Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, Fredo Durand

(ICLR 2020)

DiffTaichi: Differentiable Programming for Physical Simulation

End2end optimization of neural network controllers with gradient descent

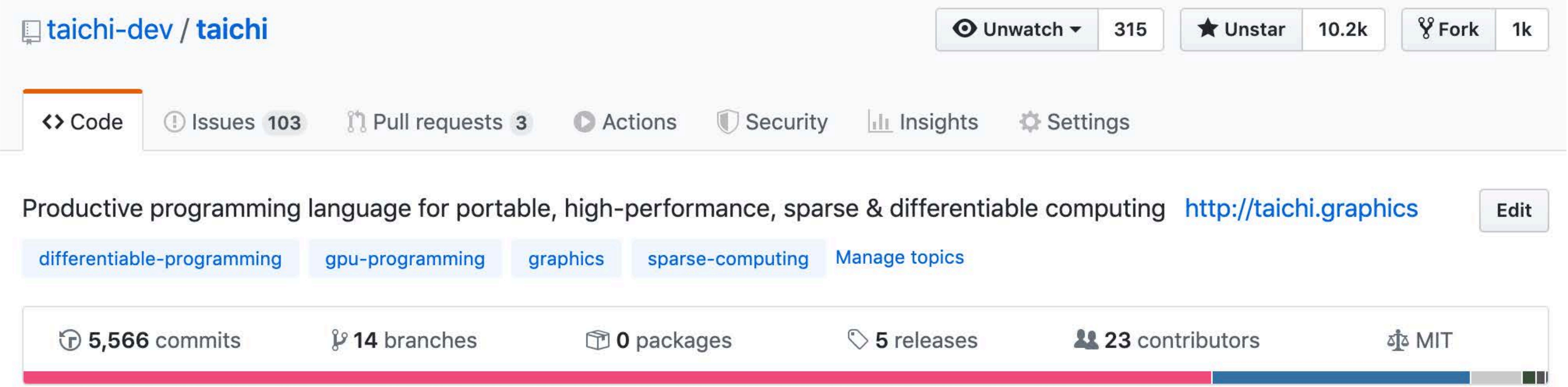
Yuanming Hu MIT CSAIL

内容概览

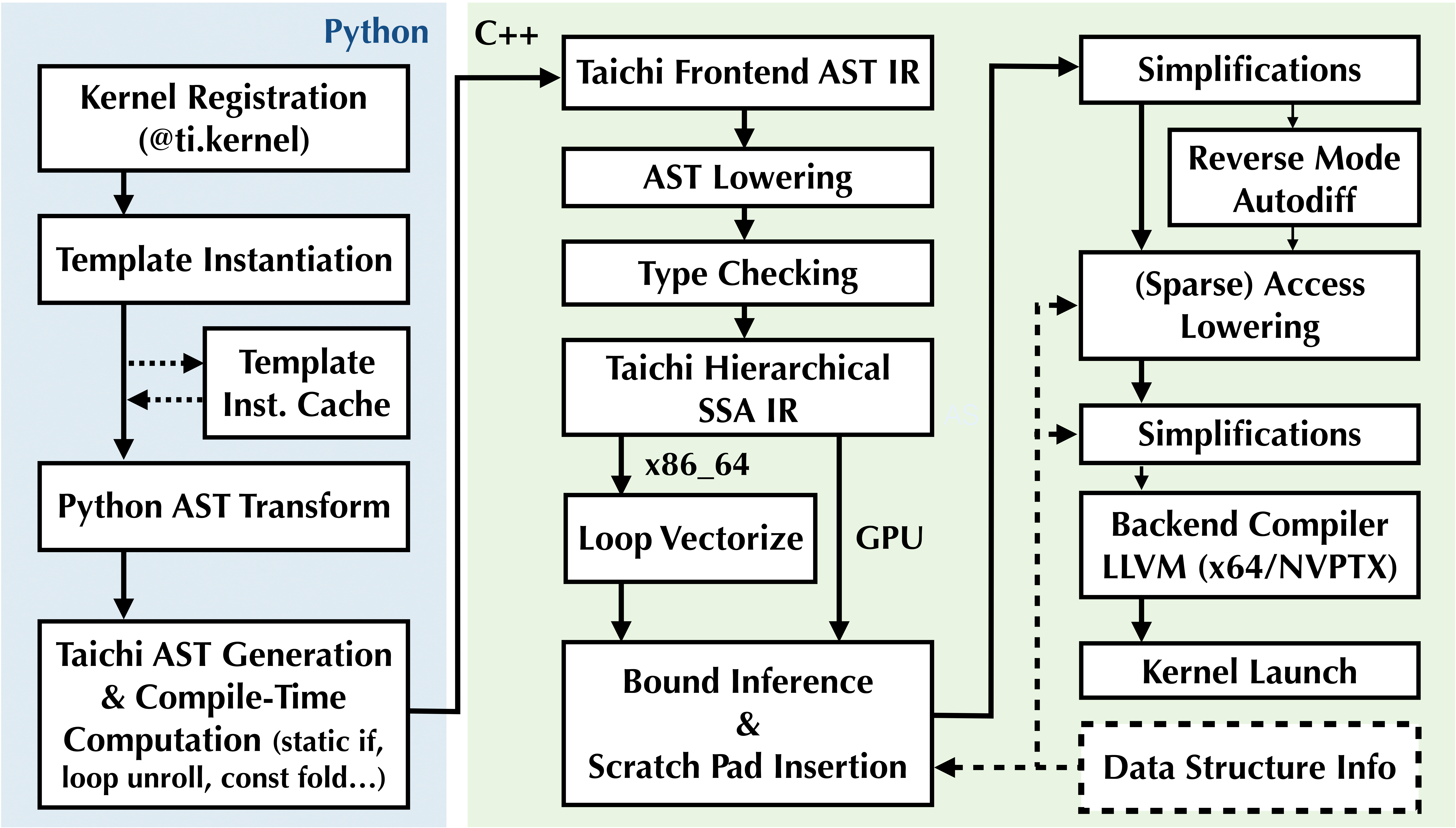
- **Taichi项目简介 (10min)**
- **DiffTaichi可微编程原理(ICLR 2020, 20min)**
- **Tachi与DiffTaichi入门教程 (5min)**
- **Q&A (10 min)**

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



Moving Least Squares Material Point Method

Hu, Fang, Ge, Qu, Zhu, Pradhana, Jiang (SIGGRAPH 2018)

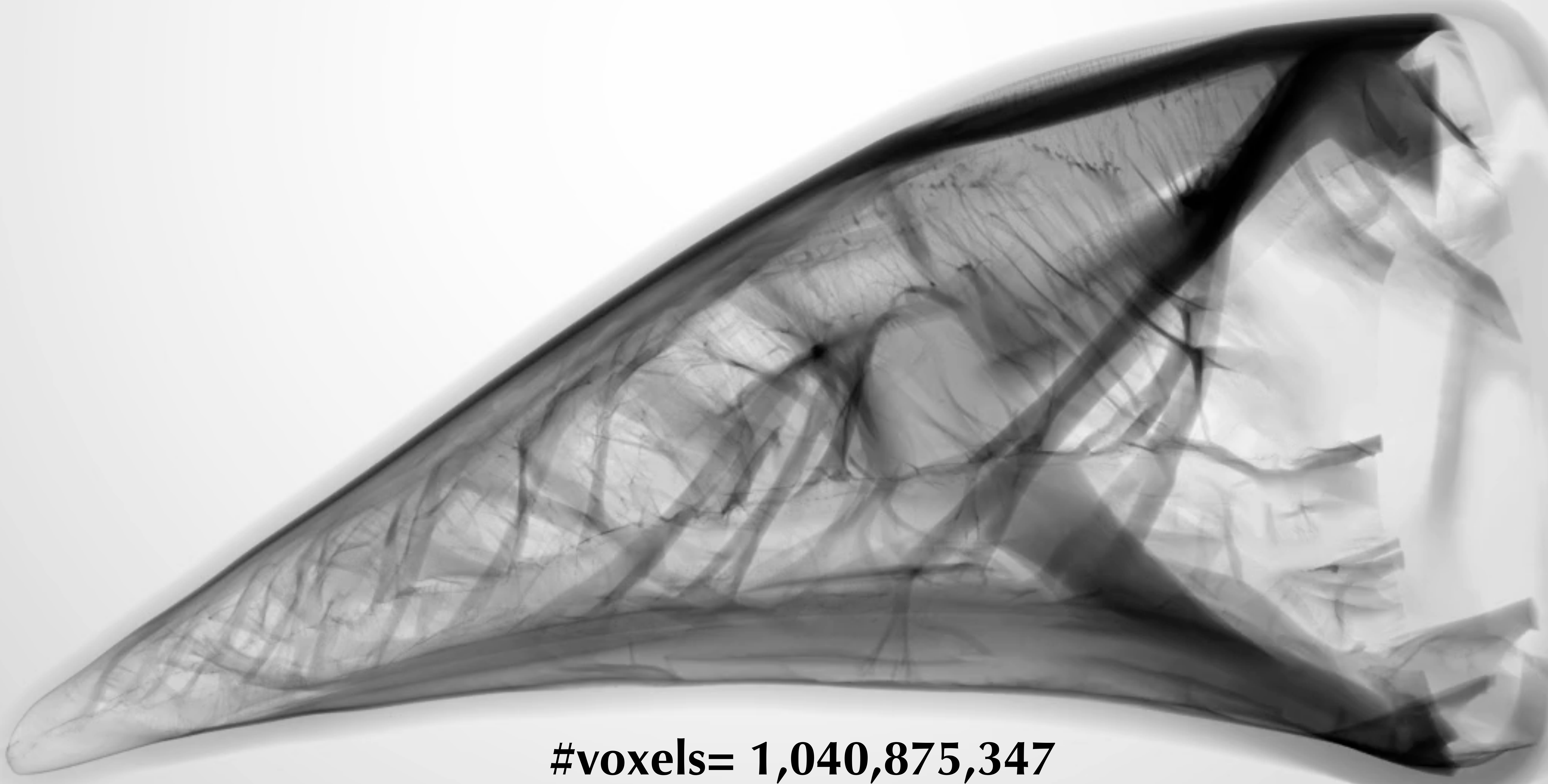
Moving Least Squares Material Point Method

Hu, Fang, Ge, Qu, Zhu, Pradhana, 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

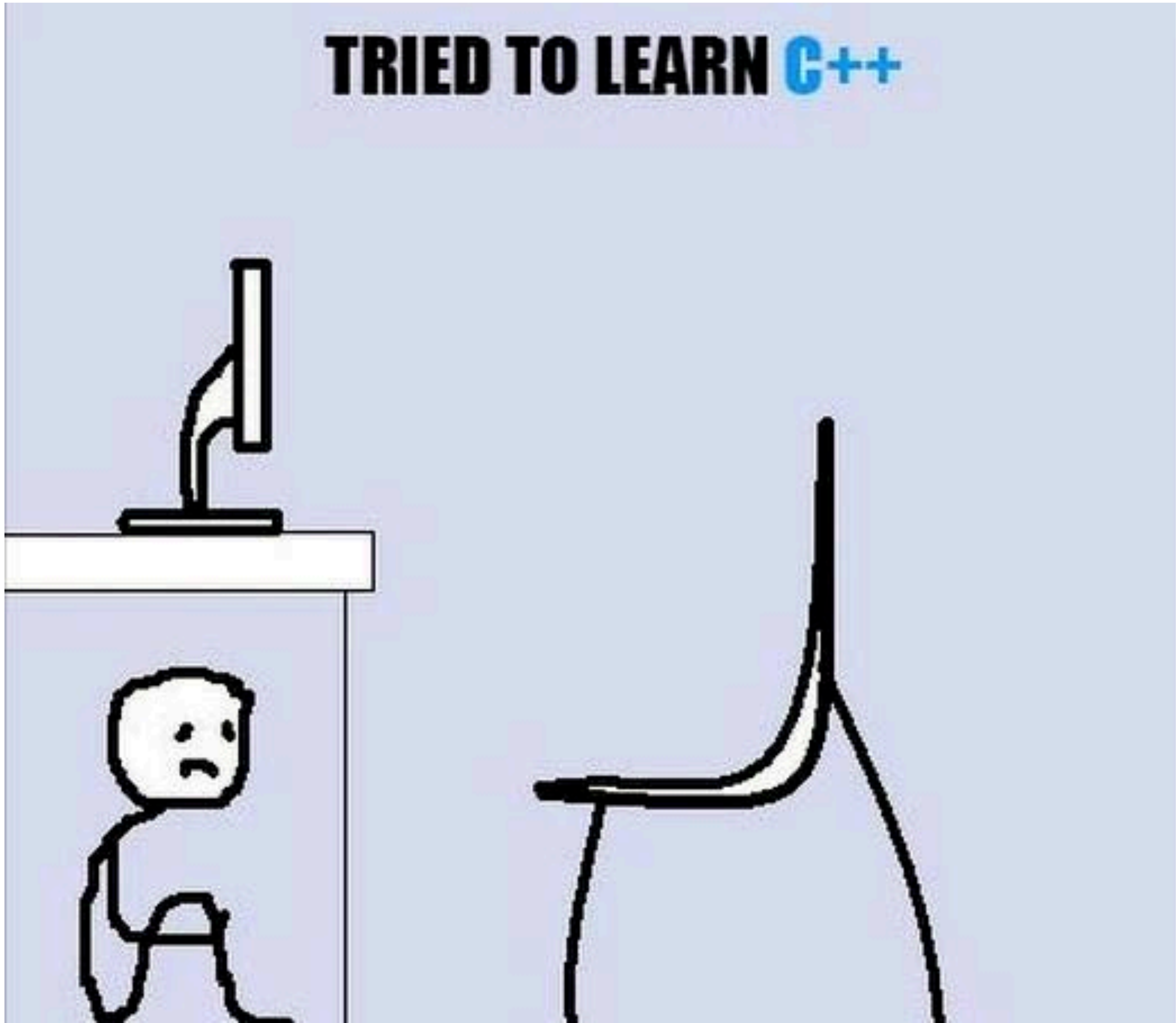
Grid resolution= 3000 × 2400 × 1600

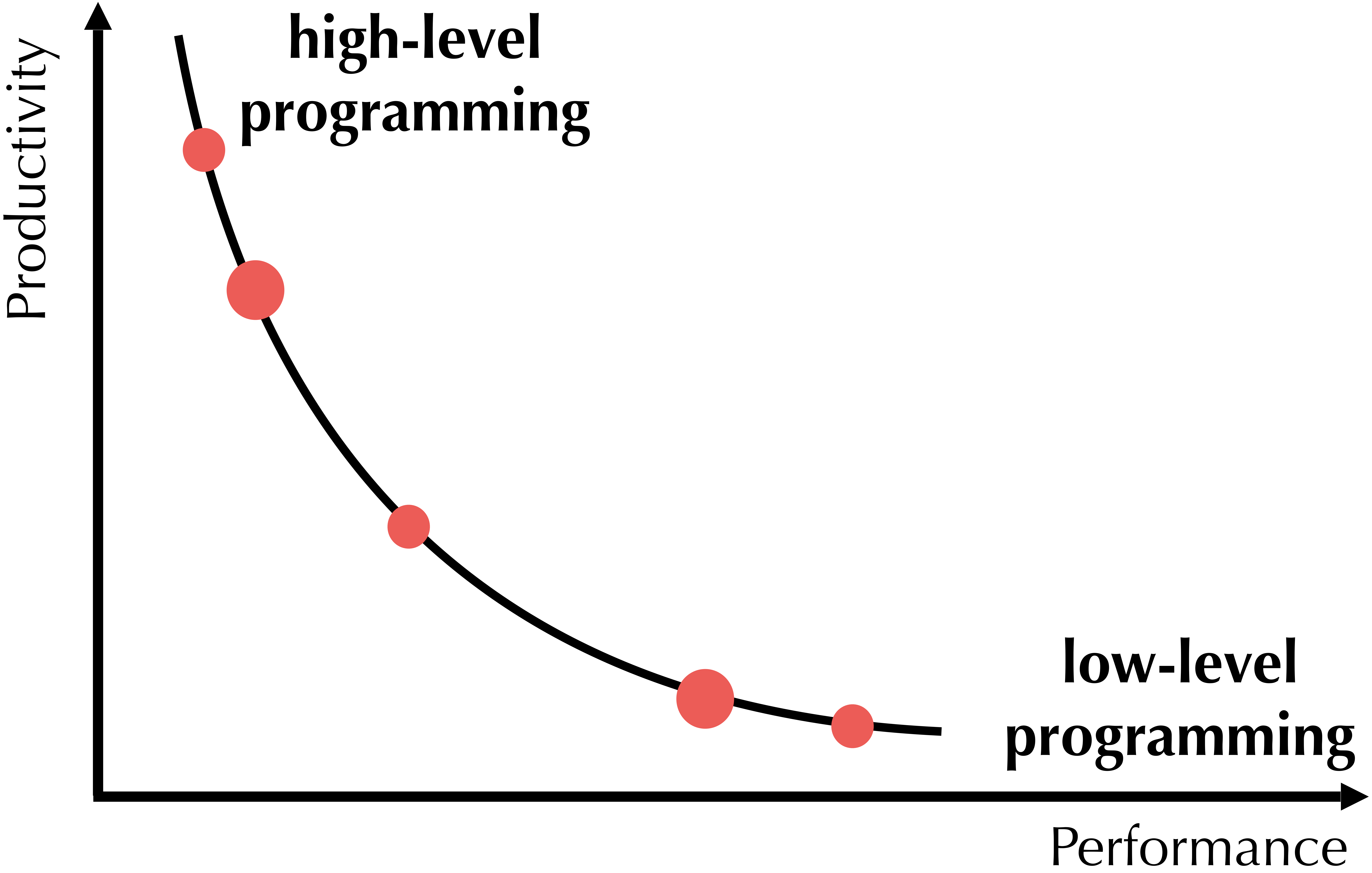
Sparse Topology Optimization *Liu, **Hu,** Zhu, Matusik, Sifakis (SIGGRAPH Asia 2018)*

