Overview

1  Why new programming language  

2  Design decisions  

3  Hello, world!  
  3.1 import taichi as ti  
  3.2 Portability  
  3.3 (Sparse) Tensors  
  3.4 Functions and kernels  
  3.5 Parallel for-loops  
  3.6 Interacting with Python  

4  Syntax  
  4.1 Kernels  
  4.2 Functions  
  4.3 Data layout  
  4.4 Scalar arithmetics  
  4.5 Debugging  
  4.6 Why Python frontend  

5  Type system  
  5.1 Supported types  
  5.2 Default precisions  
  5.3 Type casts  

6  Linear algebra  
  6.1 Matrices  
  6.2 Vectors  

7  Tensors and matrices  
  7.1 Tensors of scalars  
  7.2 Tensors of matrices  
  7.3 Matrix size  

8  Global Settings  

9  Interacting with external arrays  
  9.1 Conversion between Taichi tensors and external arrays  

---

i
9.2 Use external arrays as Taichi kernel parameters ......................................................... 22

10 Template metaprogramming ......................................................................................... 23
  10.1 Dimensionality-independent programming using grouped indices .............................. 23
  10.2 Tensor size reflection ................................................................................................. 24
  10.3 Compile-time evaluations .......................................................................................... 24
  10.4 When to use for loops with \texttt{ti.static} .................................................................... 25

11 Advanced data layouts ................................................................................................. 27
  11.1 The default data layout using \texttt{shape} .................................................................. 27
  11.2 Advanced data layout specification .......................................................................... 28
  11.3 Examples .................................................................................................................. 28
  11.4 Struct-fors on advanced (dense) data layouts ............................................................ 28

12 Sparse computation ..................................................................................................... 31

13 Differentiable programming ........................................................................................ 33

14 Objective data-oriented programming ........................................................................ 35

15 The life of a Taichi kernel ............................................................................................. 37
  15.1 Kernel registration .................................................................................................... 38
  15.2 Template instantiation and caching ........................................................................... 38
  15.3 Code transformation and optimizations ..................................................................... 39
  15.4 The just-in-time (JIT) compilation engine ................................................................ 40
  15.5 Kernel launching ....................................................................................................... 40

16 Utilities ........................................................................................................................ 41
  16.1 Logging ..................................................................................................................... 41
  16.2 Trigger GDB when the program crashes: ................................................................ 41
  16.3 Interface System ....................................................................................................... 41
  16.4 Serialization .............................................................................................................. 42
  16.5 Progress Notification ............................................................................................... 42

17 Developer installation .................................................................................................. 43
  17.1 Setting up CUDA 10.1 on Ubuntu 18.04 .................................................................. 44
  17.2 Prebuilt LLVM for Windows CI ................................................................................ 44
  17.3 Troubleshooting ..................................................................................................... 45

18 Contribution guidelines ............................................................................................... 47
  18.1 How to contribute bug fixes and new features ........................................................... 47
  18.2 High-level guidelines ............................................................................................... 47
  18.3 Commit rules ............................................................................................................ 48
  18.4 Making good pull requests ....................................................................................... 48
  18.5 Reviewing & PR merging .......................................................................................... 48
  18.6 Tips on Taichi compiler development ..................................................................... 48
  18.7 Testing ........................................................................................................................ 49
  18.8 Documentation ......................................................................................................... 49
  18.9 C++ and Python standards ....................................................................................... 49
  18.10 (Linux only) pinpointing runtime errors using GDB .............................................. 49
  18.11 Efficient Code Navigation across Python/C++ ....................................................... 49
    18.11.1 Folder structure ................................................................................................. 49

19 C++ Style ..................................................................................................................... 51
  19.1 Naming ...................................................................................................................... 51
  19.2 Dos ............................................................................................................................ 51
19.3 Don’ts ............................................. 51
19.4 Documentation .................................. 52

20 Internal designs (WIP) ........................................ 53
  20.1 Vector type system .................................. 53
  20.2 Intermediate representation .......................... 53
  20.3 Code generation .................................. 53

21 Frequently Asked Questions ................................. 55

22 Acknowledgments ........................................... 57

23 Installing the legacy Taichi Library ......................... 59
  23.1 Ubuntu, Arch Linux, and Mac OS X .................. 59
  23.2 Windows ........................................... 59
  23.3 Build with Double Precision (64 bit) Float Point .... 60
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
Design decisions

- Decouple computation from data structures
- Domain-specific compiler optimizations
- Megakernels
- Two-scale automatic differentiation
- Embedding in Python
We introduce the Taichi programming language through a very basic fractal example.

