太极图形课

第04讲 Sparse Matrix, Debugging and Code Optimization

taichi

太极图形课

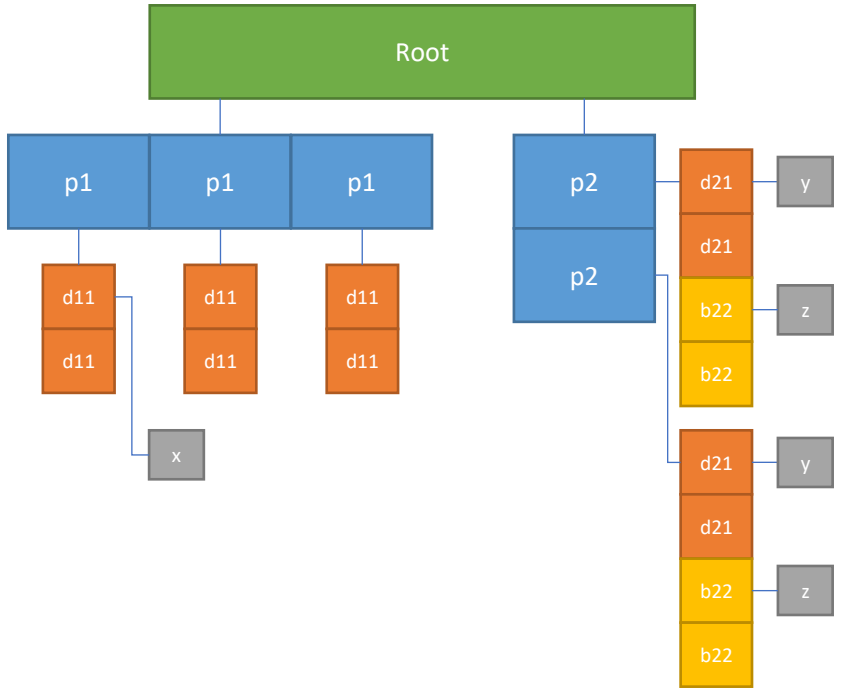
第04讲 Sparse Matrix, Debugging and Code Optimization

taichi

Recap

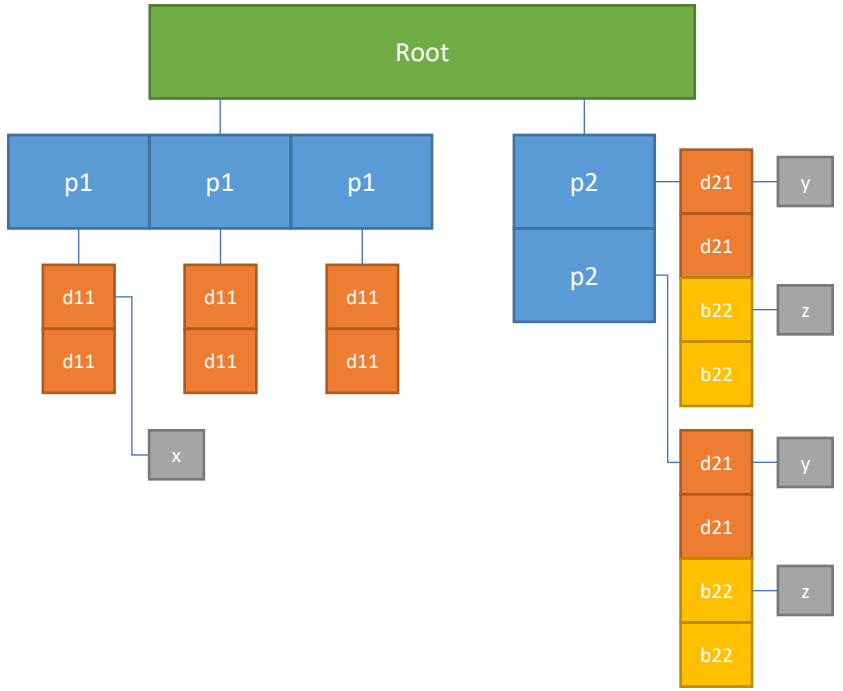
- Advanced (dense + sparse) Data Layouts
  - Better memory access ≈ better performance (why DOP)
  - Data structures are decoupled from the computation (DOP how)
  - All *fields* in Taichi are *SNode-trees*

```
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)
```



Recap

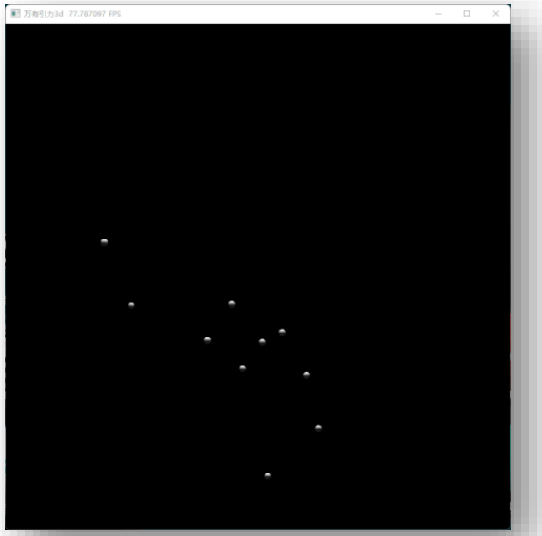
- Advanced (dense + sparse) Data Layouts
  - Changing layouts
    - Row/col-majored, flat v.s. hierarchical, AoS v.s. SoA
  - From dense to sparse
    - .dense() → .pointer(), .bitmasked()