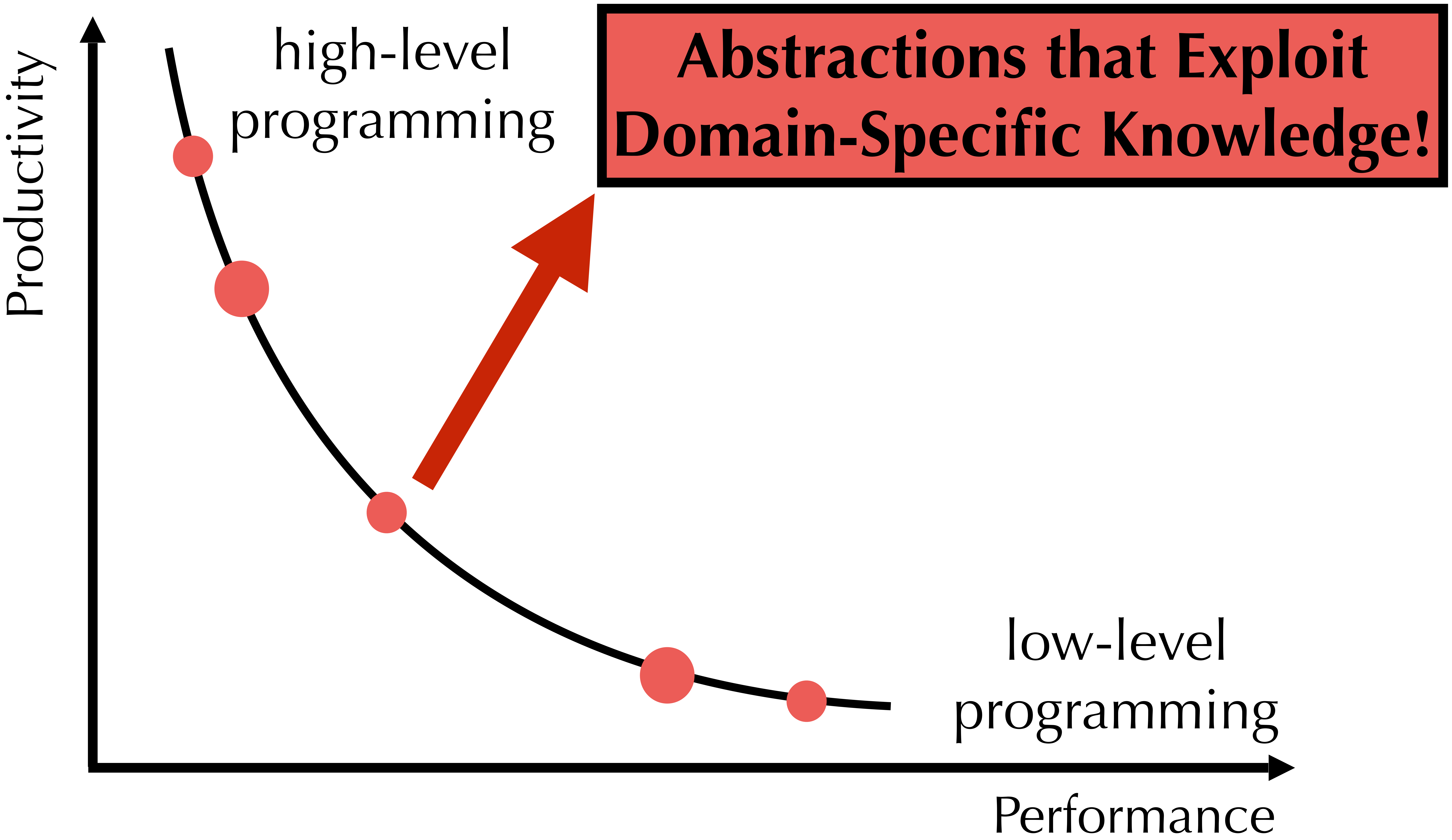
Want High-Resolution?

Want High-Resolution?

Want Performance?




high-level programming
low-level programming
Productivity
Performance



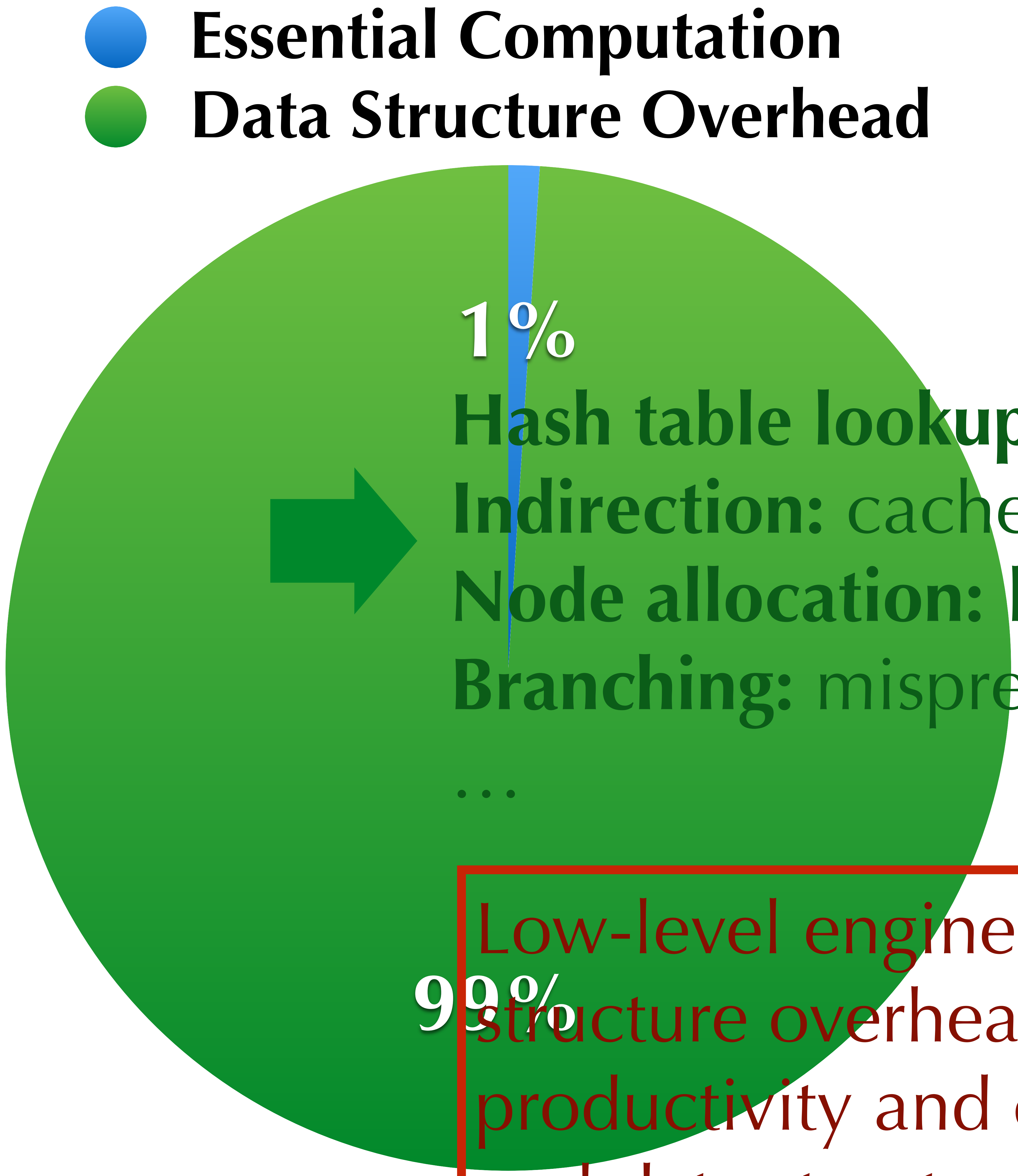
Abstractions that Exploit Domain-Specific Knowledge!
high-level programming
low-level programming
Productivity
Performance


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



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.

In reality…

Our Solution:

The Taichi Programming Language

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

10x shorter code, 4.55x faster

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);
});
```

1024^2 sparse grid with 8^2

IR & Optimizing Compiler

High-Performance CPU/GPU Kernels

Ours v.s. State-of-the-art:

MLS-MPM	13x shorter code, 1.2x faster
FEM Kernel	13x shorter code, 14.5x faster
MGPCG	7x shorter code, 1.9x faster
Sparse CNN	9x shorter code, 13x faster

Runtime System

2) **Imperative** computation language

3) **Hierarchical** data structure description language

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

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

Defining Computation

Finite Difference Stencil

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

$$\downarrow$$

Taichi Kernel

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


Sample/pixel/sec: 7.126
depth_limit: 20
density_scale: 400.000
max_density: 724.000
ground_y: 0.029
light_phi: 0.419
light_theta: 0.218
light_smoothness: 0.050
light_ambient: 0.150
exposure: 0.567
gamma: 0.500
file_id: 0
output_samples: 10
grid_level: 1
video_step: 1
Save
Render All

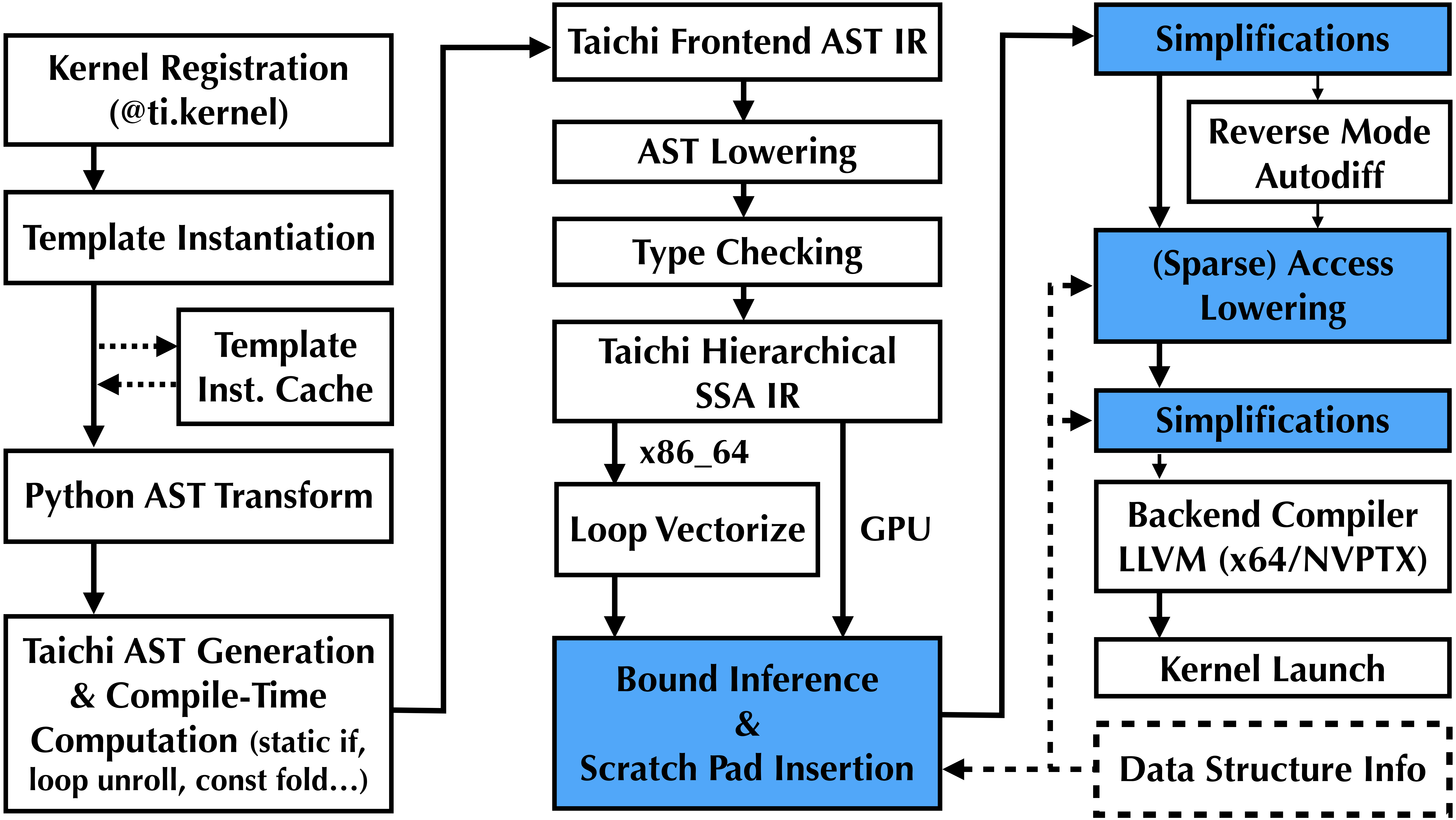
Results

10.0x shorter code
4.55x higher performance

High-Performance CPU/GPU Kernels
Ours v.s. State-of-the-art:

MLS-MPM	13x shorter code, 1.2x faster
FEM Kernel	13x shorter code, 14.5x faster
MGPCG	7x shorter code, 1.9x faster
Sparse CNN	9x shorter code, 13x faster

The Life of a Taichi Kernel



Taichi's Intermediate Representation (IR)

CHI 气

CHI **H**ierarchical **I**nstructions

「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

