taichi Documentation

Release 0.6.26

Taichi Developers

Aug 14, 2020

Overview

CHAPTER 1

Why new programming language

Taichi is a high-performance programming language for computer graphics applications. The design goals are

- Productivity
- Performance
- Portability
- Spatially sparse computation
- Differentiable programming
- Metaprogramming

1.1 Design decisions

- Decouple computation from data structures
- Domain-specific compiler optimizations
- Megakernels
- Two-scale automatic differentiation
- Embedding in Python

CHAPTER 2

Installation

Taichi can be easily installed via `pip`:

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

Note: Currently, Taichi only supports Python 3.6/3.7/3.8 (64-bit).

- On Ubuntu 19.04+, please execute `sudo apt install libtinfo5`.
- On Arch Linux, please execute `yaourt -S ncurses5-compat-libs`.
- On Windows, please install Microsoft Visual C++ Redistributable if you haven't.

2.1 Troubleshooting

2.1.1 Windows issues

- If Taichi crashes and reports `ImportError` on Windows: Please consider installing Microsoft Visual C++ Redistributable.

2.1.2 Python issues

- If `pip` complains that it could not find a satisfying package, i.e.,

```
ERROR: Could not find a version that satisfies the requirement taichi (from␣
↪versions: none)
ERROR: No matching distribution found for taichi
```

  - Make sure you're using Python version 3.6/3.7/3.8:

```
python3 -c "print(__import__('sys').version[:3])"
# 3.6, 3.7 or 3.8
```

- Make sure your Python executable is 64-bit:

```
python3 -c "print(__import__('platform').architecture()[0])"
# 64bit
```

2.1.3 CUDA issues

- If Taichi crashes with the following messages:

```
[Taichi] mode=release
[Taichi] version 0.6.0, supported archs: [cpu, cuda, opengl], commit␣
↪14094f25, python 3.8.2
[W 05/14/20 10:46:49.549] [cuda_driver.h:call_with_warning@60] CUDA Error␣
↪CUDA_ERROR_INVALID_DEVICE: invalid device ordinal while calling mem_
↪advise (cuMemAdvise)
[E 05/14/20 10:46:49.911] Received signal 7 (Bus error)
```

  This might be due to the fact that your NVIDIA GPU is pre-Pascal and has limited support for Unified Memory.

  - **Possible solution**: add `export TI_USE_UNIFIED_MEMORY=0` to your `~/.bashrc`. This disables unified memory usage in CUDA backend.

- If you find other CUDA problems:

  - **Possible solution**: add `export TI_ENABLE_CUDA=0` to your `~/.bashrc`. This disables the CUDA backend completely and Taichi will fall back on other GPU backends such as OpenGL.

2.1.4 OpenGL issues

- If Taichi crashes with a stack backtrace containing a line of `glfwCreateWindow` (see #958):

```
[Taichi] mode=release
[E 05/12/20 18.25:00.129] Received signal 11 (Segmentation Fault)
**********************************
* Taichi Compiler Stack Traceback *
**********************************

... (many lines, omitted)

/lib/python3.8/site-packages/taichi/core/../lib/taichi_core.so: _
↪glfwPlatformCreateWindow
/lib/python3.8/site-packages/taichi/core/../lib/taichi_core.so: glfwCreateWindow
/lib/python3.8/site-packages/taichi/core/../lib/taichi_core.so:␣
↪taichi::lang::opengl::initialize_opengl(bool)

... (many lines, omitted)
```

  This is likely because you are running Taichi on a (virtual) machine with an old OpenGL API. Taichi requires OpenGL 4.3+ to work.

  - **Possible solution**: add `export TI_ENABLE_OPENGL=0` to your `~/.bashrc` even if you initialize Taichi with other backends than OpenGL. This disables the OpenGL backend detection to avoid incompatibilities.

2.1.5 Linux issues

- If Taichi crashes and reports `libtinfo.so.5 not found`:
  - On Ubuntu, execute `sudo apt install libtinfo-dev`.
  - On Arch Linux, first edit `/etc/pacman.conf`, and append these lines:

    ```
    [archlinuxcn]
    Server = https://mirrors.tuna.tsinghua.edu.cn/archlinuxcn/$arch
    ```

    Then execute `sudo pacman -Syy ncurses5-compat-libs`.
- If Taichi crashes and reports `/usr/lib/libstdc++.so.6: version `CXXABI_1.3.11' not found`:

  You might be using Ubuntu 16.04, please try the solution in this thread:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test -y
sudo apt-get update
sudo apt-get install libstdc++6
```

2.1.6 Other issues

- If none of those above address your problem, please report this by opening an issue on GitHub. This would help us improve user experiences and compatibility, many thanks!

CHAPTER 3

Hello, world!

We introduce the Taichi programming language through a very basic *fractal* example.

Running the Taichi code below (`python3 fractal.py` or `ti example fractal`) will give you an animation of Julia set:

```
# fractal.py

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))
```

(continues on next page)

(continued from previous page)

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

Let's dive into this simple Taichi program.

3.1 import taichi as ti

Taichi is a domain-specific language (DSL) embedded in Python. To make Taichi as easy to use as a Python package, we have done heavy engineering with this goal in mind - letting every Python programmer write Taichi programs with minimal learning effort. You can even use your favorite Python package management system, Python IDEs and other Python packages in conjunction with Taichi.

3.2 Portability

Taichi programs run on either CPUs or GPUs. Initialize Taichi according to your hardware platform as follows:

```
# Run on GPU, automatically detect backend
ti.init(arch=ti.gpu)

# Run on GPU, with the NVIDIA CUDA backend
ti.init(arch=ti.cuda)
# Run on GPU, with the OpenGL backend
ti.init(arch=ti.opengl)
# Run on GPU, with the Apple Metal backend, if you are on OS X
ti.init(arch=ti.metal)

# Run on CPU (default)
ti.init(arch=ti.cpu)
```

Note: Supported backends on different platforms:

platform	CPU	CUDA	OpenGL	Metal
Windows	OK	OK	OK	N/A
Linux	OK	OK	OK	N/A
Mac OS X	OK	N/A	N/A	OK

(OK: supported; N/A: not available)

With `arch=ti.gpu`, Taichi will first try to run with CUDA. If CUDA is not supported on your machine, Taichi will fall back on Metal or OpenGL. If no GPU backend (CUDA, Metal, or OpenGL) is supported, Taichi will fall back on CPUs.

Note: When used with the CUDA backend on Windows or ARM devices (e.g. NVIDIA Jetson), Taichi by default allocates 1 GB GPU memory for field storage. You can override this behavior by initializing with `ti.init(arch=ti.cuda, device_memory_GB=3.4)` to allocate `3.4` GB GPU memory, or `ti.init(arch=ti.cuda, device_memory_fraction=0.3)` to allocate `30%` of the total GPU memory.

On other platforms, Taichi will make use of its on-demand memory allocator to adaptively allocate memory.

3.3 Fields

Taichi is a data-oriented programming language where dense or spatially-sparse fields are the first-class citizens. See *Scalar fields* for more details on fields.

In the code above, `pixels = ti.field(dtype=float, shape=(n * 2, n))` allocates a 2D dense field named `pixels` of size `(640, 320)` and element data type `float`.

3.4 Functions and kernels

Computation resides in Taichi **kernels** and Taichi **functions**.

Taichi **kernels** are defined with the decorator `@ti.kernel`. They can be called from Python to perform computation. Kernel arguments must be type-hinted (if any).

Taichi **functions** are defined with the decorator `@ti.func`. They can be called by Taichi kernels or other Taichi functions.

See syntax for more details about Taichi kernels and functions.

The language used in Taichi kernels and functions looks exactly like Python, yet the Taichi frontend compiler converts it into a language that is **compiled, statically-typed, lexically-scoped, parallel and differentiable**.

Note: Taichi-scopes v.s. Python-scopes:

Everything decorated with `@ti.kernel` and `@ti.func` is in Taichi-scope and hence will be compiled by the Taichi compiler.

Everything else is in Python-scope. They are simply Python native code.

Warning: Taichi kernels must be called from the Python-scope. Taichi functions must be called from the Taichi-scope.

Note: For those who come from the world of CUDA, `ti.func` corresponds to `__device__` while `ti.kernel` corresponds to `__global__`.

Warning: Nested kernels are **not supported**.

Nested functions are **supported**.

Recursive functions are **not supported for now**.

3.5 Parallel for-loops

For loops at the outermost scope in a Taichi kernel is **automatically parallelized**. For loops can have two forms, i.e. *range-for loops* and *struct-for loops*.

Range-for loops 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.

```
@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
```

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 are particularly useful when iterating over (sparse) field elements. In the code above, `for i, j in pixels` loops over all the pixel coordinates, i.e. `(0, 0), (0, 1), (0, 2), ... , (0, 319), (1, 0), ..., (639, 319)`.

Note: Struct-for is the key to *Sparse computation (WIP)* in Taichi, as it will only loop over active elements in a sparse field. In dense fields, all elements are active.

Warning: Struct-for loops must live at the outer-most scope of kernels.

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

```
@ti.kernel
def foo():
    for i in x:
        ...

@ti.kernel
def bar(k: ti.i32):
    # The outermost scope is a `if` statement
    if k > 42:
        for i in x: # Not allowed. Struct-fors must live in the outermost scope.
            ...
```

Warning: `break` **is not supported in parallel loops**:

```
@ti.kernel
def foo():
  for i in x:
      ...
      break # Error!

  for i in range(10):
      ...
      break # Error!

@ti.kernel
def foo():
  for i in x:
      for j in range(10):
          ...
          break # OK!
```

3.6 Interacting with other Python packages

3.6.1 Python-scope data access

Everything outside Taichi-scopes (`ti.func` and `ti.kernel`) is simply Python code. In Python-scopes, you can access Taichi field elements using plain indexing syntax. For example, to access a single pixel of the rendered image in Python-scope, simply use:

```
import taichi as ti
pixels = ti.field(ti.f32, (1024, 512))

pixels[42, 11] = 0.7  # store data into pixels
print(pixels[42, 11]) # prints 0.7
```

3.6.2 Sharing data with other packages

Taichi provides helper functions such as `from_numpy` and `to_numpy` for transfer data between Taichi fields and NumPy arrays, So that you can also use your favorite Python packages (e.g. `numpy`, `pytorch`, `matplotlib`) together with Taichi. e.g.:

```
import taichi as ti
pixels = ti.field(ti.f32, (1024, 512))

import numpy as np
arr = np.random.rand(1024, 512)
pixels.from_numpy(arr)   # load numpy data into taichi fields

import matplotlib.pyplot as plt
arr = pixels.to_numpy()  # store taichi data into numpy arrays
plt.imshow(arr)
plt.show()

import matplotlib.cm as cm
cmap = cm.get_cmap('magma')
gui = ti.GUI('Color map')
while gui.running:
    render_pixels()
    arr = pixels.to_numpy()
    gui.set_image(cmap(arr))
    gui.show()
```

See *Interacting with external arrays* for more details.

CHAPTER 4

Syntax

4.1 Taichi-scope vs Python-scope

Code decorated by `@ti.kernel` or `@ti.func` is in the **Taichi-scope**.

They are to be compiled and executed on CPU or GPU devices with high parallelization performance, on the cost of less flexibility.

Note: For people from CUDA, Taichi-scope = **device** side.

Code outside `@ti.kernel` or `@ti.func` is in the **Python-scope**.

They are not compiled by the Taichi compiler and have lower performance but with a richer type system and better flexibility.

Note: For people from CUDA, Python-scope = **host** side.

4.2 Kernels

A Python function decorated by `@ti.kernel` is a **Taichi kernel**:

```
@ti.kernel
def my_kernel():
    ...

my_kernel()
```

Kernels should be called from **Python-scope**.

Note: For people from CUDA, Taichi kernels = `__global__` functions.

4.2.1 Arguments

Kernels can have at most 8 parameters so that you can pass values from Python-scope to Taichi-scope easily.

Kernel arguments must be type-hinted:

```
@ti.kernel
def my_kernel(x: ti.i32, y: ti.f32):
    print(x + y)

my_kernel(2, 3.3)  # prints: 5.3
```

Note: For now, we only support scalars as arguments. Specifying `ti.Matrix` or `ti.Vector` as argument is not supported. For example:

```
@ti.kernel
def bad_kernel(v: ti.Vector):
    ...

@ti.kernel
def good_kernel(vx: ti.f32, vy: ti.f32):
    v = ti.Vector([vx, vy])
    ...
```

4.2.2 Return value

A kernel may or may not have a **scalar** return value. If it does, the type of return value must be hinted:

```
@ti.kernel
def my_kernel() -> ti.f32:
    return 233.33

print(my_kernel())  # 233.33
```

The return value will be automatically cast into the hinted type. e.g.,

```
@ti.kernel
def add_xy() -> ti.i32:  # int32
    return 233.33

print(my_kernel())  # 233, since return type is ti.i32
```

Note: For now, a kernel can only have one scalar return value. Returning `ti.Matrix` or `ti.Vector` is not supported. Python-style tuple return is not supported either. For example:

```
@ti.kernel
def bad_kernel() -> ti.Matrix:
    return ti.Matrix([[1, 0], [0, 1]])  # Error
```

(continues on next page)

(continued from previous page)

```
@ti.kernel
def bad_kernel() -> (ti.i32, ti.f32):
    x = 1
    y = 0.5
    return x, y  # Error
```

4.2.3 Advanced arguments

We also support **template arguments** (see *Template metaprogramming*) and **external array arguments** (see *Interacting with external arrays*) in Taichi kernels. Use `ti.template()` or `ti.ext_arr()` as their type-hints respectively.

Note: When using differentiable programming, there are a few more constraints on kernel structures. See the **Kernel Simplicity Rule** in *Differentiable programming*.

Also, please do not use kernel return values in differentiable programming, since the return value will not be tracked by automatic differentiation. Instead, store the result into a global variable (e.g. `loss[None]`).

4.3 Functions

A Python function decorated by `@ti.func` is a **Taichi function**:

```
@ti.func
def my_func():
    ...

@ti.kernel
def my_kernel():
    ...
    my_func()  # call functions from Taichi-scope
    ...

my_kernel()    # call kernels from Python-scope
```

Taichi functions should be called from **Taichi-scope**.

Note: For people from CUDA, Taichi functions = `__device__` functions.

Note: Taichi functions can be nested.

Warning: Currently, all functions are force-inlined. Therefore, no recursion is allowed.

4.3.1 Arguments and return values

Functions can have multiple arguments and return values. Unlike kernels, arguments in functions don't need to be type-hinted:

```
@ti.func
def my_add(x, y):
    return x + y


@ti.kernel
def my_kernel():
    ...
    ret = my_add(2, 3.3)
    print(ret)  # 5.3
    ...
```

Function arguments are passed by value. So changes made inside function scope won't affect the outside value in the caller:

```
@ti.func
def my_func(x):
    x = x + 1  # won't change the original value of x


@ti.kernel
def my_kernel():
    ...
    x = 233
    my_func(x)
    print(x)  # 233
    ...
```

4.3.2 Advanced arguments

You may use `ti.template()` as type-hint to force arguments to be passed by reference:

```
@ti.func
def my_func(x: ti.template()):
    x = x + 1  # will change the original value of x


@ti.kernel
def my_kernel():
    ...
    x = 233
    my_func(x)
    print(x)  # 234
    ...
```

Note: Unlike kernels, functions **do support vectors or matrices as arguments and return values**:

```
@ti.func
def sdf(u):  # functions support matrices and vectors as arguments. No type-hints␣
↪needed.
```

(continues on next page)

(continued from previous page)

```
    return u.norm() - 1

@ti.kernel
def render(d_x: ti.f32, d_y: ti.f32):  # kernels do not support vector/matrix␣
↪arguments yet. We have to use a workaround.
    d = ti.Vector([d_x, d_y])
    p = ti.Vector([0.0, 0.0])
    t = sdf(p)
    p += d * t
    ...
```

Warning: Functions with multiple `return` statements are not supported for now. Use a **local** variable to store the results, so that you end up with only one `return` statement:

```
# Bad function - two return statements
@ti.func
def safe_sqrt(x):
  if x >= 0:
    return ti.sqrt(x)
  else:
    return 0.0

# Good function - single return statement
@ti.func
def safe_sqrt(x):
  ret = 0.0
  if x >= 0:
    ret = ti.sqrt(x)
  else:
    ret = 0.0
  return ret
```

4.4 Scalar arithmetics

Supported scalar functions:

`ti.`**`sin`**`(`x`)`

`ti.`**`cos`**`(`x`)`

`ti.`**`asin`**`(`x`)`

`ti.`**`acos`**`(`x`)`

`ti.`**`atan2`**`(`x`,` y`)`

`ti.`**`cast`**`(`x`,` *data_type*`)`

`ti.`**`sqrt`**`(`x`)`

`ti.`**`rsqrt`**`(`x`)`

`ti.`**`floor`**`(`x`)`

`ti.`**`ceil`**`(`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*)

pow(*x*, *y*)

Note: Python 3 distinguishes `/` (true division) and `//` (floor division). For example, `1.0 / 2.0 = 0.5`, `1 / 2 = 0.5`, `1 // 2 = 0`, `4.2 // 2 = 2`. Taichi follows this design:

- **true divisions** on integral types will first cast their operands to the default float point type.
- **floor divisions** on float-point types will first cast their operands to the default integer type.

To avoid such implicit casting, you can manually cast your operands to desired types, using `ti.cast`. See *Default precisions* for more details on default numerical types.

Note: When these scalar functions are applied on *Matrices* and *Vectors*, they are applied in an element-wise manner. For example:

```
B = ti.Matrix([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
C = ti.Matrix([[3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])

A = ti.sin(B)
# is equivalent to
for i in ti.static(range(2)):
    for j in ti.static(range(3)):
        A[i, j] = ti.sin(B[i, j])

A = ti.pow(B, 2)
# is equivalent to
for i in ti.static(range(2)):
    for j in ti.static(range(3)):
        A[i, j] = ti.pow(B[i, j], 2)

A = ti.pow(B, C)
# is equivalent to
for i in ti.static(range(2)):
    for j in ti.static(range(3)):
        A[i, j] = ti.pow(B[i, j], C[i, j])

A += 2
# is equivalent to
for i in ti.static(range(2)):
    for j in ti.static(range(3)):
```

(continues on next page)

(continued from previous page)

```
        A[i, j] += 2

A += B
# is equivalent to
for i in ti.static(range(2)):
    for j in ti.static(range(3)):
        A[i, j] += B[i, j]
```

CHAPTER 5

Type system

Taichi supports common numerical data types. Each type is denoted as a character indicating its *category* and a number of *precision bits*, e.g., `i32` and `f64`.

The *category* can be one of:

- `i` for signed integers, e.g. 233, -666
- `u` for unsigned integers, e.g. 233, 666
- `f` for floating point numbers, e.g. 2.33, 1e-4

The *digital number* can be one of:

- `8`
- `16`
- `32`
- `64`

It represents how many **bits** are used in storing the data. The larger the bit number, the higher the precision is.

For example, the two most commonly used types:

- `i32` represents a 32-bit signed integer.
- `f32` represents a 32-bit floating pointer number.

5.1 Supported types

Currently, supported basic types in Taichi are

- int8 `ti.i8`
- int16 `ti.i16`
- int32 `ti.i32`

- int64 `ti.i64`
- uint8 `ti.u8`
- uint16 `ti.u16`
- uint32 `ti.u32`
- uint64 `ti.u64`
- float32 `ti.f32`
- float64 `ti.f64`

Note: Supported types on each backend:

type	CPU/CUDA	OpenGL	Metal
i8	OK	N/A	OK
i16	OK	N/A	OK
i32	OK	OK	OK
i64	OK	EXT	N/A
u8	OK	N/A	OK
u16	OK	N/A	OK
u32	OK	N/A	OK
u64	OK	N/A	N/A
f32	OK	OK	OK
f64	OK	OK	N/A

(OK: supported, EXT: require extension, N/A: not available)

Note: Boolean types are represented using `ti.i32`.

5.2 Type promotion

Binary operations on different types will give you a promoted type, following the C programming language convention, e.g.:

- `i32 + f32 = f32` (integer + float = float)
- `i32 + i64 = i64` (less-bits + more-bits = more-bits)

Basically it will try to choose the more precise type to contain the result value.

5.3 Default precisions

By default, all numerical literals have 32-bit precisions. For example, `42` has type `ti.i32` and `3.14` has type `ti.f32`.

Default integer and float-point precisions (`default_ip` and `default_fp`) can be specified when initializing Taichi:

```
ti.init(default_fp=ti.f32)
ti.init(default_fp=ti.f64)

ti.init(default_ip=ti.i32)
ti.init(default_ip=ti.i64)
```

Also note that you may use `float` or `int` in type definitions as aliases for default precisions, e.g.:

```
ti.init(default_ip=ti.i64, default_fp=ti.f32)

x = ti.var(float, 5)
y = ti.var(int, 5)
# is equivalent to:
x = ti.var(ti.f32, 5)
y = ti.var(ti.i64, 5)

def func(a: float) -> int:
    ...

