SIGGRAPH Courses '24, July 27 - August 01, 2024, Denver, CO, USA

ACM 979-8-4007-0683-7/24/07.

10.1145/3664475.3664543

WARP: DIFFERENTIABLE SPATIAL COMPUTING FOR PYTHON

“Warp accelerates Python functions with just-in-time (JIT) compilation for efficient execution on CPUs and GPUs. The course focus is on Warp’s application in physics simulation, perception, robotics, and geometry processing, and its ability to integrate with machine-learning frameworks like PyTorch and JAX”

Outline

- Introduction
- Writing Applications
- Automatic Differentiation
- Simulation and Machine Learning
- Conclusion and Q&A

Introduction to Warp

Nicolas Capens, NVIDIA

SIGGRAPH 2024

Problem Statement

- CUDA is general purpose, efficient, but device centric:
  - C++ focused
  - Not specialized for simulation
  - Build everything yourself
  - Not differentiable
- Python is the lingua franca for AI
  - Low barrier to entry
  - Fast iteration and deployment
  - Rich ecosystem of DL frameworks
  - Often has poor performance
- How can we bridge the gap between CUDA and Python for simulation developers?

NVIDIA®

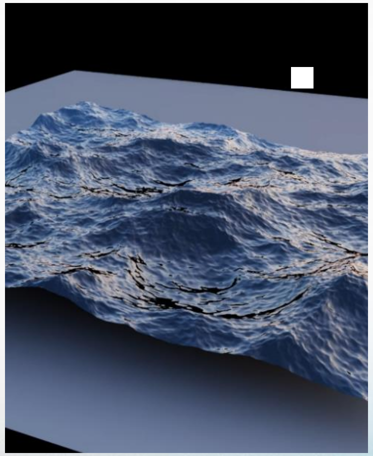
+

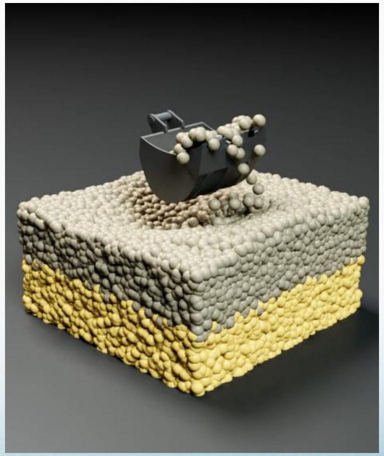
python™

Introducing Warp

1. GPU Kernels in Python
    - Write CUDA kernels in 100% Python syntax
    - Runtime JIT compilation
    - Fast developer iteration
2. Spatial Computing
    - Rich vector math library
    - Mesh processing and queries
    - Sparse volumes (OpenVDB)
    - Hash grids
3. Differentiable Programming
    - Auto-differentiation
    - Forward + backward kernel generation
    - Interop with DL frameworks (e.g.: PyTorch, JAX)
4. Omniverse Integration
    - OpenUSD import / export
    - Runtime extensions for Isaac / Composer

```python
import warp as wp

@wp.kernel
def integrate(p: wp.array(dtype=wp.vec3),
              v: wp.array(dtype=wp.vec3),
              f: wp.array(dtype=wp.vec3),
              m: wp.array(dtype=float)):

    # thread id
    tid = wp.tid()

    # Semi-implicit Euler step
    v[tid] = v[tid] + (f[tid] * m[tid] + wp.vec3(0.0, -9.8, 0.0)) * dt
    x[tid] = x[tid] + v[tid] * dt

# kernel launch
wp.launch(integrate, dim=1024, inputs=[x, v, f, …], device="cuda:0")
```

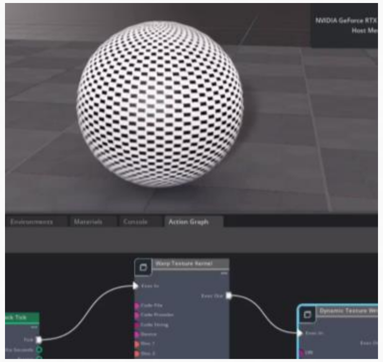




Warp Use Cases

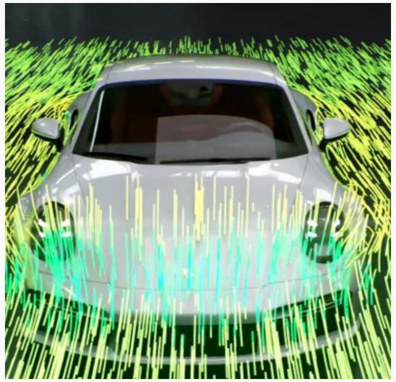
- Warp provides the building blocks for developers to create, accelerate and extend their own simulators

Data Processing



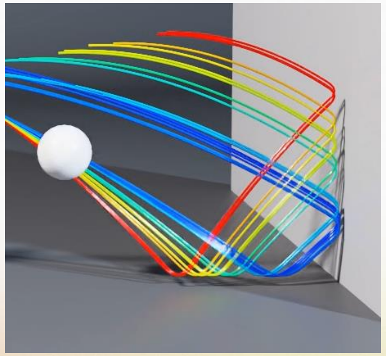
- Mesh processing
- Image processing
- Synthetic data generation

Simulation



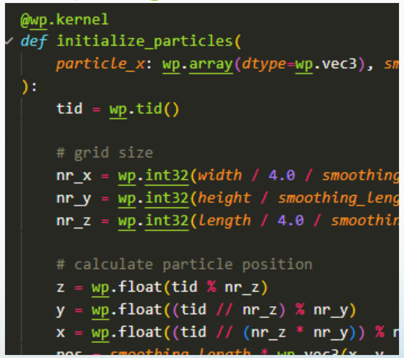
- Rigid-body dynamics
- Elasticity
- Fluid flow

Training



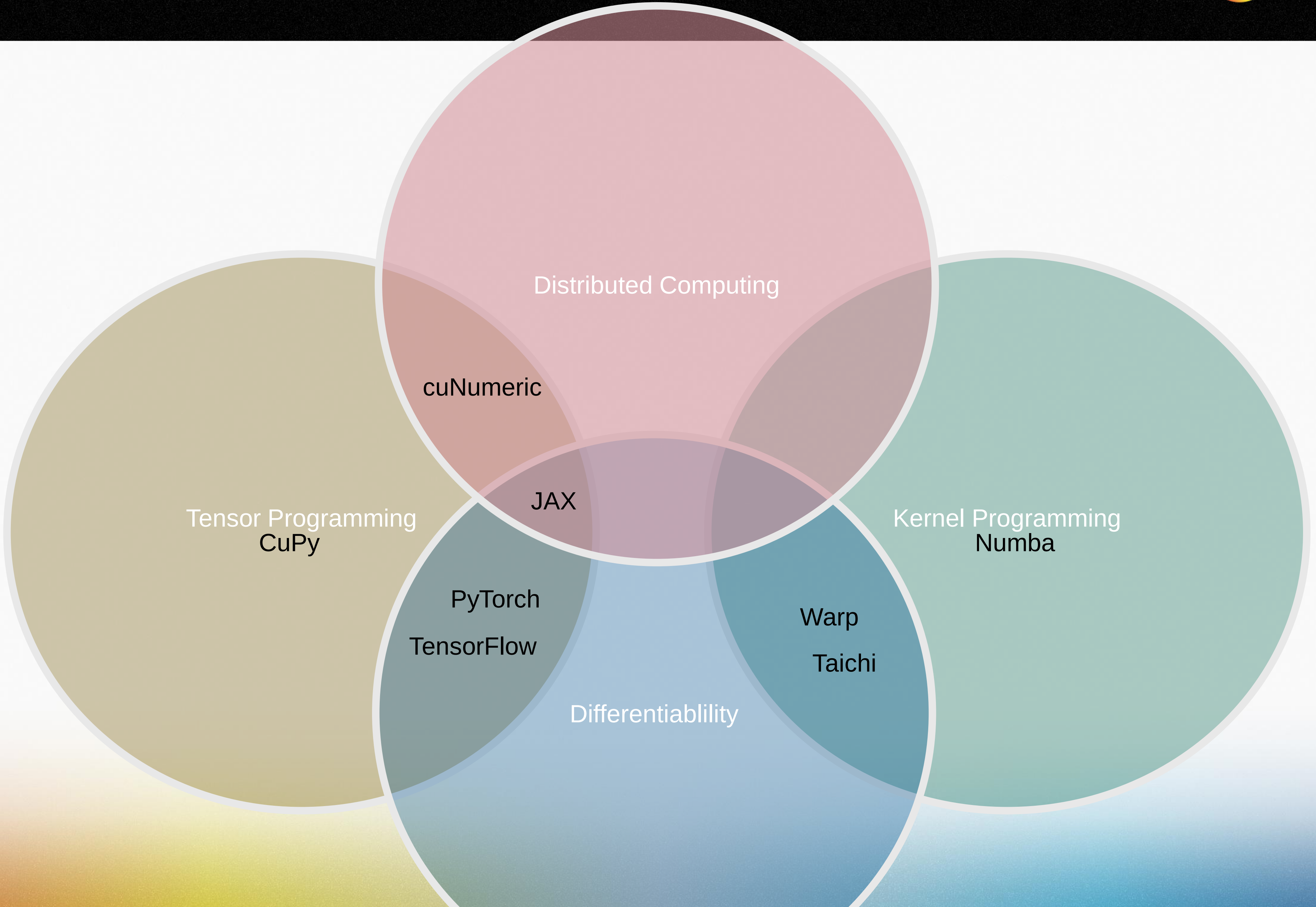
- Neural dynamics
- Parameter estimation
- Trajectory optimization
- Inverse problems

Scripting



- Loss/reward functions
- Custom forces
- Custom behaviors
- Custom boundaries

Python GPU Ecosystem



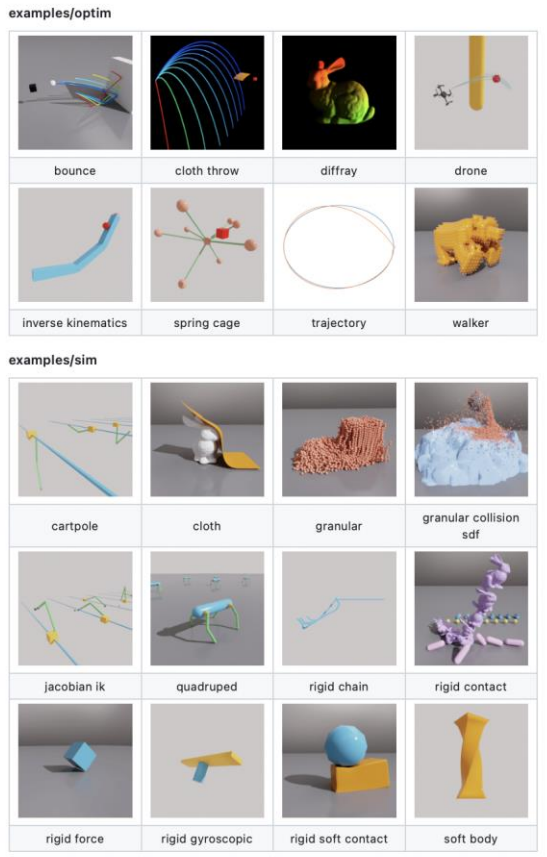
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

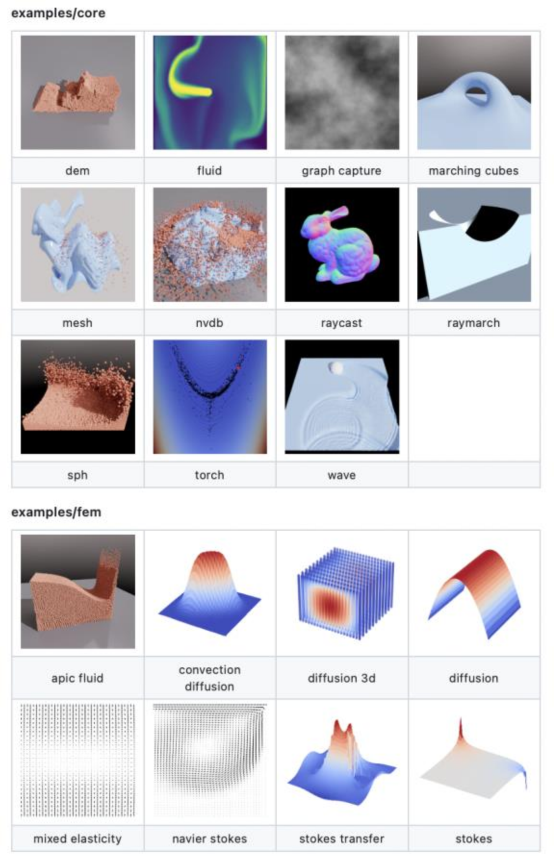
THE PREMIER CONFERENCE & EXHIBITION ON COMPUTER GRAPHICS & INTERACTIVE TECHNIQUES

WARP SDK

Warp Python Modules

- **`warp.core`**
  - Differentiable kernel coding for Python
  - Math, geometry, vector library
- **`warp.sim`**
  - Differentiable real-time simulation for robotic control + prediction
  - Rigid bodies, soft bodies, particles, cloth
  - URDF, MJCF, UsdPhysics parsers
- **`warp.fem (early access)`**
  - Differentiable PDE framework for heat transfer, diffusion, elasticity
  - Fast iteration, but offline focus
  - Not replacing existing codes, but potential to build on them
- **`warp.llm (early access)`**
  - AI agent specialized for Warp kernel coding
  - Generate simulation code directly from prompts
  - Generate loss functions from code -> optimization



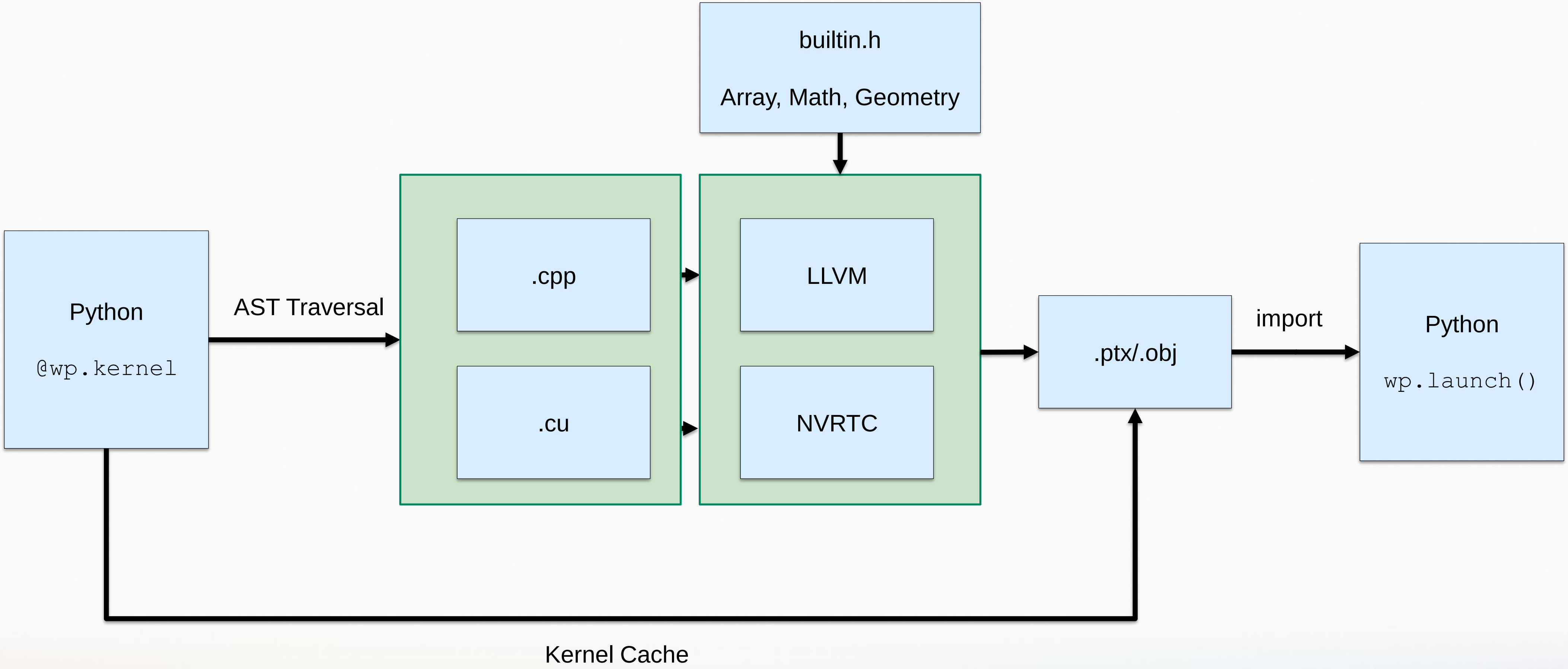


Warp Data Model

- Host/Device memory managed through wp.array type
- Built-in spatial math types similar to OpenCL/HLSL:
  - vec2, vec3, vec4, mat22, mat33, mat44, quat, transform
- Support for all common array protocols:
  - __array_interface__
  - __cuda_array_interface__
  - __dlplack__
- Zero-copy interop with PyTorch, JAX, NumPy:
  - wp.from_torch(), wp.to_torch()
  - wp.from_jax(), wp.to_jax()

```python
# 1D array of int8 types:
a = wp.zeros(shape=[64], dtype=wp.int8)

# 2D array of fp16 vec3 types:
a = wp.zeros(shape=[64, 64], dtype=wp.vec3h)

# 3D array of fp64 mat44 types:
a = wp.zeros(shape=[64, 64, 64], dtype=wp.mat44d)
```

Warp Compilation Pipeline



Warp Execution Model

- Warp exposes a thin abstraction over CUDA
- Kernels are launched over an N-dimensional grid of threads
  - Max 4-dimensional grids
  - Mapping to blocks is handled internally and not exposed to user
  - Grid stride kernels used to scale to large thread counts
    - Max up to 2^{31}-1 on each dimension
- Pure SIMT model
  - No shared memory
  - No warp-level primitives, e.g.: __shfl_sync()
- Custom native functions
  - C++/CUDA snippets

```python
@wp.kernel
def divergence(u: wp.array2d(dtype=wp.vec2),
               div: wp.array2d(dtype=float)):

    # 2D thread indices
    i, j = wp.tid()

    # boundary conditions
    if i == grid_width - 1:
        return
    if j == grid_height - 1:
        return

    # compute divergence
    dx = (u[i + 1, j][0] - u[i, j][0]) * 0.5
    dy = (u[i, j + 1][1] - u[i, j][1]) * 0.5
    div[i, j] = dx + dy

# 2d kernel launch
wp.launch(divergence, dim=[512, 512], inputs=[u, div])
```

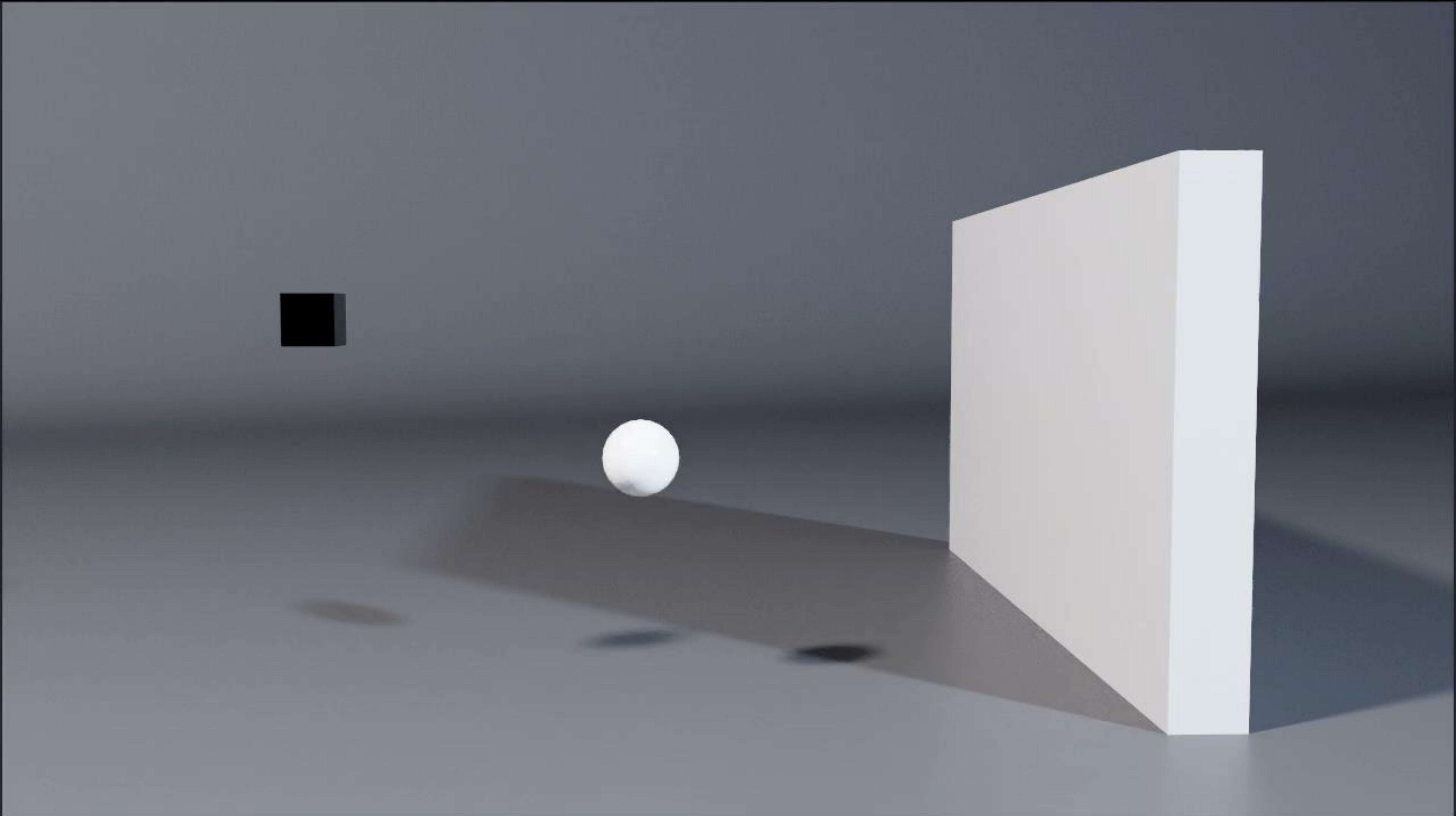
Example: 2D Divergence Calculation in Warp

Custom CUDA Code

- Add custom CUDA code snippets to Warp directly from Python via @wp.func_native decorator
- Allows easily dropping down to CUDA for access to:
  - Shared memory
  - Fine-grained synchronization
  - Cooperative operations
- Custom native backward functions
- Ability to include larger C++/CUDA codes (header-only) coming soon

```
snippet = """
out[tid] = a * x[tid] + y[tid];
"""

adj_snippet = """
adj_a = x[tid] * adj_out[tid];
adj_x[tid] = a * adj_out[tid];
adj_y[tid] = adj_out[tid];
"""

@wp.func_native(snippet, adj_snippet)
def saxpy(
    a: wp.float32,
    x: wp.array(dtype=wp.float32),
    y: wp.array(dtype=wp.float32),
    out: wp.array(dtype=wp.float32),
    tid: int,
):
    ...
```

Example: Custom CUDA snippet as a Warp function

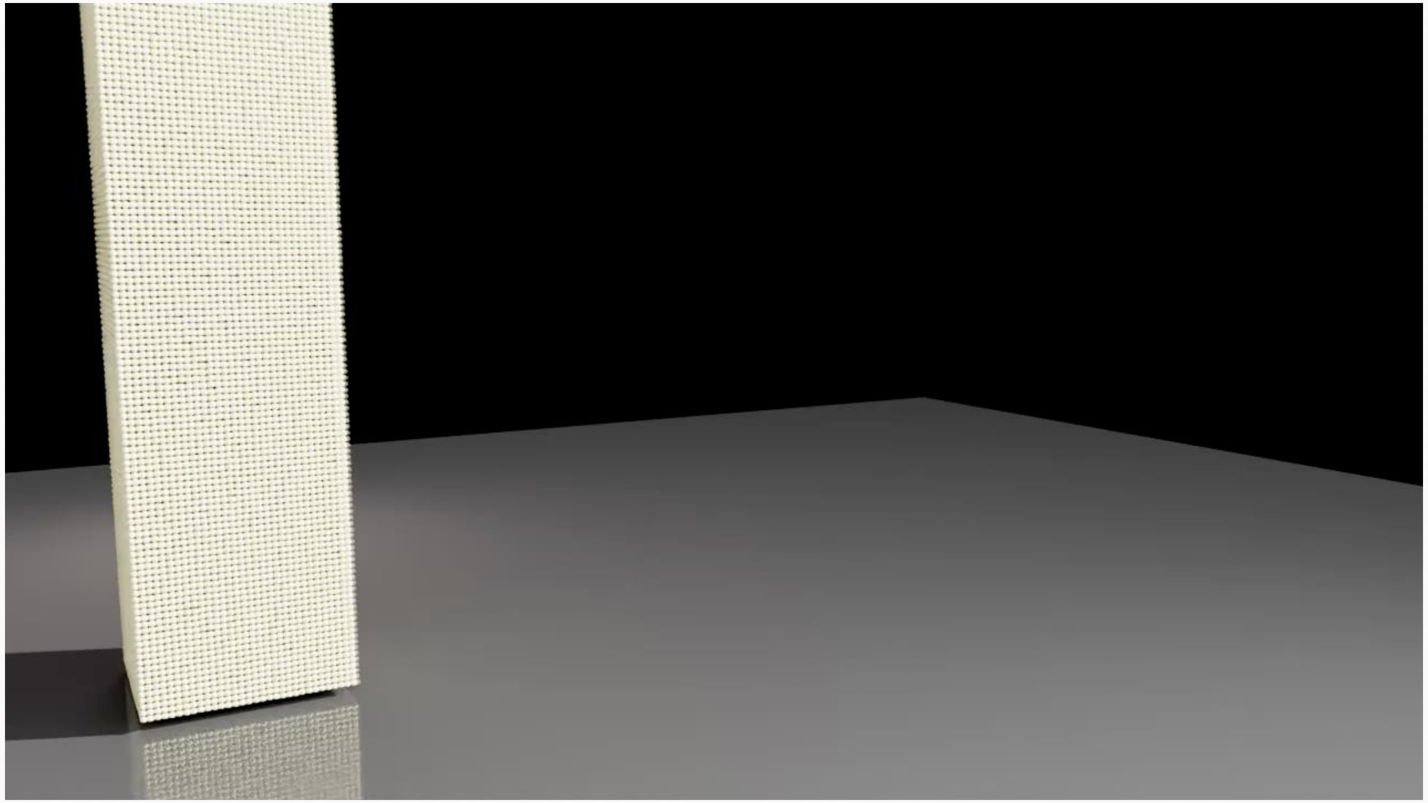


Mesh Data Structure

- Warp mesh data structure:
  - wp.Mesh
- Built-in mesh queries:
  - wp.mesh_query_point()
  - wp.mesh_query_ray()
  - wp.mesh_query_aabb()
- Fast inside/outside (sign) determination
- Fast GPU bounding volume hierarchy (BVH) refits:
  - mesh.refit()
- Example:
  - Neo-Hookean FEM elastic model
  - 60k particles versus 32k tris in 0.25ms
  - Dynamic signed-distance field (SDF) contact against meshes

Hash Grid Data Structure

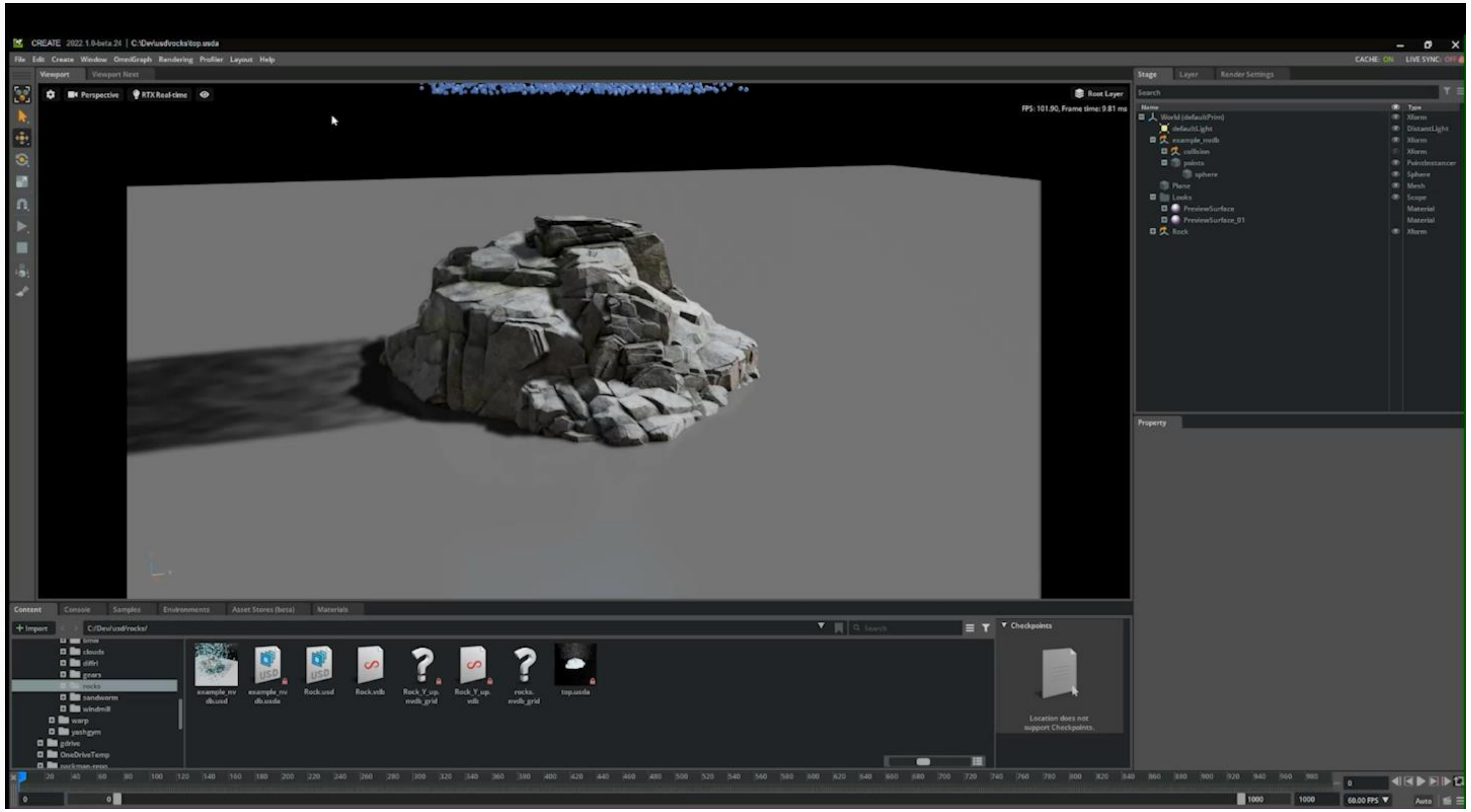
- Built-in hash-grid primitive
  - wp.HashGrid
- Fast neighbor finding / radius queries:
  - wp.hash_grid_query_point()
  - wp.hash_grid_query_next()
- GPU-side rebuild:
  - hashgrid.build()
- 10x faster than Open3D
- Example:
  - Discrete Element Method (DEM) in 200 lines of Python
  - 128k particles ~10ms/frame



Sparse Grid Data Structure

- Support for NanoVDB sparse volumes
  - wp.Volume
- Great for narrow-band SDF collision
- Simple to use API:
  - wp.volume_sample_world(vol, xyz)
  - wp.volume_sample_local(vol, uvw)
  - wp.volume_lookup(vol, ijk)
  - wp.volume_transform(vol, xyz)
  - wp.volume_transform_inv(vol, xyz)