```
x[i, j]
```

End2end access

```
access1(i,j)
access2(i,j)
```

Level-wise Access

```
$4 = [S4][root]::lookup(root, $3) coord = {$2}
     activate = false
$5 = get child [S4->S3] $4
$6 = bit_extract($2 + 0, 7~14)
$7 = linearized(ind {$6}, stride {128})
$8 = [S3][dense]::lookup($5, $7) coord = {$2} a
      = false
$9 = get child [S3->S2] $8
$10 = bit_extract($2 + 0, 0~7)
$11 = linearized(ind {$10}, stride {128})
$12 = [S2][dense]::lookup($9, $11) coord = {$2}
     activate = false
```

Taichi IR (CHI)

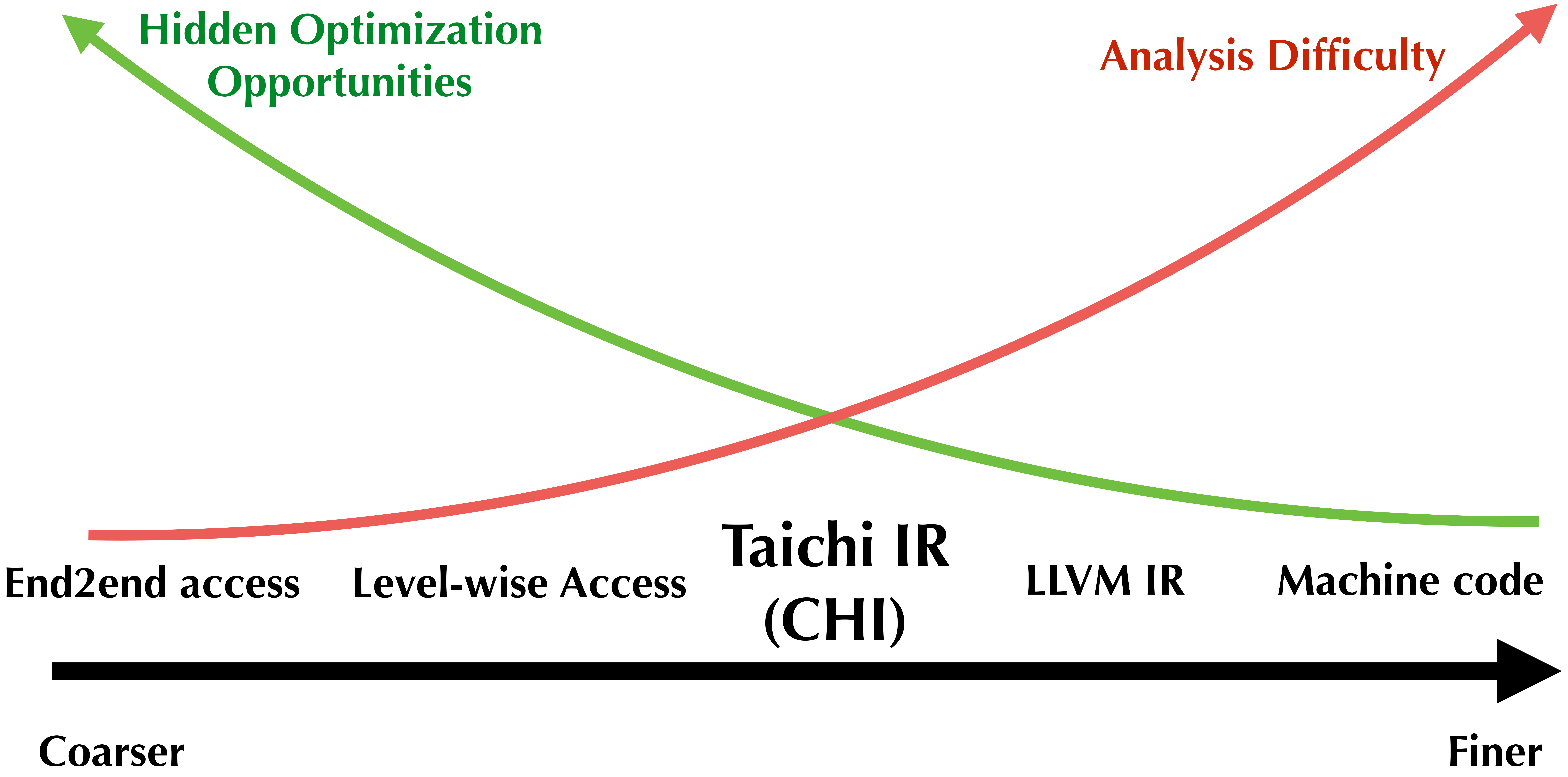
```
%63 = lshr i32 %62, 0
%64 = and i32 %63, 255
%65 = add i32 %37, 0
%66 = lshr i32 %65, 0
%67 = and i32 %66, 255
%68 = add i32 0, %64
%69 = mul i32 %68, 256
%70 = add i32 %69, %67
%71 = bitcast %struct.DenseMeta* %5 t
call void @StructMeta_set_snode_id(%s
call void @StructMeta_set_element_siz
call void @StructMeta_set_max_num_ele
call void @StructMeta_set_lookup_elem
_element)
call void @StructMeta_set_is_active(%
call void @StructMeta_set_get_num_ele
lements)
call void @StructMeta_set_from_parent_
```

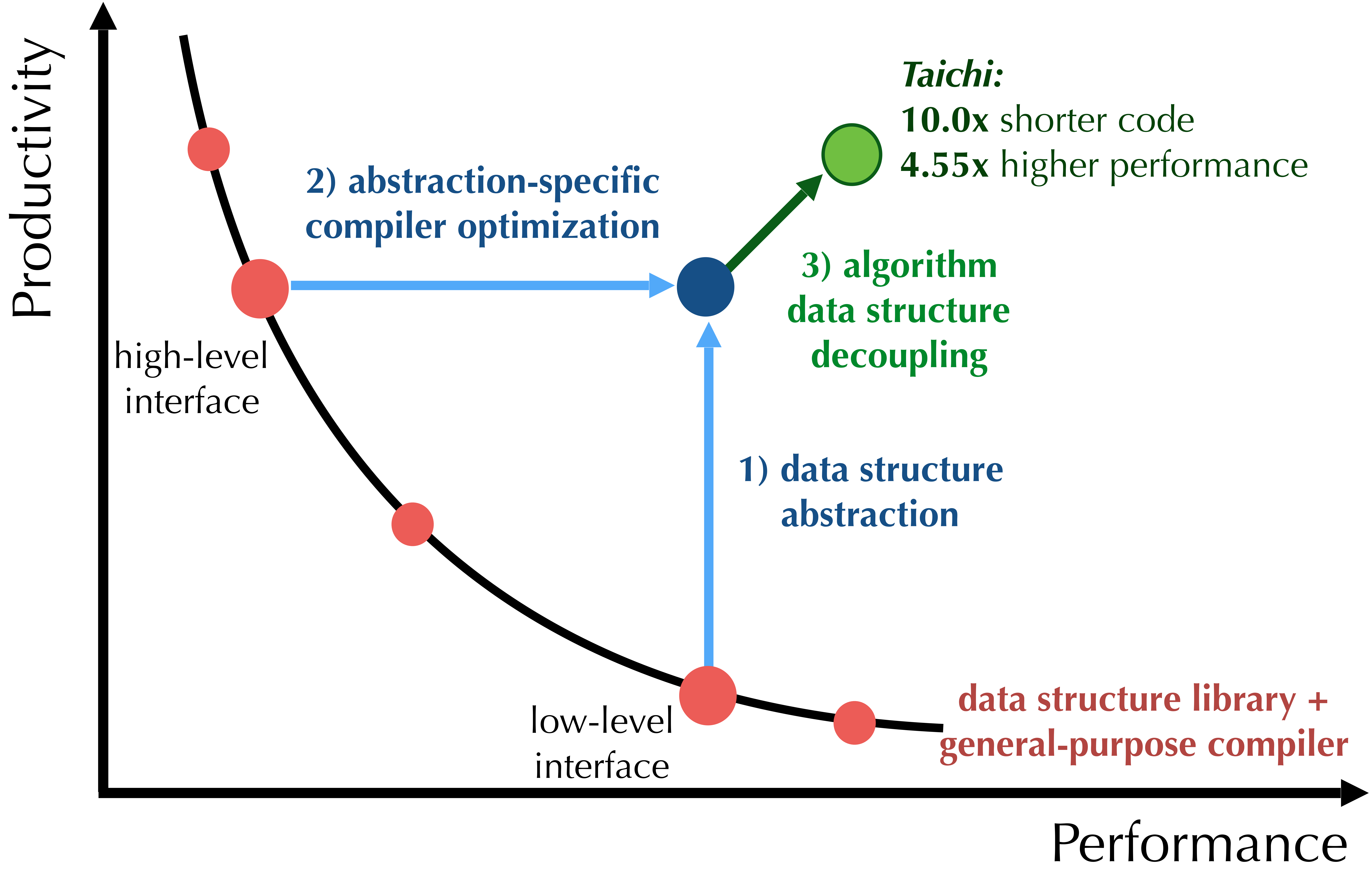
LLVM IR

```
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)
```

Machine code

Coarser → Finer


Hidden Optimization Opportunities
Analysis Difficulty
End2end access
Level-wise Access
Taichi IR (CHI)
LLVM IR
Machine code
Coarser
Finer


Productivity
Performance
high-level interface
low-level interface
2) abstraction-specific compiler optimization
1) data structure abstraction
3) algorithm data structure decoupling
Taichi:
10.0x shorter code
4.55x higher performance
data structure library + general-purpose compiler

Hu, *Anderson, Li, Sun, Carr, Ragan-Kelley, Durand (ICLR 2020)*

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[31] (TOG 2018)

Yuanming Hu[1,2] Hao He[1,2] Chenxi Xu[1,3] Baoyuan Wang[1] Stephen Lin[1]
[1]Microsoft Research [2]MIT CSAIL [3]Peking University

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**

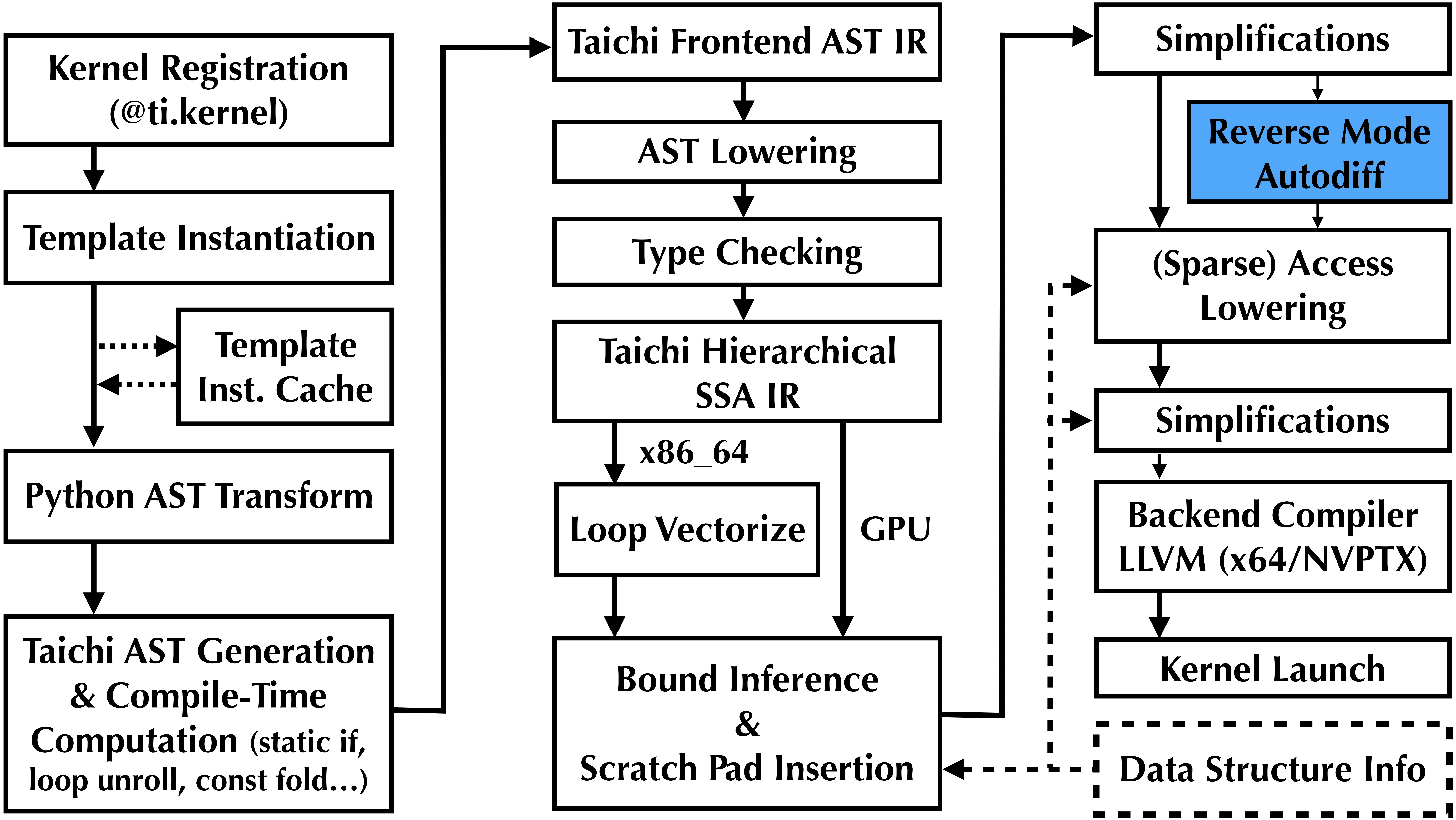
Generative Adversarial Networks

Modelling

Optimization


Iteration 58
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



Differentiable Programming v.s. **Deep Learning:** What are they?

$$L(x) \longrightarrow \frac{\partial L}{\partial x}$$

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**

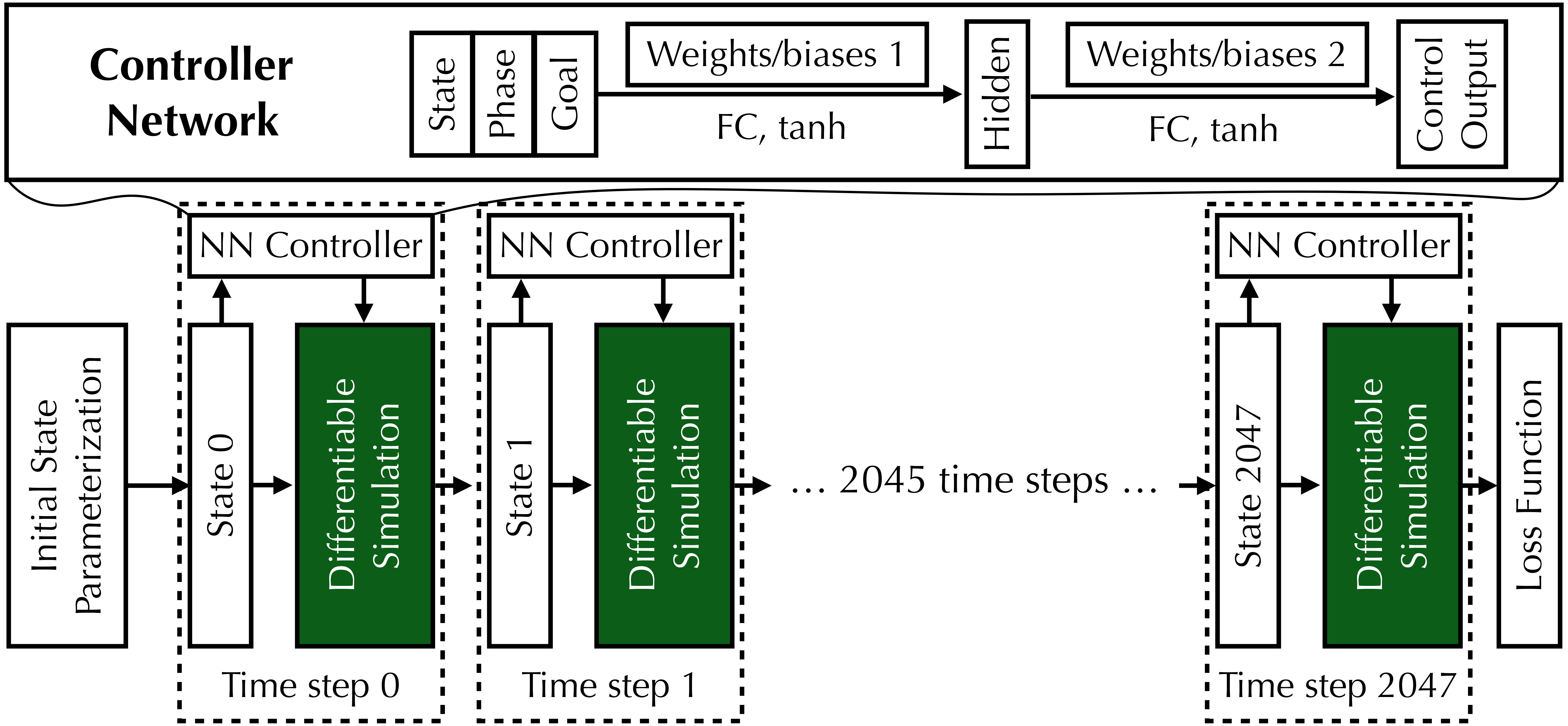
The DiffTaichi Programming Language & Compiler: Automatic Differentiation for Physical Simulation

Key language designs:

- **Differentiable**
- **Imperative**
- **Parallel**
- **Megakernels**

