The Taichi Programming Language

Advanced Features

Yuanming Hu

MIT CSAIL

Taichi **v0.6.10**

Table of Contents

Table of Contents

ODOP: Using classes in Taichi

- Taichi is a data-oriented programming (DOP) language, but simple DOP makes 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

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(2, dt=ti.f32, shape=n)
        self.v = ti.Vector(2, dt=ti.f32, shape=n)
        self.center = ti.Vector(2, dt=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 tensors) 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 tensors, classes, functions, and numerical values to `ti.template()` arguments.

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 data type i32
        y[I] = I[0] + I[1]
    # If tensor x is 2D, the above is equivalent to
    for i, j in x:
        y[i, j] = i + j
```

Tensor-size reflection

Fetch tensor dimensionality info as compile-time constants:

```
import taichi as ti

tensor = ti.var(ti.f32, shape=(4, 8, 16, 32, 64))

@ti.kernel
def print_tensor_size(x: ti.template()):
    print(x.dim())
    for i in ti.static(range(x.dim())):
        print(x.shape()[i])

print_tensor_size(tensor)
```

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(3, dt=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

When to use range-for loops?

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

Variable aliasing

Creating handy aliases for global variables and functions with cumbersome names can sometimes improve readability:

```
@ti.kernel
def my_kernel():
    for i, j in tensor_a:
        tensor_b[i, j] = some_function(tensor_a[i, j])
```

```
@ti.kernel
def my_kernel():
    a, b, fun = ti.static(tensor_a, tensor_b, some_function)
    for i,j in a:
        b[i,j] = fun(a[i,j])
```

Table of Contents

Differentiable Programming

Forward programs evaluate $f(\mathbf{x})$, differentiable 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

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

1. Allocating tensors with gradients:

```
x = ti.var(dt=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 $0-$D tensor to store the potential energy: `potential = ti.var(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 $(\mathbf{x}_{t+1}, \mathbf{v}_{t+1}, ...) = \mathbf{F}(\mathbf{x}_t, \mathbf{v}_t, ...)$
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(3, dt=ti.f32, shape=(num_particles))` that stores the latest particles, allocate for the whole simulation process `ti.Vector(3, dt=ti.f32, shape=(num_timesteps, num_particles))`. Do not overwrite! (Use **checkpointing** (later in this course) to reduce memory consumption.)

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.

Thank you!

Next steps

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

Questions are welcome!