OpenVDB



Warp Sim

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- A framework for real-time, differentiable, robotic simulation
- Physical Models
  - Rigid bodies
  - Particles
  - Constraints
  - Geometry
  - Forces
- Physical State
  - Position (q)
  - Velocity (qd)
- Integrators
  - Symplectic Euler (semi-implicit)
  - XPBD (implicit)
  - Featherstone

wp.sim.Model | wp.sim.State | wp.sim.Integrator

```python
import warp as wp
import warp.sim

sim_dt = 1/60

builder = wp.sim.ModelBuilder()
wp.sim.parse_urdf("cartpole.urdf", builder)
model = builder.finalize()

integrator = wp.sim.SemiImplicitIntegrator()
state, next_state = model.state(), model.state()

for i in range(100):
    state.clear_forces()
    state = integrator.simulate(model, state, next_state, sim_dt)
    state, next_state = next_state, state
```

Example: Creating a cartpole simulation using warp.sim

Warp FEM

Differentiable Framework for PDEs

- Flexible finite element-based (FEM/DG) framework for:
  - Diffusion
  - Convection
  - Fluid flow
  - Elasticity
- Define your weak-form PDE in high-level Python syntax:
  - wp.fem.grad()
  - wp.fem.div()
  - wp.fem.curl()
- Combine with geometry + shape functions:
  - wp.fem.TetMesh()
  - wp.fem.HexMesh()
  - wp.fem.make_polynomial_space()
- Integrate to assemble linear system:
  - wp.fem.integrate()
- Solve with block-sparse (BSR) solvers:
  - warp.sparse.BsrMatrix

Define PDE

```
import warp.fem as fem

@   .integrand
def diffusion_form s       Sample  u       Field  v       Field  nu  float
    return nu       dot
           grad u  s
           grad v  s
```

Define Geometry

```
geo        Tetmesh positions            tet_vertex_indices
scalar_space       make_polynomial_space
    geo  degree 2  element_basis     ElementBasis
```

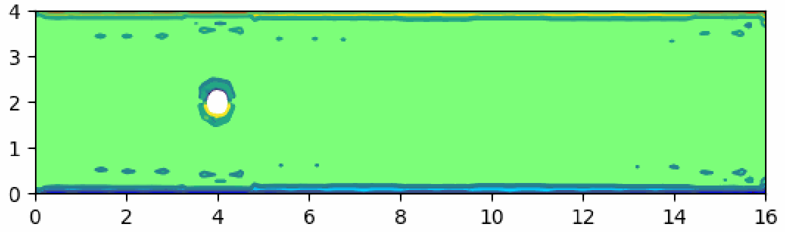
Assemble System

```
trial       make_trial
test       make_test

matrix       integrate
           fields  "u"  trial  "v"  test
           values  "nu"  0.1
```

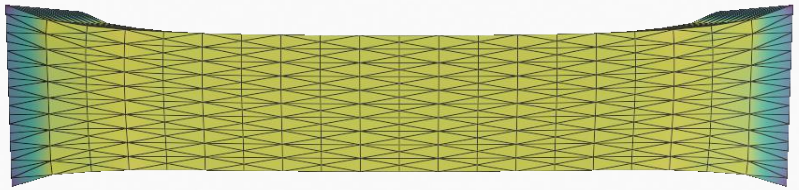
Example: Diffusion PDE bilinear form

Warp FEM

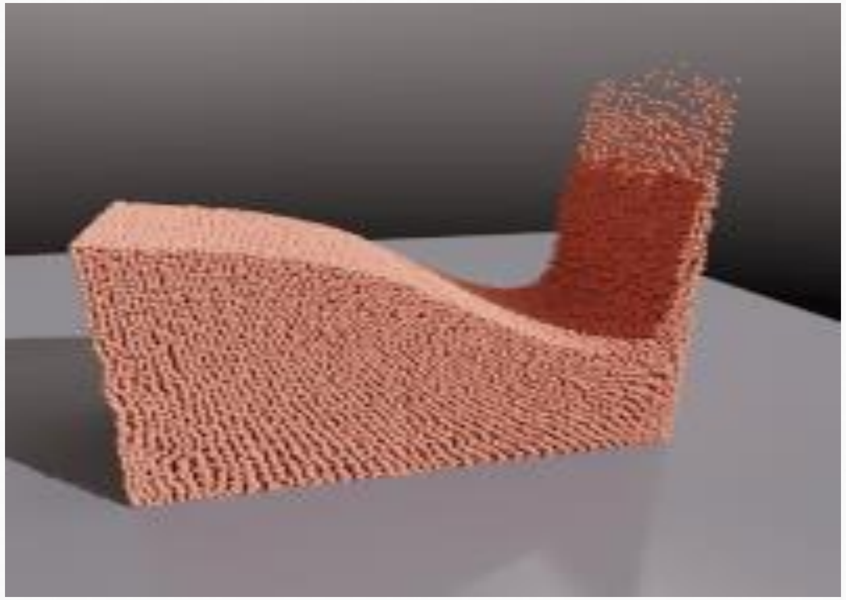
Examples



Example: Navier—Stokes flow simulation



Example: Neo-Hookean Elasticity



Example: Hybrid Eulerian—Lagrangian fluids

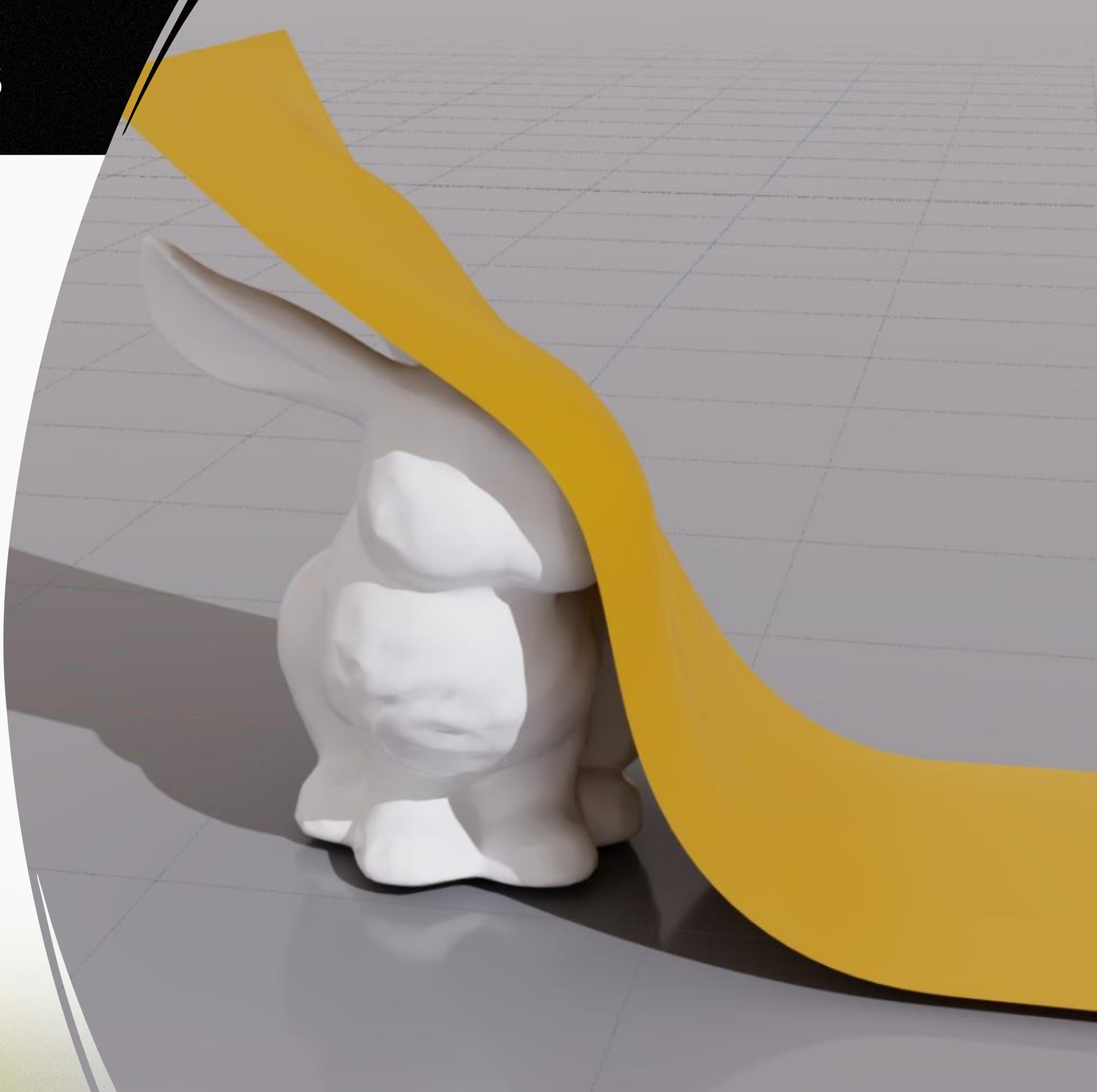
Writing Applications

Gergely Klár, NVIDIA

PART 1 – SIMULATION BASICS

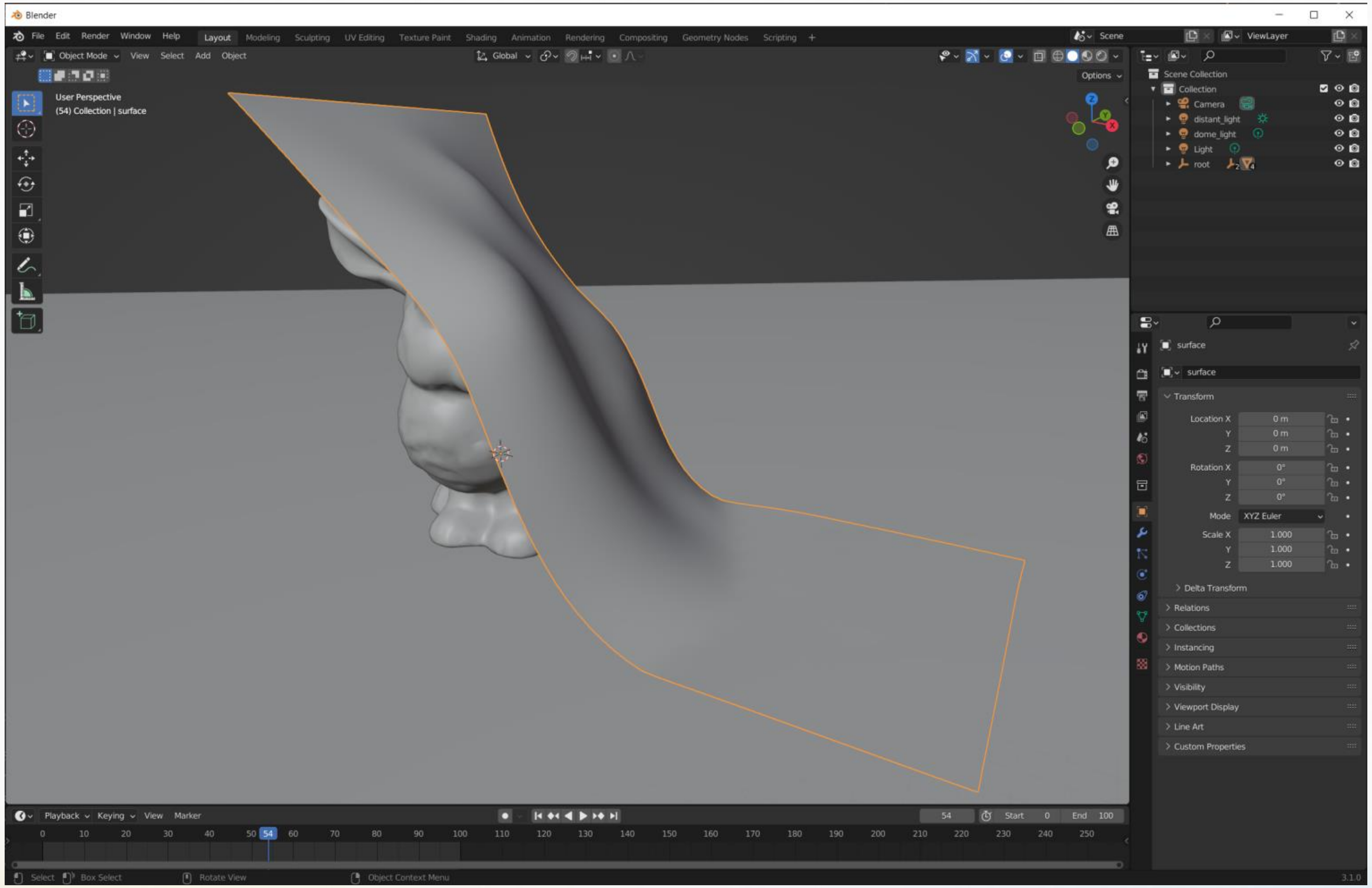
LET'S RE-CREATE THE CLOTH EXAMPLE!

- What you will learn
  - Using the warp.sim module
  - Importing and exporting USD assets
  - Using graph captures to reduce overheads

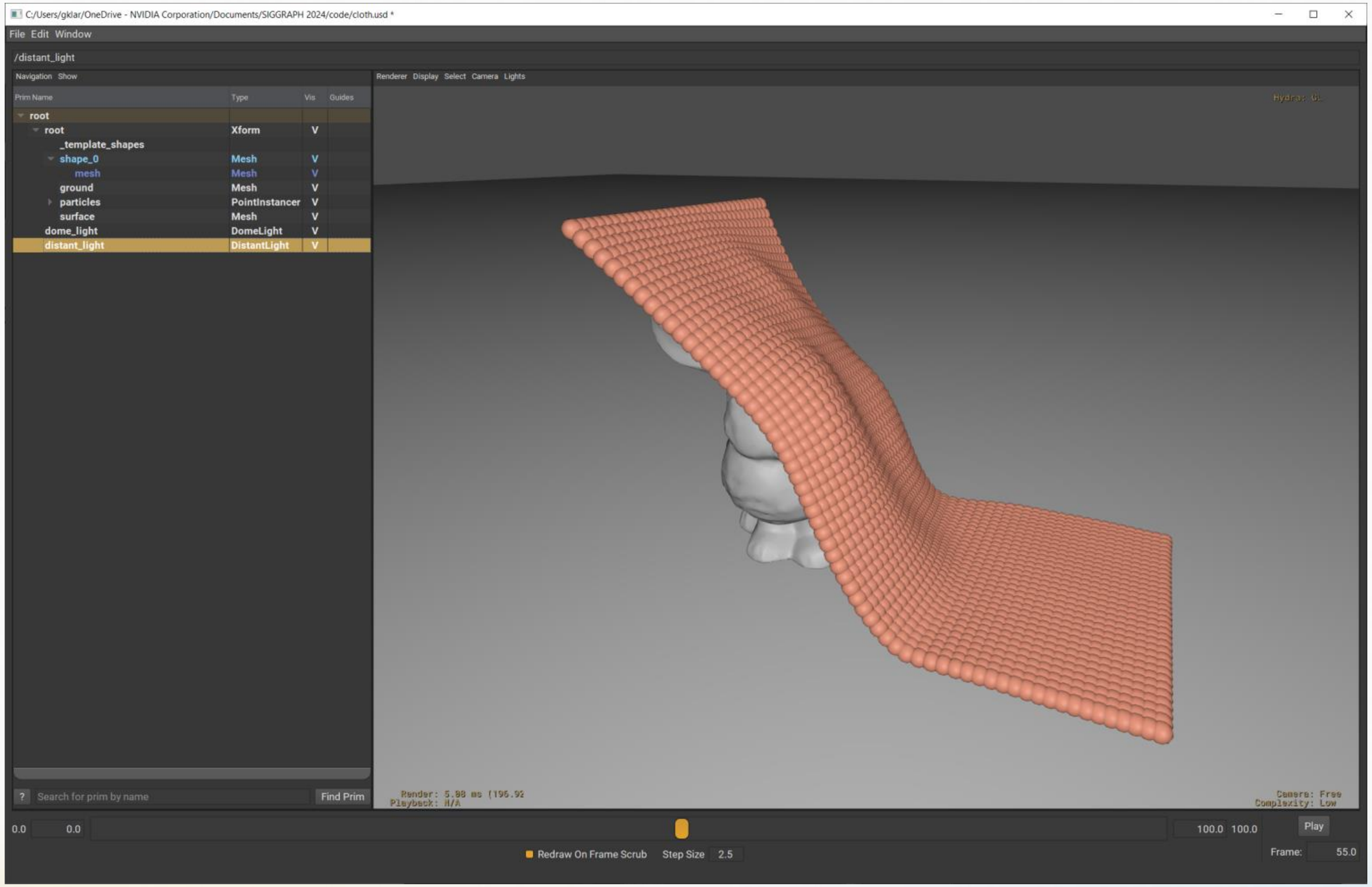


- Prerequisites:
  - Install warp-lang
    - For CUDA 12.5 Driver: Install from PyPI
    - For CUDA 11.8 Driver: Install from GitHub Releases
  - Install usd-core
    - `pip install usd-core`
  - USD viewer of your choice

RESULTS IN BLENDER



RESULTS IN USDVIEW

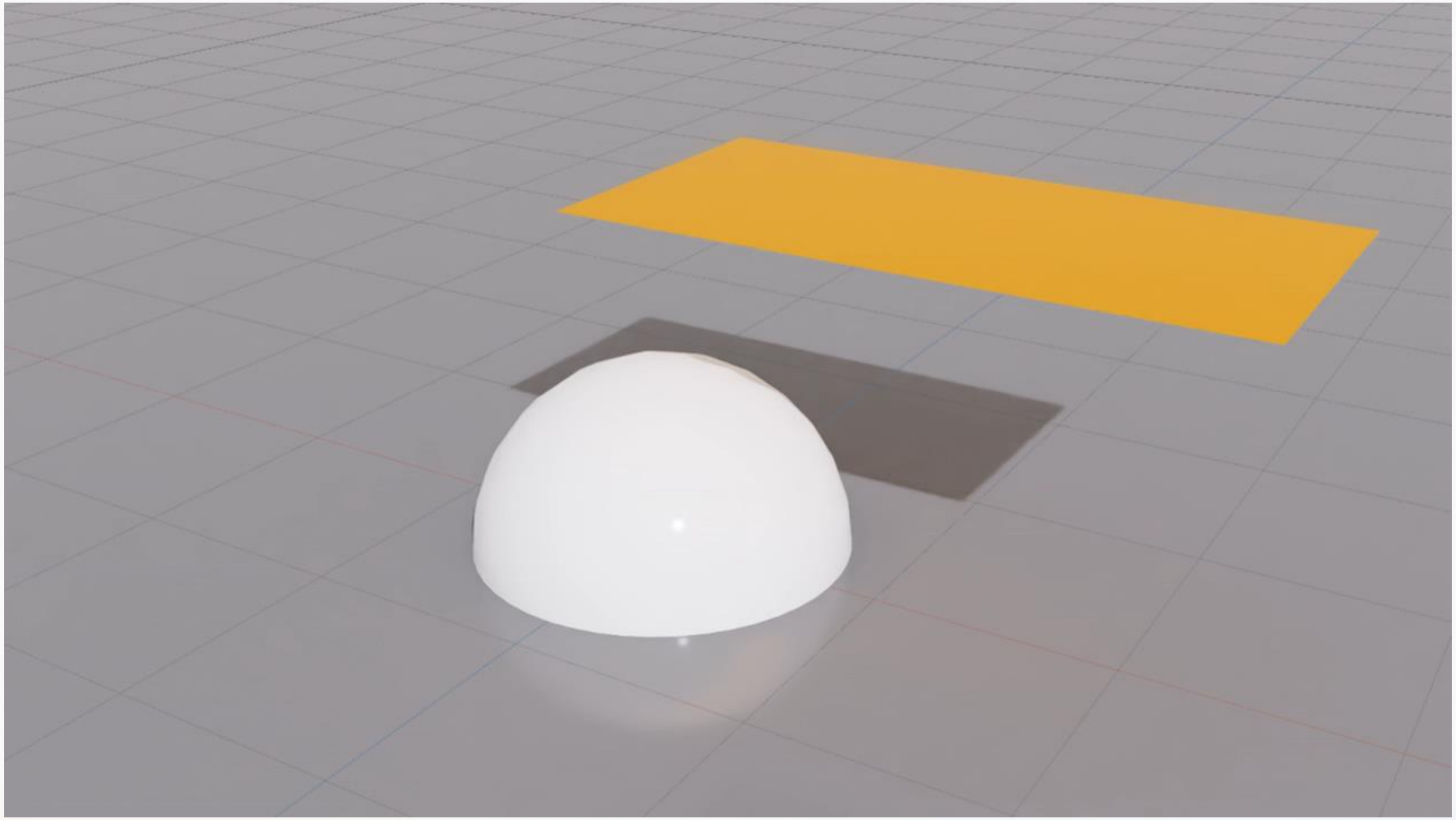


CLOTH IN < 100 LINES

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```python
import math

import warp as wp
import warp.sim
import warp.sim.render

FPS = 60
SUBSTEPS = 32
NUM_FRAMES = 300
OUTPUT_PATH = "cloth_against_sphere.usd"

wp.init()

frame_dt = 1.0 / FPS
sim_dt = frame_dt / SUBSTEPS
sim_time = 0.0
profiler = {}

builder = wp.sim.ModelBuilder()

# Adding the cloth
builder.add_cloth_grid(
    pos=wp.vec3(0.0, 4.0, -1.6),
    rot=wp.quat_from_axis_angle(wp.vec3(1.0, 0.0, 0.0), math.pi * 0.5),
    vel=wp.vec3(0.0, 0.0, 0.0),
    dim_x=64, dim_y=32,
    cell_x=0.1, cell_y=0.1,
    mass=0.1,
    fix_left=True,
    tri_ke=1.0e3, tri_ka=1.0e3, tri_kd=1.0e1,
)
# Adding the collision object
builder.add_shape_sphere(
    body=-1,
    pos=wp.vec3(0.0, 0.0, 0.0),
    radius=2.0,
    ke=1.0e2, kd=1.0e2, kf=1.0e1,
)
```

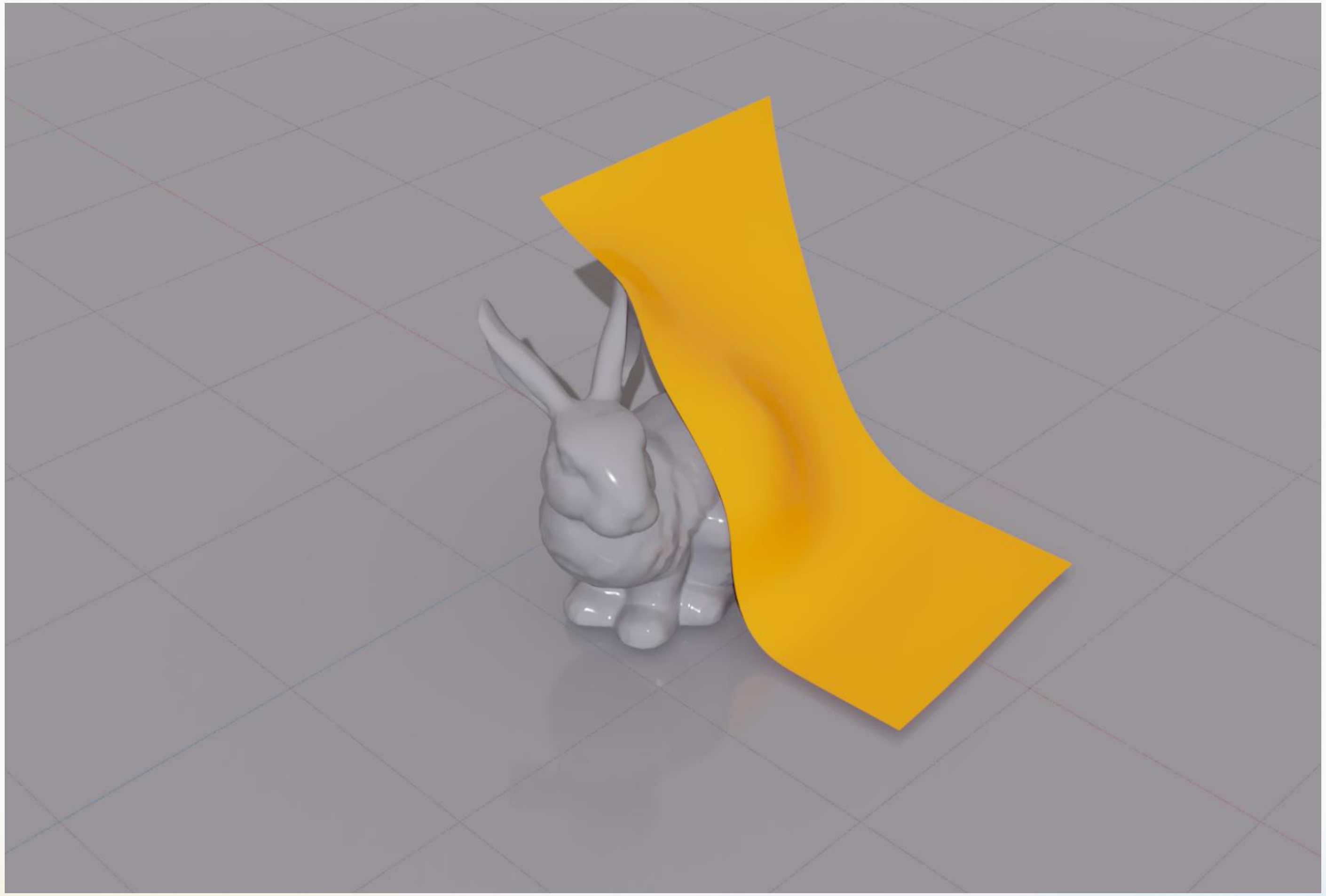
```python
integrator = wp.sim.SemiImplicitIntegrator()

model = builder.finalize()
model.ground = True
model.soft_contact_ke = 1.0e4
model.soft_contact_kd = 1.0e2

state_0 = model.state()
state_1 = model.state()

renderer = wp.sim.render.SimRenderer(model, OUTPUT_PATH, scaling=40.0)

for _ in range(NUM_FRAMES):
    # Step
    with wp.ScopedTimer("step", dict=profiler):
        wp.sim.collide(model, state_0)

        for _ in range(SUBSTEPS):
            state_0.clear_forces()
            integrator.simulate(model, state_0, state_1, sim_dt)

            # swap states
            (state_0, state_1) = (state_1, state_0)

    sim_time += frame_dt

    # Render
    with wp.ScopedTimer("render"):
        renderer.begin_frame(sim_time)
        renderer.render(state_0)
        renderer.end_frame()

renderer.save()

frame_times = profiler["step"]
print("\nTotal simulation time: {:.2f} ms".format(sum(frame_times)))
```



USD IMPORT

```python
import math

import warp as wp
import warp.sim
import warp.sim.render

FPS = 60
SUBSTEPS = 32
NUM_FRAMES = 300
OUTPUT_PATH = "cloth_against_sphere.usd"

wp.init()

frame_dt = 1.0 / FPS
sim_dt = frame_dt / SUBSTEPS
sim_time = 0.0
profiler = {}

builder = wp.sim.ModelBuilder()

# Adding the cloth
builder.add_cloth_grid(
    pos=wp.vec3(0.0, 4.0, -1.6),
    rot=wp.quat_from_axis_angle(wp.vec3(1.0, 0.0, 0.0), math.pi * 0.5),
    vel=wp.vec3(0.0, 0.0, 0.0),
    dim_x=64, dim_y=32,
    cell_x=0.1, cell_y=0.1,
    mass=0.1,
    fix_left=True,
    tri_ke=1.0e3, tri_ka=1.0e3, tri_kd=1.0e1,
)
# Adding the collision object
builder.add_shape_sphere(
    body=-1,
    pos=wp.vec3(0.0, 0.0, 0.0),
    radius=2.0,
    ke=1.0e2, kd=1.0e2, kf=1.0e1,
)
```

```python
import numpy as np
from pxr import Usd, UsdGeom

OUTPUT_PATH = "cloth_against_bunny.usd"

# Adding the collision object
usd_stage = Usd.Stage.Open("bunny.usd")
usd_geom = UsdGeom.Mesh(usd_stage.GetPrimAtPath("/root/bunny"))

mesh_points = np.array(usd_geom.GetPointsAttr().Get())
mesh_indices = np.array(usd_geom.GetFaceVertexIndicesAttr().Get())

mesh = wp.sim.Mesh(mesh_points, mesh_indices)

builder.add_shape_mesh(
    body=-1,
    mesh=mesh,
    pos=wp.vec3(1.0, 0.0, 1.0),
    rot=wp.quat_from_axis_angle(wp.vec3(0.0, 1.0, 0.0), math.pi * 0.5),
    scale=wp.vec3(2.0, 2.0, 2.0),
    ke=1.0e2,
    kd=1.0e2,
    kf=1.0e1,
)
```

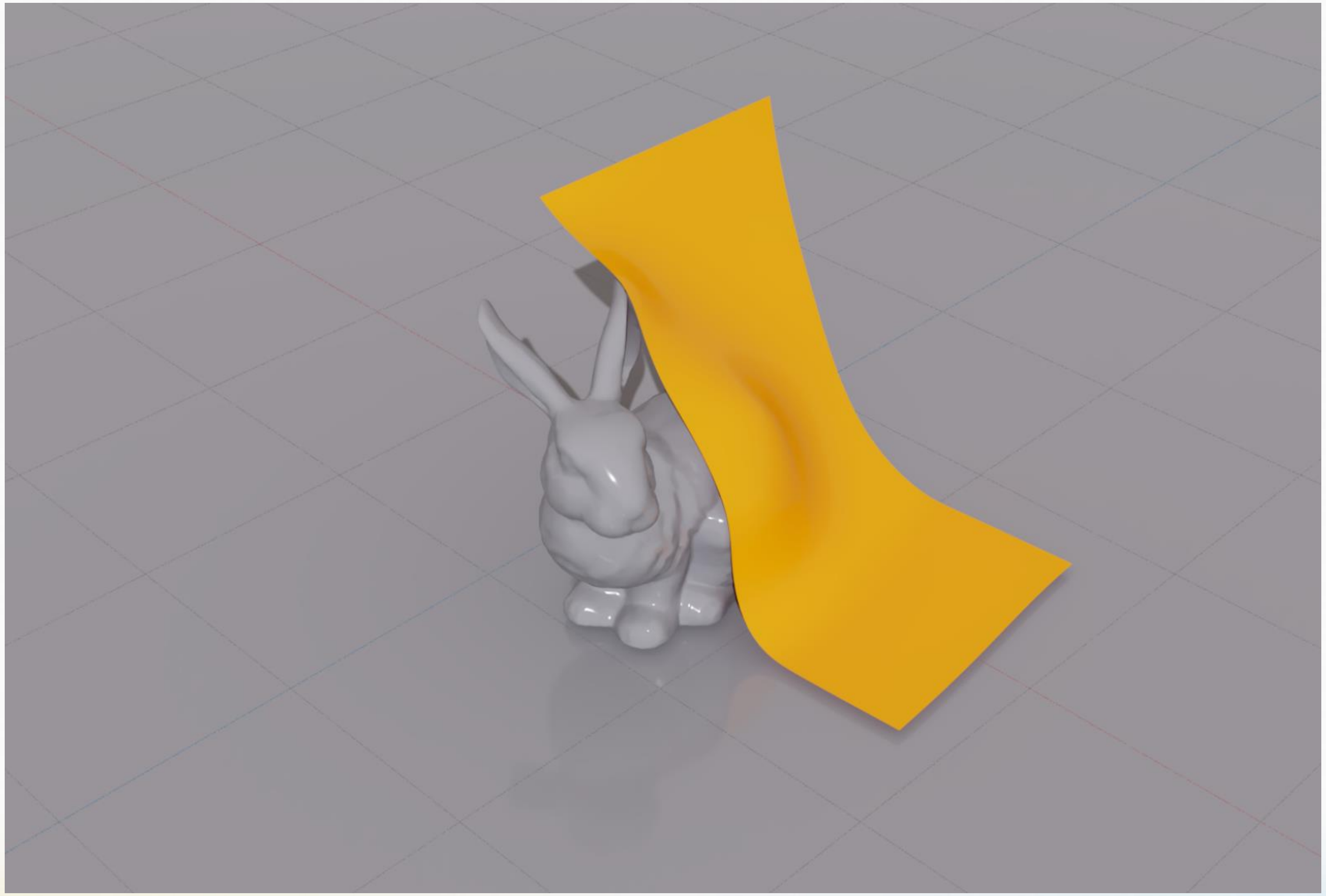


~2600 ms @ RTX3090Ti

USING GRAPH CAPTURES