# is equivalent to:
def func(a: ti.f32) -> ti.i64:
    ...
```

5.4 Type casts

5.4.1 Implicit casts

Warning: The type of a variable is **determinated on it's initialization**.

When a *low-precision* variable is assigned to a *high-precision* variable, it will be implicitly promoted to the *wide* type and no warning will be raised:

```
a = 1.7
a = 1
print(a)  # 1.0
```

When a *high-precision* variable is assigned to a *low-precision* type, it will be implicitly down-cast into the *low-precision* type and Taichi will raise a warning:

```
a = 1
a = 1.7
print(a)  # 1
```

5.4.2 Explicit casts

You may use `ti.cast` to explicitly cast scalar values between different types:

```
a = 1.7
b = ti.cast(a, ti.i32)  # 1
c = ti.cast(b, ti.f32)  # 1.0
```

Equivalently, use `int()` and `float()` to convert values to float-point or integer types of default precisions:

```
a = 1.7
b = int(a)     # 1
c = float(a)   # 1.0
```

5.4.3 Casting vectors and matrices

Type casts applied to vectors/matrices are element-wise:

```
u = ti.Vector([2.3, 4.7])
v = int(u)                # ti.Vector([2, 4])
# If you are using ti.i32 as default_ip, this is equivalent to:
v = ti.cast(u, ti.i32)  # ti.Vector([2, 4])
```

5.4.4 Bit casting

Use `ti.bit_cast` to bit-cast a value into another data type. The underlying bits will be preserved in this cast. The new type must have the same width as the the old type. For example, bit-casting `i32` to `f64` is not allowed. Use this operation with caution.

CHAPTER 6

Fields and matrices

Fields are global variables provided by Taichi. Fields can be either sparse or dense. An element of a field can be either a scalar or a vector/matrix.

Note: Matrices can be used as field elements, so you can have fields with each element being a matrix.

6.1 Scalar fields

- Every global variable is an N-dimensional field.
  - Global `scalars` are treated as 0-D scalar fields.
- Fields are always accessed using indices
  - E.g. `x[i, j, k]` if `x` is a 3D scalar field.
  - Even when accessing 0-D field `x`, use `x[None] = 0` instead of `x = 0`. Please **always** use indexing to access entries in fields.
- Field values are initially zero.
- Sparse fields are initially inactive.
- See *Scalar fields* for more details.

6.2 Matrix fields

Field elements can also be matrices.

Suppose you have a `128 x 64` field called `A`, each element containing a `3 x 2` matrix. To allocate a `128 x 64` matrix field which has a `3 x 2` matrix for each of its entry, use the statement `A = ti.Matrix.field(3, 2, dtype=ti.f32, shape=(128, 64))`.

- If you want to get the matrix of grid node `i, j`, please use `mat = A[i, j]`. `mat` is simply a `3 x 2` matrix
- To get the element on the first row and second column of that matrix, use `mat[0, 1]` or `A[i, j][0, 1]`.
- As you may have noticed, there are **two** indexing operators `[]` when you load an matrix element from a global matrix field: the first is for field indexing, the second for matrix indexing.
- `ti.Vector` is simply an alias of `ti.Matrix`.
- See *Matrices* for more on matrices.

6.3 Matrix size

For performance reasons matrix operations will be unrolled, therefore we suggest using only small matrices. For example, `2x1`, `3x3`, `4x4` matrices are fine, yet `32x6` is probably too big as a matrix size.

Warning: Due to the unrolling mechanisms, operating on large matrices (e.g. `32x128`) can lead to very long compilation time and low performance.

If you have a dimension that is too large (e.g. `64`), it's better to declare a field of size `64`. E.g., instead of declaring `ti.Matrix.field(64, 32, dtype=ti.f32, shape=(3, 2))`, declare `ti.Matrix.field(3, 2, dtype=ti.f32, shape=(64, 32))`. Try to put large dimensions to fields instead of matrices.

CHAPTER 7

Atomic operations

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

Warning: 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 since the operation is not atomic.
        total[None] = total[None] + x[i]
```

Note: When atomic operations are applied to local values, the Taichi compiler will try to demote these operations into their non-atomic counterparts.

Apart from the augmented assignments, explicit atomic operations, such as `ti.atomic_add`, also do read-modify-write atomically. These operations additionally return the **old value** of the first argument.

Below is a list of all explicit atomic operations:

`ti.`**`atomic_add`**(*x*, *y*)

`ti.`**`atomic_sub`**(*x*, *y*)

Atomically compute `x + y` or `x - y` and store the result in `x`.

Returns The old value of `x`.

For example,

```
x[i] = 3
y[i] = 4
z[i] = ti.atomic_add(x[i], y[i])
# now x[i] = 7, y[i] = 4, z[i] = 3
```

ti.**atomic_and**(*x*, *y*)

ti.**atomic_or**(*x*, *y*)

ti.**atomic_xor**(*x*, *y*)

Atomically compute `x & y` (bitwise and), `x | y` (bitwise or), or `x ^ y` (bitwise xor), and store the result in `x`.

Returns The old value of `x`.

Note: Supported atomic operations on each backend:

type	CPU/CUDA	OpenGL	Metal
`i32`	OK	OK	OK
`f32`	OK	OK	OK
`i64`	OK	EXT	N/A
`f64`	OK	EXT	N/A

(OK: supported; EXT: require extension; N/A: not available)

CHAPTER 8

Interacting with external arrays

External arrays refer to `numpy.ndarray` or `torch.Tensor`.

8.1 Conversion between Taichi fields and external arrays

Use `to_numpy`/`from_numpy`/`to_torch`/`from_torch`:

```
import taichi as ti
import numpy as np

ti.init()

n = 4
m = 7

# Taichi fields
val = ti.field(ti.i32, shape=(n, m))
vec = ti.Vector.field(3, dtype=ti.i32, shape=(n, m))
mat = ti.Matrix.field(3, 4, dtype=ti.i32, shape=(n, m))

# Scalar
arr = np.ones(shape=(n, m), dtype=np.int32)

val.from_numpy(arr)

arr = val.to_numpy()

# Vector
arr = np.ones(shape=(n, m, 3), dtype=np.int32)

vec.from_numpy(arr)

arr = np.ones(shape=(n, m, 3, 1), dtype=np.int32)
```

(continues on next page)

(continued from previous page)

```
vec.from_numpy(arr)

arr = vec.to_numpy()
assert arr.shape == (n, m, 3)

arr = vec.to_numpy(keep_dims=True)
assert arr.shape == (n, m, 3, 1)

# Matrix
arr = np.ones(shape=(n, m, 3, 4), dtype=np.int32)

mat.from_numpy(arr)

arr = mat.to_numpy()
assert arr.shape == (n, m, 3, 4)
```

TODO: add API reference

8.2 Using external arrays as Taichi kernel parameters

The type hint for external array parameters is `ti.ext_arr()`. Please see the example below. Note that struct-for's on external arrays are not supported.

```
import taichi as ti
import numpy as np

ti.init()

n = 4
m = 7

val = ti.field(ti.i32, shape=(n, m))

@ti.kernel
def test_numpy(arr: ti.ext_arr()):
  for i in range(n):
    for j in range(m):
      arr[i, j] += i + j

a = np.empty(shape=(n, m), dtype=np.int32)

for i in range(n):
  for j in range(m):
    a[i, j] = i * j

test_numpy(a)

for i in range(n):
  for j in range(m):
    assert a[i, j] == i * j + i + j
```

CHAPTER 9

Scalar fields

Taichi fields are used to store data.

Field **elements** could be either a scalar, a vector, or a matrix (see *Matrices*). In this paragraph, we will only talk about **scalar fields**, whose elements are simply scalars.

Fields can have up to eight **dimensions**.

- A 0D scalar field is simply a single scalar.
- A 1D scalar field is a 1D linear array.
- A 2D scalar field can be used to represent a 2D regular grid of values. For example, a gray-scale image.
- A 3D scalar field can be used for volumetric data.

Fields could be either dense or sparse, see ref:*sparse* for details on sparse fields. We will only talk about **dense fields** in this paragraph.

Note: We once used the term **tensor** instead of **field**. **Tensor** will no longer be used.

9.1 Declaration

`ti.field`(*dtype*, *shape* = *None*, *offset* = *None*)

Parameters

- **`dtype`** – (DataType) type of the field element
- **`shape`** – (optional, scalar or tuple) the shape of field
- **`offset`** – (optional, scalar or tuple) see *Coordinate offsets*

For example, this creates a *dense* field with four `int32` as elements:

```
x = ti.field(ti.i32, shape=4)
```

This creates a 4x3 *dense* field with `float32` elements:

```
x = ti.field(ti.f32, shape=(4, 3))
```

If shape is `()` (empty tuple), then a 0-D field (scalar) is created:

```
x = ti.field(ti.f32, shape=())
```

Then access it by passing `None` as index:

```
x[None] = 2
```

If shape is **not provided** or `None`, the user must manually `place` it afterwards:

```
x = ti.field(ti.f32)
ti.root.dense(ti.ij, (4, 3)).place(x)
# equivalent to: x = ti.field(ti.f32, shape=(4, 3))
```

Note: Not providing `shape` allows you to *place* the field in a layout other than the default *dense*, see *Advanced dense layouts* for more details.

Warning: All variables should be created and placed before any kernel invocation or any of them accessed from python-scope. For example:

```
x = ti.field(ti.f32)
x[None] = 1 # ERROR: x not placed!
```

```
x = ti.field(ti.f32, shape=())
@ti.kernel
def func():
    x[None] = 1

func()
y = ti.field(ti.f32, shape=())
# ERROR: cannot create fields after kernel invocation!
```

```
x = ti.field(ti.f32, shape=())
x[None] = 1
y = ti.field(ti.f32, shape=())
# ERROR: cannot create fields after any field accesses from the Python-scope!
```

9.2 Accessing components

You can access an element of the Taichi field by an index or indices.

a[p, q, ...]

Parameters

- **a** – (ti.field) the sclar field

- **p** – (scalar) index of the first field dimension
- **q** – (scalar) index of the second field dimension

Returns (scalar) the element at `[p, q, ...]`

This extracts the element value at index `[3, 4]` of field `a`:

```
x = a[3, 4]
```

This sets the element value at index `2` of 1D field `b` to `5`:

```
b[2] = 5
```

Note: In Python, x[(exp1, exp2, ..., expN)] is equivalent to x[exp1, exp2, ..., expN]; the latter is just syntactic sugar for the former.

Note: The returned value can also be `Vector` / `Matrix` if `a` is a vector/matrix field, see *Vectors* for more details.

9.3 Meta data

`a.`**`shape`**

Parameters **a** – (ti.field) the field

Returns (tuple) the shape of field `a`

```
x = ti.field(ti.i32, (6, 5))
x.shape  # (6, 5)

y = ti.field(ti.i32, 6)
y.shape  # (6,)

z = ti.field(ti.i32, ())
z.shape  # ()
```

`a.`**`dtype`**

Parameters **a** – (ti.field) the field

Returns (DataType) the data type of `a`

```
x = ti.field(ti.i32, (2, 3))
x.dtype  # ti.i32
```

`a.`**`parent`**(*n = 1*)

Parameters

- **a** – (ti.field) the field
- **n** – (optional, scalar) the number of parent steps, i.e. `n=1` for parent, `n=2` grandparent, etc.

Returns (SNode) the parent of `a`'s containing SNode

```
x = ti.field(ti.i32)
y = ti.field(ti.i32)
blk1 = ti.root.dense(ti.ij, (6, 5))
blk2 = blk1.dense(ti.ij, (3, 2))
blk1.place(x)
blk2.place(y)

x.parent()    # blk1
y.parent()    # blk2
y.parent(2)   # blk1
```

See *Structural nodes (SNodes)* for more details.

CHAPTER 10

Vectors

A vector in Taichi can have two forms:

- as a temporary local variable. An `n` component vector consists of `n` scalar values.
- as an element of a global field. In this case, the field is an N-dimensional array of `n` component vectors.

In fact, `Vector` is simply an alias of `Matrix`, just with `m = 1`. See *Matrices* and *Fields and matrices* for more details.

10.1 Declaration

10.1.1 As global vector fields

`ti.Vector.`**`field`**(*n*, *dtype*, *shape* = *None*, *offset* = *None*)

Parameters

- **`n`** – (scalar) the number of components in the vector
- **`dtype`** – (DataType) data type of the components
- **`shape`** – (optional, scalar or tuple) shape of the vector field, see *Fields and matrices*
- **`offset`** – (optional, scalar or tuple) see *Coordinate offsets*

For example, this creates a 3-D vector field of the shape of `5x4`:

```
# Python-scope
a = ti.Vector.field(3, dtype=ti.f32, shape=(5, 4))
```

Note: In Python-scope, `ti.field` declares *Scalar fields*, while `ti.Vector.field` declares vector fields.

10.1.2 As a temporary local variable

`ti.Vector`([*x*, *y*, ...])

Parameters

- **x** – (scalar) the first component of the vector
- **y** – (scalar) the second component of the vector

For example, this creates a 3D vector with components (2, 3, 4):

```
# Taichi-scope
a = ti.Vector([2, 3, 4])
```

10.2 Accessing components

10.2.1 As global vector fields

a[p, q, ...][i]

Parameters

- **a** – (ti.Vector.field) the vector
- **p** – (scalar) index of the first field dimension
- **q** – (scalar) index of the second field dimension
- **i** – (scalar) index of the vector component

This extracts the first component of vector `a[6, 3]`:

```
x = a[6, 3][0]

# or
vec = a[6, 3]
x = vec[0]
```

Note: **Always** use two pairs of square brackets to access scalar elements from vector fields.

- The indices in the first pair of brackets locate the vector inside the vector fields;
- The indices in the second pair of brackets locate the scalar element inside the vector.

For 0-D vector fields, indices in the first pair of brackets should be `[None]`.

10.2.2 As a temporary local variable

a[i]

Parameters

- **a** – (Vector) the vector
- **i** – (scalar) index of the component

For example, this extracts the first component of vector `a`:

```
x = a[0]
```

This sets the second component of `a` to 4:

```
a[1] = 4
```

TODO: add descriptions about `a(i, j)`

10.3 Methods

`a.`**`norm`**`(`*eps = 0*`)`

Parameters

- **a** – (ti.Vector)
- **eps** – (optional, scalar) a safe-guard value for `sqrt`, usually 0. See the note below.

Returns (scalar) the magnitude / length / norm of vector

For example,

```
a = ti.Vector([3, 4])
a.norm() # sqrt(3*3 + 4*4 + 0) = 5
```

`a.norm(eps)` is equivalent to `ti.sqrt(a.dot(a) + eps)`

Note: To safeguard the operator's gradient on zero vectors during differentiable programming, set `eps` to a small, positive value such as `1e-5`.

`a.`**`norm_sqr`**`()`

Parameters **a** – (ti.Vector)

Returns (scalar) the square of the magnitude / length / norm of vector

For example,

```
a = ti.Vector([3, 4])
a.norm_sqr() # 3*3 + 4*4 = 25
```

`a.norm_sqr()` is equivalent to `a.dot(a)`

`a.`**`normalized`**`()`

Parameters **a** – (ti.Vector)

Returns (ti.Vector) the normalized / unit vector of `a`

For example,

```
a = ti.Vector([3, 4])
a.normalized() # [3 / 5, 4 / 5]
```

`a.normalized()` is equivalent to `a / a.norm()`.

`a.`**`dot`**`(`*b*`)`

Parameters

- **a** – (ti.Vector)
- **b** – (ti.Vector)

Returns (scalar) the dot (inner) product of `a` and `b`

E.g.,

```
a = ti.Vector([1, 3])
b = ti.Vector([2, 4])
a.dot(b) # 1*2 + 3*4 = 14
```

`a.cross(b)`

Parameters

- **a** – (ti.Vector, 2 or 3 components)
- **b** – (ti.Vector of the same size as a)

Returns (scalar (for 2D inputs), or 3D Vector (for 3D inputs)) the cross product of `a` and `b`

We use a right-handed coordinate system. E.g.,

```
a = ti.Vector([1, 2, 3])
b = ti.Vector([4, 5, 6])
c = ti.cross(a, b)
# c = [2*6 - 5*3, 4*3 - 1*6, 1*5 - 4*2] = [-3, 6, -3]

p = ti.Vector([1, 2])
q = ti.Vector([4, 5])
r = ti.cross(a, b)
# r = 1*5 - 4*2 = -3
```

`a.outer_product(b)`

Parameters

- **a** – (ti.Vector)
- **b** – (ti.Vector)

Returns (ti.Matrix) the outer product of `a` and `b`

E.g.,

```
a = ti.Vector([1, 2])
b = ti.Vector([4, 5, 6])
c = ti.outer_product(a, b) # NOTE: c[i, j] = a[i] * b[j]
# c = [[1*4, 1*5, 1*6], [2*4, 2*5, 2*6]]
```

Note: The outer product should not be confused with the cross product (`ti.cross`). For example, `a` and `b` do not have to be 2- or 3-component vectors for this function.

`a.cast(dt)`

Parameters

- **a** – (ti.Vector)
- **dt** – (DataType)

Returns (ti.Vector) vector with all components of `a` casted into type `dt`

E.g.,

```
# Taichi-scope
a = ti.Vector([1.6, 2.3])
a.cast(ti.i32) # [2, 3]
```

Note: Vectors are special matrices with only 1 column. In fact, `ti.Vector` is just an alias of `ti.Matrix`.

10.4 Metadata

`a.n`

Parameters **a** – (ti.Vector or ti.Vector.field)

Returns

(scalar) return the dimensionality of vector `a`

E.g.,

```
# Taichi-scope
a = ti.Vector([1, 2, 3])
a.n  # 3
```

```
# Python-scope
a = ti.Vector.field(3, dtype=ti.f32, shape=())
a.n  # 3
```

TODO: add element wise operations docs

CHAPTER 11

Matrices

- `ti.Matrix` is for small matrices (e.g. *3x3*) only. If you have *64x64* matrices, you should consider using a 2D scalar field.
- `ti.Vector` is the same as `ti.Matrix`, except that it has only one column.
- Differentiate element-wise product `*` and matrix product `@`.
- `ti.Vector.field(n, dtype=ti.f32)` or `ti.Matrix.field(n, m, dtype=ti.f32)` to create vector/matrix fields.
- `A.transpose()`
- `R, S = ti.polar_decompose(A, ti.f32)`
- `U, sigma, V = ti.svd(A, ti.f32)` (Note that `sigma` is a `3x3` diagonal matrix)
- `any(A)` (Taichi-scope only)
- `all(A)` (Taichi-scope only)

TODO: doc here better like Vector. WIP

A matrix in Taichi can have two forms:

- as a temporary local variable. An `n by m` matrix consists of `n * m` scalar values.
- as a an element of a global field. In this case, the field is an N-dimensional array of `n by m` matrices.

11.1 Declaration

11.1.1 As global matrix fields

`ti.Matrix.`**`field`**(*n*, *m*, *dtype*, *shape* = *None*, *offset* = *None*)

Parameters

- **n** – (scalar) the number of rows in the matrix

- **m** – (scalar) the number of columns in the matrix
- **dtype** – (DataType) data type of the components
- **shape** – (optional, scalar or tuple) shape of the matrix field, see *Fields and matrices*
- **offset** – (optional, scalar or tuple) see *Coordinate offsets*

For example, this creates a 5x4 matrix field with each entry being a 3x3 matrix:

```
# Python-scope
a = ti.Matrix.field(3, 3, dtype=ti.f32, shape=(5, 4))
```

Note: In Python-scope, `ti.field` declares a *Scalar fields*, while `ti.Matrix.field` declares a matrix field.

11.1.2 As a temporary local variable

`ti.Matrix`(*[[x, y, ...][, z, w, ...], ...]*)

Parameters

- **x** – (scalar) the first component of the first row
- **y** – (scalar) the second component of the first row
- **z** – (scalar) the first component of the second row
- **w** – (scalar) the second component of the second row

For example, this creates a 2x3 matrix with components (2, 3, 4) in the first row and (5, 6, 7) in the second row:

```
# Taichi-scope
a = ti.Matrix([[2, 3, 4], [5, 6, 7]])
```

`ti.Matrix.rows`(*[v0, v1, v2, ...]*)

`ti.Matrix.cols`(*[v0, v1, v2, ...]*)

Parameters

- **v0** – (vector) vector of elements forming first row (or column)
- **v1** – (vector) vector of elements forming second row (or column)
- **v2** – (vector) vector of elements forming third row (or column)

For example, this creates a 3x3 matrix by concactinating vectors into rows (or columns):

```
# Taichi-scope
v0 = ti.Vector([1.0, 2.0, 3.0])
v1 = ti.Vector([4.0, 5.0, 6.0])
v2 = ti.Vector([7.0, 8.0, 9.0])

# to specify data in rows
a = ti.Matrix.rows([v0, v1, v2])

# to specify data in columns instead
a = ti.Matrix.cols([v0, v1, v2])

# lists can be used instead of vectors
a = ti.Matrix.rows([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
```

11.2 Accessing components

11.2.1 As global matrix fields

a[p, q, ...][i, j]

Parameters

- **a** – (ti.Matrix.field) the matrix field
- **p** – (scalar) index of the first field dimension
- **q** – (scalar) index of the second field dimension
- **i** – (scalar) row index of the matrix
- **j** – (scalar) column index of the matrix

This extracts the first element in matrix `a[6, 3]`:

```
x = a[6, 3][0, 0]

# or
mat = a[6, 3]
x = mat[0, 0]
```

Note: **Always** use two pair of square brackets to access scalar elements from matrix fields.

- The indices in the first pair of brackets locate the matrix inside the matrix fields;
- The indices in the second pair of brackets locate the scalar element inside the matrix.

For 0-D matrix fields, indices in the first pair of brackets should be `[None]`.

11.2.2 As a temporary local variable

a[i, j]

Parameters

- **a** – (Matrix) the matrix
- **i** – (scalar) row index of the matrix
- **j** – (scalar) column index of the matrix

For example, this extracts the element in row 0 column 1 of matrix `a`:

```
x = a[0, 1]
```

This sets the element in row 1 column 3 of `a` to 4:

```
a[1, 3] = 4
```

11.3 Methods

`a.transpose()`

Parameters **a** – (ti.Matrix) the matrix

Returns (ti.Matrix) the transposed matrix of `a`.

For example:

```
a = ti.Matrix([[2, 3], [4, 5]])
b = a.transpose()
# Now b = ti.Matrix([[2, 4], [3, 5]])
```

Note: `a.transpose()` will not effect the data in `a`, it just return the result.

`a.trace()`

Parameters **a** – (ti.Matrix) the matrix

Returns (scalar) the trace of matrix `a`.

The return value can be computed as `a[0, 0] + a[1, 1] + ...`.

`a.determinant()`

Parameters **a** – (ti.Matrix) the matrix

Returns (scalar) the determinant of matrix `a`.

Note: The matrix size of matrix must be 1x1, 2x2, 3x3 or 4x4 for now.

This function only works in Taichi-scope for now.

`a.inverse()`

Parameters **a** – (ti.Matrix) the matrix

Returns (ti.Matrix) the inverse of matrix `a`.

Note: The matrix size of matrix must be 1x1, 2x2, 3x3 or 4x4 for now.

This function only works in Taichi-scope for now.