If you haven’t done so, please install Taichi via pip. Depending on your hardware and OS, please execute one of the following commands:

```
# Python 3.6+ needed
# CPU only. No GPU/CUDA needed. (Linux, OS X and Windows)
python3 -m pip install taichi-nightly

# With GPU (CUDA 10.0) support (Linux only)
python3 -m pip install taichi-nightly-cuda-10-0

# With GPU (CUDA 10.1) support (Linux only)
python3 -m pip install taichi-nightly-cuda-10-1
```

Now you are ready to run the Taichi code below (python3 fractal.py) to compute a Julia set:

```
# fractal.py

import taichi as ti

ti.init(arch=ti.cuda) # Run on GPU by default

n = 320
pixels = ti.var(dt=ti.f32, shape=(n * 2, n))

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

@ti.kernel
def paint(t: ti.f32):
```

(continues on next page)
for i, j in pixels:  # Parallized over all pixels
    c = ti.Vector([-0.8, ti.sin(t) * 0.2])
    z = ti.Vector([float(i) / n - 1, float(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("Fractal", (n * 2, n))

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

Let's dive into components of this simple Taichi program.

### 3.1 import taichi as ti

Taichi is an embedded domain-specific language (DSL) in Python. It pretends to be a plain Python package, although heavy engineering has been done to make this happen.

This design decision virtually makes every Python programmer capable of writing Taichi programs, after minimal learning efforts. You can also reuse the package management system, Python IDEs, and existing Python packages.

### 3.2 Portability

Taichi supports both CPUs and NVIDIA GPUs.

```python
# Run on GPU
ti.init(arch=ti.cuda)
# Run on CPU (default)
ti.init(arch=ti.x64)
```

If the machine does not have CUDA support, Taichi will fall back to CPUs instead.

**Note:** When running the CUDA backend on Windows and ARM devices (e.g. NVIDIA Jetson), Taichi will by default allocate 1 GB memory for tensor storage. You can override this 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 total available GPU memory.

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

### 3.3 (Sparse) Tensors

Taichi is a data-oriented programming language, where dense or spatially-sparse tensors are first-class citizens. See *Sparse computation* for more details on sparse tensors.
pixels = ti.var(dt=ti.f32, shape=(n * 2, n)) allocates a 2D dense tensor named `pixel` of size (640, 320) and type `ti.f32` (i.e. `float` in C).

### 3.4 Functions and kernels

Computation happens within Taichi **kernels**. Kernel arguments must be type-hinted. 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**.

You can also define Taichi **functions** with `ti.func`, which can be called and reused by kernels and other functions.

**Note:** Taichi-scope vs. Python-scope: everything decorated with `ti.kernel` and `ti.func` is in Taichi-scope, which will be compiled by the Taichi compiler. Code outside the Taichi-scope is simply native Python code.

**Warning:** Taichi kernels must be called in the Python-scope. I.e., **nested Taichi kernels are not supported**. Nested functions are allowed. **Recursive functions are not supported for now.**

Taichi functions can only be called in Taichi-scope.

For those who came from the world of CUDA, `ti.func` corresponds to `__device__`, and `ti.kernel` corresponds to `__global__`.

### 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 that in native Python, except that it will be parallelized when used as the outermost scope. Range-for loops can be nested.

```python
@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
```

**Struct-for loops** have a cleaner syntax, and are particularly useful when iterating over tensor elements. In the fractal 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)`.

```python
@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
```

**Struct-for loops** have a cleaner syntax, and are particularly useful when iterating over tensor elements. In the fractal 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)`.

**3.4. Functions and kernels** 7
Note: Struct-for is the key to *Sparse computation* in Taichi, as it will only loop over active elements in a sparse tensor. In dense tensors, all elements are active.

