Taichi Basics Cheat Sheet

by olina via cheatography.com/165198/cs/34724/

Taichi in a Nutshell

A domain specific language embedded in Python for high-performance parallel computing
Just-in-time (JIT) compilation
Automatically parallelizes outermost for loops in a kernel
Supports multiple backends (CPUs, CUDA, OpenGL, Metal...)
Supports ahead-of-time compilation

Hello, World!

1. Install Taichi:

```
pip install -U taichi
```

2. Verify installation - Taichi gallery:

```
ti gallery
```

3. Write your first Taichi program:

```
import taichi as ti
ti.init(arch=ti.cpu)
# A backend can be either ti.cpu or ti.gpu
# When ti.gpu is specified, Taichi moves down the backend list and tries CUDA, then Vulkan, and then OpenGL
l
```

Data types

Primitive data types: `i8, i16, i32, i64, u8, u16, u32, u64, f16, f32, f64`

i: integer; u: unsigned integer; f: floating-point number
Number following i/u/f stands for precision bits

Change default types:

```
# Default integer type: ti.i32; default floati ng- point type: ti.f32
 ti.ini t(d efa ult _ip =ti.i64)  # Sets the default integer type to ti.i64
 ti.ini t(d efa ult _fp =ti.f64)  # Sets the default floati ng- point type to ti.
f64
```

Explicit type casting:

```
# Use ti.cast():
 a = 3.14
b = ti.cast(a, ti.i32)  # 3
c = ti.cast(b, ti.f32)  # 3.0
 # Use primitive types to convert a scalar variable to a different scalar type:
 a = 3.14
x = int(a)    # 3
y = float(a)  # 3.14
z = ti.i32(a) # 3
w = ti.f64(a) # 3.14
```

Implicit type casting:

Integer + floating point -> floating point
Low-precision bits + high-precision bits -> high-precision bits
Signed integer + unsigned integer -> unsigned integer

Compound data types:

Vectors and matrices:

Data types (cont)

```
vec4d = ti.typ es.v ec tor(4, ti.f64) # A 64-bit floa
ype
mat4x3i = ti.typ es.m at rix(4, 3, int) # A 4x3 integ
v = vec4d(1, 2, 3, 4) # Creates a vector instance: v
```

Structs:

```
# Defines a compound type vec3 to represent a sphere
vec3 = ti.typ es.v ec tor(3, float)
# Defines a compound type sphere _type to represent a
sphere_type = ti.typ es.s tr uct (ce nte r=vec3, radi
sphere = sphere _ty pe( cen ter =ve c3(0), radius =1
```

Quantized/low-precision data types:

```
# Defines a 5-bit unsigned integer
u5 = ti.typ es.q ua nt.i nt (bi ts=5, signed =False)
# Defines a 10-bit signed fixed point type within the
fixed_type_a = ti.typ es.q ua nt.f ix ed( bit s=10, r
# Defines a 15-bit unsigned floati ng- point type wit
float_type_b = ti.typ es.q ua nt.f lo at( exp=6, frac
```

Sparse matrix (pending)

Data container

Field (global data container):

Declare:

```
# Declares a scalar field
scalar_field = ti.fie ld(int, shape= (640, 480))
# Declares a vector field
vector_field = ti.Vec tor.fi eld (n=2, dtype= float,
# Declares a matrix field
matrix_field = ti.Mat rix.fi eld (n=3, m=2, dtype= fl
```

Index:

```
f_0d = ti.fie ld( float, shape=())
f_0d[None] = 1.0 # Accesses the element in a 0D fiel
f_1d = ti.fie ld(int, shape=10)
f_1d[5] = 1
f_2d = ti.fie ld(int, shape=(10, 10))
f_2d[1, 2] = 255
f_3d = ti.Vec tor.fi eld(3, float, shape=(10, 10, 10)
f_3d[3, 3, 3] = 1, 2, 3
f_3d[3, 3, 3][0] = 1
```

