The Taichi Programming Language

A Brief Introduction

Yuanming Hu

MIT CSAIL

June 1, 2020

Taichi@**v0.6.7**

Overview

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

- Slides will be actively updated.
- 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 :-)

What is Taichi?

High-performance domain-specific language embedded in Python

- Designed for **computer graphics applications** with **productivity** and **portability** in mind
- **Data-oriented, parallel, megakernels**
- **Decouple** data structures from computation
- Access spatially **sparse** tensors as if they are dense
- **Differentiable** programming support

Table of Contents

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
```

- 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+.

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 strongly typed. Supported basics 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.

Tensors

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

- Tensors are essentially multi-dimensional arrays
- An element of a tensor can be either a scalar (`var`), a vector (`ti.Vector`), or a matrix (`ti.Matrix`)
- Tensor elements are always accessed via the `a[i, j, k]` syntax. (No pointers!)
- Access out-of-bound is undefined behavior
- (Advanced) Tensors can be spatially sparse

Tensors v.s. Matrices in Taichi

Although mathematically matrices are treated as 2D tensors, in Taichi, *tensor* and *matrix* are two completely different concepts. Matrices can be used as tensor elements, so you can have tensors with each element being a matrix.

Playing with tensors


```
import taichi as ti
ti.init()
a = ti.var(dt=ti.f32, shape=(42, 63)) # A tensor of 42x63 scalars
b = ti.Vector(3, dt=ti.f32, shape=4)  # A tensor of 4x 3D vectors
C = ti.Matrix(2, 2, dt=ti.f32, shape=(3, 5))
# A tensor of 3x5 2x2 matrices
loss = ti.var(dt=ti.f32, shape=())
# A (0-D) tensor of a single scalar

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][0], b[0][1], b[0][2])
# print(b[0]) is not yet supported.

loss[None] = 3
print(loss[None]) # 3
```

Table of Contents

Kernels

In Taichi, Computation resides in kernels.

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

Examples

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

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

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

Functions

Taichi functions can be called by Taichi kernels and other Taichi functions. They must be decorated with `@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`
Math functions:

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

```
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**, e.g. `a < b <= c != d`.

Matrices and linear algebra

`ti.Matrix` is for small matrices (e.g. 3*x*3) only. If you have 64*x*64 matrices, you should consider using a 2D tensor of scalars.

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

Note

Differentiate element-wise product `*` and matrix product `@`

Common matrix operations:

```
A.transpose()
A.inverse()
A.trace()
A.determinant(type)
A.normalized()
```

```
A.cast(type)
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)
```

Parallel for-loops

For loops in Taichi have two forms

- **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) tensor 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.var(dt=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 tensor 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: OK
        total[None] += x[i]

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

        # Approach 3: Wrong result (the operation is not atomic.)
        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 the 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 tensors in Taichi-scope

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

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

a = ti.var(dt=ti.f32, shape=(42, 63)) # A tensor of 42x63 scalars
b = ti.Vector(3, dt=ti.f32, shape=4) # A tensor of 4x 3D vectors
C = ti.Matrix(2, 2, dt=ti.f32, shape=(3, 5)) # A tensor 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. Tensor allocation: `ti.var`, `ti.Vector`, `ti.Matrix`
3. Computation (launch kernels, access tensors 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 tensor access in Python-scope, no more tensor allocation is allowed.

Putting everything together: `fractal.py`


```python
import taichi as ti

ti.init(arch=ti.gpu)

n = 320
pixels = ti.var(dt=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

Debug mode

`ti.init(debug=True, arch=ti.cpu)` initializes Taichi in debug mode, which enables bound checkers (CPU only).

Examples

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

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

@ti.kernel
def shift():
    for i in range(10):
        a[i] = b[i + 1] # Runtime error in debug mode

shift()
```

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!