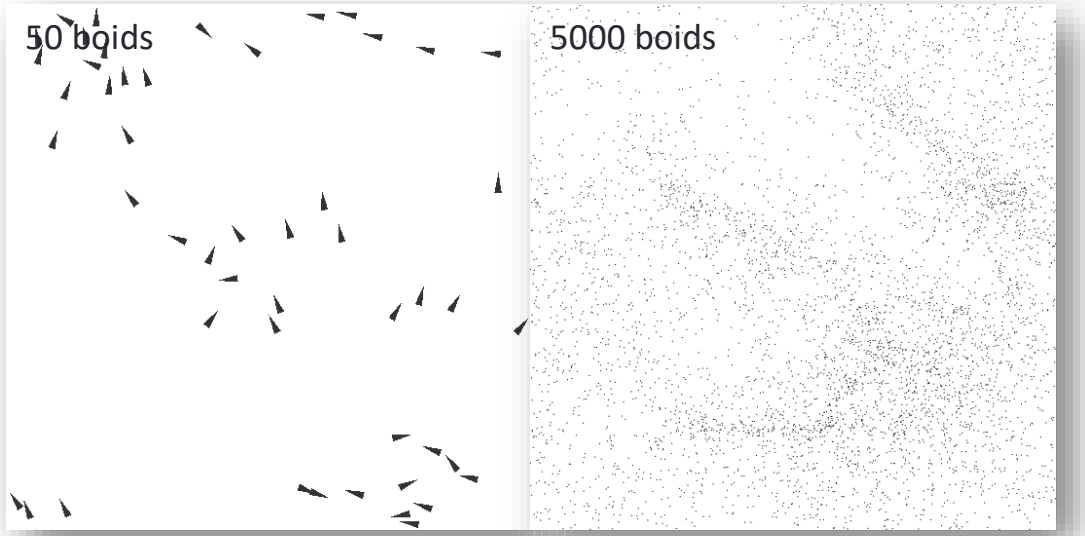
Recap

- Advanced (dense + sparse) Data Layouts

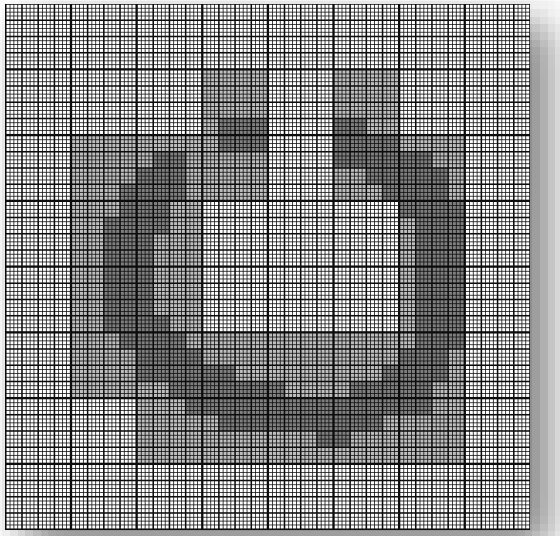
Excellent homework assignments



3D N-body
@cflw

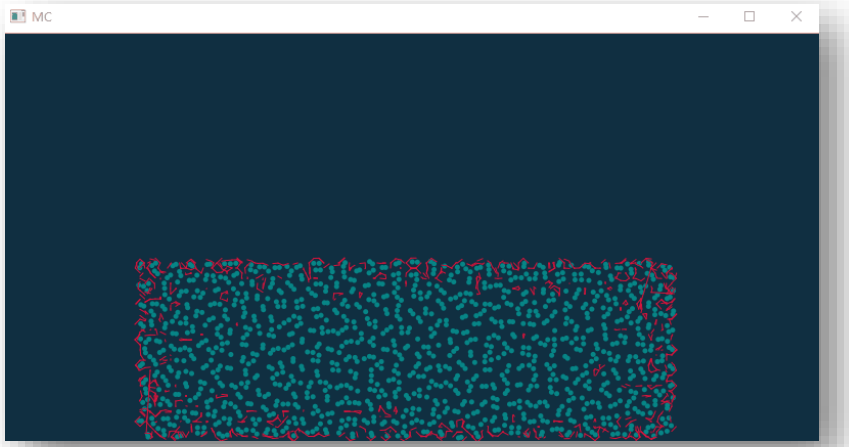


Flocking
@SIGUSR97

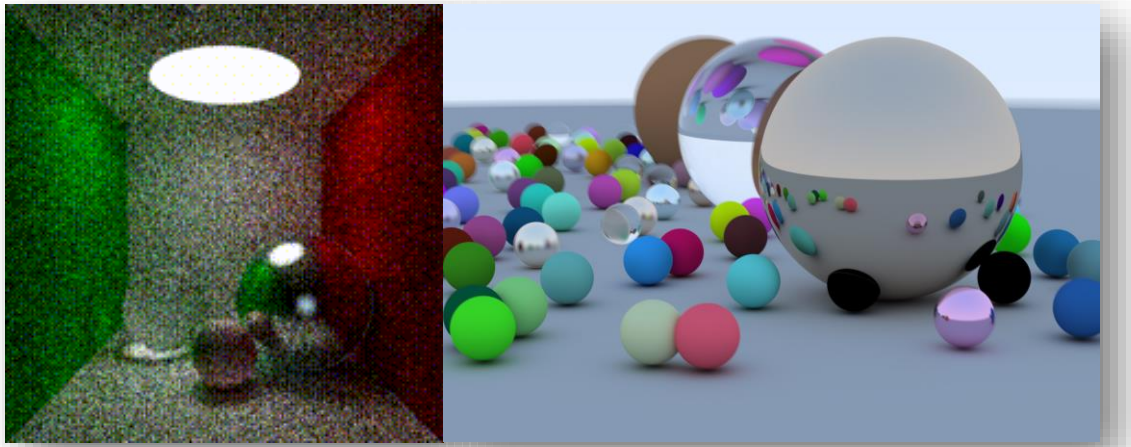


Sparse Field Test
@Vineyo

Excellent homework assignments



IISPH + Marching Cube
@MengMeng3399



Ray tracing in one weekend
@0xrabbyte

“还没有上课就把作业做完了，要不要来课上讲一”奖

Outline today

- Sparse matrix and sparse linear algebra
- Debugging a Taichi project
- Optimizing your Taichi code

Sparse Matrix

Sparse matrices in Taichi

“在助教李喆昊的帮助下，我实现了基于BESO方法的拓扑优化，……，求解器依赖了scipy。”

@AlbertLiDesign

Sparse matrix (an naïve Taichi implementation)

- Hierarchical sparse SNode-tree with bitmasked bottom level blocks
- Pros
  - Stays in your GPU
  - Consistent with your other data
- Cons
  - Hard to maintain
    - A+B / A-B / A@B / A@x ?
    - Ax = b?

Sparse matrix in Taichi (courtesy of 禹鹏/刘嘉枫)

- Sparse matrices are frequently used when solving linear systems in science and engineering. Taichi provides programmers with useful APIs for sparse matrices.

- Build a sparce matrix
- Sparse matrix operations
- Sparse linear solver

Build a sparse matrix

- Create a builder using *ti.SparseMatrixBuilder().*
- Fill the builder with your matrices' data.
- Create sparse matrices from the builder.

```python
n = 4
# step 1: create sparse matrix builder
K = ti.SparseMatrixBuilder(n, n, max_num_triplets=100)

@ti.kernel
def fill(A: ti.sparse_matrix_builder()):
    for i in range(n):
        A[i, i] += 1

# step 2: fill the builder with data.
fill(K)

print(">>>> K.print_triplets()")
K.print_triplets()
# outputs:
# >>>> K.print_triplets()
# n=4, m=4, num_triplets=4 (max=100)(0, 0) val=1.0(1, 1) val=1.0(2, 2)
val=1.0(3, 3) val=1.0

# step 3: create a sparse matrix from the builder.
A = K.build()
print(">>>> A = K.build()")
print(A)
# outputs:
# >>>> A = K.build()
# [1, 0, 0, 0]
# [0, 1, 0, 0]
# [0, 0, 1, 0]
# [0, 0, 0, 1]
```

Sparse matrix operations

- Summation: A+B
- Subtraction: A-B
- Scalar multiplication: c*A or A*c
- Element-wise multiplication: A*B
- Matrix multiplication: A@B
- Matrix-vector multiplication: A@b
- Transpose: A.transpose()
- Element access: A[i. j]

Sparse linear solver

```
# factorize
solver = ti.SparseSolver(solver_type="LLT")
solver.analyze_pattern(A)
solver.factorize(A)

# solve
x = solver.solve(b)

# check stats
isSuccessful = solver.info()
print(">>>> Solve sparse linear systems Ax = b with the solution x:")
print(x)
print(f">>>> Computation was successful?: {isSuccessful}")
```

Sparse linear solver

```
# factorize
solver = ti.SparseSolver(solver_type="LLT")
solver.analyze_pattern(A)
solver.factorize(A)

# solve
x = solver.solve(b)

# check stats
isSuccessful = solver.info()
print(">>>> Solve sparse linear systems Ax = b with the solution x:")
print(x)
print(f">>>> Computation was successful?: {isSuccessful}")
```

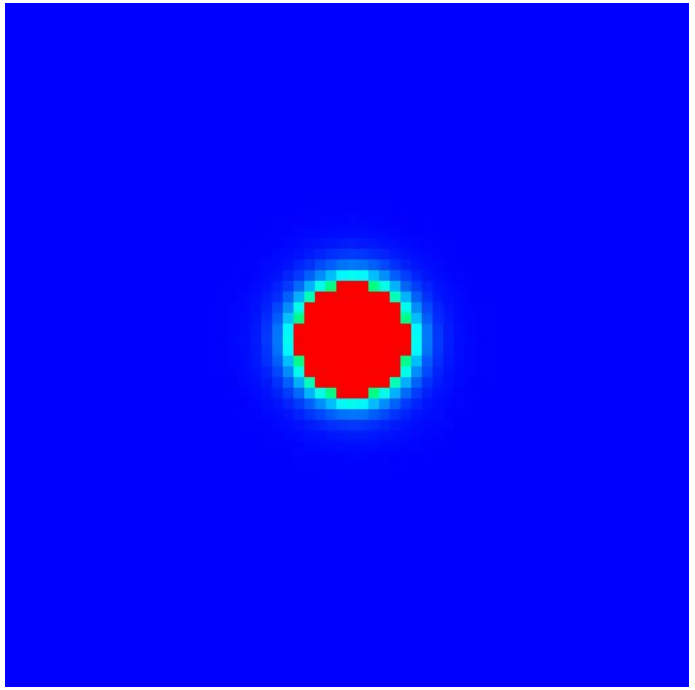
Sparse linear solver

```
# factorize
solver = ti.SparseSolver(solver_type="LLT")
solver.analyze_pattern(A)
solver.factorize(A)

# solve
x = solver.solve(b)

# check stats
isSuccessful = solver.info()
print(">>>> Solve sparse linear systems Ax = b with the solution x:")
print(x)
print(f">>>> Computation was successful?: {isSuccessful}")
```

Linear solver, one example

- 鸡兔同笼
  - $\begin{bmatrix} 1 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} n_{鸡} \\ n_{兔} \end{bmatrix} = \begin{bmatrix} n_{头} \\ n_{腿} \end{bmatrix}$.
  - Given $n_{头}$ and $n_{腿}$, compute $n_{鸡}$ and $n_{兔}$
  - $\mathbf{A}x = b$

Linear solver, another example: diffusion



Diffusion

- $\frac{\partial T}{\partial t} = \kappa \nabla^2 T$
  - T:temperature, t: time, κ: thermal diffusivity, ∇^2: Laplace operator
  - $\nabla^2 T = \nabla^2_{xx} T + \nabla^2_{yy} T$ in 2D

Diffusion

- $\frac{\partial T}{\partial t} = \kappa \nabla^2 T$
  - T:temperature, t: time, κ: thermal diffusivity, ∇^2: Laplace operator

- (Spatially-)discretized version:
  - $\frac{\partial T_{i,j}}{\partial t} = \kappa \frac{\frac{T_{i+1,j}-T_{i,j}}{\Delta x} - \frac{T_{i,j}-T_{i-1,j}}{\Delta x}}{\Delta x} + \frac{\frac{T_{i,j+1}-T_{i,j}}{\Delta x} - \frac{T_{i,j}-T_{i,j-1}}{\Delta x}}{\Delta x}$
  - $\frac{\partial T_{i,j}}{\partial t} = \frac{\kappa}{\Delta x^2}(-4T_{i,j} + T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$

	$T_{i-1,j}$	
$T_{i,j-1}$	$T_{i,j}$	$T_{i,j+1}$
	$T_{i+1,j}$	

Diffusion

- $\frac{\partial T}{\partial t} = \kappa \nabla^2 T$
  - T:temperature, t: time, κ: thermal diffusivity, ∇^2: Laplace operator

- (Spatially-)discretized version:
  - $\frac{\partial T_{i,j}}{\partial t} = \frac{\kappa}{\Delta x^2}(-4T_{i,j} + T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$

- (Temporally-)discretized version (explicit):
  - $\frac{T_{n+1} - T_n}{\Delta t} = \kappa \nabla^2 T_n$

	$T_{i-1,j}$	
$T_{i,j-1}$	$T_{i,j}$	$T_{i,j+1}$
	$T_{i+1,j}$	

Explicit diffusion (Taichi code)

- (Spatially-)discretized version:
  - $\frac{\partial T_{i,j}}{\partial t} = \frac{\kappa}{\Delta x^2}(-4T_{i,j} + T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1})$

- (Temporally-)discretized version (explicit):
  - $\frac{T_{n+1} - T_n}{\Delta t} = \kappa \nabla^2 T_n$

```
@ti.kernel
def diffuse(dt: ti.f32):
    c = dt * k / dx**2
    for i,j in t_n:
        t_np1[i,j] = t_n[i,j]
        if i-1 >= 0:
            t_np1[i, j] += c * (t_n[i-1, j] - t_n[i, j])
        if i+1 < n:
            t_np1[i, j] += c * (t_n[i+1, j] - t_n[i, j])
        if j-1 >= 0:
            t_np1[i, j] += c * (t_n[i, j-1] - t_n[i, j])
        if j+1 < n:
            t_np1[i, j] += c * (t_n[i, j+1] - t_n[i, j])
```

Explicit diffusion (matrix representation)

