

Data-driven Computer Animation

Tutorial 5 – Introduction to Taichi

Prof. Taku Komura

TAs: Zhouyingcheng Liao(zycliao@cs.hku.hk)

What is Taichi

A programming language designed for computer graphics

- Productivity
 - Friendly learning curve
 - Shorter code, higher perf.
- Portability
 - Multi-backend support
- Performance
 - Optimized for bandwidth, locality and load balancing

Why Taichi

- Productivity
- Portability
- Performance

 constraint.cpp

 constraint.h

 constraint_attachment.cpp

 constraint_penalty.cpp

constraint_spring.cpp

 constraint_tet.cpp



```

1199 Eigen::Matrix<T, Nrange, 1> f0 = Eigen::Matrix<T, Nrange, 1>::Zero();
1200
1201 Eigen::Matrix<T, Nrange, 1> result;
1202
1203 for (int j = 0; j < N; j += Nj, j++)
1204 {
1205     result.segment(j, Nj) = f0.segment(j, Nj);
1206     result.segment(j, Nj) += A.segment(j, Nj);
1207 }
1208 }
1209
1210 void Eigen::Matrix<T, Nrange, 1>::setFromVector(const Eigen::Matrix<T, Nrange, 1> &vec, const int from, const int to, const int step)
1211 {
1212     if (Nrange() != 0)
1213     {
1214         A.segment(from, to - from + 1) = vec.segment(from, to - from + 1);
1215     }
1216     else
1217     {
1218     }
1219 }
1220
1221 // returns true if gradient
1222 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1223 {
1224     Eigen::Matrix<T, Nrange, 1> g;
1225     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1226     return g.isZero();
1227 }
1228
1229 // returns true if gradient
1230 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1231 {
1232     Eigen::Matrix<T, Nrange, 1> g;
1233     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1234     return g.isZero();
1235 }
1236
1237 // returns true if
1238 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1239 {
1240     Eigen::Matrix<T, Nrange, 1> g;
1241     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1242     return g.isZero();
1243 }
1244
1245 // returns true if
1246 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1247 {
1248     Eigen::Matrix<T, Nrange, 1> g;
1249     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1250     return g.isZero();
1251 }
1252
1253 // returns true if
1254 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1255 {
1256     Eigen::Matrix<T, Nrange, 1> g;
1257     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1258     return g.isZero();
1259 }
1260
1261 // returns true if
1262 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1263 {
1264     Eigen::Matrix<T, Nrange, 1> g;
1265     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1266     return g.isZero();
1267 }
1268
1269 // returns true if
1270 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1271 {
1272     Eigen::Matrix<T, Nrange, 1> g;
1273     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1274     return g.isZero();
1275 }
1276
1277 // returns true if
1278 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1279 {
1280     Eigen::Matrix<T, Nrange, 1> g;
1281     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1282     return g.isZero();
1283 }
1284
1285 // returns true if
1286 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1287 {
1288     Eigen::Matrix<T, Nrange, 1> g;
1289     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1290     return g.isZero();
1291 }
1292
1293 // returns true if
1294 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1295 {
1296     Eigen::Matrix<T, Nrange, 1> g;
1297     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1298     return g.isZero();
1299 }
1300
1301 // returns true if
1302 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1303 {
1304     Eigen::Matrix<T, Nrange, 1> g;
1305     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1306     return g.isZero();
1307 }
1308
1309 // returns true if
1310 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1311 {
1312     Eigen::Matrix<T, Nrange, 1> g;
1313     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1314     return g.isZero();
1315 }
1316
1317 // returns true if
1318 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1319 {
1320     Eigen::Matrix<T, Nrange, 1> g;
1321     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1322     return g.isZero();
1323 }
1324
1325 // returns true if
1326 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1327 {
1328     Eigen::Matrix<T, Nrange, 1> g;
1329     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1330     return g.isZero();
1331 }
1332
1333 // returns true if
1334 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1335 {
