Taichi Lang

Cheat sheet
Version (1,3,0)

Installation

```
pip install -U taichi   # Install / upgrade
ti gallery              # Run demo gallery
ti example              # More examples
```

Quick start DOC

```
import taichi as ti
ti.init(arch=ti.cpu)
# Or use another backend
# [ti.cuda, ti.vulkan, ti.opengl, ti.metal]

@ti.kernel
# Args and return of a kernel must be type hinted
def monte_carlo_pi(n: int) -> float:
    total = 0
    for i in range(n): # A parallel for loop
        x = ti.random()
        y = ti.random()
        if x*x + y*y < 1:
            total += 1

    return 4 * total / n

print(monte_carlo_pi(100000))
```

Kernels and functions DOC

`@ti.kernel`: Entrance for Taichi's JIT to take control. Must be called from Python scope. Require type hints for the arguments and the return value. Can return at most one scalar or vector or matrix. Top-level for loops are automatically parallelized.

`@ti.func`: Must be called by kernels or other Taichi functions. Recommend type hints for arguments and return values. Can return multiple values of scalars, vectors, matrices, and structs.

Top-level for loops are automatically parallelized

```
x = ti.field(dtype=int, shape=100)
@ti.kernel
def loop_field():
    for i in range(100):   # A parallelized loop
        for j in range(10):   # Not parallelized
            ...
    for i in x:   # Also a parallelized loop
        x[i] = i
```

Serialize a top-level for loop

```
@ti.kernel
def config_loop():
    # Serialize the *next* for loop
    ti.loop_config(serialize=True)
    for i in range(100):   # Serial
        ...
    for i in range(100):   # Parallel
        ...
```

Interact with NumPy/PyTorch DOC

```
x = numpy/torch.arange(12).reshape(3, 4)
@ti.kernel
def fill_array(arr: ti.types.ndarray()):
    print(arr.shape, arr.dtype)
    for i, j in arr:
        arr[i, j] = i + j
        print(arr[i, j])

fill_array(x)   # zero copy if on the same device
```

Data types DOC

Primitive types (int, unsigned int and float in C)

```
i8  # char      u8  # uchar      f16 # half
i16 # short     u16 # ushort     f32 # float
i32 # int       u32 # uint       f64 # double
i64 # long      u64 # ulong
```

Vector and matrix types

- Vector: `ti.types.vector(dim, dtype)`

```
vec3 = ti.types.vector(3, float)
v = vec3(1, 2, 3)
```

- Matrix: `ti.types.matrix(n, m, dtype)`

```
mat2 = ti.types.matrix(2, 2, float)
m = mat2(1, 2, 3, 4)
```

Struct types: `ti.types.struct(**kwargs)`

```
sphere = ti.types.struct(
    center=vec3, radius=float
)
s = sphere(vec3(1, 2, 3), 1.0)
```

Ndarray types:

```
ti.types.ndarray(
    dtype, # Data type of the array,
    ndim    # Number of field dimentions
)
```

```
img2d_type = ti.types.ndarray(ndim=3)
noise_img = np.random.random((400, 400, 3))

@ti.kernel
def process(img: img2d_type):
    for I in ti.grouped(img):
        r, g, b = img[I]
        ...

process(noise_img)
```

Typecasting

```
x = ti.f32(1)
y = ti.u8(x)   # Equivalent to x.cast(ti.u8)
z = float(y)

u = vec3(0.5, 1.0, 1.5)
v = u.cast(int)   # Cast each entry to int
```

Performance tuning DOC

Kernel profiler (CPU and CUDA only)

Analyze the performance of Taichi kernels

```
ti.init(arch=ti.cpu, kernel_profiler=True)
@ti.kernel
def some_kernel():
    ...

some_kernel()
ti.profiler.print_kernel_profiler_info()
```

Configure loops

Set arguments in `ti.loop_config()` to control the next for loop:

1. Set the number of threads in a block on GPU: `block_dim=8`.
2. Set the number of threads to use on CPU: `parallelize=8`.
3. Whether to let the for loop execute serially: `serialize=True`.

Data containers DOC

Scalar field: `field(dtype, …)`

```
f = ti.field(int, shape=(3, 3, 3))
f[0, 1, 2] = 3  # Indexing with three integers
x = ti.field(int, shape=()) # 0-D field
x[None] = 1.0 # Use None to index 0-D field
```

Vector field: `Vector.field(dim, dtype, …)`

```
f = ti.Vector.field(3, float, (10, 10))
f[0, 1] = 1, 2, 3  # Each entry is a 3D vector
```

Matrix field: `Matrix.field(n, m, dtype, …)`

```
f = ti.Matrix.field(2, 2, float, shape=(10, 10))
f[0, 1] = mat2(1) # Each entry is a 2x2 matrix
```

Struct field: `obj.field(shape)`

```
sphere = ti.types.struct(center=vec3, radius=
    float)
f = sphere.field(shape=100)
```

Fill a field with a scalar

```
f.fill(1)
```

Copy data from/to NumPy array/PyTorch tensor

```
f.from_numpy(arr)
arr = f.to_numpy()
f.from_torch(tensor)
tensor = f.to_torch()
```

Loop over a field in parallel

```
f = ti.field(int, shape=(100, 100, 100))
@ti.kernel
def loop_field():
    for i, j, k in ti.ndrange(100, 100, 100):
        f[i, j, k] = i + j + k
    # Equivalent to the above
    for i, j, k in f:
        f[i, j,k] = i + j + k
    # Equivalent to the above
    for I in ti.grouped(f):
        # I = [i, j, k] is a 3D int vector
        f[I] = I[0] + I[1] + I[2]
```

