The Taichi Programming Language

A Hands-on Tutorial @ SIGGRAPH 2020

Yuanming Hu

What is Taichi?

High-performance domain-specific language (DSL) embedded in **Python**, for **computer graphics** applications

- **Productivity** and **portability**: easy to learn, to write, and to share
- **Performance:** data-oriented, parallel, megakernels
- **Spatially sparse** programming: save computation and storage on empty regions
- **Decouple** data structures from computation
- **Differentiable** programming support

Taichi v.s. deep learning frameworks

Why is Taichi different from TensorFlow, PyTorch, NumPy, JAX, ... ?

Quick answer: Taichi uniquely supports **megakernels** and **spatial sparsity**.

Longer answer: Those systems serve their own application domains (e.g., convolutional neural networks) very well, but their design decisions surrounding **immutable, dense tensors** (e.g., feature maps) with **simple, regular operators** (e.g., element-wise add and 2D convolutions) do not serve well more irregular computational patterns, including

- Computer graphics, including physical simulation and rendering
- Irregular neural network layers (e.g., gathering/scattering) that are emerging
- General differentiable programming cases

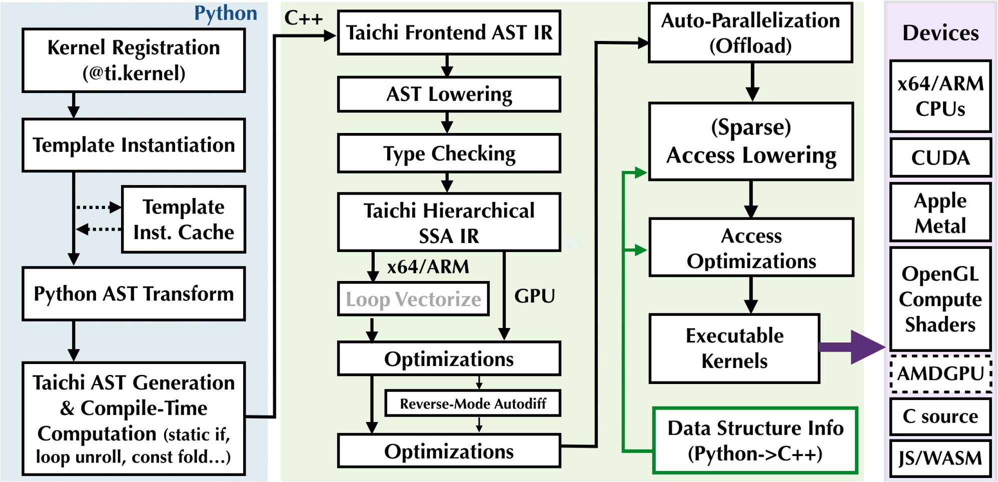
Without Taichi people tend to manually write CUDA or abuse deep learning programming interfaces. Taichi offers performance, productivity, and portability in those cases.

Hello, world! (Julia set, $z \leftarrow z^2 + c$)


```python
import taichi as ti
ti.init(arch=ti.gpu)
n = 320
pixels = ti.field(dtype=float, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: float):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

More details: ▸doc:Hello, world! Run it: `ti example fractal`

Life of a Taichi kernel



Overview

This talk serves as an introductory course on the *syntax* of the Taichi programming language.

- Advanced topics such as data layout specification, sparse data structures, and advanced differentiable programming will *not* be covered in this 1-hour course.
- Slides will be actively updated after the course to keep up with the latest Taichi system (`v0.6.22`).
- More details are available in the Taichi documentation (English & Simplified Chinese).

Note

Many features of Taichi are developed by **the Taichi community**.
Clearly, I am not the only developer :-)

Table of Contents

Installation

Taichi can be installed via `pip` on **64-bit** Python 3.6/3.7/3.8:

```
python3 -m pip install taichi
```

Notes

- Taichi supports Windows, Linux, and OS X.
- Taichi runs on both CPUs and GPUs (CUDA/OpenGL/Apple Metal).
- Build from scratch if your CPU is AArch64 or you use Python 3.9+.

Digression: Taichi's command line interface

Use `python3 -m taichi` or simply `ti` to start Taichi's CLI.

The most important Taichi CLI command: `ti example`

- `ti example`: list all examples
- `ti example mpm99/sdf_renderer/autodiff_regression/...`: run an example
- `ti example -p/-P [example]`: show the code of the example

Taichi has 40+ minimal language examples. Playing with them is the easiest way to learn about this language (and to have fun).

Initialization

Always initialize Taichi with `ti.init()` before you do any Taichi operations.
For example,

```
ti.init(arch=ti.cuda)
```

The most useful argument: `arch`, i.e., the backend (architecture) to use

- `ti.x64/arm/cuda/opengl/metal`: stick to a certain backend.
- `ti.cpu` (default), automatically detects `x64/arm` CPUs.
- `ti.gpu`, try `cuda/metal/opengl`. If none is detected, Taichi falls back on CPUs.

Many other arguments will be introduced later in this course.

Table of Contents

Data types

Taichi is statically and strongly and typed. Supported types include

- Signed integers: `ti.i8/i16/i32/i64`
- Unsigned integers: `ti.u8/u16/u32/u64`
- Float-point numbers: `ti.f32/f64`

`ti.i32` and `ti.f32` are the most commonly used types in Taichi. Boolean values are represented by `ti.i32` for now.

Data type compatibility

The CPU and CUDA backends support all data types. Other backend may miss certain data type support due to backend API constraints. See the documentation for more details.

Fields

Taichi is a *data-oriented* programming language where **fields** are first-class citizens.

- Fields are essentially multi-dimensional arrays
- An element of a field can be either a scalar (`ti.field`), a vector (`ti.Vector.field`), or a matrix (`ti.Matrix.field`)
- Field elements are *always* accessed via the `a[i, j, k]` syntax. (No pointers.)
- Access out-of-bound is undefined behavior in non-debug mode
- *(Advanced) Fields can be spatially sparse*

Table of Contents

Kernels

In Taichi, computation resides in kernels.

1. The language used in Taichi kernels is similar to Python
2. The Taichi kernel language is **compiled, statically-typed, lexically-scoped, parallel and differentiable**
3. Taichi kernels must be decorated with `@ti.kernel`
4. Kernel arguments and return values must be type-hinted

Examples

```python
@ti.kernel
def hello(i: ti.i32):
    a = 40
    print('Hello world!', a + i)

hello(2) # "Hello world! 42"
```

```python
@ti.kernel
def calc() -> ti.i32:
    s = 0
    for i in range(10):
        s += i
    return s # 45
```

Functions

Taichi functions (`@ti.func`) can be called by Taichi kernels and other Taichi functions. No type-hints needed for arguments and return values in `@ti.func`.

Examples

```
@ti.func
def triple(x):
    return x * 3

@ti.kernel
def triple_array():
    for i in range(128):
        a[i] = triple(a[i])
```

Note

Taichi functions will be force-inlined. For now, recursion is not allowed.
A Taichi function can contain at most one `return` statement.

Scalar math

Most Python math operators are supported in Taichi. E.g.,

`a + b, a / b, a // b, a % b, ...`

Math functions:

```
ti.sin(x)
ti.cos(x)
ti.asin(x)
ti.acos(x)
ti.atan2(y, x)
ti.sqrt(x)
ti.cast(x, data_type)
```

```
ti.floor(x)
ti.ceil(x)
ti.inv(x)
ti.tan(x)
ti.tanh(x)
ti.exp(x)
ti.log(x)
```

```
ti.random(data_type)
abs(x)
int(x)
float(x)
max(x, y, ...)
min(x, y, ...)
x ** y
```

Taichi supports **chaining comparisons**. For example, `a < b <= c != d`.

Matrices and linear algebra

`ti.Matrix` is for small matrices (e.g. 3×3) only. If you have 64×64 matrices, please consider using a 2D scalar field.

`ti.Vector` is the same as `ti.Matrix`, except that it has only one column.

Common matrix operations:

```
A.transpose()
A.inverse()
A.trace()
A.determinant(type)
v.normalized()
A.cast(type)
A + B, A * B, A @ B, ...
```

