DiffTaichi: Differentiable Programming for Physical Simulation

End2end optimization of neural network controllers with gradient descent

Yuanming Hu  MIT CSAIL
内容概览

- Taichi项目简介 (10min)
- DiffTaichi可微编程原理 (ICLR 2020, 20min)
- Tachi与DiffTaichi入门教程 (5min)
- Q&A (10 min)
Two Missions of the Taichi Project

✦ Explore **novel** language abstractions and compilation approaches for visual computing

✦ **Practically** simplify the process of computer graphics development/deployment
The Life of a Taichi Kernel

Python

Kernel Registration (@ti.kernel)

Template Instantiation

Template Inst. Cache

Python AST Transform

Taichi AST Generation & Compile-Time Computation (static if, loop unroll, const fold…)

C++

Taichi Frontend AST IR

AST Lowering

Type Checking

Taichi Hierarchical SSA IR

x86_64

Loop Vectorize

Bound Inference & Scratch Pad Insertion

Simplifications

Reverse Mode Autodiff

(Sparse) Access Lowering

Simplifications

Backend Compiler LLVM (x64/NVPTX)

Kernel Launch

Data Structure Info
Moving Least Squares Material Point Method
Hu, Fang, Ge, Qu, Zhu, Pradhana, Jiang (SIGGRAPH 2018)
Moving Least Squares Material Point Method
Hu, Fang, Ge, Qu, Zhu, Pradhan, Jiang (SIGGRAPH 2018)
Moving Least Squares Material Point Method
Hu, Fang, Ge, Qu, Zhu, Pradhana, Jiang (SIGGRAPH 2018)
Sparse Topology Optimization

Liu, Hu, Zhu, Matusik, Sifakis (SIGGRAPH Asia 2018)
#voxels= 1,040,875,347
Grid resolution= 3000 × 2400 × 1600
Sparse Topology Optimization  Liu, Hu, Zhu, Matusik, Sifakis (SIGGRAPH Asia 2018)
Want High-Resolution?
Want High-Resolution?
Want Performance?

TRIED TO LEARN C++
Performance vs. Productivity

- High-level programming:
  - High performance but low productivity.
- Low-level programming:
  - Low performance but high productivity.
How to get here?
Abstractions that Exploit Domain-Specific Knowledge!
3 million particles simulated with MLS-MPM; rendered with path tracing. Using programs written in *Taichi*. 

Spatial Sparsity:
Regions of interest only occupy a small fraction of the bounding volume.
Particles

1x1x1

4x4x4

16x16x16
In reality...

- **Hash table lookup:** 10s of clock cycles
- **Indirection:** cache/TLB misses
- **Node allocation:** locks, atomics, barriers
- **Branching:** misprediction / warp divergence

Low-level engineering reduces data structure overhead, but harms productivity and couples algorithms and data structures, making it difficult to explore different data structure designs and find the optimal one.
Our Solution:

The Taichi Programming Language

1) **Decouple** computation from **data structures**

2) **Imperative** computation language

3) **Hierarchical** data structure description language

4) Intermediate representation (IR) & data structure access optimizations

5) **Auto parallelization**, memory management, ...

---

**Computational Kernels**

```
Kernel(laplace).def([&]) {
    For(u, [&](Expr i, Expr j){
        auto c = 1.0f / (dx * dx);
        u[i, j] = c * (4 * v[i, j] - v[i+1, j]
                    - v[i-1, j] - v[i, j+1] - v[i, j-1]);
    });
}));
```

2D Laplace operator

**(Sparse) Data Structures**

```
Global(u, f32); Global(v, f32);
layout([&]) {
    auto ij = Indices(0, 1);
    root.dense(ij, {128, 128}).pointer()
        .dense(ij, {8, 8}).place(u, v);
}));
```

1024² sparse grid with 8²

**High-Performance CPU/GPU Kernels**

<table>
<thead>
<tr>
<th>Ours v.s. State-of-the-art:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLS-PM</td>
</tr>
<tr>
<td>FEM Kernel</td>
</tr>
<tr>
<td>MGPCG</td>
</tr>
<tr>
<td>Sparse CNN</td>
</tr>
</tbody>
</table>

**IR & Optimizing Compiler**

**Runtime System**
Defining Computation

Finite Difference Stencil

\[ u_{i,j} = \frac{1}{\Delta x^2} (4v_{i,j} - v_{i+1,j} - v_{i-1,j} - v_{i,j+1} - v_{i,j-1}) \]