```
@ti.kernel
def apply_spring_force(t: ti.i32):
  # Kernels can have parameters. Here t is a parameter with type int32.
  for i in range(n_springs): # A parallel for, preferably on GPU
    a, b = spring_anchor_a[i], spring_anchor_b[i]
    x_a, x_b = x[t - 1, a], x[t - 1, b]
    dist = x_a - x_b
    length = dist.norm() + 1e-4
    F = (length - spring_length[i]) * spring_stiffness * dist / length
    # Apply spring impulses to mass points. Use atomic_add for parallel safety.
    ti.atomic_add(force[t, a],  -F)
    ti.atomic_add(force[t, b],  F)
```

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**.

$$
\begin{aligned}
l_{i\mathbf{n}} &= \sum_\alpha \mathbf{v}_{i\alpha}\mathbf{n}_{i\alpha} && (102)\\
\mathbf{v}_{i\mathbf{t}} &= \mathbf{v}_i - l_{i\mathbf{n}}\mathbf{n}_i && (103)\\
l_{i\mathbf{t}} &= \sqrt{\sum_\alpha \mathbf{v}_{i\mathbf{t}\alpha}^2 + \varepsilon} && (104)\\
\hat{\mathbf{v}}_{i\mathbf{t}} &= \frac{1}{l_{i\mathbf{t}}}\mathbf{v}_{i\mathbf{t}} && (105)\\
l_{i\mathbf{t}}^* &= \max\{l_{i\mathbf{t}} + c_i \min\{l_{i\mathbf{n}}, 0\}, 0\} && (106)\\
\mathbf{v}_i^* &= l_{i\mathbf{t}}^*\hat{\mathbf{v}}_{i\mathbf{t}} + \max\{l_{i\mathbf{n}}, 0\}\mathbf{n}_i && (107)\\
H(x) &:= [x \geq 0] && (108)\\
R &:= l_{i\mathbf{t}} + c_i \min\{l_{i\mathbf{n},0}\} && (109)
\end{aligned}
$$

$$
\begin{aligned}
l_{\mathbf{in}} &= \sum_\alpha \mathbf{v}_{i\alpha}\mathbf{n}_{i\alpha} && (102)\\
\mathbf{v}_{\mathbf{it}} &= \mathbf{v}_i - l_{\mathbf{in}}\mathbf{n}_i && (103)\\
l_{\mathbf{it}} &= \sqrt{\sum_\alpha \mathbf{v}_{\mathbf{it}\alpha}^2 + \varepsilon} && (104)\\
\hat{\mathbf{v}}_{\mathbf{it}} &= \frac{1}{l_{\mathbf{it}}}\mathbf{v}_{\mathbf{it}} && (105)\\
l_{\mathbf{it}}^* &= \max\{l_{\mathbf{it}} + c_i \min\{l_{\mathbf{in}}, 0\}, 0\} && (106)\\
\mathbf{v}_i^* &= l_{\mathbf{it}}^* \hat{\mathbf{v}}_{\mathbf{it}} + \max\{l_{\mathbf{in}}, 0\}\mathbf{n}_i && (107)\\
H(x) &:= [x \geq 0] && (108)\\
R &:= l_{\mathbf{it}} + c_i \min\{l_{\mathbf{in},0}\} && (109)\\
\implies \frac{\partial L}{\partial l_{\mathbf{it}}^*} &= \sum_\alpha \frac{\partial L}{\partial \mathbf{v}_{i\alpha}^*}\hat{\mathbf{v}}_{\mathbf{it}\alpha} && (110)\\
\frac{\partial L}{\partial \hat{\mathbf{v}}_{\mathbf{it}}} &= \frac{\partial L}{\partial \mathbf{v}_{i\alpha}^*} l_{\mathbf{it}}^* && (111)\\
\frac{\partial L}{\partial l_{\mathbf{it}}} &= -\frac{1}{l_{\mathbf{it}}^2}\sum_\alpha \mathbf{v}_{\mathbf{it}\alpha}\frac{\partial L}{\partial \hat{\mathbf{v}}_{\mathbf{it}\alpha}} + \frac{\partial L}{\partial l_{\mathbf{it}}^*}H(R) && (112)\\
\frac{\partial L}{\partial \mathbf{v}_{\mathbf{it}\alpha}} &= \frac{\mathbf{v}_{\mathbf{it}\alpha}}{l_{\mathbf{it}}}\frac{\partial L}{\partial l_{\mathbf{it}}} + \frac{1}{l_{\mathbf{it}}}\frac{\partial L}{\partial \hat{\mathbf{v}}_{\mathbf{it}\alpha}} && (113)\\
&= \frac{1}{l_{\mathbf{it}}}\left[\frac{\partial L}{\partial l_{\mathbf{it}}}\mathbf{v}_{\mathbf{it}\alpha} + \frac{\partial L}{\partial \hat{\mathbf{v}}_{\mathbf{it}\alpha}}\right] && (114)\\
\frac{\partial L}{\partial l_{\mathbf{in}}} &= -\left[\sum_\alpha \frac{\partial L}{\partial \mathbf{v}_{\mathbf{it}\alpha}}\mathbf{n}_{i\alpha}\right] + \frac{\partial L}{\partial l_{\mathbf{it}}^*}H(R)c_i H(-l_{\mathbf{in}}) + \sum_\alpha H(l_{\mathbf{in}})\mathbf{n}_{i\alpha}\frac{\partial L}{\partial \mathbf{v}_{i\alpha}^*} && (115)\\
\frac{\partial L}{\partial \mathbf{v}_{i\alpha}} &= \frac{\partial L}{\partial l_{\mathbf{in}}}\mathbf{n}_{i\alpha} + \frac{\partial L}{\partial \mathbf{v}_{\mathbf{it}\alpha}} && (116)\\
& && (117)
\end{aligned}
$$

Reverse-Mode Auto Differentiation

$$y_i = \sin x_i^2$$

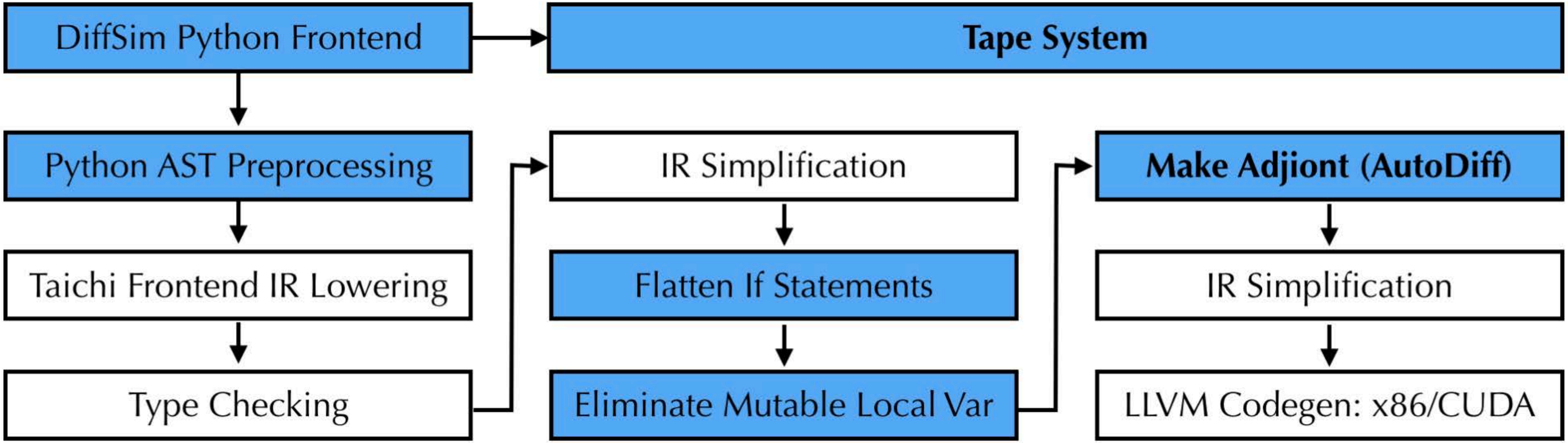
---

for i ∈ range(0, 16, step 1) **do**
  $\%1 = \text{load } x[i]$
  $\%2 = \text{mul } \%1, \%1$
  $\%3 = \sin(\%2)$
  store $y[i] = \%3$
end for

---

for i in range(0, 16, step 1) **do**
  // adjoint variables
  $\%1adj = \text{alloca } 0.0$
  $\%2adj = \text{alloca } 0.0$
  $\%3adj = \text{alloca } 0.0$
  // original forward computation
  $\%1 = \text{load } x[i]$
  $\%2 = \text{mul } \%1, \%1$
  $\%3 = sin(\%2)$
  // reverse accumulation
  $\%4 = \text{load } y\_adj[i]$
  $\%3adj \mathrel{+}= \%4$
  $\%5 = cos(\%2)$
  $\%2adj \mathrel{+}= \%3adj * \%5$
  $\%1adj \mathrel{+}= 2 * \%1 * \%2adj$
  atomic add $x\_adj[i], \%1adj$
end for

Two-Scale AutoDiff



Forward Program

