InstantMimic: A High Performance System for Learning Physics-based Skills in Seconds

IKJUN CHOI, Seoul National University, South Korea
GEONHO LEEM, Seoul National University, South Korea
JUNGDAM WON*, Seoul National University, South Korea

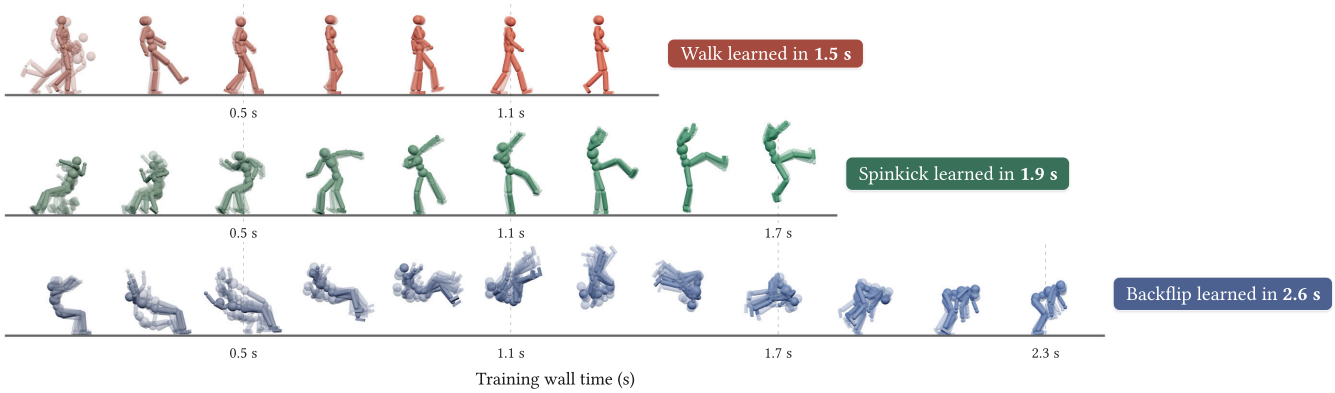


Fig. 1. A learning-progress visualization for walk, spinkick, and backflip motion.

Physics-based character control is a long-standing challenge in computer graphics and robotics, requiring policies that satisfy complex dynamics while producing realistic motion. Recent Deep RL approaches, particularly imitation learning methods such as DeepMimic, have had broad impact beyond animation, influencing robotics by enabling agile and expressive behaviors. While these approaches achieve impressive results, they remain computationally inefficient to train in practice. Despite GPU-accelerated simulation, we find that end-to-end pipelines often underutilize hardware due to overheads outside the physics solver, caused by fragmented GPU kernels and CPU memory access in the critical path. We present InstantMimic, a system that addresses these inefficiencies by making the entire training loop GPU-native. Built on a GPU-native physics backend, our unified pipeline integrates simulation, environment computation, policy inference, and policy updates within a single execution flow. As a result, InstantMimic compresses training for diverse physics-based skills to a few seconds, and makes LLM-agent-driven hyperparameter search practical.

Additional Key Words and Phrases: Character Animation, Physics-based Character Control, Reinforcement Learning, Motion Imitation, GPU-native Training, LLM-based Hyperparameter Optimization

1 Introduction

Physics-based character control has long been a central problem in computer graphics and robotics, enabling simulated agents to produce realistic and physically consistent motions. Unlike kinematic approaches, physics-based methods must satisfy complex dynamic constraints, making control significantly more challenging while offering greater realism and generalization.

To address this challenge, a line of work has leveraged deep reinforcement learning (Deep RL) to learn control policies directly from motion data. In particular, imitation learning frameworks such as DeepMimic [Peng et al. 2018] have demonstrated that neural policies can reproduce a wide range of dynamic skills, including locomotion, acrobatics, and transitions, by tracking reference motions. These approaches have significantly advanced the state of the art in physics-based animation, enabling robust and versatile controllers without requiring manual controller design. Beyond computer graphics, their impact has extended to robotics, where Deep RL-based imitation policies have become a foundation for learning agile and expressive humanoid behaviors [Liao et al. 2025; Ze et al. 2025].

Despite these advances, efficient training of such systems remains elusive. Recent GPU-accelerated physics simulators have enabled substantial speedups for reinforcement learning, yet simply executing the physics engine on the GPU does not guarantee efficient end-to-end learning. In practice, we observe that the GPU is often underutilized during training. Through profiling of a representative motion imitation pipeline, we find that the primary bottlenecks do not lie in the physics solver itself, but in the surrounding training pipeline. In particular, two sources of overhead consistently dominate runtime: *GPU kernel fragmentation*, caused by excessive fine-grained execution without sufficient batching or fusion, and *CPU memory access in the critical path*, caused by Python-level abstractions. Importantly, these inefficiencies are not specific to a particular task or framework, but commonly emerge in Deep RL training loops developed without end-to-end profiling and systems-level

*corresponding author

Authors' Contact Information: Ikjun Choi, ikjun@imo.snu.ac.kr, Seoul National University, Seoul, South Korea; Geonho Leem, geonholeem@imo.snu.ac.kr, Seoul National University, Seoul, South Korea; Jungdam Won, jungdam@imo.snu.ac.kr, Seoul National University, Seoul, South Korea.

optimization. As a result, even state-of-the-art implementations fail to fully utilize available hardware, leaving substantial performance untapped.