```python
integrator = wp.sim.SemiImplicitIntegrator()

model = builder.finalize()
model.ground = True
model.soft_contact_ke = 1.0e4
model.soft_contact_kd = 1.0e2

state_0 = model.state()
state_1 = model.state()

renderer = wp.sim.render.SimRenderer(model, OUTPUT_PATH, scaling=40.0)

for _ in range(NUM_FRAMES):
    # Step
    with wp.ScopedTimer("step", dict=profiler):
        wp.sim.collide(model, state_0)

        for _ in range(SUBSTEPS):
            state_0.clear_forces()
            integrator.simulate(model, state_0, state_1, sim_dt)

            # swap states
            (state_0, state_1) = (state_1, state_0)

    sim_time += frame_dt

    # Render
    with wp.ScopedTimer("render"):
        renderer.begin_frame(sim_time)
        renderer.render(state_0)
        renderer.end_frame()

renderer.save()

frame_times = profiler["step"]
print("\nTotal simulation time: {:.2f} ms".format(sum(frame_times)))
```

```python
integrator = wp.sim.SemiImplicitIntegrator()

model = builder.finalize()
model.ground = True
model.soft_contact_ke = 1.0e4
model.soft_contact_kd = 1.0e2

state_0 = model.state()
state_1 = model.state()

renderer = wp.sim.render.SimRenderer(model, OUTPUT_PATH, scaling=40.0)

# Using CUDA graph captures - this only works on CUDA devices
with wp.ScopedCapture() as capture:
    wp.sim.collide(model, state_0)

    for _ in range(SUBSTEPS):
        state_0.clear_forces()
        integrator.simulate(model, state_0, state_1, sim_dt)

        # swap states
        (state_0, state_1) = (state_1, state_0)
graph = capture.graph

for _ in range(NUM_FRAMES):
    # Step
    with wp.ScopedTimer("step", dict=profiler):
        wp.capture_launch(graph)

    sim_time += frame_dt

    # Render
    with wp.ScopedTimer("render"):
        renderer.begin_frame(sim_time)
        renderer.render(state_0)
        renderer.end_frame()

renderer.save()

frame_times = profiler["step"]
print("\nTotal simulation time: {:.2f} ms".format(sum(frame_times)))
```



Before:

~2600 ms @ RTX3090Ti

After:

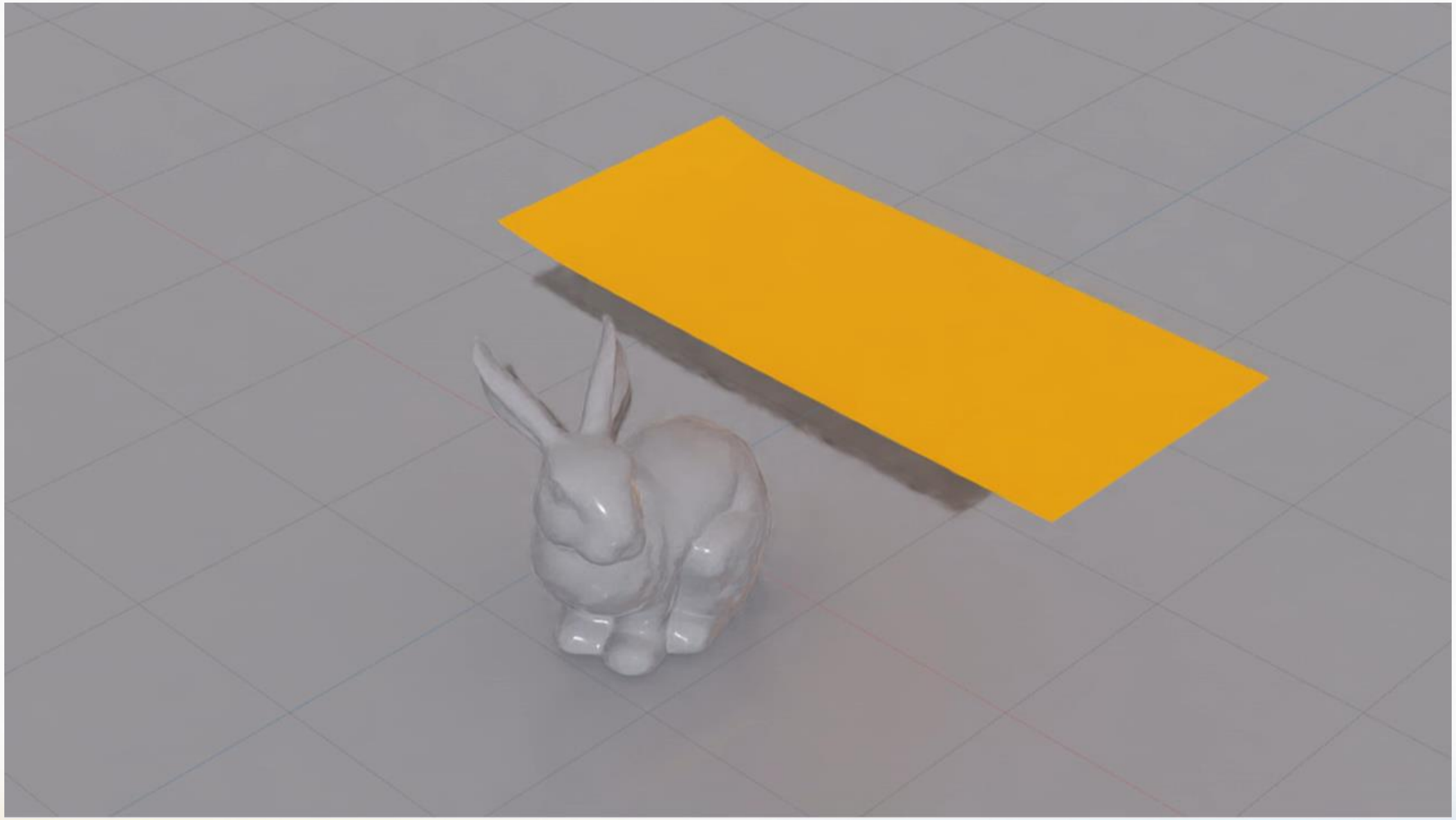
~15 ms @ RTX3090Ti

COMBINING WITH CUSTOM KERNELS


```python
# Add parameters for the cloth
WIDTH = 64
HEIGHT = 32

# Adding the cloth
builder.add_cloth_grid(
    pos=wp.vec3(0.0, 4.0, -1.6),
    rot=wp.quat_from_axis_angle(wp.vec3(1.0, 0.0, 0.0), math.pi * 0.5),
    vel=wp.vec3(0.0, 0.0, 0.0),
    dim_x=WIDTH,
    dim_y=HEIGHT,
    cell_x=0.1,
    cell_y=0.1,
    mass=0.1,
    fix_left=True,
    tri_ke=1.0e3,
    tri_ka=1.0e3,
    tri_kd=1.0e1,
)
```

```python
@wp.kernel
def move_cloth(
    x: wp.array(dtype=wp.vec3),
    x0: wp.array(dtype=wp.vec3),
    time: wp.array(dtype=wp.float32),
):
    tid = wp.tid()
    x[tid * (WIDTH + 1)] = x0[tid * (WIDTH + 1)] + wp.vec3(
        -3.0 * wp.sin(time[0]), 0.0, 0.0
    )
```

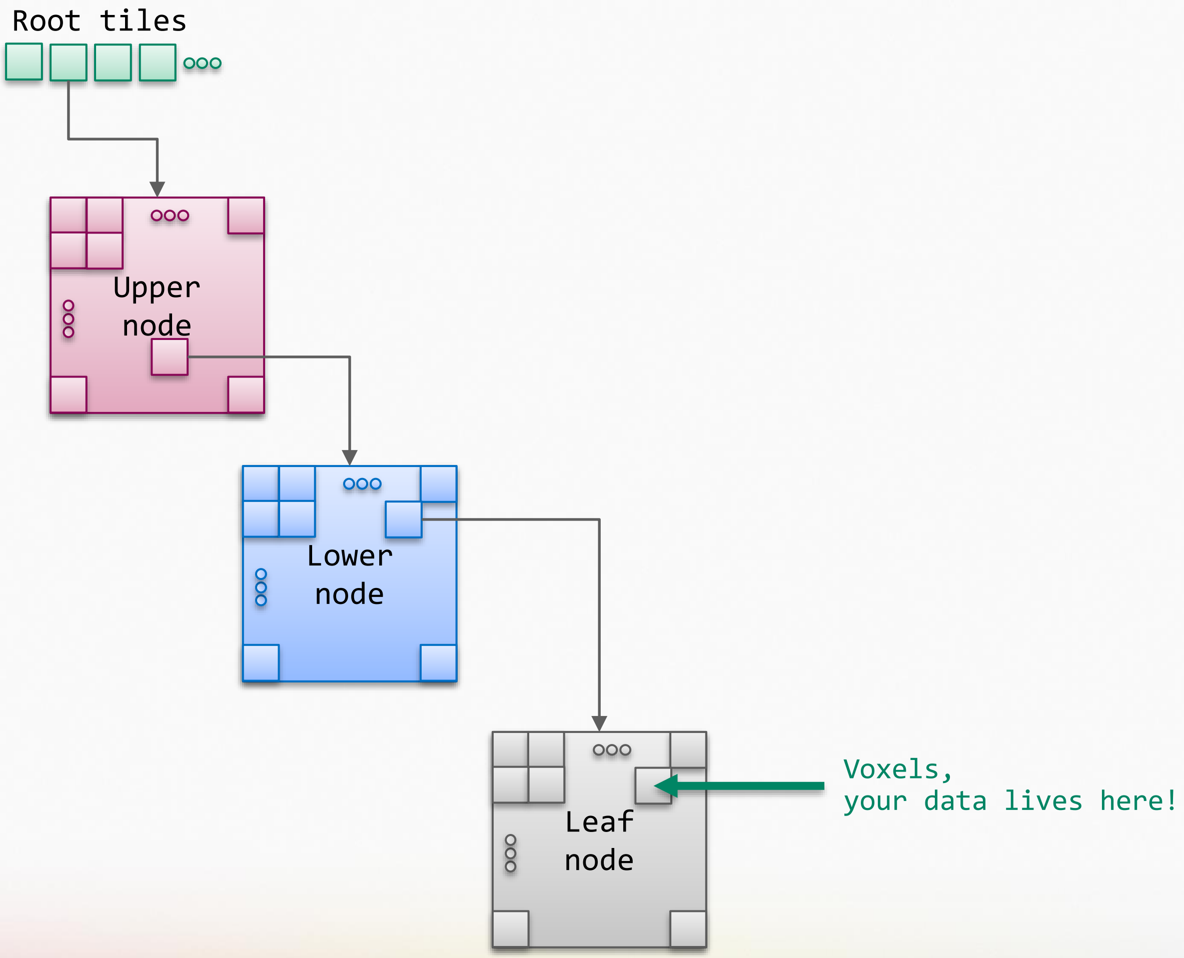
```python
# Using CUDA graph captures - this only works on CUDA devices
with wp.ScopedCapture() as capture:
    wp.sim.collide(model, state_0)
    wp.launch(
        kernel=move_cloth,
        dim=HEIGHT + 1,
        inputs=[state_0.particle_q, x0, time],
    )

    for _ in range(SUBSTEPS):
        state_0.clear_forces()
        integrator.simulate(model, state_0, state_1, sim_dt)

        # swap states
        (state_0, state_1) = (state_1, state_0)
graph = capture.graph

for _ in range(NUM_FRAMES):
    # Step
    with wp.ScopedTimer("step", dict=profiler):
        wp.capture_launch(graph)

    sim_time += frame_dt
    time.fill_(sim_time)

    # Render
    with wp.ScopedTimer("render"):
        renderer.begin_frame(sim_time)
        renderer.render(state_0)
        renderer.end_frame()

renderer.save()

frame_times = profiler["step"]
print("\nTotal simulation time: {:.2f} ms".format(sum(frame_times)))
```



PART 2 – SPARSE VOLUMES

- What you will learn
  - What Volume objects are in Warp
  - How to create and use them

THE NANOVDB BACKEND TO VOLUMES



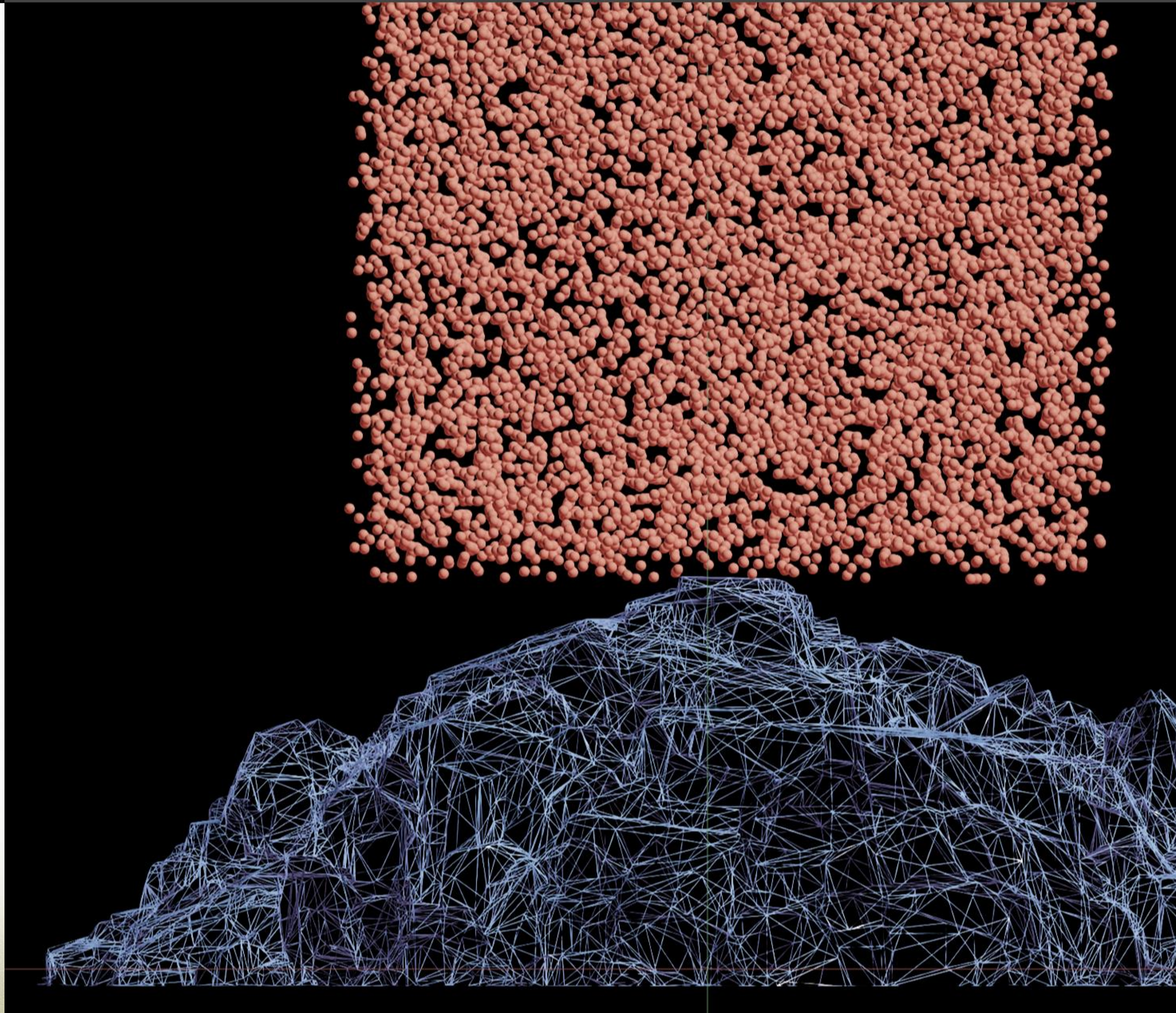

```python
class Volume:
    #: Enum value to specify nearest-neighbor interpolation during sampling
    CLOSEST = constant(0)
    #: Enum value to specify trilinear interpolation during sampling
    LINEAR = constant(1)

    def __init__(self, data: array):
        """Class representing a sparse grid.

        Args:
            data (:class:`warp.array`): Array of bytes representing the volume in NanoVDB format
        """
#...
```

- Warp Volumes are using NanoVDB for storage
- NanoVDB is a hierarchical, sparse data structure
- Individual voxels can be active or inactive
- Smallest unit of allocation is an 8x8x8 block of voxels
- Notation:
  `Volume tile` == NanoVDB leaf node

LOADING AND SAMPLING VOLUMES


```
# load collision volume
with open("rocks.nvdb", "rb") as file:
    # create Volume object
    self.volume =
wp.Volume.load_from_nvdb(file)
```

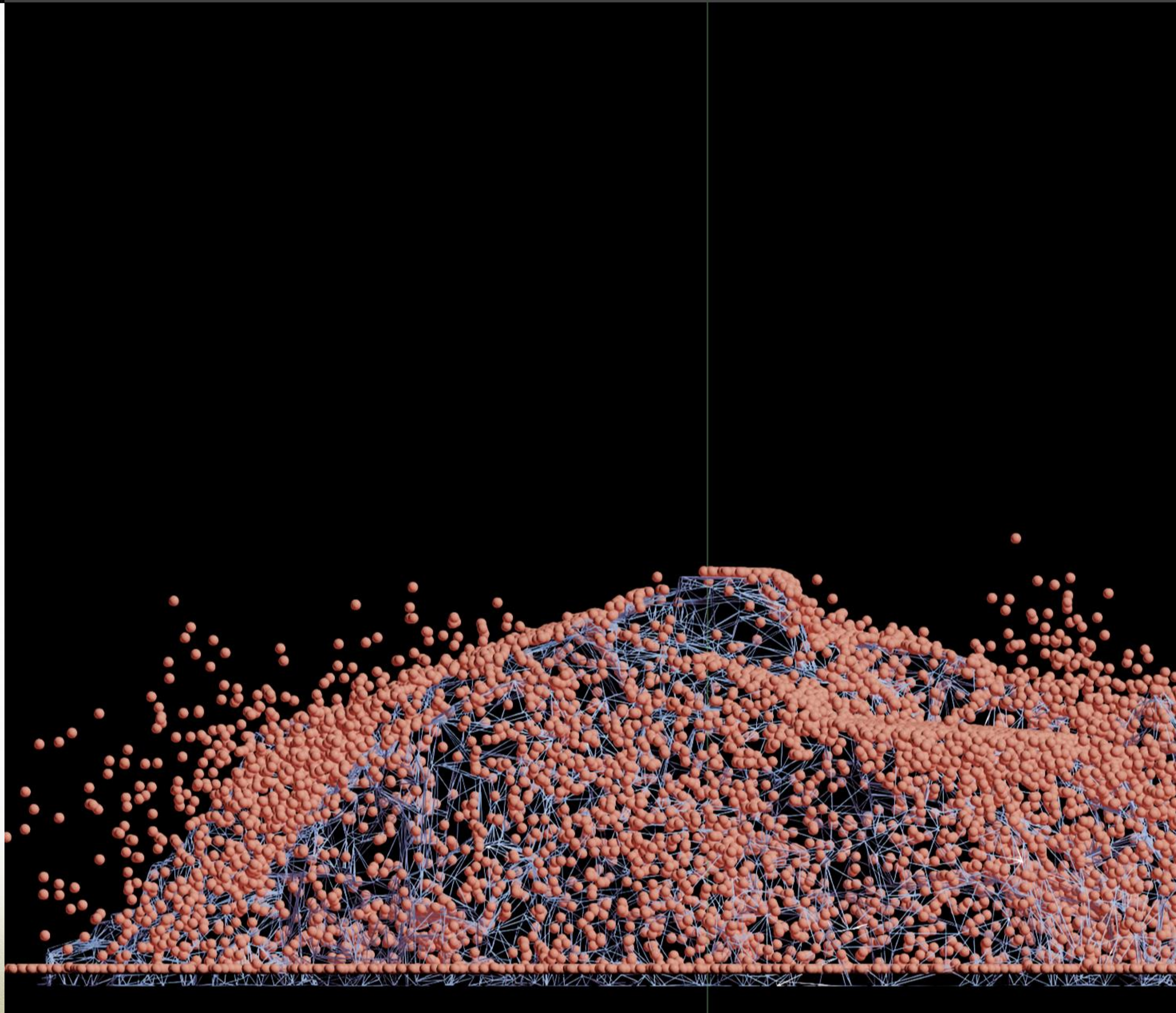
```
wp.launch(
    kernel=simulate,
    dim=self.num_particles,
    inputs=[
        self.positions,
        self.velocities,
        self.volume.id,
        self.sim_margin,
        self.sim_dt],
)
```



LOADING AND SAMPLING VOLUMES

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

```
@wp.kernel
def simulate(
    positions: wp.array(dtype=wp.vec3),
    velocities: wp.array(dtype=wp.vec3),
    volume: wp.uint64,
    margin: float,
    dt: float,
):
    tid = wp.tid()

    x = positions[tid]
    v = velocities[tid]

    v = v + wp.vec3(0.0, -9.8, 0.0) * dt - v * 0.1 * dt
    xpred = x + v * dt
    xpred_local = wp.volume_world_to_index(volume, xpred)

    n = wp.vec3()
    d = wp.volume_sample_grad_f(volume, xpred_local, wp.Volume.LINEAR, n)

    if d < margin:
        n = wp.normalize(n)
        err = d - margin

        # mesh collision
        xpred = xpred - n * err

    # ground collision
    if xpred[1] < 0.0:
        xpred = wp.vec3(xpred[0], 0.0, xpred[2])

    # pbd update
    v = (xpred - x) * (1.0 / dt)
    x = xpred

    positions[tid] = x
    velocities[tid] = v
```



CREATING VOLUMES


```python
class Volume:
    #...
    @classmethod
    def load_from_numpy(
        cls, ndarray: np.array, min_world=(0.0, 0.0, 0.0), voxel_size=1.0, bg_value=0.0, device=None
    ) -> Volume:
    #...


    @classmethod
    def allocate(
        cls,
        min: List[int], max: List[int],
        voxel_size: float,bg_value=0.0, translation=(0.0, 0.0, 0.0),
        points_in_world_space=False,
        device=None,
    ) -> Volume:
    #...


    @classmethod
    def allocate_by_tiles(
        cls, tile_points: array, voxel_size: float, bg_value=0.0, translation=(0.0, 0.0, 0.0), device=None
    ) -> Volume:
```

VOLUMES IN ACTION

A 3-D JACOBI FLUID SOLVER


```
def update(self, frame):
    for i in range(self.sim_substeps):

        dt = self.sim_dt

        # pressure solve
        launch_on_tiles(volume_zero_f, self.tiles, inputs=[self.p0.id])
        launch_on_tiles(volume_zero_f, self.tiles, inputs=[self.p1.id])

        self.solve()
        launch_on_tiles(pressure_correction, self.tiles, inputs=[self.p0.id, self.uvw0.id, self.boundary.id, self.dx])

        # semi-Lagrangian advection
        launch_on_tiles(advect, self.tiles, inputs=[self.uvw0.id, self.uvw1.id, self.rho0.id, self.rho1.id, dt, self.dx])

        launch_on_tiles(apply_dirichlet_velocity_bc, self.tiles, inputs=[self.uvw1.id, self.u_dirichlet.id, self.boundary.id])

        # swap buffers
        (self.uvw0, self.uvw1) = (self.uvw1, self.uvw0)
        (self.rho0, self.rho1) = (self.rho1, self.rho0)

        self.sim_time += dt
```

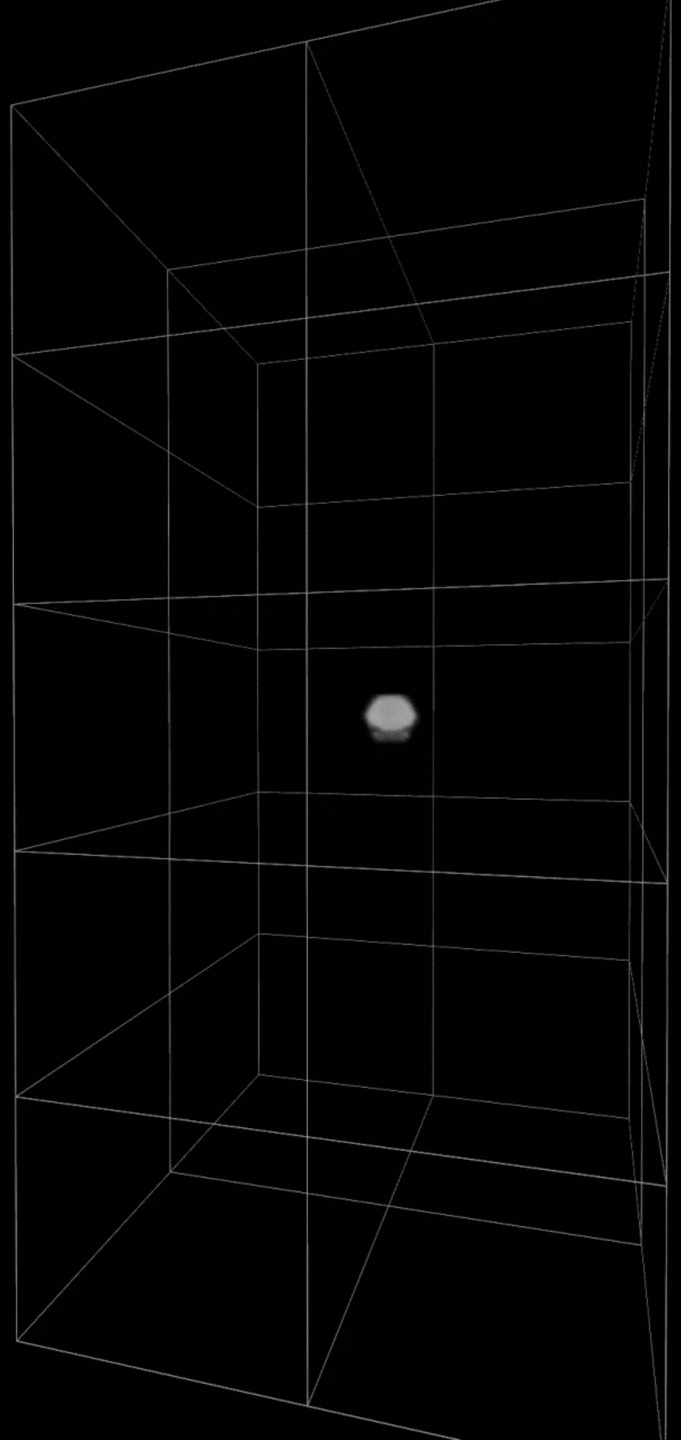
HELPER FUNCTIONS


```python
def launch_on_tiles(
    kernel, tiles, inputs:List, outputs:List=[], adj_inputs:List=[], adj_outputs:List=[],
    device=None, stream=None, adjoint=False
):
    """
        tiles: wp.array, Nx3, on device
    """
    wp.launch(
        kernel, (8, 8, 8, len(tiles)),
        [tiles] + inputs,
        outputs, adj_inputs, adj_outputs, device, stream, adjoint)


# Volume utils
@wp.kernel
def volume_zero_f(tiles: wp.array2d(dtype=wp.int32),
                  v: wp.uint64):
    ti, tj, tk, t = wp.tid()
    i = ti + tiles[t][0]
    j = tj + tiles[t][1]
    k = tk + tiles[t][2]
    wp.volume_store_f(v, i, j, k, 0.0)
```

PRESSURE CORRECTION


```python
@wp.kernel
def pressure_correction(tiles: wp.array2d(dtype=wp.int32),
                        p: wp.uint64,
                        u: wp.uint64,
                        boundary: wp.uint64,
                        voxel_size: float):

    ti, tj, tk, t = wp.tid()
    i = ti + tiles[t][0]
    j = tj + tiles[t][1]
    k = tk + tiles[t][2]

    boundary_ijk = wp.volume_lookup_i(boundary, i, j, k)
    if boundary_ijk == NEUMANN_CELL: # Neumann cells have only Dirichlet faces - nothing to 
update here
        return

    if boundary_ijk == DIRICHLET_CELL:
        p_ijk = 0.0
    else:
        p_ijk = wp.volume_lookup_f(p, i, j, k)

    # Some walls might be along a Dirichlet velocity boundary - must not update those
    boundary_iMjk = wp.volume_lookup_i(boundary, i-1, j, k)
    boundary_ijMk = wp.volume_lookup_i(boundary, i, j-1, k)
    boundary_ijkM = wp.volume_lookup_i(boundary, i, j, k-1)
    if boundary_iMjk == FLUID_CELL:
        dx = p_ijk - wp.volume_lookup_f(p, i-1, j, k)
    elif boundary_iMjk == DIRICHLET_CELL:
        dx = p_ijk
    else:
        dx = 0.0
```

```python
    # pressure_correction cont'd

    if boundary_ijMk == FLUID_CELL:
        dy = p_ijk - wp.volume_lookup_f(p, i, j-1, k)
    elif boundary_ijMk == DIRICHLET_CELL:
        dy = p_ijk
    else:
        dy = 0.0

    if boundary_ijkM == FLUID_CELL:
        dz = p_ijk - wp.volume_lookup_f(p, i, j, k-1)
    elif boundary_ijkM == DIRICHLET_CELL:
        dz = p_ijk
    else:
        dz = 0.0