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

```python
# Good kernel
@ti.func
def foo():
    for i in x:
        ...

# Bad kernel
@ti.func
def bar(k: ti.i32):
    # The outermost scope is a `if` statement, not the struct-for loop!
    if k > 42:
        for i in x:
            ...
```

**Warning:** Struct-for’s must be at the outer-most scope of kernels.

### 3.6 Interacting with Python

Everything outside Taichi-scope (*ti.func* and *ti.kernel*) is simply Python. You can use your favorite Python packages (e.g. *numpy*, *pytorch*, *matplotlib*) with Taichi.

In Python-scope, you can access Taichi tensors using plain indexing syntax, and helper functions such as *from_numpy* and *to_torch*:

```python
image[42, 11] = 0.7
print(image[1, 63])

import numpy as np
pixels.from_numpy(np.random.rand(n + 2, n))

import matplotlib.pyplot as plt
plt.imshow(pixels.to_numpy())
plt.show()
```
4.1 Kernels

Kernel arguments must be type-hinted. Kernels can have at most 8 parameters, e.g.,

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

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

- For differentiable programming kernels should better have either serial statements or a single parallel for-loop. If you don’t use differentiable programming, feel free to ignore this tip.

```python
@ti.kernel
def a_hard_kernel_to_auto_differentiate():
    sum = 0
    for i in x:
        sum += x[i]
    for i in y:
        y[i] = sum

# instead, split it into multiple kernels to be nice to the Taichi autodiff compiler:

@ti.kernel
def reduce():
    for i in x:
        sum[None] += x[i]

@ti.kernel
def assign()
```

(continues on next page)
for i in y:
    y[i] = sum[None]

def main():
    with ti.Tape(loss):
        ...
        sum[None] = 0
        reduce()
        assign()
        ...

4.2 Functions

Use @ti.func to decorate your Taichi functions. These functions are callable only in Taichi-scope. Don’t call them in Python-scope. All function calls are force-inlined, so no recursion supported.

@ti.func
def laplacian(t, i, j):
    return inv_dx2 * (-4 * p[t, i, j] + p[t, i, j - 1] + p[t, i, j + 1] + p[t, i + 1, j] + p[t, i - 1, j])

@ti.kernel
def fdtd(t: ti.i32):
    for i in range(n_grid): # Parallelized over GPU threads
        for j in range(n_grid):
            laplacian_p = laplacian(t - 2, i, j)
            laplacian_q = laplacian(t - 1, i, j)
            p[t, i, j] = 2 * p[t - 1, i, j] + (c * c * dt * dt + c * alpha * dt) * laplacian_q - p[t - 2, i, j] - c * alpha * dt * laplacian_p

Warning: Functions with multiple return values are not supported for now. Use a local variable to store the results instead:

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

# Good function
@ti.func
def safe_sqrt(x):
    rst = 0.0
    if x >= 0:
        rst = ti.sqrt(x)
    else:
        rst = 0.0
    return rst
Warning: All functions are force-inlined. Function arguments are passed by value.

4.3 Data layout

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

4.4 Scalar arithmetics

Supported scalar functions:

- `ti.sin(x)`
- `ti.cos(x)`
- `ti.cast(x, type)`
- `ti.sqr(x)`
- `ti.floor(x)`
- `ti.inv(x)`
- `ti.tan(x)`
- `ti.tanh(x)`
- `ti.exp(x)`
- `ti.log(x)`
- `abs(x)`
- `ti.random(type)`
- `max(a, b)`
- `min(a, b)`
- `ti.length(dynamic_snode)`
- Inplace adds are atomic on global data. I.e., \(a += b\) is equivalent to `ti.atomic_add(a, b)`

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`. Read Default precisions for more details on default numerical types.

4.5 Debugging

Debug your program with `print(x)`. 

4.3. Data layout

11
4.6 Why Python frontend

Embedding the language 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 tensors, even if the tensor 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 tensors with pure python scripts. A Taichi kernel should be used to initialize a huge tensor.
5.1 Supported types

Currently, supported basic types in Taichi are
- int32 `ti.i32`
- int64 `ti.i64`
- float32 `ti.f32`
- float64 `ti.f64`

Boolean types are represented using `ti.i32`.

