Lecture 13:

Simulating Virtual Worlds for Training

Visual Computing Systems
Stanford CS348K, Spring 2024
Today

- Slides on the landscape of high-performance world simulation efforts designed for improving the efficiency of training embodied AI agents
- Discussion of the Madrona system
- Discussion: do we even need to simulate from a traditional world model at all?
  - Can’t generative AI just make our training data?
  - Setup for tonight’s reading: Genie
Recall from last time

1. Training and agent to learn complex skills can require many (millions, even billions) of trial-and-error steps (aka large amounts of “training experience”)

2. Researchers create virtual environments to simulate all this experience.
Basic training system components

Database of 3D assets
(meshes, textures, collision meshes)

World Construction
Load assets, programmatically create world instance

Simulation (advance world)
Run game logic, Physics calculations, etc.

Generate Agent Observations
Given agent viewpoint, render image, lidar, visibility, etc.

Compute Rewards/Loss (from observations)

Agent Policy Inference/Learning
DNN evaluation, gradient update step

Next agent action
A simple example game: Hide and seek with four agents
Hide and seek state: world entities
Hide and seek: examples of per-entity state (for agents)
Hide and seek: example observations (which agents does each agent see?)
Common simulation approach: treat simulator as a black box, increase simulation throughput via “scale-out” parallelization

Treat existing simulation engines as an unmodifiable black box.

Run many copies of the black box in parallel.
A basic design: parallelize over workers

Node 0
Simulation
Generate Obs
Env 0 state

Node 1
Simulation
Generate Obs
Env 1 state

Node 2
Simulation
Generate Obs
Env 2 state

Node 3
Simulation
Generate Obs
Env 3 state

DNN ops
Inference + learning: update policy from experience)
\( \pi(\theta) \)

Observations
Actions
Observations
Actions
Compute Rewards
One example of this design: EnvPool (one multi-core node)

- **Pros:**
  - Use any existing simulator, unmodified
  - Collects observations from environments, provides them to Python as a Tensor

- **Cons:**
  - See upcoming slides (simulator-learning code sync costs, running many independent simulators is not optimal on high throughput machines)
Similar design for distributed system: parallelize over workers

Rollouts (list of [obs, action, reward])

Learning
Update policy model from experience

New policy

Rollouts (list of [obs, action, reward])

Node 0
Simulation
Generate Obs
Compute Rewards
Policy Inference
Env 0 state

Node 1
Simulation
Generate Obs
Compute Rewards
Policy Inference
Env 1 state

Node 2
Simulation
Generate Obs
Compute Rewards
Policy Inference
Env 0 state

Node 3
Simulation
Generate Obs
Compute Rewards
Policy Inference
Env 3 state
Example: Rapid by (OpenAI)

Optimzer + Connected Rollout Workers (x256)

- Rollout Workers
  - ~500 CPUs
  - Run episodes
    - 80% against current bot
    - 20% against mixture of past versions
  - Randomized game settings
  - Push data every 60s of gameplay
    - Discount rewards across the 60s using generalized advantage estimation

- Optimizer
  - 1 P100 GPU
  - Compute Gradients
    - Proximal Policy Optimization with Adam
    - Batches of 4096 observations
    - BPTT over 16 observations

- Rollout Data Samples
- Gradient Updates

Eval Workers

- ~2500 CPUs
- Play in various environments for evaluation
  - vs hardcoded “scripted” bot
  - vs previous similar bots (used to compute Trueskill)
  - vs self (for humans to watch and analyze)

OpenAI Five stats (Dota 2)

<table>
<thead>
<tr>
<th>OpenAI Five</th>
<th>Stanford CS348K, Spring 2024</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
<td>128,000 preemptible CPU cores on GCP</td>
</tr>
<tr>
<td>GPUs</td>
<td>256 P100 GPUs on GCP</td>
</tr>
<tr>
<td>Experience collected</td>
<td>~180 years per day (~900 years per day counting each hero separately)</td>
</tr>
<tr>
<td>Size of observation</td>
<td>~36.8 kB</td>
</tr>
<tr>
<td>Observations per second of gameplay</td>
<td>7.5</td>
</tr>
<tr>
<td>Batch size</td>
<td>1,048,576 observations</td>
</tr>
<tr>
<td>Batches per minute</td>
<td>~60</td>
</tr>
</tbody>
</table>
Example public RL learning environments

- Lower complexity worlds
- Lower fidelity observations
In contrast: what a modern GPU is designed to render
(Very high fidelity observations)
Large-scale agent training is expensive!