- (Temporally-)discretized version (explicit):
  - $\frac{T_{n+1}-T_n}{\Delta t} = \frac{\kappa}{\Delta x^2}\mathbf{D}T_n$ ➔ $T_{n+1} = \left(\mathbf{I} + \frac{\Delta t * \kappa}{\Delta x^2}\mathbf{D}\right)T_n$

$$\mathbf{D} = \begin{bmatrix} -2 & +1 & & +1 & & & & & \\ +1 & -3 & +1 & & +1 & & & & \\ & +1 & -2 & & & +1 & & & \\ +1 & & & -3 & +1 & & +1 & & \\ & +1 & & +1 & -4 & +1 & & +1 & \\ & & +1 & & +1 & -3 & & & +1 \\ & & & +1 & & & -2 & +1 & \\ & & & & +1 & & +1 & -3 & +1 \\ & & & & & +1 & & +1 & -2 \end{bmatrix}$$

	$T_{i-1,j}$	
$T_{i,j-1}$	$T_{i,j}$	$T_{i,j+1}$
	$T_{i+1,j}$	

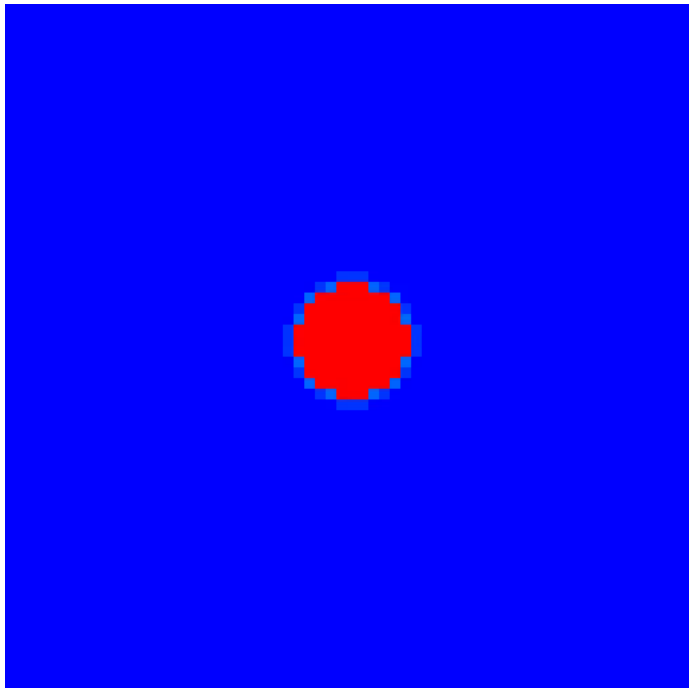
Explicit diffusion (matrix representation)

- (Temporally-)discretized version (explicit):
  - $\frac{T_{n+1}-T_n}{\Delta t} = \frac{\kappa}{\Delta x^2}\mathbf{D}T_n$ ➔ $T_{n+1} = \left(\mathbf{I} + \frac{\Delta t * \kappa}{\Delta x^2}\mathbf{D}\right)T_n$

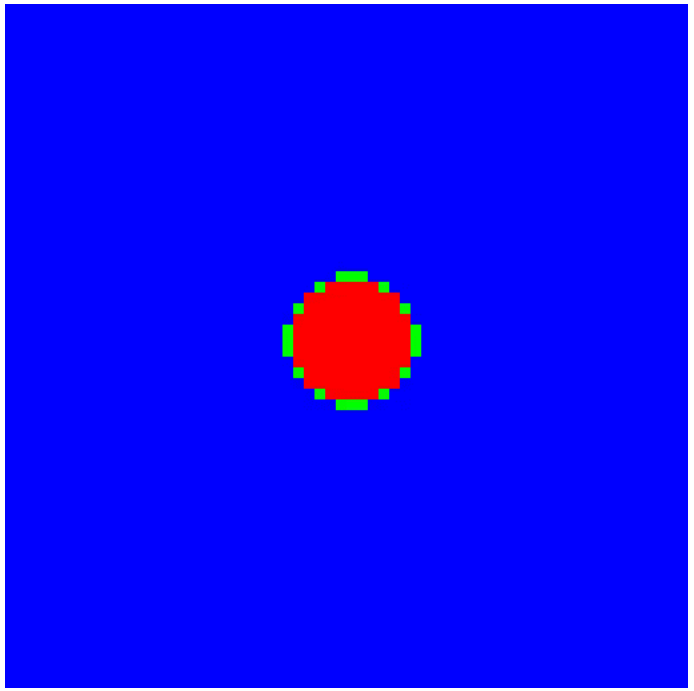
```
def diffuse(dt: ti.f32):
    c = dt * k / dx**2
    IpcD = I + c*D
    t_np1.from_numpy(IpcD@t_n.to_numpy()
    # t_np1 = t_n + c*D*t_n
```

```
@ti.kernel
def fillDiffusionMatrixBuilder(A:
ti.sparse_matrix_builder()):
    for i,j in ti.ndrange(n, n):
        count = 0
        if i-1 >= 0:
            A[ind(i,j), ind(i-1,j)] += 1
            count += 1
        if i+1 < n:
            A[ind(i,j), ind(i+1,j)] += 1
            count += 1
        if j-1 >= 0:
            A[ind(i,j), ind(i,j-1)] += 1
            count += 1
        if j+1 < n:
            A[ind(i,j), ind(i,j+1)] += 1
            count += 1
        A[ind(i,j), ind(i,j)] += -count
```

Explicit Diffusion



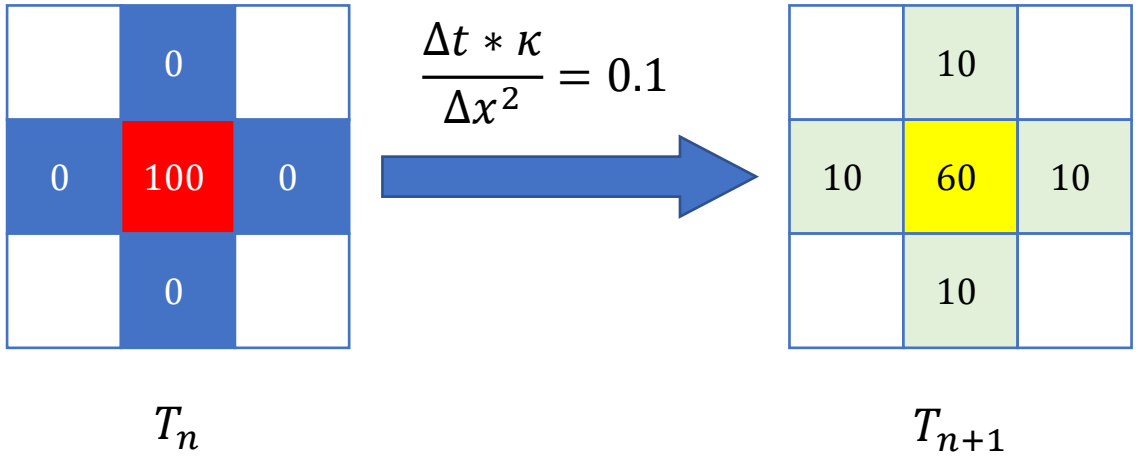
$\kappa = 50$

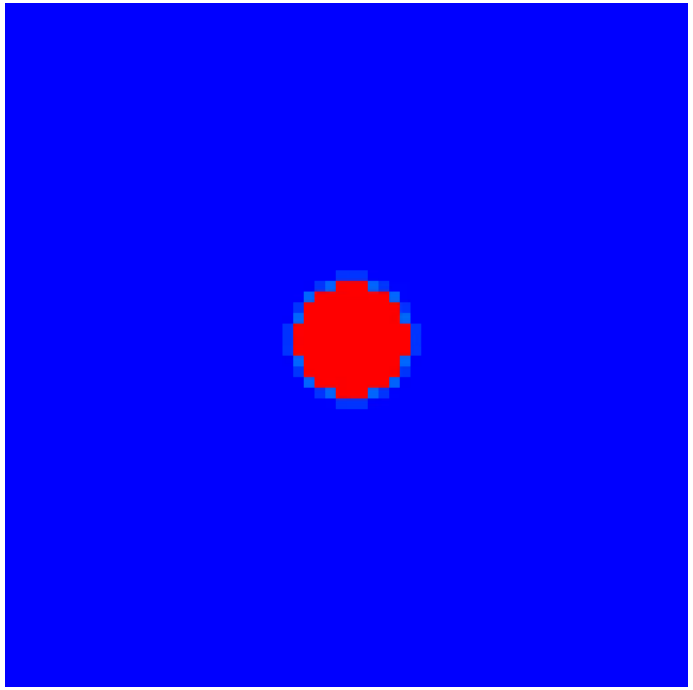


$\kappa = 500$

Explicit diffusion (Explosion)

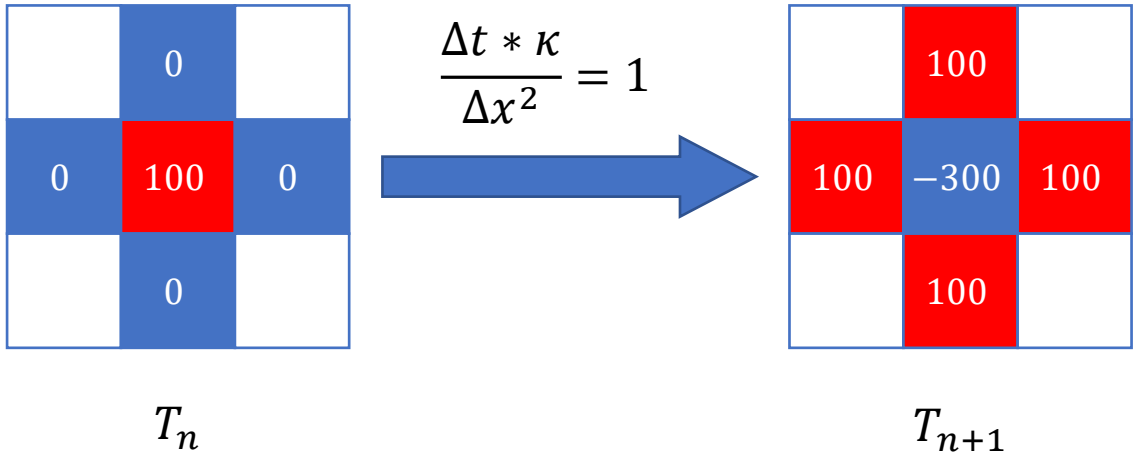
- (Temporally-)discretized version (explicit):
  - $\frac{T_{n+1}-T_n}{\Delta t} = \frac{\kappa}{\Delta x^2}\mathbf{D}T_n$ ➔ $T_{n+1} = \left(\mathbf{I} + \frac{\Delta t * \kappa}{\Delta x^2}\mathbf{D}\right)T_n$