Switch data layout between AOS and SOA DOC

```
u = ti.Vector.field(
    3, float,shape=100, layout=ti.Layout.AOS)
# array of structs [x0,y0,z0,x1,y1,z1…,]
v = ti.Vector.field(
    3, float,shape=100, layout=ti.Layout.SOA)
# struct of arrays [x0,x1…,,y0,y1…,,z0,z1…,]
```

Math functions DOC

```
import taichi as ti

ti.cos(x)
ti.sin(x)
ti.acos(x)
ti.asin(x)
ti.atan2(y, x)
ti.exp(x)
ti.log(x)
ti.ceil(x, dtype)
ti.floor(x, dtype)
ti.round(x, dtype)
ti.pow(x, a)
ti.tan(x)
ti.tanh(x)
ti.sqrt(x)
ti.max(x,y…,)
ti.min(x,y…,)
ti.random(dtype)
```

```
import taichi.math as tm

tm.cross(u,v)
tm.dot(u,v)
tm.fract(x)
tm.mod(x,y)
tm.normalize(v)
tm.smoothstep(e0,e1,x)
tm.mix(x,y,a)
tm.step(edge,x)
tm.degrees(x)
tm.radians(x)
tm.clamp(x,xmin,xmax)
tm.length(v)
tm.log2(x)
tm.inverse(mat)
tm.isnan(x)
tm.isinf(x)
tm.sign(x)
```

Operators DOC

Arithmetic operators

```
-x, x + y, x - y, x * y,
x / y, # returns a floating point. 5 / 2 = 2.5
x // y # floor of x / y. 5.0 / 2.0 = 2.0
x % y  # remainder of x / y. x & y can be floats
x ** y # x to the power of y
A @ B  # matrix multiplication
```

Comparison operators

```
x == y, x != y, x > y, x < y, x >= y, x <= y
```

Logical operators

```
not, or, and
```

Bitwise operators

```
-x, x & y, x ^ y, x | y, x << y, x >> y
```

Data-oriented programming DOC

Data-oriented class

When you have data maintained in the Python scope (such as time or user input events) and you want the kernels to track their changes, you can organize them into a data-oriented class.

```
@ti.data_oriented
class TiArray:
    def __init__(self, n):
        self.x = ti.field(dtype=ti.i32, shape=n)

    @ti.kernel
    def inc(self):
        for i in self.x:
            self.x[i] += 1

a = TiArray(32)
a.inc()
```

dataclass

A dataclass is a wrapper of `ti.types.struct`. You can define Taichi functions as its methods and call these methods in the Taichi scope.

```
@ti.dataclass
class Sphere:
    center: vec3
    radius: float
    @ti.func
    def area(self): # A Taichi function as method
        return 4 * pi * self.radius**2

@ti.kernel
def test():
    sphere = Sphere(vec3(0), radius=1.0)
    print(sphere.area())
```

Global settings DOC

You can config Taichi by passing arguments to the `ti.init()` call:

1. Choose a backend: `arch=ti.cuda`.
2. Enable debug mode: `debug=True`.
3. Enable dynamic index: `dynamic_index=True`.
4. Set floating precision: `default_fp=ti.f64`.
5. Set integer precision: `default_ip=ti.i64`.
6. Set random seed: `random_seed=0`.
7. Disable offline cache: `offline_cache=False`.
8. Enable packed mode: `packed=True`.
9. Set logging level: `log_level=ti.ERROR`.
10. Set pre-allocated memory size for CUDA: `device_memory_GB=1.3`.

Visualization

GUI system DOC

```
pixels = ti.Vector.field(3, float, (640, 480))
gui = ti.GUI('Window Title', res=(640, 480))
while gui.running:
    gui.set_image(pixels)
    gui.show()
```

GGUI system DOC

```
pixels = ti.Vector.field(3, float, (640, 480))
window = ti.ui.Window('Window Title', (640, 480))
canvas = window.get_canvas()
while window.running:
    canvas.set_image(pixels)
    window.show()

window.save_image(filename) # save image file
```

2D canvas drawing API

```
canvas.set_background_color(color)
canvas.triangles(vertices, color, indices,
    per_vertex_color)
canvas.circles(vertices, radius, color,
    per_vertex_color)
canvas.lines(vertices, width, indices, color,
    per_vertex_color)
```

3D scene drawing API

```
scene.lines(vertices, width, indices, color,
    per_vertex_color)
scene.mesh(vertices, indices, normals, color,
    per_vertex_color)
scene.particles(vertices, radius, color,
    per_vertex_color)
```

Debugging `ti.init(..., debug=True)` DOC

Debug mode can help check access out-of-bound errors and allow you to `assert` in Taichi kernels on CPU and CUDA backends.

```
ti.init(arch=ti.cpu, debug=True)
f = ti.field(int, shape=(5, 5))
print(f[7, 7])  # Only raise error in debug mode!
```

Runtime assert in Taichi kernels and functions

```
x = ti.field(dtype=ti.f32, shape=128)
@ti.kernel
def foo():
    for i in x:
        assert x[i] >= 0
        x[i] = ti.sqrt(x[i])
```

Runtime print in Taichi kernels and functions

```
@ti.kernel
def inside_taichi_scope(x: float):
    # print is supported on cpu, cuda
    # and vulkan backends
    print('hello', x)  # Cannot use f-string
```

Compile-time static-print

```
x = ti.field(ti.f32, (2, 3))
@ti.kernel
def print_field_attributes():
    ti.static_print(x.shape, x.dtype)
```

Compile-time static-assert

```
@ti.func # Assuming dst and src are fields
def copy(dst: ti.template(), src: ti.template()):
    ti.static_assert(dst.shape == src.shape)
    for I in ti.grouped(src):
        dst[I] = src[I]
```