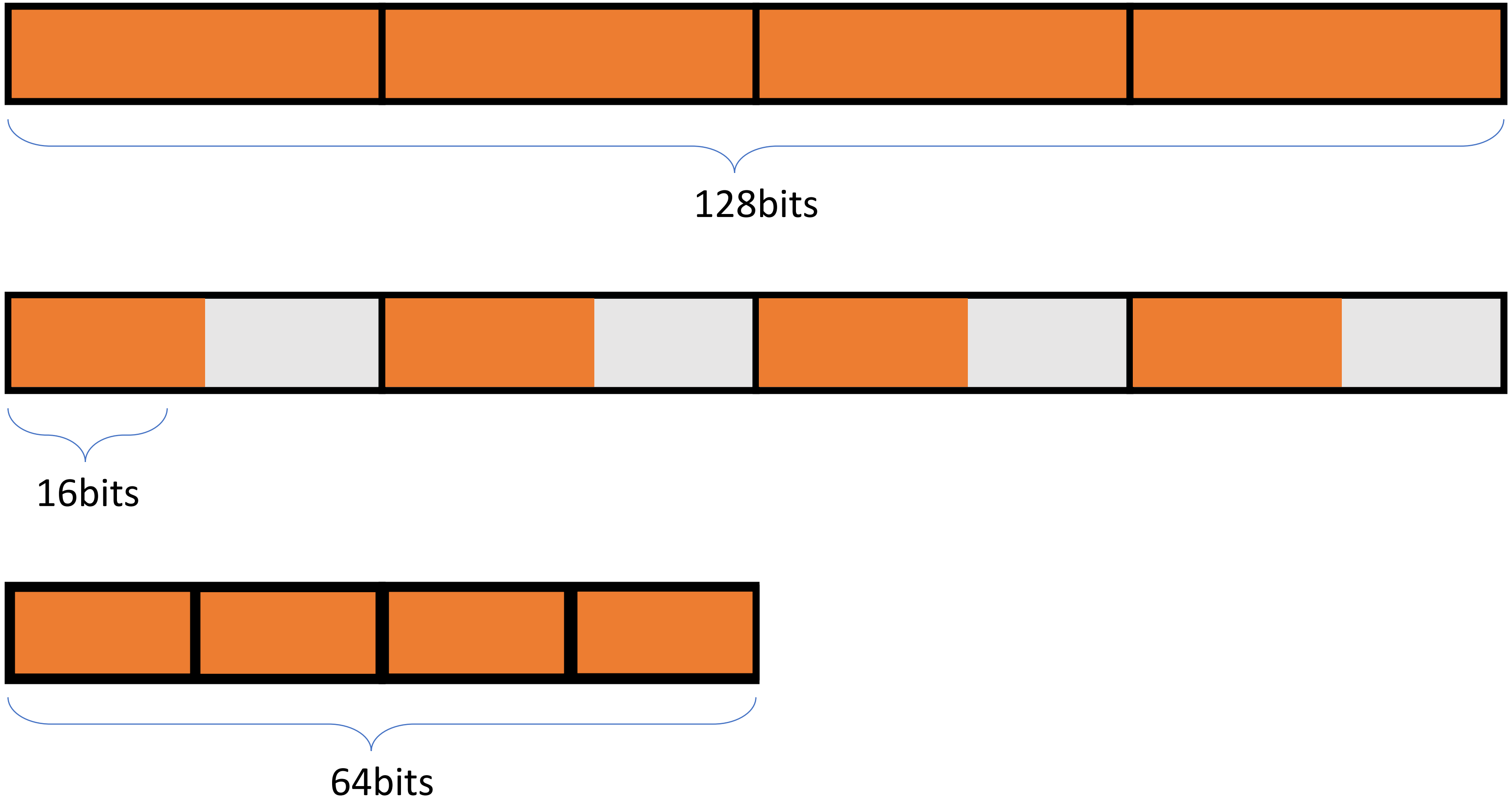


Packing quantized data types to quantized fields

A field with
regular data types

A field with
quantized data types

An ideal field with
quantized data types



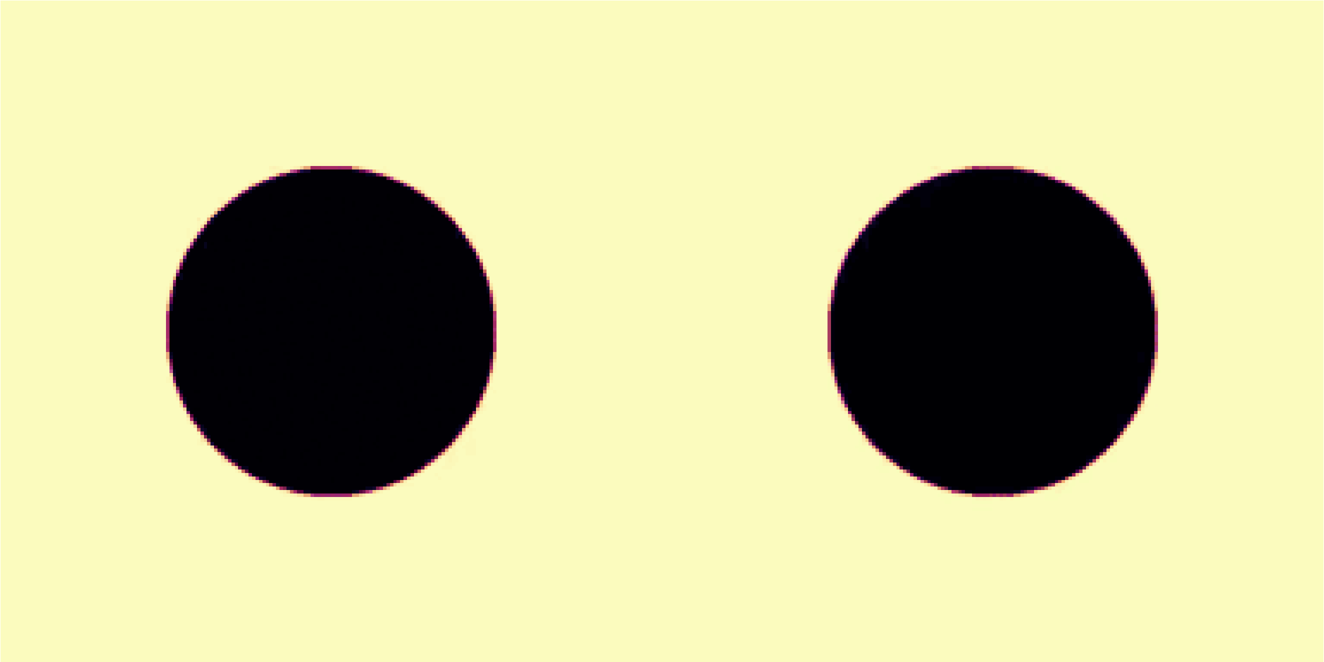
ti.BitpackedFields

- Packing multiple quantized primitives into one SNode:

```
fp_8_8 = ti.types.quant.float(exp=8, frac=8)
Q = ti.Vector.field(4, dtype=fp_8_8)
bitpack = ti.BitpackedFields(max_num_bits=64)
bitpack.place(Q)
ti.root.dense(ti.ij, (N, N)).place(bitpack)
```

And it looks bad...