    # pressure gradient
    grad_p = wp.vec3(dx, dy, dz) / voxel_size

    u_val = wp.volume_lookup_v(u, i, j, k)
    wp.volume_store_v(u, i, j, k, u_val - grad_p)
```



Automatic Differentiation

Miles Macklin, NVIDIA

GOAL

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Minimize a scalar loss function s() w.r.t system parameters $\mathbf{x}(t_0)$

$$s\left(\mathbf{x}(t_1)\right) = s\left(\mathbf{x}(t_0) + \int_{t_0}^{t_1} f\left(\mathbf{x}(t)\right)dt\right)$$

System State:

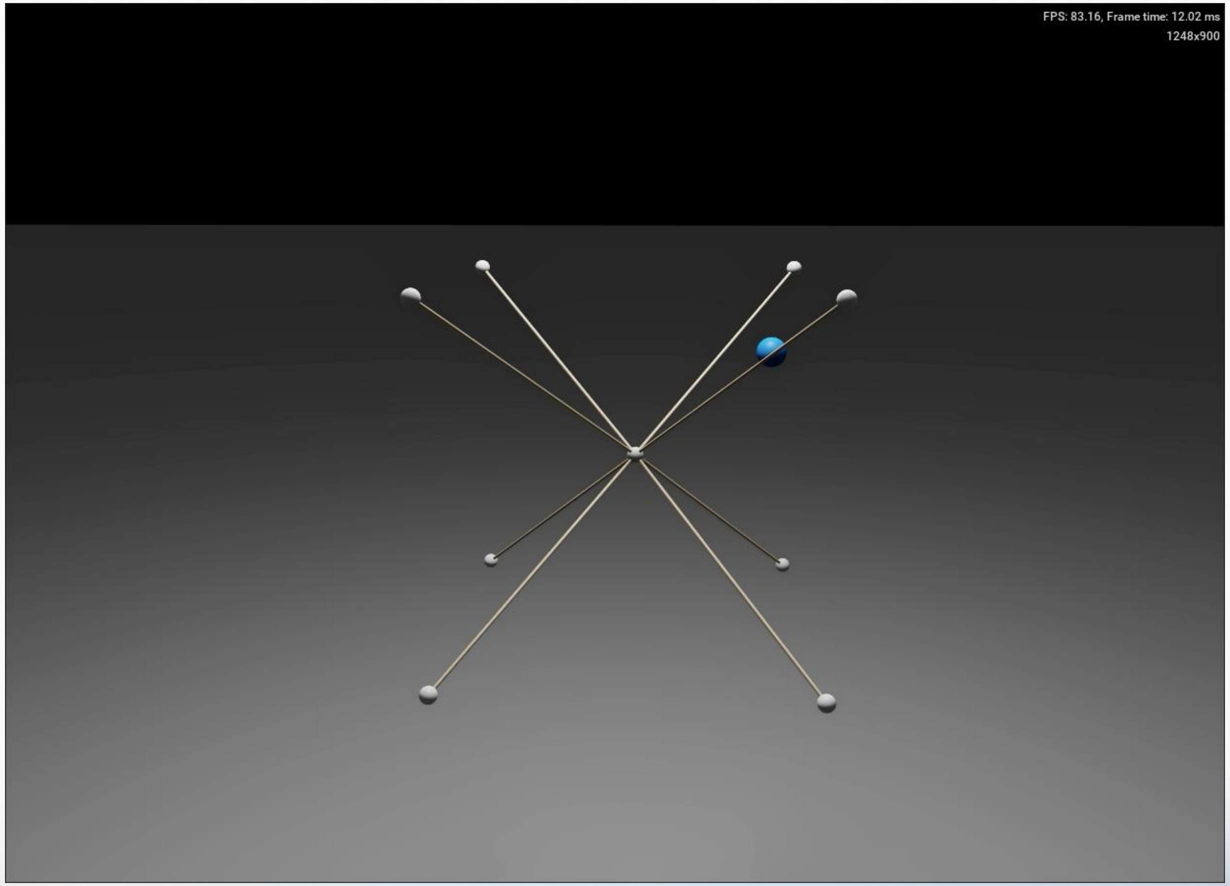
$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \\ \theta \end{bmatrix}$$

Forward ODE:

$$\dot{\mathbf{x}}(t) = f\left(\mathbf{x}(t)\right)$$

EXAMPLE - MASS–SPRING CAGE

- Minimize distance of center particle to target after 2 sec
- Optimize over spring rest lengths
- 3-4 LBFGS iterations



COMPUTING GRADIENTS

- For optimization we want the gradient of scalar loss $s(\mathbf{x})$ at $t = t_0$
- Define the adjoint of a variable as $\mathbf{x}^*$
- Goal: given $\mathbf{x}^*(t_1)$ compute $\mathbf{x}^*(t_0)$

Adjoint Variable

$$\mathbf{x}^*(t) = \frac{\partial s}{\partial \mathbf{x}}^T = \begin{bmatrix} \frac{\partial s}{\partial x_1} \\ \vdots \\ \frac{\partial s}{\partial x_n} \end{bmatrix}$$

$$\mathbf{x}, \mathbf{x}^* \in \mathbb{R}^n$$

CONTINUOUS ADJOINT METHOD

- Computes gradient of scalar loss function via reverse ODE

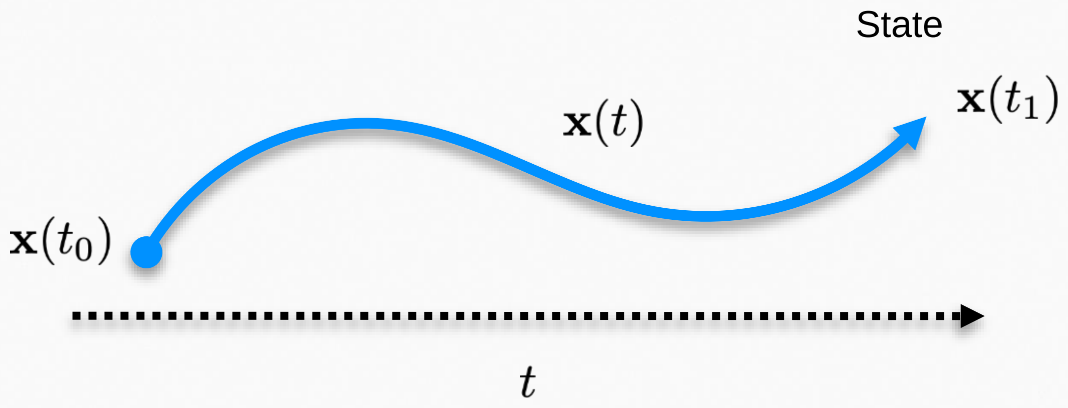
Forward ODE:

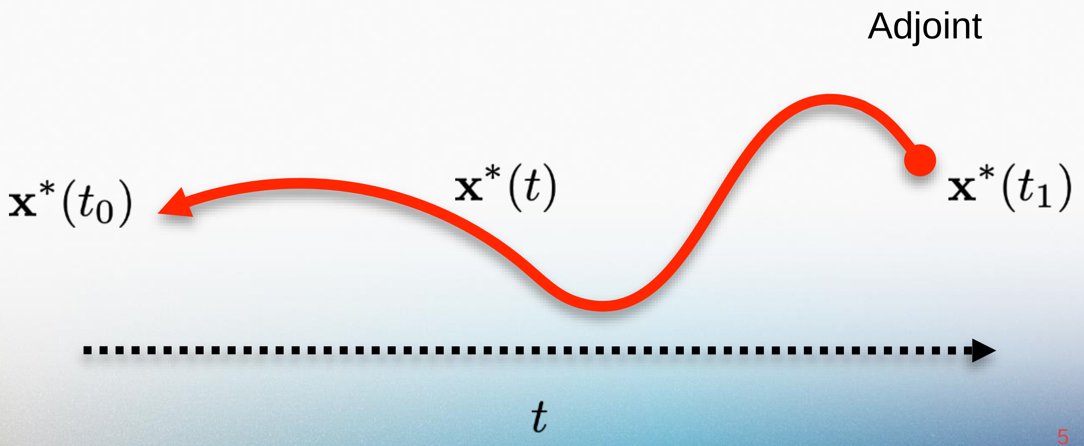
$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

Calculus of Variations

Reverse ODE:

$$\dot{\mathbf{x}}^*(t) = -\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x}^*(t)$$





DISCRETE ADJOINT METHOD

- Replace ODE with time-stepping equations:

$$\mathbf{x}^{t+1} = f(\mathbf{x}^t)$$

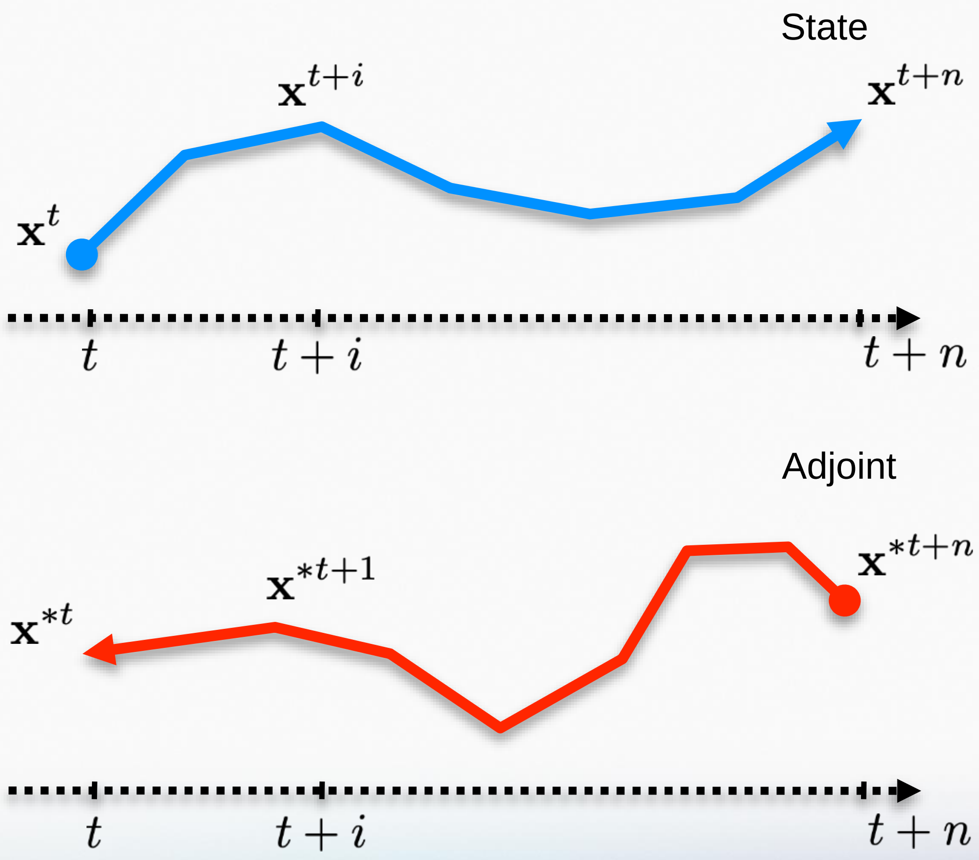
- Discrete trajectory + loss:

$$s(\mathbf{x}^{t+n}) = s(f(f(f(\mathbf{x}^t)))$$

- Apply chain rule:

$$\mathbf{x}^{*t} = \frac{\partial s}{\partial \mathbf{x}}\Big|_{t+0}^T = \frac{\partial f}{\partial \mathbf{x}}\Big|_{t+0}^T \cdot \frac{\partial f}{\partial \mathbf{x}}\Big|_{t+1}^T \cdot \frac{\partial f}{\partial \mathbf{x}}\Big|_{t+2}^T \cdot \frac{\partial s}{\partial \mathbf{x}}\Big|_{t+3}^T$$

- Two ways to evaluate the chain rule..



FORWARD ACCUMULATION (TANGENT MODE)

- Forward:

$$\frac{\partial s(f(g(x)))}{\partial x} = \frac{\partial s}{\partial f}\left(\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}\right)$$

$$s : \mathbb{R}^n \rightarrow \mathbb{R}$$
$$f : \mathbb{R}^m \rightarrow \mathbb{R}^n$$
$$g : \mathbb{R}^p \rightarrow \mathbb{R}^m$$

$$\boxed{\mathbb{R}^{1\times p}} = \boxed{\mathbb{R}^{1\times n}} \cdot \left(\boxed{\mathbb{R}^{n\times m}} \cdot \boxed{\mathbb{R}^{m\times p}} \right)$$

- Evaluate inside->out
- Simple, but large matrix multiplies are expensive
- Use forward mode when outputs >> params (e.g.: vector-valued loss)

REVERSE ACCUMULATION (ADJOINT MODE)

- Reverse:

$$\frac{\partial s(f(g(x))}{\partial x} = \left(\frac{\partial s}{\partial f}\frac{\partial f}{\partial g}\right)\frac{\partial g}{\partial x}$$

$$s : \mathbb{R}^n \to \mathbb{R}$$
$$f : \mathbb{R}^m \to \mathbb{R}^n$$
$$g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\boxed{\mathbb{R}^{1\times p}} = \left(\boxed{\mathbb{R}^{1\times n}} \cdot \boxed{\mathbb{R}^{n\times m}}\right) \cdot \boxed{\mathbb{R}^{m\times p}}$$

- Evaluate outside->in
- Use reverse mode when outputs << params (e.g.: scalar-valued loss)

CONTINUOUS/DISCRETE SIDE-BY-SIDE

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

Continuous Loss:

$$s\left(\mathbf{x}(t_0) + \int_{t_0}^{t_1} f(\mathbf{x}(t))dt\right)$$

Forward ODE:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t))$$

Reverse ODE:

$$\dot{\mathbf{x}}^*(t) = -\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{x}^*(t)$$

Discrete Loss:

$$s(\mathbf{x}^{t+n}) = s(f(f(f(\mathbf{x}^t))$$

Forward Time-stepping:

$$\mathbf{x}^{t+1} = f(\mathbf{x}^t)$$

Reverse Time-stepping:

$${\mathbf{x}^*}^{t-1} = \frac{\partial f}{\partial \mathbf{x}}^T {\mathbf{x}^*}^t$$

ADJOINT OF A FUNCTION

- Given a function:

$$f(\mathbf{x}, \mathbf{y}) \rightarrow \mathbf{z}$$

- Define adjoint (*) as follows:

Adjoint Variable: $\mathbf{z}^* = \frac{\partial s}{\partial \mathbf{z}}^T$

$$f^*(\mathbf{x}, \mathbf{y}, \mathbf{z}^*) \rightarrow (\mathbf{x}^*, \mathbf{y}^*)$$

$$f^*(\mathbf{x}, \mathbf{y}, \mathbf{z}^*) \equiv \left(\frac{\partial f}{\partial \mathbf{x}}^T \mathbf{z}^*, \frac{\partial f}{\partial \mathbf{y}}^T \mathbf{z}^* \right)$$

- Adjoint returns derivative of scalar loss with respect to function inputs

ADJOINT EXAMPLES

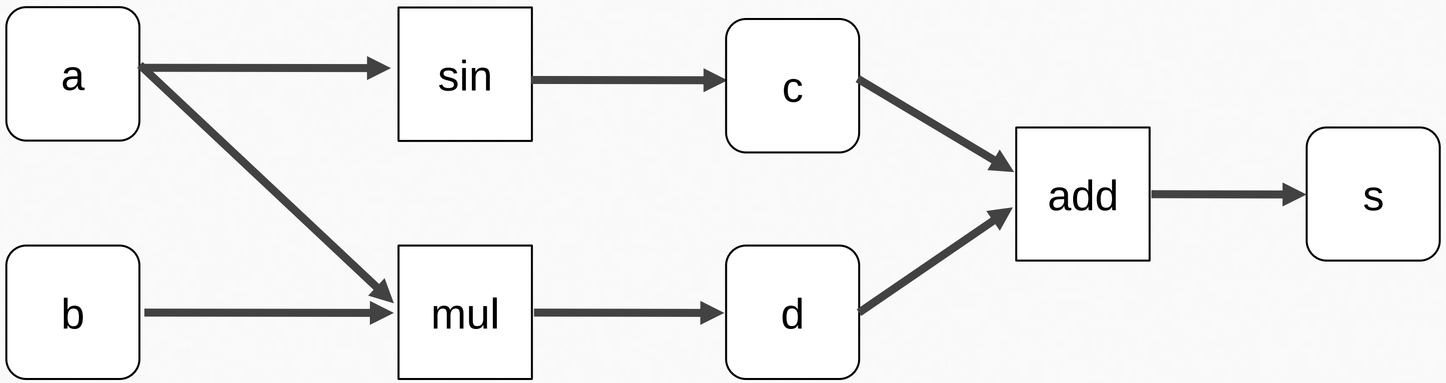
$$z = f(x, y) = x + y \qquad f^*(x, y, z^*) = [z^*, z^*]$$

$$z = f(x, y) = xy \qquad f^*(x, y, z^*) = [yz^*, xz^*]$$

$$z = f(x) = sin(x) \qquad f^*(x, z^*) = [cos(x)z^*]$$

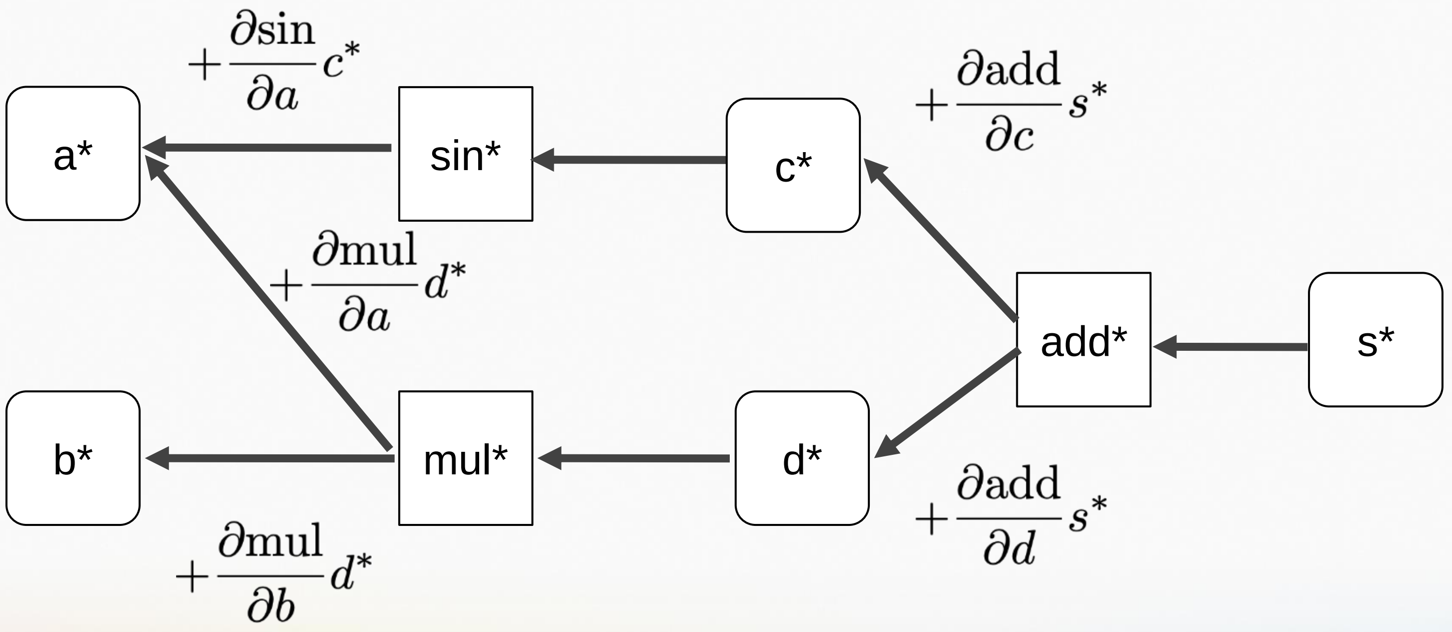
REVERSE MODE AUTO. DIFF

- Example: $s(a, b) = \sin(a) + ab$
- Forward evaluation graph:



REVERSE MODE AUTO. DIFF

- Example: $s(a, b) = \sin(a) + ab$



Solution:

$$\frac{\partial s}{\partial a} = \cos(a) + b$$
$$\frac{\partial s}{\partial b} = a$$

Adjoint Variables:

$$a^* = \frac{\partial s}{\partial a}^T$$

Seed Variable:

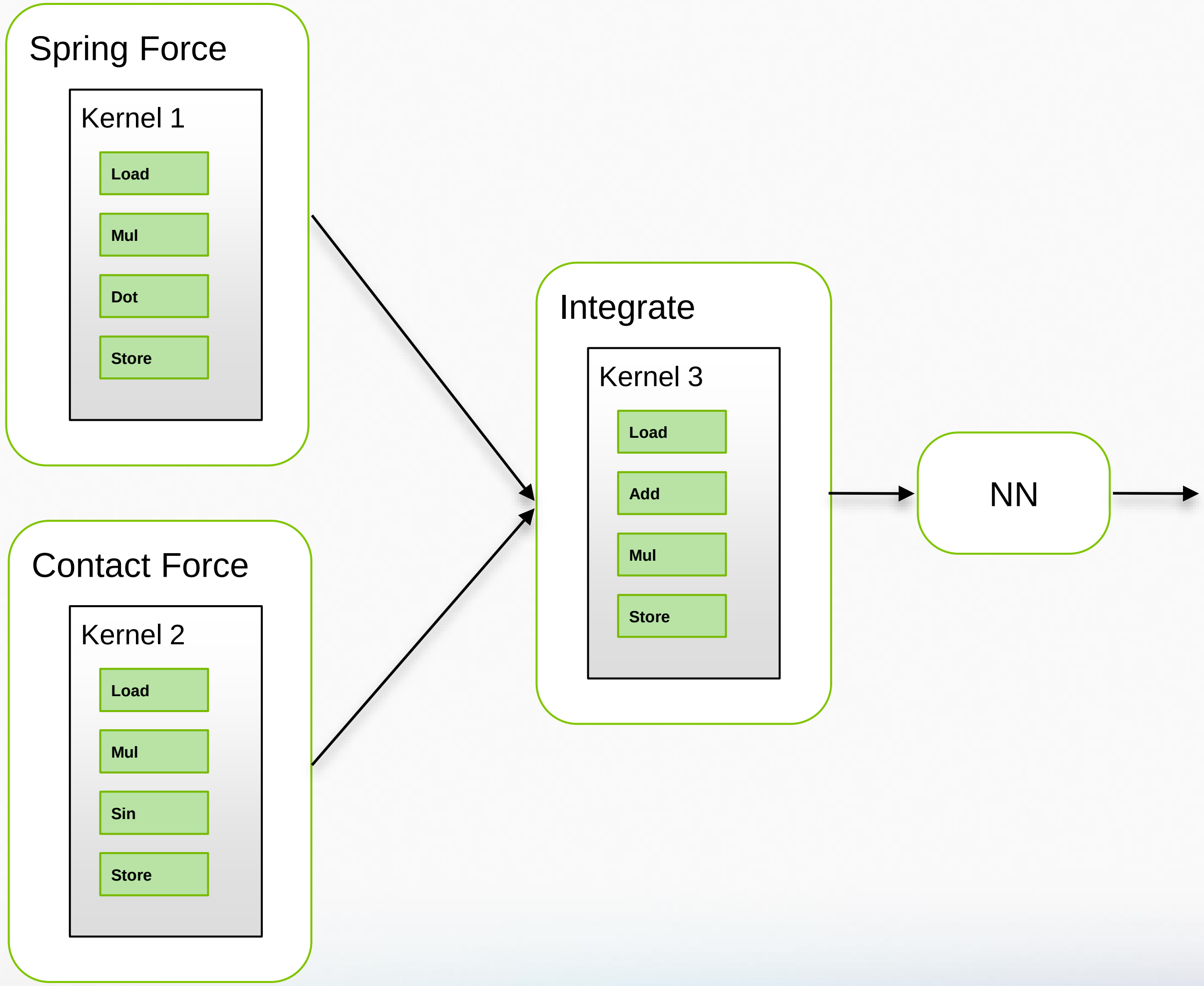
$$s^* = \frac{\partial s}{\partial s}^T = 1$$

AUTODIFF FRAMEWORKS

- Graph Evaluation
  - Runtime
  - Functional, tensor centric
  - PyTorch, TensorFlow
- Program Transformation
  - Compile time
  - Imperative, thread centric
  - DiffTaichi, Google Tangent, Tapenade, dFlex
- Symbolic
  - Expression rewriting
  - Matlab, Mathematica, Maple

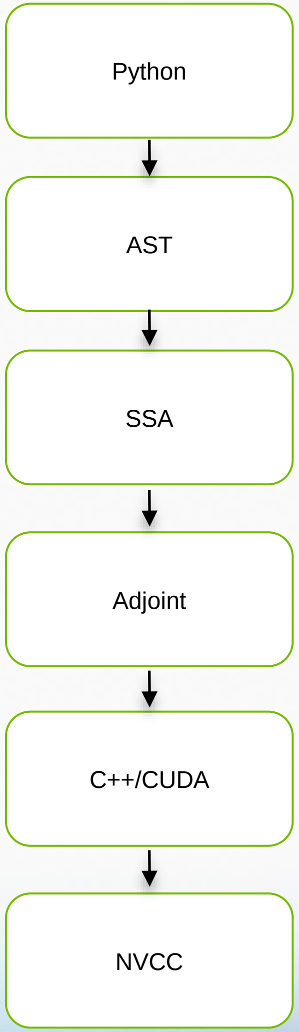
3-LEVEL AUTO DIFF

- Top level
  - computation graph + tape
  - e.g.: loss functions, NN model
- Middle level
  - forward/backward kernels
  - e.g.: force evaluation
- Bottom level
  - mathematical primitives
  - e.g.: sin, cos, dot, cross, etc



PROGRAM TRANSFORMATION

- Middle level auto-diff
- Given abstract syntax tree generate a function’s adjoint:
  - Traverse tree (`import ast`)
  - Convert to static single assignment (SSA) form
  - Run function forward (recording state)
  - Run function backward (accumulate gradients)



SIMPLE EXAMPLE

- Python->C++ SSA
- State is local in registers
- Kernel fusion is implicit
- Flexible indexing
- Gather/Scatter Ops
- Runtime JIT compilation

```python
@wp.func
def simple(a : float, b: float):
    return wp.sin(a) + a*b
```

↓

```cpp
void adj_simple(
    float var_a,
    float var_b,
    float& adj_a,
    float& adj_b,
    float adj_s)
{
    //---------
    // dual vars
    float adj_0 = 0.0f;
    float adj_1 = 0.0f;
    float adj_2 = 0.0f;
    //---------
    // forward
    float var_0 = wp::sin(var_a);
    float var_1 = wp::mul(var_a, var_b);
    float var_2 = wp::add(var_0, var_1);
    //---------
    // reverse
    wp::adj_add(var_0, var_1, adj_0, adj_1, adj_2);
    wp::adj_mul(var_a, var_b, adj_a, adj_b, adj_1);
    wp::adj_sin(var_a, adj_a, adj_s);
}
```

COMPLEX KERNEL

- Example: Closest-point on triangle function
- Conditional logic
- Multiple early-outs
- Pain for tensor-frameworks:
  - Hand-convert to conditional selects/masks
  - Unreliable and opaque kernel fusion
  - Poor support for scattered writes/atomic accumulate

```
@wp.func
def triangle_closest_point(a: wp.vec3, b: wp.vec3, c: wp.vec3, p: wp.vec3):
    ab = b - a
    ac = c - a
    ap = p - a

    d1 = wp.dot(ab, ap)
    d2 = wp.dot(ac, ap)

    if (d1 <= 0.0 and d2 <= 0.0):
        return wp.vec3(1.0, 0.0, 0.0)

    bp = p - b
    d3 = wp.dot(ab, bp)
    d4 = wp.dot(ac, bp)

    if (d3 >= 0.0 and d4 <= d3):
        return wp.vec3(0.0, 1.0, 0.0)

    vc = d1 * d4 - d3 * d2
    v = d1 / (d1 - d3)
    if (vc <= 0.0 and d1 >= 0.0 and d3 <= 0.0):
        return wp.vec3(1.0 - v, v, 0.0)

    cp = p - c
    d5 = wp.dot(ab, cp)
    d6 = wp.dot(ac, cp)

    if (d6 >= 0.0 and d5 <= d6):
        return wp.vec3(0.0, 0.0, 1.0)

    vb = d5 * d2 - d1 * d6
    w = d2 / (d2 - d6)
    if (vb <= 0.0 and d2 >= 0.0 and d6 <= 0.0):
        return wp.vec3(1.0 - w, 0.0, w)

    va = d3 * d6 - d5 * d4
    w = (d4 - d3) / ((d4 - d3) + (d5 - d6))
    if (va <= 0.0 and (d4 - d3) >= 0.0 and (d5 - d6) >= 0.0):
        return wp.vec3(0.0, w, 1.0 - w)

    denom = 1.0 / (va + vb + vc)
    v = vb * denom
    w = vc * denom

    return wp.vec3(1.0 - v - w, v, w)
```

PROGRAM TRANSFORMATION - ALGORITHM

- Given an AST (`import ast`)
- Recursive program to generate the adjoint of a function
- Assume that each node knows how to compute f, f*

```
import ast

forward_ops = []
reverse_ops = []

eval(ast.Node):

    # evaluate inputs
    inputs = []
    for each input i in node.f:
        inputs.push_back(eval(i))

    # output
    forward_ops.push_back{var_n = node.f(inputs)}
    reverse_ops.push_front{[adj_j, adj_k, …] += node.fadj(inputs)}

    return var_n
```

VERIFICATION

- Check gradients via finite differencing
- torch.autograd.gradcheck
- Call function adjoint with each basis vector to evaluate full Jacobian

$$\mathbf{J}_i^T = f^*(\delta_i)$$

- *n* calls, one for each output

wp.Tape()

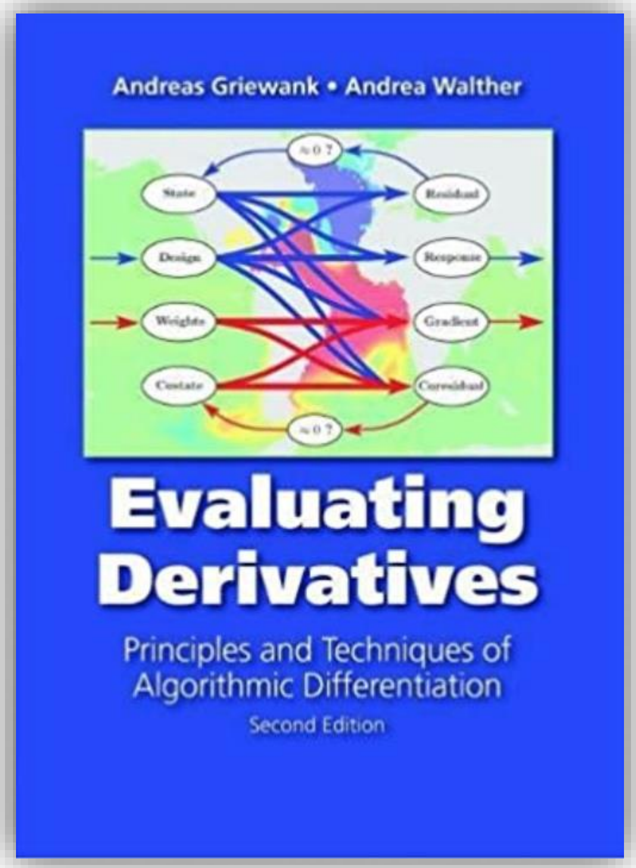
- Reverse-mode automatic differentiation
- Kernel adjoint codes are generated automatically
- Store and reply kernels using wp.Tape()
- User-provided custom gradients
- Limitations:
  - First-order derivatives only
  - Dynamic loops and side-effects
- Capture entire backward pass in a CUDA graph

```
tape = wp.Tape()

with tape:
    wp.launch(kernel=kernel_1, dim=dim, inputs=[a], outputs=[b])
    wp.launch(kernel=kernel_2, dim=dim, inputs=[b], outputs=[c])
    wp.launch(kernel=kernel_3, dim=dim, inputs=[c], outputs=[loss])