In this work, we revisit the design of the Deep RL training loop for physics-based character control from a systems perspective. We focus on eliminating the systemic inefficiencies that hinder existing approaches. Our key idea is to restructure the entire training loop to be fully GPU-native, minimizing kernel launch overhead through batching and fusion, and avoiding implicit synchronization. We design a unified execution model in which simulation, observation construction, reward evaluation, and policy updates are tightly integrated within a training loop, ensuring continuous device utilization throughout training.

As a result, our approach enables a dramatic reduction in training time. Our system learns policies that reproduce diverse physics-based skills within only a few seconds of training, while maintaining high motion fidelity and robustness. Beyond raw speed, this shift fundamentally changes how such systems are used. When training completes in seconds, manual hyperparameter tuning becomes the bottleneck: a human cannot iterate fast enough to exploit the speedup. An LLM agent offers two things manual tuning lacks: iteration at machine speed and unattended operation across many cycles, exploring the hyperparameter space and reliably discovering high-performing configurations. This transforms physics-based skill learning from a slow, trial-and-error process into a rapid and largely automated workflow.

We summarize our contributions as follows:

- **A system-level analysis of inefficiencies in the Deep RL training loop** for physics-based character control, identifying data movement, CPU-GPU synchronization, and kernel launch overhead as key bottlenecks.
- **Seconds-level learning of physics-based skills**, reducing training time from minutes or hours to only a few seconds.
- **Enabling automated experimentation**, including LLM agent-based hyperparameter optimization that iterates at machine speed with minimal human intervention.
- **A training framework for future research**, with the code and configurations used in this paper to be released publicly.

2 Related Work

2.1 GPU-accelerated Physics Simulation

Physics-based character control has long relied on CPU-based rigid body simulators such as ODE [Smith 2005] and MuJoCo [Todorov et al. 2012], which supported trajectory optimization and early deep RL in motor skill learning [Bergamin et al. 2019; Coros et al. 2010; Hodgins et al. 1995; Lee et al. 2010; Peng et al. 2018, 2021; Sok et al. 2007; Won et al. 2020; Yin et al. 2007]. As RL matured, on-policy algorithms such as PPO requiring billions of environment interactions made simulation throughput the binding bottleneck. Isaac Gym [Makoviychuk et al. 2021] and Brax [Freeman et al. 2021] addressed this by performing rigid body dynamics in parallel on the GPU, reducing humanoid controller training from days to minutes [Rudin et al. 2022]. This throughput in turn enabled skill learning frameworks that train on hours of unstructured motion data [Peng et al. 2022; Tessler et al. 2024, 2023]. Related efforts in differentiable and deformable simulation, including Taichi-based systems [Hu et al. 2019a,b], explored similar GPU-centric execution strategies for particle and continuum dynamics. More recently, MuJoCo Warp [Google DeepMind and NVIDIA 2025], built on NVIDIA Warp [Macklin 2022], brings this level of parallelism to the MuJoCo stack. We build our framework on MuJoCo Warp.

2.2 Optimization Principles in GPU Computing

The massive parallelism of modern GPUs has driven large speedups in deep RL training. Realizing this speedup in practice, however, requires careful attention to three classes of overhead that are well-established in the parallel computing literature [Kirk and Wen-Mei 2016].

Kernel launch overhead. A GPU kernel is a function-like program that is launched by the CPU and executed in parallel by many GPU threads. Every kernel launch carries a fixed CPU-side dispatch cost on the order of microseconds. For short-running kernels, such as small matrix multiplications, this cost can approach or exceed the kernel's own execution time, leaving the GPU idle between launches. **Kernel fusion** facilitated by JIT compilation frameworks such as Warp [Macklin 2022], merges multiple operations into a single kernel, reducing both the number of launches and intermediate memory traffic. **CUDA Graphs** [Gray 2019] instead capture a sequence of launches once and replay it with a single API call, eliminating per-launch CPU overhead without changing the kernels themselves. The two are orthogonal, one a compile-time transformation and the other a runtime batching mechanism, so a well-optimized pipeline uses both.

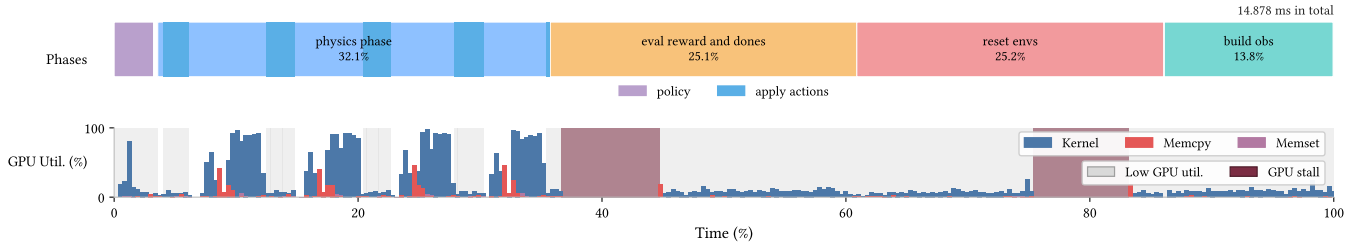
Data movement. On modern systems, data movement, not arithmetic, dominates latency [Horowitz 2014]. The principle is to access data in place rather than copy it, both across the CPU–GPU boundary and within device memory. End-to-end GPU simulators such as Isaac Gym [Makoviychuk et al. 2021] and Brax [Freeman et al. 2021] apply this zero-copy discipline by keeping rollout state resident on the device. The same discipline applies to *every* stage of the training loop such as policy inference, reward computation, and logging.