Binary operations on different types will give you a promoted type, following the C programming language, e.g.
- `i32 + f32 = f32`
- `f32 + f64 = f64`
- `i32 + i64 = i64`

5.2 Default precisions

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

Default precisions 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)
```
5.3 Type casts

Use `ti.cast` to type-cast scalar values.

```python
a = 1.4
b = ti.cast(a, ti.i32)
c = ti.cast(b, ti.f32)

# Equivalently, use `int()` and `float()`
# to converting to default float-point/integer types
b = int(a)
c = float(b)

# Element-wise casts in matrices
mat = ti.Matrix([[3.0, 0.0], [0.3, 0.1]])
mat_int = mat.cast(int)
mat_int2 = mat.cast(ti.i32)
```
CHAPTER 6

Linear algebra

6.1 Matrices

- `ti.Matrix` is for small matrices (e.g. 3x3) only. If you have 64x64 matrices, you should consider using a 2D tensor of scalars.
- `ti.Vector` is the same as `ti.Matrix`, except that it has only one column.
- Differentiate element-wise product `*` and matrix product `@`.
- `ti.transposed(A)` or simply `A.T()`
- `ti.inverse(A)`
- `ti.Matrix.abs(A)`
- `ti.tr(A)`
- `ti.determinant(A, type)`
- `ti.cross(a, b)`, where `a` and `b` are 3D vectors (i.e. 3x1 matrices)
- `A.cast(type)`
- `R, S = ti.polar_decompose(A, ti.f32)`
- `U, sigma, V = ti.svd(A, ti.f32)` (Note that sigma is a 3x3 diagonal matrix)

6.2 Vectors

Vectors are special matrices with only 1 column. In fact, `ti.Vector` is just an alias of `ti.Matrix`.
- Dot product: `a.dot(b)`, where `a` and `b` are vectors. `ti.transposed(a) @ b` will give you a matrix of size 1x1, which is not a scalar.
- Outer product: `ti.outer_product(a, b)`
- 1-2 norm: `a.norm(eps = 0)`
– returns $\sqrt{\sum_i (x_i ^ 2) + \text{eps}}$
– Set $\text{eps} = 1e^{-5}$ for example, to safeguard the operator’s gradient on zero vectors during differentiable programming.
Tensors and matrices

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

Although mathematically matrices are treated as 2D tensors, in Taichi, tensor and matrix are two completely different things. Matrices can be used as tensor elements, so you have tensors of matrices.

7.1 Tensors of scalars

- Every global variable is an N-dimensional tensor.
- Global scalars are treated as 0-D tensors of scalars.
- Tensors are accessed using indices, e.g. \( x[i, j, k] \) if \( x \) is a scalar 3D tensor. For a 0-D tensor, access it as \( x[\text{None}] \).
- Even when accessing 0-D tensor \( x \), use \( x[\text{None}] = 0 \) instead of \( x = 0 \). Please always use indexing to access entries in tensors.
- Tensor values are initially zero.
- Sparse tensors are initially inactive.

7.2 Tensors of matrices

Suppose you have a \( 128 \times 64 \) global grid \( A \), each node containing a \( 3 \times 2 \) matrices. In this case you need to allocate a \( 128 \times 64 \) tensor of \( 3 \times 2 \) matrix, using the statement \( A = \text{ti.Matrix}(3, 2, \text{dt=ti.f32, shape=(128, 64)}) \).

- If you want to get the matrix of grid node \( i, j \), please use \( \text{mat} = A[i, j] \). \( \text{mat} \) is simply a \( 3 \times 2 \) matrix.
- To get the element on the first row and second column of that matrix, use \( \text{mat}[0, 1] \) or \( A[i, j][0, 1] \).
• As you may have noticed, there are two indexing operators [], the first is for tensor indexing, the second for matrix indexing.
• For a tensor $F$ of element $\text{ti.Matrix}$, make sure you first index the tensor dimensions, and then the matrix dimensions: $F[i, j, k][0, 2]$. (Assuming $F$ is a 3D tensor with $\text{ti.Matrix}$ of size $3\times3$ as elements)
• $\text{ti.Vector}$ is simply an alias of $\text{ti.Matrix}$.
• See *Linear algebra* for more on matrices.

### 7.3 Matrix size

For performance reasons matrix operations will be unrolled, therefore we suggest using only small matrices. For example, $2\times1$, $3\times3$, $4\times4$ matrices are fine, yet $32\times6$ is probably too big as a matrix size.

| Warning: Due to the unrolling mechanisms, operating on large matrices (e.g. $32\times128$) can lead to long compilation time and low performance. |

If you have a dimension that is too large (e.g. $64$), it’s better to declare a tensor of size $64$. E.g., instead of declaring $\text{ti.Matrix}(64, 32, \text{dt=ti.f32, shape=(3, 2)})$, declare $\text{ti.Matrix}(3, 2, \text{dt=ti.f32, shape=(64, 32)})$. Try to put large dimensions to tensors instead of matrices.
Global Settings

- Restart the Taichi runtime system (clear memory, destroy all variables and kernels): `ti.reset()
- Eliminate verbose outputs: `ti.get_runtime().set_verbose(False)
- To specify which GPU to use: `export CUDA_VISIBLE_DEVICES=0`
chapter 9

interacting with external arrays

here external arrays refer to numpy.ndarray or torch.tensor.

9.1 conversion between taichi tensors and external arrays

use to_numpy/from_numpy/to_torch/from_torch:

```python
n = 4
m = 7

# taichi tensors
val = ti.var(ti.i32, shape=(n, m))
vec = ti.Vector(3, dt=ti.i32, shape=(n, m))
mat = ti.Matrix(3, 4, dt=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)
vec.from_numpy(arr)
arr = vec.to_numpy()
assert arr.shape == (n, m, 3, 1)
```

(continues on next page)
arr = vec.to_numpy(as_vector=True)
assert arr.shape == (n, m, 3)

# 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)

## 9.2 Use 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.

```python
n = 4
m = 7
val = ti.var(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
```
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.

### 10.1 Dimensionality-independent programming using grouped indices

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

@ti.kernel
def array_op(x: ti.template(), y: ti.template()):
    # If tensor x is 2D
    for I in ti.grouped(x):  # I is a vector of size x.dim() and data type i32
        y[I + ti.Vector([0, 1])] = I[0] + I[1]
    # is equivalent to
    for i, j in x:
        y[i, j + 1] = i + j
```
10.2 Tensor size reflection

Sometimes it will be useful to get the dimensionality (`tensor.dim()`) and shape (`tensor.shape()`) of tensors. These functions can be used in both Taichi kernels and python scripts.

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

For sparse tensors, the full domain shape will be returned.

10.3 Compile-time evaluations

Using compile-time evaluation will allow certain computation to happen when kernels are instantiated. Such computation has no overhead at runtime.

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

```python
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

```python
@ti.kernel
def g2p(f: ti.i32):
    for p in range(0, n_particles):
        base = ti.cast(x[f, p] * inv_dx - 0.5, ti.i32)
        fx = x[f, p] * inv_dx - ti.cast(base, real)
        w = [0.5 * ti.sqr(1.5 - fx), 0.75 - ti.sqr(fx - 1.0),
             0.5 * ti.sqr(fx - 0.5)]
        new_v = ti.Vector([0.0, 0.0])
        new_C = ti.Matrix([[0.0, 0.0], [0.0, 0.0]])
        # Unrolled 9 iterations for higher performance
        for i in ti.static(range(3)):
            for j in ti.static(range(3)):
                dpos = ti.cast(ti.Vector([i, j]), real) - fx
                g_v = grid_v_out[base(0) + i, base(1) + j]
                weight = w[i](0) * w[j](1)
                new_v += weight * g_v
                new_C += 4 * weight * ti.outer_product(g_v, dpos) * inv_dx
        v[f + 1, p] = new_v
        x[f + 1, p] = x[f, p] + dt * v[f + 1, p]
        C[f + 1, p] = new_C
```
10.4 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 tensors can be run-time variables. For example, if $x$ is a 1-D tensor of 3D vector, accessed as $x[tensor\_index][matrix\_index]$. The first index can be variable, yet the second must be a constant.

For example, code for resetting this tensor of vectors should be