# run backward
tape.backward(loss)

# gradients of loss w.r.t inputs
print(a.grad)
print(b.grad)
print(c.grad)
```

Example: Auto-differentiation through multiple kernel launches

FURTHER READING

- [Griewank & Walther 2008]
- Covers program transformation approach in detail
- Many more optimizations are possible
- I rely on NVCC to do the heavy lifting

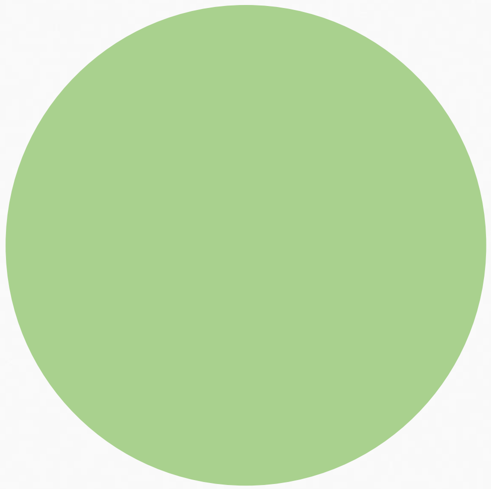


Using Warp in Machine Learning and Optimization Problems

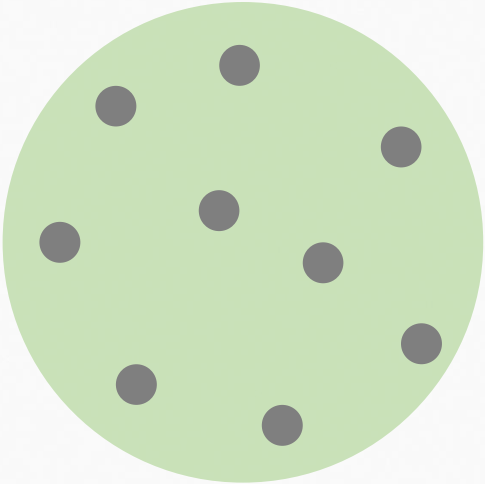
Neural Stress Fields for Reduced-order Elastoplasticity and Fracture

Peter Yichen Chen, MIT CSAIL

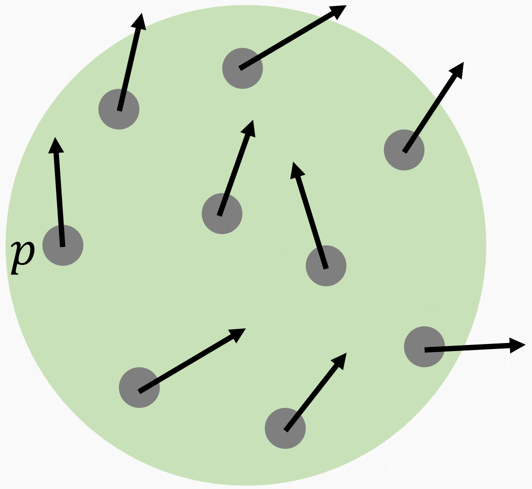
A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)



A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)

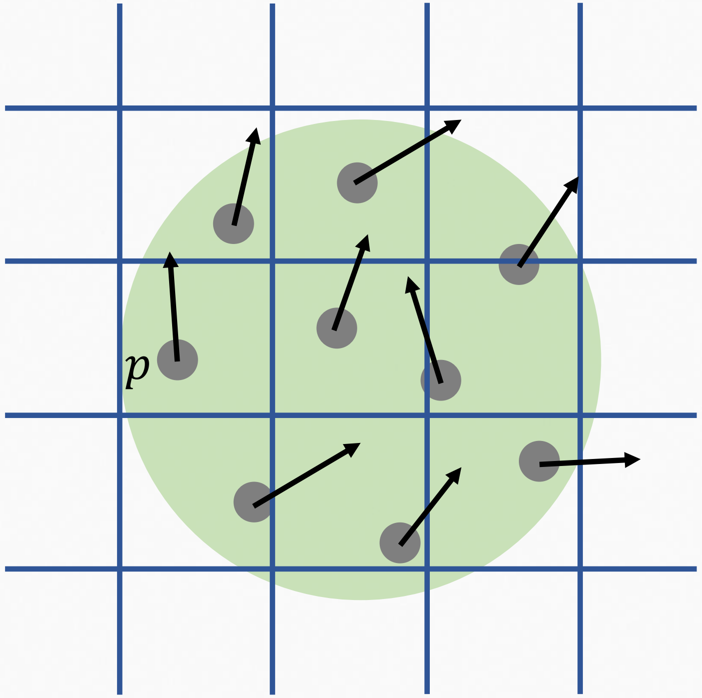


A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)



Each particle p has position x_p, mass m_p, and velocity v_p.

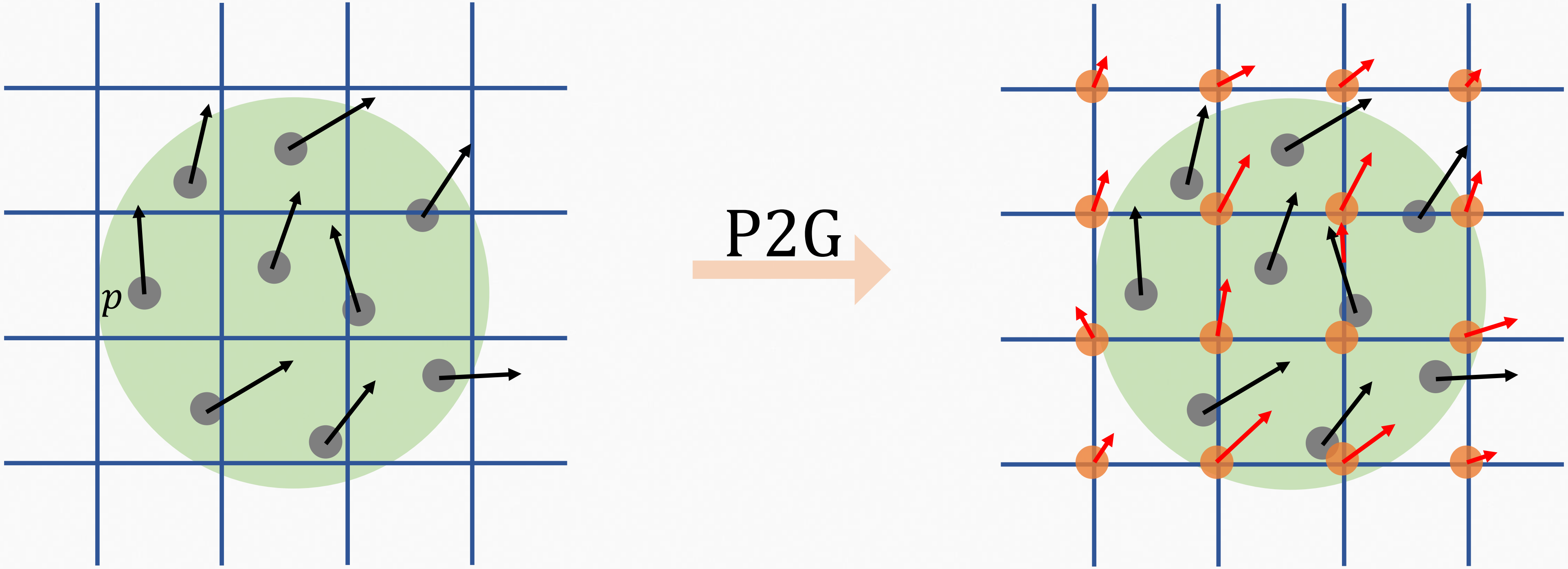
A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)



Each particle p has position x_p, mass m_p, and velocity v_p.

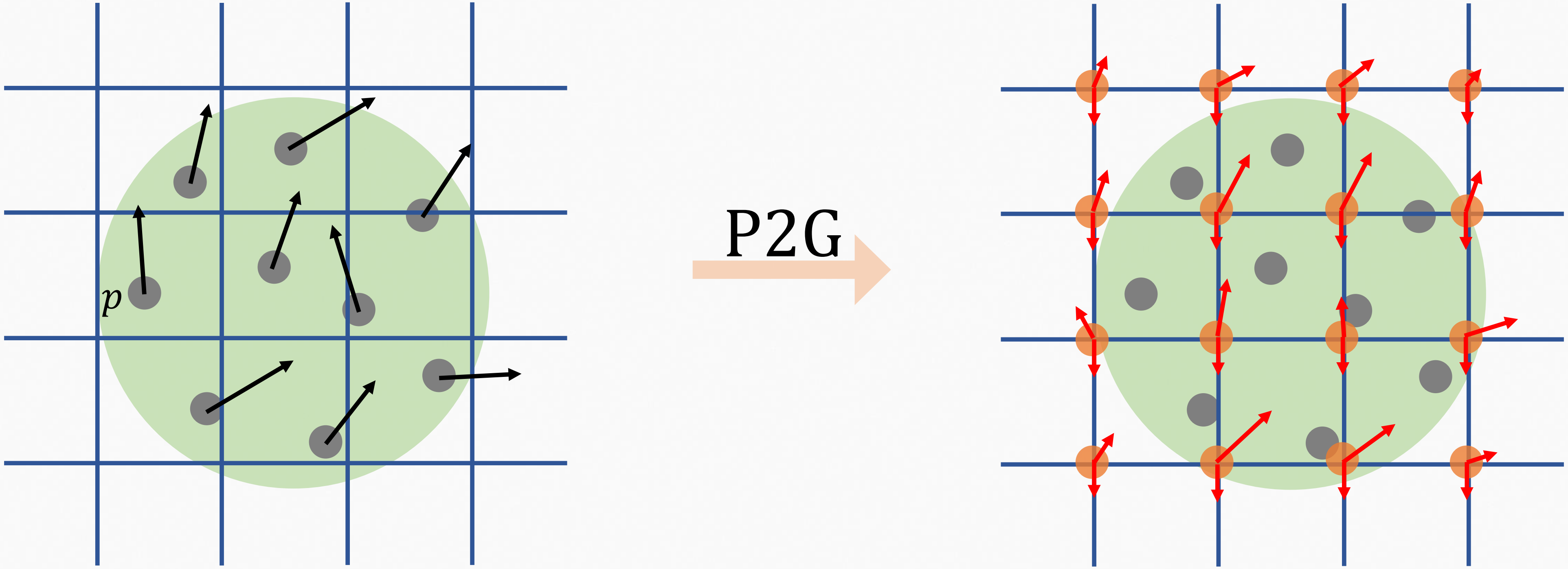
A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG



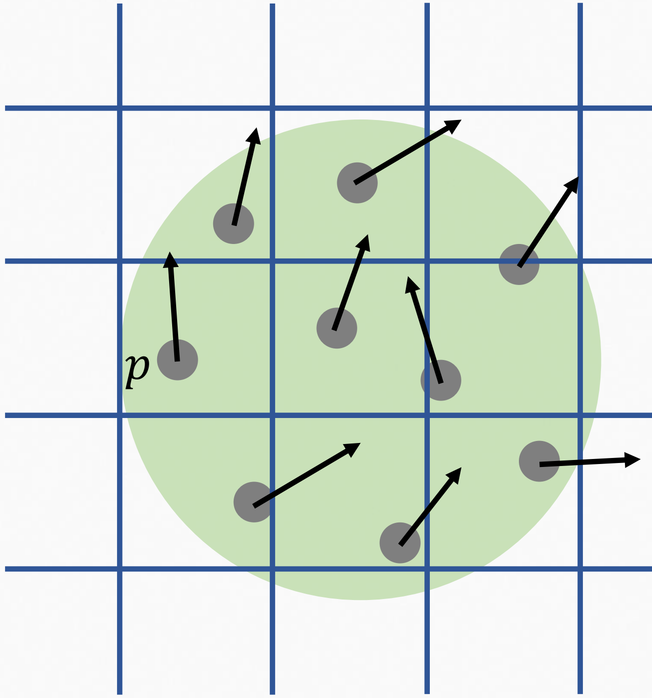
Each particle p has position x_p, mass m_p, and velocity v_p.

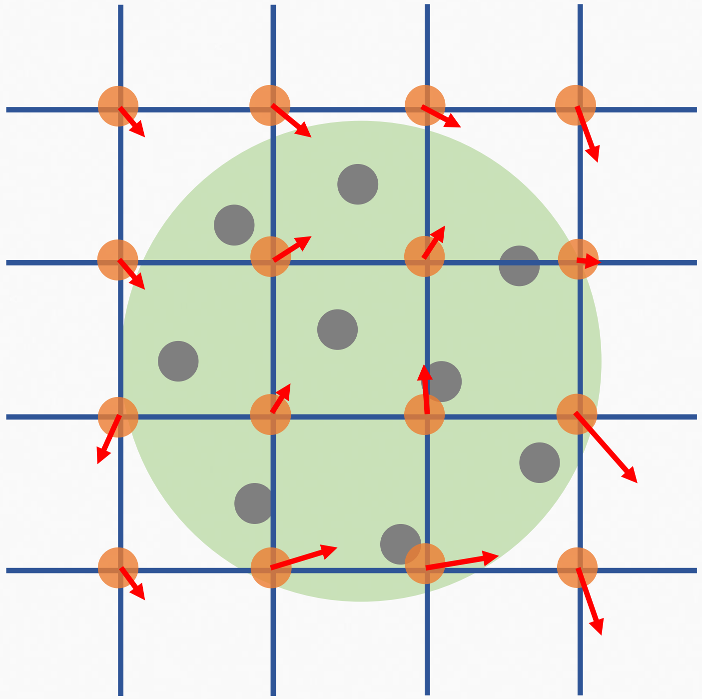
A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)



Each particle p has position x_p, mass m_p, and velocity v_p.

A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)

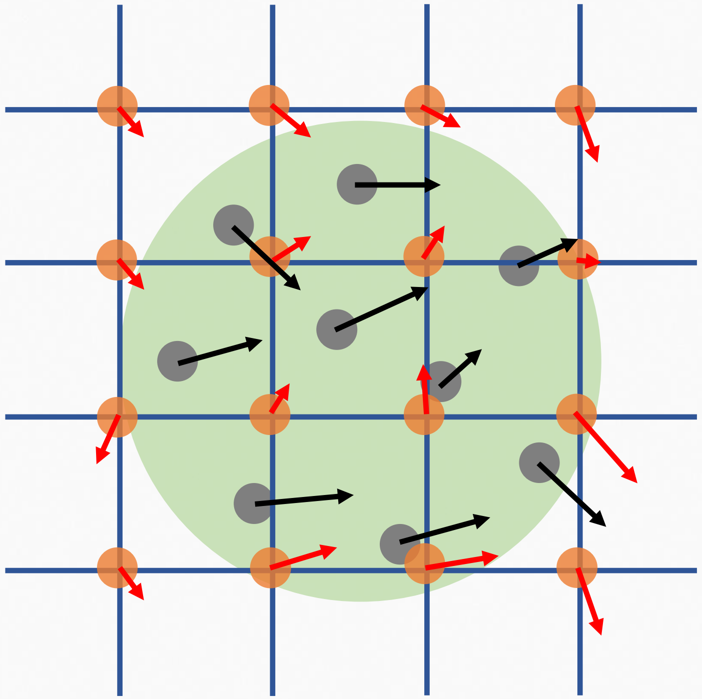




Each particle p has position x_p, mass m_p, and velocity v_p.

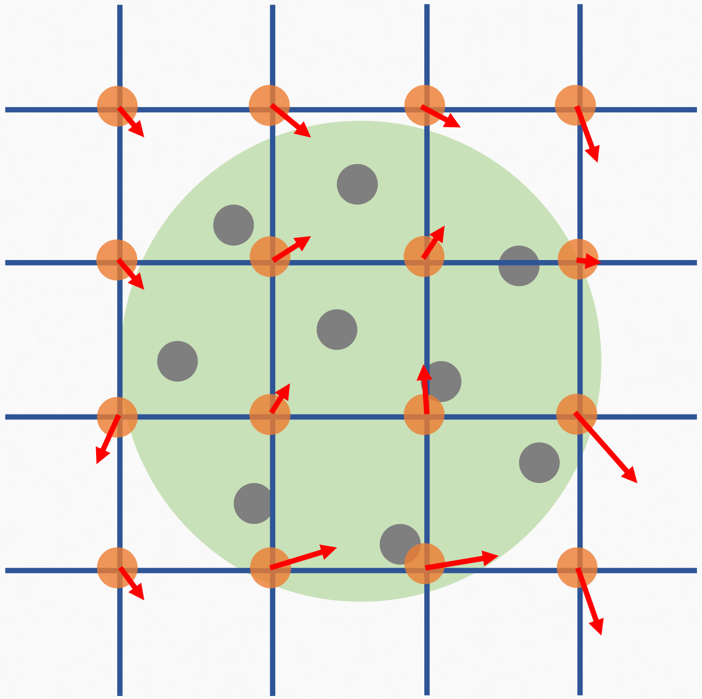
Time integration to obtain new grid velocities.

A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)



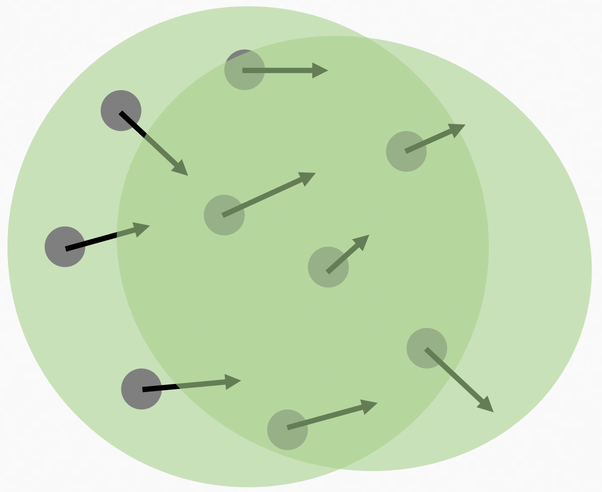
Obtain to particle velocities v_p.

G2P

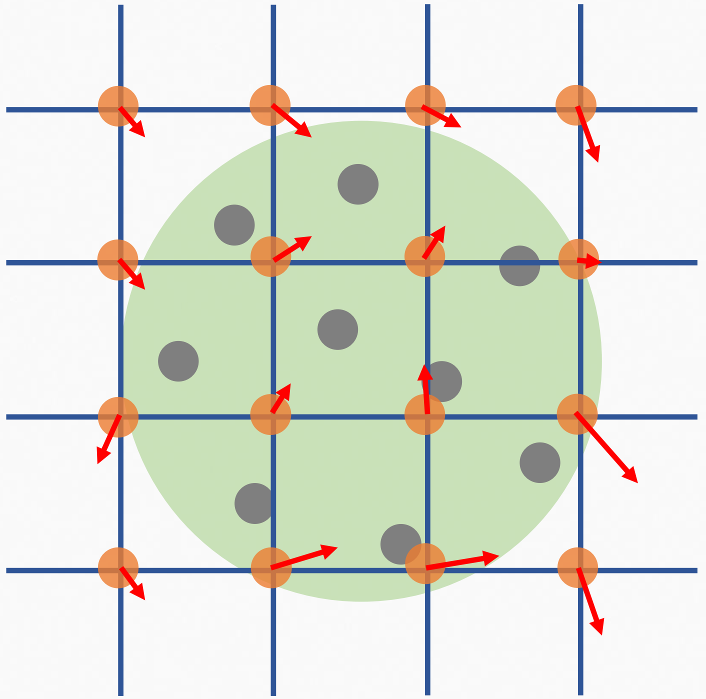


Time integration to obtain new grid velocities.

A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)



G2P



Obtain to particle velocities v_p.

Particles move to new positions x_p.

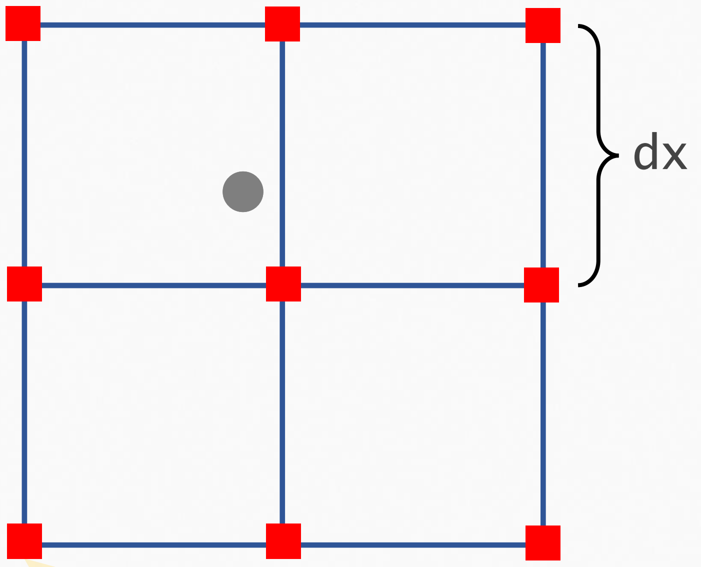
Time integration to obtain new grid velocities.

A BRIEF INTRO TO MATERIAL POINT METHOD (MPM)

- More details on MPM: Jiang, Chenfanfu, et al. "The material point method for simulating continuum materials." *ACM SIGGRAPH 2016 courses*.
- MPM can be efficiently implemented in Warp
  - We are moving each particle
  - Particle-wise operations can be easily parallelized in Warp

IMPLEMENT MPM IN WARP


```python
@wp.kernel
def p2g(state: MPMStateStruct, model: MPMModelStruct):
  p = wp.tid() # thread, also particle index

  grid_pos = state.particle_x[p] * model.inv_dx
  base_idx_x = wp.int(grid_pos[0] - 0.5)
  base_idx_y = wp.int(grid_pos[1] - 0.5)
  base_idx_z = wp.int(grid_pos[2] - 0.5)
  w = wp.mat33(...) # 3by3 weight matrix

  for i in range(0, 3):
    for j in range(0, 3):
      for k in range(0, 3):
        ix = base_idx_x + i
        iy = base_idx_y + j
        iz = base_idx_z + k
        weight = w[0, i] * w[1, j] * w[2, k] # interpolation weight
        wp.atomic_add(state.grid_m, ix, iy, iz, weight * state.particle_mass[p])
        wp.atomic_add(state.grid_v, ix, iy, iz,
             weight * state.particle_mass[p] * state.particle_v[p])
```



Indexed by [base_idx_x, base_idx_y]

IMPLEMENT MPM IN WARP