Synchronization. CPU–GPU data transfers are also the most common trigger of implicit synchronization: reading a GPU scalar on the CPU forces the CPU to wait for all preceding GPU work to complete, stalling the GPU pipeline even when the transfer itself is small. CUDA streams and events let the CPU and GPU execute asynchronously, but this benefit is lost whenever such a blocking access is issued, often implicitly through a high-level tensor API [Makoviychuk et al. 2021].

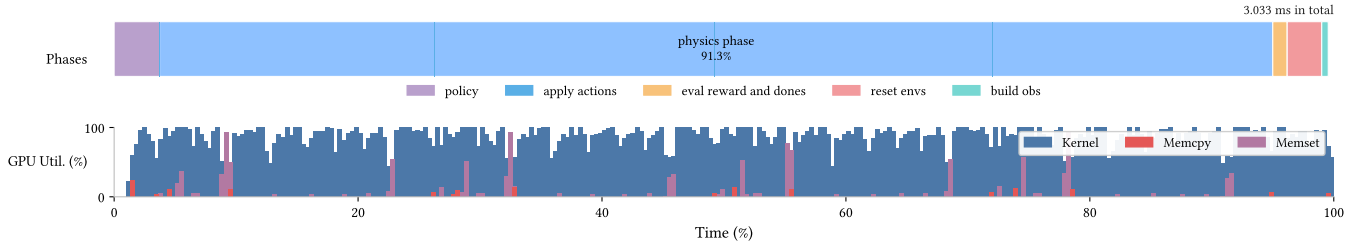
Madrona [Shacklett et al. 2023] applies all three principles to batch environment simulation, leaving policy inference and optimization to an external learning framework. However, we find that these stages, not the simulator, are the true bottleneck of the training loop.

2.3 Hyperparameter Optimization for Deep RL

The performance of a deep RL policy is sensitive to the hyperparameters used during training, and automated hyperparameter optimization (HPO) has therefore been a research topic since the early stages



(a) Isaac Lab-based training loop (baseline).



(b) GPU-native training loop (ours).

Fig. 2. GPU profiling timeline of a single rollout step in two PPO training loops.

of machine learning [Bergstra and Bengio 2012]. Since hyperparameters are not differentiable with respect to the training loss, most HPO methods treat training as a black box. Bayesian optimization (BO), which fits a probabilistic surrogate over past evaluations [Jones et al. 1998; Shahriari et al. 2015], is a representative example and serves as the backbone of production HPO services such as Google Vizier [Golovin et al. 2017] and Amazon SageMaker [Perrone et al. 2021]. For deep RL specifically, where the optimal hyperparameters are non-stationary over the course of training, Population Based Training (PBT) [Jaderberg et al. 2017] has been proposed to adapt a schedule of hyperparameters over the course of learning.

These methods, however, share a structural limitation: the number of evaluations required to reach a target performance grows rapidly with the dimensionality of the search space [Wang and Yeung 2016], forcing researchers in practice to restrict optimization to a small, hand-picked subset of hyperparameters. Designing such a search space well is crucial for achieving high performance [Bergstra and Bengio 2012]. However, search space design is still largely driven by human expertise.

Recent work has demonstrated that LLMs can act as a proxy for human expertise in configuring training. Some methods prompt an LLM to propose hyperparameters directly or use an LLM as a surrogate inside BO [Liu et al. 2024; Zhang et al. 2023], reducing the search cost over wide configuration spaces. More open-ended approaches go further by allowing LLM agents to directly modify training code [Chan et al. 2024; Ferreira et al. 2026; Karpathy 2025]. Because this process can be automated, LLM-based methods can explore configuration spaces more extensively than approaches that rely on human expertise.

3 Profiling Bottlenecks in the Baseline

Profiling a DeepMimic-style training loop on Isaac Lab [Mittal et al. 2025] reveals that, even with a GPU-based physics engine, the GPU is stalled or underutilized for 77% of the wall time per rollout step (Figure 2a).

```
while training:
    for _ in range(horizon): # rollout steps
        action = policy(obs)

        # physics phase
        for _ in range(num_physics_substeps):
            apply_action(action)
            simulate_physics()

        # post physics phase
        reward, terminated, truncated = eval_reward_and_dones()
        reset_envs(terminated | truncated)
        storage.write(obs, action, reward, terminated, truncated)
        obs = build_obs()

    update_policy(storage)
```

Fig. 3. Pseudocode of the PPO training loop.

To localize where these bottlenecks arise, we decompose a standard PPO training loop [Towers et al. 2025] shown in Figure 3. Each step of the rollout loop, from policy evaluation to rollout-storage write, is a *rollout step*. Within each rollout step, the *physics phase* covers action application and simulator substeps, and the *post-physics phase* covers the remaining work (reward, reset, observation, storage updates).