CHAPTER 12

Structural nodes (SNodes)

After writing the computation code, the user needs to specify the internal data structure hierarchy. Specifying a data structure includes choices at both the macro level, dictating how the data structure components nest with each other and the way they represent sparsity, and the micro level, dictating how data are grouped together (e.g. structure of arrays vs. array of structures). Taichi provides *Structural Nodes (SNodes)* to compose the hierarchy and particular properties. These constructs and their semantics are listed below:

- dense: A fixed-length contiguous array.
- bitmasked: This is similar to dense, but it also uses a mask to maintain sparsity information, one bit per child.
- pointer: Store pointers instead of the whole structure to save memory and maintain sparsity.
- dynamic: Variable-length array, with a predefined maximum length. It serves the role of `std::vector` in C++ or `list` in Python, and can be used to maintain objects (e.g. particles) contained in a block.

See *Advanced dense layouts* for more details. `ti.root` is the root node of the data structure.

`snode.`**`place`**`(`*x*, ...`)`

Parameters

- **`snode`** – (SNode) where to place
- **`a`** – (ti.field) field(s) to be placed

Returns (SNode) the `snode` itself

The following code places two 0-D fields named `x` and `y`:

```
x = ti.field(dtype=ti.i32)
y = ti.field(dtype=ti.f32)
ti.root.place(x, y)
assert x.snode == y.snode
```

`field.`**`shape`**`()`

Parameters **`a`** – (ti.field)

Returns (tuple of integers) the shape of field

Equivalent to `field.snode.shape`.

For example,

```
ti.root.dense(ti.ijk, (3, 5, 4)).place(x)
x.shape  # returns (3, 5, 4)
```

`field.`**`snode`**`()`

Parameters **a** – (ti.field)

Returns (SNode) the structual node where `field` is placed

```
x = ti.field(dtype=ti.i32)
y = ti.field(dtype=ti.f32)
blk1 = ti.root.dense(ti.i, 4)
blk1.place(x, y)
assert x.snode == blk1
```

`snode.`**`shape`**`()`

Parameters **snode** – (SNode)

Returns (tuple) the size of node along that axis

```
blk1 = ti.root
blk2 = blk1.dense(ti.i,  3)
blk3 = blk2.dense(ti.jk, (5, 2))
blk4 = blk3.dense(ti.k,  2)
blk1.shape  # ()
blk2.shape  # (3, )
blk3.shape  # (3, 5, 2)
blk4.shape  # (3, 5, 4)
```

`snode.`**`parent`**`(`*`n = 1`*`)`

Parameters

- **snode** – (SNode)
- **n** – (optional, scalar) the number of steps, i.e. `n=1` for parent, `n=2` grandparent, etc.

Returns (SNode) the parent node of `snode`

```
blk1 = ti.root.dense(ti.i, 8)
blk2 = blk1.dense(ti.j, 4)
blk3 = blk2.bitmasked(ti.k, 6)
blk1.parent()  # ti.root
blk2.parent()  # blk1
blk3.parent()  # blk2
blk3.parent(1) # blk2
blk3.parent(2) # blk1
blk3.parent(3) # ti.root
blk3.parent(4) # None
```

12.1 Node types

`snode.`**`dense`**`(`*`indices`*`, `*`shape`*`)`

Parameters

- **snode** – (SNode) parent node where the child is derived from
- **indices** – (Index or Indices) indices used for this node
- **shape** – (scalar or tuple) shape of the field

Returns (SNode) the derived child node

The following code places a 1-D field of size `3`:

```
x = ti.field(dtype=ti.i32)
ti.root.dense(ti.i, 3).place(x)
```

The following code places a 2-D field of shape `(3, 4)`:

```
x = ti.field(dtype=ti.i32)
ti.root.dense(ti.ij, (3, 4)).place(x)
```

Note: If `shape` is a scalar and there are multiple indices, then `shape` will be automatically expanded to fit the number of indices. For example,

```
snode.dense(ti.ijk, 3)
```

is equivalent to

```
snode.dense(ti.ijk, (3, 3, 3))
```

`snode.`**`dynamic`**(*index*, *size*, *chunk_size* = *None*)

Parameters

- **snode** – (SNode) parent node where the child is derived from
- **index** – (Index) the `dynamic` node indices
- **size** – (scalar) the maximum size of the dynamic node
- **chunk_size** – (optional, scalar) the number of elements in each dynamic memory allocation chunk

Returns (SNode) the derived child node

`dynamic` nodes acts like `std::vector` in C++ or `list` in Python. Taichi's dynamic memory allocation system allocates its memory on the fly.

The following places a 1-D dynamic field of maximum size `16`:

```
ti.root.dynamic(ti.i, 16).place(x)
```

`snode.`**`bitmasked`**`()`

`snode.`**`pointer`**`()`

`snode.`**`hash`**`()`

TODO: add descriptions here

12.2 Working with `dynamic` SNodes

`ti.`**`length`**(*snode*, *indices*)

Parameters

- **snode** – (SNode, dynamic)
- **indices** – (scalar or tuple of scalars) the `dynamic` node indices

Returns (int32) the current size of the dynamic node

`ti.`**append**(*snode*, *indices*, *val*)

Parameters

- **snode** – (SNode, dynamic)
- **indices** – (scalar or tuple of scalars) the `dynamic` node indices
- **val** – (depends on SNode data type) value to store

Returns (int32) the size of the dynamic node, before appending

Inserts `val` into the `dynamic` node with indices `indices`.

12.3 Taichi fields like powers of two

Non-power-of-two field dimensions are promoted into powers of two and thus these fields will occupy more virtual address space. For example, a (dense) field of size `(18, 65)` will be materialized as `(32, 128)`.

12.4 Indices

`ti.`**i**

`ti.`**j**

`ti.`**k**

`ti.`**ij**

`ti.`**ji**

`ti.`**jk**

`ti.`**kj**

`ti.`**ik**

`ti.`**ki**

`ti.`**ijk**

`ti.`**ijkl**

`ti.`**indices**(*a*, *b*, ...)

(TODO)

CHAPTER 13

Metaprogramming

Taichi provides metaprogramming infrastructures. Metaprogramming can

- Unify the development of dimensionality-dependent code, such as 2D/3D physical simulations
- Improve run-time performance by from run-time costs to compile time
- 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.

13.1 Template metaprogramming

You may use `ti.template()` as a type hint to pass a field as an argument. For example:

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

a = ti.field(ti.f32, 4)
b = ti.field(ti.f32, 4)
c = ti.field(ti.f32, 12)
d = ti.field(ti.f32, 12)
copy(a, b)
copy(c, d)
```

As shown in the example above, template programming may enable us to reuse our code and provide more flexibility.

13.2 Dimensionality-independent programming using grouped indices

However, the `copy` template shown above is not perfect. For example, it can only be used to copy 1D fields. What if we want to copy 2D fields? Do we have to write another kernel?

```
@ti.kernel
def copy2d(x: ti.template(), y: ti.template()):
    for i, j in x:
        y[i, j] = x[i, j]
```

Not necessary! Taichi provides `ti.grouped` syntax which enables you to pack loop indices into a grouped vector to unify kernels of different dimensionalities. For example:

```
@ti.kernel
def copy(x: ti.template(), y: ti.template()):
    for I in ti.grouped(y):
        # I is a vector with same dimensionality with x and data type i32
        # If y is 0D, then I = ti.Vector([]), which is equivalent to `None` when used␣
↪in x[I]
        # If y is 1D, then I = ti.Vector([i])
        # If y is 2D, then I = ti.Vector([i, j])
        # If y is 3D, then I = ti.Vector([i, j, k])
        # ...
        x[I] = y[I]

@ti.kernel
def array_op(x: ti.template(), y: ti.template()):
    # if field x is 2D:
    for I in ti.grouped(x): # I is simply a 2D vector with data type i32
        y[I + ti.Vector([0, 1])] = I[0] + I[1]

    # then it is equivalent to:
    for i, j in x:
        y[i, j + 1] = i + j
```

13.3 Field metadata

Sometimes it is useful to get the data type (`field.dtype`) and shape (`field.shape`) of fields. These attributes can be accessed in both Taichi- and Python-scopes.

```
@ti.func
def print_field_info(x: ti.template()):
  print('Field dimensionality is', len(x.shape))
  for i in ti.static(range(len(x.shape))):
    print('Size alone dimension', i, 'is', x.shape[i])
  ti.static_print('Field data type is', x.dtype)
```

See *Scalar fields* for more details.

Note: For sparse fields, the full domain shape will be returned.

13.4 Matrix & vector metadata

Getting the number of matrix columns and rows will allow you to write dimensionality-independent code. For example, this can be used to unify 2D and 3D physical simulators.

`matrix.m` equals to the number of columns of a matrix, while `matrix.n` equals to the number of rows of a matrix. Since vectors are considered as matrices with one column, `vector.n` is simply the dimensionality of the vector.

```
@ti.kernel
def foo():
  matrix = ti.Matrix([[1, 2], [3, 4], [5, 6]])
  print(matrix.n)  # 2
  print(matrix.m)  # 3
  vector = ti.Vector([7, 8, 9])
  print(vector.n)  # 3
  print(vector.m)  # 1
```

13.5 Compile-time evaluations

Using compile-time evaluation will allow certain computations to happen when kernels are being instantiated. This saves the overhead of those computations at runtime.

- Use `ti.static` for compile-time branching (for those who come from C++17, this is if constexpr.):

```
enable_projection = True

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

- Use `ti.static` for forced loop unrolling:

```
@ti.kernel
def func():
  for i in ti.static(range(4)):
      print(i)

  # is equivalent to:
  print(0)
  print(1)
  print(2)
  print(3)
```

13.6 When to use for loops with `ti.static`

There are several reasons why `ti.static` for loops should be used.

- Loop unrolling for performance.
- Loop over vector/matrix elements. Indices into Taichi matrices must be a compile-time constant. Indexing into taichi fields can be run-time variables. For example, if you want to access a vector field `x`, accessed as `x[field_index][vector_component_index]`. The first index can be variable, yet the second must be a constant.

For example, code for resetting this vector fields should be

```
@ti.kernel
def reset():
  for i in x:
    for j in ti.static(range(x.n)):
      # The inner loop must be unrolled since j is a vector index instead
      # of a global field index.
      x[i][j] = 0
```

CHAPTER 14

Advanced dense layouts

Fields (*Scalar fields*) can be *placed* in a specific shape and *layout*. Defining a proper layout can be critical to performance, especially for memory-bound applications. A carefully designed data layout can significantly improve cache/TLB-hit rates and cacheline utilization. Although when performance is not the first priority, you probably don't have to worry about it.

Taichi decouples algorithms from data layouts, and the Taichi compiler automatically optimizes data accesses on a specific data layout. These Taichi features allow programmers to quickly experiment with different data layouts and figure out the most efficient one on a specific task and computer architecture.

In Taichi, the layout is defined in a recursive manner. See *Structural nodes (SNodes)* for more details about how this works. We suggest starting with the default layout specification (simply by specifying `shape` when creating fields using `ti.field/ti.Vector.field/ti.Matrix.field`), and then migrate to more advanced layouts using the `ti.root.X` syntax if necessary.

14.1 From `shape` to `ti.root.X`

For example, this declares a 0-D field:

```
x = ti.field(ti.f32)
ti.root.place(x)
# is equivalent to:
x = ti.field(ti.f32, shape=())
```

This declares a 1D field of size `3`:

```
x = ti.field(ti.f32)
ti.root.dense(ti.i, 3).place(x)
# is equivalent to:
x = ti.field(ti.f32, shape=3)
```

This declares a 2D field of shape `(3, 4)`:

```
x = ti.field(ti.f32)
ti.root.dense(ti.ij, (3, 4)).place(x)
# is equivalent to:
x = ti.field(ti.f32, shape=(3, 4))
```

You may wonder, why not simply specify the `shape` of the field? Why bother using the more complex version? Good question, let go forward and figure out why.

14.2 Row-major versus column-major

Let's start with the simplest layout.

Since address spaces are linear in modern computers, for 1D Taichi fields, the address of the `i`-th element is simply `i`.

To store a multi-dimensional field, however, it has to be flattened, in order to fit into the 1D address space. For example, to store a 2D field of size `(3, 2)`, there are two ways to do this:

1. The address of `(i, j)`-th is `base + i * 2 + j` (row-major).
2. The address of `(i, j)`-th is `base + j * 3 + i` (column-major).

To specify which layout to use in Taichi:

```
ti.root.dense(ti.i, 3).dense(ti.j, 2).place(x)     # row-major (default)
ti.root.dense(ti.j, 2).dense(ti.i, 3).place(y)     # column-major
```

Both `x` and `y` have the same shape of `(3, 2)`, and they can be accessed in the same manner, where `0 <= i < 3 && 0 <= j < 2`. They can be accessed in the same manner: `x[i, j]` and `y[i, j]`. However, they have a very different memory layouts:

```
#     address low ........................... address high
# x:  x[0,0]   x[0,1]   x[0,2] | x[1,0]   x[1,1]   x[1,2]
# y:  y[0,0]   y[1,0] | y[0,1]   y[1,1] | y[0,2]   y[1,2]
```

See? `x` first increases the first index (i.e. row-major), while `y` first increases the second index (i.e. column-major).

Note: For those people from C/C++, here's what they looks like:

```
int x[3][2];  // row-major
int y[2][3];  // column-major

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 2; j++) {
        do_something ( x[i][j] );
        do_something ( y[j][i] );
    }
}
```

14.3 Array of Structures (AoS), Structure of Arrays (SoA)

Fields of same size can be placed together.

For example, this places two 1D fields of size `3` (array of structure, AoS):

```
ti.root.dense(ti.i, 3).place(x, y)
```

Their memory layout:

```
#  address low ............. address high
#  x[0]   y[0] | x[1]  y[1] | x[2]   y[2]
```

In contrast, this places two field placed separately (structure of array, SoA):

```
ti.root.dense(ti.i, 3).place(x)
ti.root.dense(ti.i, 3).place(y)
```

Now, their memory layout:

```
#  address low ............. address high
#  x[0]  x[1]   x[2] | y[0]   y[1]   y[2]
```

Normally, you don't have to worry about the performance nuances between different layouts, and should just define the simplest layout as a start. However, locality sometimes have a significant impact on the performance, especially when the field is huge.

To improve spatial locality of memory accesses (i.e. cache hit rate / cacheline utilization), it's sometimes helpful to place the data elements within relatively close storage locations if they are often accessed together. Take a simple 1D wave equation solver for example:

```
N = 200000
pos = ti.field(ti.f32)
vel = ti.field(ti.f32)
ti.root.dense(ti.i, N).place(pos)
ti.root.dense(ti.i, N).place(vel)

@ti.kernel
def step():
    pos[i] += vel[i] * dt
    vel[i] += -k * pos[i] * dt
```

Here, we placed `pos` and `vel` seperately. So the distance in address space between `pos[i]` and `vel[i]` is `200000`. This will result in a poor spatial locality and lots of cache-misses, which damages the performance. A better placement is to place them together:

```
ti.root.dense(ti.i, N).place(pos, vel)
```

Then `vel[i]` is placed right next to `pos[i]`, this can increase the cache-hit rate and therefore increase the performance.

14.4 Flat layouts versus hierarchical layouts

By default, when allocating a `ti.field`, it follows the simplest data layout.

```
val = ti.field(ti.f32, shape=(32, 64, 128))
# C++ equivalent: float val[32][64][128]
```

However, at times this data layout can be suboptimal for certain types of computer graphics tasks. For example, `val[i, j, k]` and `val[i + 1, j, k]` are very far away (`32 KB`) from each other, and leads to poor access

locality under certain computation tasks. Specifically, in tasks such as texture trilinear interpolation, the two elements are not even within the same `4KB` pages, creating a huge cache/TLB pressure.

A better layout might be

```
val = ti.field(ti.f32)
ti.root.dense(ti.ijk, (8, 16, 32)).dense(ti.ijk, (4, 4, 4)).place(val)
```

This organizes `val` in `4x4x4` blocks, so that with high probability `val[i, j, k]` and its neighbours are close to each other (i.e., in the same cacheline or memory page).

14.5 Struct-fors on advanced dense data layouts

Struct-fors on nested dense data structures will automatically follow their data order in memory. For example, if 2D scalar field `A` is stored in row-major order,

```
for i, j in A:
  A[i, j] += 1
```

will iterate over elements of `A` following row-major order. If `A` is column-major, then the iteration follows the column-major order.

If `A` is hierarchical, it will be iterated level by level. This maximizes the memory bandwidth utilization in most cases.

Struct-for loops on sparse fields follow the same philosophy, and will be discussed further in *Sparse computation (WIP)*.

14.6 Examples

2D matrix, row-major

```
A = ti.field(ti.f32)
ti.root.dense(ti.ij, (256, 256)).place(A)
```

2D matrix, column-major

```
A = ti.field(ti.f32)
ti.root.dense(ti.ji, (256, 256)).place(A) # Note ti.ji instead of ti.ij
```

8x8 blocked 2D array of size *1024x1024*

```
density = ti.field(ti.f32)
ti.root.dense(ti.ij, (128, 128)).dense(ti.ij, (8, 8)).place(density)
```

3D Particle positions and velocities, AoS

```
pos = ti.Vector.field(3, dtype=ti.f32)
vel = ti.Vector.field(3, dtype=ti.f32)
ti.root.dense(ti.i, 1024).place(pos, vel)
# equivalent to
ti.root.dense(ti.i, 1024).place(pos(0), pos(1), pos(2), vel(0), vel(1), vel(2))
```

3D Particle positions and velocities, SoA

```
pos = ti.Vector.field(3, dtype=ti.f32)
vel = ti.Vector.field(3, dtype=ti.f32)
for i in range(3):
  ti.root.dense(ti.i, 1024).place(pos(i))
for i in range(3):
  ti.root.dense(ti.i, 1024).place(vel(i))
```

CHAPTER 15

Sparse computation (WIP)

Warning: The Taichi compiler backend is under migration from source-to-source compilation to LLVM for compilation speed and portability. Sparse computation with the new LLVM backend is not yet fully implemented on multithreaded CPUs and GPUs.

If you are interested in sparse computation in Taichi, please read our paper, watch the introduction video, or check out the SIGGRAPH Asia 2019 slides.

The legacy source-to-source backend (commit `dc162e11`) provides full sparse computation functionality. However, since little engineering has been done to make that commit portable (i.e. easy to compile on different platforms), we suggest waiting until the LLVM version of sparse computation is fully implemented.

Sparse computation functionalities with the new LLVM backend will be back online by the end of December 2019.

CHAPTER 16

Coordinate offsets

- A Taichi field can be defined with **coordinate offsets**. The offsets will move field bounds so that field origins are no longer zero vectors. A typical use case is to support voxels with negative coordinates in physical simulations.
- For example, a matrix of `32x64` elements with coordinate offset `(-16, 8)` can be defined as the following:

```
a = ti.Matrix(2, 2, dtype=ti.f32, shape=(32, 64), offset=(-16, 8))
```

In this way, the field's indices are from `(-16, 8)` to `(16, 72)` (exclusive).

```
a[-16, 32]  # lower left corner
a[16, 32]   # lower right corner
a[-16, 64]  # upper left corner
a[16, 64]   # upper right corner
```

Note: The dimensionality of field shapes should **be consistent** with that of the offset. Otherwise, a `AssertionError` will be raised.

```
a = ti.Matrix.field(2, 3, dtype=ti.f32, shape=(32,), offset=(-16, ))           # Works!
b = ti.Vector.field(3, dtype=ti.f32, shape=(16, 32, 64), offset=(7, 3, -4))    # Works!
c = ti.Matrix.field(2, 1, dtype=ti.f32, shape=None, offset=(32,))              #␣
↪AssertionError
d = ti.Matrix.field(3, 2, dtype=ti.f32, shape=(32, 32), offset=(-16, ))        #␣
↪AssertionError
e = ti.field(dtype=ti.i32, shape=16, offset=-16)                               # Works!
f = ti.field(dtype=ti.i32, shape=None, offset=-16)                             #␣
↪AssertionError
g = ti.field(dtype=ti.i32, shape=(16, 32), offset=-16)                         #␣
↪AssertionError
```

CHAPTER 17

Differentiable programming

We suggest starting with the `ti.Tape()`, and then migrate to more advanced differentiable programming using the `kernel.grad()` syntax if necessary.

17.1 Introduction

For example, you have the following kernel:

```
x = ti.var(ti.f32, ())
y = ti.var(ti.f32, ())

@ti.kernel
def compute_y():
    y[None] = ti.sin(x[None])
```

Now if you want to get the derivative of y corresponding to x, i.e., dy/dx. You may want to implement the derivative kernel by yourself:

```
x = ti.var(ti.f32, ())
y = ti.var(ti.f32, ())
dy_dx = ti.var(ti.f32, ())

@ti.kernel
def compute_dy_dx():
    dy_dx[None] = ti.cos(x[None])
```

But wait, what if I changed the original `compute_y`? We will have to recalculate the derivative by hand and rewrite `compute_dy_dx` again, which is very error-prone and not convenient at all.

If this situation occurs, don't worry! Taichi provides a handy autodiff system that can help you obtain the derivative of a kernel without any pain!

17.2 Using `ti.Tape()`

Let's still take the `compute_y` in above example for explaination. What's the most convienent way to obtain a kernel that computes x to dy/dx?

1. Use the `needs_grad=True` option when declaring fields involved in the derivative chain.
2. Use `with ti.Tape(y):` to embrace the invocation into kernel(s) you want to compute derivative.
3. Now `x.grad[None]` is the dy/dx value at current x.

```
x = ti.var(ti.f32, (), needs_grad=True)
y = ti.var(ti.f32, (), needs_grad=True)

@ti.kernel
def compute_y():
    y[None] = ti.sin(x[None])

with ti.Tape(y):
    compute_y()

print('dy/dx =', x.grad[None])
print('at x =', x[None])
```

It's equivalant to:

```
x = ti.var(ti.f32, ())
y = ti.var(ti.f32, ())
dy_dx = ti.var(ti.f32, ())

@ti.kernel
def compute_dy_dx():
    dy_dx[None] = ti.cos(x[None])

compute_dy_dx()

print('dy/dx =', dy_dx[None])
print('at x =', x[None])
```

17.2.1 Usage example

For a physical simulation, sometimes it could be easy to compute the energy but hard to compute the force on each particles.

But recall that we can differentiate (negative) potential energy to get forces. a.k.a.: `F_i = -dU / dx_i`. So once you've write a kernel that is able to compute the potential energy, you may use Taichi's autodiff system to obtain the derivative of it and then the force on each particles.

