Taichi-Lang

Cheat sheet Version (1, 3, 0)

Installation

```
pip install taichi
ti example
```

Quick start API

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

@ti.kernel
def monte_carlo_pi() -> float:
    total = 0
    for i in range(100000):
        x = ti.random()
        y = ti.random()
        if (x*x + y*y) < 1:
            total += 1

    return 4 * total / n
```

Supported backends

ti.cpu, ti.cuda, ti.metal, ti.opengl, ti.vulkan

Data types API

primitive tyes

```
i8, i16, i32, i64, u8, u16, u32, u64, f16, f32, f64
```

types.vector(n, dtype)

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

types.matrix(n, m, dtype)

```
mat2x2 = ti.types.matrix(2, 2, float)
m = mat2x2(1, 2, 3, 4)
```

types.struct(**kwargs)

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

Qaunt types API

```
u5 = ti.types.quant.int(bits=5, signed=False)
fixed_a = ti.types.quant.fixed(bits=10, max_value=20.0)
float_b = ti.types.quant.float(exp=6, frac=9, signed=False)
```

Data container API

field(dtype, shape, ...)

```
f = ti.field(int, shape=(3, 3, 3))
```

Vector.field(dim, dtype, shape, ...)

```
f = ti.Vector.field(3, float, (10, 10))
```

Matrix.field(dim, dtype, shape, ...)

```
f = ti.Matrix.field(3, 3, float, shape=(10, 10))
```

Struct.field(dict, shape)

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

Frequently used features
fill(val)

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

from_numpy(arr)
to_numpy()

Kernels and functions

`@ti.kernel` : Called from Python scope. Require type hints for arguments and return values. Can return scalar, vector and matrix.
`@ti.func` : Called from Taichi scope. Recommend type hints for arguments and return values. Can return scalar, vector, matrix and struct.

Interactive with Numpy arrays API

```
arr = numpy.arange(12).reshape(3, 4)

@ti.kernel
def example(x: ti.types.ndarray()):
    for i, j in ti.ndrange(arr.shape[0], arr.shape[1]):
        arr[i, j] = i + j
```

Data-oriented programming API

data-oriented class
A data-oriented class is used when your data is actively updated in the Python scope (such as current time and user input events) and tracked in Taichi kernels.

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

    @ti.kernel # Defines Taichi kernels in the data-oriented
           Python class
    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): # Defines a Taichi function as method
        return 4 * math.pi * self.radius**2

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

Commonly-used functions API

ti.acos(x), ti.asin(x), ti.atan2(x), ti.ceil(x), ti.clamp(x, xmin, xmax), ti.cos(x), ti.cross(x, y), ti.dot(x,y), ti.exp(x), ti.floor(x),ti.fract(x), ti.inverse(mat), ti.norm(x), ti.log(x), ti.max(x, y, ...), ti.min(x, y, ...), tm.mod(x,y), tm.normalize(x), tm.pow(x, a), ti.round(x), ti.sign(x), ti.sin(x), tm.smoothstep(e0, e1, x), ti.sqrt(x), tm.step(edge, x),ti.tan(x), ti.tanh(x), tm.degrees(x), tm.radians(x)

Operators API

comparison operators
==, !=, >, <, >=, <=

logical operators
not, or, and

bitwise operators
~, &, ^, |, «, »

Visualization

GUI system API

```
gui = ti.GUI('Window Title', (640, 360)) # Creates a window
while not gui.get_event(ti.GUI.ESCAPE, ti.GUI.EXIT):
    gui.show() # Displays the window
```

GGUI system API

```
pixels = ti.Vector.field(3, float, (640, 480))
window = ti.ui.Window("Window Title", (640, 360))# Creates a
       window
canvas = window.get_canvas() # Creates a canvas

while window.running:
    canvas.set_image(pixels)
    window.show()
```

Performance tuning API

Kernel profiler (CPU and CUDA only):
To analyze the performance of Taichi kernels

```
ti.init(ti.cpu, kernel_profiler=True)
ti.profiler.print_kernel_profiler_info()
```

Configure loops:
To serialize the outermost for loop that immediately follows the line

```
ti.loop_config(serialize=True)
```

To designate No. of threads on the CPU backend

```
ti.loop_config(parallelize=8)
```

To designate No. of threads in each block of the GPU backend

```
ti.loop_config(block_dim=16)
```

Debugging API

Activate debug mode:

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

Runtime print in Taichi scope:

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

Runtime assert in Taichi scope:

```
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])
```

Compile-time static-print:

```
x = ti.field(ti.f32, (2, 3))
y = 1

@ti.kernel
def inside_taichi_scope():
    ti.static_print(y) # => 1
    ti.static_print(x.shape) # => (2, 3)
    ti.static_print(x.dtype) # => DataType.float32
```

Compile-time static-assert:

```
@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]
```

Serial execution:
To serialize the program

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

To serializes the for loop that immediately follows the line

```
ti.loop_config(serialize=True)
```

Access a conciser version of traceback message:

```
import sys
sys.tracebacklimit = 0
```