1336     Eigen::Matrix<T, Nrange, 1> g;
1337     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1338     return g.isZero();
1339 }
1340
1341 // returns true if
1342 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1343 {
1344     Eigen::Matrix<T, Nrange, 1> g;
1345     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1346     return g.isZero();
1347 }
1348
1349 // returns true if
1350 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1351 {
1352     Eigen::Matrix<T, Nrange, 1> g;
1353     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1354     return g.isZero();
1355 }
1356
1357 // returns true if
1358 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1359 {
1360     Eigen::Matrix<T, Nrange, 1> g;
1361     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1362     return g.isZero();
1363 }
1364
1365 // returns true if
1366 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1367 {
1368     Eigen::Matrix<T, Nrange, 1> g;
1369     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1370     return g.isZero();
1371 }
1372
1373 // returns true if
1374 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1375 {
1376     Eigen::Matrix<T, Nrange, 1> g;
1377     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1378     return g.isZero();
1379 }
1380
1381 // returns true if
1382 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1383 {
1384     Eigen::Matrix<T, Nrange, 1> g;
1385     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1386     return g.isZero();
1387 }
1388
1389 // returns true if
1390 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1391 {
1392     Eigen::Matrix<T, Nrange, 1> g;
1393     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1394     return g.isZero();
1395 }
1396
1397 // returns true if
1398 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1399 {
1400     Eigen::Matrix<T, Nrange, 1> g;
1401     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1402     return g.isZero();
1403 }
1404
1405 // returns true if
1406 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1407 {
1408     Eigen::Matrix<T, Nrange, 1> g;
1409     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1410     return g.isZero();
1411 }
1412
1413 // returns true if
1414 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1415 {
1416     Eigen::Matrix<T, Nrange, 1> g;
1417     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1418     return g.isZero();
1419 }
1420
1421 // returns true if
1422 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1423 {
1424     Eigen::Matrix<T, Nrange, 1> g;
1425     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1426     return g.isZero();
1427 }
1428
1429 // returns true if
1430 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1431 {
1432     Eigen::Matrix<T, Nrange, 1> g;
1433     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1434     return g.isZero();
1435 }
1436
1437 // returns true if
1438 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1439 {
1440     Eigen::Matrix<T, Nrange, 1> g;
1441     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1442     return g.isZero();
1443 }
1444
1445 // returns true if
1446 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1447 {
1448     Eigen::Matrix<T, Nrange, 1> g;
1449     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1450     return g.isZero();
1451 }
1452
1453 // returns true if
1454 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1455 {
1456     Eigen::Matrix<T, Nrange, 1> g;
1457     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1458     return g.isZero();
1459 }
1460
1461 // returns true if
1462 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1463 {
1464     Eigen::Matrix<T, Nrange, 1> g;
1465     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1466     return g.isZero();
1467 }
1468
1469 // returns true if
1470 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1471 {
1472     Eigen::Matrix<T, Nrange, 1> g;
1473     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1474     return g.isZero();
1475 }
1476
1477 // returns true if
1478 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1479 {
1480     Eigen::Matrix<T, Nrange, 1> g;
1481     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1482     return g.isZero();
1483 }
1484
1485 // returns true if
1486 bool Eigen::Matrix<T, Nrange, 1>::isZero() const
1487 {
1488     Eigen::Matrix<T, Nrange, 1> g;
1489     g = A - Eigen::Matrix<T, Nrange, 1>::Zero();
1490     return g.isZero();
1491 }
1492
149
```