```
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_force.grad(0)
```

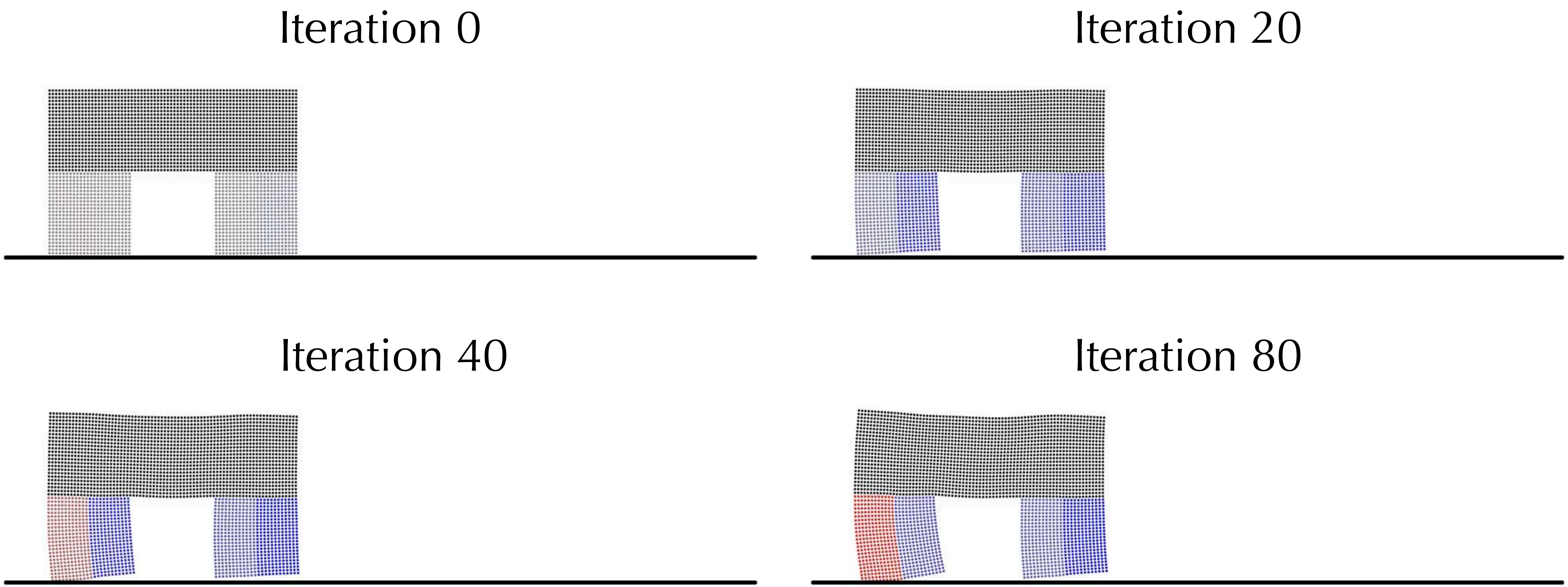
Related Work

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).

(DiffSim=DiffTaichi)

Feature	DiffSim	PyTorch	TensorFlow	Enoki	JAX	Halide	Julia	Swift
GPU Megakernels	✓	Δ	Δ	✓	✓	✓		
Imperative Scheme	✓			✓			✓	✓
Parallelism	✓	✓	✓	✓	✓	✓		
Flexible Indexing	✓					✓	✓	✓

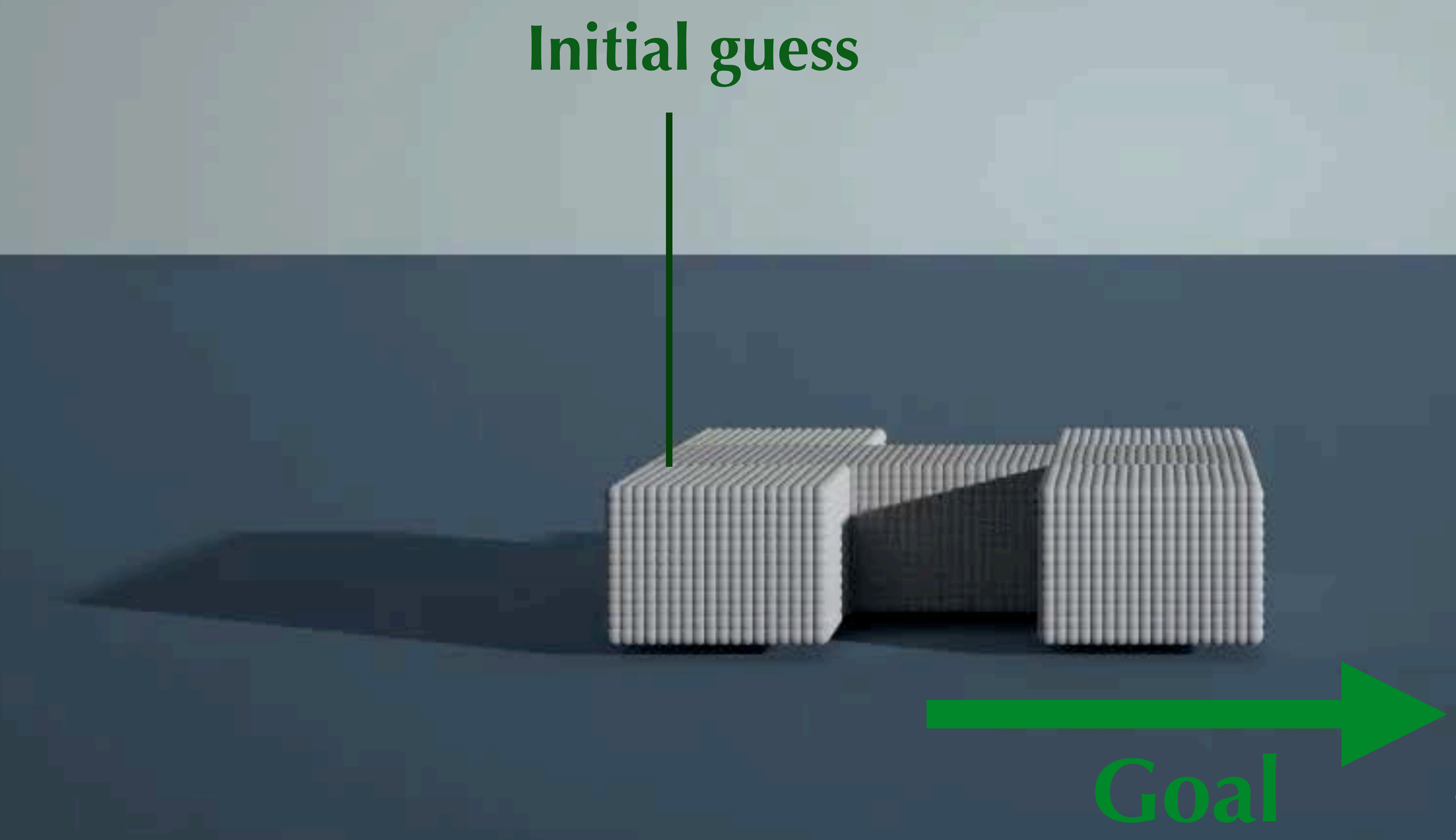
Differentiable Elastic Object Simulation



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)

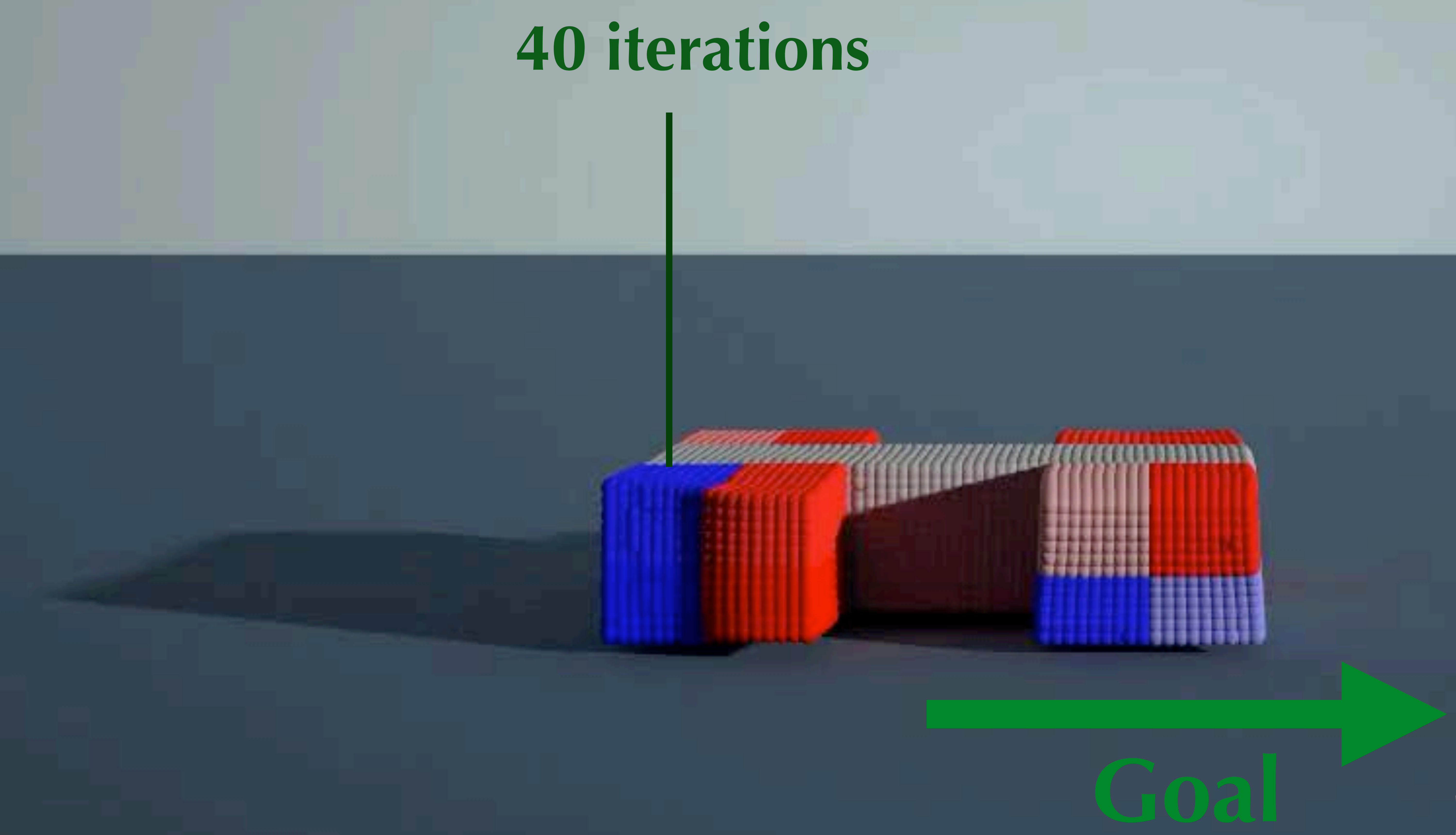
30.5K particles, 512 time steps, 40 gradient descent iter.
Total run time=4min. Red=extension blue=contraction.



Reproduce: `python3 diffmpm3d.py`

Differentiable Elastic Object Simulation (3D)

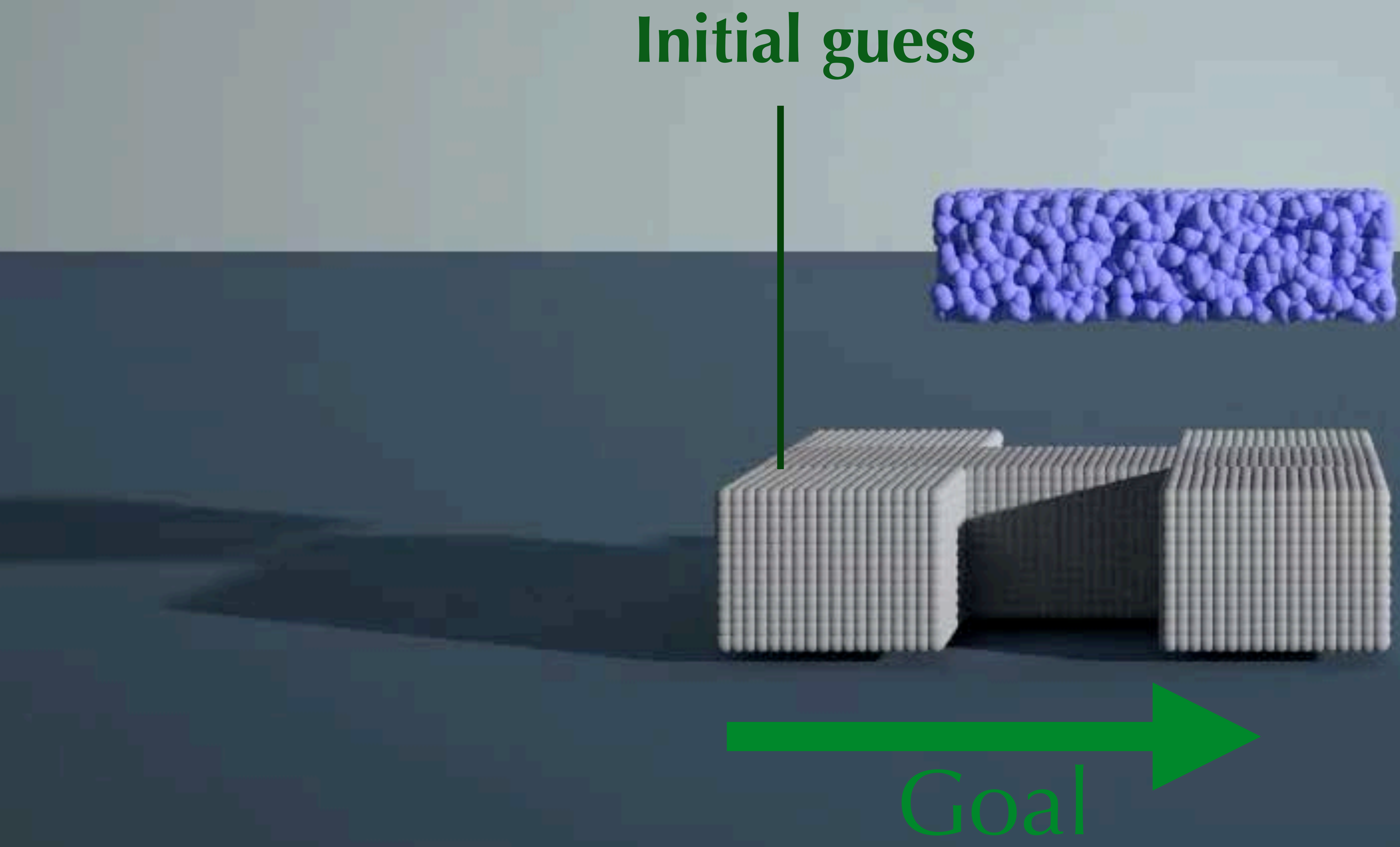
30.5K particles, 512 time steps, 40 gradient descent iter.
Total run time=4min. Red=extension blue=contraction.



Reproduce: `python3 diffmpm3d.py`

Differentiable Liquid Simulation (3D)

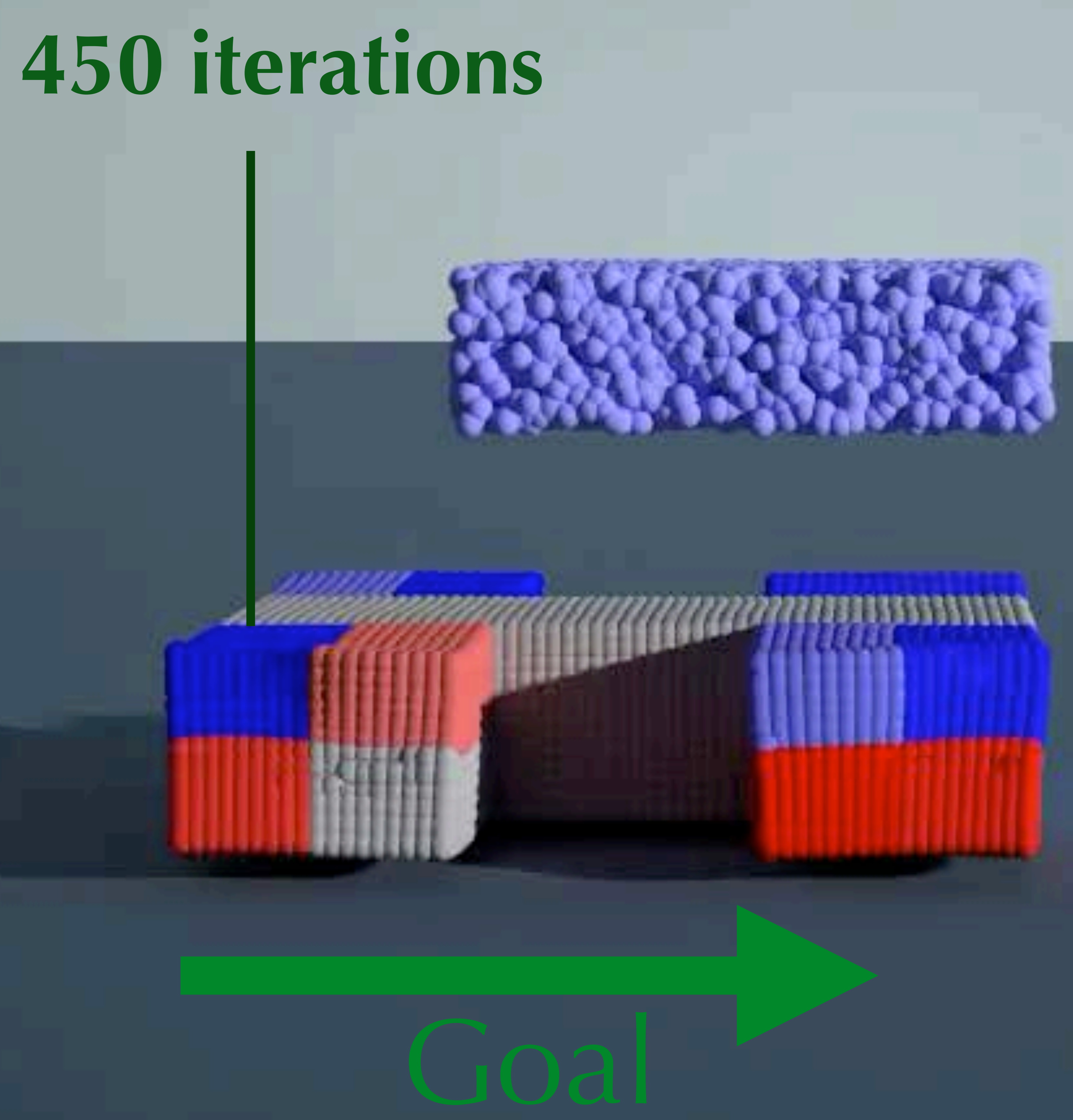
Couples with elastic objects. 43.5K particles in total, 512 time steps, 450 gradient descent iter.
Run time=45min. Red=extension blue=contraction.



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.



Reproduce: `python3 liquid.py`

Differentiable Mass-Spring Simulation

Random Initialization

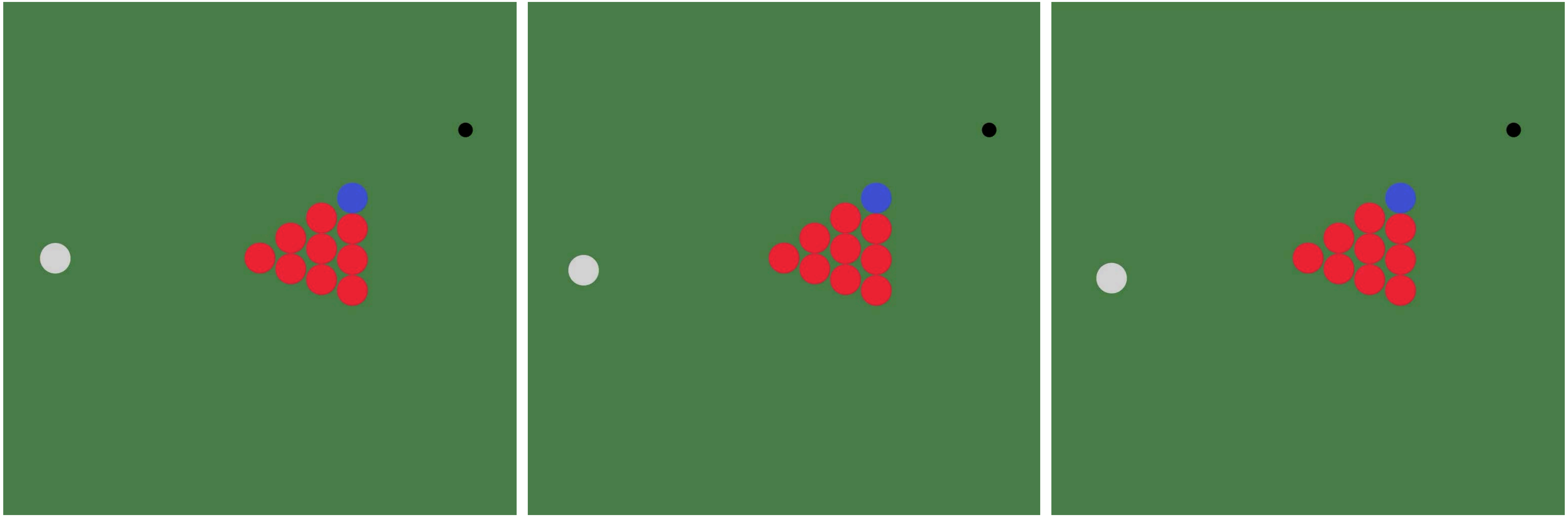
Iteration 100

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

iter. 0 **iter. 40** **iter. 100**

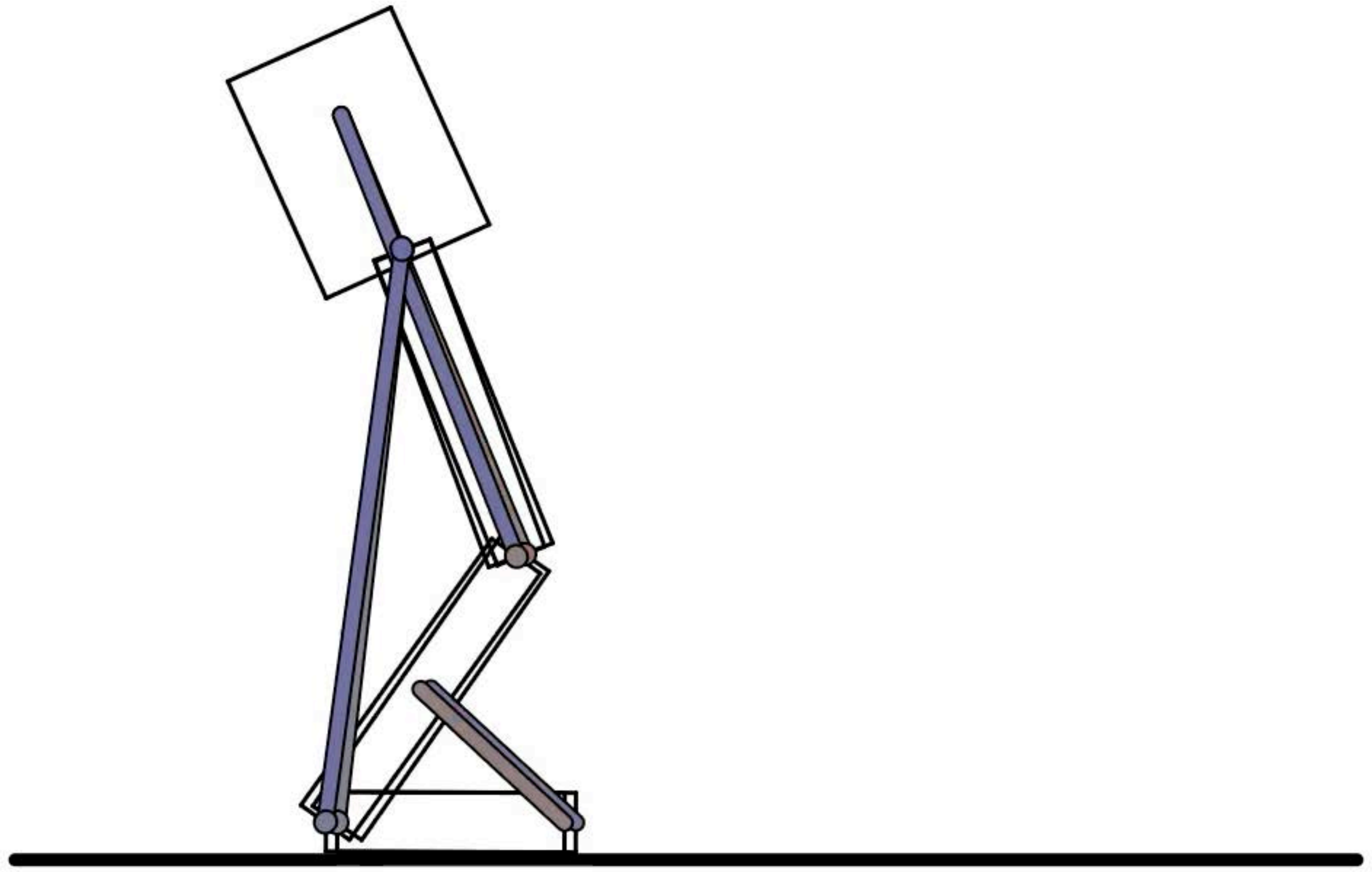


Optimize the **initial position** and **velocity** of the white ball so that the blue ball goes to the black destination