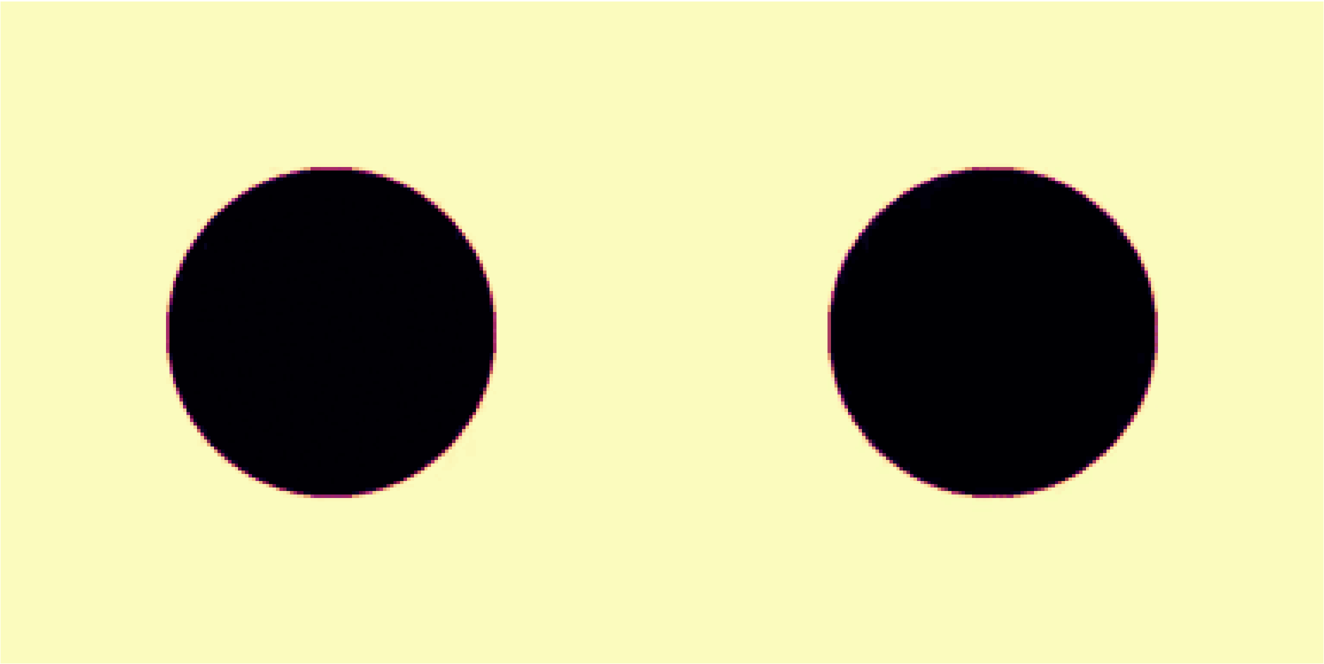
```
fp_8_8 = ti.types.quant.float(exp=8, frac=8)
Q = ti.Vector.field(4, dtype=fp_8_8)
bitpack = ti.BitpackedFields(max_num_bits=64)
bitpack.place(Q)
ti.root.dense(ti.ij, (N, N)).place(bitpack)
```



Full precision Quantized

Why?