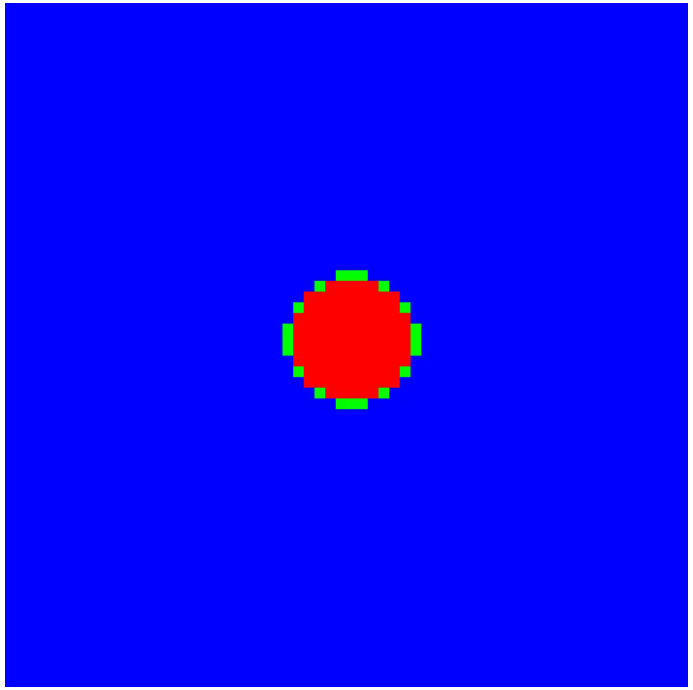




Explicit diffusion (Explosion)

- (Temporally-)discretized version (explicit):
  - $\frac{T_{n+1}-T_n}{\Delta t} = \frac{\kappa}{\Delta x^2}\mathbf{D}T_n$ ➔ $T_{n+1} = \left(\mathbf{I} + \frac{\Delta t * \kappa}{\Delta x^2}\mathbf{D}\right)T_n$





Implicit diffusion (matrix representation)

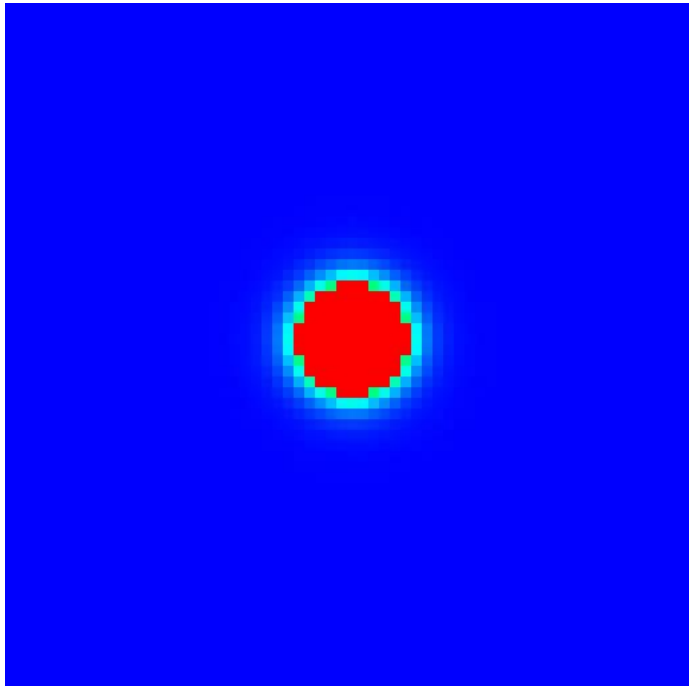
- (Temporally-)discretized version (implicit):
  - $\frac{T_{n+1}-T_n}{\Delta t} = \frac{\kappa}{\Delta x^2}\mathbf{D}T_{n+1}$ ➔ $T_{n+1} = \left(\mathbf{I} - \frac{\Delta t * \kappa}{\Delta x^2}\mathbf{D}\right)^{-1} T_n$

```
def diffuse(dt: ti.f32):
    c = dt * k / dx**2
    ImcD = I - c*D

    # linear solve: factorize
    solver = ti.SparseSolver(solver_type="LLT")
    solver.analyze_pattern(ImcD)
    solver.factorize(ImcD)

    # linear solve: solve
    t_np1.from_numpy(solver.solve(t_n))
    # t_np1 = t_n + c*D*t_np1
```

```
@ti.kernel
def fillDiffusionMatrixBuilder(A:
ti.sparse_matrix_builder()):
    for i,j in ti.ndrange(n, n):
        count = 0
        if i-1 >= 0:
            A[ind(i,j), ind(i-1,j)] += 1
            count += 1
        if i+1 < n:
            A[ind(i,j), ind(i+1,j)] += 1
            count += 1
        if j-1 >= 0:
            A[ind(i,j), ind(i,j-1)] += 1
            count += 1
        if j+1 < n:
            A[ind(i,j), ind(i,j+1)] += 1
            count += 1
        A[ind(i,j), ind(i,j)] += -count
```

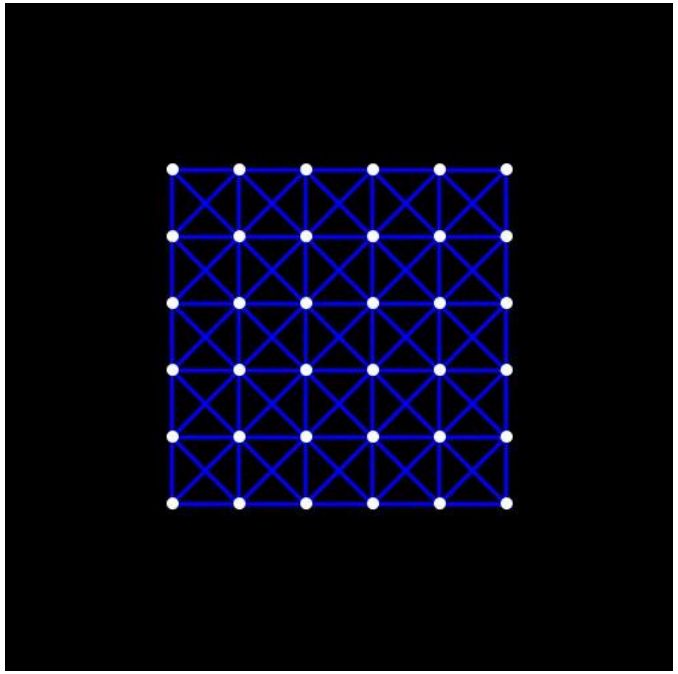
Diffusion: [Code]

- https://github.com/taichiCourse01/--Diffuse

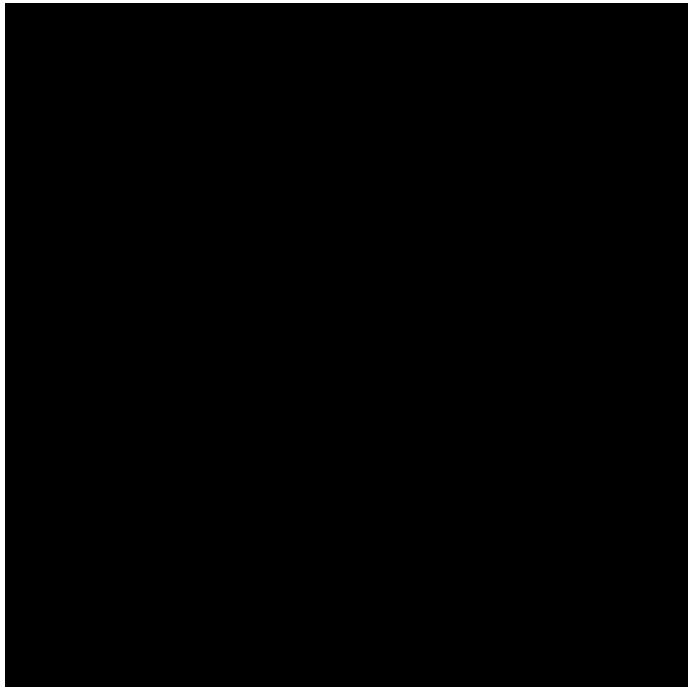


$\kappa = 2000$

Sparse linear system, two more examples



Implicit Mass-spring Simulation @禹鹏 [code]



Stable Fluid @刘嘉枫 [code]

Remark

- Current APIs and docs: [Link]
  - [SparseMatrixBuilder] = ti.SparseMatrixBuilder()
  - [SparseMatrix] = [SparseMatrixBuilder].build()
  - [NumpyArray] = [SparseMatrix].solve([Field])
- Supports the CPU backend only
- Matrix-vector multiplication and sparse linear system solver returns
  - A numpy array
- Useful for the purpose of prototyping
  - We'd suggest a Multi-grid Preconditioned Conjugate Gradient solver for your final product

Remark

- Update your Taichi to 0.8.3 or above to use these features of sparse matrices
  - python -m pip install taichi==0.8.3
  - python -m pip install taichi --upgrade

- Raise issues at: https://github.com/taichi-dev/taichi/issues

Debugging

Debugging

- Debugging tips in Taichi
- Mostly seen problems

Run-time print

```
@ti.kernel
def inside_taichi_scope():
    x = 256
    print('hello', x)
    #=> hello 256

    print('hello', x * 2 + 200)
    #=> hello 712

    print('hello', x, sep='')
    #=> hello256

    print('hello', x, sep='', end='')
    print('world', x, sep='')
    #=> hello256world256

    m = ti.Matrix([[2, 3, 4], [5, 6, 7]])
    print('m =', m)
    #=> m = [[2, 3, 4], [5, 6, 7]]

    v = ti.Vector([3, 4])
    print('v =', v)
    #=> v = [3, 4]

    ray = ti.Struct({
        "ori": ti.Vector([0.0, 0.0, 0.0]),
        "dir": ti.Vector([0.0, 0.0, 1.0]),
        "len": 1.0
        })
    print('ray.ori =', ray.ori, ', ray.dir =', ray.dir, ', ray.len =', ray.len)
    #=> ray.ori = [0.0, 0.0, 0.0], ray.dir = [0.0, 0.0, 1.0], ray.len = 1.0
```