We identify two root causes for the observed bottlenecks: (i) fragmented post-physics kernels that prevent the GPU from staying busy between physics phases, and (ii) CPU memory access stalls that disrupt continuous GPU execution.

3.1 Low GPU Utilization from GPU Kernel Fragmentation

One might expect physics computation to dominate the *rollout step* wall time. However, profiling tells a different story. The *post-physics phase* accounts for 64% of each rollout step, exceeding the *physics*

phase at 32% (the sum of orange, red, cyan blocks vs. the blue block in Figure 2a). Even within the *physics phase*, `apply_action` alone consumes roughly 29% of this phase. Consequently, the physics simulation itself accounts for only 22.3% of the full rollout step. Worse, outside physics simulation, the GPU is active for only 7.3% of the total wall time on average, well below full utilization. The GPU therefore sits underutilized for the majority of each rollout step, despite every operation being GPU-resident in principle.

We attribute this underutilization to *kernel fragmentation*, where computation is dispatched as many short GPU kernels separated by CPU launch gaps, rather than a few large kernels. In the training loop, fragmentation arises from the implementation pattern. All *rollout step* work outside the physics simulation is commonly written as a sequence of PyTorch tensor operations invoked from the Python interpreter, and each operation dispatches its own GPU kernel even when it performs only a small amount of work. For such fragmented kernels, the kernel launch overhead becomes comparable to, or larger than, the GPU execution time, so the GPU frequently waits for the CPU to enqueue the next operation. The resulting timeline (gray block in Figure 2a) shows persistently low GPU utilization during these stages, consistent with fragmented short-kernel execution and CPU-side launch overhead.

3.2 CPU Memory Access in Critical Path Stalls the GPU

Another source of inefficiency is visible in the burgundy regions of Figure 2a, where the GPU remains stalled during CPU–GPU memory exchanges. Whenever data must be exchanged between CPU and GPU memory, the GPU cannot proceed until the transfer completes, and the CPU cannot proceed until the GPU has produced or consumed the data. Each such exchange therefore stalls the GPU on the critical path of the *rollout step*. Python abstractions hide these costs behind a tensor-like interface, leaving them invisible at the call site; profiling reveals them at a glance. We illustrate with two representative patterns from the Isaac Lab baseline.

The state accessor API of Isaac Lab `body_com_pose_b` (Figure 4a), invoked during `build_obs`, reads from a CPU-resident simulation buffer and copies the result to the GPU on every call, aligning with an approximately 15.9 % GPU stall in Figure 2a.

The `reset` method stalls the GPU through a transfer in the opposite direction, from GPU back to CPU. Termination flags are computed on the GPU, but the Python branch must read a scalar back to the CPU via `.item()` before launching reset (Figure 4b). The lone `.item()` call is easy to overlook in code review yet forces a full GPU sync, placing a synchronization point on the *rollout step*.

4 GPU-Native Training Loop

We built a GPU-native training loop that addresses the bottlenecks identified in Section 3. To reduce kernel fragmentation, we apply kernel fusion and CUDA Graph replay techniques across the *post-physics phase* and PPO updates. To eliminate CPU memory access, we keep state access, reset handling, and training summaries on the GPU. This loop is built on MuJoCo Warp, a GPU-native MuJoCo backend that exposes simulator state as GPU arrays replacing Isaac Lab used in the baseline.

```python
@property
def body_com_pose_b(self) -> torch.Tensor:
    # Below copies CPU memory into GPU
    pose = self._root_physx_view.get_coms().to(self.device)
    pose[..., 3:7] = math_utils.convert_quat(pose[..., 3:7],
    ↪  to="wxyz")
    return pose
```

(a) Simulation state accessors.

```python
terminated = compute_termination()
if terminated.any().item():
    reset(terminated)
```

(b) Conditional reset.

Fig. 4. Two CPU memory access patterns in Isaac Lab.

4.1 Reduce GPU Kernel Fragmentation

We illustrate this on a reward computation by describing two techniques for reducing fragmentation, which together yield a roughly 90× speedup (Figure 5). The first reduces launch gaps between existing kernels. The second fuses operations into larger kernels, avoiding intermediate operations and temporary tensors.

To isolate the effect of each technique, we prepare four variants of the reward computation — vanilla PyTorch (*vanilla*), PyTorch JIT (*jit*), CUDA Graph (*cgraph*), and NVIDIA Warp (*warp*). The *vanilla* uses the typical PyTorch tensor operation API. The *jit* partially fuses tensor operations using the PyTorch JIT API, but still leaves the reward computation split across multiple GPU kernels; this technique is used by Isaac Lab. The *cgraph* uses CUDA Graphs to preserve the original kernel function bodies and launch sequence while reducing gaps between kernel launches through single-launch graph replay. The *warp* variant instead uses NVIDIA Warp to JIT-compile the reward computation into a single GPU kernel, eliminating intermediate operations and approaching the fragmentation-free limit.