```
R, S = ti.polar_decompose(A, ti.f32)
U, sigma, V = ti.svd(A, ti.f32)
# sigma is a diagonal *matrix*

ti.sin(A)/cos(A)... # element-wise
u.dot(v) # returns a scalar
u.outer_product(v) # returns a matrix
```

Warning

Element-wise product `*` and matrix product `@` have different behaviors.

Parallel for-loops

Two types of `for` loops in Taichi:

- **Range-for loops**, which are no different from Python for loops, except that it will be parallelized when used at the outermost scope. Range-for loops can be nested.
- **Struct-for loops**, which iterates over (sparse) field elements. (More on this later.)

For loops at the outermost scope in a Taichi kernel are **automatically parallelized**.

Range-for loops

Examples

```
@ti.kernel
def fill():
    for i in range(10): # Parallelized
        x[i] += i

        s = 0
        for j in range(5): # Serialized in each parallel thread
            s += j

        y[i] = s

@ti.kernel
def fill_3d():
    # Parallelized for all 3 <= i < 8, 1 <= j < 6, 0 <= k < 9
    for i, j, k in ti.ndrange((3, 8), (1, 6), 9):
        x[i, j, k] = i + j + k
```

Range-for loops

Note

It is the loop **at the outermost scope** that gets parallelized, not the outermost loop.

```
@ti.kernel
def foo():
    for i in range(10): # Parallelized
        ...

@ti.kernel
def bar(k: ti.i32):
    if k > 42:
        for i in range(10): # Serial
            ...
```

Struct-for loops

Examples

```
import taichi as ti

ti.init(arch=ti.gpu)

n = 320
pixels = ti.field(dtype=ti.f32, shape=(n * 2, n))

@ti.kernel
def paint(t: ti.f32):
    for i, j in pixels:  # Parallized over all pixels
        pixels[i, j] = i * 0.001 + j * 0.002 + t

paint(0.3)
```

The struct-for loops iterates over all the field coordinates, i.e. $(0,0),(0,1),(0,2),...,(0,319),(1,0),...,(639,319)$.

Atomic operations

In Taichi, augmented assignments (e.g., `x[i] += 1`) are automatically atomic.

Examples

When modifying global variables in parallel, make sure you use atomic operations. For example, to sum up all the elements in `x`,

```
@ti.kernel
def sum():
    for i in x:
        # Approach 1: Correct
        total[None] += x[i]

        # Approach 2: Correct
        ti.atomic_add(total[None], x[i])

        # Approach 3: Wrong result due to data races
        total[None] = total[None] + x[i]
```

Taichi-scope v.s. Python-scope

Definition

Taichi-scope: Everything decorated with `ti.kernel` and `ti.func`.

Definition

Python-scope: Code outside Taichi-scope.

Note

1. Code in Taichi-scope will be compiled by the Taichi compiler and run on parallel devices.
2. Code in Python-scope is simply Python code and will be executed by the Python interpreter.

Playing with fields in Taichi-scope

Of course, fields can be manipulated in Taichi-scope as well:

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

a = ti.field(dtype=ti.f32, shape=(42, 63)) # A field of 42x63 scalars
b = ti.Vector.field(3, dtype=ti.f32, shape=4) # A field of 4x 3D vectors
C = ti.Matrix.field(2, 2, dtype=ti.f32, shape=(3, 5)) # A field of 3x5 2x2 matrices

@ti.kernel
def foo():
    a[3, 4] = 1
    print('a[3, 4] =', a[3, 4])
    # "a[3, 4] = 1.000000"

    b[2] = [6, 7, 8]
    print('b[0] =', b[0], ', b[2] =', b[2])
    # "b[0] = [[0.000000], [0.000000], [0.000000]] , b[2] = [[6.000000], [7.000000], [8.000000]]"

    C[2, 1][0, 1] = 1
    print('C[2, 1] =', C[2, 1])
    # C[2, 1] = [[0.000000, 1.000000], [0.000000, 0.000000]]

foo()
```

Phases of a Taichi program

1. Initialization: `ti.init(...)`
2. Field allocation: `ti.field`, `ti.Vector.field`, `ti.Matrix.field`
3. Computation (launch kernels, access fields in Python-scope)
4. Optional: restart the Taichi system (clear memory, destroy all variables and kernels): `ti.reset()`

Note

For now, after the first kernel launch or field access in Python-scope, no more field allocation is allowed.