Taichi Kernel

```
1 @ti.kernel
2 def laplace():
3     for i, j in u:
4         c = 1 / (dx * dx)
5         u[i, j] = c * (4.0 * v[i, j] - v[i-1, j] - v[i+1, j]
6             - v[i, j-1] - v[i, j+1])
```

- Program on **sparse** data structures as if they are **dense**;
- **Parallel** for-loops (Single-Program-Multiple-Data, like CUDA/ispc);
- Loop over only active elements in the sparse data structure;
- Complex **control flows** (e.g. If, While) supported.
Results

10.0x shorter code
4.55x higher performance

<table>
<thead>
<tr>
<th>High-Performance CPU/GPU Kernels</th>
<th>Ours v.s. State-of-the-art:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLS-MPM</td>
<td>13x shorter code, 1.2x faster</td>
</tr>
<tr>
<td>FEM Kernel</td>
<td>13x shorter code, 14.5x faster</td>
</tr>
<tr>
<td>MGPCG</td>
<td>7x shorter code, 1.9x faster</td>
</tr>
<tr>
<td>Sparse CNN</td>
<td>9x shorter code, 13x faster</td>
</tr>
</tbody>
</table>
The Life of a Taichi Kernel

1. **Kernel Registration (@ti.kernel)**
2. **Template Instantiation**
   - **Template Inst. Cache**
3. **Python AST Transform**
4. **Taichi AST Generation & Compile-Time Computation (static if, loop unroll, const fold...)**
5. **Taichi Frontend AST IR**
   - **AST Lowering**
   - **Type Checking**
   - **Simplifications**
   - **Reverse Mode Autodiff**
   - **(Sparse) Access Lowering**
   - **Simplifications**
6. **Taichi Hierarchical SSA IR**
   - **x86_64**
   - **Loop Vectorize**
   - **GPU**
   - **Bound Inference & Scratch Pad Insertion**
7. **Backend Compiler LLVM (x64/NVPTX)**
8. **Kernel Launch**
9. **Data Structure Info**
Taichi’s Intermediate Representation (IR)

CHI  气

CHI Hierarchical Instructions

「阴阳，气之大者也。」 ——《庄子·则阳》 ~300 B.C.
Optimization-Oriented Intermediate Representation Design

✧ Hierarchical IR
  ○ Keeps loop information
  ○ Static scoping
  ○ Strictly (strongly) & statically typed

✧ Static Single Assignment (SSA)

✧ Progressive lowering. ~70 Instructions in total.
Why can’t traditional compilers do the optimizations?

1) Index analysis
2) Instruction granularity
3) Data access semantics
The Granularity Spectrum

- **Finer**: Taichi IR (CHI)
- **Coarser**: LLVM IR, Machine code

---

**End2end access**

- access1(i,j)
- access2(i,j)

---

**Level-wise Access**

---

**Machine code**

```
movl $0, %eax
addl %eax, %ebx
popl %eax
looptop:
  imul %edx
  andl $0xFF, %eax
cmpl $100, %eax
  jb looptop
  leal 4(%esp), %ebp
  movl %esi, %edi
  subl $8, %edi
  shrl %cl, %ebx
  movw %bx, -2(%ebp)
movl $0, %eax
  addl %eax, %ebx
  popl %eax
  looptop:
  imul %edx
  andl $0xFF, %eax
cmpl $100, %eax
  jb looptop
  leal 4(%esp), %ebp
  movl %esi, %edi
  subl $8, %edi
  shrl %cl, %ebx
  movw %bx, -2(%ebp)
```
Hidden Optimization Opportunities

Analysis Difficulty

End2end access

Level-wise Access

Taichi IR (CHI)

LLVM IR

Machine code

Coarser

Finer
1) data structure abstraction

2) abstraction-specific compiler optimization

3) algorithm data structure decoupling

Taichi:
10.0x shorter code
4.55x higher performance
**DiffTaichi:**
Differentiable Programming on Taichi
(for physical simulation and many other apps)

*End2end optimization of neural network controllers with gradient descent*
Exposure: A White-Box Photo Post-Processing Framework
(TOG 2018)

Yuanming Hu\textsuperscript{1,2} Hao He\textsuperscript{1,2} Chenxi Xu\textsuperscript{1,3} Baoyuan Wang\textsuperscript{1} Stephen Lin\textsuperscript{1}

\textsuperscript{1}Microsoft Research \quad \textsuperscript{2}MIT CSAIL \quad \textsuperscript{3}Peking University

---

**User Rating**

- **Average Photo:** 2.47
- **CycleGAN:** 3.30
- **Human:** 3.37
- **Pix2pix (paired data needed):** 3.43
- **Exposure (ours):** 3.66
- **Human (expert):** 3.66
Exposure:
Learn **image operations**, instead of **pixels**.