OpenAI Five

Robotics in Virtual World

OpenAI Hide and Seek

Learning Dota 2: Months of training

64 GPUs over 2.5 days (2B experience samples)

High-level strategies emerge after billions of world time steps

<table>
<thead>
<tr>
<th>CPUs</th>
<th>128,000 preemptible CPU cores on GCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs</td>
<td>256 P100 GPUs on GCP</td>
</tr>
<tr>
<td>Experience collected</td>
<td>~180 years per day (~900 years per day counting each hero separately)</td>
</tr>
</tbody>
</table>
Design issues of basic scale-out approach

- Inefficient simulation/rendering: low-complexity worlds do not make good use of a modern parallel processor’s resources
  - GPUs won’t achieve high-throughput rendering/physics with smaller workloads

- Inefficient communication between simulation and inference/training

- Duplication of computation and memory footprint (for scene data) across environment simulator instances

- Seems wasteful, right?
A new visual computing systems research question:

Can we execute embodied AI training more efficiently if we architect a world simulation engine from the ground up to process many independent worlds at once?
Batch environment simulation
CuLE: Rewriting an Atari emulator in CUDA

- One CUDA thread = work for one enumerator instance
- Large numbers of threads execute

(a) Assault, 20M training frames
(b) Asterix, 20M training frames
(c) Ms-Pacman, 20M training frames
(d) Pong, 8M training frames
NVIDIA Issac Gym

- Batched many-environment execution applied to rigid body physics sim
- Simulate 100's to 1000's of world environments simultaneously on the GPU
- Current state for all environments packaged in a single PyTorch tensor
- User can write GPU-accelerated loss/reward functions in PyTorch on this tensor
- Result: tight loop of simulate/infer/train
Waymax

- Self-driving car simulator built using Jax programming environment
- Environment state stored in JAX tensors (max number of objects across all environments in a batch)

<table>
<thead>
<tr>
<th>Device</th>
<th>BS-1</th>
<th>BS-16</th>
<th>Reset</th>
<th>Step</th>
<th>Transition</th>
<th>Metrics</th>
<th>RolloutExpert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Agent Env</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>✓</td>
<td></td>
<td>1.09</td>
<td>131</td>
<td>0.90</td>
<td>112</td>
<td>1.0 × 10⁴</td>
</tr>
<tr>
<td>CPU</td>
<td>✓</td>
<td>✓</td>
<td>12.2</td>
<td>1.7 × 10³</td>
<td>10.9</td>
<td>1.69 × 10³</td>
<td>1.4 × 10⁵</td>
</tr>
<tr>
<td>GPU-v100</td>
<td>✓</td>
<td></td>
<td>0.58</td>
<td>0.75</td>
<td>0.47</td>
<td>0.21</td>
<td>56.2</td>
</tr>
<tr>
<td>GPU-v100</td>
<td>✓</td>
<td>✓</td>
<td>0.67</td>
<td>2.48</td>
<td>0.52</td>
<td>2.27</td>
<td>279</td>
</tr>
<tr>
<td>Multi-Agent Env</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>✓</td>
<td></td>
<td>6.23</td>
<td>129</td>
<td>1.01</td>
<td>112</td>
<td>1.1 × 10⁴</td>
</tr>
<tr>
<td>CPU</td>
<td>✓</td>
<td>✓</td>
<td>49.8</td>
<td>1.1 × 10³</td>
<td>14.3</td>
<td>1.72 × 10³</td>
<td>1.6 × 10⁵</td>
</tr>
<tr>
<td>GPU-v100</td>
<td>✓</td>
<td></td>
<td>0.64</td>
<td>0.92</td>
<td>0.53</td>
<td>0.19</td>
<td>73.3</td>
</tr>
<tr>
<td>GPU-v100</td>
<td>✓</td>
<td>✓</td>
<td>0.81</td>
<td>2.86</td>
<td>0.51</td>
<td>2.24</td>
<td>OOM</td>
</tr>
</tbody>
</table>