Run-time print

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

@ti.kernel
def kern():
    print('inside kernel')

print('before kernel')
kern()
print('after kernel')
ti.sync()
print('after sync')

# print ==>
# before kernel
# after kernel
# inside kernel
# after sync
```

Run-time print

- Hurts your performance, for sure
  - print requires a system call
- Produces randomly ordered results in your parallel for loops
- Supports comma-separated parameters in the Taichi Scope

```
@ti.kernel
def foo():
    print('a[0] = ', a[0]) # right
    print(f'a[0] = {a[0]}') # wrong, f-string is not supported
    print("a[0] = %f" % a[0]) # wrong, formatted string is not supported
```

Compile time ti.static_print

- Print *python-scope objects* and *constants* in Taichi scope
  - Will print only once at compile-time

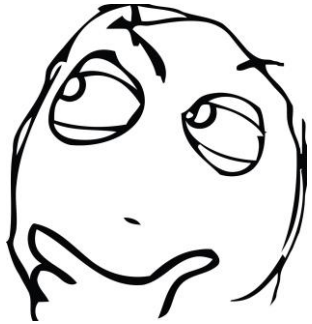
```
x = ti.field(ti.f32, (2, 3))
y = 1
A = ti.Matrix([[1, 2], [3, 4], [5, 6]])

@ti.kernel
def inside_taichi_scope():
    ti.static_print(y)
    # => 1
    ti.static_print(x.shape)
    # => (2, 3)
    ti.static_print(A.n)
    # => 3
    for i in range(4):
        ti.static_print(A.m)
        # => 2
        # will only print once
```

Visualize your fields

- Print a Taichi field / numpy array will truncate your results:

```
x = ti.field(ti.f32, (256, 256))

@ti.kernel
def foo():
    for i,j in x:
        x[i,j] = (i+j)/512.0

foo()
print(x)
```

```
[Taichi] version 0.8.3, llvm 10.0.0, commit 021af5d2, win, python 3.8.10
[Taichi] Starting on arch=x64
[[0.         0.00195312 0.00390625 ... 0.49414062 0.49609375 0.49804688]
 [0.00195312 0.00390625 0.00585938 ... 0.49609375 0.49804688 0.5       ]
 [0.00390625 0.00585938 0.0078125  ... 0.49804688 0.5        0.5019531 ]
 ...
 [0.49414062 0.49609375 0.49804688 ... 0.98828125 0.9902344  0.9921875 ]
 [0.49609375 0.49804688 0.5        ... 0.9902344  0.9921875  0.9941406 ]
 [0.49804688 0.5        0.5019531  ... 0.9921875  0.9941406  0.99609375]]
```

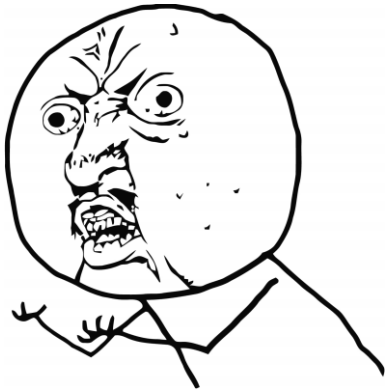


Visualize your fields

- Turn a Taichi field into a list if you REALLY wants to print the full list:

```
x = ti.field(ti.f32, (256, 256))

@ti.kernel
def foo():
    for i,j in x:
        x[i,j] = (i+j)/512.0

foo()
print(x.to_numpy().tolist())
```

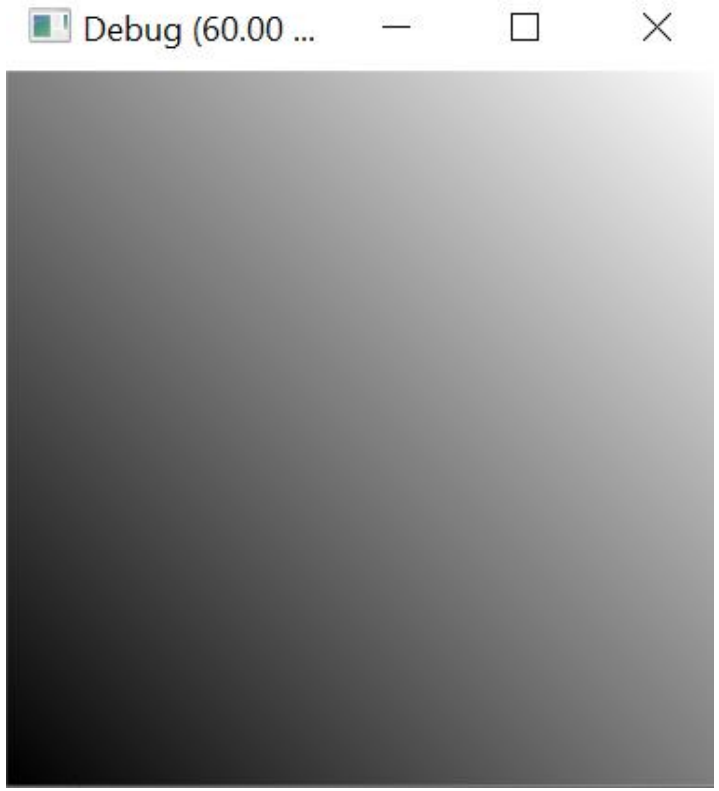
0.21679687, 0.21875, 0.22070312, 0.2226562, 0.2246093, 0.2265625, 0.2285156, 0.2304687, 0.2324218, 0.234375, 0.2363281, 0.2382812, 0.240
267578125, 0.26953125, 0.271484375, 0.2734375, 0.275390625, 0.27734375, 0.279296875, 0.28125, 0.283203125, 0.28515625, 0.287109375, 0.2890625, 0.2910
0.318359375, 0.3203125, 0.322265625, 0.32421875, 0.326171875, 0.328125, 0.330078125, 0.33203125, 0.333984375, 0.3359375, 0.337890625, 0.33984375, 0.3
5, 0.369140625, 0.37109375, 0.373046875, 0.375, 0.376953125, 0.37890625, 0.380859375, 0.3828125, 0.384765625, 0.38671875, 0.388671875, 0.390625, 0.39
75, 0.419921875, 0.421875, 0.423828125, 0.42578125, 0.427734375, 0.4296875, 0.431640625, 0.43359375, 0.435546875, 0.4375, 0.439453125, 0.44140625, 0.
75, 0.470703125, 0.47265625, 0.474609375, 0.4765625, 0.478515625, 0.48046875, 0.482421875, 0.484375, 0.486328125, 0.48828125, 0.490234375, 0.4921875,
84375, 0.0234375, 0.025390625, 0.02734375, 0.029296875, 0.03125, 0.033203125, 0.03515625, 0.037109375, 0.0390625, 0.041015625, 0.04296875, 0.04492187
265625, 0.07421875, 0.076171875, 0.078125, 0.080078125, 0.08203125, 0.083984375, 0.0859375, 0.087890625, 0.08984375, 0.091796875, 0.09375, 0.09570312
.123046875, 0.125, 0.126953125, 0.12890625, 0.130859375, 0.1328125, 0.134765625, 0.13671875, 0.138671875, 0.140625, 0.142578125, 0.14453125, 0.146484
173828125, 0.17578125, 0.177734375, 0.1796875, 0.181640625, 0.18359375, 0.185546875, 0.1875, 0.189453125, 0.19140625, 0.193359375, 0.1953125, 0.19726
0.224609375, 0.2265625, 0.228515625, 0.23046875, 0.232421875, 0.234375, 0.236328125, 0.23828125, 0.240234375, 0.2421875, 0.244140625, 0.24609375, 0.2
0.275390625, 0.27734375, 0.279296875, 0.28125, 0.283203125, 0.28515625, 0.287109375, 0.2890625, 0.291015625, 0.29296875, 0.294921875, 0.296875, 0.298
, 0.326171875, 0.328125, 0.330078125, 0.33203125, 0.333984375, 0.3359375, 0.337890625, 0.33984375, 0.341796875, 0.34375, 0.345703125, 0.34765625, 0.3
0.376953125, 0.37890625, 0.380859375, 0.3828125, 0.384765625, 0.38671875, 0.388671875, 0.390625, 0.392578125, 0.39453125, 0.396484375, 0.3984375, 0.
125, 0.427734375, 0.4296875, 0.431640625, 0.43359375, 0.435546875, 0.4375, 0.439453125, 0.44140625, 0.443359375, 0.4453125, 0.447265625, 0.44921875,
5625, 0.478515625, 0.48046875, 0.482421875, 0.484375, 0.486328125, 0.48828125, 0.490234375, 0.4921875, 0.494140625, 0.49609375, 0.498046875, 0.5, 0.5



Visualize your fields

- Visualize your fields using GUI/GGUI

```
x = ti.field(ti.f32, (256, 256))

@ti.kernel
def foo():
    for i,j in x:
        x[i,j] = (i+j)/512.0

foo()

gui = ti.GUI("Debug", (256, 256))
while gui.running:
    gui.set_image(x)
    gui.show()
```



Make sure the debug mode is on when you debug

- The **debug mode** is turned **off** by default

```
ti.init(arch=ti.cpu, debug=False)

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

print("x[4] =", x[4]) # x[4] = 0.0
```

```
ti.init(arch=ti.cpu, debug=True)

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

print("x[4] =", x[4]) # RuntimeError
```

Make run-time assertions

- When the **debug mode** is on, an assertion failure triggers a RuntimeError

```
ti.init(arch=ti.cpu, debug=True)

x = ti.field(ti.f32, 128)

@ti.kernel
def do_sqrt_all():
    for i in x:
        assert x[i] >= 0
        x[i] = ti.sqrt(x[i])
```

Make run-time assertions

- Do not put any actual computations inside the assertions…

```
ti.init(arch=ti.cpu, debug=True)

x = ti.field(ti.f32, 128)
y = ti.field(ti.f32, 128)