- **Differentiable Photo Postprocessing Model**
  - resolution independent
  - content preserving
  - human-understandable

- **Deep Reinforcement Learning**
  - Learn image **operations**, instead of **pixels**

- **Generative Adversarial Networks**
Hand-written CUDA 132x faster than TensorFlow

ChainQueen: Differentiable MLS-MPM
Hu, Liu, Spielberg, Tenenbaum
Freeman, Wu, Rus, Matusik (ICRA 2019)
The Life of a Taichi Kernel

Kernel Registration (@ti.kernel)

Template Instantiation

Template Inst. Cache

Python AST Transform

Taichi AST Generation & Compile-Time Computation (static if, loop unroll, const fold...)

Taichi Frontend AST IR

AST Lowering

Type Checking

Taichi Hierarchical SSA IR

x86_64

Loop Vectorize

GPU

Bound Inference & Scratch Pad Insertion

Simplifications

Reverse Mode Autodiff

(Sparse) Access Lowering

Simplifications

Backend Compiler LLVM (x64/NVPTX)

Kernel Launch

Data Structure Info
Differentiable Programming v.s. Deep Learning: What are they?

\[ L(x) \rightarrow \frac{\partial L}{\partial x} \]

Optimization/Learning via gradient descent!
Differentiable Programming v.s. Deep Learning: What are the differences?

✦ Deep learning operations:
  - convolution, batch normalization, pooling...

✦ Differentiable programming further enables
  - Stencils, gathering/scattering, fine-grained branching and loops...
  - More expressive & higher performance for irregular operations

✦ Granularity
  - Why not TensorFlow/PyTorch?
    - Physical simulator written in TF is 132x slower than CUDA [Hu et al. 2019, ChainQueen]

✦ Reverse-Mode Automatic Differentiation is the key component to differentiable programming
The DiffTaichi Programming Language & Compiler:
Automatic Differentiation for Physical Simulation

Key language designs:

- Differentiable
- Imperative
- Parallel
- Megakernels

4.2x shorter code compared to hand-engineered CUDA.
188x faster than TensorFlow.
Please check out our paper for more details.
Our language allows programmers to easily build differentiable physical modules that work in deep neural networks. The whole program is end-to-end differentiable.
\[ l_{in} = \sum_{\alpha} v_{i\alpha} n_{i\alpha} \]  
(102)

\[ v_{it} = v_i - l_{in} n_i \]  
(103)

\[ l_{it} = \sqrt{\sum_{\alpha} v_{it\alpha}^2 + \varepsilon} \]  
(104)

\[ \hat{v}_{it} = \frac{1}{l_{it}} v_{it} \]  
(105)

\[ l_{it}^* = \max\{l_{it} + c_i \min\{l_{in}, 0\}, 0\} \]  
(106)

\[ v_i^* = l_{it}^* \hat{v}_{it} + \max\{l_{in}, 0\} n_i \]  
(107)

\[ H(x) := [x \geq 0] \]  
(108)