Take examples/ad_gravity.py as an example:

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

N = 8
dt = 1e-5

x = ti.Vector.var(2, ti.f32, N, needs_grad=True)  # position of particles
```

(continues on next page)

(continued from previous page)

```
v = ti.Vector.var(2, ti.f32, N)                    # velocity of particles
U = ti.var(ti.f32, (), needs_grad=True)            # potential energy


@ti.kernel
def compute_U():
    for i, j in ti.ndrange(N, N):
        r = x[i] - x[j]
        # r.norm(1e-3) is equivalent to ti.sqrt(r.norm()**2 + 1e-3)
        # This is to prevent 1/0 error which can cause wrong derivative
        U[None] += -1 / r.norm(1e-3)  # U += -1 / |r|


@ti.kernel
def advance():
    for i in x:
        v[i] += dt * -x.grad[i]  # dv/dt = -dU/dx
    for i in x:
        x[i] += dt * v[i]        # dx/dt = v


def substep():
    with ti.Tape(U):
        # every kernel invocation within this indent scope
        # will also be accounted into the partial derivate of U
        # with corresponding input variables like x.
        compute_U()  # will also computes dU/dx and save in x.grad
    advance()


@ti.kernel
def init():
    for i in x:
        x[i] = [ti.random(), ti.random()]


init()
gui = ti.GUI('Autodiff gravity')
while gui.running:
    for i in range(50):
        substep()
    print('U = ', U[None])
    gui.circles(x.to_numpy(), radius=3)
    gui.show()
```

Note: The argument `U` to `ti.Tape(U)` must be a 0D field.

For using autodiff with multiple output variables, please see the `kernel.grad()` usage below.

Note: `ti.Tape(U)` will automatically set *U[None]*' to 0 on start up.

See examples/mpm_lagrangian_forces.py and examples/fem99.py for examples on using autodiff for MPM and FEM.

17.3 Using `kernel.grad()`

TODO: Documentation WIP.

17.4 Kernel Simplicity Rule

Unlike tools such as TensorFlow where **immutable** output buffers are generated, the **imperative** programming paradigm adopted in Taichi allows programmers to freely modify global fields.

To make automatic differentiation well-defined under this setting, we make the following assumption on Taichi programs for differentiable programming:

Global Data Access Rules:

- If a global field element is written more than once, then starting from the second write, the write **must** come in the form of an atomic add ("accumulation", using `ti.atomic_add` or simply `+=`).
- No read accesses happen to a global field element, until its accumulation is done.

Kernel Simplicity Rule: Kernel body consists of multiple *simply nested* for-loops. I.e., each for-loop can either contain exactly one (nested) for-loop (and no other statements), or a group of statements without loops.

Example:

```
@ti.kernel
def differentiable_task():
  for i in x:
    x[i] = y[i]

  for i in range(10):
    for j in range(20):
      for k in range(300):
        ... do whatever you want, as long as there are no loops

  # Not allowed. The outer for loop contains two for loops
  for i in range(10):
    for j in range(20):
      ...
    for j in range(20):
      ...
```

Taichi programs that violate this rule will result in an error.

Note: **static for-loops** (e.g. `for i in ti.static(range(4))`) will get unrolled by the Python frontend preprocessor and therefore does not count as a level of loop.

17.5 DiffTaichi

The DiffTaichi repo contains 10 differentiable physical simulators built with Taichi differentiable programming. A few examples with neural network controllers optimized using differentiable simulators and brute-force gradient descent:

Check out the DiffTaichi paper and video to learn more about Taichi differentiable programming.

CHAPTER 18

Objective data-oriented programming

Taichi is a data-oriented programming (DOP) language. However, simple DOP makes modularization hard.

To allow modularized code, Taichi borrow some concepts from object-oriented programming (OOP).

For convenience, let's call the hybrid scheme **objective data-oriented programming** (ODOP).

TODO: More documentation here.

A brief example:

```
import taichi as ti

ti.init()

@ti.data_oriented
class Array2D:
  def __init__(self, n, m, increment):
    self.n = n
    self.m = m
    self.val = ti.field(ti.f32)
    self.total = ti.field(ti.f32)
    self.increment = increment
    ti.root.dense(ti.ij, (self.n, self.m)).place(self.val)
    ti.root.place(self.total)

  @staticmethod
  @ti.func
  def clamp(x):  # Clamp to [0, 1)
      return max(0, min(1 - 1e-6, x))

  @ti.kernel
  def inc(self):
    for i, j in self.val:
      ti.atomic_add(self.val[i, j], self.increment)

  @ti.kernel
```

(continues on next page)

(continued from previous page)

```
  def inc2(self, increment: ti.i32):
    for i, j in self.val:
      ti.atomic_add(self.val[i, j], increment)

  @ti.kernel
  def reduce(self):
    for i, j in self.val:
      ti.atomic_add(self.total, self.val[i, j] * 4)

arr = Array2D(128, 128, 3)

double_total = ti.field(ti.f32, shape=())

ti.root.lazy_grad()

arr.inc()
arr.inc.grad()
assert arr.val[3, 4] == 3
arr.inc2(4)
assert arr.val[3, 4] == 7

with ti.Tape(loss=arr.total):
  arr.reduce()

for i in range(arr.n):
  for j in range(arr.m):
    assert arr.val.grad[i, j] == 4

@ti.kernel
def double():
  double_total[None] = 2 * arr.total

with ti.Tape(loss=double_total):
  arr.reduce()
  double()

for i in range(arr.n):
  for j in range(arr.m):
    assert arr.val.grad[i, j] == 8
```

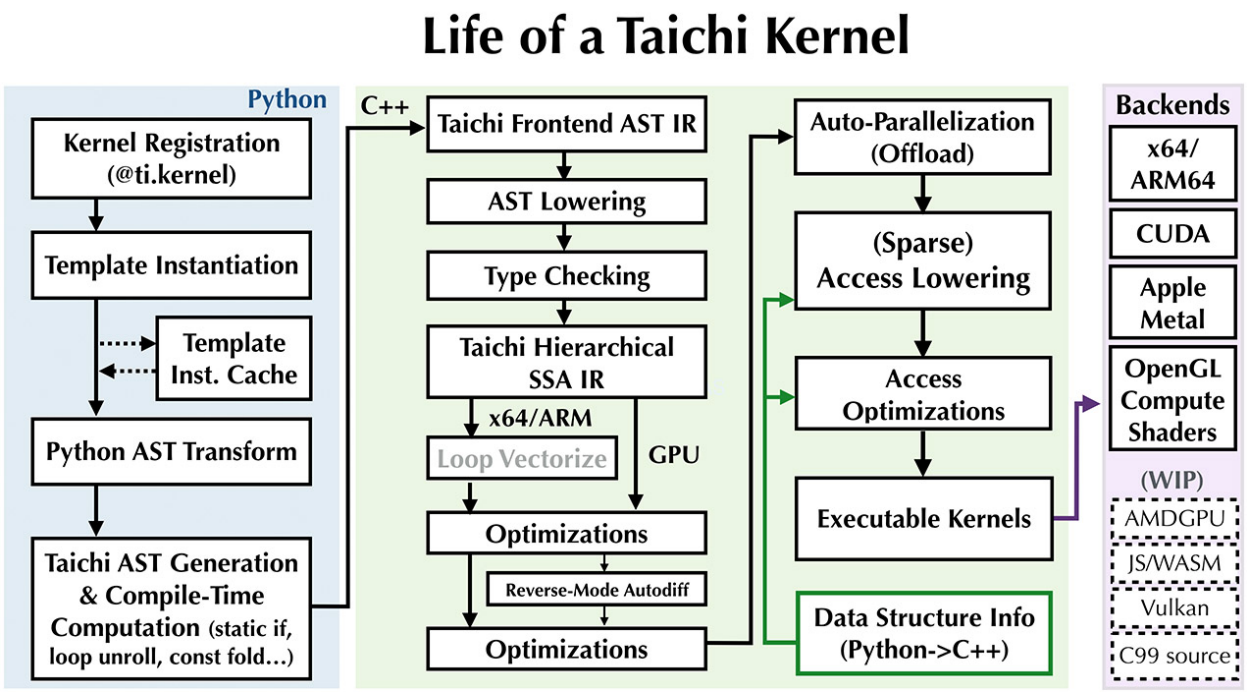
CHAPTER 19

Life of a Taichi kernel

Sometimes it is helpful to understand the life cycle of a Taichi kernel. In short, compilation will only happen on the first invocation of an instance of a kernel.

The life cycle of a Taichi kernel has the following stages:

- Kernel registration
- Template instantiation and caching
- Python AST transforms
- Taichi IR compilation, optimization, and executable generation
- Launching



Let's consider the following simple kernel:

```
@ti.kernel
def add(field: ti.template(), delta: ti.i32):
  for i in field:
    field[i] += delta
```

We allocate two 1D fields to simplify discussion:

```
x = ti.field(dtype=ti.f32, shape=128)
y = ti.field(dtype=ti.f32, shape=16)
```

19.1 Kernel registration

When the `ti.kernel` decorator is executed, a kernel named `add` is registered. Specifically, the Python Abstract Syntax Tree (AST) of the `add` function will be memorized. No compilation will happen until the first invocation of `add`.

19.2 Template instantiation and caching

```
add(x, 42)
```

When `add` is called for the first time, the Taichi frontend compiler will instantiate the kernel.

When you have a second call with the same **template signature** (explained later), e.g.,

```
add(x, 1)
```

Taichi will directly reuse the previously compiled binary.

Arguments hinted with `ti.template()` are template arguments, and will incur template instantiation. For example,

```
add(y, 42)
```

will lead to a new instantiation of **add**.

Note: **Template signatures** are what distinguish different instantiations of a kernel template. The signature of `add(x, 42)` is `(x, ti.i32)`, which is the same as that of `add(x, 1)`. Therefore, the latter can reuse the previously compiled binary. The signature of `add(y, 42)` is `(y, ti.i32)`, a different value from the previous signature, hence a new kernel will be instantiated and compiled.

Note: Many basic operations in the Taichi standard library are implemented using Taichi kernels using metaprogramming tricks. Invoking them will incur **implicit kernel instantiations**.

Examples include `x.to_numpy()` and `y.from_torch(torch_tensor)`. When you invoke these functions, you will see kernel instantiations, as Taichi kernels will be generated to offload the hard work to multiple CPU cores/GPUs.

As mentioned before, the second time you call the same operation, the cached compiled kernel will be reused and no further compilation is needed.

19.3 Code transformation and optimizations

When a new instantiation happens, the Taichi frontend compiler (i.e., the `ASTTransformer` Python class) will transform the kernel body AST into a Python script, which, when executed, emits a Taichi frontend AST. Basically, some patches are applied to the Python AST so that the Taichi frontend can recognize it.

The Taichi AST lowering pass translates Taichi frontend IR into hierarchical static single assignment (SSA) IR, which allows a series of further IR passes to happen, such as

- Loop vectorization
- Type inference and checking
- General simplifications such as common subexpression elimination (CSE), dead instruction elimination (DIE), constant folding, and store forwarding
- Access lowering
- Data access optimizations
- Reverse-mode automatic differentiation (if using differentiable programming)
- Parallelization and offloading
- Atomic operation demotion

19.4 The just-in-time (JIT) compilation engine

Finally, the optimized SSA IR is fed into backend compilers such as LLVM or Apple Metal/OpenGL shader compilers. The backend compilers then generate high-performance executable CPU/GPU programs.

19.5 Kernel launching

Taichi kernels will be ultimately launched as multi-threaded CPU tasks or GPU kernels.

CHAPTER 20

Syntax sugars

20.1 Aliases

Creating aliases for global variables and functions with cumbersome names can sometimes improve readability. In Taichi, this can be done by assigning kernel and function local variables with `ti.static()`, which forces Taichi to use standard python pointer assignment.

For example, consider the simple kernel:

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

The fields and function be aliased to new names with `ti.static`:

```
@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])
```

Aliases can also be created for class members and methods, which can help prevent cluttering objective data-oriented programming code with `self`.

For example, consider class kernel to compute the 2-D laplacian of some field:

```
@ti.kernel
def compute_laplacian(self):
  for i, j in a:
    self.b[i, j] = (self.a[i + 1,j] - 2.0*self.a[i, j] + self.a[i-1, j])/(self.dx**2)␣
↪\
                + (self.a[i,j + 1] - 2.0*self.a[i, j] + self.a[i, j-1])/(self.dy**2)
```

Using `ti.static()`, it can be simplified to:

```
@ti.kernel
def compute_laplacian(self):
  a,b,dx,dy = ti.static(self.a,self.b,self.dx,self.dy)
  for i,j in a:
    b[i,j] = (a[i+1, j] - 2.0*a[i, j] + a[i-1, j])/(dx**2) \
           + (a[i, j+1] - 2.0*a[i, j] + a[i, j-1])/(dy**2)
```

Note: `ti.static` can also be used in combination with `if` (compile-time branching) and `for` (compile-time unrolling). See *Metaprogramming* for more details.

Here, we are using it for *compile-time const values*, i.e. the **field/function handles** are constants at compile time.

CHAPTER 21

Developer installation

Note this is for the compiler developers of the Taichi programming language. End users should use the pip packages instead of building from source. To build with NVIDIA GPU support, CUDA 10.0+ is needed. This installation guide works for Ubuntu 16.04+ and OS X 10.14+. For precise build instructions on Windows, please check out appveyor.yml, which does basically the same thing as the following instructions. We use MSBUILD.exe to build the generated project. Please note that Windows could have multiple instances of MSBUILD.exe shipped with different products. Please make sure you add the path for MSBUILD.exe within your MSVS directory and make it a higher priority (for instance than the one shipped with .NET).

Note that on Linux/OS X, `clang` is the only supported compiler for compiling the Taichi compiler. On Windows only MSVC supported.

21.1 Installing Dependencies

- Make sure you are using Python 3.6/3.7/3.8
- Install Python dependencies:

```
python3 -m pip install --user setuptools astpretty astor pybind11 Pillow dill
python3 -m pip install --user pytest pytest-rerunfailures pytest-xdist yapf
python3 -m pip install --user numpy GitPython coverage colorama autograd
```

- Make sure you have `clang` with version >= 7:
  - On Windows: Download clang-10. Make sure you add the `bin` folder containing `clang.exe` to the `PATH` environment variable.
  - On OS X: you don't need to do anything.
  - On Ubuntu, execute `sudo apt install libtinfo-dev clang-8`.
  - On Arch Linux, execute `sudo pacman -S clang`. (This is `clang-10`).
  - On other Linux distributions, please search this site for clang version >= 7.

- Make sure you have LLVM 10.0.0. Note that Taichi uses a **customized LLVM** so the pre-built binaries from the LLVM official website or other sources probably won't work. Here we provide LLVM binaries customized for Taichi, which may or may not work depending on your system environment:
  - LLVM 10.0.0 for Linux
  - LLVM 10.0.0 for Windows MSVC 2019
  - LLVM 10.0.0 for OS X

Note: On Windows, if you use the pre-built LLVM for Taichi, please add `$LLVM_FOLDER/bin` to `PATH`. Later, when you build Taichi using `CMake`, set `LLVM_DIR` to `$LLVM_FOLDER/lib/cmake/llvm`.

- If the downloaded LLVM does not work, please build from source:
  - On Linux or OS X:

```
wget https://github.com/llvm/llvm-project/releases/download/llvmorg-10.
↪0.0/llvm-10.0.0.src.tar.xz
tar xvJf llvm-10.0.0.src.tar.xz
cd llvm-10.0.0.src
mkdir build
cd build
cmake .. -DLLVM_ENABLE_RTTI:BOOL=ON -DBUILD_SHARED_LIBS:BOOL=OFF -
↪DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD="X86;NVPTX" -DLLVM_
↪ENABLE_ASSERTIONS=ON
# If you are building on NVIDIA Jetson TX2, use -DLLVM_TARGETS_TO_
↪BUILD="ARM;NVPTX"

make -j 8
sudo make install

# Check your LLVM installation
llvm-config --version  # You should get 10.0.0
```

  - On Windows:

```
# LLVM 10.0.0 + MSVC 2019
cmake .. -G"Visual Studio 16 2019" -A x64 -DLLVM_ENABLE_RTTI:BOOL=ON -DBUILD_
↪SHARED_LIBS:BOOL=OFF -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD=
↪"X86;NVPTX" -DLLVM_ENABLE_ASSERTIONS=ON -Thost=x64 -DLLVM_BUILD_
↪TESTS:BOOL=OFF -DCMAKE_INSTALL_PREFIX=installed
```

    * Then open `LLVM.sln` and use Visual Studio 2017+ to build.
    * Please make sure you are using the `Release` configuration. After building the `INSTALL` project (under folder `CMakePredefinedTargets` in the Solution Explorer window).
    * If you use MSVC 2019, **make sure you use C++17** for the `INSTALL` project.
    * After the build is complete, find your LLVM binaries and headers in `build/installed`.

    Please add `build/installed/bin` to `PATH`. Later, when you build Taichi using `CMake`, set `LLVM_DIR` to `build/installed/lib/cmake/llvm`.

21.2 Setting up CUDA (optional)

If you don't have CUDA, go to this website and download the installer.

- To check if CUDA is installed, run `nvcc --version` or `cat /usr/local/cuda/version.txt`.
- On **Ubuntu** we recommend choosing `deb (local)` as **Installer Type**.
- On **Arch Linux**, you can easily install CUDA via `pacman -S cuda` without downloading the installer manually.

21.3 Setting up Taichi for development

- Set up environment variables for Taichi:
  - On Linux / OS X, please add the following script to your rc file (`~/.bashrc`, `~/.zshrc` or etc. , same for other occurrences in this documentation):

```
export TAICHI_REPO_DIR=/path/to/taichi  # Path to your taichi repository
export PYTHONPATH=$TAICHI_REPO_DIR/python:$PYTHONPATH
export PATH=$TAICHI_REPO_DIR/bin:$PATH
# export CXX=/path/to/clang  # Uncomment if you encounter issue about␣
↪compiler in the next step.
# export PATH=/opt/llvm/bin:$PATH  # Uncomment if your llvm or clang is␣
↪installed in /opt
```

    Then execute `source ~/.bashrc` to reload shell config.
  - On Windows, please add these variables by accessing your system settings:
    1. Add `TAICHI_REPO_DIR` whose value is the path to your taichi repository so that Taichi knows you're a developer.
    2. Add or append `PYTHONPATH` with `%TAICHI_REPO_DIR%/python` so that Python imports Taichi from the local repo.
    3. Add or append `PATH` with `%TAICHI_REPO_DIR%/bin` so that you can use `ti` command.
    4. Add or append `PATH` with path to LLVM binary directory installed in previous section.
- Clone the taichi repo **recursively**, and build:

```
git clone https://github.com/taichi-dev/taichi --depth=1 --branch=master
cd taichi
git submodule update --init --recursive --depth=1
mkdir build
cd build
cmake ..
# On Linux / OS X, if you do not set clang as the default compiler
# use the line below:
#   cmake .. -DCMAKE_CXX_COMPILER=clang
#
# Alternatively, if you would like to set clang as the default compiler
# On Unix CMake honors environment variables $CC and $CXX upon deciding which C␣
↪and C++ compilers to use
make -j 8
```

- Check out `examples` for runnable examples. Run them with commands like `python3 examples/mpm128.py`.
- Execute `python3 -m taichi test` to run all the tests. It may take up to 5 minutes to run all tests.

21.4 Troubleshooting Developer Installation

- If `make` fails to compile and reports `fatal error: 'spdlog/XXX.h' file not found`, please try runing `git submodule update --init --recursive --depth=1`.
- If importing Taichi causes

```
FileNotFoundError: [Errno 2] No such file or directory: '/root/taichi/python/
↪taichi/core/../lib/taichi_core.so' -> '/root/taichi/python/taichi/core/../lib/
↪libtaichi_core.so'``
```

  Please try adding `TAICHI_REPO_DIR` to environment variables, see *Setting up Taichi for development*.

- If the build succeeded but running any Taichi code results in errors like `Bitcode file (/tmp/taichi-tero94pl/runtime//runtime_x64.bc) not found`, please double check `clang` is in your `PATH`:

```
clang --version
# version should be >= 7
```

  and our **Taichi configured** `llvm-as`:

```
llvm-as --version
# version should be >= 8
which llvm-as
# should be /usr/local/bin/llvm-as or /opt/XXX/bin/llvm-as, which is our␣
↪configured installation
```

  If not, please install `clang` and **build LLVM from source** with instructions above in *Developer installation*, then add their path to environment variable `PATH`.

- If you encounter other issues, feel free to report by opening an issue on GitHub. We are willing to help!
- See also *Troubleshooting* for issues that may share with end-user installation.

21.5 Docker

For those who prefer to use Docker, we also provide a Dockerfile which helps setup the Taichi development environment with CUDA support based on Ubuntu docker image.

Note: In order to follow the instructions in this section, please make sure you have the Docker DeskTop (or Engine for Linux) installed and set up properly.

21.5.1 Build the Docker Image

From within the root directory of the taichi Git repository, execute `docker build -t taichi:latest .` to build a Docker image based off the local master branch tagged with *latest*. Since this builds the image from source, please expect up to 40 mins build time if you don't have cached Docker image layers.

Note: In order to save the time on building Docker images, you could always visit our Docker Hub repository and pull the versions of pre-built images you would like to use. Currently the builds are triggered per taichi Github release.

For example, to pull a image built from release v0.6.17, run `docker pull taichidev/taichi:v0.6.17`

21.5.2 Use Docker Image on macOS (cpu only)

1. Make sure `XQuartz` and `socat` are installed:

```
brew cask install xquartz
brew install socat
```

2. Temporally disable the xhost access-control: `xhost +`
3. Start the Docker container with `docker run -it -e DISPLAY=$(ipconfig getifaddr en0):0 taichidev/taichi:v0.6.17`
4. Do whatever you want within the container, e.g. you could run tests or an example, try: `ti test` or `ti example mpm88`
5. Exit from the container with `exit` or `ctrl+D`
6. [To keep your xhost safe] Re-enable the xhost access-control: `xhost -`

21.5.3 Use Docker Image on Ubuntu (with CUDA support)

1. Make sure your host machine has CUDA properly installed and configured. Usually you could verify it by running `nvidia-smi`
2. Make sure ‘ NVIDIA Container Toolkit <https://github.com/NVIDIA/nvidia-docker>‘_ is properly installed:

```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list |␣
↪sudo tee /etc/apt/sources.list.d/nvidia-docker.list

sudo apt-get update && sudo apt-get install -y nvidia-container-toolkit
sudo systemctl restart docker
```

3. Make sure `xorg` is installed: `sudo apt-get install xorg`
4. Temporally disable the xhost access-control: `xhost +`
5. Start the Docker container with `sudo docker run -it --gpus all -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix taichidev/taichi:v0.6.17`
6. Do whatever you want within the container, e.g. you could run tests or an example, try: `ti test` or `ti example mpm88`
7. Exit from the container with `exit` or `ctrl+D`
8. [To keep your xhost safe] Re-enable the xhost access-control: `xhost -`

Warning: The nature of Docker container determines that no changes to the file system on the container could be preserved once you exit from the container. If you want to use Docker as a persistent development environment, we recommend you mount the taichi Git repository to the container as a volume and set the Python path to the mounted directory.

CHAPTER 22

Contribution guidelines

First of all, thank you for contributing! We welcome contributions of all forms, including but not limited to

- Bug fixes
- Proposing and implementing new features
- Documentation improvement and translations (e.g. Simplified Chinese)
- Improved error messages that are more user-friendly
- New test cases
- New examples
- Compiler performance patches
- Blog posts and tutorials on Taichi
- Participation in the Taichi forum
- Introduce Taichi to your friends or simply star the project.
- Typo fixes in the documentation, code or comments (please directly make a pull request for minor issues like these)

22.1 How to contribute bug fixes and new features

Issues marked with "good first issue" are great chances for starters.

- Please first leave a note (e.g. *I know how to fix this and would like to help!*) on the issue, so that people know someone is already working on it. This helps prevent redundant work;
- If no core developer has commented and described a potential solution on the issue, please briefly describe your plan, and wait for a core developer to reply before you start. This helps keep implementations simple and effective.

Issues marked with "welcome contribution" are slightly more challenging but still friendly to beginners.

22.2 High-level guidelines

- Be pragmatic: practically solving problems is our ultimate goal.
- No overkills: always use *easy* solutions to solve easy problems, so that you have time and energy for real hard ones.
- Almost every design decision has pros and cons. A decision is *good* if its pros outweigh its cons. Always think about both sides.
- Debugging is hard. Changesets should be small so that sources of bugs can be easily pinpointed.
- Unit/integration tests are our friends.

Note: "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. *The first method is far more difficult*." — C.A.R. Hoare

---

One thing to keep in mind is that, Taichi was originally born as an academic research project. This usually means that some parts did not have the luxury to go through a solid design. While we are always trying to improve the code quality, it doesn't mean that the project is free from technical debts. Some places may be confusing or overly complicated. Whenever you spot one, you are more than welcome to shoot us a PR! :-)

22.3 Effective communication

- How much information we effectively convey, is way more important than how many words we typed.
- Be constructive. Be polite. Be organized. Be concise.
- Bulleted lists are our friends.
- Proofread before you post: if you are the reader, can you understand what you typed?
- If you are not a native speaker, consider using a spell checker such as Grammarly.

Please base your discussion and feedback on facts, and not personal feelings. It is very important for all of us to maintain a friendly and blame-free community. Some examples:

Tip: (Acceptable) This design could be confusing to new Taichi users.

Warning: (Not Acceptable) This design is terrible.

22.4 Making good pull requests

- PRs with **small** changesets are preferred. A PR should ideally address **only one issue**.
  - It is fine to include off-topic **trivial** refactoring such as typo fixes;
  - The reviewers reserve the right to ask PR authors to remove off-topic **non-trivial** changes.
- All commits in a PR will always be **squashed and merged into master as a single commit**.

- PR authors **should not squash commits on their own**;
- When implementing a complex feature, consider breaking it down into small PRs, to keep a more detailed development history and to interact with core developers more frequently.
- If you want early feedback from core developers
  - Open a PR in Draft state on GitHub so that you can share your progress;
  - Make sure you @ the corresponding developer in the comments or request the review.
- If you are making multiple PRs
  - Independent PRs should be based on **different** branches forking from `master`;
  - PRs with dependencies should be raised only after all prerequisite PRs are merged into `master`.
- All PRs should ideally come with corresponding **tests**;
- All PRs should come with **documentation update**, except for internal compiler implementations;
- All PRs must pass **continuous integration tests** before they get merged;
- PR titles should follow *PR title format and tags*;
- A great article from Google on how to have your PR merged quickly. [PDF]

22.5 Reviewing & PR merging

- Please try to follow these tips from Google
  - Code Health: Understanding Code In Review; [PDF]
  - Code Health: Respectful Reviews == Useful Reviews. [PDF]
- The merger should always **squash and merge** PRs into the master branch;
- The master branch is required to have a **linear history**;
- Make sure the PR passes **continuous integration tests**, except for cases like documentation updates;
- Make sure the title follows *PR title format and tags*.

22.6 Using continuous integration

- Continuous Integration (CI), will **build** and **test** your commits in a PR against in environments.
- Currently, Taichi uses Travis CI (for OS X and Linux) and AppVeyor (for Windows).
- CI will be triggered every time you push commits to an open PR.
- You can prepend `[skip ci]` to your commit message to avoid triggering CI. e.g. `[skip ci] This commit will not trigger CI`
- A tick on the right of commit hash means CI passed, a cross means CI failed.

22.7 Enforcing code style

- Locally, you can run `ti format` in the command line to re-format code style. Note that you have to install `clang-format-6.0` and `yapf v0.29.0` locally before you use `ti format`.
- If you don't have to install these formatting tools locally, use the **format server**. It's an online version of `ti format`.
  - Go to http://kun.csail.mit.edu:31415/, and click at the desired PR id.
  - Come back to the PR page, you'll see a user called @taichi-gardener (bot) pushed a commit named `[skip ci] enforce code format`.
  - You won't see the bot's commit if it didn't find anything not matching the format.
  - Then please run `git pull` in your local branch to pull the formatted code.
  - Note that commit messages marked with `[format]` will automatically trigger the format server. e.g. `[format] your commit message`

22.8 PR title format and tags

PR titles will be part of the commit history reflected in the `master` branch, therefore it is important to keep PR titles readable.

- Please always prepend **at least one tag** such as `[Lang]` to PR titles:
  - When using multiple tags, make sure there is exactly one space between tags;
  - E.g., "[Lang][refactor]" (no space) should be replaced by "[Lang] [refactor]";
- The first letter of the PR title body should be capitalized:
  - E.g., `[Doc] improve documentation` should be replaced by `[Doc] Improve documentation`;
  - `[Lang] "ti.sqr(x)" is now deprecated` is fine because `"` is a symbol.
- Please do not include back quotes ("``") in PR titles.
- For example, "[Metal] Support bitmasked SNode", "[OpenGL] AtomicMin/Max support", or "[Opt] [IR] Enhanced constant folding".

Frequently used tags:

- `[Metal], [OpenGL], [CPU], [CUDA]`: backends;
- `[LLVM]`: the LLVM backend shared by CPUs and CUDA;
- `[Lang]`: frontend language features, including syntax sugars;
- `[Std]`: standard library, e.g. `ti.Matrix` and `ti.Vector`;
- `[Sparse]`: sparse computation;
- `[IR]`: intermediate representation;
- `[Opt]`: IR optimization passes;
- `[GUI]`: the built-in GUI system;
- `[Refactor]`: code refactoring;
- `[CLI]`: commandline interfaces, e.g. the `ti` command;

- `[Doc]`: documentation under `docs/`;
- `[Example]`: examples under `examples/`;
- `[Test]`: adding or improving tests under `tests/`;
- `[Linux]`: Linux platform;
- `[Mac]`: Mac OS X platform;
- `[Windows]`: Windows platform;
- `[Perf]`: performance improvements;
- `[Misc]`: something that doesn't belong to any category, such as version bump, reformatting;
- `[Bug]`: bug fixes;
- Check out more tags in misc/prtags.json.
- When introducing a new tag, please update the list in `misc/prtags.json` in the first PR with that tag, so that people can follow.

Note: We do appreciate all kinds of contributions, yet we should not expose the title of every PR to end-users. Therefore the changelog will distinguish *what the user should know* from *what the developers are doing*. This is done by **capitalizing PR tags**:

- PRs with visible/notable features to the users should be marked with tags starting with **the first letter capitalized**, e.g. `[Metal]`, `[OpenGL]`, `[IR]`, `[Lang]`, `[CLI]`. When releasing a new version, a script (`python/taichi/make_changelog.py`) will generate a changelog with these changes (PR title) highlighted. Therefore it is **important** to make sure the end-users can understand what your PR does, **based on your PR title**.
- Other PRs (underlying development/intermediate implementation) should use tags with **everything in lowercase letters**: e.g. `[metal]`, `[opengl]`, `[ir]`, `[lang]`, `[cli]`.
- Because of the way the release changelog is generated, there should be **at most one captialized tag** in a PR title to prevent duplicate PR highlights. For example, `[GUI] [Mac] Support modifier keys` (#1189) is a bad example, we should use `[gui] [Mac] Support modifier keys in GUI` instead. Please capitalize the tag that is most relevant to the PR.

22.9 C++ and Python standards

The C++ part of Taichi is written in C++17, and the Python part in 3.6+. You can assume that C++17 and Python 3.6 features are always available.

22.10 Tips on the Taichi compiler development

Life of a Taichi kernel may worth checking out. It explains the whole compilation process.

See also *Benchmarking and regression tests* if your work involves IR optimization.

When creating a Taichi program using `ti.init(arch=desired_arch, **kwargs)`, pass in the following parameters to make the Taichi compiler print out IR:

- `print_preprocessed = True`: print results of the frontend Python AST transform. The resulting scripts will generate a Taichi Frontend AST when executed.

- `print_ir = True`: print the Taichi IR transformation process of kernel (excluding accessors) compilation.
- `print_accessor_ir = True`: print the IR transformation process of data accessors, which are special and simple kernels. (This is rarely used, unless you are debugging the compilation of data accessors.)
- `print_struct_llvm_ir = True`: save the emitted LLVM IR by Taichi struct compilers.
- `print_kernel_llvm_ir = True`: save the emitted LLVM IR by Taichi kernel compilers.
- `print_kernel_llvm_ir_optimized = True`: save the optimized LLVM IR of each kernel.
- `print_kernel_nvptx = True`: save the emitted NVPTX of each kernel (CUDA only).

Note: Data accessors in Python-scope are implemented as special Taichi kernels. For example, `x[1, 2, 3] = 3` will call the writing accessor kernel of `x`, and `print(y[42])` will call the reading accessor kernel of `y`.

22.11 Folder structure

Key folders are

- `taichi`: The core compiler implementation
  - `program`: Top-level constructs
  - `ir`: Intermediate representation
  - `analysis`: Static analysis passes
  - `transforms`: IR transform passes
  - `inc`: Small definition files to be included repeatedly
  - `jit`: Just-In-Time compilation base classes
  - `llvm`: LLVM utilities
  - `runtime`: LLVM runtime environments
  - `struct`: Struct compiler base classes
  - `codegen`: Code generation base classes
  - `backends`: Device-dependent code generators/runtime environments
    - `cpu`: CPU backend implementation
    - `cuda`: CUDA backend implementation
    - `opengl`: OpenGL backend implementation
    - `metal`: Metal backend implementation
    - `cc`: C backend implementation (WIP)
  - `gui`: GUI system
  - `math`: Math utilities
  - `python`: C++/Python interfaces
  - `platform`: Platform supports
  - `system`: OS-related infrastructure
  - `util`: Miscellaneous utilities

- `python/taichi`: Python frontend implementation
  - `core`: Loading & interacting with Taichi core
  - `lang`: Python-embbed Taichi language & syntax (major)
  - `misc`: Miscellaneous utilities
  - `tools`: Handy end-user tools
- `tests`: Functional tests
  - `python`: Python tests (major)
  - `cpp`: C++ tests
- `examples`: Examples
- `docs`: Documentation
- `benchmarks`: Performance benchmarks
- `external`: External libraries
- `misc`: Random (yet useful) files
- ...

22.12 Testing

Tests should be added to `tests/`.

22.12.1 Command line tools

- Use `ti test` to run all the tests.
- Use `ti test -v` for verbose outputs.
- Use `ti test -C` to run tests and record code coverage, see *Code coverage* for more infomations.
- Use `ti test -a <arch(s)>` for testing against specified backend(s). e.g. `ti test -a cuda,metal`.
- Use `ti test -na <arch(s)>` for testing all architectures excluding some of them. e.g. `ti test -na opengl,x64`.
- Use `ti test <filename(s)>` to run specific tests in filenames. e.g. `ti test numpy_io` will run all tests in `tests/python/test_numpy_io.py`.
- Use `ti test -c` to run only the C++ tests. e.g. `ti test -c alg_simp` will run `tests/cpp/test_alg_simp.cpp`.
- Use `ti test -k <key>` to run tests that match the specified key. e.g. `ti test linalg -k "cross or diag"` will run the `test_cross` and `test_diag` in `tests/python/test_linalg.py`.

For more options, see `ti test -h`.

For more details on how to write a test case, see write_test.

22.13 Documentation

Documentations are put under the folder `docs/`.

- We use reStructured text (.rst) to write documentation.
- We host our documentation online using readthedocs.io.
- Use `ti doc` to build the documentation locally.
- Open the documentation at `docs/build/index.html`.

Note: On Linux/OS X, use `watch -n 1 ti doc` to continuously build the documentation.

If the OpenGL backend detector keeps creating new windows, execute `export TI_WITH_OPENGL=0` for `ti doc`.

22.14 Efficient code navigation across Python/C++

If you work on the language frontend (Python/C++ interface), to navigate around the code base, ffi-navigator allows you to jump from Python bindings to their definitions in C++. Follow their README to set up your editor.

22.15 Upgrading CUDA

Right now we are targeting CUDA 10. When upgrading CUDA version, the file `external/cuda_libdevice/slim_libdevice.10.bc` should also be replaced with a newer version.

To generate the slimmed version of libdevice based on a full `libdevice.X.bc` file from a CUDA installation, use `ti task make_slim_libdevice [libdevice.X.bc file]`

CHAPTER 23

Workflow for writing a Python test

Normally we write functional tests in Python.

- We use pytest for our Python test infrastructure.
- Python tests should be added to `tests/python/test_xxx.py`.

For example, you've just added a utility function `ti.log10`. Now you want to write a **test**, to test if it functions properly.

23.1 Adding a new test case

Look into `tests/python`, see if there's already a file suit for your test. If not, feel free to create a new file for it :) So in this case let's create a new file `tests/python/test_logarithm.py` for simplicity.

Add a function, the function name **must** be started with `test_` so that `pytest` could find it. e.g:

```
import taichi as ti

def test_log10():
    pass
```

Add some simple code make use of our `ti.log10` to make sure it works well. Hint: You may pass/return values to/from Taichi-scope using 0-D fields, i.e. `r[None]`.

```
import taichi as ti

def test_log10():
    ti.init(arch=ti.cpu)

    r = ti.var(ti.f32, ())

    @ti.kernel
    def foo():
```

(continues on next page)

(continued from previous page)

```
        r[None] = ti.log10(r[None])

    r[None] = 100
    foo()
    assert r[None] == 2
```

Execute `ti test logarithm`, and the functions starting with `test_` in `tests/python/test_logarithm.py` will be executed.

23.2 Testing against multiple backends

The above method is not good enough, for example, `ti.init(arch=ti.cpu)`, means that it will only test on the CPU backend. So do we have to write many tests `test_log10_cpu`, `test_log10_cuda`, ... with only the first line different? No worries, we provide a useful decorator `@ti.test`:

```
import taichi as ti

# will test against both CPU and CUDA backends
@ti.test(ti.cpu, ti.cuda)
def test_log10():
    r = ti.var(ti.f32, ())

    @ti.kernel
    def foo():
        r[None] = ti.log10(r[None])

    r[None] = 100
    foo()
    assert r[None] == 2
```

And you may test against **all backends** by simply not specifying the argument:

```
import taichi as ti

# will test against all backends available on your end
@ti.test()
def test_log10():
    r = ti.var(ti.f32, ())

    @ti.kernel
    def foo():
        r[None] = ti.log10(r[None])

    r[None] = 100
    foo()
    assert r[None] == 2
```

Cool! Right? But that's still not good enough.

23.3 Using `ti.approx` for comparison with tolerance

Sometimes the math percison could be poor on some backends like OpenGL, e.g. `ti.log10(100)` may return `2.001` or `1.999` in this case.

To cope with this behavior, we provide `ti.approx` which can tolerate such errors on different backends, for example `2.001 == ti.approx(2)` will return `True` on the OpenGL backend.

```
import taichi as ti

# will test against all backends available on your end
@ti.test()
def test_log10():
    r = ti.var(ti.f32, ())

    @ti.kernel
    def foo():
        r[None] = ti.log10(r[None])

    r[None] = 100
    foo()
    assert r[None] == ti.approx(2)
```

Warning: Simply using `pytest.approx` won't work well here, since it's tolerance won't vary among different Taichi backends. It'll be likely to fail on the OpenGL backend.

`ti.approx` also do treatments on boolean types, e.g.: `2 == ti.approx(True)`.

Great on improving stability! But the test is still not good enough, yet.

23.4 Parametrize test inputs

For example, `r[None] = 100`, means that it will only test the case of `ti.log10(100)`. What if `ti.log10(10)`? `ti.log10(1)`?

We may test against different input values using the `@pytest.mark.parametrize` decorator:

```
import taichi as ti
import pytest
import math

@pytest.mark.parametrize('x', [1, 10, 100])
@ti.test()
def test_log10(x):
    r = ti.var(ti.f32, ())

    @ti.kernel
    def foo():
        r[None] = ti.log10(r[None])

    r[None] = x
    foo()
    assert r[None] == math.log10(x)
```

Use a comma-separated list for multiple input values:

```
import taichi as ti
import pytest
import math
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.parametrize('x,y', [(1, 2), (1, 3), (2, 1)])
@ti.test()
def test_atan2(x, y):
    r = ti.var(ti.f32, ())
    s = ti.var(ti.f32, ())

    @ti.kernel
    def foo():
        r[None] = ti.atan2(r[None])

    r[None] = x
    s[None] = y
    foo()
    assert r[None] == math.atan2(x, y)
```

Use two separate `parametrize` to test **all combinations** of input arguments:

```
import taichi as ti
import pytest
import math

@pytest.mark.parametrize('x', [1, 2])
@pytest.mark.parametrize('y', [1, 2])
# same as:  .parametrize('x,y', [(1, 1), (1, 2), (2, 1), (2, 2)])
@ti.test()
def test_atan2(x, y):
    r = ti.var(ti.f32, ())
    s = ti.var(ti.f32, ())

    @ti.kernel
    def foo():
        r[None] = ti.atan2(r[None])

    r[None] = x
    s[None] = y
    foo()
    assert r[None] == math.atan2(x, y)
```

23.5 Specifying `ti.init` configurations

You may specify keyword arguments to `ti.init()` in `ti.test()`, e.g.:

```
@ti.test(ti.cpu, debug=True, log_level=ti.TRACE)
def test_debugging_utils():
    # ... (some tests have to be done in debug mode)
```

is the same as:

```
def test_debugging_utils():
    ti.init(arch=ti.cpu, debug=True, log_level=ti.TRACE)
    # ... (some tests have to be done in debug mode)
```

23.6 Exclude some backends from test

Sometimes some backends are not capable of specific tests, we have to exclude them from test:

```
# Run this test on all backends except for OpenGL
@ti.test(excludes=[ti.opengl])
def test_sparse_field():
    # ... (some tests that requires sparse feature which is not supported by OpenGL)
```

You may also use the `extensions` keyword to exclude backends without specific feature:

```
# Run this test on all backends except for OpenGL
@ti.test(extensions=[ti.extension.sparse])
def test_sparse_field():
    # ... (some tests that requires sparse feature which is not supported by OpenGL)
```

CHAPTER 24

Developer utilities

This section provides a detailed description of some commonly used utilities for Taichi developers.

24.1 Logging

Taichi uses spdlog as its logging system. Logs can have different levels, from low to high, they are:

```
trace
debug
info
warn
error
```

The higher the level is, the more critical the message is.

The default logging level is `info`. You may override the default logging level by:

1. Setting the environment variable like `export TI_LOG_LEVEL=warn`.
2. Setting the log level from Python side: `ti.set_logging_level(ti.WARN)`.

In Python, you may write logs using the `ti.*` interface:

```
# Python
ti.trace("Hello world!")
ti.debug("Hello world!")
ti.info("Hello world!")
ti.warn("Hello world!")
ti.error("Hello world!")
```

In C++, you may write logs using the `TI_*` interface:

```
// C++
TI_TRACE("Hello world!");
```

(continues on next page)

(continued from previous page)

```
TI_DEBUG("Hello world!");
TI_INFO("Hello world!");
TI_WARN("Hello world!");
TI_ERROR("Hello world!");
```

If one raises a message of the level `error`, Taichi will be **terminated** immediately and result in a `RuntimeError` on Python side.

