

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 to ti.cuda, ti.opengl, and ti.metal if available
1
```

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 floating-point type: ti.f32
ti.init(default_ip=ti.i64) # Sets the default integer type to ti.i64
ti.init(default_fp=ti.f64) # Sets the default floating-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.types.vector(4, ti.f64) # A 64-bit floating-point vector type
mat4x3i = ti.types.matrix(4, 3, int) # A 4x3 integer matrix type
v = vec4d(1, 2, 3, 4) # Creates a vector instance: v
```

Structs:

```
# Defines a compound type vec3 to represent a sphere
vec3 = ti.types.vector(3, float)
# Defines a compound type sphere_type to represent a sphere
sphere_type = ti.types.structured(center=vec3, radius=float)
sphere = sphere_type(center=vec3(0), radius=1)
```

Quantized/low-precision data types:

```
# Defines a 5-bit unsigned integer
u5 = ti.types.quant.int(bits=5, signed=False)
# Defines a 10-bit signed fixed point type within the range [-1, 1]
fixed_type_a = ti.types.quant.fixed(bits=10, min=-1, max=1)
# Defines a 15-bit unsigned floating-point type within the range [0, 1]
float_type_b = ti.types.quant.float(bits=15, min=0, max=1)
```

Sparse matrix (pending)

Data container

Field (global data container):

Declare:

```
# Declares a scalar field
scalar_field = ti.field(int, shape= (640, 480))
# Declares a vector field
vector_field = ti.Vector.field(n=2, dtype= float,
# Declares a matrix field
matrix_field = ti.Matrix.field(n=3, m=2, dtype= f
```

Index:

```
f_0d = ti.field( float, shape=())
f_0d[None] = 1.0 # Accesses the element in a 0D field
f_1d = ti.field(int, shape=10)
f_1d[5] = 1
f_2d = ti.field(int, shape=(10, 10))
f_2d[1, 2] = 255
f_3d = ti.Vector.field(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.field( ti.f32, 4)
x_np = x.to_numpy() # Exports data in Taichi fields to NumPy arrays
x.from_numpy(x_np) # Imports data from NumPy arrays to Taichi fields
x_torch = x.to_torch() # Exports data in Taichi fields to PyTorch tensors
x.from_torch(torch.tensor([1, 7, 3, 5])) # Imports data from PyTorch tensors to Taichi fields
@ti.kernel
def numpy_as_ndarray(arr: ti.ndarray): # Passes a NumPy array to a Taichi kernel
    for i in ti.ndrange(arr.shape[0]):
        ...
```



By **olina**
cheatography.com/olina/

Not published yet.
Last updated 18th October, 2022.
Page 1 of 5.

Sponsored by **Readable.com**
Measure your website readability!
<https://readable.com>

Data container (cont)

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

```
pos = ti.Vector.nd array(2, ti.f32, N)
vel = ti.Vector.nd array(2, ti.f32, N)
force = ti.Vector.nd array(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 ('Window Title', (640, 360)) # Creates a window
while not gui.get_event(t ti.GUI.ESCAPE, ti.GUI.EXIT):
    gui.show() # Displays the window
```

GGUI system:

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

while window.running:
    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.data_oriented # Decorates a class with a @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 class
    def inc(self):
        for i in self.x:
            self.x[i] += 1

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

Taichi dataclass:

A dataclass is a wrapper of `ti.types.struct`. You can define Taichi methods in the Taichi scope.

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

@ti.kernel
def test():
    sphere = Sphere(vec3(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 mathematical functions in the Taichi scope
@ti.kernel
def test():
    a = tm.vec3(1, 2, 3) # A function can take vector
    x = tm.sin(a) # [0.841471, 0.909297, 0.141120] #
    y = tm.floor(a) # [1.000000, 2.000000, 3.000000]
    z = tm.degrees(a) # [57.295780, 114.591560, 171.887340]
```

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 operations
    # ...

    v = tm.vec3(0, 1, 2)
    w = tm.smoothstep(0, 1, v)
    w = tm.clamp(w, 0.2, 0.8)
    w = tm.reflect(v, tm.normalize(tm.vec3(1)))
```

Complex number operations in the form of 2D vectors:



By **olina**
cheatography.com/olina/

Not published yet.
Last updated 18th October, 2022.
Page 2 of 5.

Sponsored by **Readable.com**
Measure your website readability!
<https://readable.com>

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.atan2(x, y)	tm.mod(x, y)
tm.ceil(x)	tm.normalize(x)
tm.clamp(x, xmin, xmax)	tm.pow(x, a)
	tm.round(x)
tm.cos(x)	tm.sign(x)
tm.coss(x, y)	tm.sin(x)
tm.dot(x, y)	tm.smoothstep(e0, e1, x)
tm.exp(x)	
tm.floor(x)	tm.sqrt(x)
tm.fract(x)	tm.step(edge, x)
tm.inverses(mat)	tm.tan(x)
tm.length(x)	tm.tanh(x)
tm.log(x)	tm.degrees(x)
tm.max(x, y, ...)	tm.radians(x)

Performance

Profiling:

scoped_profiler (default):

```
# Analyzes the performance of the JIT compiler
ti.profiler.print_scoped_profiler_info()
```

kernel_profiler:

```
# Analyzes the performance of Taichi kernels
ti.init(t.cpu, kernel_profiler=True) # Enables the profiler
ti.profiler.print_kernel_profiler_info() # Displays the results
```

Tuning:

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

```
ti.loop_config(serialize=True)
ti.loop_config(parallelize=8) # Uses 8 threads on the CPU backend
ti.loop_config(block_dim=16) # Uses 16 threads in each block of the GPU backend
```

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

```
ti.init(offline_cache=True)
```

Debugging

Activate debug mode:

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

Concise tracebacks:

```
import sys
sys.tracebacklimit = 1
```

Runtime print in Taichi scope:

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

Serial execution:

```
# Serializes the parallel execution
ti.init(arch=ti.cpu)
# Serializes the parallel execution
ti.loop_config(serialize=True)
```

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
```

Runtime assert in Taichi

```
# Activate debug mode
ti.init(arch=ti.cpu)

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

Compile-time static assert:

```
@ti.func
def copy(dst: ti.template(), src: ti.template()):
    ti.static_assert(dst.shape == src.shape, "copy()")
    for I in ti.grouped(src):
        dst[I] = src[I]
```