- We reduced the sign and fraction bits too much from 24 to 8

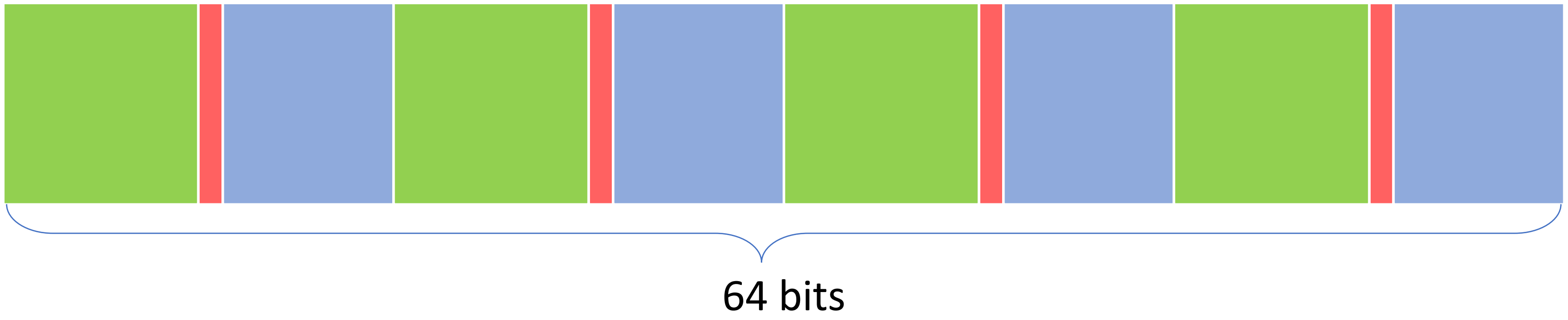


Full precision　　　　Quantized

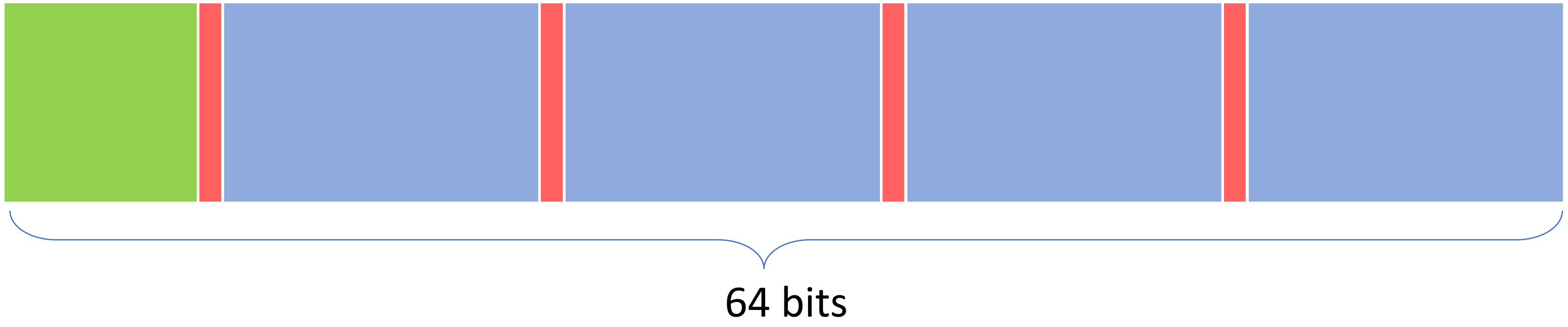
Share exponent using shared_exponent

```
fp_8_8 = ti.types.quant.float(exp=8, frac=8)
Q = ti.Vector.field(4, dtype=fp_8_8)
bitpack = ti.BitpackedFields(max_num_bits=64)
bitpack.place(Q)
ti.root.dense(ti.ij, (N, N)).place(bitpack)
```

```
fp_8_14 = ti.types.quant.float(exp=8, frac=14)
Q = ti.Vector.field(4, dtype=fp_8_14)
bitpack = ti.BitpackedFields(max_num_bits=64)
bitpack.place(Q, shared_exponent=True)
ti.root.dense(ti.ij, (N, N)).place(bitpack)
```



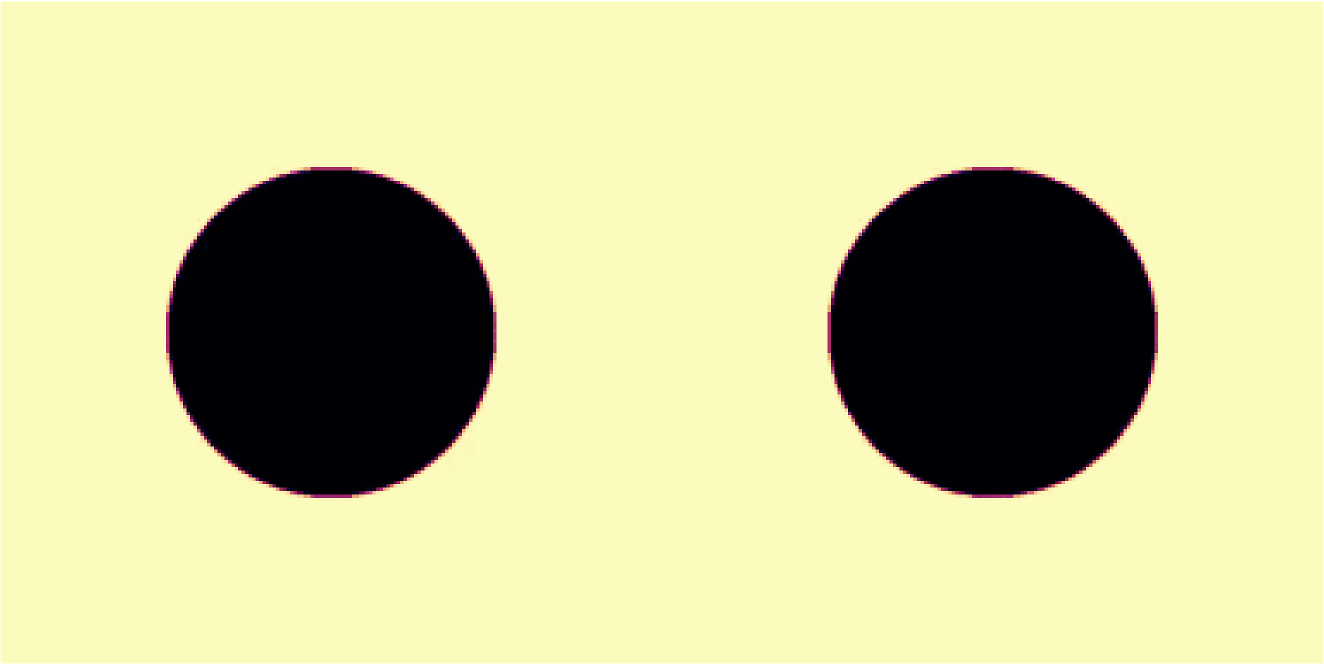
Exponent: 8 bits
Sign: 1 bit
Fraction: 7bits