1206
1207 }
1208

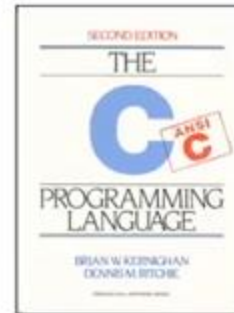
Why Taichi

- Productivity
- Portability
- Performance



Why Taichi

- Productivity
- Portability
- Performance



ti.init()

- Entrance to all Taichi projects

```
• import taichi as ti  
• ti.init(arch=ti.gpu)
```

ti.init()

- arch:
 - ti.cpu/ti.gpu/ti.arm/ti.cuda...

```
• import taichi as ti  
• ti.init(arch=ti.gpu)
```

	CPU	CUDA	OpenGL	Apple Metal	Vulkan
Windows	YES	YES	YES	NO	WIP
Linux	YES	YES	YES	NO	YES
macOS	YES	NO	NO	YES	WIP

ti.init()

- To specify the GPU ID
 - for CUDA: export CUDA_VISIBLE_DEVICES=[gpuid]
 - for Vulkan: export TI_VISIBLE_DEVICES=[gpuid]

	CPU	CUDA	OpenGL	Apple Metal	Vulkan
Windows	YES	YES	YES	NO	WIP
Linux	YES	YES	YES	NO	YES
macOS	YES	NO	NO	YES	WIP

Python vs Taichi

```
import taichi as ti

ti.init(arch=ti.cpu)

d = 1

def foo():
    d_python = d
    print("d_python =", d_python)
```

```
@ti.kernel
def bar():
    d_taichi = d
    print("d_taichi =", d_taichi)
```

```
d = d + 1 # d = 2
foo()    # d_python = 2
bar()    # d_taichi = 2
d = d + 1 # d = 3
foo()    # d_python = 3
bar()    # d_taichi = 2
```

Python-scope

Taichi-scope

Only codes in ti.kernel and ti.func are
in Taichi-scope

Tachi – Data types

- signed integers: ti.i8, ti.i16, ti.i32, ti.i64
- unsigned integers: ti.u8, ti.u16, ti.u32, ti.u64
- floating points: ti.f32, ti.f64

Backend	i8	i16	i32	i64	u8	u16	u32	u64	f16	f32	f64
CPU	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CUDA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OpenGL	✗	✗	✓	○	✗	✗	✗	✗	✗	✓	✓
Metal	✓	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗
Vulkan	○	○	✓	○	○	○	✓	○	✓	✓	○

○: Requiring extensions for the backend.

Data Type Aliases

- Default types can be changed using the configuration option `default_ip` and `default_fp`

```
ti.init(default_ip=ti.i64, default_fp=ti.f64)

@ti.kernel
def example_cast() -> int: # the returned type is ti.i64
    x = 3.14 # x is of ti.f64 type
    y = int(x) # equivalent to ti.i64(x)
    return y
```

Data Type Aliases

- Do not mix up Taichi's int and other int

```
x = numpy.array([1, 2, 3, 4], dtype=int)  # NumPy's int64 type  
y = int(3.14)  # Python's built-in int type
```

Type Casts

- Implicit casts
 - static types **within the Taichi scope**

```
import taichi as ti

ti.init(arch=ti.cpu)

def foo():
    a = 1
    a = 2.7
    print(a)

foo() #2.7
```

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = 1
    a = 2.7
    print(a)

foo() #2
```

Type Casts

- `variable = ti.cast(variable, type)`

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = 1.7
    b = ti.cast(a, ti.i32)
    c = ti.cast(b, ti.f32)
    print("b =", b) # b = 1
    print("c =", c) # c = 1.0

foo()
```

Compound Types

- Using `ti.types` to **create** compound types including:
 - vector / matrix / struct

```
import taichi as ti

ti.init(arch=ti.cpu)

vec3f = ti.types.vector(3, ti.f32)
mat2f = ti.types.matrix(2, 2, ti.f32)
ray = ti.types.struct(ro=vec3f, rd=vec3f, l=ti.f32)

@ti.kernel
def foo():
    a = vec3f(0.0)
    print(a)                # [0.0, 0.0, 0.0]
    d = vec3f(0.0, 1.0, 0.0)
    print(d)                # [0.0, 1.0, 0.0]
    B = mat2f([[1.5, 1.4], [1.3, 1.2]])
    print("B =", B)        # B = [[1.5, 1.4], [1.3, 1.2]]
    r = ray(ro=a, rd=d, l=1)
    print("r.ro =", r.ro)   # r.ro = [0.0, 0.0, 0.0]
    print("r.rd =", r.rd)   # r.rd = [0.0, 1.0, 0.0]

foo()
```