Interact with external arrays:

```
x = ti.fie ld( ti.f32, 4)
x_np = x.to_n umpy() # Exports data in Taichi fields
x.from_numpy(x_np) # Imports data from NumPy arrays t
x_torch = x.to_t orch() # Exports data in Taichi fiel
x.from_torch(torch.tensor([1, 7, 3, 5])) # Imports da
@ti.kernel
def numpy_ as_ nda rra y(arr: ti.nda rra y()): # Pass
     for i in ti.ndr ang e(a rr.s ha pe[0]):
         ...
```

Data container (cont)

Ndarray: A multidimensional container of elements of the same type and size

```
pos = ti.Vec tor.nd arr ay(2, ti.f32, N)
vel = ti.Vec tor.nd arr ay(2, ti.f32, N)
force = ti.Vec tor.nd arr ay(2, ti.f32, N)
```

Kernels and functions

Kernel: An entry point where Taichi's runtime begins to take over computation tasks. The outermost for loops in a kernel are automatically parallelized.

Taichi function: A building block of kernels. you can split your tasks into multiple Taichi functions to improve readability and reuse them in different kernels.

Taichi kernel vs. Taichi function

	Taichi kernel	Taichi function
Decorated with	@ti.kernel	@ti.func
Called from	Python scope	Taichi scope
Type hint arguments	Required	Recommended
Type hint return values	Required	Recommended
Return type	Scalar/`ti.Vector`/`ti.Matrix`	Scalar/`ti.Vector`/`ti.Matrix`/`ti.Struct`/...
Max. No. of elements in arguments	32 (for OpenGL) 64 (for others)	Unlimited
Max. No. of return values	1	Unlimited

Visualization

GUI system:

```
gui = ti.GUI ('W indow Title', (640, 360)) # Creates a window
 while not gui.ge t_e ven t(t i.G UI.E SCAPE, ti.GUI.EXIT):
    gui.show() # Displays the window
```

GGUI system:

```
pixels = ti.Vec tor.fi eld(3, float, (640, 480))
 window = ti.ui.W in dow ("Window Title", (640, 360)) # Creates a windo
w
 canvas = window.ge t_c anvas() # Creates a canvas

 while window.ru nning:
    canvas.set_image(pixels)
    window.show()
```

Data-oriented programming

Data-oriented class:

A data-oriented class is used when your data is actively updated in the time and user input events) and tracked in Taichi kernels.

```
@ti.da ta_ ori ented # Decorates a class with a @ti.c
tor
class TiArray:
     def __init __( self, n):
         self.x = ti.fie ld( dty pe= ti.i32, shape=n)

     @ti.kernel # Defines Taichi kernels in the data-

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


 a = TiArra y(32)
 a.inc()
```

Taichi dataclass:

A dataclass is a wrapper of `ti.typ es.s truct`. You can define Tai call these methods in the Taichi scope.

```
@ti.da taclass
class Sphere:
    center: vec3
    radius: float
    @ti.func
    def area(s elf): # Defines a Taichi function as m
        return 4  math.pi  self.r adi us**2

@ti.kernel
 def test():
    sphere = Sphere (ve c3(0), radius =1.0)
    print(sphere.area())
```

Math

Import Taichi's math module:

```
import taichi.math as tm
```

The module supports the following:

Mathematical functions:

```
# Call mathem atical functions in the Taichi scope
@ti.kernel
def test():
    a = tm.vec3(1, 2, 3) # A function can take vecto
    x = tm.sin(a) # [0.841471, 0.909297, 0.141120] #
    y = tm.flo or(a) # [1.000000, 2.000000, 3.000000
    z = tm.deg rees(a) # [57.29 5780, 114.59 1560, 1
```

Small vector and matrix types:

vec2/vec3/vec4: 2D/3D/4D floating-point vector types
ivec2/ivec3/ivec4: 2D/3D/4D integer vector types
uvec2/uvec3/uvec4: 2D/3D/4D unsigned integer vector types
mat2/mat3/mat4: 2D/3D/4D floating-point square matrix types