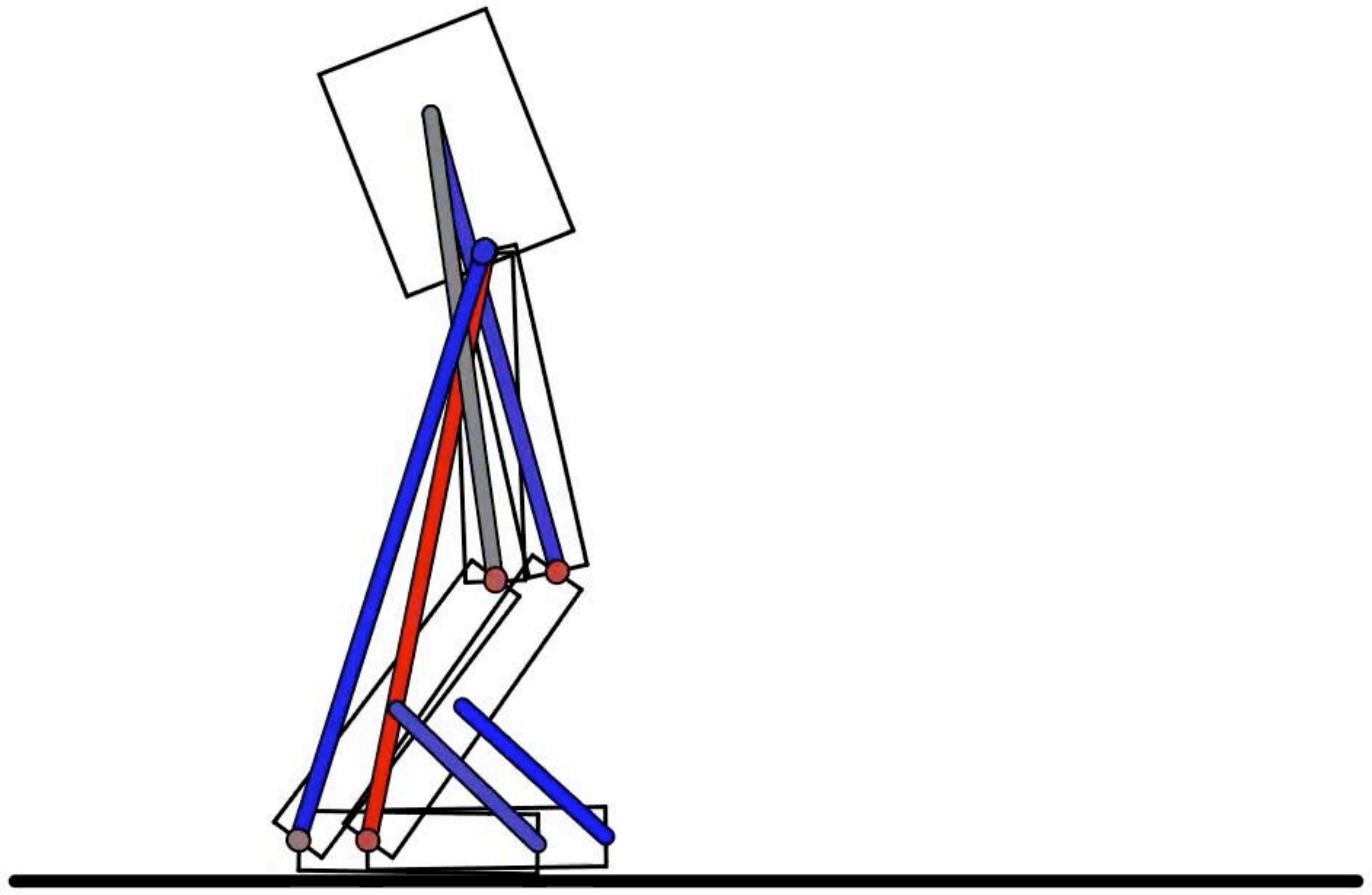
Reproduce: `python3 billiards.py`

Differentiable Rigid Body Simulation

Random Initialization



Iteration 20



Two rigid body robots that learn to move.

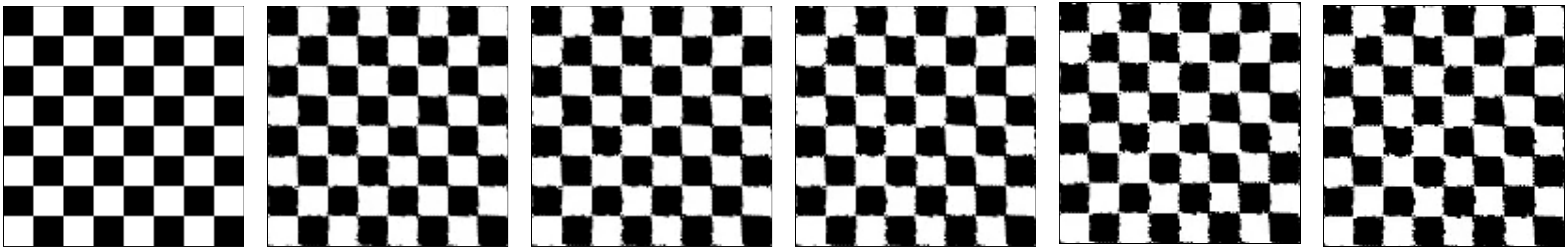
Closed-loop controller. Red=extension blue=contraction.

Reproduce:

`python3 rigid_body.py 1/2 train`

Differentiable Incompressible Fluid Simulation

iter. 0 **iter. 10** **iter. 30** **iter. 60** **iter. 120** **iter. 200**

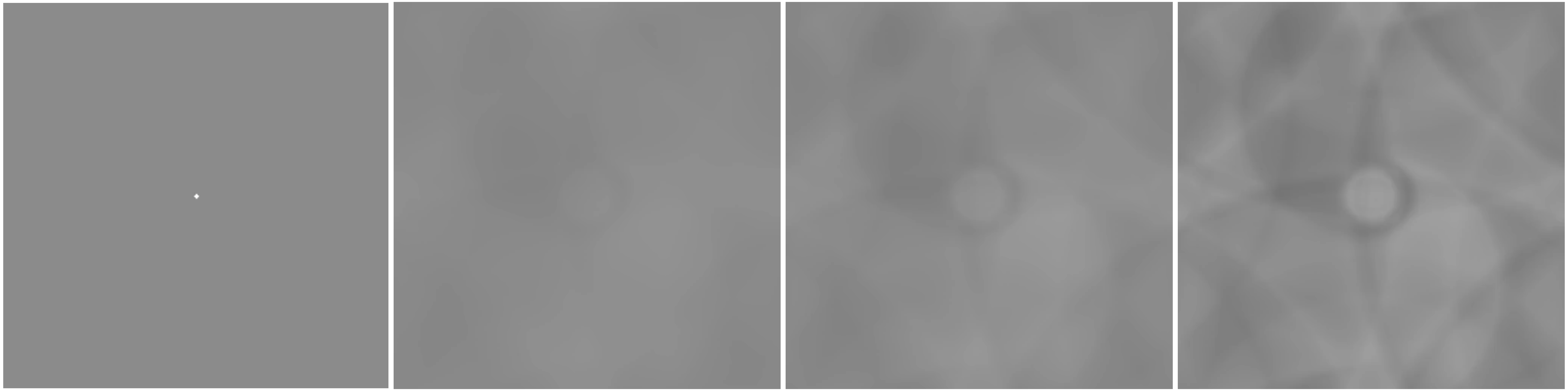


Optimize the **initial velocity field** so that the ink forms “Taichi” after 100 time steps.
10 Jacobi iterations are applied per time step for incompressibility.
Optimized using 200 gradient descent iterations.

Reproduce: `python3 smoke.py`

Differentiable Water Wave Simulation

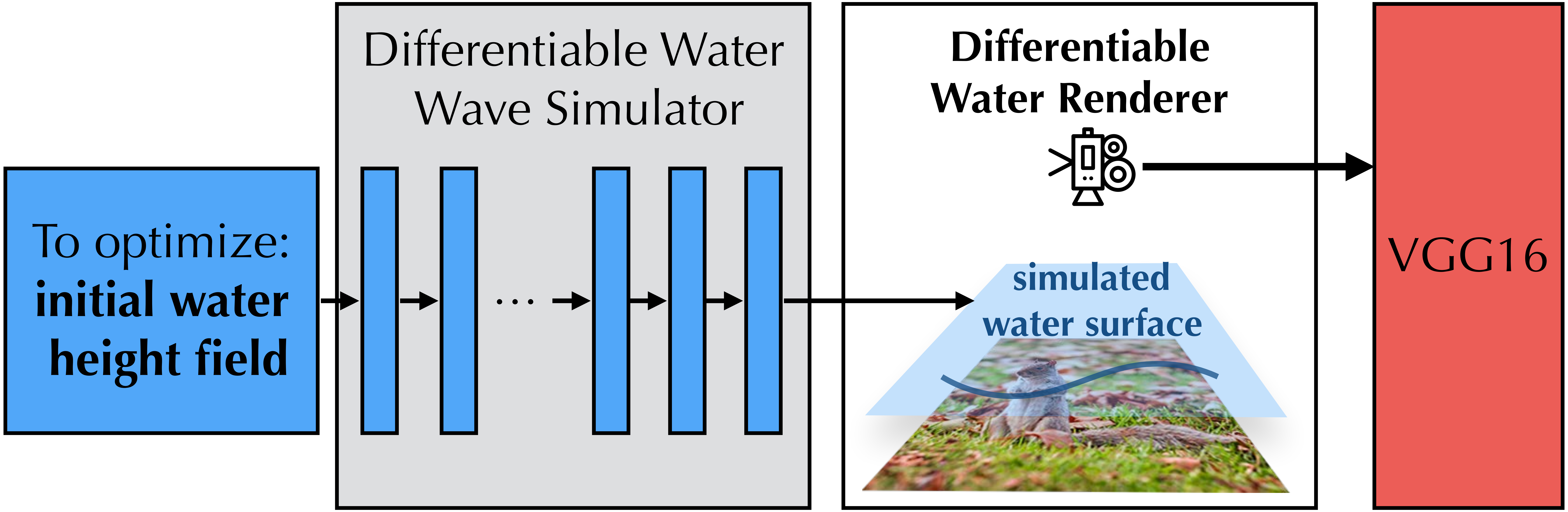
Center Activation | Iteration 20 | Iteration 60 | Iteration 180



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

Reproduce: `python3 wave.py`

Differentiable Water Renderer

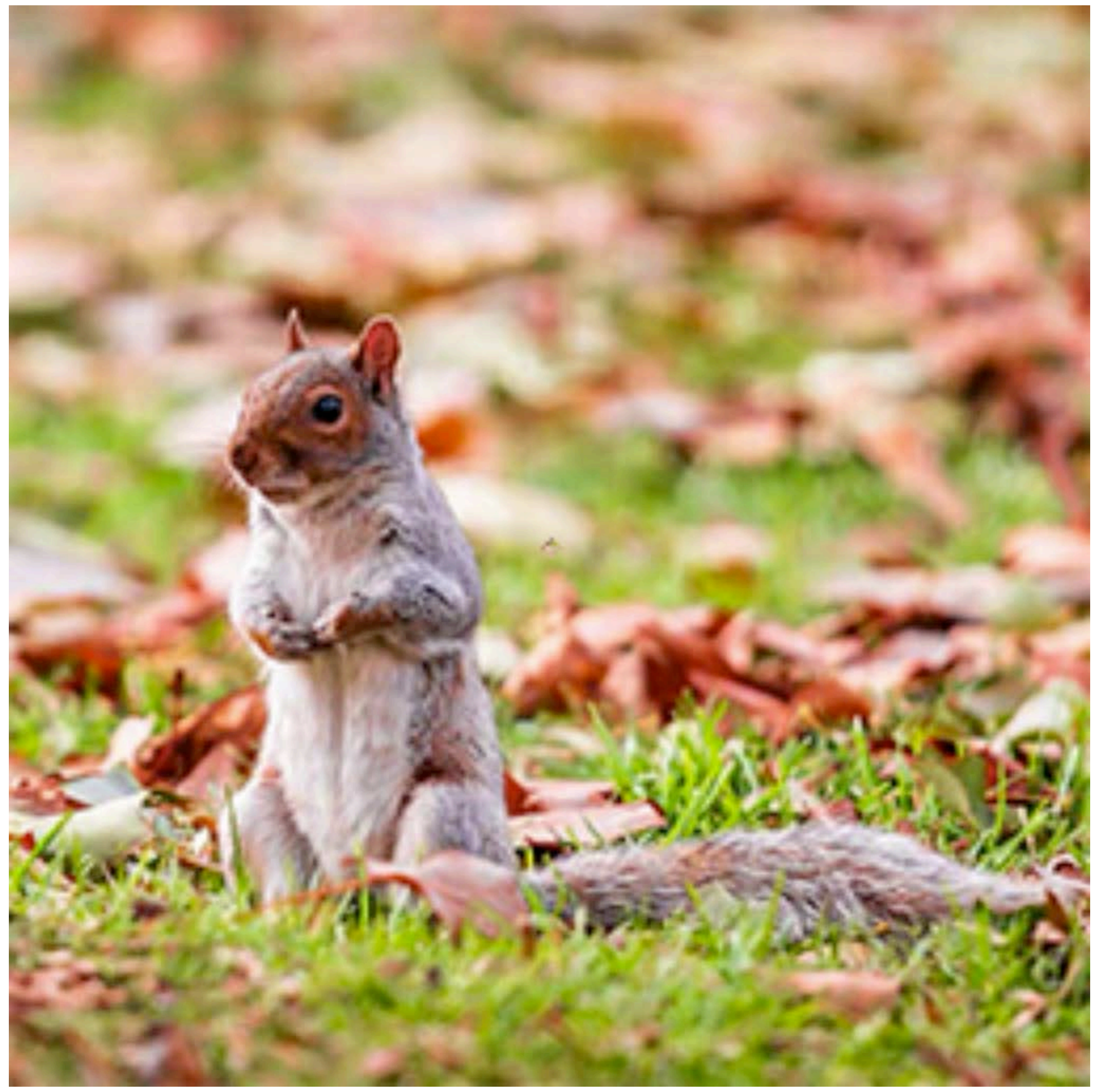


We wrote a differentiable water renderer to simulate refraction.
Then we connect the shader with the water wave simulator and VGG16.

Differentiable Water Renderer

Input image:

Center ripple



Iter. 10

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`

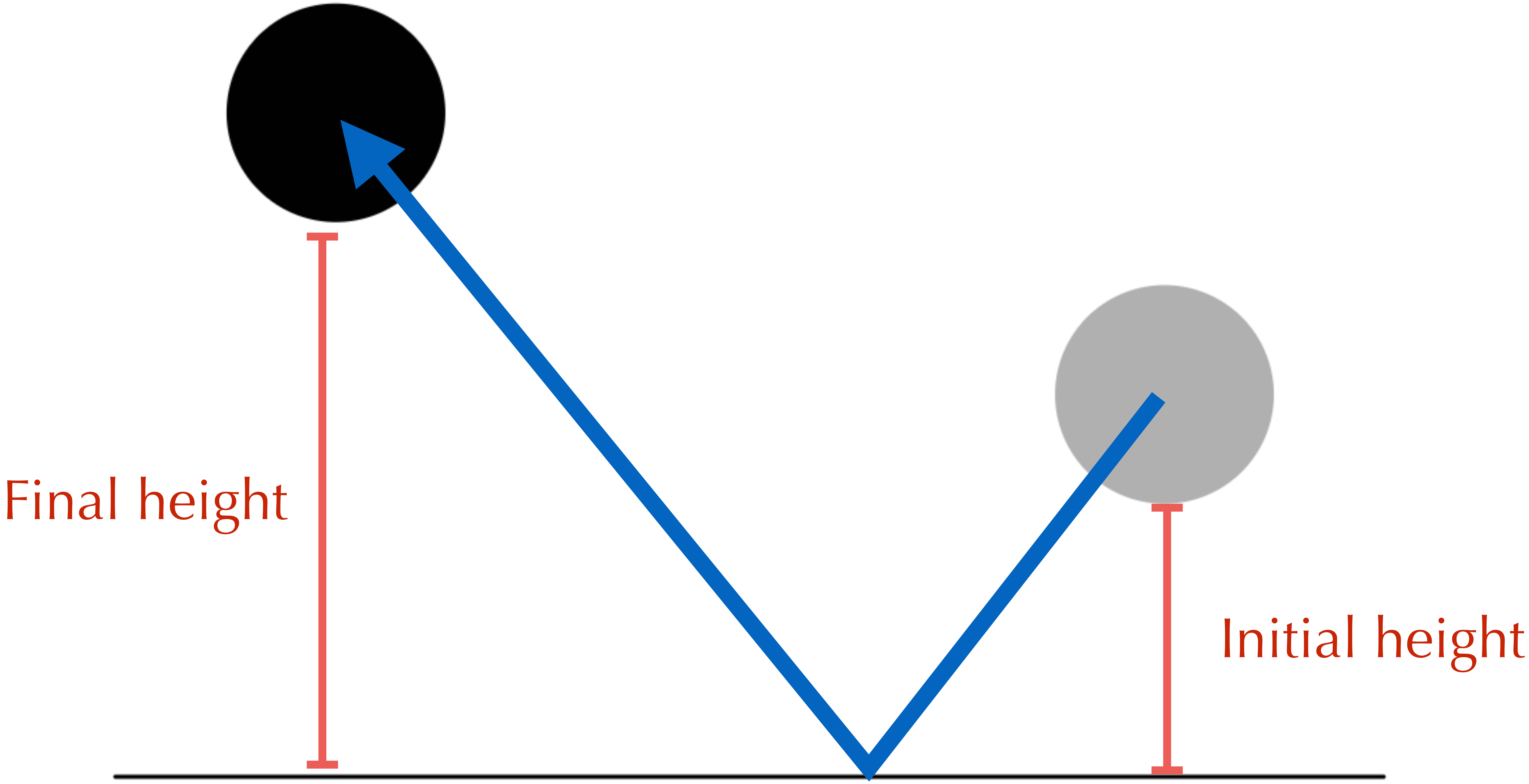
Building **Robust** Differentiable Physical Simulators

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

How Gradients Go Wrong

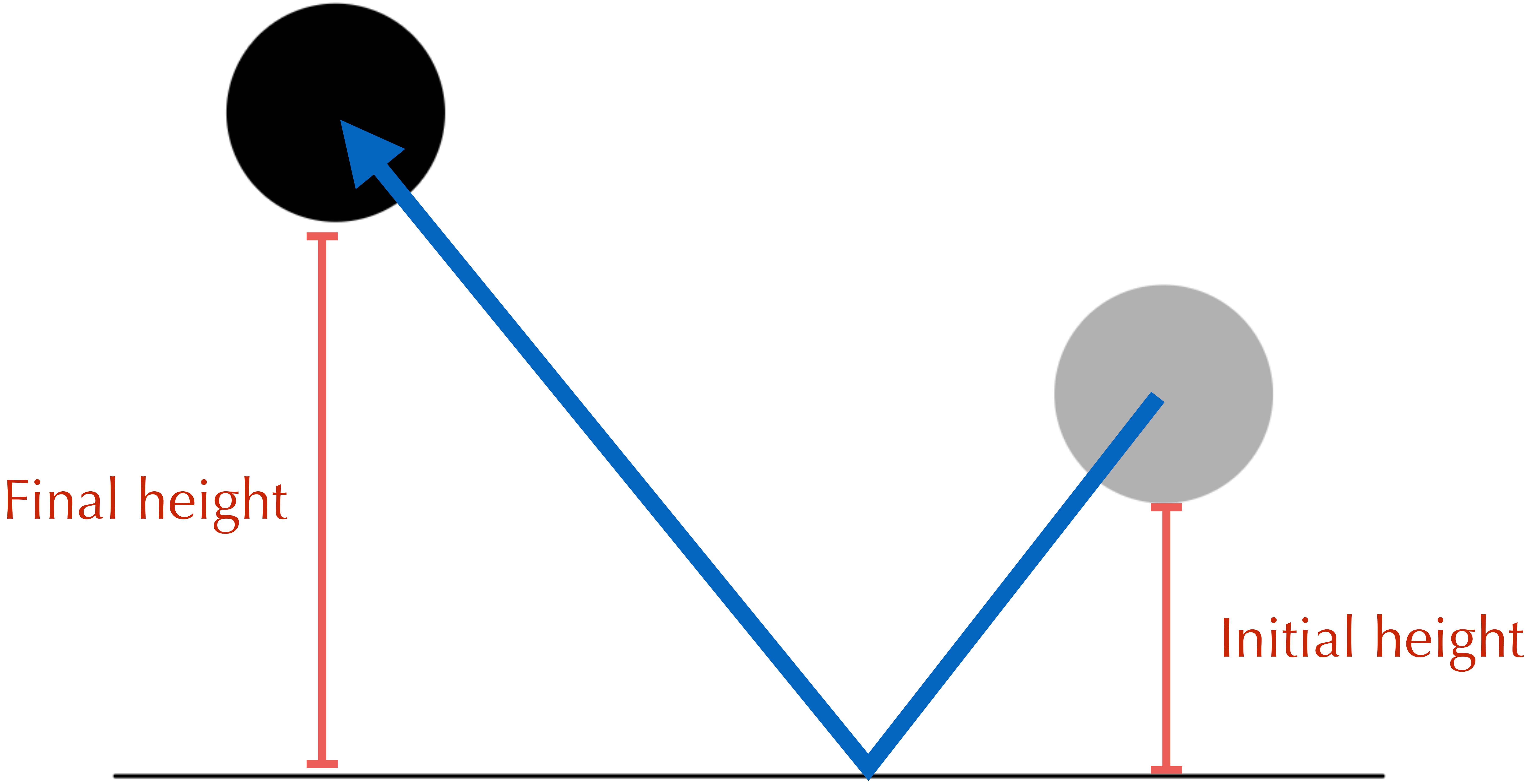
Consider this example where a rigid ball hits a friction-less ground.