```python
@ti.kernel
def reset():
    for i in x:
        for j in ti.static(range(3)):
            # The inner loop must be unrolled since j is a vector index instead
            # of a global tensor index.
            x[i][j] = 0
```

```python
@ti.kernel
def reset():
    for i in x:
        for j in ti.static(range(3)):
            # The inner loop must be unrolled since j is a vector index instead
            # of a global tensor index.
            x[i][j] = 0
```
Memory layout is key to performance, especially for memory-bound applications. A carefully designed data layout can significantly improve cache/TLB-hit rates and cacheline utilization.

We suggested starting with the default layout specification (simply by specifying `shape` when creating tensors using `ti.var/Vector/Matrix`), and then migrate to more advanced layouts using the `ti.root.X` syntax.

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.

### 11.1 The default data layout using `shape`

By default, when allocating a `ti.var`, it follows the most naive data layout

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

Or equivalently, the same data layout can be specified using advanced data layout description:

```python
# Create the global tensor
val = ti.var(ti.f32)
# Specify the shape and layout
ti.root.dense(ti.ijk, (32, 64, 128)).place(val)
```

However, oftentimes this data layout is suboptimal for 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.
11.2 Advanced data layout specification

A better layout might be

```python
val = ti.var(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).

11.3 Examples

2D matrix, row-major

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

2D matrix, column-major

```python
A = ti.var(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

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

3D Particle positions and velocities, arrays-of-structures

```python
pos = ti.Vector(3, dt=ti.f32)
vel = ti.Vector(3, dt=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, structures-of-arrays

```python
pos = ti.Vector(3, dt=ti.f32)
vel = ti.Vector(3, dt=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))
```

11.4 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 tensor `A` is stored in row-major order,

```python
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 blocked, the iteration will happen within each block first. This maximizes memory bandwidth utilization in most cases.

Struct-fors on sparse tensors follows the same philosophy, and will be discussed further in *Sparse computation*. 
Sparse computation

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

The DiffTaichi repo contains 10 differentiable physical simulators built with Taichi differentiable programming.

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

```python
@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 has an undefined gradient behavior.

---

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

---

A few examples with neural network controllers optimized using differentiable simulators and brute-force gradient descent:

Documentation WIP.
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:

```python
@ti.data_oriented
class Array2D:
    def __init__(self, n, m, increment):
        self.n = n
        self.m = m
        self.val = ti.var(ti.f32)
        self.total = ti.var(ti.f32)
        self.increment = increment

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

    def place(self, root):
        root.dense(ti.ij, (self.n, self.m)).place(self.val)
        root.place(self.total)

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

@ti.kernel
def inc2(self, increment: ti.i32):
    for i, j in self.val:
```

(continues on next page)
```python
@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.var(ti.f32)

@ti.layout
def place():
    ti.root.place(
        arr)  # Place an object. Make sure you defined place for that obj
ti.root.place(double_total)
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 14. Objective data-oriented programming
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.

Life cycle of a Taichi kernel looks like this:

- Kernel registration
- Template instantiation and caching
- Python AST transforms
- Taichi IR compilation, optimization, and binary generation
- Launching
The Life of a Taichi Kernel

Let’s consider the following simple kernel:

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

We also allocate two 1D tensors to simplify discussion:

```python
x = ti.var(dt=ti.f32, shape=128)
y = ti.var(dt=ti.f32, shape=16)
```

### 15.1 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`.

### 15.2 Template instantiation and caching

```python
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.,
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, therefore a new instantiation and compilation will happen.

**Note:** Many basic operations in the Taichi standard library is implemented using Taichi kernels for performance, with more or less 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.

### 15.3 Code transformation and optimizations

When a new instantiation happens, the Taichi frontend compiler 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
15.4 The just-in-time (JIT) compilation engine

Finally, the optimized SSA IR is fed into the LLVM IR codegen, and LLVM JIT generates high-performance executable CPU/GPU programs.

15.5 Kernel launching

Taichi kernels will be ultimately launched as multi-threaded CPU tasks or CUDA kernels.
TODO: update

16.1 Logging

```python
level can be {}
    ti.TRACE
    ti.DEBUG
    ti.INFO
    ti.WARN
    ti.ERR
    ti.CRITICAL
...
ti.set_logging_level(level)
```

The default logging level is `ti.INFO`. You can also override default logging level by setting the environment variable `TI_LOG_LEVEL` to values such as `trace` and `warn`.