@ti.kernel
def do_sqrt_all():
    for i in x:
        assert (y[i] = y[i] - x[i]) >= 0
        y[i] = ti.sqrt(y[i])
```

Compile-time *ti.static_assert(cond, msg=None)*

- No run-time cost
- Useful for assertions on data types / dimensionality / shapes.
- Works on release mode as well

```
@ti.func
def copy(dst: ti.template(), src: ti.template()):
    ti.static_assert(dst.shape == src.shape, "copy() needs src and
dst fields to be same shape")
    for I in ti.grouped(src):
        dst[I] = src[I]
    return x % 2 == 1
```

Assertion failure traceback (run-time)

```python
import taichi as ti
ti.init(arch=ti.cpu, debug=True)

@ti.func
def func3():
    assert(1 + 1 == 3)

@ti.func
def func2():
    func3()

@ti.func
def func1():
    func2()

@ti.kernel
def func0():
    func1()

func0()
```

```
0x7ffc9c80b31d: Py_RunMain in python38.dll
0x7ffc9c80aecd: Py_Main in python38.dll
0x7ff743091258: Unknown Function in python.exe
0x7ffd1af97034: BaseThreadInitThunk in KERNEL32.DLL
0x7ffd1cbc2651: RtlUserThreadStart in ntdll.dll

Internal error occurred. Check out this page for possible solutions:
https://docs.taichi.graphics/lang/articles/misc/install
Traceback (most recent call last):
  File "c:\Users\Tiantian\OneDrive\Test Scripts\taichi\diffuse\test.py", line 20, in <module>
    func0()
```

Assertion failure traceback (compile-time)

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

@ti.func
def func3():
    ti.static_assert(1 + 1 == 3)

@ti.func
def func2():
    func3()

@ti.func
def func1():
    func2()

@ti.kernel
def func0():
    func1()

func0()
```

Assertion failure traceback (Pretty mode ☺) *excepthook=True*

```python
import taichi as ti
ti.init(arch=ti.cpu,
excepthook=True)

@ti.func
def func3():
    ti.static_assert(1 + 1 == 3)

@ti.func
def func2():
    func3()

@ti.func
def func1():
    func2()

@ti.kernel
def func0():
    func1()

func0()
```

```
----------------------------------------
In func0() at c:\Users\Tiantian\OneDrive\Test Scripts\taichi\diffuse\test.py:18:
----------------------------------------
    func2()

@ti.kernel
def func0():
    func1()  <--

func0()
----------------------------------------
In func1() at c:\Users\Tiantian\OneDrive\Test Scripts\taichi\diffuse\test.py:14:
----------------------------------------
    func3()

@ti.func
def func1():
    func2()  <--

@ti.kernel
----------------------------------------
In func2() at c:\Users\Tiantian\OneDrive\Test Scripts\taichi\diffuse\test.py:10:
----------------------------------------
    ti.static_assert(1 + 1 == 3)

@ti.func
def func2():
    func3()  <--

@ti.func
----------------------------------------
In func3() at c:\Users\Tiantian\OneDrive\Test Scripts\taichi\diffuse\test.py:6:
----------------------------------------
ti.init(arch=ti.cpu, excepthook=True)

@ti.func
def func3():
    ti.static_assert(1 + 1 == 3)  <--

@ti.func
----------------------------------------
AssertionError
```

Turn off optimizations from Taichi

- Turn off parallelization

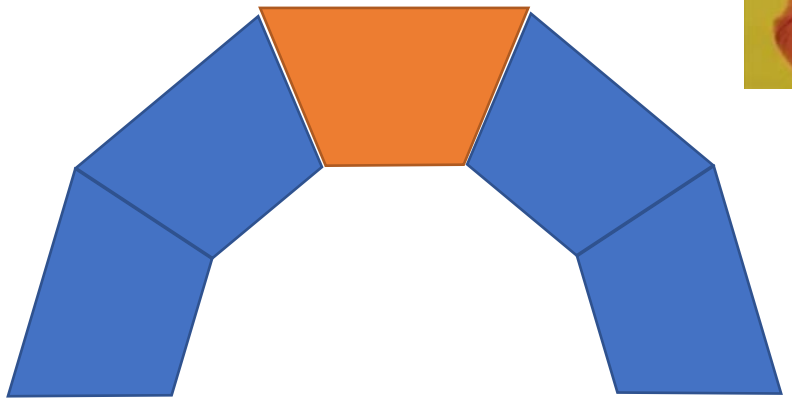
```
import taichi as ti
ti.init(arch=ti.cpu, cpu_max_num_threads=1)
```

- Turn off advanced optimization

```
import taichi as ti
ti.init(advanced_optimization=False)
```

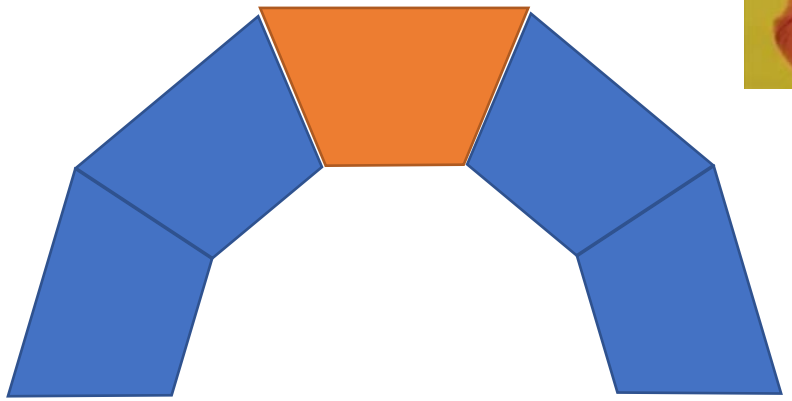
Keep your code executable

Write 1000 lines of code without any bug.
Execute your code the first time right after the final type.

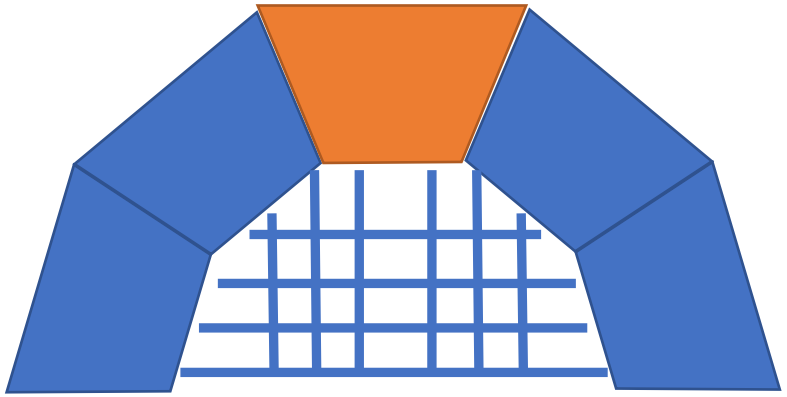


Keep your code executable

Write 1000 lines of code without any bug.
Execute your code the first time right after the final type.



Check your code every time when you finish a building block.
Keep the entire codebase executable.



Mostly seen problems

Mostly seen problems

- Data race (example 1)

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

@ti.kernel
def foo():
    for i in range(1,15):
        x[i-1] = x[i-1] + 2
        x[i+1] = x[i+1] + 4
```

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

@ti.kernel
def foo():
    for i in range(1,15):
        x[i-1] += 2
        x[i+1] += 4
```

Mostly seen problems

- Data race (example 2)

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

@ti.kernel
def foo():
    for i in range(1,16):
        x[i] += x[i-1]
```

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

@ti.kernel
def foo():
    for i in range(1,16):
        y[i] += x[i-1]

y.copy_from(x)
foo()
x.copy_from(y)
```

Mostly seen problems

- Calling Python functions from the Taichi scope triggers an undefined behavior
  - including functions from other imported Python packages

```
def foo():
    ...
    # do something

@ti.kernel
def bar():
    foo()
```

Mostly seen problems

- Stay alert when copying two variables in the Python scope

```
a = ti.field(ti.f32, shape=())
b = ti.field(ti.f32, shape=())

a[None] = 0
b[None] = 1
print(a[None]) # 0.0
b = a
b[None] = 128
print(a[None]) # 128.0
```

```
a = ti.field(ti.f32, shape=())
b = ti.field(ti.f32, shape=())

a[None] = 0
b[None] = 1
print(a[None]) # 0.0
b.copy_from(a)
b[None] = 128
print(a[None]) # 0.0
```

Mostly seen problems

- The data types in Taichi are static

```
x = ti.field(ti.f32, shape=128)

@ti.kernel
def reduction():
    sum = 0
    for i in x:
        sum += x[i]

    print(sum) # integer
```

```
[Taichi] version 0.8.3, llvm 10.0.0, commit 021af5d2, win, python 3.8.10
[Taichi] Starting on arch=x64
[W 10/19/21 11:01:47.465 2260] [type_check.cpp:taichi::lang::TypeCheck::visit@70] [$15]
Atomic add (f32 to i32) may lose precision, at
[W 10/19/21 11:01:47.467 2260] [type_check.cpp:taichi::lang::TypeCheck::visit@71]

0
```

Mostly seen problems

- The data in Taichi have a static lexical scope

```
@ti.kernel
def foo():
    i = 20

    for i in range(10): # Error
        ...

    print(i)

foo()
```

```
def foo():
    i = 20

    for i in range(10):
        ...

    print(i) # 9

foo()
```

Mostly seen problems

- Multiple returns in a single @ti.func are not supported

```
@ti.func
def abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

```
@ti.func
def abs(x):
    res = x
    if x < 0:
        res = -x
    return res
```

Mostly seen problems

- Data access using slices is not supported