GLSL-standard functions:

```
@ti.kernel
def example():
    # Takes vectors and matrices as arguments and ope

    v = tm.vec3(0, 1, 2)
    w = tm.smo oth step(0, 1, v)
        w = tm.cla mp(w, 0.2, 0.8)
    w = tm.ref lect(v, tm.nor mal ize (tm.ve c3(1)))
```

Complex number operations in the form of 2D vectors:

Math (cont)

```
@ti.kernel
def test():
    x = tm.vec2(1, 1) # Complex number 1+1j
    y = tm.vec2(0, 1) # Complex number 1j
    z = tm.cmul(x, y) # vec2(-1, 1) = -1+1j
    w = tm.cdiv(x, y) # vec2(2, 0) = 2+0j
```

Commonly used functions:

```
tm.acos(x)                  tm.min(x, y, ...)
 tm.asin(x)                   tm.mix(x, y, a)
 tm.ata n2(x)                 tm.mod (x,y)
 tm.ceil(x)                   tm.nor mal ize(x)
 tm.cla mp(x, xmin, xmax)     tm.pow(x, a)
                              tm.rou nd(x)
 tm.cos(x)                    tm.sign(x)
 tm.cro ss(x, y)               tm.sin(x)
 tm.dot (x,y)                  tm.smo oth ste p(e0, e1, x
 tm.exp(x)                   )
 tm.flo or(x)                 tm.sqrt(x)
tm.fract(x)                   tm.ste p(edge, x)
 tm.inv ers e(mat)            tm.tan(x)
 tm.len gth(x)                 tm.tanh(x)
 tm.log(x)                    tm.deg rees(x)
 tm.max(x, y, ...)            tm.rad ians(x)
```

Performance

Profiling:

scoped _pr ofiler (default):

```
# Analyzes the perfor mance of the JIT compiler
 ti.pro fil er.p ri nt_ sco ped _pr ofi ler _info()
```

kernel _pr ofiler:

```
# Analyzes the perfor mance of Taichi kernels
 ti.ini t(t i.cpu, kernel _pr ofi ler =True) # Enables the profiler
 ti.pro fil er.p ri nt_ ker nel _pr ofi ler _info() # Displays the results
```

Tuning:

loop_c onfig(): Serializes the outermost for loop that immediately follows it

```
ti.loo p_c onf ig( ser ial ize =True)
 ti.loo p_c onf ig( par all eli ze=8) # Uses 8 threads on the CPU backend
 ti.loo p_c onf ig( blo ck_ dim=16) # Uses 16 threads in each block of the GPU b
ackend
```

Offline cache (default): Saves compilation cache on disk for future runs

```
ti.ini t(o ffl ine _ca che =True)
```

Debugging

Activate debug mode:

```
ti.ini t(a rch =ti.cpu, debug= T
rue)
```

Conciser tracebacks:

```
import sys
sys.tracebacklimit
```

Runtime `print` in Taichi scope:

```
@ti.kernel
 def inside _ta ich i_s cope():
    x = 256
    print('hello', x)
    #=> hello 256
```

Serial execution:

```
# Serializes the p
 ti.ini t(a rch =t
 # Serializes the
 ti.loo p_c onf ig
```

Compile-time `ti.sta tic _print`:

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

 @ti.kernel
 def inside _ta ich i_s cope():
    ti.static_print(y)
    # => 1
    ti.static_print(x.shape)
    # => (2, 3)
    ti.static_print(x.dtype)
    # => DataTy pe.f loat32
```

Runtime assert in Taichi

```
# Activate debug m
 ti.ini t(a rch =t

    x = ti.fie ld(

 @ti.kernel
 def do_sqr t_a ll
    for i in x:
        assert x[i
        x[i] = ti.
```

Compile-time `ti.sta tic _assert`:

```
@ti.func
 def copy(dst: ti.tem pla te(), src: ti.tem pla te())
    ti.static_assert(dst.shape == src.shape, " copy()
    for I in ti.gro upe d(src):
        dst[I] = src[I]
```