We break down the contribution of each technique by comparing the variants. The *jit* applies partial fusion but inherits *vanilla*'s per-kernel launch gaps, yielding only a 2.5× speedup (281 → 112 μs). The *cgraph* keeps the same kernels but eliminates the launch gaps via single-shot graph replay, yielding a 9.7× speedup (29 μs). The *warp* variant, which expresses the reward as a single GPU kernel, benefits from both: no intermediate tensors (one kernel body) and no launch gaps (one launch). The result is 89.7× (3.2 μs), nearly two orders of magnitude over *vanilla*. Equivalently, the shrinking idle gaps across these variants (Figure 5c) confirm the diagnosis in Section 3: kernel fragmentation causes low GPU utilization.

We therefore implement the *post-physics phase* as explicit *warp* kernels. For PPO updates we use the PyTorch-side equivalent of *warp* (`torch.compile`).

4.2 Eliminate CPU Memory Access in Critical Path

Eliminating CPU memory access on the *rollout step* turns the stalled profile of Figure 2a into the stall-free profile of Figure 2b, recovering the 16% of wall time the GPU previously spent waiting on the CPU. We achieve this by relocating the two patterns identified in Section 3. Our GPU-native implementation reads directly from GPU-resident

simulation arrays, so *post-physics phase* kernels obtain simulator state without the CPU-side copy of Figure 4a. The conditional reset of Figure 4b becomes unconditional; the rollout graph always executes the reset logic, and a GPU-side mask restricts the actual update to the terminated environments.

```python
def tracking_reward_torch(...):
    pose_err = (body_pos - ref_body_pos).square().sum(dim=(-1, -2))
    vel_err = (body_vel - ref_body_vel).square().sum(dim=(-1, -2))
    com_err = (com_pos - ref_com_pos).square().sum(dim=-1)

    reward = (0.60 * torch.exp(-4.0 * pose_err)
            + 0.25 * torch.exp(-0.25 * vel_err)
            + 0.15 * torch.exp(-8.0 * com_err))
    done = (root_height < 0.55) | (reward < 0.15)

    reward_out.copy_(reward)
    done_out.copy_(done.to(torch.int32))
```

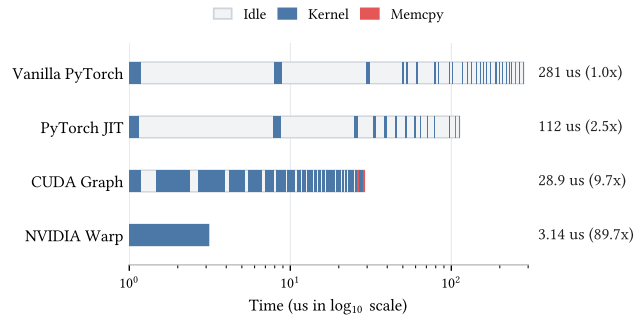
(a) PyTorch-based reward computation.

```python
@wp.kernel
def tracking_reward_warp(...):
    env = wp.tid()
    pose_err = float(0.0)
    vel_err = float(0.0)

    for body in range(NUM_BODIES):
        dp = body_pos[env, body] - ref_body_pos[env, body]
        dv = body_vel[env, body] - ref_body_vel[env, body]
        pose_err += wp.dot(dp, dp)
        vel_err += wp.dot(dv, dv)

    dc = com_pos[env] - ref_com_pos[env]
    reward = (0.60 * wp.exp(-4.0 * pose_err)
            + 0.25 * wp.exp(-0.25 * vel_err)
            + 0.15 * wp.exp(-8.0 * wp.dot(dc, dc)))
    reward_out[env] = reward
    done_out[env] = int(root_height[env] < 0.55 or reward < 0.15)
```

(b) NVIDIA Warp-based reward computation.



(c) GPU kernel profile for vanilla PyTorch, PyTorch JIT, CUDA Graph, and Warp implementations.

Fig. 5. Reward computation benchmark illustrating the techniques for reducing kernel fragmentation.

5 LLM Agent Based Hyperparameter Optimization

Due to the high computational efficiency of our framework, evaluating controller performance under different hyperparameter configurations can be performed within seconds. This drastically reduces the cost of hyperparameter search compared to conventional approaches, where each trial may require substantial training time. Such efficiency enables a fundamentally different approach to hyperparameter optimization. Instead of relying on manual tuning or coarse search strategies, our system allows for rapid iterative experimentation, making it feasible to integrate agentic AI that autonomously explore and refine hyperparameters through repeated trials. In this setting, the agent can actively design experiments, evaluate outcomes, and adapt its strategy in a closed loop, effectively discovering high-performing configurations with minimal human intervention.

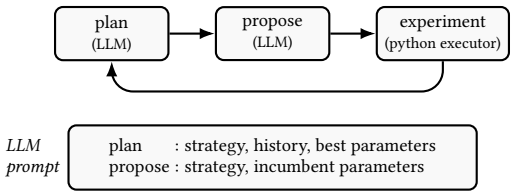


Fig. 6. LLM agent hyperparameter optimization loop.