16.2 Trigger GDB when the program crashes:

16.3 Interface System

Print all interfaces and units

```python
ti.core.print_all_units()
```
16.4 Serialization

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.

```c
// 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();
```

16.5 Progress Notification

The taichi messager can send an email to $TI_MONITOR_EMAIL$ when the task finished or crashed. To enable:

```python
from taichi.tools import messager
messager.enable(task_id='test')
```
CHAPTER 17

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 scratch. 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.

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

• Make sure you are using Python 3.6/3.7/3.8

• Execute

```bash
python3 -m pip install --user setuptools astpretty astor pytest opencv-python
python3 -m pip install --user Pillow numpy scipy GitPython yapf colorama psutil
```

• (If on Ubuntu) Execute `sudo apt install libtinfo-dev clang-8`. clang-7 should work as well.

• Make sure you have LLVM 8.0.1 built from scratch (Download). To do so, download and unzip the llvm source, move to the llvm folder, and execute

```bash
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"
made -j 8
sudo make install
```

• Clone the taichi repo, and then
cd taichi
mkdir build
cd build
cmake ..
# if you are building with CUDA, say, 10.0, then please use "cmake .. -DCUDA_  
→VERSION=10.0 -DTI_WITH_CUDA:BOOL=True"
make -j 8

- Add the following to your ~/.bashrc (or ~/.zshrc if you use zsh)

```bash
export TAICHI_REPO_DIR=/home/XXX/taichi  # Path to your taichi repository
export PYTHONPATH=$TAICHI_REPO_DIR/python/:$PYTHONPATH
export PATH=$TAICHI_REPO_DIR/bin/:$PATH
```

- Execute source ~/.bashrc to reload shell config
- Execute ti test to run all the tests. It may take up to 5 minutes to run all tests. (On Windows the ti command should be replaced by python -m taichi)
- Check out examples for runnable examples. Run them with python3.

## 17.1 Setting up CUDA 10.1 on Ubuntu 18.04

First, make sure you have CUDA 10.1 installed. Check this by running nvcc --version or cat /usr/local/  
cuda/version.txt

If you don’t have it - go ahead to this website and download it.

These instructions were copied from the website above for x86_64 architecture

```bash
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/cuda-  
→ubuntu1804.pin
sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
→cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.243-418.87.00/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

## 17.2 Prebuilt LLVM for Windows CI