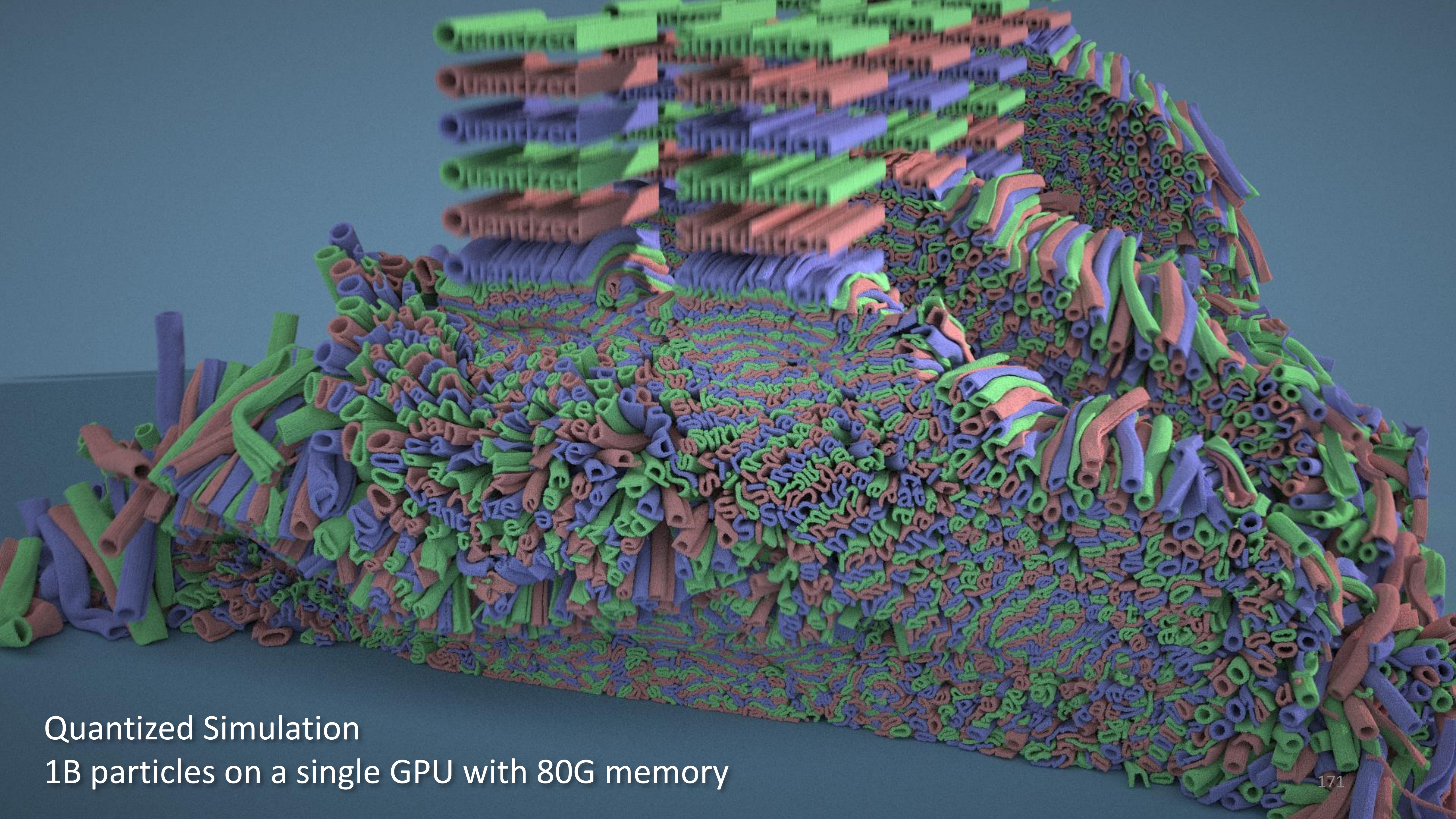
Shared Exponent: 8 bits
Sign: 1 bit
Fraction: 13bits

Now we have a better looking simulation

- … and reduce the memory consumption of Q by half.