No gravity, no friction, fully elastic collision.

How Gradients Go Wrong



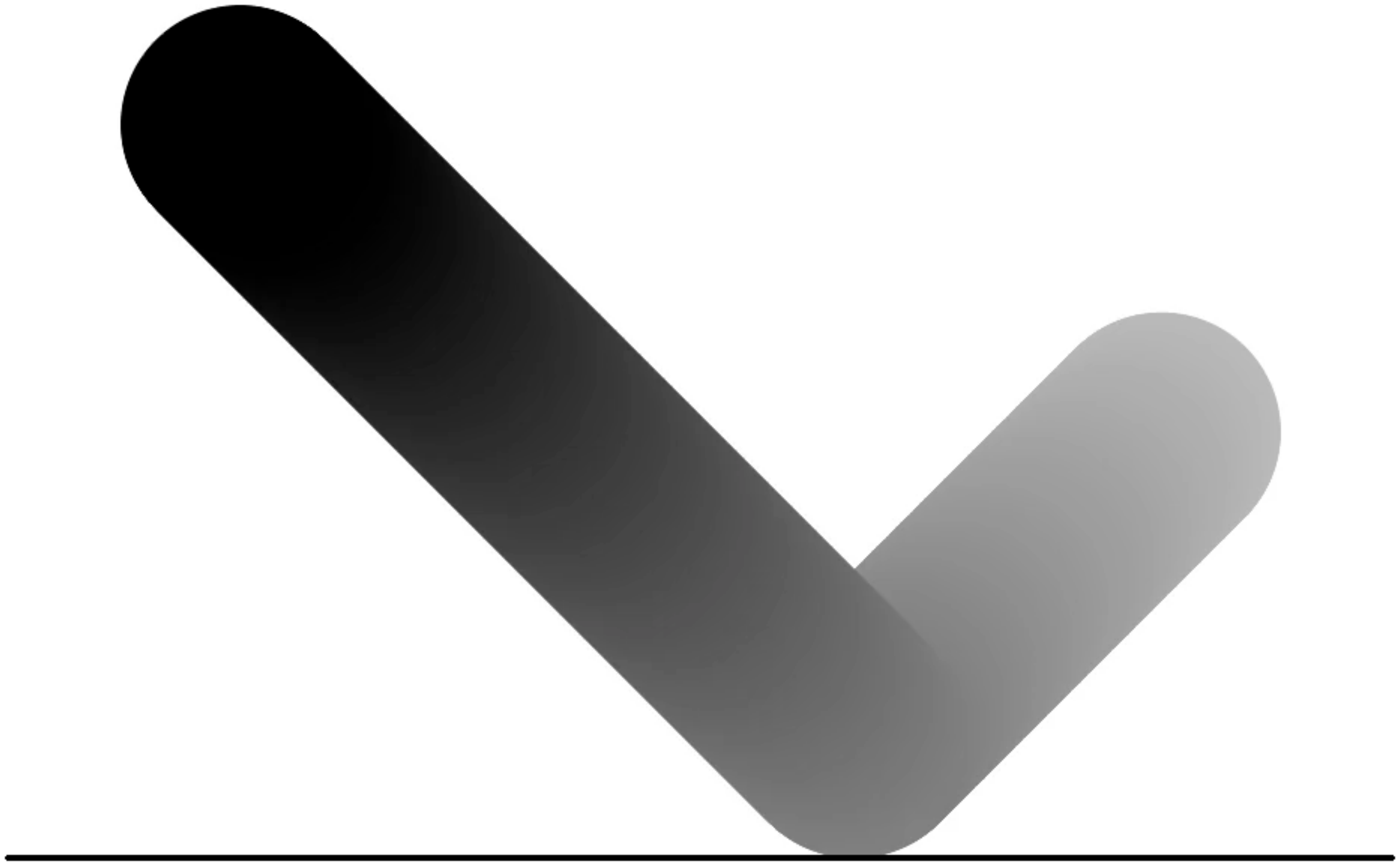
Consider this example where a rigid ball hits a friction-less ground.
No gravity, no friction, fully elastic collision.

How Gradients Go Wrong



Initial height + final height = time · $\mathbf{v_y}$**=constant**

How Gradients Go Wrong



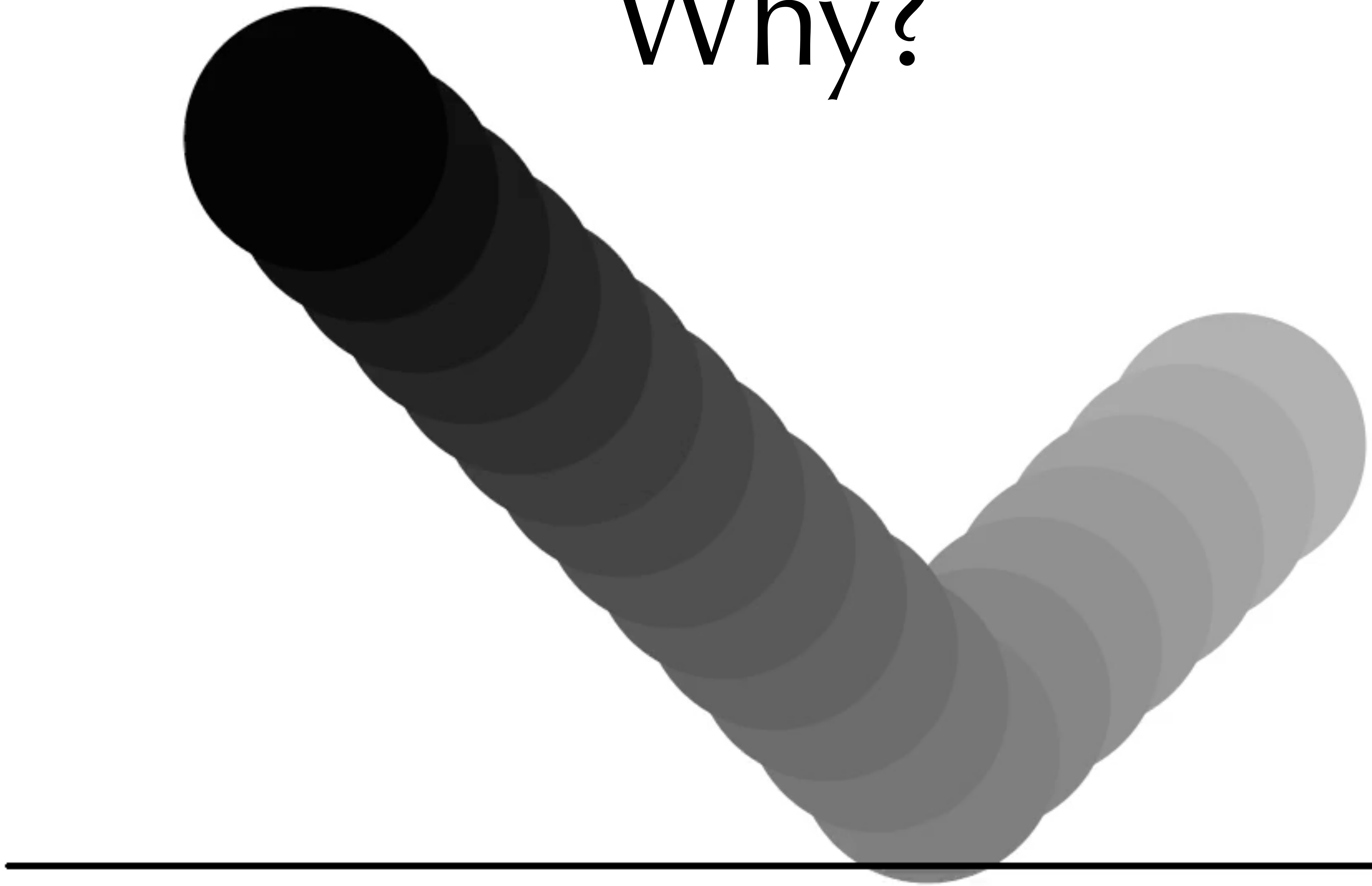
Initial height + final height = **constant** ⟶ $\dfrac{\partial \text{ final height}}{\partial \text{ initial height}} = -1$

But the differentiable simulator may tell you

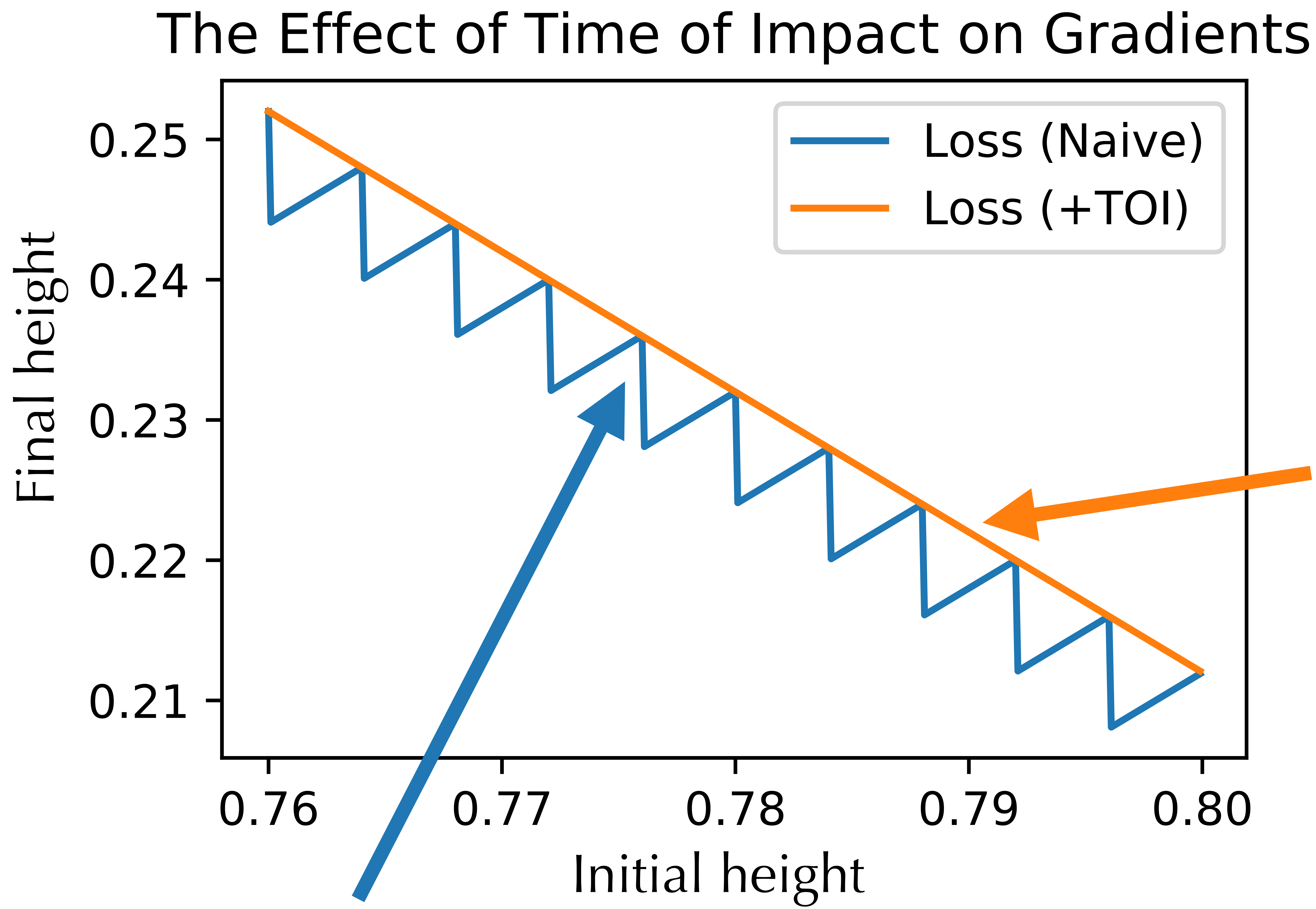
$$\frac{\partial\ \text{final height}}{\partial\ \text{initial height}} = 1$$

(instead of -1)

Why?



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.



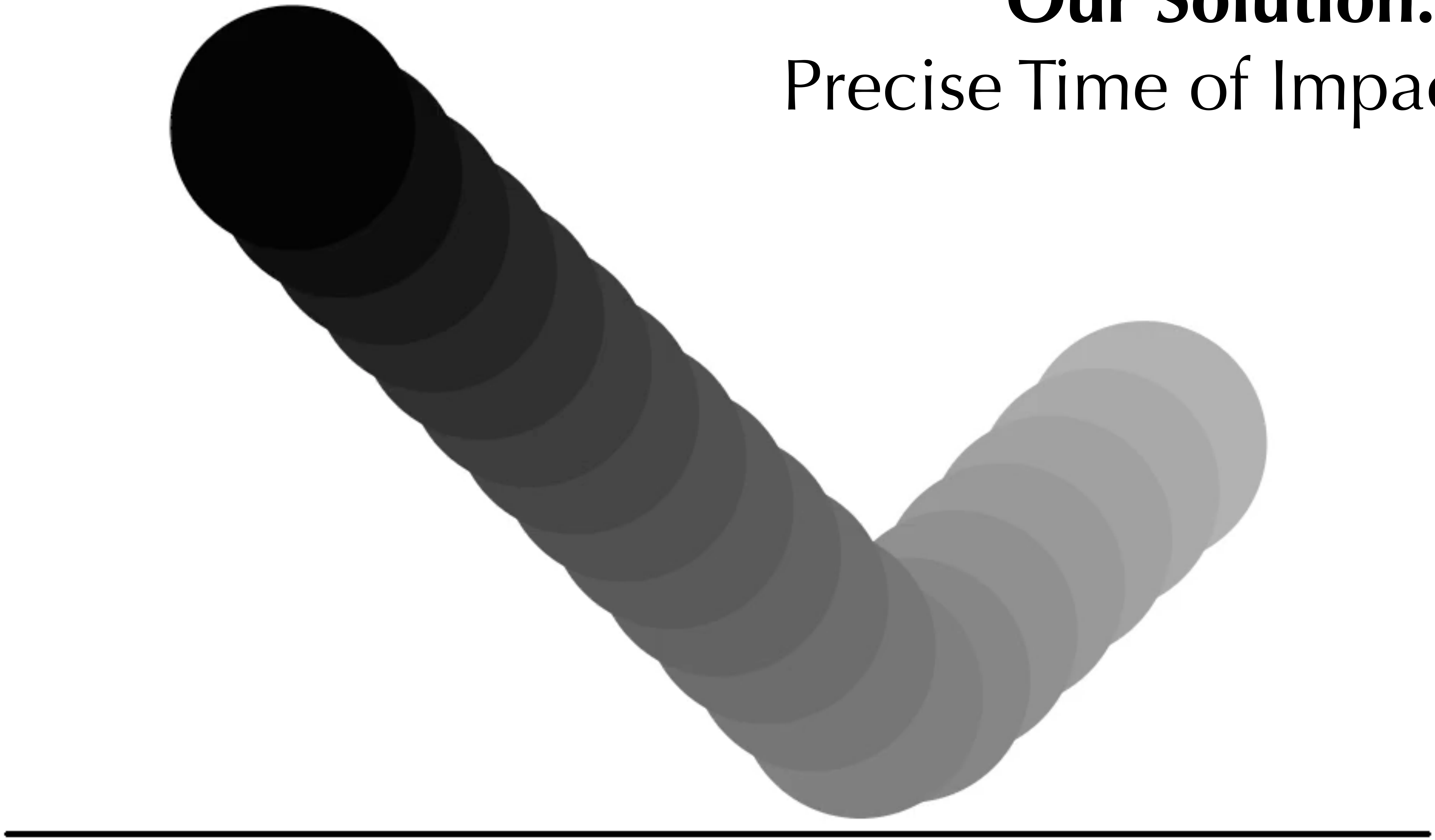
Question:

how can we get this?

A naive time integrator leads to saw-tooth like this: correct tendency, but completely wrong gradients

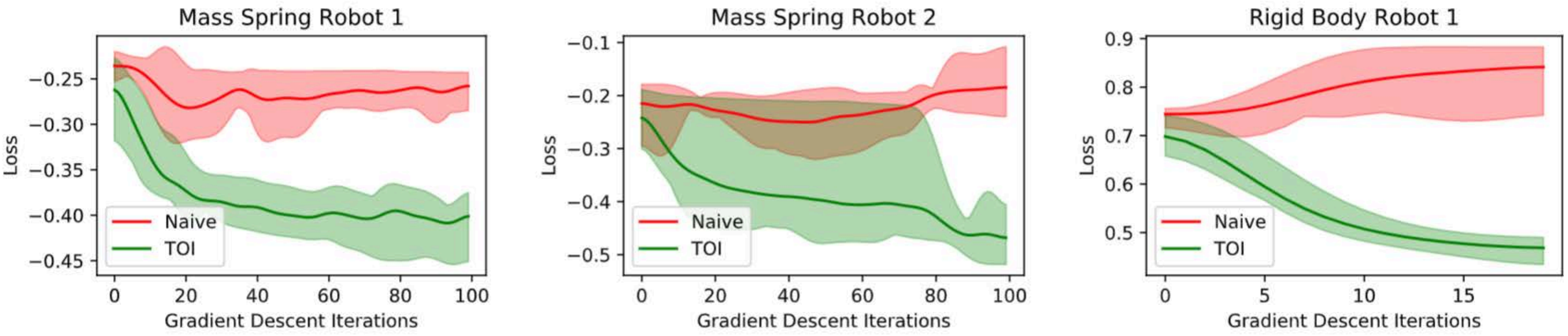
Our Solution:

Precise Time of Impact (TOI)



Initial height + final height = **constant** $\longrightarrow$ $\dfrac{\partial \text{ final height}}{\partial \text{ initial height}} = -1$

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.

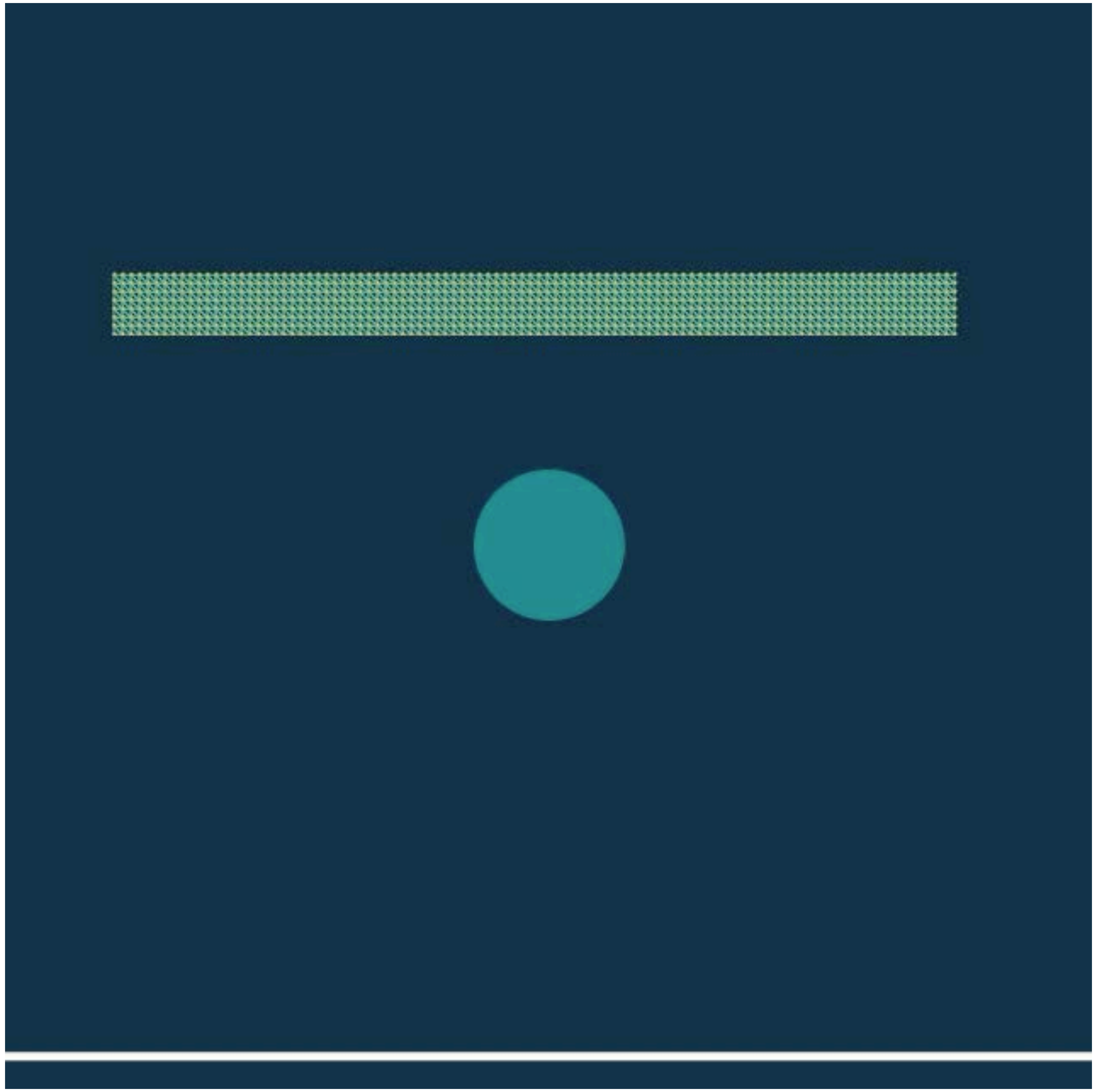
Takeaways:

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



potential energy

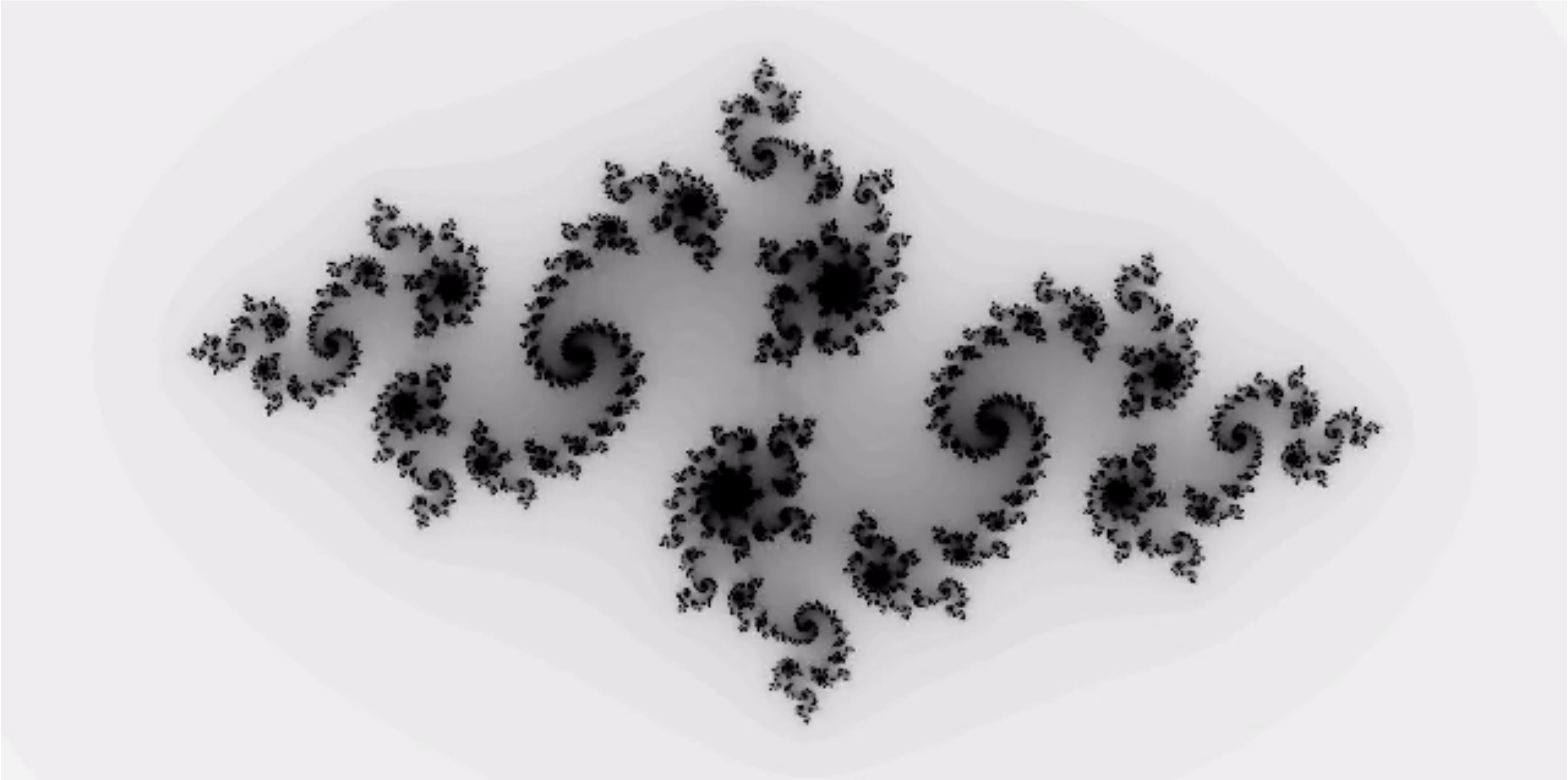
$$-\boldsymbol{f}_i(\hat{\boldsymbol{x}}) = \frac{\partial \Phi}{\partial \hat{\boldsymbol{x}}_i}(\hat{\boldsymbol{x}})$$

particle force

particle position

`fractal.py`: Your First Taichi Program

`fractal.py`: Your First Taichi Program