```bash
cmake .. -G"Visual Studio 15 2017 Win64"  
→DLLVM_ENABLE_RTTI:BOOL=ON -DBUILD_SHARED_  
→LIBS:BOOL=OFF -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD="X86;NVPTX" -DLLVM_  
→ENABLE_ASSERTIONS:ON -Ghost=x64 -DLLVM_BUILD_TESTS:BOOL=OFF -DCMAKE_INSTALL_  
→PREFIX=installed
```

Then use Visual Studio to build. After building the INSTALL project (under folder “CMakePredefinedTargets”). After  
build completes, find your LLVM binaries/headers in build/include.
17.3 Troubleshooting

- Run with debug mode to see if there’s any illegal memory access
- Disable compiler optimizations to quickly confirm that the issue is not cause by optimization
Contribution guidelines

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

- Bug fixes
- New features
- Documentation
- More user-friendly syntax error messages
- New example programs
- Compiler performance patches
- Minor typo fixes in the documentation, code, comments (please directly make a pull request for minor issues like these)

18.1 How to contribute bug fixes and new features

Issues marked with “welcome contribution” are easy ones to start with.

- 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 some one is already working on it. (“Prevent redundant work”)
- If no lead developer has commented and described a potential solution on the issue, please also briefly describe your plan and wait for a lead developer to reply before you start. (“Keep solutions simple”)

18.2 High-level guidelines

- Almost every design decision has pros and cons. Good decisions are good because their pros outweigh their cons. Always think of both sides of your decision.
- No overkils: always use the easiest solutions to solve easy problems, so that you have time and energy for real hard ones.
• Debugging is hard. Changesets should be small so that sources of bugs can be easily pinpointed.
• Unit/integration tests are our friends.

### 18.3 Commit rules

• No commits with local (i.e., the contributor’s local environment) compilation errors should be made;
• Commit messages should be concise and meaningful;
• The master branch is required to have a linear history.

### 18.4 Making good pull requests

• PRs with small changesets are preferred. A PR should ideally address only one issue.
• 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 should always be rebased onto the HEAD of master before merging;
• All PRs should pass continuous integration tests (build + testing for Mac/Windows) before they get merged;
• PR authors should not squash commits. Whether squashing all commits in a PR or not, will be decided by the reviewer who merges the PR into the master branch, depending on how trivially correct the commits are.

### 18.5 Reviewing & PR merging

• Error-prone commits such as IR passes & codegen will be rebased on master (without squashing) once approved;
• Other commits with more trivial correctness (e.g. examples, GUI, benchmark cases, typo fixes) will first get squashed into a single commit and then rebased on master, for a cleaner master commit log.

### 18.6 Tips on Taichi compiler development

*The life of a Taichi kernel* may worth checking out. It explains the whole compilation process.

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_kernel_llvm_ir = True`: print the emitted LLVM IR by Taichi.
• `print_kernel_llvm_ir_optimized = True`: print the optimized LLVM IR for each kernel.
• **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.)

**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`.

### 18.7 Testing

Tests should be added to `taichi/tests/python`. Use `ti test` to run all the tests. Use `ti test_verbose` to test with verbose outputs.

### 18.8 Documentation

Use `ti doc` to build the documentation locally. Open the documentation at `taichi/doc/build/index.html`.

### 18.9 C++ and Python standards

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

### 18.10 (Linux only) pinpointing runtime errors using GDB

A quick way to pinpoint common runtime errors such as segmentation faults/assertion failures. 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.

### 18.11 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.

#### 18.11.1 Folder structure

Key folders are

- **taichi**: The core compiler implementation
  - **analysis**: Static analysis passes
  - **runtime**: Runtime functions
  - **backends**: Code generators
  - **transforms**: IR transform passes
- python_bindings: C++/Python interfaces
  - python: Python frontend implementation
  - examples: Examples
  - docs: Documentation
  - tests: C++ and Python tests
  - benchmarks: Performance benchmarks
  - misc: Random (yet useful) files
  - ...

Chapter 18. Contribution guidelines
19.1 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 `TC`, 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`.

19.2 Dos

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

19.3 Don’ts

- C language legacies:
  - `printf` (use `fmtlib::print` instead).
  - `new` & `free`. Use smart pointers (`std::unique_ptr, std::shared_ptr` instead for ownership management).
- Prefix member functions with `_m_` or `_`.
- Virtual function call in constructors/destructors.
- `NULL`, use `nullptr` instead.
• using namespace std; in global scope.
• typedef. Use using instead.

19.4 Documentation

• To build the documentation: ti doc
CHAPTER 20

Internal designs (WIP)

20.1 Vector type system
20.2 Intermediate representation
20.3 Code generation
Can a user iterate over irregular topology instead of grids, such as tetrahedra meshes, line segment vertices? These structures have to be represented using 1D arrays in Taichi. You can still iterate over it using `for i in x` or `for i in range(n)`. 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 run time cache behaviors and performance numbers.

**Can potential energies be differentiated automatically to get forces?** Yes. Taichi supports automatic differentiation. We do have an example for this.

**Does the compiler backend support the same quality of optimizations for the GPU and CPU?** For instance, if I switch to using the CUDA backend, do I lose the cool hash-table optimizations? Mostly. The CPU/GPU compilation workflow are basically the same, except for vectorization on SIMD CPUs. You still have the hash table optimization on GPUs.
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 23

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 scratch, 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+`.

### 23.1 Ubuntu, Arch Linux, and Mac OS X

```bash
wget https://raw.githubusercontent.com/yuanming-hu/taichi/legacy/install.py
direction install.py
```

Note, if python complains that a package is missing, simply rerun install.py and the package should be loaded.

### 23.2 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`.

59
23.3 Build with Double Precision (64 bit) Float Point

```bash
export TC_USE_DOUBLE=1
ti build
```