Full precision Quantized



Quantized Simulation
1B particles on a single GPU with 80G memory

Remark: Data Quantization

- Quantization
  - Trade off precision for less memory consumption
- QuanTaichi
  - `ti.types.quant`: quantized primitive data types
  - `ti.BitpackedFields`: quantized fields
    - `shared_exponent`: quantized fields with shared exponent

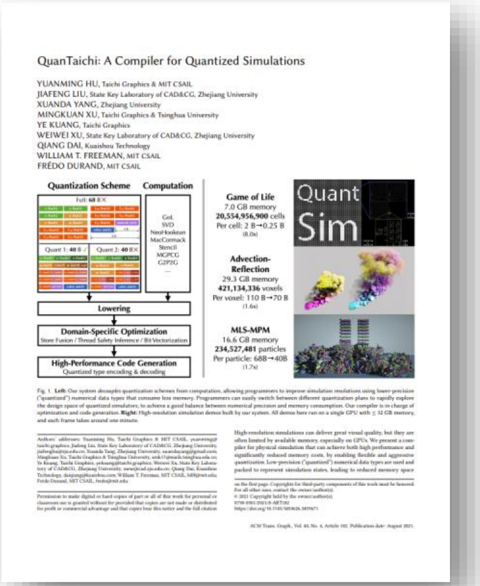
Remark: Data Quantization

- Quantization
  - Trade off precision for less memory consumption
- QuanTaichi
  - `ti.types.quant`: quantized primitive data types
  - `ti.BitpackedFields`: quantized fields
    - `shared_exponent`: quantized fields with shared exponent

Further Readings

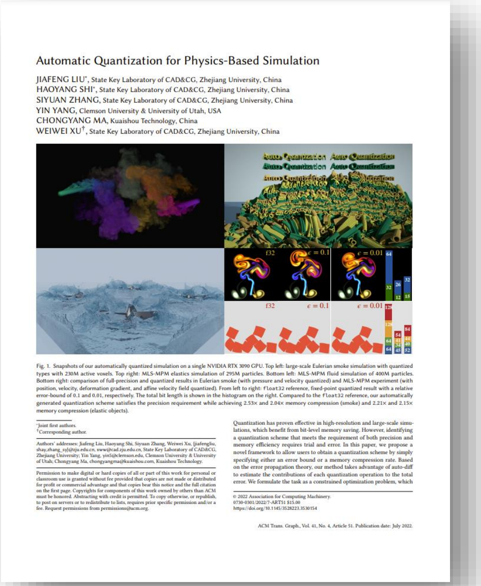
- Technical details described in [Hu et al. 2021]
  - An automatic version of QuanTaichi: [Liu et al. 2022]
- Check more examples at the QuanTaichi repo

QuanTaichi


QuanTaichi: A Compiler for Quantized Simulations

YUANMING HU, Taichi Graphics & MIT CSAIL
JIAFENG LIU, State Key Laboratory of CAD&CG, Zhejiang University
XUANDA YANG, Zhejiang University
MINGKUAN XU, Taichi Graphics & Tsinghua University
YE KUANG, Taichi Graphics
WEIWEI XU, State Key Laboratory of CAD&CG, Zhejiang University
QIANG DAI, Kuaishou Technology
WILLIAM T. FREEMAN, MIT CSAIL
FRÉDO DURAND, MIT CSAIL

[Hu et al. 2021]