Compound Types

- Predefined keywords for compound types:
 - ti.Vector / ti.Matrix / ti.Struct

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = ti.Vector([0.0, 0.0, 0.0])
    print(a)                # [0.0, 0.0, 0.0]
    d = ti.Vector([0.0, 1.0, 0.0])
    print(d)                # [0.0, 1.0, 0.0]
    B = ti.Matrix([[1.5, 1.4], [1.3, 1.2]])
    print("B =", B)         # B = [[1.5, 1.4], [1.3, 1.2]]
    r = ti.Struct(v1=a, v2=d, l=1)
    print("r.v1 =", r.v1)   # r.v1 = [0.0, 0.0, 0.0]
    print("r.v2 =", r.v2)   # r.v2 = [0.0, 1.0, 0.0]

foo()
```


Compound Types - Indexing

- Access compound elements using [i,j,k,...] indexing

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo():
    a = ti.Vector([1.0, 2.0, 3.0])
    print(a[1]) # 2.0

    B = ti.Matrix([[1.5, 1.4], [1.3, 1.2]])
    print(B[1,0]) # 1.3

foo()
```

ti.field

- “a global N-d array of elements”

```
heat_field = ti.field(dtype=ti.f32, shape=(256, 256))
```

ti.field

- “a **global** **N-d** array of **elements**”
 - **global**: can be read/written from both the Taichi-scope and the Python-scope
 - **N-d**: (Scalar: $N=0$), (Vector: $N=1$), (Matrix: $N=2$), ($N = 3, 4, 5, \dots$)
 - **elements**: scalar, vector, matrix, struct

ti.field

- “a **global** **N-d** array of **elements**”
 - global: can be read/written from both the Taichi-scope and the Python-scope
 - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, ...)
 - elements: scalar, vector, matrix, struct

ti.field

- “a **global** **N-d** array of **elements**”
 - **global**: can be read/written from both the Taichi-scope and the Python-scope
 - **N-d**: (Scalar: $N=0$), (Vector: $N=1$), (Matrix: $N=2$), ($N = 3, 4, 5, \dots$)
 - **elements**: scalar, vector, matrix, struct

ti.field

- “3D gravitational field in a 256x256x128 room”

```
gravitational_field = ti.Vector.field(n = 3, dtype=ti.f32, shape=(256, 256, 128))
```

- “2D strain-tensor field in a 64x64 grid”

```
strain_tensor_field = ti.Matrix.field(n = 2, m = 2, dtype=ti.f32, shape=(64, 64))
```

- “a global scalar that I want to access in a Taichi kernel”

```
global_scalar = ti.field(dtype=ti.f32, shape=())
```

ti.field

- “a **global** **N-d** array of **elements**”
 - global: can be read/written from both the Taichi-scope and the Python-scope
 - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, ...)
 - elements: scalar, vector, matrix, struct
- access elements in a field using [i,j,k,...] indexing

```
import taichi as ti

ti.init(arch=ti.cpu)

pixels = ti.field(dtype=float, shape=(16, 8))

pixels[1, 2] = 42.0
```

```
import taichi as ti

ti.init(arch=ti.cpu)

vf = ti.Vector.field(3, ti.f32, shape=4)

@ti.kernel
def foo():
    v = ti.Vector([1, 2, 3])
    vf[0] = v
```

ti.field

- “a **global** **N-d** array of **elements**”
 - global: can be read/written from both the Taichi-scope and the Python-scope
 - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, ...)
 - elements: scalar, vector, matrix, struct
- access elements in a field using [i,j,k,...] indexing
 - Special case, access a zero-d field using [None]

```
zero_d_scalar = ti.field(ti.f32, shape=())  
zero_d_scalar[None] = 1.5
```

```
zero_d_vec = ti.Vector.field(2, ti.f32, shape=())  
zero_d_vec[None] = ti.Vector([2.5, 2.6])
```


ti.field

- “a **global** **N-d** array of **elements**”
 - global: can be read/written from both the Taichi-scope and the Python-scope
 - N-d: (Scalar: N=0), (Vector: N=1), (Matrix: N=2), (N = 3, 4, 5, ...)
 - elements: scalar, vector, matrix, struct
- access elements in a field using [i,j,k,...] indexing
 - Special case, access a zero-d field using [None]

```
zero_d_scalar = ti.field(ti.f32, shape=())  
zero_d_scalar[None] = 1.5
```

```
zero_d_vec = ti.Vector.field(2, ti.f32, shape=())  
zero_d_vec[None] = ti.Vector([2.5, 2.6])
```

ti.grouped()