Table 2: Runtime benchmark in milliseconds: the environment controls all objects in the scene (up to 128 as defined in WOD).
Example: navigation in 3D scanned environments
(these are multi-room floorplans...)
40,000 fps batch rendering of small images from popular 3D scanned virtual environments (Gibson/Matterport)
100x End-to-End Gain with Optimized Policy DNN

ICLR 2020 Baseline (ResNet50)

Our System (ResNet50)

Our System (ResNet9)

Frames of Experience Per Second

Benchmarked with RTX 3090
2.5B frames of experience in 48 hrs on a single RTX 3090

“PointGoal Navigation” in Gibson environments (Habitat Labs)

Run render->infer->train loop runs at 13,000 fps per GPU

"Large Batch Simulation for Deep Reinforcement Learning", B. Shacklett, E. Wijmans, A. Petrenko, M. Savva, D. Batra, V. Koltun, K. Fatahalian, ICLR 2021
The story so far... reimplement learning environments as "batch simulators" for 100x Speedups

Isaac Gym: GPU Physics

~10000 envs per GPU

CuLE: Atari

~5000 envs per GPU

BPS3D: Home Navigation

~1000 envs per GPU

MuJoCo MJX

MuJoCo XLA (MJX)

Starting with version 3.0.0, MuJoCo includes MuJoCo XLA (MJX) under the mjx directory. MJX allows MuJoCo to run on compute hardware supported by the XLA compiler via the JAX framework. MJX runs on all platforms supported by JAX: Nvidia and AMD GPUs, Apple Silicon, and Google Cloud TPUs.

The MJX API is consistent with the main simulation functions in the MuJoCo API, although it is currently missing some features. While the API documentation is applicable to both libraries, we indicate features unsupported by MJX in the notes below.

MJX is distributed as a separate package called mujoco-npx on PyPI. Although it depends on the main mujoce package for model compilation and visualization, it is a re-implementation of MuJoCo.
What about training new kinds of agents for new tasks?
We need a “game engine” for building batch simulators!

Maximize throughput: millions of sim steps/sec for simple 3D environments (When running many environments in parallel)

Programmable: environment creators should be able to author diverse set of worlds, define custom world rules/behavior

Productive: quickly be able to create novel worlds
Key idea: the Entity Component System (ECS) architecture, which is a programming structure used in some games today, provides useful structure needed to build a many-world game engine on the GPU
Entities (ECS)
## Components (ECS)

### Agents

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
<th>Action</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
<td>[0,0,.5]</td>
<td>{min...}</td>
<td>LEFT</td>
<td>0.1</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>[2,1,0]</td>
<td>{min...}</td>
<td>FWD</td>
<td>-0.1</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
<td>[1.5,0,1]</td>
<td>{min...}</td>
<td>FWD</td>
<td>-2.5</td>
</tr>
</tbody>
</table>

### Obstacles

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
<td>[0.5,0,.5]</td>
<td>{min...}</td>
</tr>
<tr>
<td>72</td>
<td>0</td>
<td>[0,1,3]</td>
<td>{min...}</td>
</tr>
<tr>
<td>61</td>
<td>0</td>
<td>[1,1,2]</td>
<td>{min...}</td>
</tr>
</tbody>
</table>
## Batch ECS: Store Data Across All Environments in Unified Tables in GPU Memory

### Agents

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
<th>Action</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
<td>[0,0,.5]</td>
<td>{min...}</td>
<td>LEFT</td>
<td>0.1</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>[2,1,0]</td>
<td>{min...}</td>
<td>FWD</td>
<td>-0.1</td>
</tr>
<tr>
<td>51</td>
<td>0</td>
<td>[1.5,0,1]</td>
<td>{min...}</td>
<td>FWD</td>
<td>-2.5</td>
</tr>
<tr>
<td>62</td>
<td>1</td>
<td>[1.5,0.5,1]</td>
<td>{min...}</td>
<td>RIGHT</td>
<td>0.3</td>
</tr>
<tr>
<td>65</td>
<td>1</td>
<td>[-0.5,0,0]</td>
<td>{min...}</td>
<td>RIGHT</td>
<td>0.5</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>[0.5,1,1]</td>
<td>{min...}</td>
<td>LEFT</td>
<td>1.5</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>[-1.5,0,0]</td>
<td>{min...}</td>
<td>LEFT</td>
<td>10.5</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>[0.5,1,0]</td>
<td>{min...}</td>
<td>BACK</td>
<td>-10</td>
</tr>
</tbody>
</table>