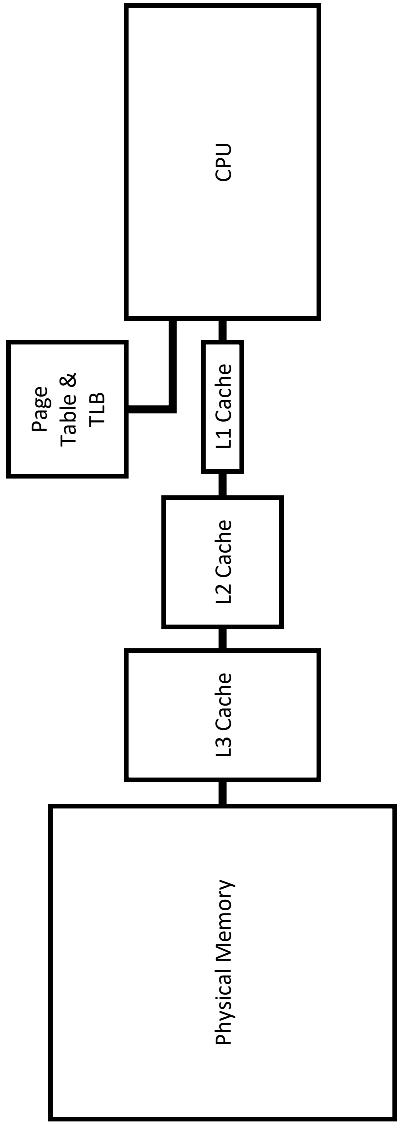

Automatic Quantization for Physics-Based Simulation

JIAFENG LIU*, State Key Laboratory of CAD&CG, Zhejiang University, China
HAOYANG SHI*, State Key Laboratory of CAD&CG, Zhejiang University, China
SIYUAN ZHANG, State Key Laboratory of CAD&CG, Zhejiang University, China
YIN YANG, Clemson University & University of Utah, USA
CHONGYANG MA, Kuaishou Technology, China
WEIWEI XU†, State Key Laboratory of CAD&CG, Zhejiang University, China

[Liu et al. 2022]

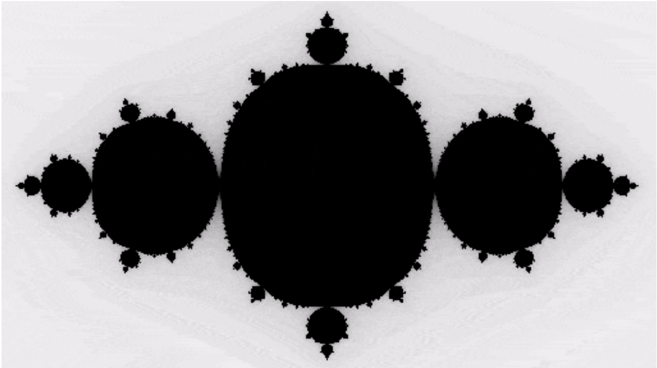
To summarize

- Data matters a lot in computer graphics applications
- The SNode system in Taichi
  - Changing data layout
  - Constructing sparse data structures
- MeshTaichi
  - Parallelize mesh-based operations in patches
  - Reorder / cache mesh attributes without changing computation
- Tell the Taichi compiler to optimize cached data manually
  - `ti.block_local / ti.mesh_local`
- QuanTaichi
  - Reduce memory consumption using quantization