Putting everything together: `fractal.py`

```
import taichi as ti

ti.init(arch=ti.gpu)

n = 320
pixels = ti.field(dtype=ti.f32, shape=(n * 2, n))

@ti.func
def complex_sqr(z):
    return ti.Vector([z[0]**2 - z[1]**2, z[1] * z[0] * 2])

@ti.kernel
def paint(t: ti.f32):
    for i, j in pixels:  # Parallized over all pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        iterations = 0
        while z.norm() < 20 and iterations < 50:
            z = complex_sqr(z) + c
            iterations += 1
        pixels[i, j] = 1 - iterations * 0.02

gui = ti.GUI("Julia Set", res=(n * 2, n))

for i in range(1000000):
    paint(i * 0.03)
    gui.set_image(pixels)
    gui.show()
```

Table of Contents

ODOP: Using classes in Taichi

- Taichi is a data-oriented programming (DOP) language...
- ... but simple DOP makes code modularization hard
- To improve code reusability, Taichi borrows some concepts from object-oriented programming (OOP)
- The hybrid scheme is called **objective data-oriented programming** (ODOP)
- Three important decorators
    - Use `@ti.data_oriented` to decorate your `class`
    - Use `@ti.kernel` to decorate class members functions that are Taichi kernels
    - Use `@ti.func` to decorate class members functions that are Taichi functions
- Development story (Chinese)

ODOP: An example

Demo: `ti example odop_solar` $\mathbf{a} = GM\mathbf{r}/||\mathbf{r}||_2^3$

```
import taichi as ti


@ti.data_oriented
class SolarSystem:
    def __init__(self, n, dt):
        self.n = n
        self.dt = dt
        self.x = ti.Vector.field(2, dtype=ti.f32, shape=n)
        self.v = ti.Vector.field(2, dtype=ti.f32, shape=n)
        self.center = ti.Vector.field(2, dtype=ti.f32, shape=())

    @staticmethod
    @ti.func
    def random_around(center, radius):
        # random number in [center - radius, center + radius)
        return center + radius * (ti.random() - 0.5) * 2

    @ti.kernel
    def initialize(self):
        for i in range(self.n):
            offset = ti.Vector([0.0, self.random_around(0.3, 0.15)])
            self.x[i] = self.center[None] + offset
            self.v[i] = [-offset[1], offset[0]]
            self.v[i] *= 1.5 / offset.norm()
```

ODOP: An example (continued)


```python
    @ti.func
    def gravity(self, pos):
        offset = -(pos - self.center[None])
        return offset / offset.norm()**3

    @ti.kernel
    def integrate(self):
        for i in range(self.n):
            self.v[i] += self.dt * self.gravity(self.x[i])
            self.x[i] += self.dt * self.v[i]


solar = SolarSystem(9, 0.0005)
solar.center[None] = [0.5, 0.5]
solar.initialize()

gui = ti.GUI("Solar System", background_color=0x25A6D9)

while True:
    if gui.get_event():
        if gui.event.key == gui.SPACE and gui.event.type == gui.PRESS:
            solar.initialize()
    for i in range(10):
        solar.integrate()
    gui.circle([0.5, 0.5], radius=20, color=0x8C274C)
    gui.circles(solar.x.to_numpy(), radius=5, color=0xFFFFFF)
    gui.show()
```

Table of Contents

Metaprogramming

Taichi provides metaprogramming tools. Metaprogramming can

- Allow users to pass almost anything (including Taichi fields) to Taichi kernels
- Improve run-time performance by moving run-time costs to compile time
- Achieve dimensionality independence (e.g. write 2D and 3D simulation code simultaneously.)
- Simplify the development of Taichi standard library

Taichi kernels are **lazily instantiated** and a lot of computation can happen at compile time. Every kernel in Taichi is a template kernel, even if it has no template arguments.