```
M = ti.Matrix(4,4)
...
M_sub = M[1:2, 1:2]
```

```
M = ti.Matrix(4,4)
...
M_sub = M[(1,2), (1,2)]
```

Debugging

- Tips
  - Visualize your intermediate results (print/GUI)
  - Make assertions
  - Keep your code executable
  - Turn off all the performance-oriented optimizations
  - Make sure the debug mode is on
  - ----------
  - Be aware of race conditions (in the parallel for loops)
  - Be aware of the static data types
  - Be aware of the static lexical scope

Code Optimization

Code optimization

- Performance profiling
- Performance tuning tips

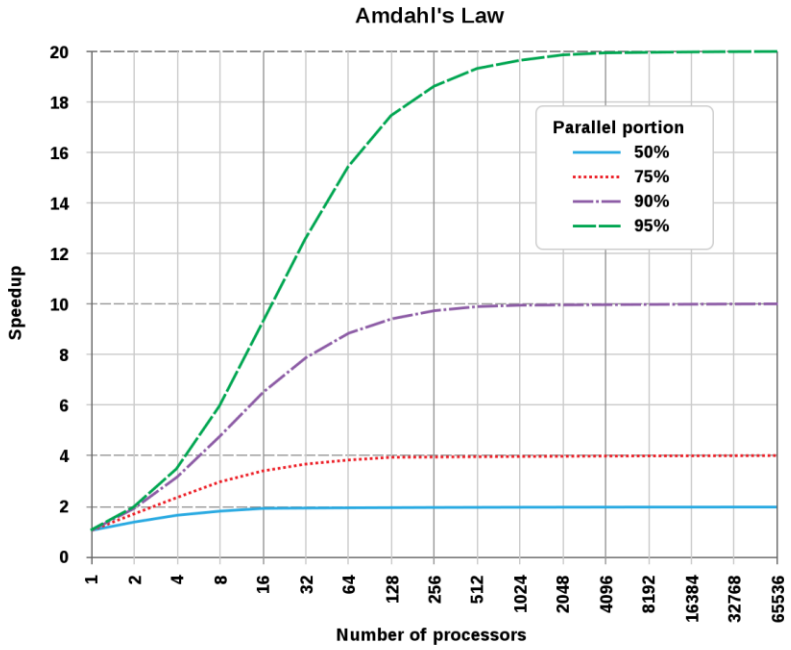
Why “Profiling”?

- If you optimize one method in your program to 10x faster:
  - If that method costs 90% of your run time: → you are 5.26x faster
  - If that method costs 10% of your run time: → you are 1.10x faster

Why “Profiling”?

- If you optimize one method in your program to 10x faster:
  - If that method costs 90% of your run time: → you are 5.26x faster
  - If that method costs 10% of your run time: → you are 1.10x faster

- Amdahl's law



A simple profiler

- Enable Taichi's kernel profiler:

```
ti.init(kernel_profiler=True, arch=ti.gpu)
```

- Output your profiling info:

```
ti.print_kernel_profile_info('count')
```

- Clear profiling info:

```
ti.clear_kernel_profile_info()
```

A simple profiler

- Your HW2: --Galaxy

```python
import taichi as ti
from celestial_objects import Star, Planet

if __name__ == "__main__":
    ti.init(kernel_profiler=True, arch=ti.cuda)

    ...

    stars.computeForce()
    planets.computeForce(stars)
    for celestial_obj in (stars, planets):
        celestial_obj.update(h)
    ti.clear_kernel_profile_info()

    for i in range(17): # while my_gui.running:

        ...

        if not paused:
            stars.computeForce()
            planets.computeForce(stars)
            for celestial_obj in (stars, planets):
                celestial_obj.update(h)
            i += 1

        stars.display(my_gui, radius=10, color=0xffd500)
        planets.display(my_gui)
        if export_images:
            my_gui.show(f"images\output_{i:05}.png")
        else:
            my_gui.show()

    ti.print_kernel_profile_info('count')
```

```python
    @ti.kernel
    def computeForce(self):
        self.clearForce()
        for i in range(self.n):
            p = self.pos[i]
            for j in range(self.n):
                if j != i:
                    diff = self.pos[j] - p
                    r = diff.norm(1e-2)
                    self.force[i] += self.G * self.m
* self.m * diff / r**3
```

Read your profiling results:

```
=========================================================================
Kernel Profiler(count) @ CUDA
=========================================================================
[ % total count | min avg max ] Kernel name
-------------------------------------------------------------------------
[ 82.20% 0.002 s 17x | 0.090 0.105 0.113 ms] computeForce_c38_0_kernel_8_range_for
[ 3.90% 0.000 s 17x | 0.002 0.005 0.013 ms] matrix_to_ext_arr_c14_1_kernel_12_range_for
[ 3.17% 0.000 s 17x | 0.003 0.004 0.004 ms] computeForce_c34_0_kernel_5_range_for
[ 2.17% 0.000 s 17x | 0.002 0.003 0.004 ms] matrix_to_ext_arr_c14_0_kernel_11_range_for
[ 2.16% 0.000 s 17x | 0.002 0.003 0.004 ms] update_c36_0_kernel_9_range_for
[ 2.09% 0.000 s 17x | 0.002 0.003 0.004 ms] computeForce_c34_0_kernel_6_range_for
[ 2.05% 0.000 s 17x | 0.002 0.003 0.003 ms] update_c36_1_kernel_10_range_for
[ 1.98% 0.000 s 17x | 0.002 0.003 0.003 ms] computeForce_c38_0_kernel_7_range_for
[ 0.28% 0.000 s 2x | 0.002 0.003 0.004 ms] jit_evaluator_2_kernel_4_serial
-------------------------------------------------------------------------
[100.00%] Total execution time: 0.002 s number of results: 9
=========================================================================
```

Performance profiling → performance tuning

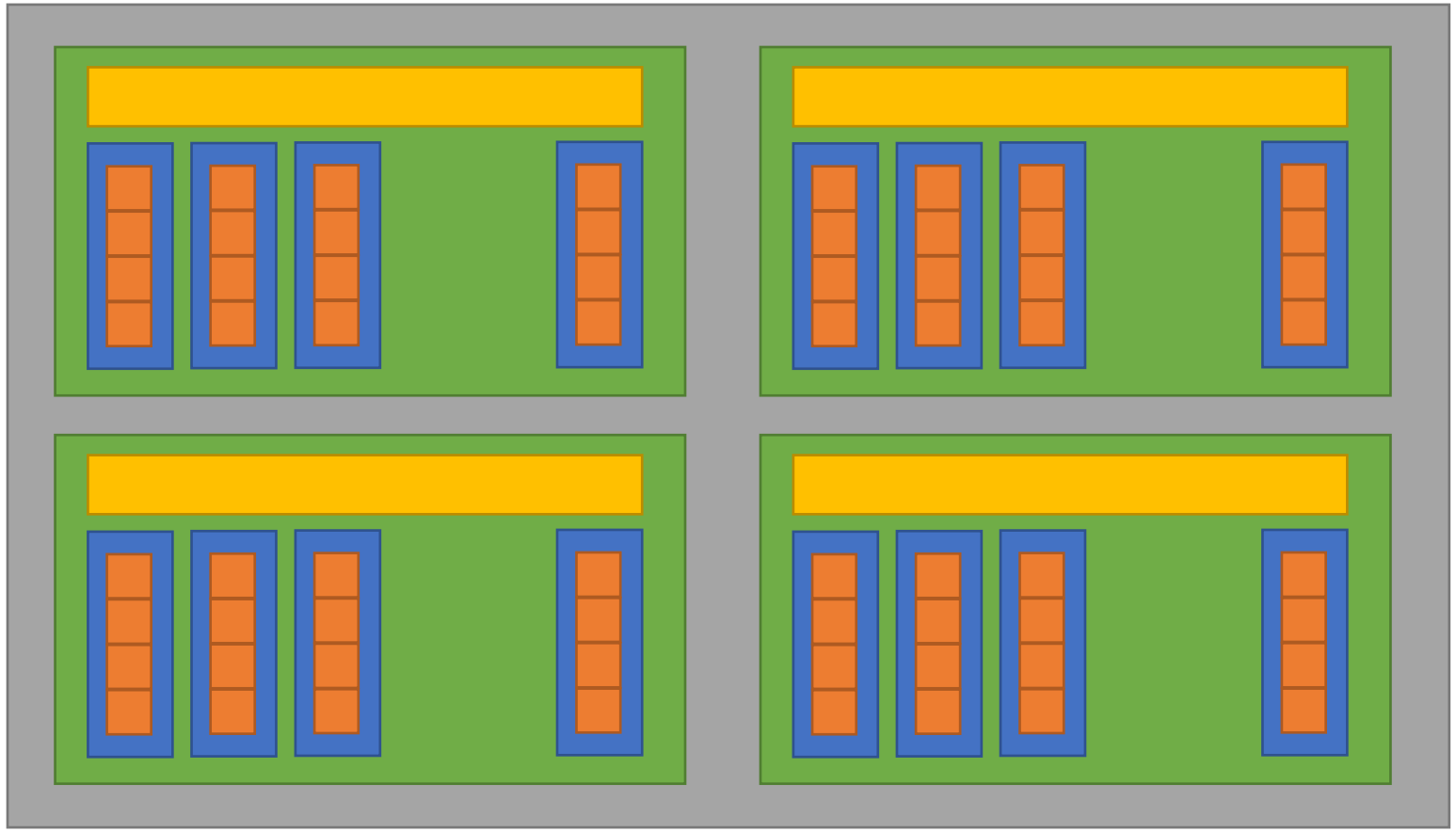
- Find the slowest component in our program

↓

- Optimize that slowest component

Performance tuning

- Background: Thread hierarchy of Taichi in GPU



Iteration: each iteration in a for-loop

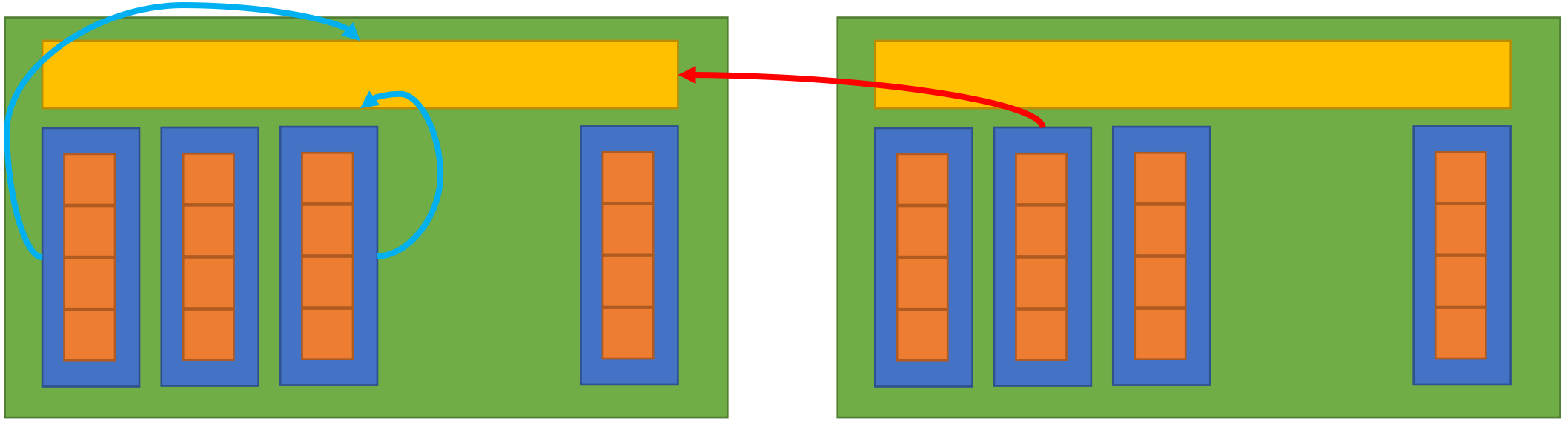
Thread: the minimal parallelizable unit

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

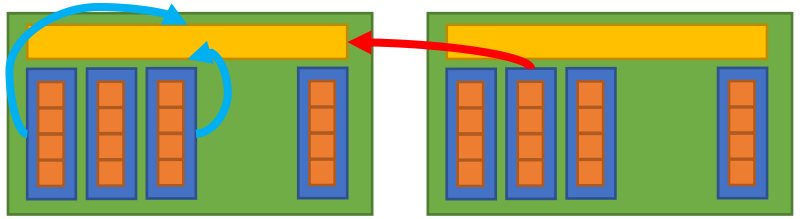
Grid: the minimal unit that being launched from the host

The block local storage (BLS)

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

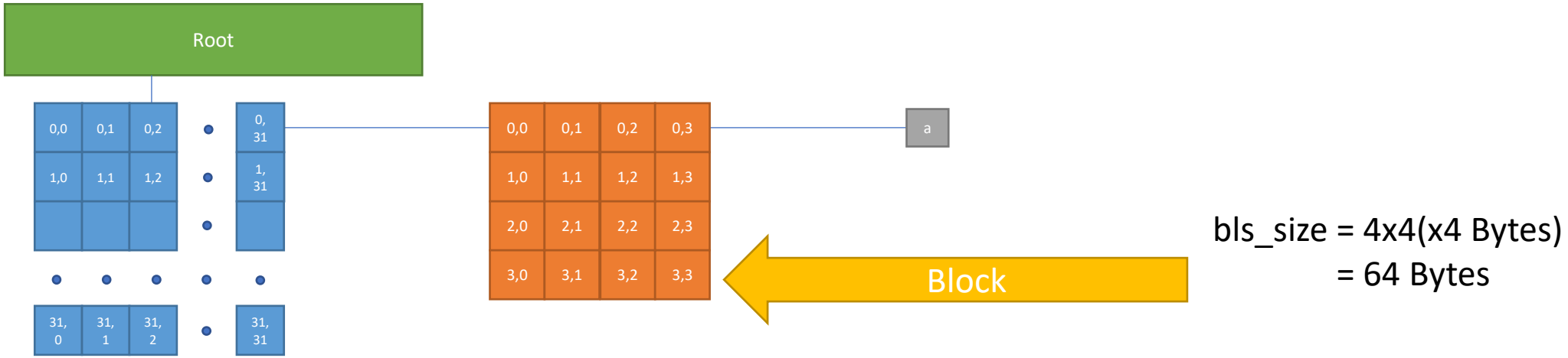


The block local storage (BLS)



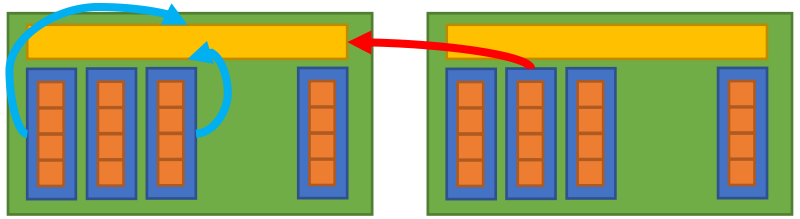
- Block size of an hierarchically defined field (SNode-tree):