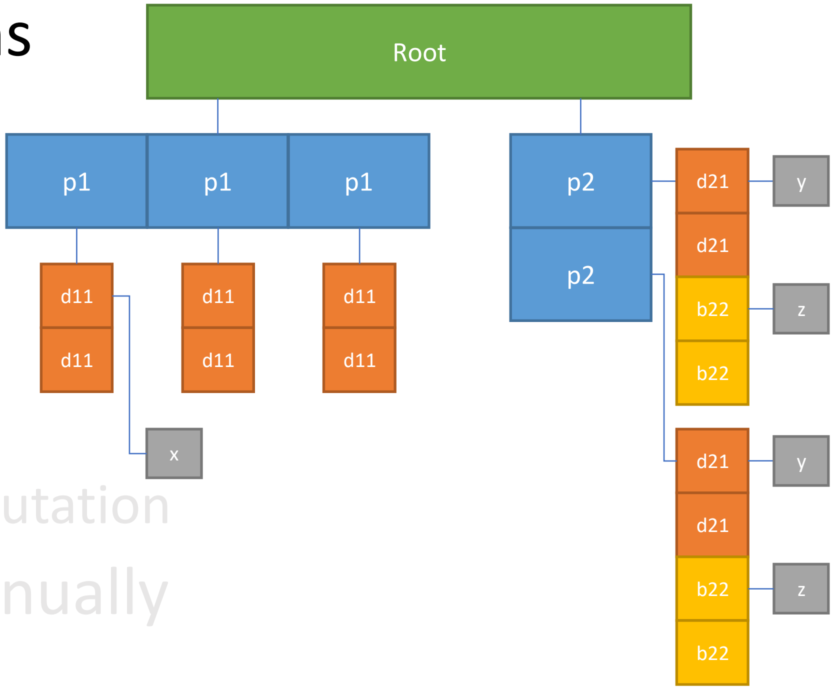
To summarize

- Data matters a lot in computer graphics applications
- The SNode system in Taichi
  - Changing data layout
  - Constructing sparse data structures
- MeshTaichi
  - Parallelize mesh-based operations in patches
  - Reorder / cache mesh attributes without changing computation
- Tell the Taichi compiler to optimize cached data manually
  - `ti.block_local / ti.mesh_local`
- QuanTaichi
  - Reduce memory consumption using quantization



To summarize

- Data matters a lot in computer graphics applications
- The SNode system in Taichi
  - Changing data layout
  - Constructing sparse data structures
- MeshTaichi
  - Parallelize mesh-based operations in patches
  - Reorder / cache mesh attributes without changing computation
- Tell the Taichi compiler to optimize cached data manually
  - `ti.block_local / ti.mesh_local`
- QuanTaichi
  - Reduce memory consumption using quantization

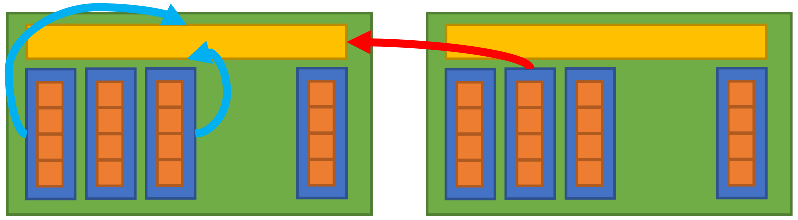


To summarize

- Data matters a lot in computer graphics applications
- The SNode system in Taichi
  - Changing data layout
  - Constructing sparse data structures
- MeshTaichi
  - Parallelize mesh-based operations in patches
  - Reorder / cache mesh attributes without changing computation
- Tell the Taichi compiler to optimize cached data manually
  - `ti.block_local / ti.mesh_local`
- QuanTaichi
  - Reduce memory consumption using quantization

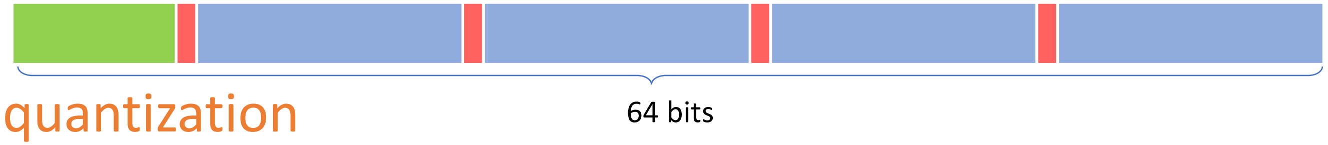
To summarize

- Data matters a lot in computer graphics applications
- The SNode system in Taichi
  - Changing data layout
  - Constructing sparse data structures
- MeshTaichi
  - Parallelize mesh-based operations in patches
  - Reorder / cache mesh attributes without changing computation
- Tell the Taichi compiler to optimize cached data manually
  - `ti.block_local / ti.mesh_local`
- QuanTaichi
  - Reduce memory consumption using quantization



To summarize

- Data matters a lot in computer graphics applications
- The SNode system in Taichi
  - Changing data layout
  - Constructing sparse data structures
- MeshTaichi
  - Parallelize mesh-based operations in patches
  - Reorder / cache mesh attributes without changing computation
- Tell the Taichi compiler to optimize cached data manually
  - `ti.block_local / ti.mesh_local`
- QuanTaichi
  - Reduce memory consumption using quantization



Thank you

[Taichi Lang]

[Taichi Lang GitHub]

[Twitter]

SIGGRAPH ASIA 2022 DAEGU