```python
def advance_one_timestep(...):
  # other operations ...
  # apply p2g
  wp.launch(
      kernel=p2g,
      dim=n_particles,
      inputs=[mpm_state, mpm_model],
      device="cuda")
  # other operations ...
  # ...
```

Set `dim` to be total number of particles.

Then inside at `p = wp.tid()`,
`p ∈ {0,1,2,…,n_particles-1}`

IMPLEMENT MPM IN WARP

- Open-sourced MPM solver:

  https://github.com/zeshunzong/warp-mpm

- Research based on our solver:

  Neural Stress Fields for Reduced-order Elastoplasticity and Fracture (Siggraph Asia 2023)

  PhysGaussian: Physics-Integrated 3D Gaussians for Generative Dynamics (CVPR 2024)

  DreamPhysics: Learning Physical Properties of Dynamic 3D Gaussians from Video Diffusion Priors

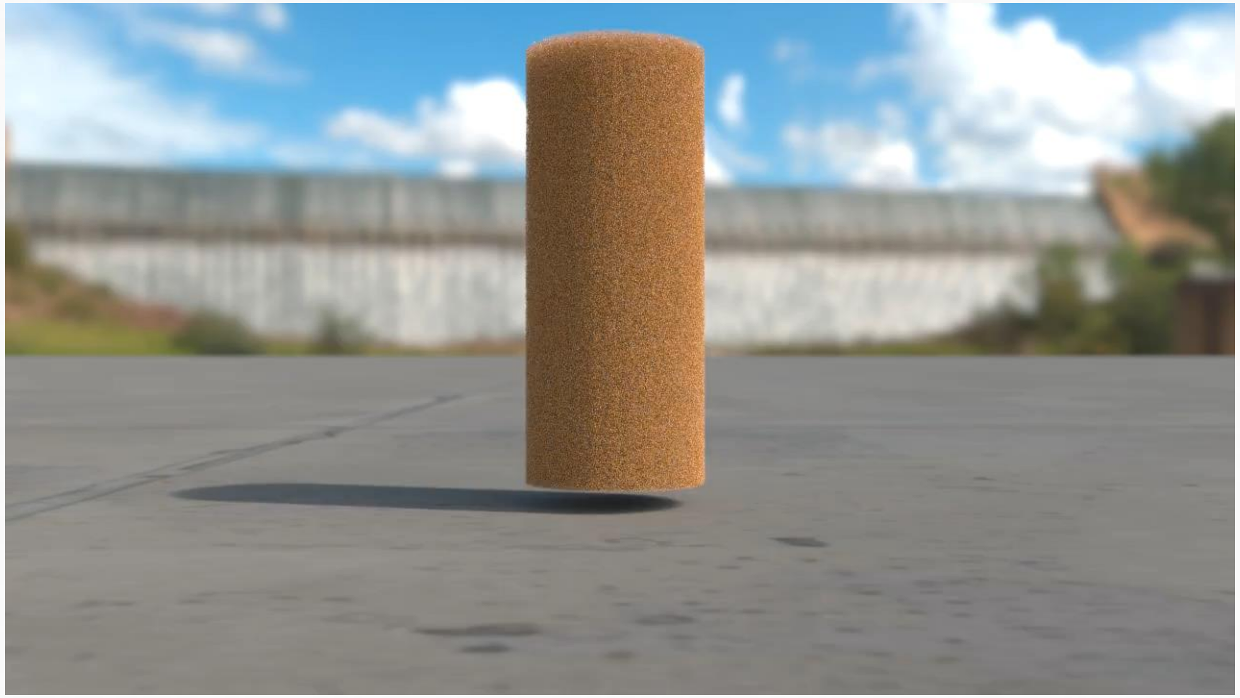
  PhysDreamer: Physics-Based Interaction with 3D Objects via Video Generation

  …

The goal is to

- accelerate MPM simulation
- reduce memory cost

via some reduced-order methods.

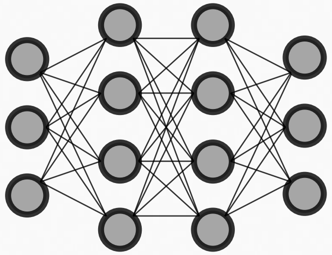


P : the total # of particles.
Need to store the state variables of all P particles!

METHOD

dim=3P

Dimension reduction
via neural fields



dim=5~8.

METHOD

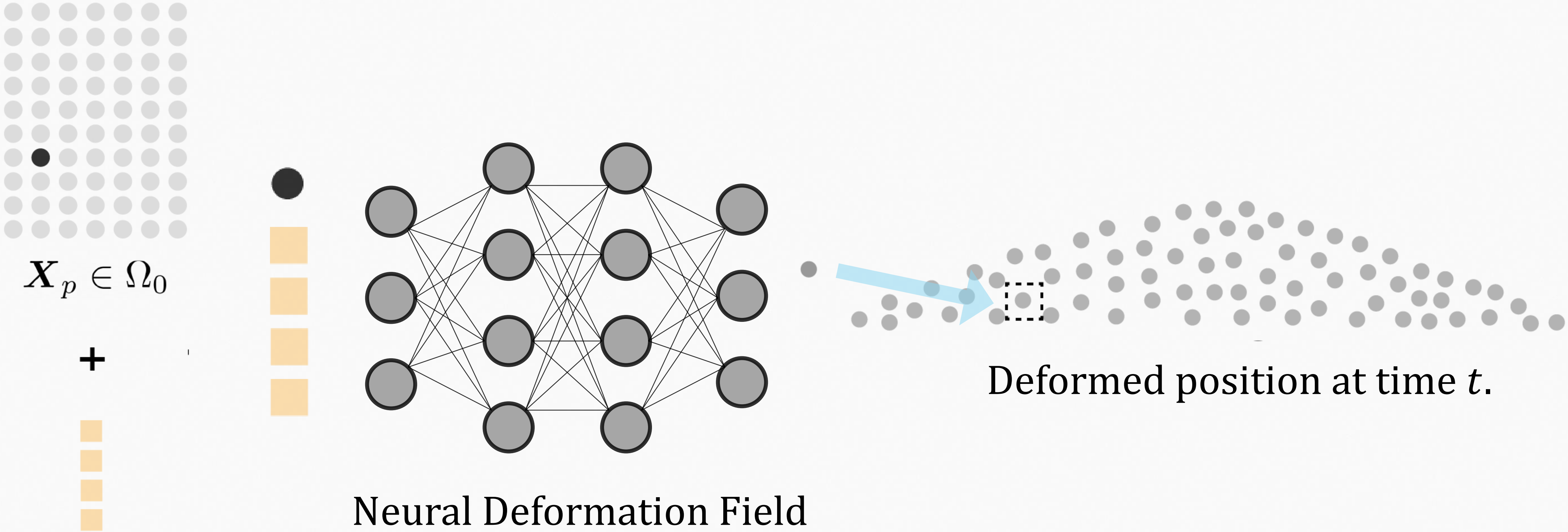
Position $\boldsymbol{x}$

Stress $\boldsymbol{\tau}$

Affine momentum $\boldsymbol{C}$

They are all you need for MPM time stepping!

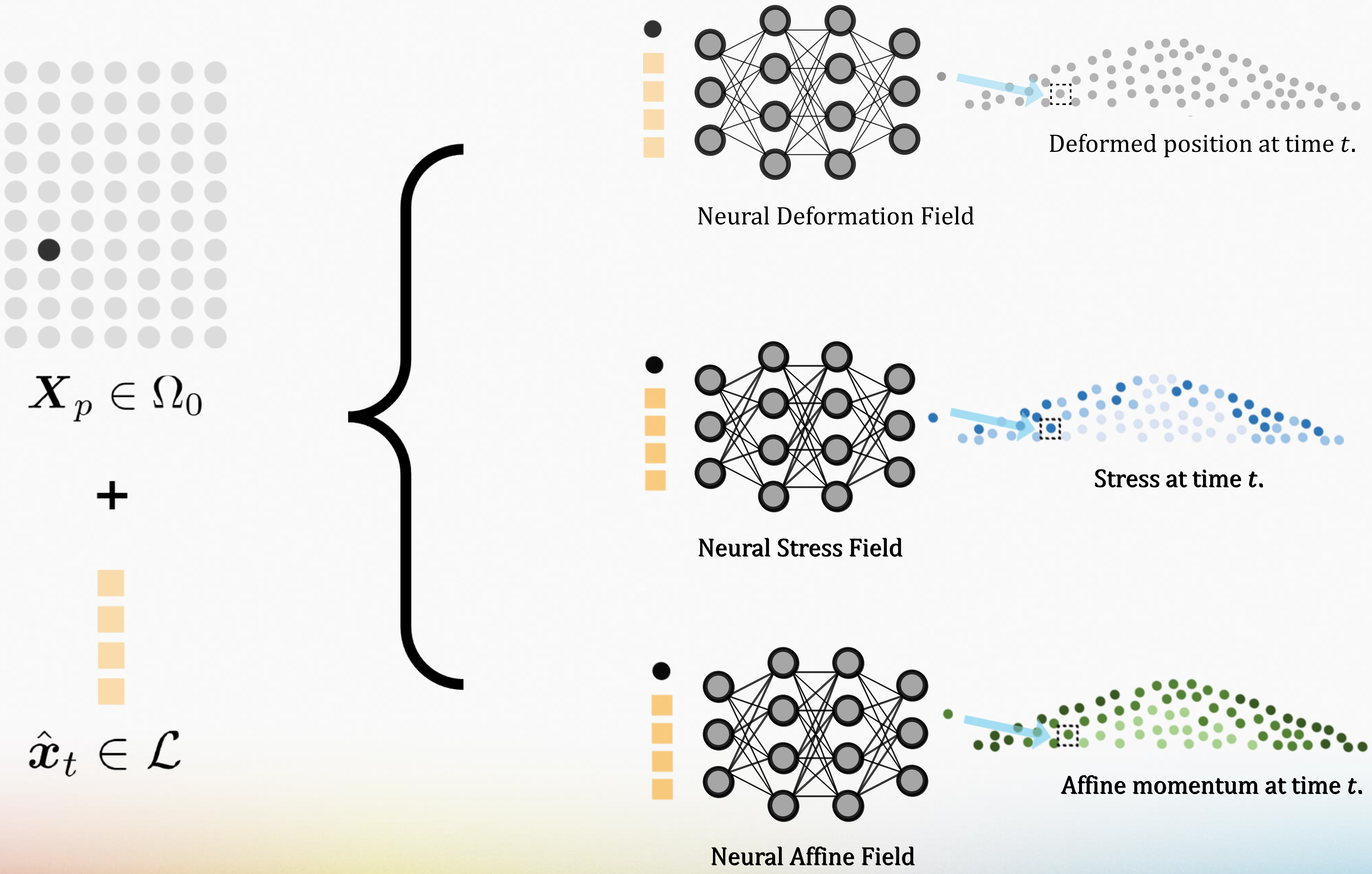
METHOD



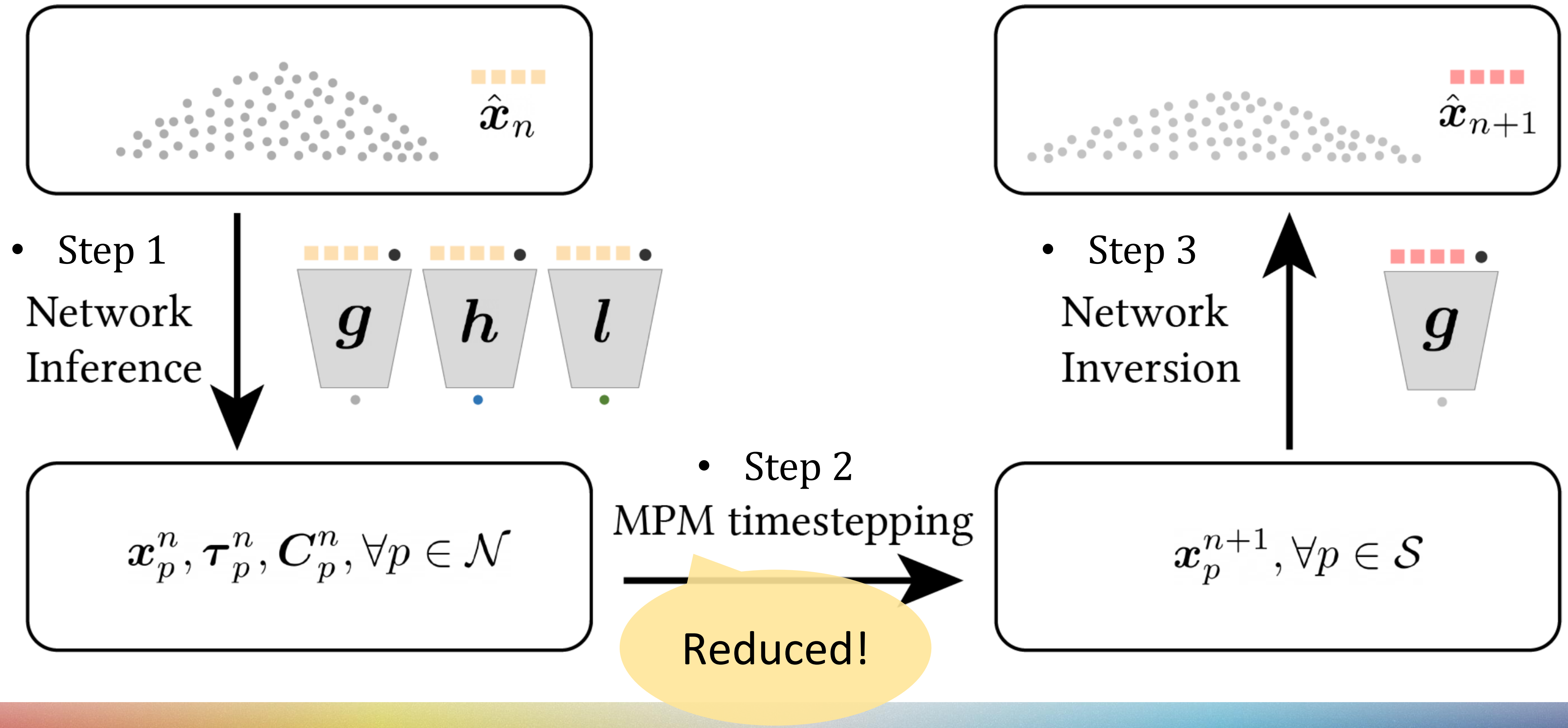
$\hat{\boldsymbol{x}}_t \in \mathcal{L}$

$$\boldsymbol{g}(\boldsymbol{X}, \hat{\boldsymbol{x}}_t) \approx \boldsymbol{\phi}(\boldsymbol{X}, t) = \boldsymbol{x}_t$$

METHOD


$\boldsymbol{X}_p \in \Omega_0$
+
$\hat{\boldsymbol{x}}_t \in \mathcal{L}$
Deformed position at time t.
Neural Deformation Field
Stress at time t.
Neural Stress Field
Affine momentum at time t.
Neural Affine Field

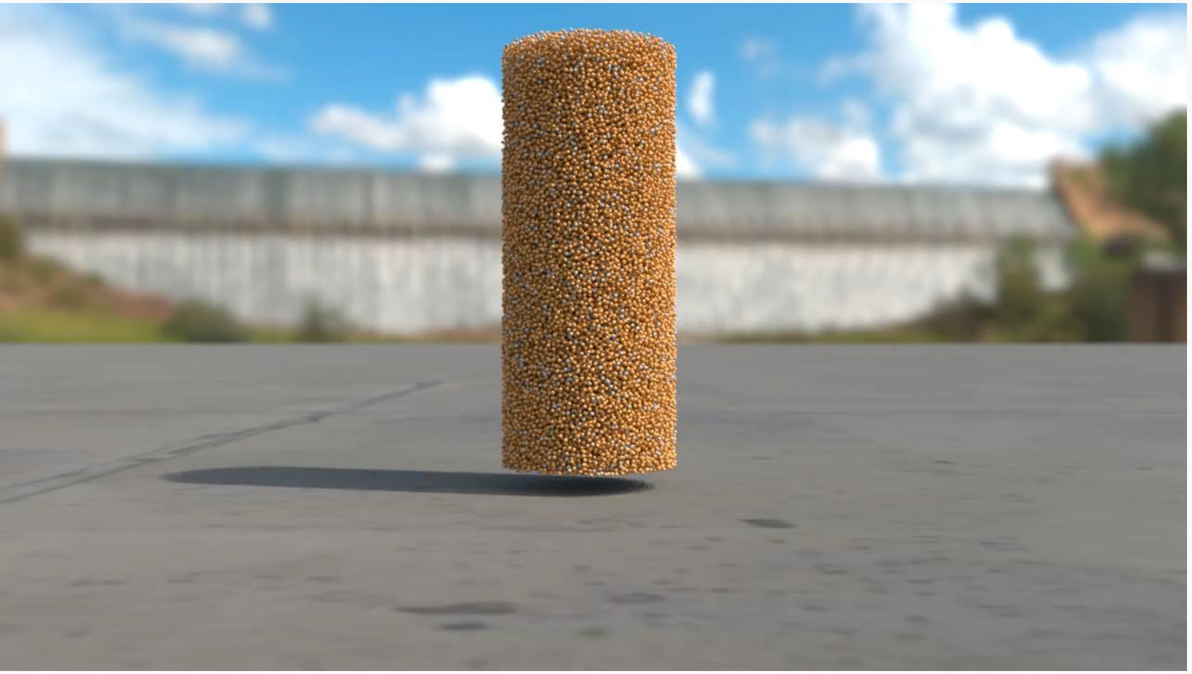
LATENT SPACE DYNAMICS: EVOLVE $\hat{x}_n$ TO $\hat{x}_{n+1}$ …



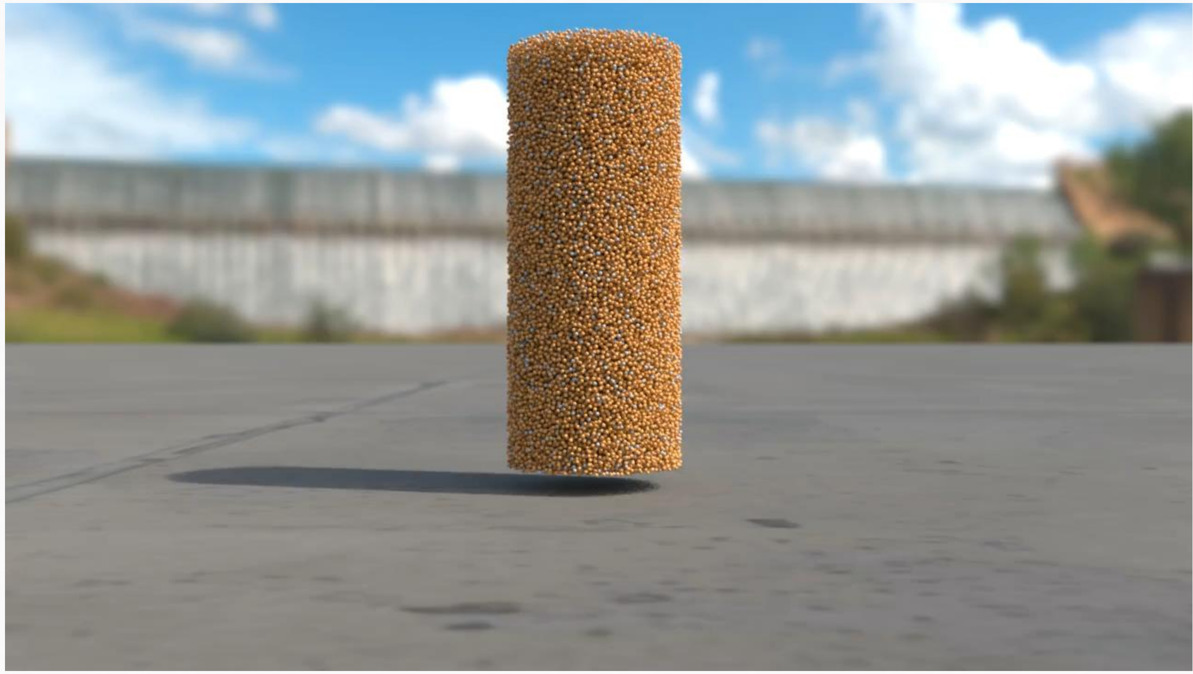
RESULTS

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

22°

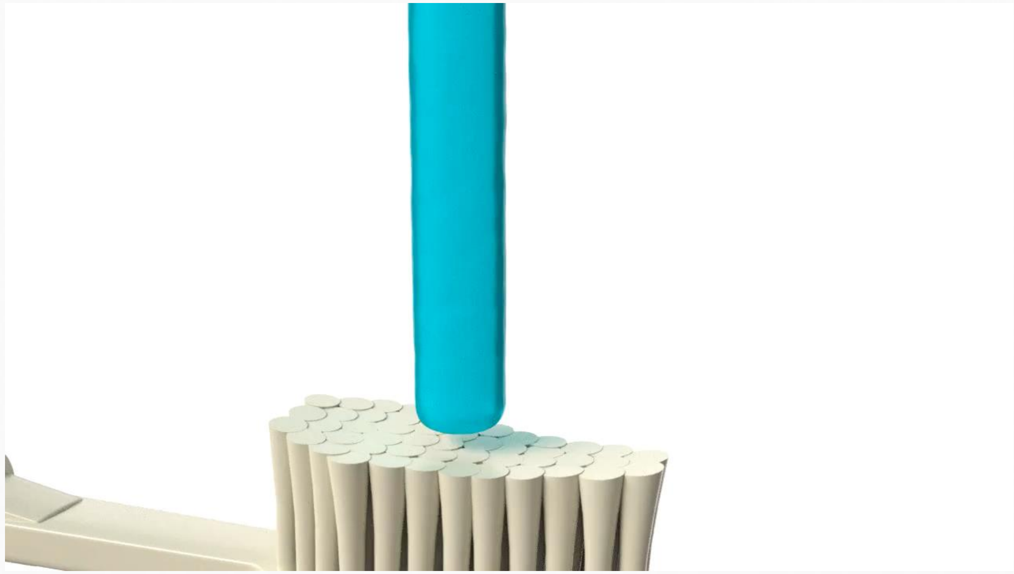


35°



Latent space dimension is 6.

RESULTS



Latent space dimension is 6.

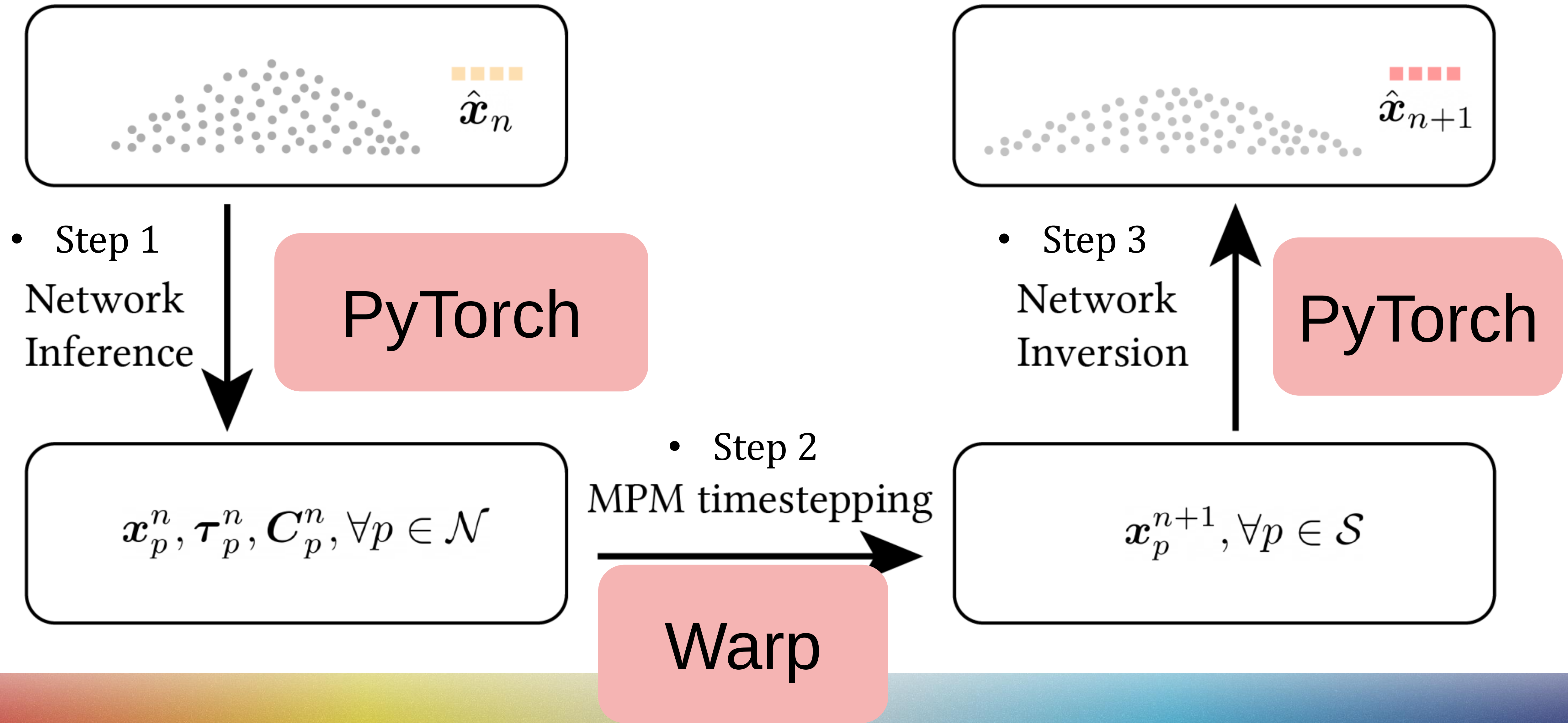
RESULTS

$E = 1.2$

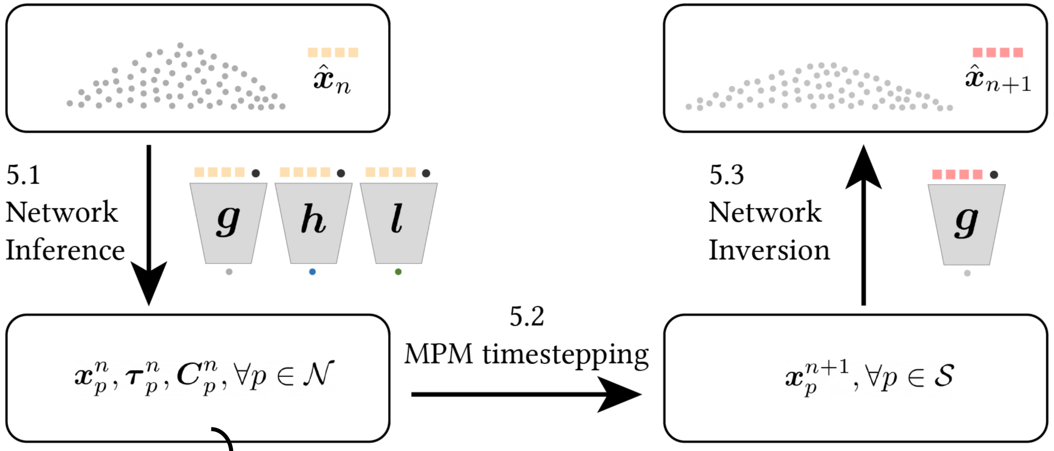
$E = 3.9$

$E = 9.0$

LATENT SPACE DYNAMICS: EVOLVE $\hat{x}_n$ TO $\hat{x}_{n+1}$ ...



SAMPLE CODE




```
for step in range(N):
  # In torch,
  # given xhat at t_n, query neural nets to get positions, etc.
  xhat_expanded = xhat_n.squeeze(0).expand(P, -1) # expand to size of X
  x_n = decoder_g.forward(torch.cat((X, xhat_all), 1))
  tau_n = decoder_h.forward(torch.cat((X, xhat_all), 1))
  C_n = decoder_l.forward(torch.cat((X, xhat_all), 1))

  # feed data from torch to warp
  mpm_state.particle_x = wp.from_torch(x_n)
  mpm_state.particle_stress = wp.from_torch(tau_n)
  mpm_state.particle_C = wp.from_torch(C_n)