### Obstacles

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
<td>[0.5,0.5]</td>
<td>{min...}</td>
</tr>
<tr>
<td>72</td>
<td>0</td>
<td>[0,1,3]</td>
<td>{min...}</td>
</tr>
<tr>
<td>61</td>
<td>0</td>
<td>[1,1,2]</td>
<td>{min...}</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>[0.5,1,2]</td>
<td>{min...}</td>
</tr>
<tr>
<td>70</td>
<td>1</td>
<td>[-0.5,1,0]</td>
<td>{min...}</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>[1.5,0,2.5]</td>
<td>{min...}</td>
</tr>
<tr>
<td>81</td>
<td>3</td>
<td>[2.5,0.5,2]</td>
<td>{min...}</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>[1.5,0.5,2]</td>
<td>{min...}</td>
</tr>
</tbody>
</table>
## Systems (ECS)

### Agents

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
<th>Action</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
<td>[0,0,.5]</td>
<td>{min...}</td>
<td>LEFT</td>
<td>0.1</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>[2,1,0]</td>
<td>{min...}</td>
<td>FWD</td>
<td>-0.1</td>
</tr>
<tr>
<td>51</td>
<td>2</td>
<td>[1.5,0,1]</td>
<td>{min...}</td>
<td>FWD</td>
<td>2.5</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>[2.5,0,1.5]</td>
<td>{min...}</td>
<td>RIGHT</td>
<td>1.1</td>
</tr>
</tbody>
</table>

### Obstacles

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
<td>[0.5,0,.5]</td>
<td>{min...}</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>[0,1,3]</td>
<td>{min...}</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>[1,1,2]</td>
<td>{min...}</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>[1.5,1,2.5]</td>
<td>{min...}</td>
</tr>
</tbody>
</table>

### ProcessActions

- **Inputs:** Pos, Action

### Collisions

- **Inputs:** Id, Pos, Bbox

### ComputeRewards

- **Inputs:** Pos, Reward
## Systems (ECS)

### Agents

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
<th>Action</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td>[0,0,.5]</td>
<td>{min...}</td>
<td>LEFT</td>
<td>0.1</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>[2,1,0]</td>
<td>{min...}</td>
<td>FWD</td>
<td>-0.1</td>
</tr>
<tr>
<td>51</td>
<td></td>
<td>[1.5,0,1]</td>
<td>{min...}</td>
<td>FWD</td>
<td>2.5</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>[2.5,0,1.5]</td>
<td>{min...}</td>
<td>RIGHT</td>
<td>1.1</td>
</tr>
</tbody>
</table>

### Obstacles

<table>
<thead>
<tr>
<th>Id</th>
<th>EnvID</th>
<th>Pos</th>
<th>Bbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
<td>[0.5,0,.5]</td>
<td>{min...}</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>[0,1,3]</td>
<td>{min...}</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>[1,1,2]</td>
<td>{min...}</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>[1.5,1,2.5]</td>
<td>{min...}</td>
</tr>
</tbody>
</table>

## Processes

- **ProcessActions**
  - Pos, Action

- **Collisions**
  - Id, Pos, Bbox

- **ComputeRewards**
  - Pos, Reward
ECS Systems Combined into Task Graph and Executed in Parallel on the GPU

ProcessActions
  ↓
GenerateEnv
  ↓
BVHRefit
  ↓
FindOverlaps
  ↓

BVHRefit
  ↓
PosSolve
  ↓
Integrate
  ↓
CheckVisible
  ↓
Lidar
  ↓

VelSolve
  ↓
NarrowPhase
  ↓
Integrate
  ↓

Observations
  ↓
MaskOutput
  ↓
ComputeRewards
Scheduling the ECS on the GPU
Fully GPU-Driven Scheduling Challenges