Templates

```python
@ti.kernel
def copy(x: ti.template(), y: ti.template(), c: ti.f32):
    for i in x:
        y[i] = x[i] + c
```

Template instantiation

Kernel templates will be instantiated on the first call, and cached for later calls with the same template signature (see doc for more details).

Template argument takes (almost) everything

Feel free to pass fields, classes, functions, strings, and numerical values to arguments hinted as `ti.template()`.

Template kernel instantiation

Be careful!

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

@ti.kernel
def hello(i: ti.template()):
    print(i)

for i in range(100):
    hello(i) # 100 different kernels will be created

@ti.kernel
def world(i: ti.i32):
    print(i)

for i in range(100):
    world(i) # The only instance will be reused
```

Dimensionality-independent programming

Examples

```
@ti.kernel
def copy(x: ti.template(), y: ti.template()):
    for I in ti.grouped(y):
        x[I] = y[I]

@ti.kernel
def array_op(x: ti.template(), y: ti.template()):
    for I in ti.grouped(x):
        # I is a vector of size x.dim() and dtype i32
        y[I] = I[0] + I[1]
    # If x is 2D field, the above is equivalent to
    for i, j in x:
        y[i, j] = i + j
```

Application: write simulation code that works for both 2D & 3D.

Field-size reflection

Fetch field dimensionality info as compile-time constants:

```
import taichi as ti

ti.init()
field = ti.field(dtype=ti.f32, shape=(4, 8, 16, 32, 64))

@ti.kernel
def print_shape(x: ti.template()):
    ti.static_print(x.shape)
    for i in ti.static(range(len(x.shape))):
        print(x.shape[i])

print_shape(field)
```

Compile-time branching

Using compile-time evaluation will allow certain computations to happen when kernels are being instantiated. This saves the overhead of those computations at runtime. (C++17 equivalence: `if constexpr`.)

```
enable_projection = True

@ti.kernel
def static():
    if ti.static(enable_projection): # No runtime overhead
        x[0] = 1
```

Forced loop-unrolling

Use `ti.static(range(...))` to unroll the loops at compile time:

```
import taichi as ti

ti.init()
x = ti.Vector.field(3, dtype=ti.i32, shape=16)

@ti.kernel
def fill():
    for i in x:
        for j in ti.static(range(3)):
            x[i][j] = j
        print(x[i])

fill()
```

Forced loop-unrolling

Why unroll the range-for loops?

- To optimize for performance.
- To loop over vector/matrix elements. Indices into Taichi *vectors or matrices* must be **compile-time constants**. Indices into Taichi *fields* can be run-time variables. For example, if `x` is a 1D field of 3D vectors, accessed as `x[field_index][matrix_index]`. The first index can be a variable, yet the second must be a compile-time constant.

Variable aliasing

Taichi allows programmers to create aliases using `ti.static`. For example, `a = ti.static(a_field_or_kernel_with_very_long_name)`.
This can sometimes improve readability. For example,

```
@ti.kernel
def my_kernel():
    for i, j in field_a:
        field_b[i, j] = some_function(field_a[i, j]) + some_function
            (field_a[i + 1, j])
```

can be simplified into

```
@ti.kernel
def my_kernel():
    a, b, fun = ti.static(field_a, field_b, some_function)
    for i,j in a:
        b[i,j] = fun(a[i,j]) + fun(a[i + 1,j])
```

Table of Contents

Differentiable Programming

Forward programs evaluate $f(\mathbf{x})$; backward (gradient) programs evaluate $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$.

Taichi supports **reverse-mode automatic differentiation (AutoDiff)** that back-propagates gradients w.r.t. a scalar (loss) function $f(\mathbf{x})$.

Two ways to compute gradients:

1. Use Taichi's tape (`ti.Tape(loss)`) for both forward and gradient evaluation.
2. Explicitly use **gradient kernels** for gradient evaluation with more controls.

Gradient-based optimization

$$\mathbf{min}_{\mathbf{x}} \quad L(\mathbf{x}) = \frac{1}{2}\sum_{i=0}^{n-1}(\mathbf{x}_i - \mathbf{y}_i)^2.$$

1. Allocating fields with gradients:

```
x = ti.field(dtype=ti.f32, shape=n, needs_grad=True)
```

2. Defining loss function kernel(s):