```python
# fractal.py
import taichi as ti
ti.init(arch=ti.cuda) # Run on GPU by default
```

Initialization

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

Tensor Allocation

```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):
  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
```

Computation Kernel

```python
gui = ti.GUI("Fractal", (n * 2, n))
for i in range(1000000):
  paint(i * 0.03)
  gui.set_image(pixels)
  gui.show()
```

Main program & Visualization

Initialization

```python
# fractal.py
import taichi as ti
ti.init(arch=ti.cuda) # Run on GPU by default
```

Tensor Allocation

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

Computation Kernel

```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):
  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
```

Main program & Visualization

```python
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

`ti.init`

- **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`
  - …

Initialization

```python
# fractal.py
import taichi as ti
ti.init(arch=ti.cuda) # Run on GPU by default
```

Tensor Allocation

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

Computation Kernel

```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):
  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
```

Main program & Visualization

```python
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**

- `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).**

Initialization

```python
# fractal.py
import taichi as ti
ti.init(arch=ti.cuda) # Run on GPU by default
```

Tensor Allocation

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

Computation Kernel

```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):
  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
```

Main program & Visualization

```python
gui = ti.GUI("Fractal", (n * 2, n))
for i in range(1000000):
  paint(i * 0.03)
  gui.set_image(pixels)
  gui.show()
```

Kernels

```python
@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
```

- **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

```
@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
    …
```

- **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**

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.**

Initialization

```python
# fractal.py
import taichi as ti
ti.init(arch=ti.cuda) # Run on GPU by default
```

Tensor Allocation

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

Computation Kernel

```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):
  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
```

Main program & Visualization

```python
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**:

```
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…

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

gui = ti.GUI("Fractal", (n * 2, n))
for i in range(1000000):
  paint(i * 0.03)
  gui.set_image(pixels)
  gui.show()
```

as if it is a Python function!

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

Programming Language

https://github.com/taichi-dev/taichi

「工欲善其事，必先利其器。」 ——《论语·卫灵公》

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

高级物理引擎实战2020 （基于太极编程语言）

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!