```
int func(void *p) {
  if (p == nullptr)
    TI_ERROR("The pointer cannot be null!");

  // will not reach here if p == nullptr
  do_something(p);
}
```

Note: For people from Linux kernels, `TI_ERROR` is just `panic`.

You may also simplify the above code by using `TI_ASSERT`:

```
int func(void *p) {
  TI_ASSERT_INFO(p != nullptr, "The pointer cannot be null!");
  // or
  // TI_ASSERT(p != nullptr);

  // will not reach here if p == nullptr
  do_something(p);
}
```

24.2 Benchmarking and regression tests

- Run `ti benchmark` to run tests in benchmark mode. This will record the performance of `ti test`, and save it in `benchmarks/output`.
- Run `ti regression` to show the difference between the previous result in `benchmarks/baseline`. And you can see if the performance is increasing or decreasing after your commits. This is really helpful when your work is related to IR optimizations.
- Run `ti baseline` to save the benchmark result to `benchmarks/baseline` for future comparison, this may be executed on performance-related PRs, before they are merged into master.

For example, this is part of the output by `ti regression` after enabling constant folding optimization pass:

```
linalg____________________polar_decomp_____________________________
codegen_offloaded_tasks                    37 ->    39    +5.4%
codegen_statements                       3179 ->  3162    -0.5%
codegen_kernel_statements                2819 ->  2788    -1.1%
codegen_evaluator_statements                0 ->    14    +inf%

linalg____________________init_matrix_from_vectors___________________
codegen_offloaded_tasks                    37 ->    39    +5.4%
codegen_statements                       3180 ->  3163    -0.5%
```

(continues on next page)

(continued from previous page)

```
codegen_kernel_statements                  2820 ->  2789    -1.1%
codegen_evaluator_statements                  0 ->    14    +inf%
```

Note: Currently `ti benchmark` only supports benchmarking number-of-statements, no time benchmarking is included since it depends on hardware performance and therefore hard to compare if the baseline is from another machine. We are to purchase a fixed-performance machine as a time benchmark server at some point. Discussion at: https://github.com/taichi-dev/taichi/issue/948

The suggested workflow for the performance-related PR author to run the regression tests is:

- Run `ti benchmark && ti baseline` in `master` to save the current performance as a baseline.
- Run `git checkout -b your-branch-name`.
- Do works on the issue, stage 1.
- Run `ti benchmark && ti regression` to obtain the result.
- (If result BAD) Do further improvements, until the result is satisfying.
- (If result OK) Run `ti baseline` to save stage 1 performance as a baseline.
- Go forward to stage 2, 3, . . . , and the same workflow is applied.

24.3 (Linux only) Trigger `gdb` when programs crash

```
# Python
ti.set_gdb_trigger(True)

// C++
CoreState::set_trigger_gdb_when_crash(true);

# Shell
export TI_GDB_TRIGGER=1
```

Note: **Quickly pinpointing segmentation faults/assertion failures using** `gdb`: When Taichi crashes, `gdb` will be triggered and attach to the current thread. You might be prompt to enter sudo password required for gdb thread attaching. After entering `gdb`, check the stack backtrace with command `bt` (`backtrace`), then find the line of code triggering the error.

24.4 Code coverage

To ensure that our tests covered every situation, we need to have **coverage report**. That is, to detect how many percents of code lines in is executed in test.

- Generally, the higher the coverage percentage is, the stronger our tests are.
- When making a PR, we want to **ensure that it comes with corresponding tests**. Or code coverage will decrease.
- Code coverage statuses are visible at Codecov.
- Currently, Taichi is only set up for Python code coverage report, not for C++ yet.

```
ti test -C       # run tests and save results to .coverage
coverage report  # generate a coverage report on terminal output
coverage html    # generate a HTML form report in htmlcov/index.html
```

24.5 Interface system (legacy)

Print all interfaces and units

```
ti.core.print_all_units()
```

24.6 Serialization (legacy)

The serialization module of taichi allows you to serialize/deserialize objects into/from binary strings.

You can use `TI_IO` macros to explicitly define fields necessary in Taichi.

```
// TI_IO_DEF
struct Particle {
    Vector3f position, velocity;
    real mass;
    string name;

    TI_IO_DEF(position, velocity, mass, name);
}

// TI_IO_DECL
struct Particle {
    Vector3f position, velocity;
    real mass;
    bool has_name
    string name;

    TI_IO_DECL() {
        TI_IO(position);
        TI_IO(velocity);
        TI_IO(mass);
        TI_IO(has_name);
        // More flexibility:
        if (has_name) {
            TI_IO(name);
        }
    }
}

// TI_IO_DEF_VIRT();
```

24.7 Progress notification (legacy)

The Taichi messenger can send an email to `$TI_MONITOR_EMAIL` when the task finishes or crashes. To enable:

```
from taichi.tools import messenger
messenger.enable(task_id='test')
```

CHAPTER 25

Profiler

Taichi's profiler can help you analyze the run-time cost of your program. There are two profiling systems in Taichi: `ScopedProfiler` and `KernelProfiler`.

25.1 ScopedProfiler

1. `ScopedProfiler` measures time spent on the **host tasks** hierarchically.
2. This profiler is automatically on. To show its results, call `ti.print_profile_info()`. For example:

```
import taichi as ti

ti.init(arch=ti.cpu)
var = ti.field(ti.f32, shape=1)


@ti.kernel
def compute():
    var[0] = 1.0
    print("Setting var[0] =", var[0])


compute()
ti.print_profile_info()
```

`ti.print_profile_info()` prints profiling results in a hierarchical format.

Note: `ScopedProfiler` is a C++ class in the core of Taichi. It is not exposed to Python users.

25.2 KernelProfiler

1. `KernelProfiler` records the costs of Taichi kernels on devices. To enable this profiler, set `kernel_profiler=True` in `ti.init`.
2. Call `ti.kernel_profiler_print()` to show the kernel profiling result. For example:

```
import taichi as ti

ti.init(ti.cpu, kernel_profiler=True)
var = ti.field(ti.f32, shape=1)


@ti.kernel
def compute():
    var[0] = 1.0


compute()
ti.kernel_profiler_print()
```

The outputs would be:

```
[ 22.73%] jit_evaluator_0_kernel_0_serial                min   0.001 ms   avg   0.001 ms␣
↪  max   0.001 ms   total   0.000 s [      1x]
[  0.00%] jit_evaluator_1_kernel_1_serial                min   0.000 ms   avg   0.000 ms␣
↪  max   0.000 ms   total   0.000 s [      1x]
[ 77.27%] compute_c4_0_kernel_2_serial                   min   0.004 ms   avg   0.004 ms␣
↪  max   0.004 ms   total   0.000 s [      1x]
```

CHAPTER 26

C++ style

We generally follow Google C++ Style Guide.

26.1 Naming

- Variable names should consist of lowercase words connected by underscores, e.g. `llvm_context`.
- Class and struct names should consist of words with first letters capitalized, e.g. `CodegenLLVM`.
- Macros should be capital start with `TI`, such as `TI_INFO`, `TI_IMPLEMENTATION`.
  - We do not encourage the use of macro, although there are cases where macros are inevitable.
- Filenames should consist of lowercase words connected by underscores, e.g. `ir_printer.cpp`.

26.2 Dos

- Use `auto` for local variables when appropriate.
- Mark `override` and `const` when necessary.

26.3 Don'ts

- C language legacies:
  - `printf` (Use `fmtlib::print` instead).
  - `new` and `free`. (Use smart pointers `std::unique_ptr, std::shared_ptr` instead for ownership management).
  - `#include <math.h>` (Use `#include <cmath>` instead).
- Exceptions (We are on our way to **remove** all C++ exception usages in Taichi).

- Prefix member functions with `m_` or `_`.
- Virtual function call in constructors/destructors.
- `NULL` (Use `nullptr` instead).
- `using namespace std;` in the global scope.
- `typedef` (Use `using` instead).

26.4 Automatic code formatting

- Please run `ti format`

CHAPTER 27

Internal designs (WIP)

27.1 Intermediate representation

Use `ti.init(print_ir=True)` to print IR on the console.

27.2 List generation (WIP)

Struct-fors in Taichi loop over all active elements of a (sparse) data structure **in parallel**. Evenly distributing work onto processor cores is challenging on sparse data structures: naively splitting an irregular tree into pieces can easily lead to partitions with drastically different numbers of leaf elements.

Our strategy is to generate lists of active SNode elements layer by layer. The list generation computation happens on the same device as normal computation kernels, depending on the `arch` argument when the user calls `ti.init`.

List generations flatten the data structure leaf elements into a 1D dense array, circumventing the irregularity of incomplete trees. Then we can simply invoke a regular **parallel for** over the list.

For example,

```
# misc/listgen_demo.py

import taichi as ti

ti.init(print_ir=True)

x = ti.field(ti.i32)
ti.root.dense(ti.i, 4).bitmasked(ti.i, 4).place(x)

@ti.kernel
def func():
    for i in x:
        print(i)
```

(continues on next page)

(continued from previous page)

```
func()
```

gives you the following IR:

```
$0 = offloaded clear_list S1dense
$1 = offloaded listgen S0root->S1dense
$2 = offloaded clear_list S2bitmasked
$3 = offloaded listgen S1dense->S2bitmasked
$4 = offloaded struct_for(S2bitmasked) block_dim=0 {
  <i32 x1> $5 = loop index 0
  print i, $5
}
```

Note that `func` leads to two list generations:

- (Tasks `$0` and `$1`) based on the list of `root` node (`S0`), generate the list of the `dense` nodes (`S1`);
- (Tasks `$2` and `$3`) based on the list of `dense` nodes (`S1`), generate the list of `bitmasked` nodes (`S2`).

The list of `root` node always has exactly one element (instance), so we never clear or re-generate this list.

Note: The list of `place` (leaf) nodes (e.g., `S3` in this example) is never generated. Instead, we simply loop over the list of their parent nodes, and for each parent node we enumerate the `place` nodes on-the-fly (without actually generating a list).

The motivation for this design is to amortize list generation overhead. Generating one list element per leaf node (`place` SNode) element is too expensive, likely much more expensive than the essential computation happening on the leaf element. Therefore we only generate their parent element list, so that the list generation cost is amortized over multiple child elements of a second-to-last-level SNode element.

In the example above, although we have `16` instances of `x`, we only generate a list of `4 bitmasked` nodes (and `1 dense` node).

27.3 Code generation

27.4 Statistics

In some cases, it is helpful to gather certain quantitative information about internal events during Taichi program execution. The `Statistics` class is designed for this purpose.

Usage:

```
#include "taichi/util/statistics.h"

// add 1.0 to counter "codegen_offloaded_tasks"
taichi::stat.add("codegen_offloaded_tasks");

// add the number of statements in "ir" to counter "codegen_statements"
taichi::stat.add("codegen_statements", irpass::analysis::count_statements(this->ir));
```

Note the keys are `std::string` and values are `double`.

To print out all statistics in Python:

```
ti.core.print_stat()
```

27.5 Why Python frontend

Embedding Taichi in `python` has the following advantages:

- Easy to learn. Taichi has a very similar syntax to Python.
- Easy to run. No ahead-of-time compilation is needed.
- This design allows people to reuse existing python infrastructure:
  - IDEs. A python IDE mostly works for Taichi with syntax highlighting, syntax checking, and autocomplete.
  - Package manager (pip). A developed Taichi application and be easily submitted to `PyPI` and others can easily set it up with `pip`.
  - Existing packages. Interacting with other python components (e.g. `matplotlib` and `numpy`) is just trivial.
- The built-in AST manipulation tools in `python` allow us to do magical things, as long as the kernel body can be parsed by the Python parser.

However, this design has drawbacks as well:

- Taichi kernels must parse-able by Python parsers. This means Taichi syntax cannot go beyond Python syntax.
  - For example, indexing is always needed when accessing elements in Taichi fields, even if the fields is 0D. Use `x[None] = 123` to set the value in `x` if `x` is 0D. This is because `x = 123` will set `x` itself (instead of its containing value) to be the constant `123` in python syntax, and, unfortunately, we cannot modify this behavior.
- Python has relatively low performance. This can cause a performance issue when initializing large Taichi fields with pure python scripts. A Taichi kernel should be used to initialize a huge fields.

27.6 Virtual indices v.s. physical indices

In Taichi, *virtual indices* are used to locate elements in fields, and *physical indices* are used to specify data layouts in memory.

For example,

- In `a[i, j, k]`, `i`, `j`, and `k` are **virtual** indices.
- In `for i, j in x:`, `i` and `j` are **virtual** indices.
- `ti.i, ti.j, ti.k, ti.l, ...` are **physical** indices.
- In struct-for statements, `LoopIndexStmt::index` is a **physical** index.

The mapping between virtual indices and physical indices for each `SNode` is stored in `SNode::physical_index_position`. I.e., `physical_index_position[i]` answers the question: **which physical index does the i-th virtual index** correspond to?

Each `SNode` can have a different virtual-to-physical mapping. `physical_index_position[i] == -1` means the `i`-th virtual index does not corrspond to any physical index in this `SNode`.

`SNode` s in handy dense fields (i.e., `a = ti.field(ti.i32, shape=(128, 256, 512))`) have **trivial** virtual-to-physical mapping, e.g. `physical_index_position[i] = i`.

However, more complex data layouts, such as column-major 2D fields can lead to `SNodes` with `physical_index_position[0] = 1` and `physical_index_position[1] = 0`.

```
a = ti.field(ti.f32, shape=(128, 32, 8))

b = ti.field(ti.f32)
ti.root.dense(ti.j, 32).dense(ti.i, 16).place(b)

ti.get_runtime().materialize()

mapping_a = a.snode().physical_index_position()

assert mapping_a == {0: 0, 1: 1, 2: 2}

mapping_b = b.snode().physical_index_position()

assert mapping_b == {0: 1, 1: 0}
# Note that b is column-major:
# the virtual first index exposed to the user comes second in memory layout.
```

Taichi supports up to 8 (`constexpr int taichi_max_num_indices = 8`) virtual indices and physical indices.

CHAPTER 28

TaichiCon

We are hosting a series of online **TaichiCon** events for developers to gather and share their Taichi experiences.

28.1 Past Conferences

Everything on previous TaichiCons is available in the TaichiCon repository.

28.2 Format

Each TaichiCon consists of two parts: **talks** and **free discussions**.

- The first 1-hour consists of four 10-minute talks, each with 5-minute Q&A;
- After the formal talks, attendees are free to chat about Taichi in any aspects.

The conference format may evolve in the future.

28.3 Language

The `i`-th TaichiCon will be hosted in

- English `(if i % 2 == 1)`;
- Chinese (Mandarin) `(if i % 2 == 0)`.

28.4 Time and frequency

Taichi developers are scattered around the world, so it's important to pick a good time so that people in different time zones can attend.

A good time for people in Asia and the U.S.:

- (China, Beijing) Sunday 10:00 - 11:00
- (Japan, Tokyo) Sunday 11:00 - 12:00
- (New Zealand, Wellington) Sunday 14:00 - 15:00
- (U.S., East coast) Saturday 22:00 - 23:00
- (U.S., West coast) Saturday 19:00 - 20:00

For people in Europe, we may need to find a better time. There is probably no perfect solution, so it may make sense to pick different times for TaichiCons to provide equal opportunities to people around the world.

TaichiCon will be hosted roughly once per month.

28.5 Attending TaichiCon

Everyone interested in Taichi or related topics (computer graphics, compilers, high-performance computing, computational fluid dynamics, etc.) is welcome to participate!

The Zoom meeting room has a capacity of 300 participants. A few tips:

- It's recommended to change your Zoom display name to a uniform `name (company)` format. For example, `Yuanming Hu (MIT CSAIL)`;
- Please keep muted during the talks to avoid background noises;
- If you have questions, feel free to raise them in the chat window;
- Video recordings and slides will be uploaded after the conference;
- Usually people do not open their cameras during TaichiCon to save network bandwidth for people in unsatisfactory network conditions.

28.6 Preparing for a talk at TaichiCon

We welcome any topics about Taichi, including but not limited to

- API proposals (e.g., "I think the `Matrix` class needs to be refactored. Here are my thoughts . . . ")
- Applications (e.g., "I wrote a new fluid solver using Taichi and would like to share . . . ")
- Integrations (e.g., "Here's how I integrated Taichi into Blender . . . ")
- Internal designs (e.g., "How a new backend is implemented")
- . . .

The body of a TaichiCon talk should be 10 min. After each talk, there are 5 minutes for Q & A. The host may stop the talk if a speaker's talk lasts beyond 15 minutes.

28.7 Organizing TaichiCon

Before the conference:

- Pick a nice time for the event;

- Invite speakers: ask for a talk **title**, **abstract**, and a brief **speaker introduction**;
- Advertise the event on social networks (Facebook, Twitter, Zhihu etc.);
- Make sure all speakers are already in the (virtual) conference room 10 minutes before the conference begins;
  - If a speaker does not show up, try to remind him to attend via email.

Hosting the conference:

- Make sure the Zoom session is being recorded;
- Remember to welcome everyone to attend :-)
- Before each talk, introduce the speaker;
- In the end, thank all the speakers and attendees.

After the conference:

- Upload the video to Youtube and Bilibili;
- Collect speakers' slides in PDF format;
  - Make a representative image with screenshots of the first slides of all the talks;
- Update the TaichiCon repository following the format of TaichiCon 0.
- Update this documentation page if you find any opportunities to improve the workflow of TaichiCon.

CHAPTER 29

Versioning and releases

29.1 Pre-1.0 versioning

Taichi follows Semantic Versioning 2.0.0.

Since Taichi is still under version 1.0.0, we use minor version bumps (e.g., `0.6.17->0.7.0`) for breaking API changes, and patch version bumps (e.g., `0.6.9->0.6.10`) for backward-compatible changes.

29.2 Workflow: releasing a new version

- Trigger a Linux build on Jenkins to see if CUDA passes all tests. Note that Jenkins is the only build bot we have that tests CUDA. (This may take half an hour.)
- Create a branch for the release PR, forking from the latest commit of the `master` branch.
  - Update Taichi version number at the beginning of `CMakeLists.txt`. For example, change `SET(TI_VERSION_PATCH 9)` to `SET(TI_VERSION_PATCH 10)`.
  - Rerun cmake so that `docs/version` gets updated.
  - commit with message "[release] vX.Y.Z", e.g. "[release] v0.6.10".
  - You should see two changes in this commit: one line in `CMakeLists.txt` and one line in `docs/version`.
  - Execute `ti changelog` and save its outputs. You will need this later.
- Open a PR titled "[release] vX.Y.Z" with the branch and commit you just now created.
  - Use the `ti changelog` output you saved in the previous step as the content of the PR description.
  - Wait for all the checks and build bots to complete. (This step may take up to two hours).
- Squash and merge the PR.
- Trigger the Linux build on Jenkins, again, so that Linux packages are uploaded to PyPI.

- Wait for all build bots to finish. This step uploads PyPI packages for OS X and Windows. You may have to wait for up to two hours.
- Update the `stable` branch so that the head of that branch is your release commit on `master`.
- Draft a new release (here):
  - The title should be "vX.Y.Z".
  - The tag should be "vX.Y.Z".
  - Target should be "recent commit" -> the release commit.
  - The release description should be copy-pasted from the release PR description.
  - Click the "Publish release" button.

29.3 Release cycle

Taichi releases new versions twice per week:

- The first release happens on Wednesdays.
- The second release happens on Saturdays.

Additional releases may happen if anything needs an urgent fix.

CHAPTER 30

Frequently asked questions

Q: Installing Taichi with `pip`, complains `package not found`.

A: Is your Python version >= 3.6, and 64-bit? See *Troubleshooting*.

Q: Do we have something like `ti.pi`?

A: No, but you may use `math.pi` or `numpy.pi` instead. Taichi is able to bake in these constants during JIT, so your kernels incur no runtime cost.

Q: How do I **force** an outermost loop to be serial, i.e. **not parallelized**?

A: Try this trick:

```
for _ in range(1):  # I'm the outer-most loop!
    for i in range(100):  # This loop will not be parallelized
        ...
```

Q: What's the most convenient way to load images / textures into Taichi fields?

A: Simply use `field.from_numpy(ti.imread('filename.png'))`.

Q: Can Taichi co-operate with **other Python packages** like `matplotlib`?

A: Yes, as long as that *package* provides an interface with `numpy`, see *Interacting with other Python packages*.

Q: Shall we add some handy functions like `ti.smoothstep` or `ti.vec3`?

A: No, but we provide them in an extension library Taichi GLSL , install it using:

```
python -m pip install taichi_glsl
```

Q: How can I **render 3D results** without writing a ray tracer myself?

A: You may export it with *Export PLY files* so that you could view it in Houdini or Blender. Or make use the extension library Taichi THREE to render images and update to GUI in real-time.

Q: How do I declare a field with **dynamic length**?

A: What you want may be the `dynamic` SNode, a kind of sparse field, see *Working with dynamic SNodes*. Or simply allocate a dense field large enough, and another 0-D field `field_len[None]` for length record. But in fact, the `dynamic` SNode could be slower than the latter solution, due to the cost of maintaining the sparsity information.

Q: Can a user iterate over irregular topologies (e.g., graphs or tetrahedral meshes) instead of regular grids?

A: These structures have to be represented using 1D arrays in Taichi. You can still iterate over them using `for i in x` or `for` However, at compile time, there's little the Taichi compiler can do for you to optimize it. You can still tweak the data layout to get different runtime cache behaviors and performance numbers.

CHAPTER 31

GUI system

Taichi has a built-in GUI system to help users visualize results.

31.1 Create a window

`ti.GUI`(*title = 'Taichi'*, *res = (512, 512)*, *background_color = 0x000000*)

Parameters

- **`title`** – (optional, string) the window title
- **`res`** – (optional, scalar or tuple) resolution / size of the window
- **`background_color`** – (optional, RGB hex) background color of the window

Returns (GUI) an object represents the window

Create a window. If `res` is scalar, then width will be equal to height.

The following code creates a window of resolution `640x360`:

```
gui = ti.GUI('Window Title', (640, 360))
```

`gui.show`(*filename = None*)

Parameters

- **`gui`** – (GUI) the window object
- **`filename`** – (optional, string) see notes below

Show the window on the screen.

Note: If `filename` is specified, a screenshot will be saved to the file specified by the name. For example, the following saves frames of the window to `.png`'s:

```
for frame in range(10000):
    render(img)
    gui.set_image(img)
    gui.show(f'{frame:06d}.png')
```

31.2 Paint on a window

`gui.`**`set_image`**`(`*img*`)`

Parameters

- **gui** – (GUI) the window object
- **img** – (np.array or ti.field) field containing the image, see notes below

Set an image to display on the window.

The image pixels are set from the values of `img[i, j]`, where `i` indicates the horizontal coordinates (from left to right) and `j` the vertical coordinates (from bottom to top).

If the window size is `(x, y)`, then `img` must be one of:

- `ti.field(shape=(x, y))`, a grey-scale image
- `ti.field(shape=(x, y, 3))`, where *3* is for `(r, g, b)` channels
- `ti.field(shape=(x, y, 2))`, where *2* is for `(r, g)` channels
- `ti.Vector.field(3, shape=(x, y))` `(r, g, b)` channels on each component (see *Vectors*)
- `ti.Vector.field(2, shape=(x, y))` `(r, g)` channels on each component
- `np.ndarray(shape=(x, y))`
- `np.ndarray(shape=(x, y, 3))`
- `np.ndarray(shape=(x, y, 2))`

The data type of `img` must be one of:

- `uint8`, range `[0, 255]`
- `uint16`, range `[0, 65535]`
- `uint32`, range `[0, 4294967295]`
- `float32`, range `[0, 1]`
- `float64`, range `[0, 1]`

Note: When using `float32` or `float64` as the data type, `img` entries will be clipped into range `[0, 1]` for display.

`gui.`**`get_image`**`()`

Returns (np.array) the current image shown on the GUI

Get the 4-channel (RGBA) image shown in the current GUI system.

`gui.`**`circle`**`(`*pos*, *color = 0xFFFFFF*, *radius = 1*`)`

Parameters

- **gui** – (GUI) the window object
- **pos** – (tuple of 2) the position of the circle
- **color** – (optional, RGB hex) the color to fill the circle
- **radius** – (optional, scalar) the radius of the circle

Draw a solid circle.

`gui.`**`circles`**(*pos*, *color = 0xFFFFFF*, *radius = 1*)

Parameters

- **gui** – (GUI) the window object
- **pos** – (np.array) the positions of the circles
- **color** – (optional, RGB hex or np.array of uint32) the color(s) to fill the circles
- **radius** – (optional, scalar or np.array of float32) the radius (radii) of the circles

Draw solid circles.

Note: If `color` is a numpy array, the circle at `pos[i]` will be colored with `color[i]`. In this case, `color` must have the same size as `pos`.

`gui.`**`line`**(*begin*, *end*, *color = 0xFFFFFF*, *radius = 1*)

Parameters

- **gui** – (GUI) the window object
- **begin** – (tuple of 2) the first end point position of line
- **end** – (tuple of 2) the second end point position of line
- **color** – (optional, RGB hex) the color of line
- **radius** – (optional, scalar) the width of line

Draw a line.

`gui.`**`lines`**(*begin*, *end*, *color = 0xFFFFFF*, *radius = 1*)

Parameters

- **gui** – (GUI) the window object
- **begin** – (np.array) the positions of the first end point of lines
- **end** – (np.array) the positions of the second end point of lines
- **color** – (optional, RGB hex or np.array of uint32) the color(s) of lines
- **radius** – (optional, scalar or np.array of float32) the width(s) of the lines

Draw lines.

`gui.`**`triangle`**(*a*, *b*, *c*, *color = 0xFFFFFF*)

Parameters

- **gui** – (GUI) the window object
- **a** – (tuple of 2) the first end point position of triangle

- **b** – (tuple of 2) the second end point position of triangle
- **c** – (tuple of 2) the third end point position of triangle
- **color** – (optional, RGB hex) the color to fill the triangle

Draw a solid triangle.

gui.**triangles**(*a*, *b*, *c*, *color = 0xFFFFFF*)

Parameters

- **gui** – (GUI) the window object
- **a** – (np.array) the positions of the first end point of triangles
- **b** – (np.array) the positions of the second end point of triangles
- **c** – (np.array) the positions of the third end point of triangles
- **color** – (optional, RGB hex or np.array of uint32) the color(s) to fill the triangles

Draw solid triangles.

gui.**rect**(*topleft*, *bottomright*, *radius = 1*, *color = 0xFFFFFF*)

Parameters

- **gui** – (GUI) the window object
- **topleft** – (tuple of 2) the top-left point position of rectangle
- **bottomright** – (tuple of 2) the bottom-right point position of rectangle
- **color** – (optional, RGB hex) the color of stroke line
- **radius** – (optional, scalar) the width of stroke line

Draw a hollow rectangle.

gui.**text**(*content*, *pos*, *font_size = 15*, *color = 0xFFFFFF*)

Parameters

- **gui** – (GUI) the window object
- **content** – (str) the text to draw
- **pos** – (tuple of 2) the top-left point position of the fonts / texts
- **font_size** – (optional, scalar) the size of font (in height)
- **color** – (optional, RGB hex) the foreground color of text

Draw a line of text on screen.

ti.rgb_to_hex(rgb):

Parameters **rgb** – (tuple of 3 floats) The (R, G, B) float values, in range [0, 1]

Returns (RGB hex or np.array of uint32) The converted hex value

Convert a (R, G, B) tuple of floats into a single integer value. E.g.,

```
rgb = (0.4, 0.8, 1.0)
hex = ti.rgb_to_hex(rgb)  # 0x66ccff

rgb = np.array([[0.4, 0.8, 1.0], [0.0, 0.5, 1.0]])
hex = ti.rgb_to_hex(rgb)  # np.array([0x66ccff, 0x007fff])
```

The return values can be used in GUI drawing APIs.

31.3 Event processing

Every event have a key and type.

Event type is the type of event, for now, there are just three type of event:

```
ti.GUI.RELEASE  # key up or mouse button up
ti.GUI.PRESS    # key down or mouse button down
ti.GUI.MOTION   # mouse motion or mouse wheel
```

Event key is the key that you pressed on keyboard or mouse, can be one of:

```
# for ti.GUI.PRESS and ti.GUI.RELEASE event:
ti.GUI.ESCAPE  # Esc
ti.GUI.SHIFT   # Shift
ti.GUI.LEFT    # Left Arrow
'a'            # we use lowercase for alphabet
'b'
...
ti.GUI.LMB     # Left Mouse Button
ti.GUI.RMB     # Right Mouse Button

# for ti.GUI.MOTION event:
ti.GUI.MOVE    # Mouse Moved
ti.GUI.WHEEL   # Mouse Wheel Scrolling
```

A *event filter* is a list combined of *key*, *type* and *(type, key)* tuple, e.g.:

```
# if ESC pressed or released:
gui.get_event(ti.GUI.ESCAPE)

# if any key is pressed:
gui.get_event(ti.GUI.PRESS)

# if ESC pressed or SPACE released:
gui.get_event((ti.GUI.PRESS, ti.GUI.ESCAPE), (ti.GUI.RELEASE, ti.GUI.SPACE))
```

`gui.`**`running`**

Parameters **`gui`** – (GUI)

Returns (bool) `True` if `ti.GUI.EXIT` event occurred, vice versa

`ti.GUI.EXIT` occurs when you click on the close (X) button of a window. So `gui.running` will obtain `False` when the GUI is being closed.

For example, loop until the close button is clicked:

```
while gui.running:
    render()
    gui.set_image(pixels)
    gui.show()
```

You can also close the window by manually setting `gui.running` to `False`:

```
while gui.running:
    if gui.get_event(ti.GUI.ESCAPE):
        gui.running = False

    render()
    gui.set_image(pixels)
    gui.show()
```

gui.**get_event**(*a*, ...)

Parameters

- **gui** – (GUI)
- **a** – (optional, EventFilter) filter out matched events

Returns (bool) `False` if there is no pending event, vise versa

Try to pop a event from the queue, and store it in `gui.event`.

For example:

```
if gui.get_event():
    print('Got event, key =', gui.event.key)
```

For example, loop until ESC is pressed:

```
gui = ti.GUI('Title', (640, 480))
while not gui.get_event(ti.GUI.ESCAPE):
    gui.set_image(img)
    gui.show()
```

gui.**get_events**(*a*, ...)

Parameters

- **gui** – (GUI)
- **a** – (optional, EventFilter) filter out matched events

Returns (generator) a python generator, see below

Basically the same as `gui.get_event`, except for this one returns a generator of events instead of storing into `gui.event`:

```
for e in gui.get_events():
    if e.key == ti.GUI.ESCAPE:
        exit()
    elif e.key == ti.GUI.SPACE:
        do_something()
    elif e.key in ['a', ti.GUI.LEFT]:
        ...
```

gui.**is_pressed**(*key*, ...)

Parameters

- **gui** – (GUI)
- **key** – (EventKey) keys you want to detect

Returns (bool) `True` if one of the keys pressed, vice versa

Warning: Must be used together with `gui.get_event`, or it won't be updated! For example:

```
while True:
    gui.get_event()  # must be called before is_pressed
    if gui.is_pressed('a', ti.GUI.LEFT):
        print('Go left!')
    elif gui.is_pressed('d', ti.GUI.RIGHT):
        print('Go right!')
```

`gui.`**`get_cursor_pos`**`()`

Parameters **`gui`** – (GUI)

Returns (tuple of 2) current cursor position within the window

For example:

```
mouse_x, mouse_y = gui.get_cursor_pos()
```

`gui.`**`fps_limit`**

Parameters **`gui`** – (GUI)

Returns (scalar or None) the maximum FPS, `None` for no limit

The default value is 60.

For example, to restrict FPS to be below 24, simply `gui.fps_limit = 24`. This helps reduce the overload on your hardware especially when you're using OpenGL on your intergrated GPU which could make desktop slow to response.

31.4 GUI Widgets

Sometimes it's more intuitive to use widgets like slider, button to control program variables instead of chaotic keyboard bindings. Taichi GUI provides a set of widgets that hopefully could make variable control more intuitive:

`gui.`**`slider`**`(`*text*, *minimum*, *maximum*, *step=1*`)`

Parameters

- **`text`** – (str) the text to be displayed above this slider.
- **`minumum`** – (float) the minimum value of the slider value.
- **`maxumum`** – (float) the maximum value of the slider value.
- **`step`** – (optional, float) the step between two separate value.

Returns (WidgetValue) a value getter / setter, see *`WidgetValue`*.

The widget will be display as: `{text}: {value:.3f}`, followed with a slider.

`gui.`**`label`**`(`*text*`)`

Parameters **`text`** – (str) the text to be displayed in the label.

Returns (WidgetValue) a value getter / setter, see *`WidgetValue`*.

The widget will be display as: `{text}: {value:.3f}`.

`gui.`**`button`**`(`*text*, *event_name=None*`)`

Parameters

- **text** – (str) the text to be displayed in the button.
- **event_name** – (optional, str) customize the event name.

Returns (EventKey) the event key for this button, see *Event processing*.

class WidgetValue

A getter / setter for widget values.

value

Get / set the current value in the widget where we're returned from.

For example:

```
radius = gui.slider('Radius', 1, 50)

while gui.running:
    print('The radius now is', radius.value)
    ...
    radius.value += 0.01
    ...
    gui.show()
```

31.5 Image I/O

ti.**imwrite**(*img*, *filename*)

Parameters

- **img** – (ti.Vector.field or ti.field) the image you want to export
- **filename** – (string) the location you want to save to

Export a `np.ndarray` or Taichi field (`ti.Matrix.field`, `ti.Vector.field`, or `ti.field`) to a specified location `filename`.

Same as `ti.GUI.show(filename)`, the format of the exported image is determined by **the suffix of** `filename` as well. Now `ti.imwrite` supports exporting images to `png`, `img` and `jpg` and we recommend using `png`.

Please make sure that the input image has **a valid shape**. If you want to export a grayscale image, the input shape of field should be `(height, weight)` or `(height, weight, 1)`. For example:

```
import taichi as ti

ti.init()

shape = (512, 512)
type = ti.u8
pixels = ti.field(dtype=type, shape=shape)

@ti.kernel
def draw():
    for i, j in pixels:
        pixels[i, j] = ti.random() * 255    # integars between [0, 255] for ti.u8

draw()
```

(continues on next page)

(continued from previous page)

```
ti.imwrite(pixels, f"export_u8.png")
```

Besides, for RGB or RGBA images, `ti.imwrite` needs to receive a field which has shape `(height, width, 3)` and `(height, width, 4)` individually.

Generally the value of the pixels on each channel of a `png` image is an integar in [0, 255]. For this reason, `ti.imwrite` will **cast fields** which has different datatypes all **into integars between [0, 255]**. As a result, `ti.imwrite` has the following requirements for different datatypes of input fields:

- For float-type (`ti.f16`, `ti.f32`, etc) input fields, **the value of each pixel should be float between [0.0, 1.0]**. Otherwise `ti.imwrite` will first clip them into [0.0, 1.0]. Then they are multiplied by 256 and casted to integaters ranging from [0, 255].
- For int-type (`ti.u8`, `ti.u16`, etc) input fields, **the value of each pixel can be any valid integer in its own bounds**. These integers in this field will be scaled to [0, 255] by being divided over the upper bound of its basic type accordingly.

Here is another example:

```
import taichi as ti

ti.init()

shape = (512, 512)
channels = 3
type = ti.f32
pixels = ti.Matrix.field(channels, dtype=type, shape=shape)

@ti.kernel
def draw():
    for i, j in pixels:
        for k in ti.static(range(channels)):
            pixels[i, j][k] = ti.random()   # floats between [0, 1] for ti.f32

draw()

ti.imwrite(pixels, f"export_f32.png")
```

`ti.`**`imread`**(*filename*, *channels=0*)

Parameters

- **`filename`** – (string) the filename of the image to load
- **`channels`** – (optional int) the number of channels in your specified image. The default value `0` means the channels of the returned image is adaptive to the image file

Returns (np.ndarray) the image read from `filename`

This function loads an image from the target filename and returns it as a `np.ndarray(dtype=np.uint8)`.

Each value in this returned field is an integer in [0, 255].

`ti.`**`imshow`**(*img*, *windname*)

Parameters

- **`img`** – (ti.Vector.field or ti.field) the image to show in the GUI
- **`windname`** – (string) the name of the GUI window

This function will create an instance of `ti.GUI` and show the input image on the screen.

It has the same logic as `ti.imwrite` for different datatypes.

CHAPTER 32

Debugging

Debugging a parallel program is not easy, so Taichi provides builtin utilities that could hopefully help you debug your Taichi program.

32.1 Run-time `print` in kernels

print (*arg1*, ..., *sep=' '*, *end='n'*)

Debug your program with `print()` in Taichi-scope. For example:

```
@ti.kernel
def inside_taichi_scope():
    x = 233
    print('hello', x)
    #=> hello 233

    print('hello', x * 2 + 200)
    #=> hello 666

    print('hello', x, sep='')
    #=> hello233

    print('hello', x, sep='', end='')
    print('world', x, sep='')
    #=> hello233world233

    m = ti.Matrix([[2, 3, 4], [5, 6, 7]])
    print('m =', m)
    #=> m = [[2, 3, 4], [5, 6, 7]]

    v = ti.Vector([3, 4])
    print('v =', v)
    #=> v = [3, 4]
```

For now, Taichi-scope `print` supports string, scalar, vector, and matrix expressions as arguments. `print` in Taichi-scope may be a little different from `print` in Python-scope. Please see details below.

Warning: For the **CPU and CUDA backend**, `print` will not work in Graphical Python Shells including IDLE and Jupyter notebook. This is because these backends print the outputs to the console instead of the GUI. Use the **OpenGL or Metal backend** if you wish to use `print` in IDLE / Jupyter.

Warning: For the **CUDA backend**, the printed result will not show up until `ti.sync()` is called:

```
import taichi as ti
ti.init(arch=ti.cuda)

@ti.kernel
def kern():
    print('inside kernel')

print('before kernel')
kern()
print('after kernel')
ti.sync()
print('after sync')
```

obtains:

```
before kernel
after kernel
inside kernel
after sync
```

Note that host access or program end will also implicitly invoke `ti.sync()`.

Note: Note that `print` in Taichi-scope can only receive **comma-separated parameter**. Neither f-string nor formatted string should be used. For example:

```
import taichi as ti
ti.init(arch=ti.cpu)
a = ti.field(ti.f32, 4)


@ti.kernel
def foo():
    a[0] = 1.0
    print('a[0] = ', a[0]) # right
    print(f'a[0] = {a[0]}') # wrong, f-string is not supported
    print("a[0] = %f" % a[0]) # wrong, formatted string is not supported

foo()
```

32.2 Compile-time `ti.static_print`

Sometimes it is useful to print Python-scope objects and constants like data types or SNodes in Taichi-scope. So, similar to `ti.static` we provide `ti.static_print` to print compile-time constants. It is similar to Python-scope `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
    for i in range(4):
            ti.static_print(i.dtype)
            # => DataType.int32
            # will only print once
```

Unlike `print`, `ti.static_print` will only print the expression once at compile-time, and therefore it has no runtime cost.

32.3 Runtime `assert` in kernel

Programmers may use `assert` statements in Taichi-scope. When the assertion condition failed, a `RuntimeError` will be raised to indicate the error.

To make `assert` work, first make sure you are using the **CPU backend**. For performance reason, `assert` only works when `debug` mode is on, For example:

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

x = ti.field(ti.f32, 128)

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

When you are done with debugging, simply set `debug=False`. Now `assert` will be ignored and there will be no runtime overhead.

32.4 Compile-time `ti.static_assert`

`ti.static_assert`(*cond*, *msg=None*)

Like `ti.static_print`, we also provide a static version of `assert`: `ti.static_assert`. It can be useful to make assertions on data types, dimensionality, and shapes. It works whether `debug=True` is specified or not. When an assertion fails, it will raise an `AssertionError`, just like a Python-scope `assert`.

For example:

```
@ti.func
def copy(dst: ti.template(), src: ti.template()):
    ti.static_assert(dst.shape == src.shape, "copy() needs src and dst fields to be␣
↪same shape")
    for I in ti.grouped(src):
        dst[I] = src[I]
    return x % 2 == 1
```

32.5 Pretty Taichi-scope traceback

As we all know, Python provides a useful stack traceback system, which could help you locate the issue easily. But sometimes stack tracebacks from **Taichi-scope** could be extremely complicated and hard to read. For example:

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

@ti.func
def func3():
    ti.static_assert(1 + 1 == 3)

@ti.func
def func2():
    func3()

@ti.func
def func1():
    func2()

@ti.kernel
def func0():
    func1()

func0()
```

Running this code, of course, will result in an `AssertionError`:

```
Traceback (most recent call last):
  File "misc/demo_excepthook.py", line 20, in <module>
    func0()
  File "/root/taichi/python/taichi/lang/kernel.py", line 559, in wrapped
    return primal(*args, **kwargs)
  File "/root/taichi/python/taichi/lang/kernel.py", line 488, in __call__
    self.materialize(key=key, args=args, arg_features=arg_features)
  File "/root/taichi/python/taichi/lang/kernel.py", line 367, in materialize
    taichi_kernel = taichi_kernel.define(taichi_ast_generator)
  File "/root/taichi/python/taichi/lang/kernel.py", line 364, in taichi_ast_generator
    compiled()
  File "misc/demo_excepthook.py", line 18, in func0
    func1()
  File "/root/taichi/python/taichi/lang/kernel.py", line 39, in decorated
    return fun.__call__(*args)
  File "/root/taichi/python/taichi/lang/kernel.py", line 79, in __call__
    ret = self.compiled(*args)
```

(continues on next page)

(continued from previous page)

```
  File "misc/demo_excepthook.py", line 14, in func1
    func2()
  File "/root/taichi/python/taichi/lang/kernel.py", line 39, in decorated
    return fun.__call__(*args)
  File "/root/taichi/python/taichi/lang/kernel.py", line 79, in __call__
    ret = self.compiled(*args)
  File "misc/demo_excepthook.py", line 10, in func2
    func3()
  File "/root/taichi/python/taichi/lang/kernel.py", line 39, in decorated
    return fun.__call__(*args)
  File "/root/taichi/python/taichi/lang/kernel.py", line 79, in __call__
    ret = self.compiled(*args)
  File "misc/demo_excepthook.py", line 6, in func3
    ti.static_assert(1 + 1 == 3)
  File "/root/taichi/python/taichi/lang/error.py", line 14, in wrapped
    return foo(*args, **kwargs)
  File "/root/taichi/python/taichi/lang/impl.py", line 252, in static_assert
    assert cond
AssertionError
```

You may already feel brain fried by the annoying `decorated`'s and `__call__`'s. These are the Taichi internal stack frames. They have almost no benefit for end-users but make the traceback hard to read.

For this purpose, we may want to use `ti.init(excepthook=True)`, which *hooks* on the exception handler, and make the stack traceback from Taichi-scope easier to read and intuitive. e.g.:

```
import taichi as ti
ti.init(excepthook=True)  # just add this option!