```
@ti.kernel
def reduce():
    for i in range(n):
        L[None] += 0.5 * (x[i] - y[i])**2
```

3. Compute loss `with ti.Tape(loss=L): reduce()`
4. Gradient descent: `for i in x: x[i] -= x.grad[i] * 0.1`

Demo: `ti example autodiff_minimization`

Another demo: `ti example autodiff_regression`

Application 1: Forces from potential energy gradients

From the definition of potential energy:

$$\mathbf{f}_i = -\frac{\partial U(\mathbf{x})}{\partial \mathbf{x}_i}$$

Manually deriving gradients is hard. Let's use AutoDiff:

1. Allocate a 0D field to store the potential energy:
   `potential = ti.field(ti.f32, shape=())`.
2. Define forward kernels that computes potential energy from `x[i]`.
3. In a `ti.Tape(loss=potential)`, call the forward kernels.
4. Force on each particle is `-x.grad[i]`.

Application 2: Differentiating a whole physical process

10 Demos: DiffTaichi $(\mathbf{x}_{t+1}, \mathbf{v}_{t+1}, \ldots) = \mathbf{F}(\mathbf{x}_t, \mathbf{v}_t, \ldots)$

Pattern:

```
with ti.Tape(loss=loss):
    for i in range(steps - 1):
        simulate(i)
```

Computational history

Always keep the whole computational history of time steps for end-to-end differentiation. I.e., instead of only allocating `ti.Vector.field(3, dtype=ti.f32, shape=(num_particles))` that stores the latest particles, allocate for the whole simulation process `ti.Vector.field(3, dtype=ti.f32, shape=(num_timesteps, num_particles))`. Do not overwrite! (Use **checkpointing** to reduce memory consumption.)

Table of Contents

Debug mode

`ti.init(debug=True, arch=ti.cpu)` initializes Taichi in debug mode, which enables bound checkers (CPU and CUDA). See the doc more on debug mode.

Examples

```
import taichi as ti
ti.init(debug=True)

a = ti.field(ti.i32, shape=10)
b = ti.field(ti.i32, shape=10)

@ti.kernel
def shift():
    for i in range(10):
        a[i] = b[i + 1] # Runtime error (out-of-bound)
        assert i < 5 # Runtime assertion failure

shift()
```

Table of Contents

Visualize you results

Visualizing 2D results

Simply make use of Taichi's GUI system. Useful functions:

- `gui = ti.GUI("Taichi MLS-MPM-128", res=512, background_color=0x112F41)`
- `gui.circle/gui.circles(x.to_numpy(), radius=1.5, color=colors.to_numpy())`
- `gui.line/triangle/set_image/show/...` [doc]

Visualizing 3D results

Exporting 3D particles and meshes using `ti.PLYWriter` [doc]
Demo: `ti example export_ply/export_mesh`
Use Houdini/Blender to view (and render) your 3D results.

Making a video

Make an mp4 video out of your 2D frames

1. Use `ti.GUI.show` [doc] to save the screenshots. Or simply use `ti.imwrite(img, filename)` [doc].
2. `ti video` creates `video.mp4` using frames under the current folder. To specify frame rate, use `ti video -f 24` or `ti video -f 60`.
3. Convert `mp4` to `gif` and share it online: `ti gif -i input.mp4`.

Make sure `ffmpeg` works!

- Linux and OS X: with high probability you already have `ffmpeg`.
- Windows: install `ffmpeg` on your own [doc].

More information: [Documentation] Export your results.

Thank you!

Next steps

More details: Please check out the Taichi documentation
Found a bug in Taichi? Raise an issue
Join us: Contribution Guidelines

Acknowledgements

Yuanming Hu is grateful to his Ph.D. advisors Prof. Frédo Durand and Prof. Bill Freeman at MIT, and his internship mentor Dr. Vinod Grover at NVIDIA, for supporting the development of Taichi.
Taichi is a collaborative project. We appreciate everyone's contributions.

SIGGRAPH 2020 Taichi Course Online Q&A Session

Time: **Friday, 28 August 2020 9:00am - 9:30am** (Pacific Time)
Please come chat with us! Questions are welcome :-)