  # In warp,
  ## MPM timestepping
  p2g2p(mpm_state, ...)

  # export data from Warp to torch
  x_np1 = wp.to_torch(mpm_state.particle_x)

  # In torch,
  # find xhat at t_{n+1}, using xhat_n as initial guess
  xhat_n = nonlinear_solver.solve(decoder_g, xhat_n, x_np1)
```

Network Inference (in torch)

Torch to Warp

MPM timestepping (in Warp)

Warp to torch

Network Inversion (in torch)

- See our paper https://arxiv.org/abs/2310.17790 for more details.

Using Warp in Machine Learning and Optimization Problems

Learning Neural Constitutive Laws from Motion Observations for Generalizable PDE Dynamics

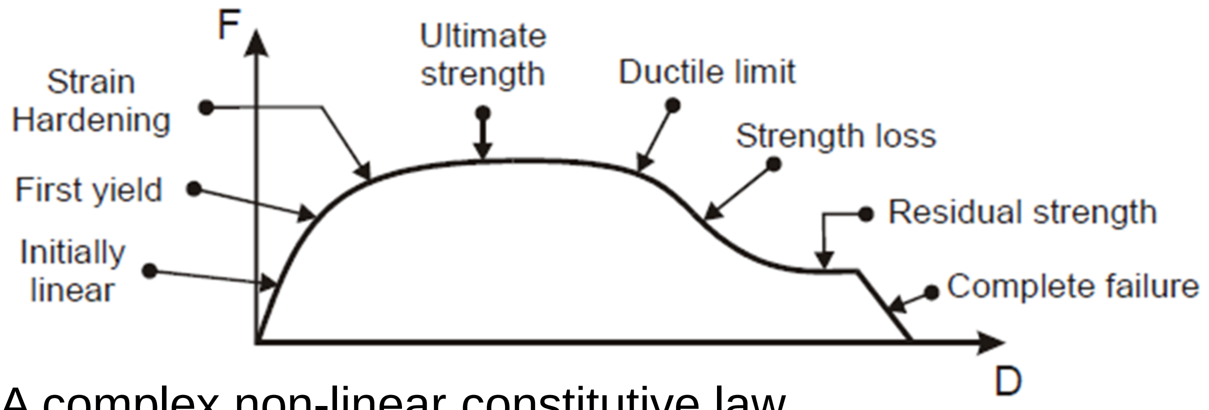
Pingchuan Ma, MIT CSAIL

CONSTITUTIVE LAWS

Q: What are **constitutive laws**?

Q: What are **constitutive laws**?

A: Constitutive law defines a material.
It usually is a **complex** function **hand-made** by **experts**.



A complex non-linear constitutive law.

https://wiki.csiamerica.com/display/kb/Material+nonlinearity

WHY?

Q: **Why do we care** constitutive laws?

WHY?

Q: **Why do we care** constitutive laws?

Constitutive Laws

A: Constitutive laws generate diverse physics.

Elastic Material

Granular Material

Plastic Material

Fluid

HOW?

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

Q: **How to generate** constitutive laws?

HOW?

Q: **How to generate** constitutive laws?

Constitutive Laws

A: Learn it with PyTorch! 🤩

Elastic Material

Granular Material

Plastic Material

Fluid

YES!

PyTorch `nn.Module`

forward

PyTorch

✓ **User-friendly Operations**
✓ **GPU Support**

backward

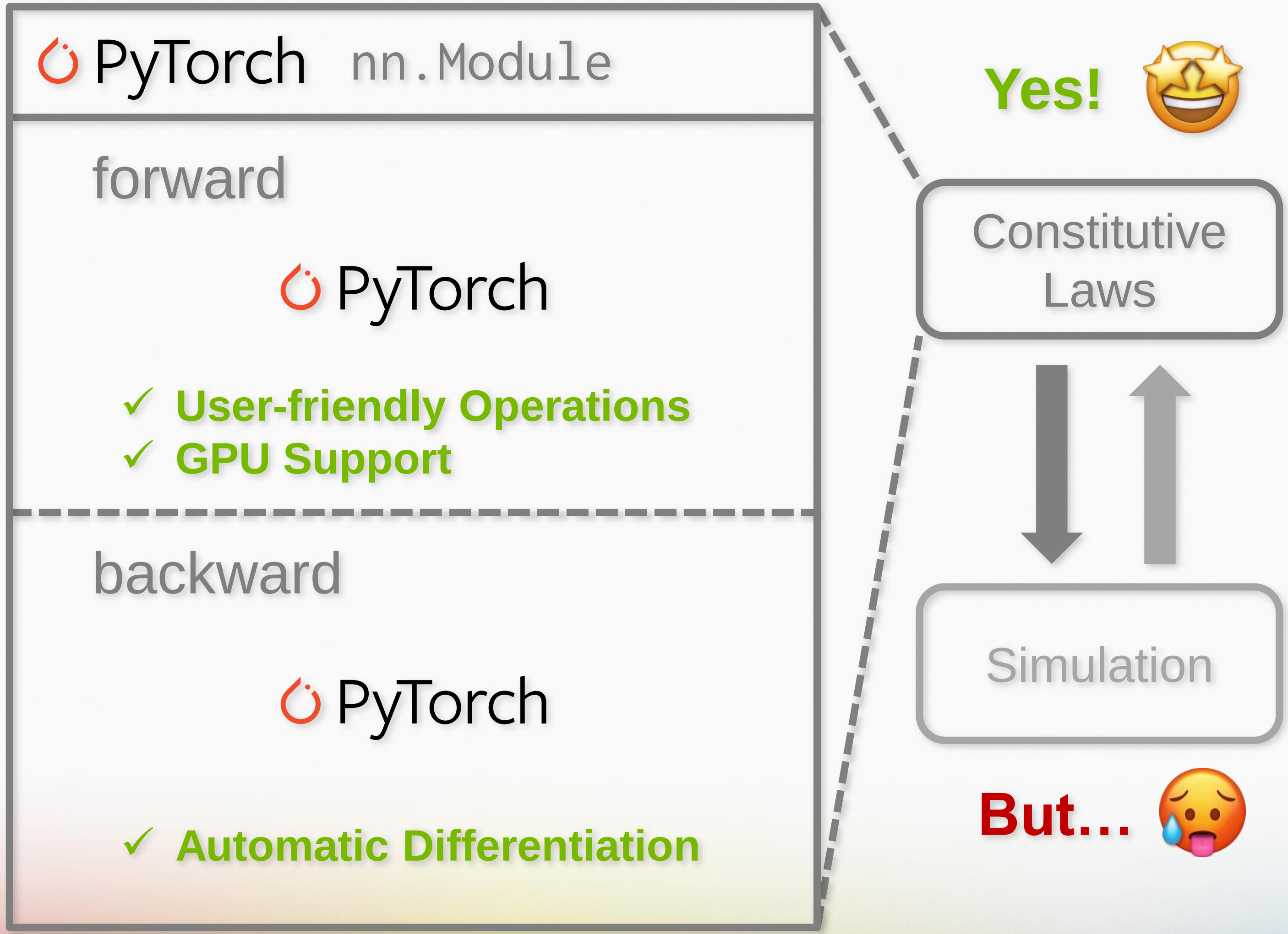
PyTorch

✓ **Automatic Differentiation**

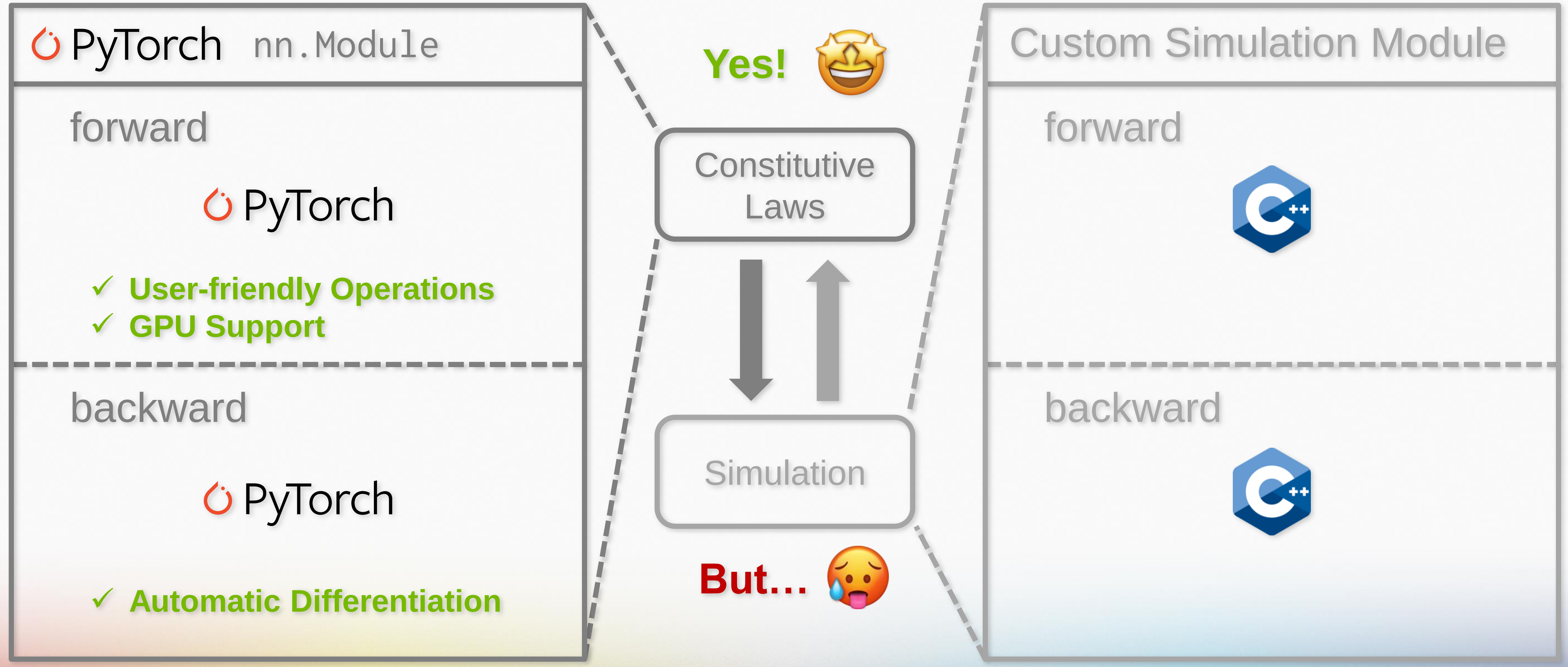
Yes! 🤩

Constitutive Laws

BUT…


PyTorch nn.Module
forward
PyTorch
✓ User-friendly Operations
✓ GPU Support
backward
PyTorch
✓ Automatic Differentiation
Yes!
Constitutive Laws
Simulation
But…

BUT…


PyTorch nn.Module
forward
PyTorch
✓ User-friendly Operations
✓ GPU Support
backward
PyTorch
✓ Automatic Differentiation
Yes!
Constitutive Laws
Simulation
But…
Custom Simulation Module
forward
backward

BUT...

PyTorch `nn.Module`

forward

PyTorch

- ✓ User-friendly Operations
- ✓ GPU Support

backward

PyTorch

- ✓ Automatic Differentiation

Yes! 🤩

Constitutive Laws

Simulation

But... 🥵

Custom Simulation Module

forward

- × Heavy Coding
- × Few GPU Support

backward

- × Incompatible Computation Graph
- × Manual Differentiation

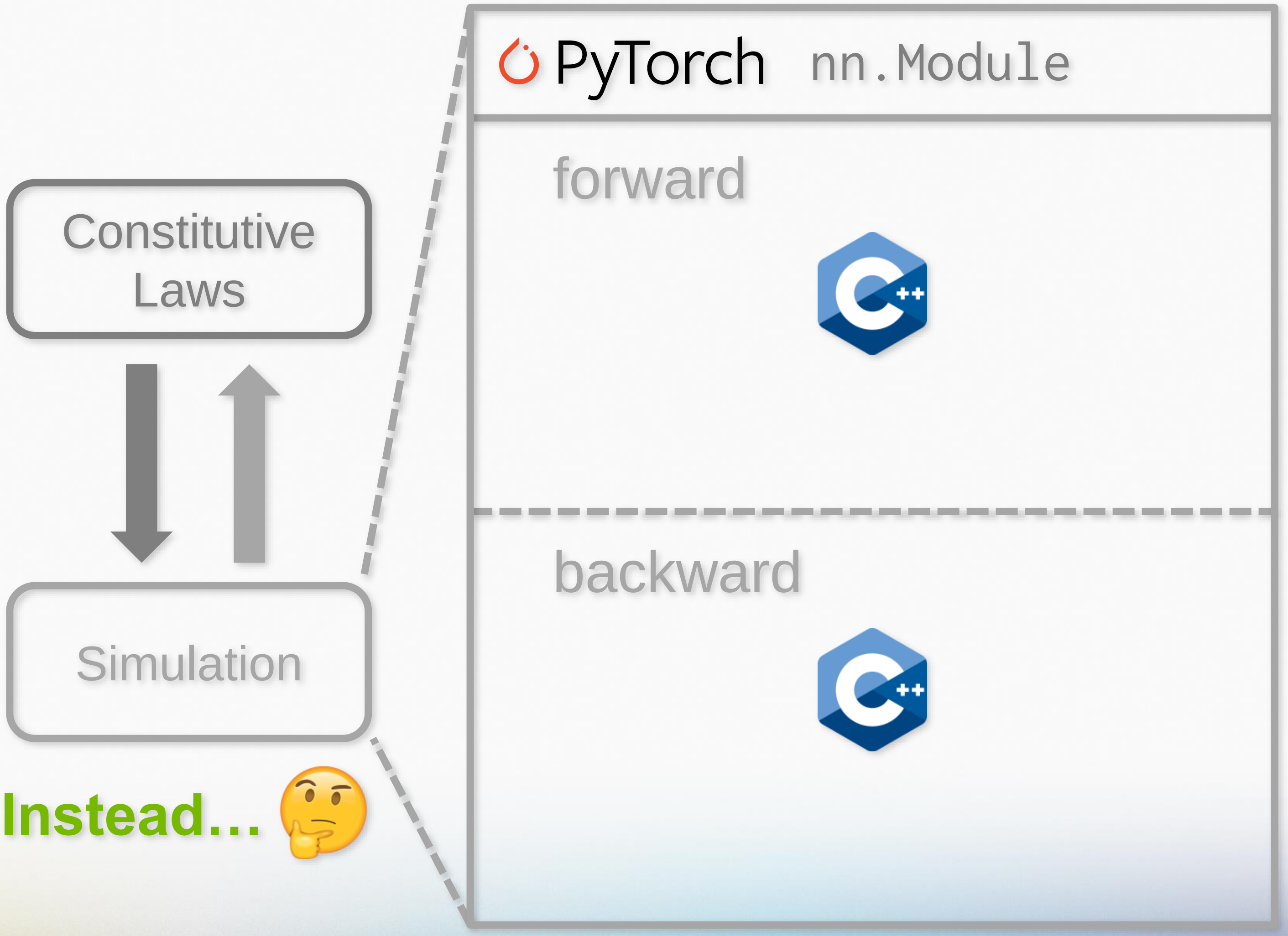
HOW?

Q: <u>**How to build**</u> a learning pipeline that is

1. Differentiable
2. GPU-empowered
3. PyTorch-interoperable
4. Easy to write (preferably in Python)

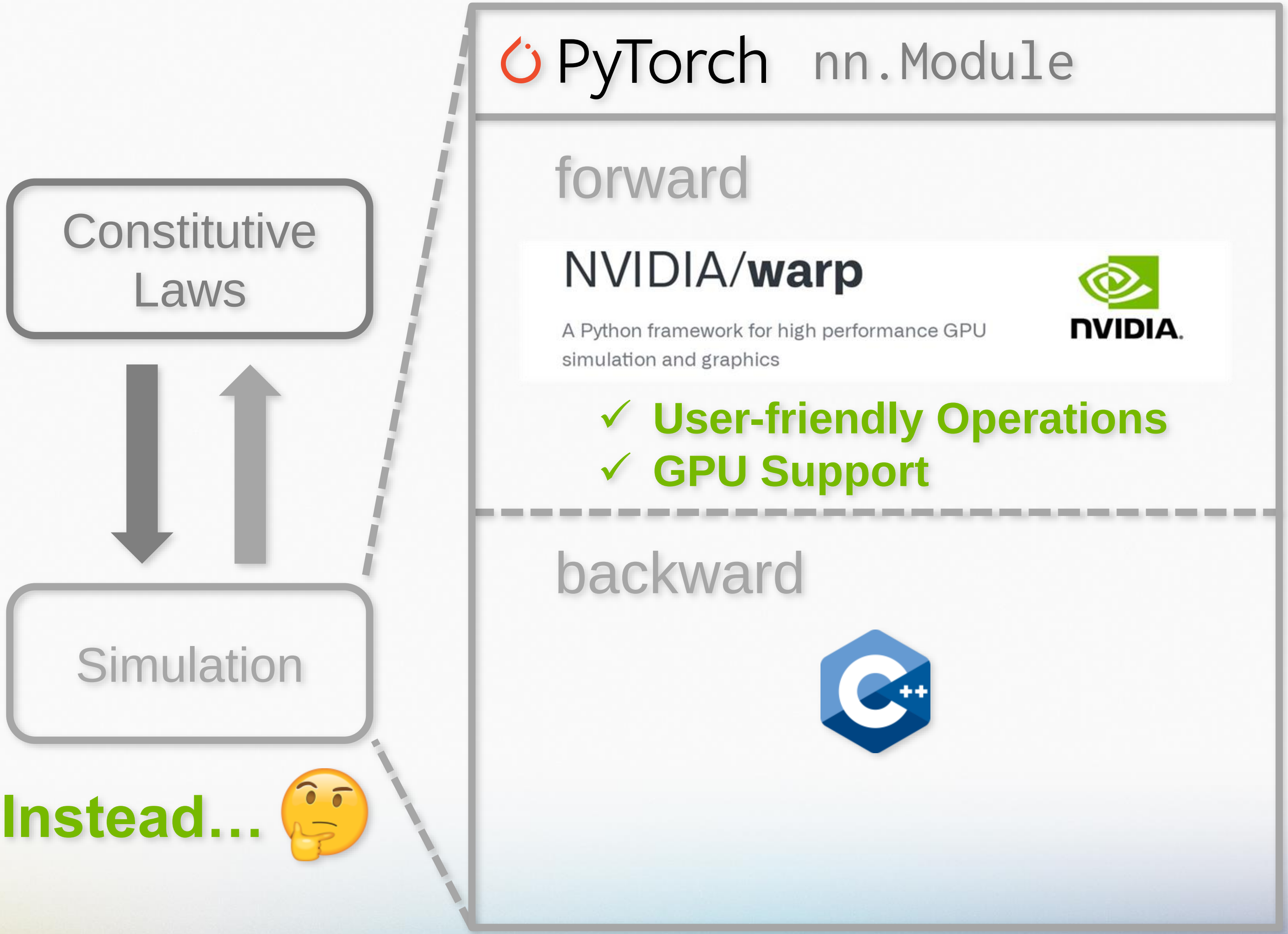
INSTEAD…

1. Integrate the computation graph into `nn.Module`



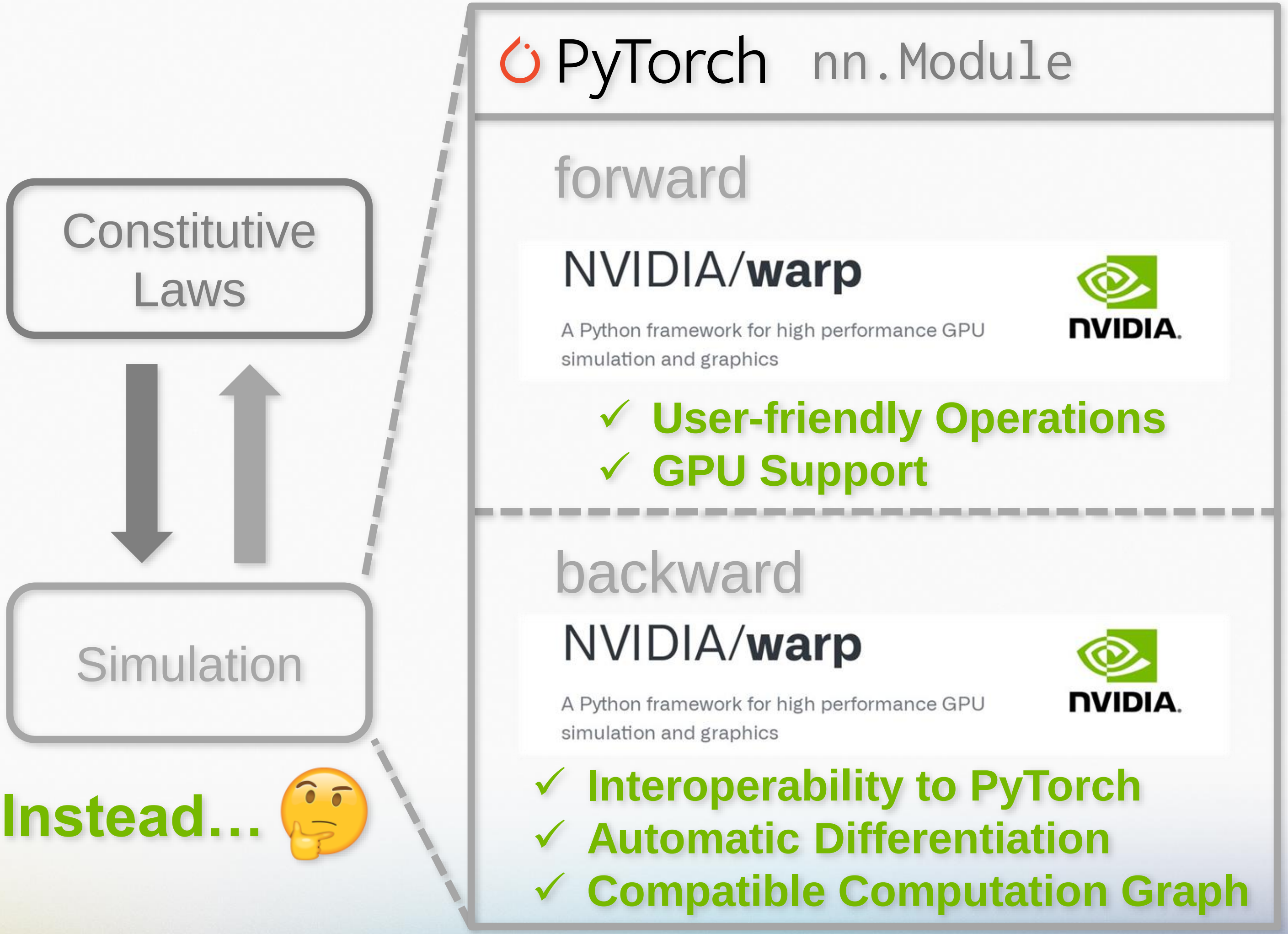
INSTEAD…

1. Integrate the computation graph into `nn.Module`
2. Rewrite the simulation with NVIDIA Warp, which uses regular Python syntax but JIT compiles them to both CPU and GPU.



INSTEAD…

1. Integrate the computation graph into `nn.Module`
2. Rewrite the simulation with NVIDIA Warp, which uses regular Python syntax but JIT compiles them to both CPU and GPU.
3. Let NVIDIA Warp auto-diff your forward simulation.



END-TO-END DIFFERENTIABLE GPU PIPELINE IN PYTHON

PyTorch nn.Module

forward

PyTorch

✓ User-friendly Operations
✓ GPU Support

backward

PyTorch

✓ Automatic Differentiation

Constitutive Laws

Simulation

PyTorch nn.Module

forward

NVIDIA/**warp**

A Python framework for high performance GPU simulation and graphics

NVIDIA

✓ User-friendly Operations
✓ GPU Support

backward

NVIDIA/**warp**

A Python framework for high performance GPU simulation and graphics

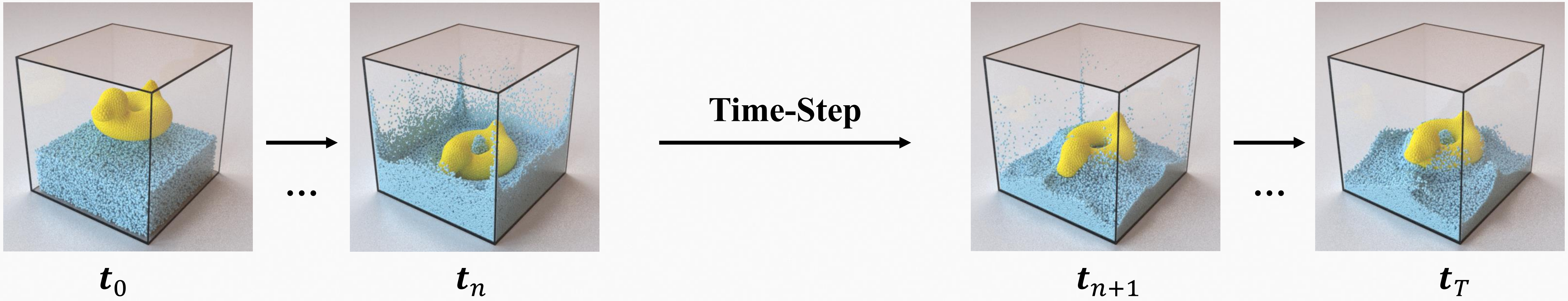
NVIDIA

✓ Interoperability to PyTorch
✓ Automatic Differentiation
✓ Compatible Computation Graph

Learning **Neural Constitutive Laws** from Motion Observations for Generalizable PDE Dynamics

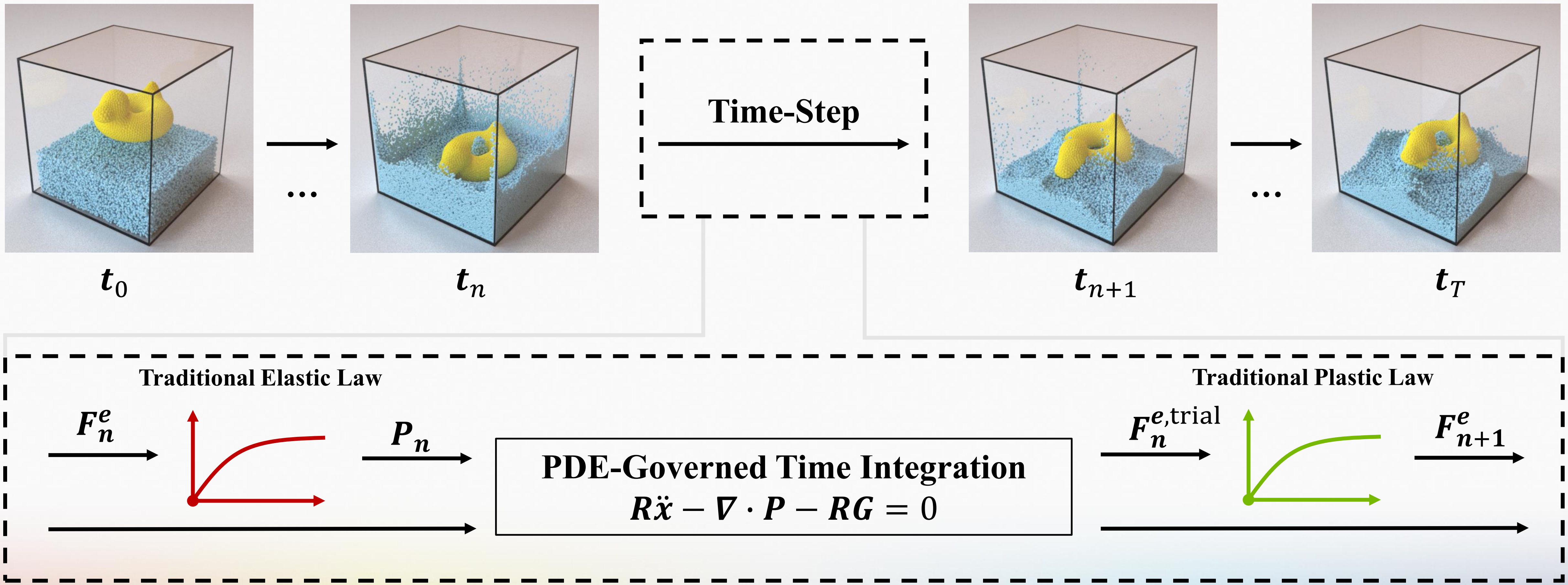
ICML 2023

TECHNICAL OVERVIEW



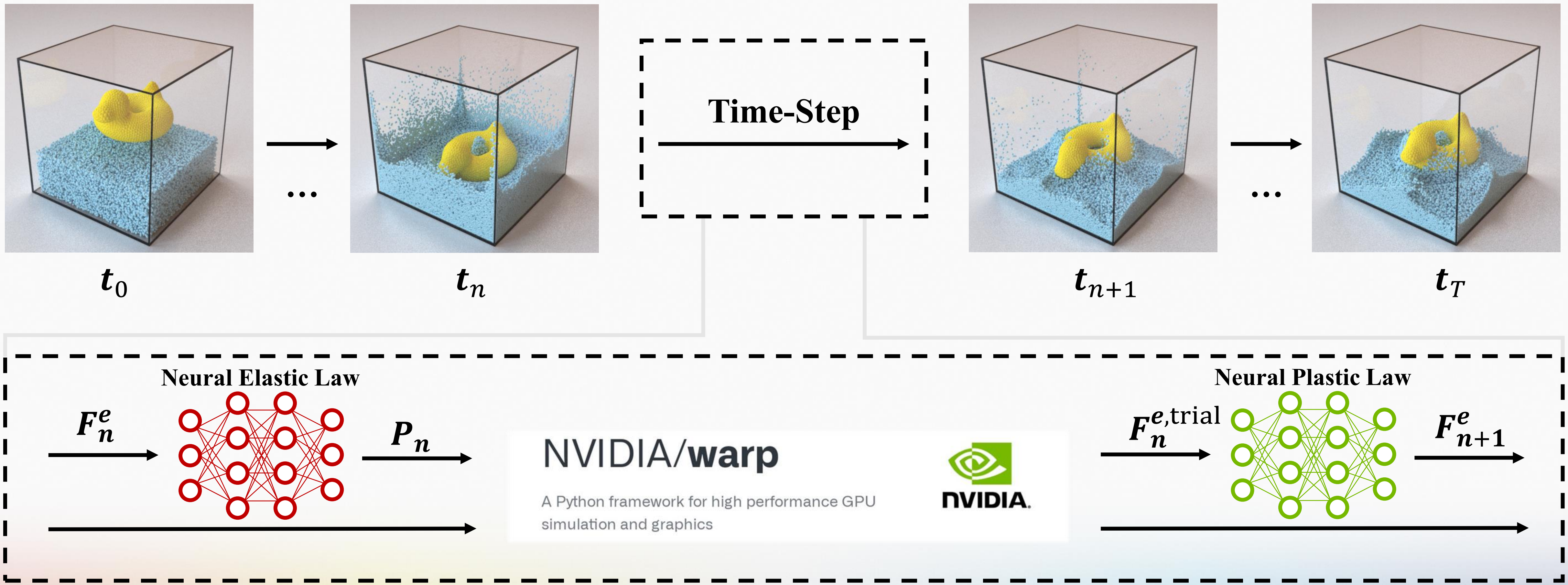
APPLICATIONS: NEURAL CONSTITUTIVE LAWS

TECHNICAL OVERVIEW



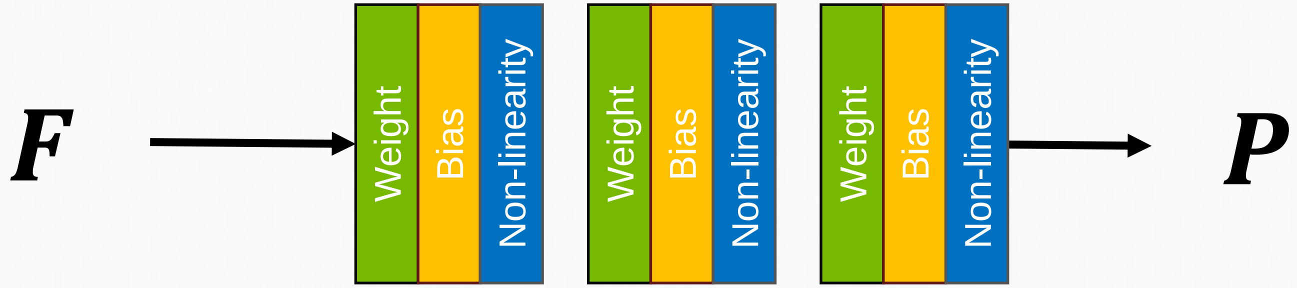
APPLICATIONS: NEURAL CONSTITUTIVE LAWS

TECHNICAL OVERVIEW

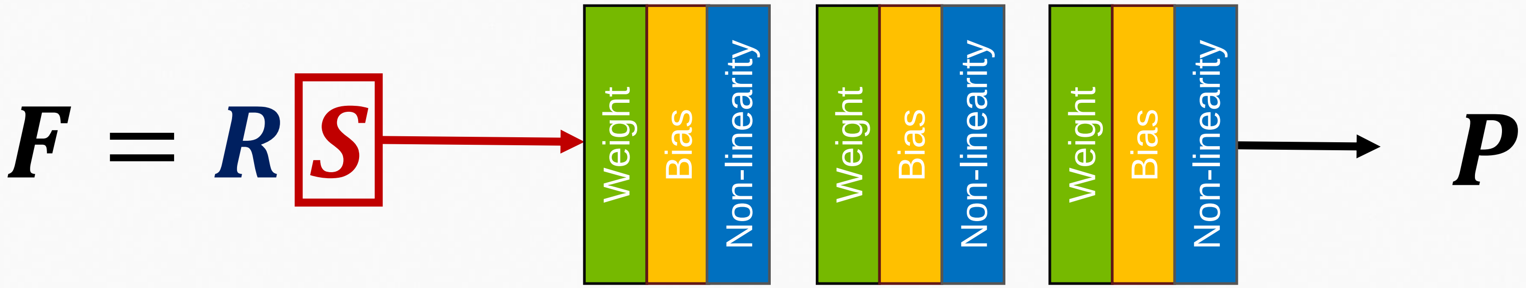


PHYSICS-AWARE NETWORK ARCHITECTURE

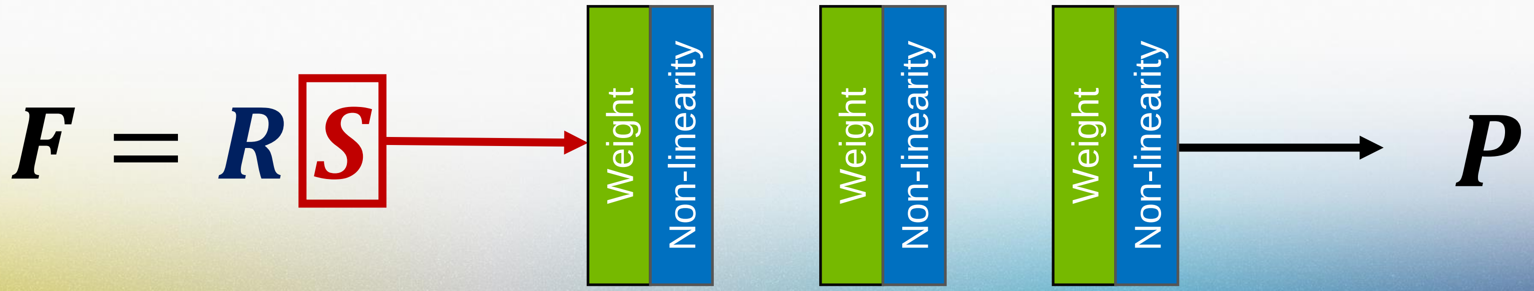
Original Network Architecture



(1) Rotation Equivariance



(2) Undeformed State Equilibrium

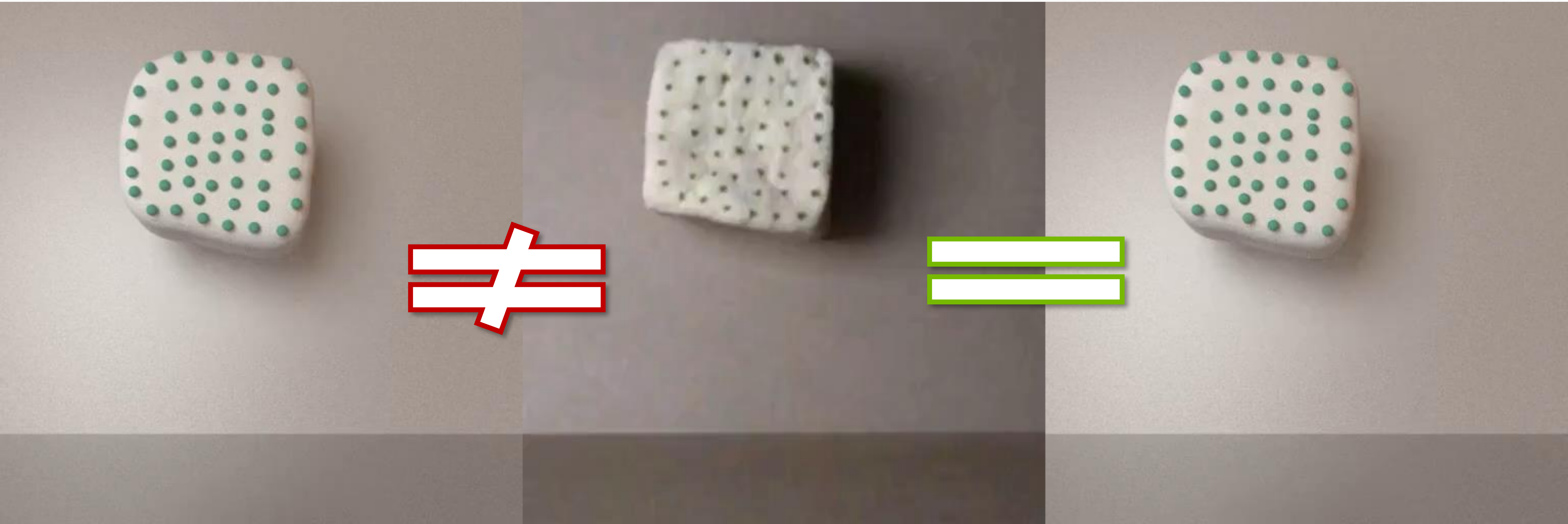


REAL-WORLD: RECONSTRUCTION

Traditional Sys-ID Reconstruction

Real-World Video Clip

NCLaw Reconstruction

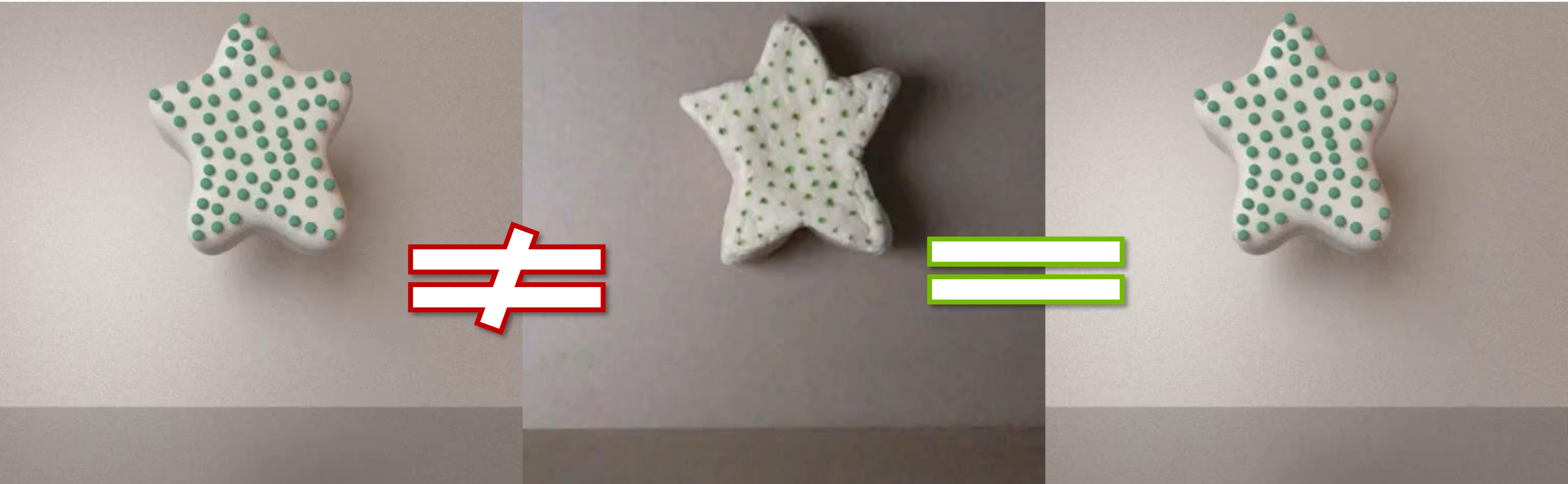


REAL-WORLD: GENERALIZATION

Traditional Sys-ID Evaluation

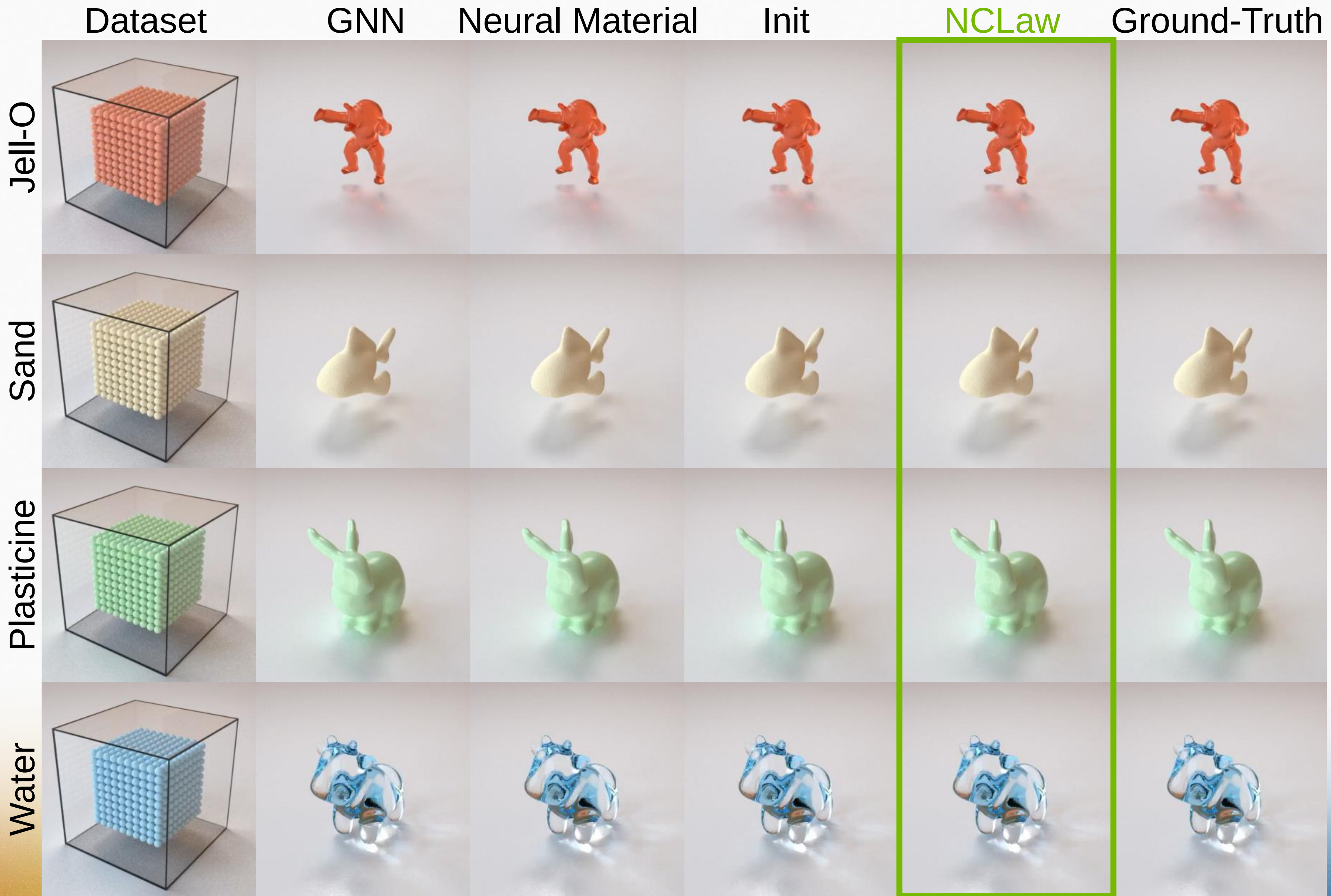
Real-World Video Clip

NCLaw Evaluation



Note: All results are single-shot learning

APPLICATIONS: NEURAL CONSTITUTIVE LAWS



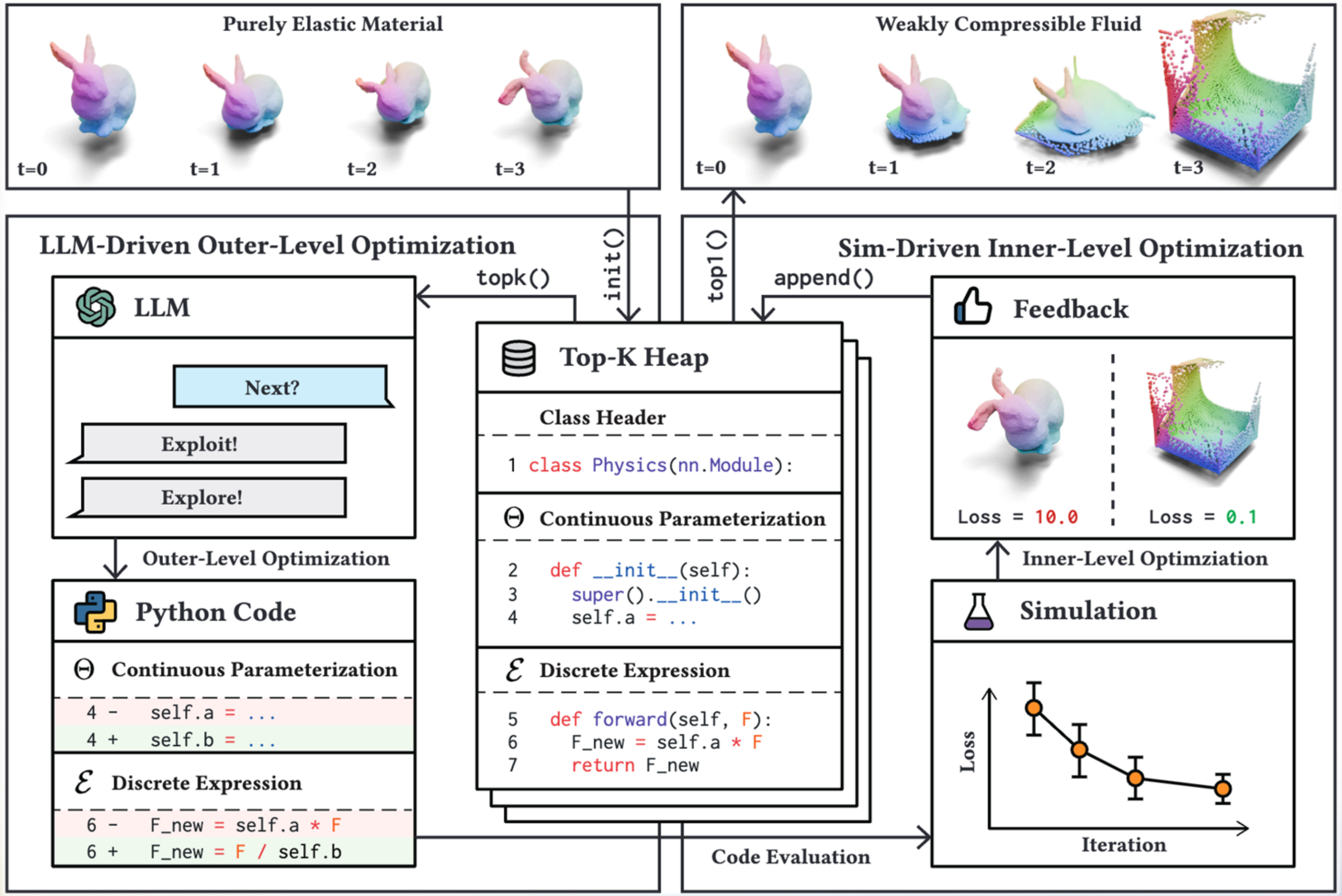
LLM and Simulation as Bilevel Optimizers:

A New Paradigm to Advance Physical Scientific Discovery

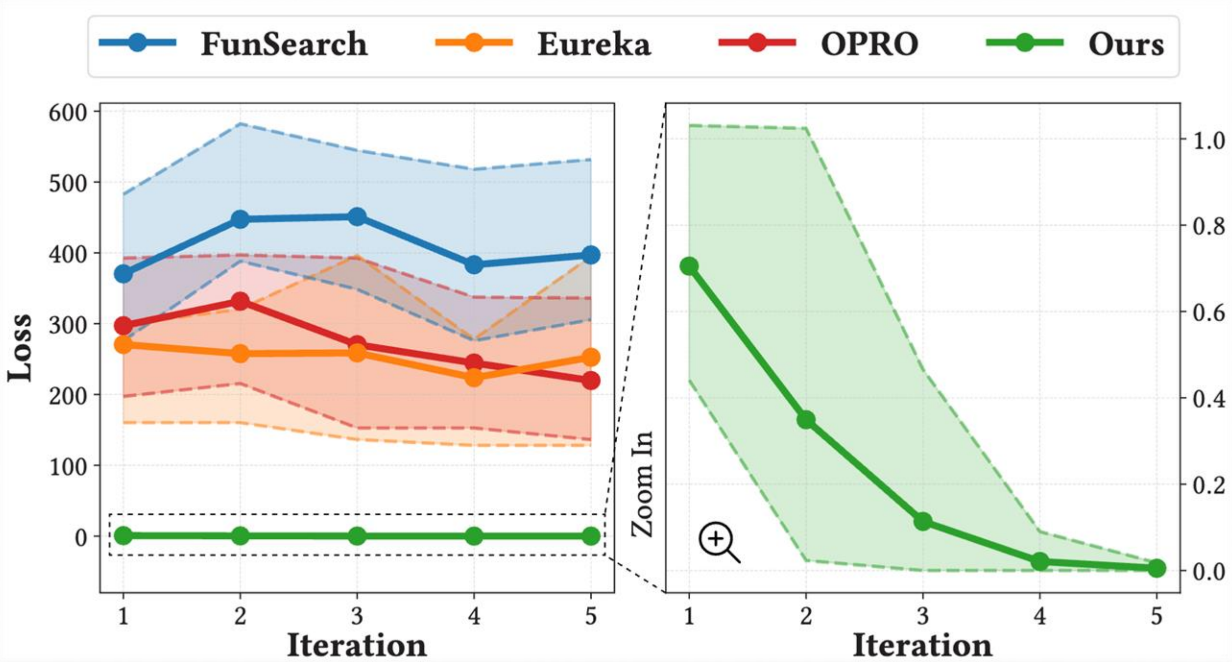
ICML 2024

Key idea:

Similar to Neural Constitutive Laws but (1) replace the neural networks with purely symbolic PyTorch code to keep interpretability and (2) optimize it with LLMs and NVIDIA Warp-driven differentiable simulation.



APPLICATIONS: SCIENTIFIC GENERATIVE AGENT



With the help of NVIDIA Warp-driven simulation, our method optimizes much faster and better than the baselines that purely optimizes with LLMs.

TAKEAWAYS

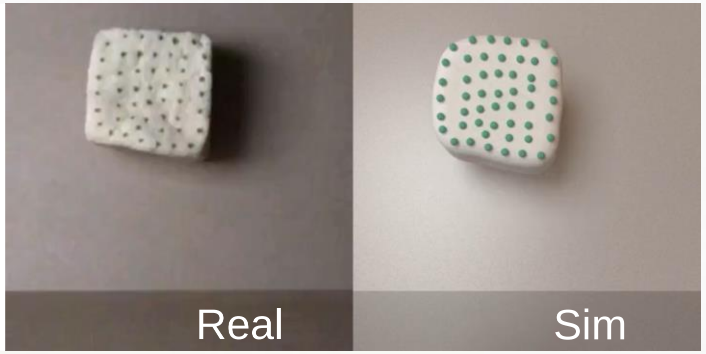
PyTorch | Constitutive Laws ⇄ Simulation | NVIDIA/warp

A Python framework for high performance GPU simulation and graphics

NVIDIA

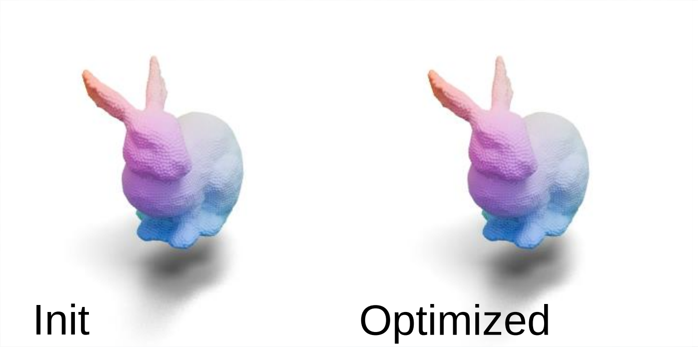
We use NVIDIA Warp to build end-to-end differentiable PyTorch-compatible GPU learning pipeline in Python.

Neural Constitutive Laws



Neural Networks + NVIDIA Warp

Scientific Generative Agent



LLMs + NVIDIA Warp

Using Warp in Machine Learning and Optimization Problems

PhysGaussian: Physics-Integrated 3D Gaussians for Generative Dynamics

Zeshun Zong, UCLA

- Traditional simulation pipeline:

Object → Geometry → Simulation → Rendering → Result

Can we?

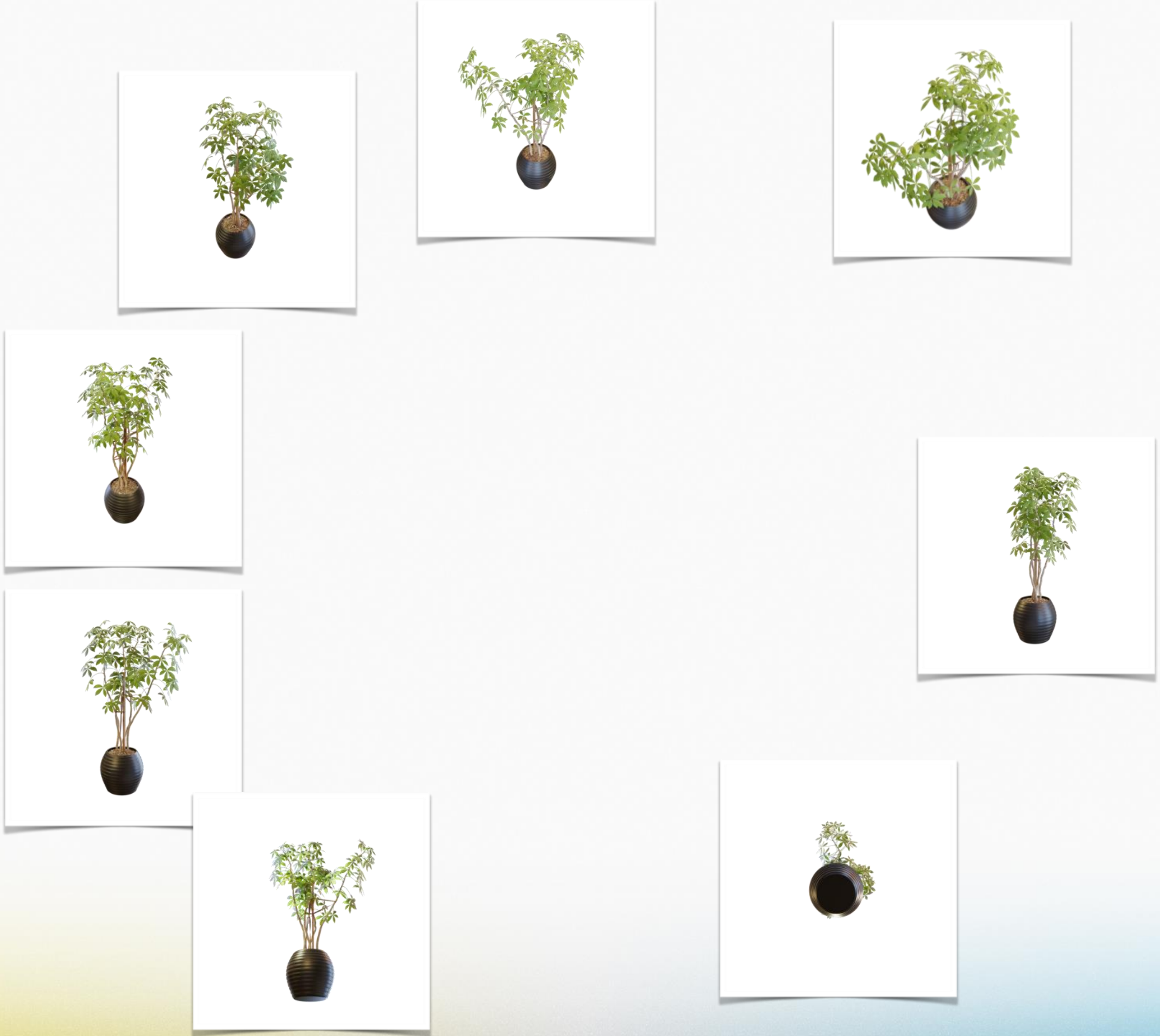
PHYSGAUSSIAN

Multi-view Images

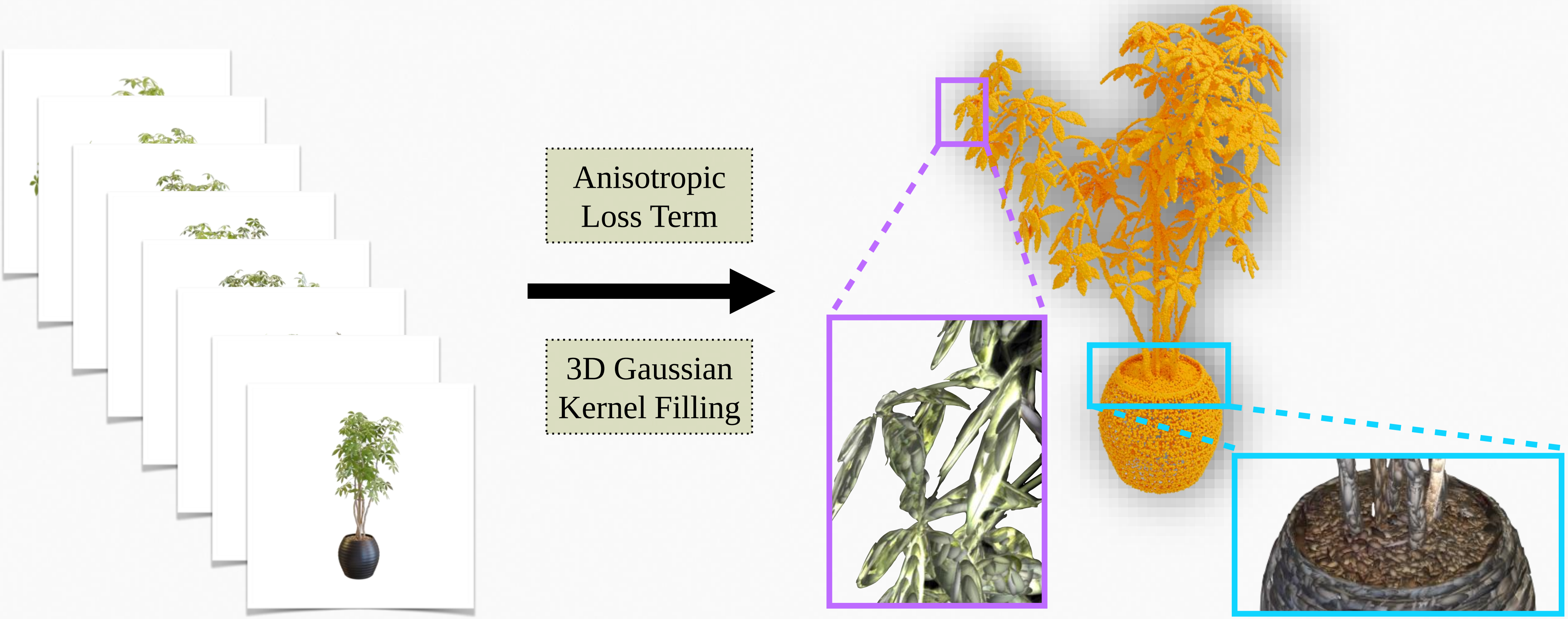
Physics-based Dynamics

PHYSGAUSSIAN

PHYSGAUSSIAN



PHYSGAUSSIAN



Kinematics

- Gaussian Evolution
- Harmonics Transform

Dynamics

- Continuum Mechanics
- Time Integration

Handled by MPM!

Kinematics

- Gaussian Evolution
- Harmonics Transform

Dynamics

- Continuum Mechanics
- Time Integration

Physics-grounded
Novel Motion

SAMPLE CODE


TEAR TO SQUEEZE
DIP & SQUEEZE
HEINZ TOMATO KETCHUP
TEAR TO SQUEEZE
DIP & SQUEEZE
HEINZ TOMATO KETCHUP

PHYSGAUSSIAN

SAMPLE CODE