...
```

And the result will be:

```
========== Taichi Stack Traceback ==========
In <module>() at misc/demo_excepthook.py:21:
-------------------------------------------
@ti.kernel
def func0():
    func1()

func0()  <--
-------------------------------------------
In func0() at misc/demo_excepthook.py:19:
-------------------------------------------
    func2()

@ti.kernel
def func0():
    func1()  <--

func0()
-------------------------------------------
In func1() at misc/demo_excepthook.py:15:
-------------------------------------------
    func3()

@ti.func
```

(continues on next page)

(continued from previous page)

```
def func1():
    func2()  <--

@ti.kernel
-----------------------------------------
In func2() at misc/demo_excepthook.py:11:
-----------------------------------------
    ti.static_assert(1 + 1 == 3)

@ti.func
def func2():
    func3()  <--

@ti.func
-----------------------------------------
In func3() at misc/demo_excepthook.py:7:
-----------------------------------------
ti.enable_excepthook()

@ti.func
def func3():
    ti.static_assert(1 + 1 == 3)  <--

@ti.func
-----------------------------------------
AssertionError
```

See? Our exception hook has removed some useless Taichi internal frames from traceback. What's more, although not visible in the doc, the output is **colorful**!

Note: For IPython / Jupyter notebook users, the IPython stack traceback hook will be overriden by the Taichi one when `ti.enable_excepthook()`.

32.6 Debugging Tips

Debugging a Taichi program can be hard even with the builtin tools above. Here we showcase some common bugs that one may encounter in a Taichi program.

32.6.1 Static type system

Python code in Taichi-scope is translated into a statically typed language for high performance. This means code in Taichi-scope can have a different behavior compared with that in Python-scope, especially when it comes to types.

The type of a variable is simply **determined at its initialization and never changes later**.

Although Taichi's static type system provides better performance, it may lead to bugs if programmers carelessly used the wrong types. For example,

```
@ti.kernel
def buggy():
    ret = 0  # 0 is an integer, so `ret` is typed as int32
    for i in range(3):
```

(continues on next page)

(continued from previous page)

```
        ret += 0.1 * i  # i32 += f32, the result is still stored in int32!
    print(ret)  # will show 0

buggy()
```

The code above shows a common bug due to Taichi's static type system. The Taichi compiler should show a warning like:

```
[W 06/27/20 21:43:51.853] [type_check.cpp:visit@66] [$19] Atomic add (float32 to␣
↪int32) may lose precision.
```

This means that Taichi cannot store a `float32` result precisely to `int32`. The solution is to initialize `ret` as a float-point value:

```
@ti.kernel
def not_buggy():
    ret = 0.0  # 0 is a floating point number, so `ret` is typed as float32
    for i in range(3):
        ret += 0.1 * i  # f32 += f32. OK!
    print(ret)  # will show 0.6

not_buggy()
```

32.6.2 Advanced Optimization

Taichi has an advanced optimization engine to make your Taichi kernel to be as fast as it could. But like what `gcc -O3` does, advanced optimization may occasionally lead to bugs as it tries too hard. This includes runtime errors such as:

```
`RuntimeError:  [verify.cpp:basic_verify@40] stmt 8 cannot have operand 7.`
```

You may use `ti.init(advanced_optimization=False)` to turn off advanced optimization and see if the issue still exists:

```
import taichi as ti

ti.init(advanced_optimization=False)

...
```

Whether or not turning off optimization fixes the issue, please feel free to report this bug on GitHub. Thank you!

CHAPTER 33

Extension libraries

The Taichi programming language offers a minimal and generic built-in standard library. Extra domain-specific functionalities are provided via **extension libraries**:

33.1 Taichi GLSL

Taichi GLSL is an extension library of Taichi, aiming at providing useful helper functions including:

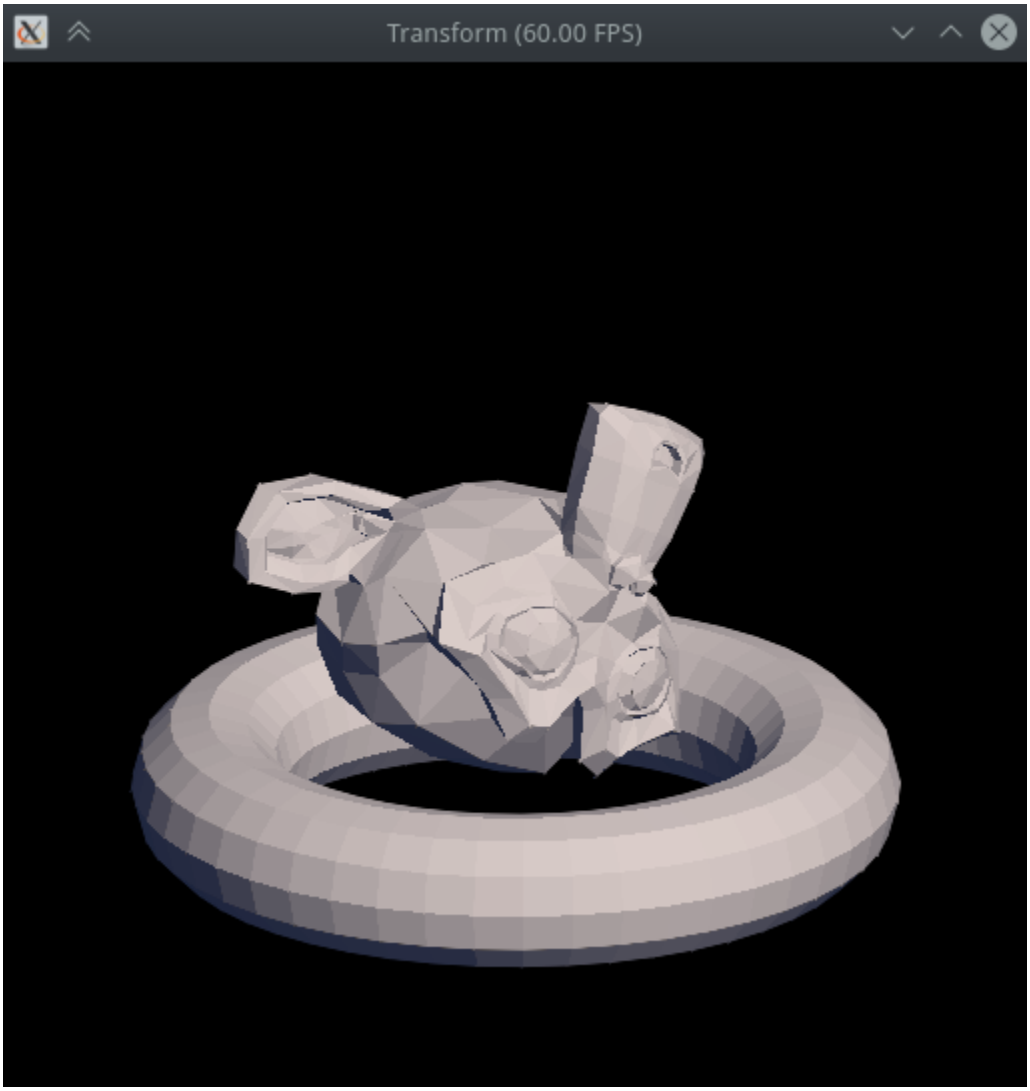
1. Handy scalar functions like `clamp`, `smoothstep`, `mix`, `round`.
2. GLSL-alike vector functions like `normalize`, `distance`, `reflect`.
3. Well-behaved random generators including `randUnit2D`, `randNDRange`.
4. Handy vector and matrix initializer: `vec` and `mat`.
5. Handy vector component shuffle accessor like `v.xy`.

Click here for Taichi GLSL Documentation.

```
python3 -m pip install taichi_glsl
```

33.2 Taichi THREE

Taichi THREE is an extension library of Taichi to render 3D scenes into nice-looking 2D images in real-time (work in progress).



Click here for Taichi THREE Tutorial.

```
python3 -m pip install taichi_three
```

CHAPTER 34

Export your results

Taichi has functions that help you **export visual results to images or videos**. This tutorial demonstrates how to use them step by step.

34.1 Export images

- There are two ways to export visual results of your program to images.
- The first and easier way is to make use of `ti.GUI`.
- The second way is to call some Taichi functions such as `ti.imwrite`.

34.1.1 Export images using `ti.GUI.show`

- `ti.GUI.show(filename)` can not only display the GUI canvas on your screen, but also save the image to your specified `filename`.
- Note that the format of the image is fully determined by the suffix of `filename`.
- Taichi now supports saving to `png`, `jpg`, and `bmp` formats.
- We recommend using `png` format. For example:

```
import taichi as ti
import os

ti.init()

pixels = ti.field(ti.u8, shape=(512, 512, 3))

@ti.kernel
def paint():
    for i, j, k in pixels:
```

(continues on next page)

(continued from previous page)

```
        pixels[i, j, k] = ti.random() * 255

iterations = 1000
gui = ti.GUI("Random pixels", res=512)

# mainloop
for i in range(iterations):
    paint()
    gui.set_image(pixels)

    filename = f'frame_{i:05d}.png'   # create filename with suffix png
    print(f'Frame {i} is recorded in {filename}')
    gui.show(filename)  # export and show in GUI
```

- After running the code above, you will get a series of images in the current folder.

34.1.2 Export images using `ti.imwrite`

To save images without invoking `ti.GUI.show(filename)`, use `ti.imwrite(filename)`. For example:

```
import taichi as ti

ti.init()

pixels = ti.field(ti.u8, shape=(512, 512, 3))

@ti.kernel
def set_pixels():
    for i, j, k in pixels:
        pixels[i, j, k] = ti.random() * 255

set_pixels()
filename = f'imwrite_export.png'
ti.imwrite(pixels.to_numpy(), filename)
print(f'The image has been saved to {filename}')
```

- `ti.imwrite` can export Taichi fields (`ti.Matrix.field`, `ti.Vector.field`, `ti.field`) and numpy arrays `np.ndarray`.
- Same as above `ti.GUI.show(filename)`, the image format (`png`, `jpg` and `bmp`) is also controlled by the suffix of `filename` in `ti.imwrite(filename)`.
- Meanwhile, the resulted image type (grayscale, RGB, or RGBA) is determined by **the number of channels in the input field**, i.e., the length of the third dimension (`field.shape[2]`).
- In other words, a field that has shape `(w, h)` or `(w, h, 1)` will be exported as a grayscale image.
- If you want to export `RGB` or `RGBA` images instead, the input field should have a shape `(w, h, 3)` or `(w, h, 4)` respectively.

Note: All Taichi fields have their own data types, such as `ti.u8` and `ti.f32`. Different data types can lead to different behaviors of `ti.imwrite`. Please check out *GUI system* for more details.

- Taichi offers other helper functions that read and show images in addition to `ti.imwrite`. They are also demonstrated in *GUI system*.

34.2 Export videos

Note: The video export utilities of Taichi depend on `ffmpeg`. If `ffmpeg` is not installed on your machine, please follow the installation instructions of `ffmpeg` at the end of this page.

- `ti.VideoManager` can help you export results in `mp4` or `gif` format. For example,

```
import taichi as ti

ti.init()

pixels = ti.field(ti.u8, shape=(512, 512, 3))

@ti.kernel
def paint():
    for i, j, k in pixels:
        pixels[i, j, k] = ti.random() * 255

result_dir = "./results"
video_manager = ti.VideoManager(output_dir=result_dir, framerate=24, automatic_
↪build=False)

for i in range(50):
    paint()

    pixels_img = pixels.to_numpy()
    video_manager.write_frame(pixels_img)
    print(f'\rFrame {i+1}/50 is recorded', end='')

print()
print('Exporting .mp4 and .gif videos...')
video_manager.make_video(gif=True, mp4=True)
print(f'MP4 video is saved to {video_manager.get_output_filename(".mp4")}')
print(f'GIF video is saved to {video_manager.get_output_filename(".gif")}')
```

After running the code above, you will find the output videos in the `./results/` folder.

34.3 Install ffmpeg

34.3.1 Install ffmpeg on Windows

- Download the `ffmpeg` archive(named `ffmpeg-2020xxx.zip`) from ffmpeg;
- Unzip this archive to a folder, such as "D:/YOUR_FFMPEG_FOLDER";
- **Important:** add `D:/YOUR_FFMPEG_FOLDER/bin` to the `PATH` environment variable;
- Open the Windows `cmd` or `PowerShell` and type the line of code below to test your installation. If `ffmpeg` is set up properly, the version information will be printed.

```
ffmpeg -version
```

34.3.2 Install `ffmpeg` on Linux

- Most Linux distribution came with `ffmpeg` natively, so you do not need to read this part if the `ffmpeg` command is already there on your machine.
- Install `ffmpeg` on Ubuntu

```
sudo apt-get update
sudo apt-get install ffmpeg
```

- Install `ffmpeg` on CentOS and RHEL

```
sudo yum install ffmpeg ffmpeg-devel
```

- Install `ffmpeg` on Arch Linux:
- Test your installation using

```
ffmpeg -h
```

34.3.3 Install `ffmpeg` on OS X

- `ffmpeg` can be installed on OS X using `homebrew`:

```
brew install ffmpeg
```

34.4 Export PLY files

- `ti.PLYwriter` can help you export results in the `ply` format. Below is a short example of exporting 10 frames of a moving cube with vertices randomly colored,

```
import taichi as ti
import numpy as np

ti.init(arch=ti.cpu)

num_vertices = 1000
pos = ti.Vector.field(3, dtype=ti.f32, shape=(10, 10, 10))
rgba = ti.Vector.field(4, dtype=ti.f32, shape=(10, 10, 10))


@ti.kernel
def place_pos():
    for i, j, k in pos:
        pos[i, j, k] = 0.1 * ti.Vector([i, j, k])


@ti.kernel
def move_particles():
    for i, j, k in pos:
        pos[i, j, k] += ti.Vector([0.1, 0.1, 0.1])


@ti.kernel
```

(continues on next page)

(continued from previous page)

```
def fill_rgba():
    for i, j, k in rgba:
        rgba[i, j, k] = ti.Vector(
            [ti.random(), ti.random(), ti.random(), ti.random()])


place_pos()
series_prefix = "example.ply"
for frame in range(10):
    move_particles()
    fill_rgba()
    # now adding each channel only supports passing individual np.array
    # so converting into np.ndarray, reshape
    # remember to use a temp var to store so you dont have to convert back
    np_pos = np.reshape(pos.to_numpy(), (num_vertices, 3))
    np_rgba = np.reshape(rgba.to_numpy(), (num_vertices, 4))
    # create a PLYWriter
    writer = ti.PLYWriter(num_vertices=num_vertices)
    writer.add_vertex_pos(np_pos[:, 0], np_pos[:, 1], np_pos[:, 2])
    writer.add_vertex_rgba(
        np_rgba[:, 0], np_rgba[:, 1], np_rgba[:, 2], np_rgba[:, 3])
    writer.export_frame_ascii(frame, series_prefix)
```

After running the code above, you will find the output sequence of `ply` files in the current working directory. Next, we will break down the usage of `ti.PLYWriter` into 4 steps and show some examples.

- Setup `ti.PLYWriter`

```
# num_vertices must be a positive int
# num_faces is optional, default to 0
# face_type can be either "tri" or "quad", default to "tri"

# in our previous example, a writer with 1000 vertices and 0 triangle faces is created
num_vertices = 1000
writer = ti.PLYWriter(num_vertices=num_vertices)

# in the below example, a writer with 20 vertices and 5 quadrangle faces is created
writer2 = ti.PLYWriter(num_vertices=20, num_faces=5, face_type="quad")
```

- Add required channels

```
# A 2D grid with quad faces
#     y
#     |
# z---/
#    x
#         19---15---11---07---03
#         |    |    |    |    |
#         18---14---10---06---02
#         |    |    |    |    |
#         17---13---19---05---01
#         |    |    |    |    |
#         16---12---08---04---00

writer = ti.PLYWriter(num_vertices=20, num_faces=12, face_type="quad")

# For the vertices, the only required channel is the position,
```

(continues on next page)

(continued from previous page)

```
# which can be added by passing 3 np.array x, y, z into the following function.

x = np.zeros(20)
y = np.array(list(np.arange(0, 4))*5)
z = np.repeat(np.arange(5), 4)
writer.add_vertex_pos(x, y, z)

# For faces (if any), the only required channel is the list of vertex indices that␣
↪each face contains.
indices = np.array([0, 1, 5, 4]*12)+np.repeat(
    np.array(list(np.arange(0, 3))*4)+4*np.repeat(np.arange(4), 3), 4)
writer.add_faces(indices)
```

- Add optional channels

```
# Add custome vertex channel, the input should include a key, a supported datatype␣
↪and, the data np.array
vdata = np.random.rand(20)
writer.add_vertex_channel("vdata1", "double", vdata)

# Add custome face channel
foo_data = np.zeros(12)
writer.add_face_channel("foo_key", "foo_data_type", foo_data)
# error! because "foo_data_type" is not a supported datatype. Supported ones are
# ['char', 'uchar', 'short', 'ushort', 'int', 'uint', 'float', 'double']

# PLYwriter already defines several useful helper functions for common channels
# Add vertex color, alpha, and rgba
# using float/double r g b alpha to reprent color, the range should be 0 to 1
r = np.random.rand(20)
g = np.random.rand(20)
b = np.random.rand(20)
alpha = np.random.rand(20)
writer.add_vertex_color(r, g, b)
writer.add_vertex_alpha(alpha)
# equivilantly
# add_vertex_rgba(r, g, b, alpha)

# vertex normal
writer.add_vertex_normal(np.ones(20), np.zeros(20), np.zeros(20))

# vertex index, and piece (group id)
writer.add_vertex_id()
writer.add_vertex_piece(np.ones(20))

# Add face index, and piece (group id)
# Indexing the existing faces in the writer and add this channel to face channels
writer.add_face_id()
# Set all the faces is in group 1
writer.add_face_piece(np.ones(12))
```

- Export files

```
series_prefix = "example.ply"
series_prefix_ascii = "example_ascii.ply"
# Export a single file
# use ascii so you can read the content
```

(continues on next page)

(continued from previous page)

```
writer.export_ascii(series_prefix_ascii)

# alternatively, use binary for a bit better performance
# writer.export(series_prefix)

# Export a sequence of files, ie in 10 frames
for frame in range(10):
    # write each frame as i.e. "example_000000.ply" in your current running folder
    writer.export_frame_ascii(frame, series_prefix_ascii)
    # alternatively, use binary
    # writer.export_frame(frame, series_prefix)

    # update location/color
    x = x + 0.1*np.random.rand(20)
    y = y + 0.1*np.random.rand(20)
    z = z + 0.1*np.random.rand(20)
    r = np.random.rand(20)
    g = np.random.rand(20)
    b = np.random.rand(20)
    alpha = np.random.rand(20)
    # re-fill
    writer = ti.PLYWriter(num_vertices=20, num_faces=12, face_type="quad")
    writer.add_vertex_pos(x, y, z)
    writer.add_faces(indices)
    writer.add_vertex_channel("vdata1", "double", vdata)
    writer.add_vertex_color(r, g, b)
    writer.add_vertex_alpha(alpha)
    writer.add_vertex_normal(np.ones(20), np.zeros(20), np.zeros(20))
    writer.add_vertex_id()
    writer.add_vertex_piece(np.ones(20))
    writer.add_face_id()
    writer.add_face_piece(np.ones(12))
```

34.4.1 Import `ply` files into Houdini and Blender

Houdini supports importing a series of `ply` files sharing the same prefix/post-fix. Our `export_frame` can achieve the requirement for you. In Houdini, click `File->Import->Geometry` and navigate to the folder containing your frame results, who should be collapsed into one single entry like `example_$F6.ply (0-9)`. Double-click this entry to finish the importing process.

Blender requires an add-on called Stop-motion-OBJ to load the result sequences. Detailed documentation is provided by the author on how to install and use the add-on. If you're using the latest version of Blender (2.80+), download and install the latest release of Stop-motion-OBJ. For Blender 2.79 and older, use version `v1.1.1` of the add-on.

CHAPTER 35

Command line utilities

A successful installation of Taichi should add a CLI (Command-Line Interface) to your system, which is helpful to perform several rountine tasks quickly. To invoke the CLI, please run `ti` or `python3 -m taichi`.

35.1 Examples

Taichi provides a set of bundled examples. You could run `ti example -h` to print the help message and get a list of available example names. For instance, to run the basic *fractal* example, try: `ti example fractal` from your shell. (`ti example fractal.py` should also work)

35.2 Changelog

Sometimes it's convenient to view the changelog of the current version of Taichi, to do so from your shell, you could run `ti changelog`.

CHAPTER 36

Global settings

36.1 Backends

- To specify which Arch to use: `ti.init(arch=ti.cuda)`.
- To specify pre-allocated memory size for CUDA: `ti.init(device_memory_GB=0.5)`.
- To disable unified memory usage on CUDA: `ti.init(use_unified_memory=False)`.
- To specify which GPU to use for CUDA: `export CUDA_VISIBLE_DEVICES=[gpuid]`.
- To disable a backend on start up, say, CUDA: `export TI_ENABLE_CUDA=0`.

36.2 Compilation

- Disable advanced optimization to save compile time & possible errors: `ti.init(advanced_optimization=False)`.
- Disable fast math to prevent possible undefined math behavior: `ti.init(fast_math=False)`.
- To print preprocessed Python code: `ti.init(print_preprocessed=True)`.
- To show pretty Taichi-scope stack traceback: `ti.init(excepthook=True)`.
- To print intermediate IR generated: `ti.init(print_ir=True)`.

36.3 Runtime

- Restart the entire Taichi system (destroy all fields and kernels): `ti.reset()`.
- To start program in debug mode: `ti.init(debug=True)` or `ti debug your_script.py`.
- To disable importing torch on start up: `export TI_ENABLE_TORCH=0`.

36.4 Logging

- Show more detailed log to level TRACE: `ti.init(log_level=ti.TRACE)` or `ti.set_logging_level(ti.TRACE)`.
- Eliminate verbose outputs: `ti.init(verbose=False)`.

36.5 Develop

- To trigger GDB when Taichi crashes: `ti.init(gdb_trigger=True)`.
- Cache compiled runtime bitcode in **dev mode** to save start up time: `export TI_CACHE_RUNTIME_BITCODE=1`.
- To specify how many threads to run test: `export TI_TEST_THREADS=4` or `ti test -t4`.

36.6 Specifying `ti.init` arguments from environment variables

Arguments for `ti.init` may also be specified from environment variables. For example:

- `ti.init(arch=ti.cuda)` is equivalent to `export TI_ARCH=cuda`.
- `ti.init(log_level=ti.TRACE)` is equivalent to `export TI_LOG_LEVEL=trace`.
- `ti.init(debug=True)` is equivalent to `export TI_DEBUG=1`.
- `ti.init(use_unified_memory=False)` is equivalent to `export TI_USE_UNIFIED_MEMORY=0`.

If both `ti.init` argument and the corresponding environment variable are specified, then the one in the environment variable will **override** the one in the argument, e.g.:

- if `ti.init(arch=ti.cuda)` and `export TI_ARCH=opengl` are specified at the same time, then Taichi will choose `ti.opengl` as backend.
- if `ti.init(debug=True)` and `export TI_DEBUG=0` are specified at the same time, then Taichi will disable debug mode.

Note: If `ti.init` is called twice, then the configuation in first invocation will be completely discarded, e.g.:

```
ti.init(debug=True)
print(ti.cfg.debug)  # True
ti.init()
print(ti.cfg.debug)  # False
```

CHAPTER 37

Acknowledgments

Taichi depends on other open-source projects, which are shipped with taichi and users do not have to install manually: pybind11, fmt, Catch2, spdlog, stb_image, stb_image_write, stb_truetype, tinyobjloader, ffmpeg, miniz.

Halide has been a great reference for us to learn about the Apple Metal API and the LLVM NVPTX backend API.

CHAPTER 38

Installing the legacy Taichi Library

Note: This is NOT for installing the Taichi programming language. Unless you are building a legacy project based on the legacy Taichi library (e.g. taichi_mpm and spgrid_topo_opt) you should always install Taichi using `pip`.

If you are working on the Taichi compiler and need to build from source, see *Developer installation*.

Supported platforms:

- Ubuntu (gcc 5+)
- Mac OS X (gcc 5+, clang 4.0+)
- Windows (Microsoft Visual Studio 2017)

Make sure you have `python 3.5+`.

38.1 Ubuntu, Arch Linux, and Mac OS X

```
wget https://raw.githubusercontent.com/yuanming-hu/taichi/legacy/install.py
python3 install.py
```

Note, if python complains that a package is missing, simply rerun install.py and the package should be loaded.

38.2 Windows

Download and execute this script with python3.

Additional environment variables: (assuming taichi is installed in `DIR/taichi`) Set `TAICHI_REPO_DIR` as `DIR/taichi` (e.g. `E:/repos/taichi`). Add `%TAICHI_REPO_DIR%/python` to `PYTHONPATH`, `DIR/taichi/bin` (e.g. `E:/repos/taichi/bin`) to `PATH`. Restart cmd or PowerShell, and you should be able to run command `ti`.

38.3 Build with Double Precision (64 bit) Float Point

```
export TC_USE_DOUBLE=1
ti build
```

Index

T

V

W