\[ R := l_{it} + c_i \min\{l_{in}, 0\} \]  
(109)
\[ l_{in} = \sum_\alpha v_{i \alpha} n_{i \alpha} \]  
\[ v_{it} = v_i - l_{in} n_i \]  
\[ l_{it} = \sqrt{\sum_\alpha v_{i \alpha}^2 + \varepsilon} \]  
\[ \dot{v}_{it} = \frac{1}{l_{it}} v_{it} \]  
\[ l_{it}^* = \max\{l_{it} + c_i \min\{l_{in}, 0\}, 0\} \]  
\[ v_{i}^* = l_{it}^* \dot{v}_{it} + \max\{l_{in}, 0\} n_i \]  
\[ H(x) := \begin{cases} x \geq 0 \\ 0 \end{cases} \]  
\[ R := l_{it} + c_i \min\{l_{in}, 0\} \]  
\[ \frac{\partial L}{\partial l_{it}^*} = \sum_\alpha \frac{\partial L}{\partial v_{i \alpha}^*} \dot{v}_{i \alpha} \]  
\[ \frac{\partial L}{\partial v_{it}} = \frac{\partial L}{\partial v_{i \alpha}^*} l_{it}^* \]  
\[ \frac{\partial L}{\partial l_{it}} = -\frac{1}{l_{it}^2} \sum_\alpha v_{i \alpha} \frac{\partial L}{\partial \dot{v}_{i \alpha}} + \frac{\partial L}{\partial l_{it}^*} H(R) \]  
\[ \frac{\partial L}{\partial v_{i \alpha}} = \frac{v_{i \alpha}}{l_{it}} \frac{\partial L}{\partial l_{it}} + \frac{1}{l_{it}} \frac{\partial L}{\partial \dot{v}_{i \alpha}} \]  
\[ = \frac{1}{l_{it}} \left[ \frac{\partial L}{\partial l_{it}} v_{i \alpha} + \frac{\partial L}{\partial \dot{v}_{i \alpha}} \right] \]  
\[ \frac{\partial L}{\partial l_{in}} = \left[ \sum_\alpha \frac{\partial L}{\partial v_{i \alpha}} n_{i \alpha} \right] + \frac{\partial L}{\partial l_{it}^*} H(R)c_i H(-l_{in}) + \sum_\alpha H(l_{in}) n_{i \alpha} \frac{\partial L}{\partial v_{i \alpha}^*} \]  
\[ \frac{\partial L}{\partial v_{i \alpha}} = \frac{\partial L}{\partial l_{in}} n_{i \alpha} + \frac{\partial L}{\partial \dot{v}_{i \alpha}} \]
Reverse-Mode Auto Differentiation

\[ y_i = \sin x_i^2 \]

```python
for i in range(0, 16, step 1) do
    // adjoint variables
    %1adj = alloca 0.0
    %2adj = alloca 0.0
    %3adj = alloca 0.0
    // original forward computation
    %1 = load x[i]
    %2 = mul %1, %1
    %3 = sin(%2)
    // reverse accumulation
    %4 = load y_adj[i]
    %3adj += %4
    %5 = cos(%2)
    %2adj += %3adj * %5
    %1adj += 2 * %1 * %2adj
    atomic add x_adj[i], %1adj
end for
```
Two-Scale AutoDiff

DiffSim Python Frontend

Python AST Preprocessing

Taichi Frontend IR Lowering

Type Checking

Tape System

IR Simplification

Flatten If Statements

Eliminate Mutatable Local Var

Make Adjoint (AutoDiff)

IR Simplification

LLVM Codegen: x86/CUDA

Forward Program

```python
with ti.Tape(loss):
    for i in range(3):
        compute_force(i)
        move_partcies(i)
    compute_loss()
```

Tape Contents

- `compute_force, args=(0)`
- `move_partcies, args=(0)`
- `compute_force, args=(1)`
- `move_partcies, args=(1)`
- `compute_force, args=(2)`
- `move_partcies, args=(2)`
- `compute_loss, args=()`

Backward Program

- `compute_loss.grad()`
- `move_partcies.grad(2)`
- `compute_force.grad(2)`
- `move_partcies.grad(1)`
- `compute_force.grad(1)`
- `move_partcies.grad(0)`
- `compute_loss.grad()`
Related Work

Table 3: Comparisons between DiffTaichi and other differentiable programming tools. Note that this table only discusses features related to differentiable physical simulation, and the other tools may not have been designed for this purpose. For example, PyTorch and TensorFlow are designed for classical deep learning tasks and have proven successful in their target domains. Also note that the XLA backend of TensorFlow and JIT feature of PyTorch allow them to fuse operators to some extent, but for simulation we want complete operator fusion within megakernels. “Swift” AD (Wei et al., 2019) is partially implemented as of November 2019. “Julia” refers to Innes et al. (2019).

<table>
<thead>
<tr>
<th>Feature</th>
<th>DiffSim</th>
<th>PyTorch</th>
<th>TensorFlow</th>
<th>Enoki</th>
<th>JAX</th>
<th>Halide</th>
<th>Julia</th>
<th>Swift</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Megakernels</td>
<td>✓</td>
<td>Δ</td>
<td>Δ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Imperative Scheme</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Parallelism</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Flexible Indexing</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Differentiable Elastic Object Simulation

Continuum modeled with both particles and grids. Open-loop controller. 4.2x shorter code than ChainQueen [Hu et al. ICRA 2019]; 188x faster than TensorFlow. 1024 time steps, 80 gradient descent iter. Run time=2min. Red=extension blue=contraction. Reproduce: python3 diffmpm.py
Differentiable Elastic Object Simulation (3D)

30.5K particles, 512 time steps, 40 gradient descent iter. Total run time=4min. Red=extension blue=contraction.