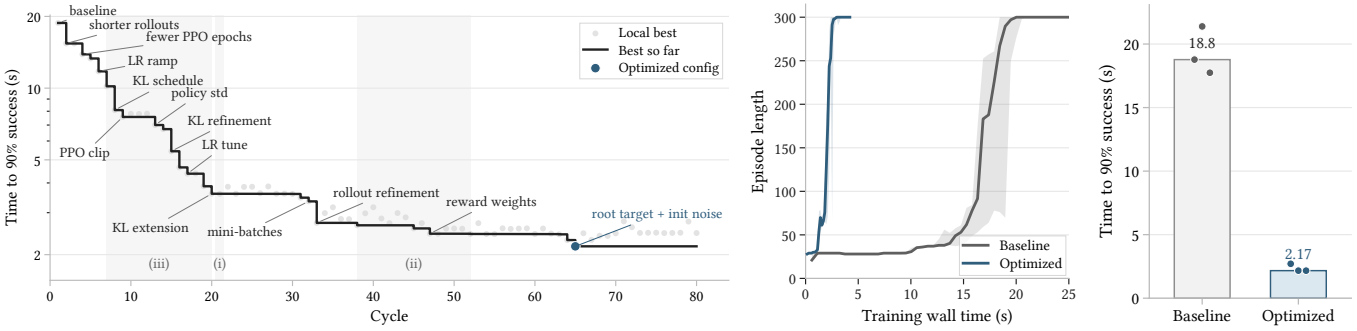
Specifically, we use an LLM agent to find hyperparameters, where it interprets training metrics and reshapes the search space as the run progresses. The procedure is a loop with three stages (Figure 6). The agent maintains four artifacts as persistent state across cycles: a natural-language *strategy* that records what to focus on next, a *history* of prior cycles' candidates and results, the *incumbent* parameters that the current search builds on, and the global *best* parameters observed so far. The cycle begins with *plan*, an LLM call that reads the strategy, history, and best parameters, and rewrites the strategy with the next concrete move: which hyperparameter axis to sweep, whether to expand its bounds, or whether to pivot to a different class. *propose*, a second LLM call, then reads the revised strategy and the incumbent and emits candidate values for one hyperparameter. *experiment*, a deterministic stage implemented in python, trains and benchmarks every candidate with multiple seeds and aggregates the results into the history, updates the incumbent if any candidate improves on it, and updates the best if a new global high is reached. The human supplies only the benchmark protocol and the stopping criterion; everything else is the agent's responsibility.

6 Results

We remove the training-loop bottlenecks (Section 3) with a GPU-native training loop (Section 4), and use its seconds-level training time to drive an LLM agent hyperparameter optimization (Section 5). This section evaluates the training loop throughput, motion tracking, and scaling to large motion datasets with downstream tasks.

6.1 Implementation Details

We train a humanoid character on a single NVIDIA RTX 5090 with MuJoCo Warp as the simulator. The reward is a DeepMimic-style imitation reward [Peng et al. 2018] written in multiplicative form $r = \prod_{k \in \mathcal{K}} \exp(-w_k e_k)$ over joint pose, joint velocity, center-of-mass position, and end-effector position and orientation errors. Observations follow PHC [Luo et al. 2023]: root kinematics, joint state, body poses and velocities expressed in the character's heading frame using the continuous 6D orientation representation [Zhou et al. 2019], future reference end-effector frames, and the previous action. Actions are normalized PD targets mapped to per-joint scales.



(a) Optimization trajectory of the LLM agent over successive tuning cycles.

(b) Learning curves for each configurations.

(c) Training wall time across repeated runs.

Fig. 7. LLM agent hyperparameter optimization on the backflip motion tracking task. The objective is the train time required to reach a success rate over 0.9.

Table 1. Three representative optimization plans corresponding to the regions marked (i)–(iii) in Figure 7a

#	Strategy	Reason
(i)	Select `ratio_clip`, the PPO update-size knob most coupled to the new high-KL regime, to stabilize tracking.	Cycle 21: with `kl_threshold` newly raised to 0.12, success rate held at 1.0 but average pose error and end-effector tracking error worsened.
(ii)	Pivot out of the PPO class. Reward-weight sweeps over cycles 47–52 produced a new best of 2.449 s at cycle 47 (`com_reward_weight`).	Cycles 38–46: six PPO/exploration axes (`motion_rand_std`, policy log-std, rollouts, action scale, future frames, prev-actions observation) had all failed to advance from a 2.656 s plateau.
(iii)	Ratchet the bound outward each cycle, walking 0.004 → 0.12 (30× range).	Cycles 7–20: each winning value sat at the current upper bound of `kl_threshold`.

We train with PPO, with early termination and adaptive episode-initialization sampling following [Peng et al. 2018; Won et al. 2020; Won and Lee 2019].

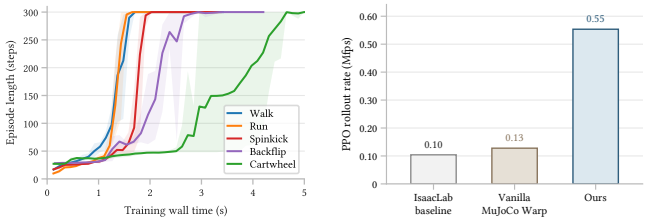
6.2 LLM Agent Hyperparameter Optimization

We use an LLM agent (GPT 5.5 high) to find hyperparameters that train a backflip motion tracking policy, where the LLM agent achieves an 8.6× speedup (2.17 s vs. 18.77 s) on the backflip motion tracking (Figure 7). Of greater interest than the speedup itself is the character of the agent's decisions during the search. Table 1 (shaded area in the Figure 7a) catalogs three exemplar *plan* steps from our run, each tied to the cycle-log evidence that triggered it. These exemplify three forms of search-structure modification: (i) inferring the mechanism behind a sub-metric regression, (ii) pivoting between hyperparameter classes when one is exhausted, and (iii) expanding the search bounds when winners sit at the boundary. None of these is available to a fixed-search-space optimizer such as Bayesian optimization (BO), nor to a script that sweeps a predefined grid. Such reasoning requires expert-level knowledge that fixed-search-space optimizers do not provide, applied at a per-trial pace no human can match. The LLM agent closes both gaps.