```
## first train tailored Gaussian Splatting

# load the GS model
gaussians = load_checkpoint(model_path)

# create a MPM solver
mpm_solver = MPM_Simulator_WARP()
# load the Gaussian point cloud for mpm simulation
mpm_solver.load_initial_data(gaussians, ...)

for frame in range(frame_num):
  # particles move, following MPM timestepping
  mpm_solver.p2g2p(...)

  # the new positions of center of Gaussian kernels
  pos = wp.to_torch(mpm_solver.mpm_state.particle_x)
  # covariance and rotation of kernels
  cov3D = wp.to_torch(mpm_solver.mpm_state.particle_cov)
  rot = wp.to_torch(mpm_solver.mpm_state.particle_rot)

  # render images using the updated gaussian kernels
  rasterize(pos, cov3D, rot, ...)
```

Train and load Gaussian Splatting (in torch)

Setup simulator and pass data to Warp

MPM timestepping (in Warp)

Warp to torch

Gaussian Splatting rendering (in torch)

- See our paper https://arxiv.org/abs/2311.12198 for more details.

Using Warp in Machine Learning and Optimization Problems

Atlas3D: Physically Constrained Self-Supporting Text-to-3D for Simulation and Fabrication

Zeshun Zong, UCLA

RECALL WARP'S AUTOMATIC DIFFERENTIATION

Simulator in Warp: Sim

Initial object: X

Boundary conditions: g → Final pose $x = Sim(X, g, \mu)$

Simulation parameters: μ

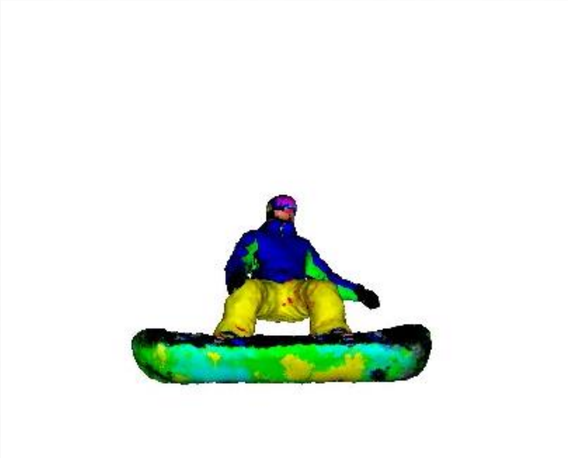
You can differentiate x with respect to $X, g,$ or μ!

Wrap all of those in Torch and blend with other losses

ATLAS3D: LET MODELS BE STANDABLE!

"a snowboarding man" → **Text-3D Generative Model** → 

- Optimize Score Distillation Sampling (SDS) loss *"a snowboarding man"* → **Text-3D Generative Model** →



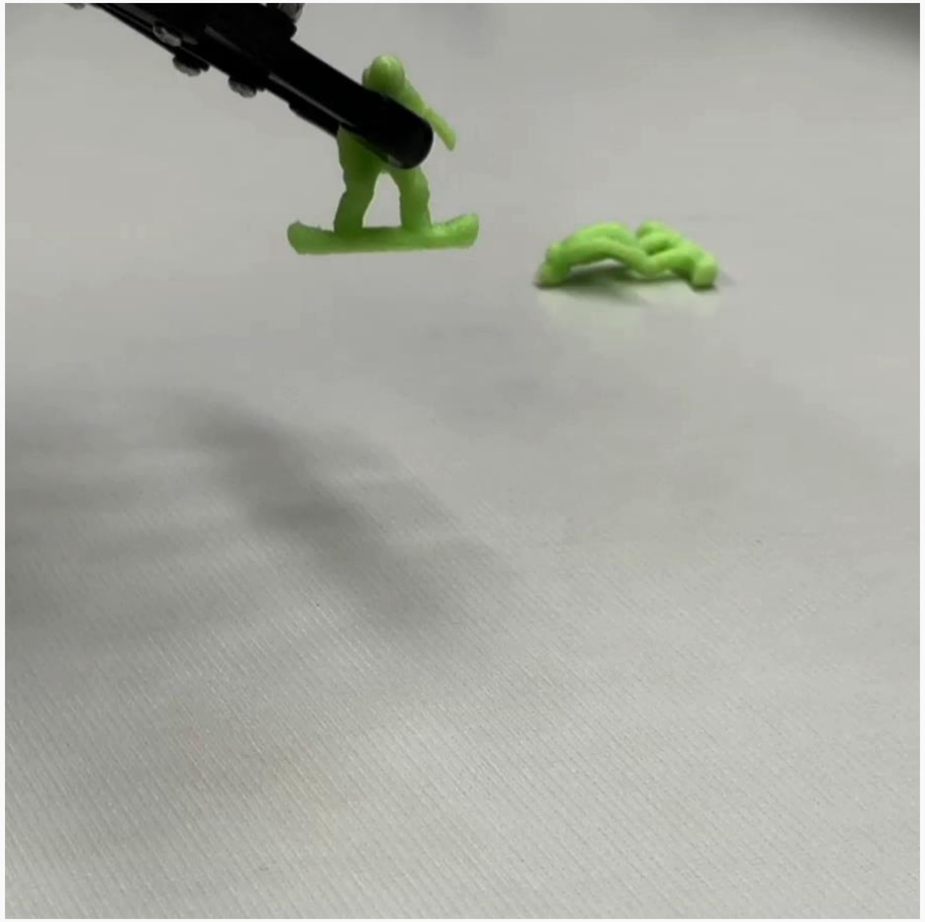
+

Warp simulator wrapped in Torch →

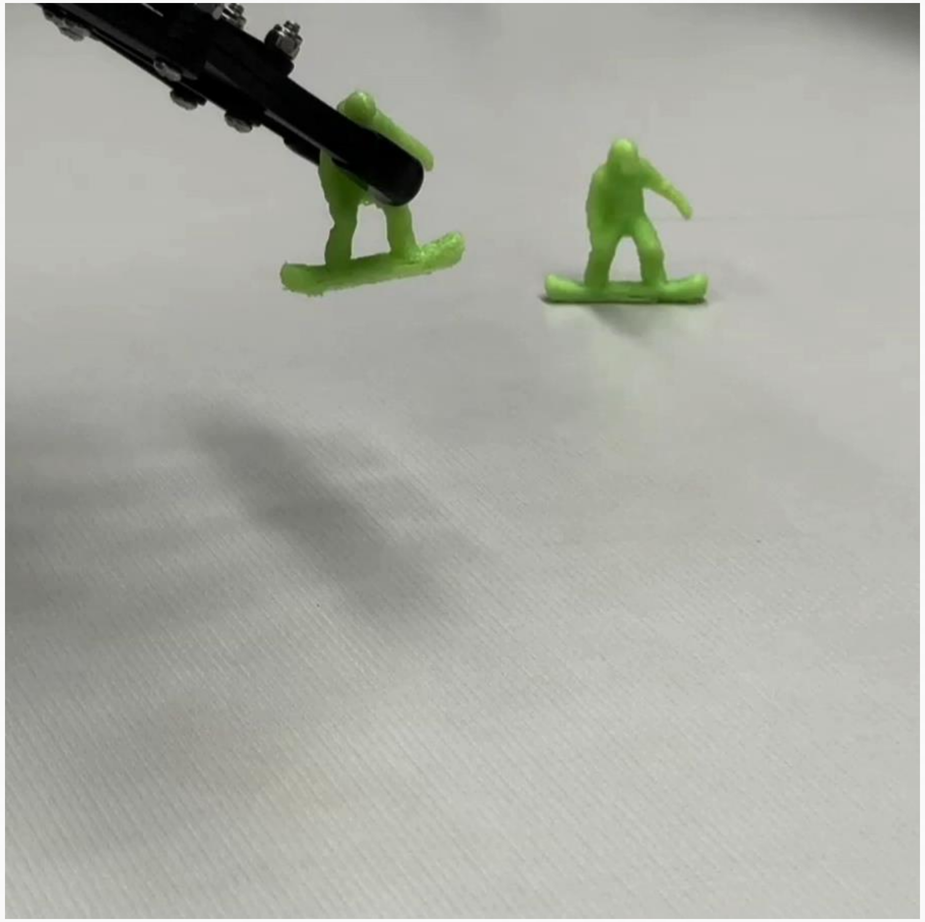
Penalize the rotation of final simulated pose x

w/o Atlas3D

w/ Atlas3D, not a static image!



w/ Atlas3D



w/o Atlas3D

- See our paper https://arxiv.org/abs/2405.18515v1 for more details.

Conclusion and Q&A

Miles Macklin, NVIDIA

Road Map

- Integration with nvmath Python libraries
- Device-side versions of CUDA libraries:
  - **cuBLASDx**
  - **cuFFTDx**
- Example: Invoke FFT directly from inside Warp kernels
- Fully fused execution with user code

```python
import nvmath as nvm
import warp as wp

FFT_fwd = nvm.device.fft(type='c2c', size=size, precision=wp.float32, direction='forward', dsl='warp')
FFT_inv = nvm.device.fft(type='c2c', size=size, precision=wp.float32, direction='inverse', dsl='warp')

@wp.kernel
def compute(input: wp.array(dtype=float),
            output: wp.array(dtype=float),
            fwd: FFT_fwd,
            inv: FFT_inv):

    i = wp.tid()

    # epilogue
    thread_data = user_func(input[i])

    # execute foward FFT
    fwd(FFT_fwd, thread_data)

    # frequency space convolution
    thread_data /= scale

    # execute inverse FFT
    inv(FFT_inv, thread_data)

    # prologue
    output[i] = thread_data
```

Example: Fused Forward/Reverse FFT

Road Map

Warp Neural Networks

- Leverage Tensor-Core operations (MMA) inside Warp kernels, e.g.:
  - wp.linear()
  - wp.map()
- Ideal for building fused neural network inference in Warp kernels
- Keep all intermediate values local to kernel
- Network inference in-situ

```python
@wp.kernel
def eval(layer0: Layer,
         layer1: Layer,
         loss: wp.array(dtype=float)):

    # setup feature vectors / input
    x = wp.vector(dtype=wp.float16, length=DIM_IN)
    x[0] = 0.0
    x[1] = 1.0
    x[2] = 2.0
    x[3] = 3.0

    # network eval
    z = wp.linear(layer0.weights, layer0.bias, DIM_HID, z)
    z = wp.map(relu, z)
    z = wp.linear(layer1.weights, layer1.bias, DIM_OUT, z)
    z = wp.map(relu, z)

    # compute loss
    l = wp.length(z)


    # write loss
    wp.atomic_add(loss, 0, l)
```

Example: Fused MLP Inference

Road Map


```python
@wp.kernel
def eval_tetrahedra(args):

    tid = wp.tid()

    ...

    act = activation[tid]

    k_mu = materials[tid, 0]
    k_lambda = materials[tid, 1]
    k_damp = materials[tid, 2]

    Ds = wp.mat33(x10, x20, x30)
    Dm = pose[tid]

    inv_rest_volume = wp.determinant(Dm) * 6.0
    rest_volume = 1.0 / inv_rest_volume

    alpha = 1.0 + k_mu / k_lambda - k_mu / (4.0 * k_lambda)

    # scale stiffness coefficients to account for area
    k_mu = k_mu * rest_volume
    k_lambda = k_lambda * rest_volume
    k_damp = k_damp * rest_volume

    # F = Xs*Xm^-1
    F = Ds * Dm
    dFdt = wp.mat33(v10, v20, v30) * Dm

    col1 = wp.vec3(F[0, 0], F[1, 0], F[2, 0])
    col2 = wp.vec3(F[0, 1], F[1, 1], F[2, 1])
    col3 = wp.vec3(F[0, 2], F[1, 2], F[2, 2])

    # -----------------------------
    # Neo-Hookean (with rest stability [Smith et al 2018])

    Ic = dot(col1, col1) + dot(col2, col2) + dot(col3, col3)

    # deviatoric part
    P = F * k_mu * (1.0 - 1.0 / (Ic + 1.0)) + dFdt * k_damp
    H = P * wp.transpose(Dm)

    f1 = wp.vec3(H[0, 0], H[1, 0], H[2, 0])
    f2 = wp.vec3(H[0, 1], H[1, 1], H[2, 1])
    f3 = wp.vec3(H[0, 2], H[1, 2], H[2, 2])

    J = wp.determinant(F)

    s = inv_rest_volume / 6.0
    dJdx1 = wp.cross(x20, x30) * s
    dJdx2 = wp.cross(x30, x10) * s
    dJdx3 = wp.cross(x10, x20) * s

    f_volume = (J - alpha + act) * k_lambda
    f_damp = (wp.dot(dJdx1, v1) + wp.dot(dJdx2, v2) + wp.dot(dJdx3, v3)) * k_damp

    f_total = f_volume + f_damp

    f1 = f1 + dJdx1 * f_total
    f2 = f2 + dJdx2 * f_total
    f3 = f3 + dJdx3 * f_total
    f0 = (f1 + f2 + f3) * (0.0 - 1.0)

    # apply forces
    wp.atomic_sub(f, i, f0)
    wp.atomic_sub(f, j, f1)
    wp.atomic_sub(f, k, f2)
    wp.atomic_sub(f, l, f3)
```

Traditional Elastic Model

Neural Network

```python
@wp.kernel
def eval_tetrahedra(args):

    ...

    # setup feature vectors
    x = wp.vector(dtype=wp.float16, length=DIM_IN)

    # evaluate layers
    z = wp.linear(layer0.weights, layer0.bias, DIM_HID, x)
    z = wp.map(relu, z)
    z = wp.linear(layer1.weights, layer1.bias, DIM_OUT, z)
    z = wp.map(relu, z)

    # apply forces
    wp.atomic_sub(f, i, f0)
    wp.atomic_sub(f, j, f1)
    wp.atomic_sub(f, k, f2)
    wp.atomic_sub(f, l, f3)
```

Learned Constitutive Model

Roadmap

- Wrap Warp kernels as JAX primitives with one line of code
- Invoke JAX primitive inside @jax.jit code
- Very convenient way to mix JAX and Warp
- Available now in wp.jax_experimental
- Sharding, vmap(), pmap() support coming soon

```
import warp as wp
import jax
import jax.numpy as jp

# import Warp->JAX adapter
from warp.jax_experimental import jax_kernel

@wp.kernel
def triple_kernel(input: wp.array(dtype=float), output: wp.array(dtype=float)):
    tid = wp.tid()
    output[tid] = 3.0 * input[tid]

# create a JAX primitive from a Warp kernel
jax_triple = jax_kernel(triple_kernel)

# use the Warp kernel in a JAX jit'd function
@jax.jit
def f():
    x = jp.arange(0, 64, dtype=jp.float32)
    return jax_triple(x)

print(f())
```

Example: Warp kernel wrapped as a JAX primitive

Next Steps

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

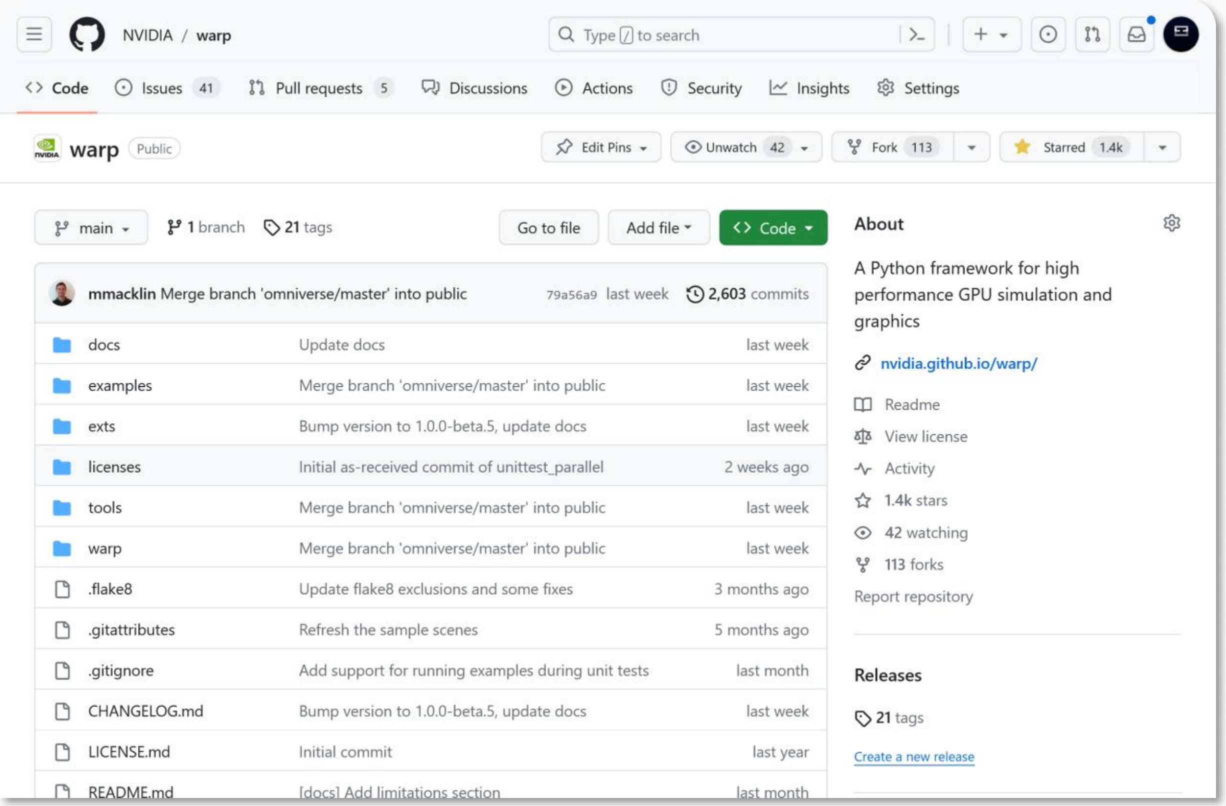
Platform support:

- Windows, Linux, macOS
- CUDA 11.5+, x86, aarch64, Jetson/Tegra

Deployment:

- Released as a standalone, open-source library
- Binary packages with monthly release cadence on PyPI
- Diverse set of example scripts
- omni.warp available in Omniverse extensions registry

- Links:
  - Repo: https://github.com/NVIDIA/warp
  - Docs: https://nvidia.github.io/warp/



```
pip install warp-lang
```

https://nvidia.github.io/warp/