Initial guess

Goal

Reproduce: python3 diffmpm3d.py
Differentiable Elastic Object Simulation (3D)

30.5K particles, 512 time steps, 40 gradient descent iter.
Total run time=4min. Red=extension blue=contraction.

40 iterations

Reproduce: python3 diffmpm3d.py
Differentiable Liquid Simulation (3D)

Couples with elastic objects. 43.5K particles in total, 512 time steps, 450 gradient descent iter. Run time = 45min. Red = extension blue = contraction.

Initial guess

Goal

Reproduce: python3 liquid.py
Differentiable Liquid Simulation (3D)

Couples with elastic objects. 43.5K particles in total, 512 time steps, 450 gradient descent iter. Run time=45min. Red=extension blue=contraction.

Goal

Reproduce: python3 liquid.py
Three mass-spring robots that learn to move. Closed-loop NN controller. Red=extension blue=contraction.
Reproduce: python3 mass_spring.py 1/2/3 train
Differentiable Billiard Simulation

Optimize the **initial position** and **velocity** of the white ball so that the blue ball goes to the black destination.

Reproduce: python3 billiards.py
Differentiable Rigid Body Simulation

Random Initialization

Iteration 20

Two rigid body robots that learn to move. Closed-loop controller. \textcolor{red}{Red}=extension \textcolor{blue}{blue}=contraction.

Reproduce:
\texttt{python3 rigid\_body.py 1/2 train}
Differentiable Incompressible Fluid Simulation

Optimize the initial velocity field so that the ink forms “Taichi” after 100 time steps. 10 Jacobi iterations are applied per time step for incompressibility. Optimized using 200 gradient descent iterations.

Reproduce: python3 smoke.py
Differentiable Water Wave Simulation

<table>
<thead>
<tr>
<th>Center Activation</th>
<th>Iteration 20</th>
<th>Iteration 60</th>
<th>Iteration 180</th>
</tr>
</thead>
</table>

Height field fluid simulation.
Optimize the initial height field so that it forms “Taichi” after 256 time steps.

Reproduce: python3 wave.py
We wrote a differentiable water renderer to simulate refraction. Then we connect the shader with the water wave simulator and VGG16.
Differentiable Water Renderer

The optimization goal is to find an initial water height field, so that after simulation and shading, VGG16 thinks the squirrel image is a goldfish.

Reproduce: python3 water_renderer.py
Differentiable Electric Field Simulation

The eight electrodes (yellow) changes its amount of charge to repulse the red ball, so that it follows the blue dot.

Reproduce: python3 electric.py
Building **Robust** Differentiable Physical Simulators

Differentiating physical simulators does not always yield useful gradients of the physical system being simulated.
How Gradients Go Wrong

Consider this example where a rigid ball hits a friction-less ground.
No gravity, no friction, fully elastic collision.
How Gradients Go Wrong

Consider this example where a rigid ball hits a friction-less ground. No gravity, no friction, fully elastic collision.
How Gradients Go Wrong

Initial height + final height = time \cdot v_y = \text{constant}
How Gradients Go Wrong

Initial height + final height = constant \[ \frac{\partial \text{ final height}}{\partial \text{ initial height}} = -1 \]
But the differentiable simulator may tell you

\[ \frac{\partial \text{final height}}{\partial \text{initial height}} = 1 \]

(instead of -1)
Using a large time step, it is easy to see that the final height actually raises together with the initial height, except for a few discontinuities.
A naive time integrator leads to saw-tooth like this: correct tendency, but completely wrong gradients.

**Question:** how can we get this?
Our Solution:
Precise Time of Impact (TOI)

Initial height + final height = constant

\[ \frac{\partial \text{ final height}}{\partial \text{ initial height}} = -1 \]
After fixing “wrong” gradients, robots now learn much better.
Optimize the Controller (needs gradients)

Optimized without TOI

Optimized with TOI

When gradient needed, without TOI the optimization fails.
Test the Optimized Controller (forward only)

Test environment with TOI  Test environment without TOI

When only forward needed, without TOI the simulator is good enough.
Takeaways:

Differentiating physical simulators does not always yield useful gradients of the physical system being simulated.