- Taichi provides `ti.grouped` syntax which supports grouping loop indices into a `ti.Vector`.
- It enables dimensionality-independent programming, i.e., code are adaptive to scenarios of different dimensionalities automatically

```
# without ti.grouped
for I in ti.ndrange(2, 3):
    print(I)
prints 0, 1, 2, 3, 4, 5
```

```
# with ti.grouped
for I in ti.grouped(ndrange(2, 3)):
    print(I)
prints [0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2]
```

ti.grouped()

- Taichi provides `ti.grouped` syntax which supports grouping loop indices into a `ti.Vector`.
- It enables dimensionality-independent programming, i.e., code are adaptive to scenarios of different dimensionalities automatically

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

a = ti.Matrix.field(n=2, m=3, dtype=ti.f32, shape=(2, 2))
@ti.kernel
def test():
    for i in ti.grouped(a):
        # a[i] is a 2x3 matrix
        a[i] = [[1, 1, 1], [1, 1, 1]]
```

Matrix Size Consideration

- Matrix operations are unrolled at compile time. For performance reasons, it is recommended that you keep your matrices small.

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

a = ti.Matrix.field(n=2, m=3, dtype=ti.f32, shape=(2, 2))
@ti.kernel
def test():
    for i in ti.grouped(a):
        # a[i] is a 2x3 matrix
        a[i] = [[1, 1, 1], [1, 1, 1]]
        # The assignment is unrolled to the following at compile time:
        # a[i][0, 0] = 1
        # a[i][0, 1] = 1
        # a[i][0, 2] = 1
        # a[i][1, 0] = 1
        # a[i][1, 1] = 1
        # a[i][1, 2] = 1
```

Matrix Size Consideration

- Matrix operations are unrolled at compile time. For performance reasons, it is recommended that you keep your matrices small.

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

a = ti.Matrix.field(n=2, m=3, dtype=ti.f32, shape=(2, 2))
@ti.kernel
def test():
    for i in ti.grouped(a):
        # a[i] is a 2x3 matrix
        a[i] = [[1, 1, 1], [1, 1, 1]]
        # The assignment is unrolled to the following at compile time:
        # a[i][0, 0] = 1
        # a[i][0, 1] = 1
        # a[i][0, 2] = 1
        # a[i][1, 0] = 1
        # a[i][1, 1] = 1
        # a[i][1, 2] = 1
```

Matrix Size Consideration

- Workaround: When declaring a matrix field, leave large dimensions to the fields, rather than to the matrices. If you have a 3x2 field of 64x32 matrices:
- Not recommended: `ti.Matrix.field(64, 32, dtype=ti.f32, shape=(3, 2))`
- Recommended: `ti.Matrix.field(3, 2, dtype=ti.f32, shape=(64, 32))`

Computation Kernel

- A Python function decorated by `@ti.kernel` is a Taichi kernel
 - Taichi kernels can only be called **from the Python scope**

```
import taichi as ti
```

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

```
@ti.kernel
```

```
def foo():  
    print("foo")
```

```
@ti.kernel
```

```
def bar():  
    print("bar")
```

```
foo()
```

```
bar()
```

```
import taichi as ti
```

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

```
def foo():  
    print("foo")  
    bar()
```

```
@ti.kernel
```

```
def bar():  
    print("bar")
```

```
foo()
```

```
import taichi as ti
```

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

```
@ti.kernel
```

```
def foo():  
    print("foo")  
    bar()
```

```
@ti.kernel
```

```
def bar():  
    print("bar")
```

```
foo()
```

Loops

- For loops at the **outermost scope** in a Taichi kernel is **automatically parallelized**

```
@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

    for k in range(20): # Parallelized
        z[k] = k
```


Loops

- Outermost scope ?

```
import taichi as ti

ti.init(arch=ti.cpu)

@ti.kernel
def foo(k: ti.i32):
    for i in range(10): # Parallelized :-)
        if k > 42:
            ...