```
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

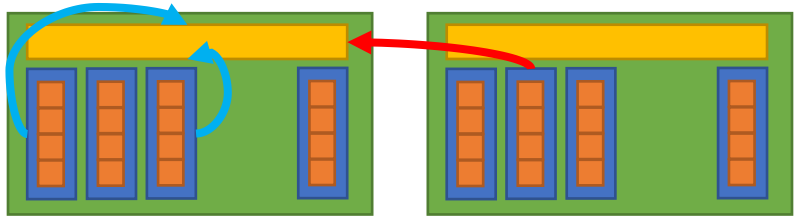
The block local storage (BLS)



- 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=256
        ...

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

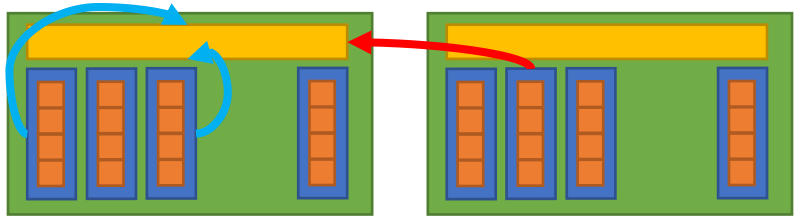
The block local storage (BLS)



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

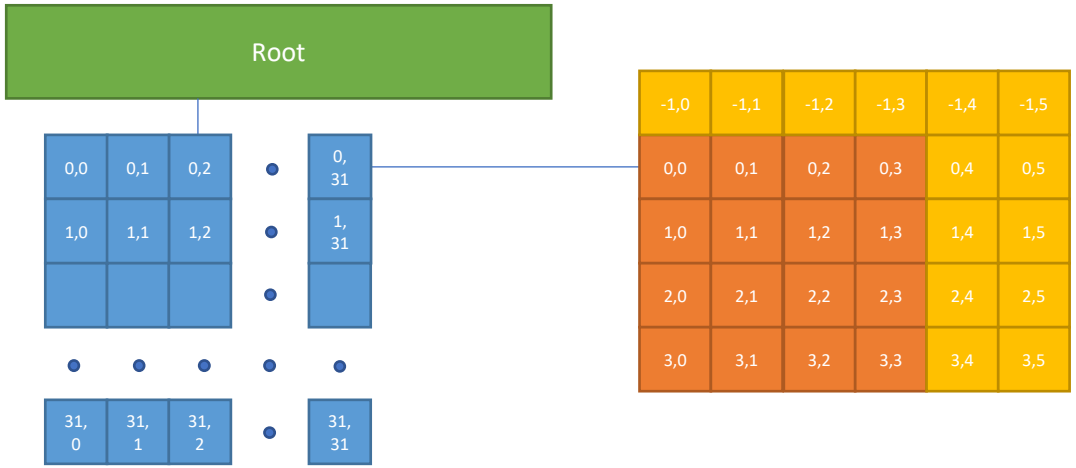
```
a = ti.field(ti.f32)
# `a` has a block size of 4x4
ti.root.pointer(ti.ij, 32).dense(ti.ij, 4).place(a)

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

The block local storage (BLS)



- Cache the most frequently-used data into BLS 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

Code optimization (for performance)

- Docs
  - Taichi Profiler: [link]
  - Performance tuning tips: [link]

- Always profile your code first
  - Find the slowest component to optimize

- Performance tuning
  - Take full advantage of the block local storage (shared memory)
  - Try different data layouts
  - Try different *ti.block_dim()* for your parallel for loops
  - Cache the most important field manually using *ti.block_local()*

Remark

- Sparse matrix and sparse linear algebra
  - [SparseMatrixBuilder] = ti.SparseMatrixBuilder()
  - [SparseMatrix] = [SparseMatrixBuilder].build()
  - [NumpyArray] = [SparseMatrix].solve([Field])
- Debugging a Taichi project
  - Visualize results, make assertions, disable optimizations, use debug mode
- Optimizing your Taichi code
  - Profile your project
  - Optimize your performance

Remark

- Sparse matrix and sparse linear algebra
  - [SparseMatrixBuilder] = ti.SparseMatrixBuilder()
  - [SparseMatrix] = [SparseMatrixBuilder].build()
  - [NumpyArray] = [SparseMatrix].solve([Field])
- Debugging a Taichi project
  - Visualize results, make assertions, disable optimizations, use debug mode
- Optimizing your Taichi code
  - Profile your project
  - Optimize your performance

Remark

- Sparse matrix and sparse linear algebra
  - [SparseMatrixBuilder] = ti.SparseMatrixBuilder()
  - [SparseMatrix] = [SparseMatrixBuilder].build()
  - [NumpyArray] = [SparseMatrix].solve([Field])
- Debugging a Taichi project
  - Visualize results, make assertions, disable optimizations, use debug mode
- Optimizing your Taichi code
  - Profile your project
  - Optimize your performance

Homework

Check our examples

- Sparse matrix:
  - Diffuse: https://github.com/taichiCourse01/--Diffuse
  - Implicit mass-spring: https://github.com/taichi-dev/taichi/blob/master/examples/simulation/implicit_mass_spring.py
  - Stable fluid: https://github.com/taichi-dev/taichi/blob/master/examples/simulation/stable_fluid.py

- Performance profiling:
  - Galaxy: https://github.com/taichiCourse01/--Galaxy
  - Diffuse: https://github.com/taichiCourse01/--Diffuse

Share your homework

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

Note

- Last class in Taichi!
  - Using docs.taichi.graphics as your manual
  - Visit api-docs.taichi.graphics for APIs (page WIP)

- Will start the computer graphics topics from the next week
  - Procedural animation (1 class)
  - Render (2 classes)
  - Deformable body simulation (2 classes)
  - Fluid simulation (2 classes)

Questions?

本次答疑：10/21
下次直播：10/26
直播回放：Bilibili 搜索「太极图形」
主页&课件：https://github.com/taichiCourse01