6.3 Motion Tracking

All five reference motions converge within seconds (Figure 8a). The motion set covers locomotion (walk, run) and acrobatic skills (spinkick, backflip, cartwheel); each policy reaches its tracking target between 1.5 and 4.5 seconds of training. Representative frames from each trained policy are shown in Figure 9.

Our training loop achieves 5.32× higher rollout throughput (Mfps) than the Isaac Lab baseline and 4.33× higher than vanilla MuJoCo Warp on the walk-tracking task (Figure 8b).



(a) Learning curves for the five reference motions.

(b) Rollout throughput on the walk motion tracking task.

Fig. 8. Training performance of our GPU-native training loop.

6.4 Scaling to Large Motion Datasets

We pretrain a VAE-based latent controller on the AMASS [Mahmood et al. 2019] training split of PHC [Luo et al. 2023] (37.4 hours of motion), reaching a success rate comparable to the PHC baseline within 30 minutes (Figure 10). To demonstrate downstream reuse, we freeze the pretrained decoder and train a goal-position tracking policy in its latent action space within one minute; qualitative behavior is shown in the supplemental video.

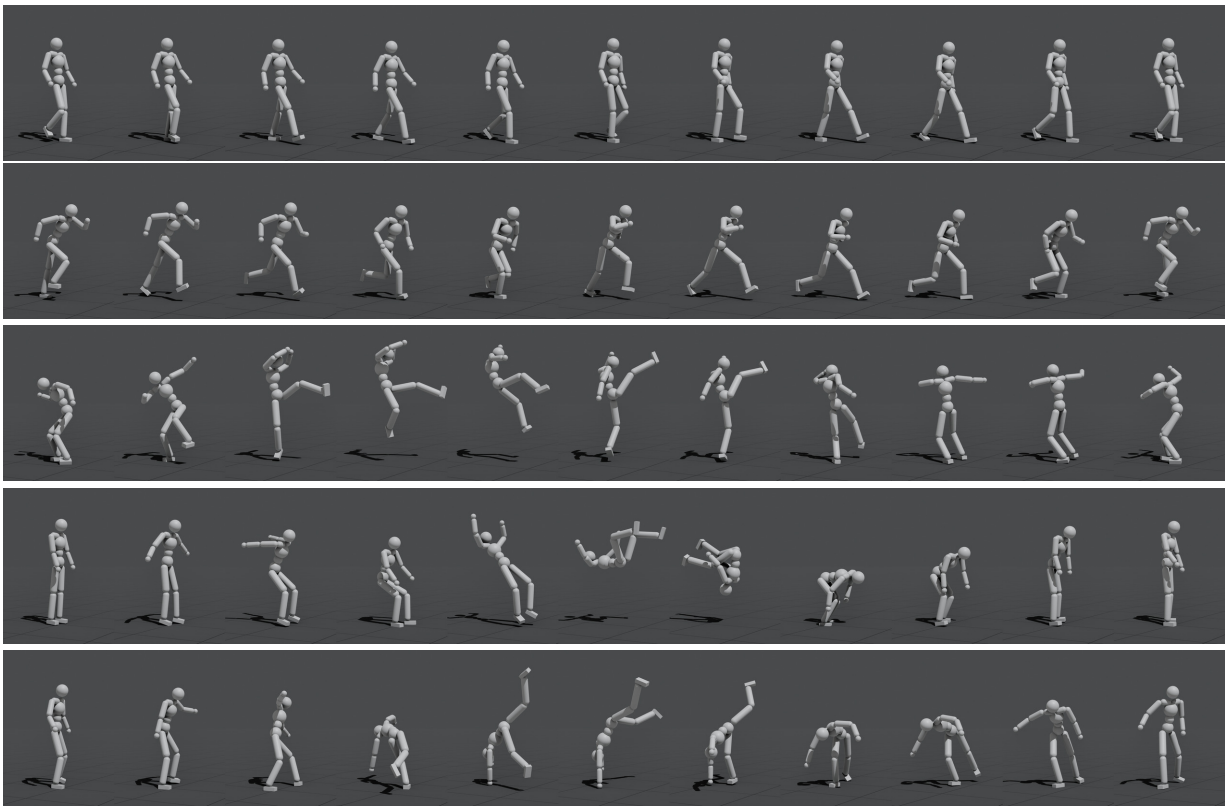


Fig. 9. Qualitative motion-tracking results of five motions. From top to bottom: walk, run, spinkick, backflip, and cartwheel.

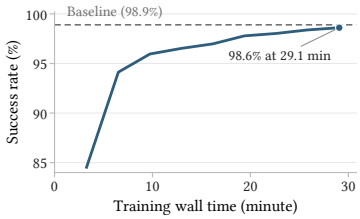


Fig. 10. Learning curves for latent-controller pretraining on 37.4 hours of AMASS motion data.