A simulation good enough for forward simulation may not be good enough for backpropagation.

Check out our paper for more details on building simulators with robust gradients, and how to use the gradients effectively.
Automatically Computing Forces by Differentiating Potential Energy

\[
- \mathbf{f}_i(\mathbf{x}) = \frac{\partial \Phi}{\partial \hat{x}_i}(\mathbf{\hat{x}})
\]

potential energy

particle force

particle position
fractal.py:
Your First Taichi Program
fractal.py: Your First Taichi Program
```python
import taichi as ti

# Run on GPU by default

@ti.init(arch=ti.cuda)
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):
    for i, j in 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()
```
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):
    for i, j in 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()
import taichi as ti

- Taichi is an embedded domain-specific language (DSL) in Python. It pretends to be a plain Python package.

- Virtually every Python programmer is capable of writing Taichi programs
  - ...after minimal learning efforts
  - also reuse the package management system, Python IDEs, and existing Python packages
Initialize a Taichi program (storage + computational kernels), with optional arguments

- **arch** (automatically fallback to host arch if target not found)
  - `ti.x64` (default)
  - `ti.arm`
  - `ti.cuda`
  - `ti.metal`
  - `ti.opengl`
- **debug**=True/False
- ...

```
# fractal.py

```python
import taichi as ti

# Run on GPU by default

ti.init(arch=ti.cuda)

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):
    for i, j in 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()
```
(Sparse) Tensors

Taichi is a data-oriented programming language, where dense or spatially-sparse tensors are first-class citizens.

- $\texttt{pixels} = \texttt{ti.var(dt=ti.f32, shape=(n \times 2, n))}$ allocates a 2D dense tensor named pixel of size $(640, 320)$ and type $\texttt{ti.f32}$ (i.e. float in C).
```python
# 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):
    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()
```
Kernels

- Computation happens within Taichi kernels.
- Kernel arguments must be type-hinted
  - The language used in Taichi kernels and functions looks exactly like Python
  - The Taichi frontend compiler converts it into a language that is compiled, statically-typed, lexically-scoped, parallel, and differentiable.
Functions

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

All function calls are force-inlined

```python
@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):
    ...
    z = complex_sqr(z) + c
    ...
```
Taichi-scope v.s. 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-scopes is simply native Python code.
# fractal.py

```python
import taichi as ti
# 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):
    for i, j in 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()
```
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()
```

Performance Tip:
Accessing single elements is slow. Use [from/to]_[numpy/torch] as much as possible!
Calling Taichi kernels...

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

gui = ti.GUI("Fractal", (n * 2, n))
for i in range(1000000):
    paint(i * 0.03)  # as if it is a Python function!
gui.set_image(pixels)
gui.show()
```
Linear Algebra

- `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 “@”

- Other useful functions:
  - `ti.transposed(A), A.T()`
  - `ti.inverse(A)`
  - `ti.Matrix.abs(A)`
  - `ti.trace(A)`
  - `ti.determinant(A, type)`
  - `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)
Differentiable programming

✧ 10 examples at https://github.com/yuanming-hu/difftaichi
pip3 install taichi

Taichi is currently being developed by the Taichi community

前端语法 中间表示优化 编译器后端 文档翻译... 欢迎加入我们!
GAMES 201
Advanced Physics Engines 2020: A Hands-on Tutorial

Yuanming Hu  胡渊鸣  MIT CSAIL
麻省理工学院  计算机科学与人工智能实验室

(Taichi Programming Language)
高级物理引擎实战2020

✦ 课程目标：自己动手打造影视级物理引擎
✦ 适合人群：0-99岁的计算机图形学爱好者
✦ 预备知识：高等数学、Python或任何一门程序设计语言
✦ 课程安排：每周一北京时间晚上20:30-21:30 共10节课

✦ 课程内容：Taichi语言基础 刚体 液体 烟雾 弹塑性体 PIC/FLIP法 Krylov-子空间求解器 预条件 无矩阵法 多重网格 弱形式与有限元 隐式积分器 辛积分器 拓扑优化 带符号距离场 自由表面追踪 物质点法 大规模物理效果渲染 现代处理器微架构 内存层级 并行编程 GPU编程 稀疏数据结构 可微编程…

2020年六一儿童节开课 不见不散!
Questions are welcome!