Dynamic GPU-Driven Memory Allocation:
- Game logic needs to create entities at runtime (during a simulation step)
- Entity lifetimes can vary wildly (<1 frame to hundreds)

How to Efficiently Execute Task Graph Given Dynamic Workload Each Frame?
- Task graph can contain > 100 nodes!
- # of entities matching each system may depend on prior nodes in task graph
Growable ECS Table Storage By (Ab)using GPU Virtual Memory Support

<table>
<thead>
<tr>
<th>Obstacles</th>
<th>Obstacles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Id</strong></td>
<td><strong>EnvID</strong></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New Entities Created

Unbacked Virtual Memory
Achieving Dynamic GPU Memory Allocation & Improving Coherence Using Parallel Radix Sort
Low-Overhead Dynamic Task Graph Execution Using Persistent Megakernel Design

Megakernel(taskgraph, envs):

while true:
    system_id, invocation_id = getNext(taskgraph)
    switch system_id:
        case 0: physicsSystemEntry(envs, invocation_id)
        case 1: ProcessActionEntry(envs, invocation_id)
        case 2: visibilitySystemEntry(envs, invocation_id)
        case -1: break

    threadFinished(taskgraph)

getNext(taskgraph):

node = taskgraph.currentNode()
if node.currentInvocation < node.numInvocations:
    return node.systemID, node.currentInvocation++

return taskgraph.advanceNode()
Example ECS System: Mapping GPU Threads to Hide & Seek ProcessAction

ProcessAction(env, id, pos, force, team, action):
    if action.type == MOVE:
        force = computeMovementForce(action.dir)
    if action.type == LOCK:
        hit_obj = raycastForward(env, pos)
        if hit_obj:
            ...

ProcessActionEntry(envs, ecs_state, gpu_thread_idx):
    ids, world_ids, positions, forces, teams, actions =
    ecs_state.getColumns<Id, EnvID, Pos, Force, Team, Action>()
    row = gpu_thread_idx
    env_id = env_ids[row]
    if env_id is not valid:
        return
    
    ProcessAction(envs[env_id], ids[row], positions[row],
                  forces[row], teams[row], actions[row])
Mitigating Megakernel Inefficiencies Using Profile-Guided Optimization

• Megakernel Implies One-Size-Fits-All Register Allocation
  - Observation: Can afford more than 1 kernel launch per batched simulation step

• Profile-Guided Optimization: Empirically test performance of each system with different register allocations & choose best!
  - Negligible cost in a 100 million step training run
Mitigating Megakernel Inefficiencies Using Profile-Guided Optimization

ProcessActions
  ↓
GenerateEnv
  ↓
BVHRefit
  ↓
FindOverlaps

VelSolve
  ↓
PosSolve
  ↓
Integrate
  ↓
NarrowPhase
  ↓
BVHRefit
  ↓
CheckVisible
  ↓
Lidar
  ↓
Observations
  ↓
MaskOutput
  ↓
ComputeRewards

2 Thread Blocks Per SM

4 Thread Blocks Per SM
Performance Analysis: One Step Across 16K Environments

1. BVH & Broad Phase
2. Physics Sub-Step
3. Agent Observations & Rewards
4. LIDAR
Procedural content creation
Significant value in diversity of scenes

Example: ProcTHOR

Procedurally generated floorplans, furniture arrangements, random material assignments, etc.
Greater diversity of scenes wins

Better off training on a large number of highly diverse scenes, than a small number of photorealistic ones
Generative AI as a means to generate world simulation output
Enhancing CG images to look like real-world images using image-to-image transfer
Modifying real-world images to create novel situations

Remove or move this car.
Genie

- Key idea: learn a world simulator from videos of video game play
  - From video, learn latency user actions, and dynamics model that steps work given (current state, action)
  - Then at “test time” given a novel world state (perhaps one generated from a prompt), and given user input, time step the novel world forward in time
Next time: the great debate

- Class debate: you have to choose a side!

- What’s the right way to build a world simulator?
  - Like a “game engine”: humans model and build a simulator
  - Data driven: just learn it from big data

- What are the pros/cons?

- In what situations might one be preferable?