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

Loops

- Design your for loops for best performance

```
def my_for_loop():  
    for i in range(10): # I don't want to parallelize this for  
        for j in range(100): # I want to parallelize this for  
            ...  
  
my_for_loop()
```

```
def my_for_loop():  
    for i in range(10):  
        my_taichi_for()  
  
@ti.kernel  
def my_taichi_for():  
    for j in range(100):  
        ...  
  
my_for_loop()
```



Loops

- break is NOT supported in the parallel for-loops

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

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

Loops

- Race condition
 - Taichi uses += as an atomic add
 - The compiler optimizes for unnecessary atomic operations

```
@ti.kernel
def sum():
    for i in range(10):
        # 1. OK
        total[None] += x[i]

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

        # 3. data race
        total[None] = total[None] + x[i]
```

Loops

- Types of for-loops in Taichi
 - range-for: loops over a **range**, identical to Python range-for
 - struct-for: loops over a **ti.field**, only lives at the outermost scope

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
    for i in range(N):
        x[i] = i

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.Vector.field(2, dtype=ti.i32, shape=(N, N))

@ti.kernel
def foo():
    for i, j in x:
        x[i, j] = ti.Vector([i, j])

foo()
```

Loops

- Types of for-loops in Taichi

- range-for: loops over a **range**, identical to Python range-for
- struct-for: loops over a **ti.field**, only lives at the outermost scope

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.field(dtype=ti.i32, shape=N)

@ti.kernel
def foo():
    for i in range(N):
        x[i] = i

foo()
```

```
import taichi as ti

ti.init(arch=ti.cpu)

N = 10
x = ti.Vector.field(2, dtype=ti.i32, shape=(N,N))

@ti.kernel
def foo():
    for i, j in x:
        x[i, j] = ti.Vector([i, j])

foo()
```

Kernel Arguments

A kernel can accept multiple arguments.

However, it's important to note that you can't pass arbitrary Python objects to a kernel.

Python objects can be dynamic and may contain data that the Taichi compiler cannot recognize.

Kernel Arguments

- Scalars
- `ti.types.matrix()`
- `ti.types.vector()`
- `ti.types.struct()`
- `ti.types.ndarray()`
- ...

Kernel Arguments

- Passed by value
 - `ti.types.matrix()`
 - `ti.types.vector()`
 - `ti.types.struct()`
- Passed by reference
 - `ti.types.ndarray()`
 - `ti.template()`

Kernel Arguments

- Must have type hint

```
transform_type = ti.types.struct(R=ti.math.mat3, T=ti.math.vec3)
pos_type = ti.types.struct(x=ti.math.vec3, trans=transform_type)

@ti.kernel
def kernel_with_nested_struct_arg(p: pos_type) -> ti.math.vec3:
    return p.trans.R @ p.x + p.trans.T

trans = transform_type(ti.math.mat3(1), [1, 1, 1])
p = pos_type(x=[1, 1, 1], trans=trans)
print(kernel_with_nested_struct_arg(p)) # [4., 4., 4.]
```

Kernel Return Value

- Must have the type hint
- Could either be a scalar, `ti.types.matrix()`, or `ti.types.vector()`
- In CPU and CUDA backend, could also be `ti.types.struct()`
- If the return value is a vector or matrix, please ensure that it contains no more than 32 elements. (Warning otherwise)

Kernel Return Value

- At most one return value

```
vec2 = ti.math.vec2

@ti.kernel
def test(x: float, y: float) -> vec2: # Return value must be type hinted
    # Return x, y # Compilation error: Only one return value is allowed
    return vec2(x, y) # Fine
```

- At most one return statement

```
@ti.kernel
def test_sign(x: float) -> float:
    if x >= 0:
        return 1.0
    else:
        return -1.0
# Error: multiple return statements
```

Taichi Function

- Taichi functions are fundamental units of a kernel and can only be called from within a kernel or another Taichi function.

Kernel vs Function

	Kernel	Taichi Function
Call scope	Python scope	Taichi scope
Type hint arguments	Mandatory	Recommended
Type hint return values	Mandatory	Recommended
Return type	<ul style="list-style-type: none">• Scalar• <code>ti.types.matrix()</code>• <code>ti.types.vector()</code>• <code>ti.types.struct()</code> (Only on LLVM-based backends)	<ul style="list-style-type: none">• Scalar• <code>ti.types.matrix()</code>• <code>ti.types.vector()</code>• <code>ti.types.struct()</code>• ...
Maximum number of elements in arguments	<ul style="list-style-type: none">• 32 (OpenGL)• 64 (otherwise)	Unlimited
Maximum number of return values in a return statement	1	Unlimited