7 Conclusion

We revisited the design of the Deep RL training loop for physics-based character control from a systems perspective. Profiling revealed that the Isaac Lab baseline underutilized the GPU for most of each rollout step due to kernel fragmentation around physics execution and CPU-mediated memory access on the rollout critical path. By introducing optimized GPU kernels and direct GPU-memory state access, we substantially improved training throughput and reduced rollout overhead.

The resulting system enables training times on the order of seconds for standard motion-tracking tasks and reduces large-scale latent-controller pretraining on the 37.4-hour AMASS dataset to 30 minutes. These speedups also make automated experimentation significantly more practical. In particular, an LLM agent-based optimization was able to iteratively refine training configurations and reduce backflip training time from 18.77 s to 2.17 s over 80 optimization cycles.

Our findings suggest that, in modern GPU RL pipelines, the primary bottlenecks often lie not in simulation itself but in the surrounding training infrastructure. More broadly, we believe that aggressively reducing iteration time changes how RL systems should be developed: once experiments become cheap enough to run at scale, automated search and optimization become increasingly powerful tools for controller design. We hope our work provides a foundation for future research in this direction.

References

Kevin Bergamin, Simon Clavet, Daniel Holden, and James Richard Forbes. 2019. DReCon: data-driven responsive control of physics-based characters. *ACM Transactions On Graphics (TOG)* 38, 6 (2019), 1–11.

James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, 2 (2012).

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. 2024. Mlebench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095* (2024).

Stelian Coros, Philippe Beaudoin, and Michiel Van de Panne. 2010. Generalized biped walking control. *ACM Transactions On Graphics (TOG)* 29, 4 (2010), 1–9.

Fabio Ferreira, Lucca Wobbe, Arjun Krishnakumar, Frank Hutter, and Arber Zela. 2026. Can LLMs Beat Classical Hyperparameter Optimization Algorithms? A Study on autoresearch. *arXiv preprint arXiv:2603.24647* (2026).

C Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. Brax–a differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281* (2021).

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495.

Google DeepMind and NVIDIA. 2025. MuJoCo Warp: GPU-optimized MuJoCo simulation with NVIDIA Warp. https://github.com/google-deepmind/mujoco_warp. Accessed: 2026-04-24.

Alan Gray. 2019. Getting Started with CUDA Graphs. NVIDIA Technical Blog. https://developer.nvidia.com/blog/cuda-graphs/ Accessed: 2026-04-24.

Jessica K Hodgins, Wayne L Wooten, David C Brogan, and James F O'Brien. 1995. Animating human athletics. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 71–78.

Mark Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. doi:10.1109/ISSCC.2014.6757323

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019a. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.

Yuanming Hu, Jiancheng Liu, Andrew Spielberg, Joshua B Tenenbaum, William T Freeman, Jiajun Wu, Daniela Rus, and Wojciech Matusik. 2019b. Chainqueen: A real-time differentiable physical simulator for soft robotics. In *2019 International conference on robotics and automation (ICRA)*. IEEE, 6265–6271.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).

Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global optimization* 13, 4 (1998), 455–492.

Andrej Karpathy. 2025. autoresearch. https://github.com/karpathy/autoresearch. GitHub repository.

David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

Yoonsang Lee, Sungeun Kim, and Jehee Lee. 2010. Data-driven biped control. In *ACM SIGGRAPH 2010 papers*. 1–8.

Qiayuan Liao, Takara E Truong, Xiaoyu Huang, Yuman Gao, Guy Tevet, Koushil Sreenath, and C Karen Liu. 2025. Beyondmimic: From motion tracking to versatile humanoid control via guided diffusion. *arXiv preprint arXiv:2508.08241* (2025).

Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. 2024. Large language models to enhance bayesian optimization. *arXiv preprint arXiv:2402.03921* (2024).

Zhengyi Luo, Jinkun Cao, Kris Kitani, Weipeng Xu, et al. 2023. Perpetual humanoid control for real-time simulated avatars. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 10895–10904.

Miles Macklin. 2022. Warp: A High-performance Python Framework for GPU Simulation and Graphics. https://github.com/NVIDIA/warp NVIDIA GPU Technology Conference (GTC).

Naureen Mahmood, Nima Ghorbani, Nikolaus F Troje, Gerard Pons-Moll, and Michael J Black. 2019. AMASS: Archive of motion capture as surface shapes. In *Proceedings of the IEEE/CVF international conference on computer vision*. 5442–5451.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. 2021. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470* (2021).

Mayank Mittal, Pascal Roth, James Tigue, Antoine Richard, Octi Zhang, Peter Du, Antonio Serrano-Munoz, Xinjie Yao, René Zurbrügg, Nikita Rudin, et al. 2025. Isaac lab: A gpu-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831* (2025).

Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel Van de Panne. 2018. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions On Graphics (TOG)* 37, 4 (2018), 1–14.

Xue Bin Peng, Yunrong Guo, Lina Halper, Sergey Levine, and Sanja Fidler. 2022. Ase: Large-scale reusable adversarial skill embeddings for physically simulated characters. *ACM Transactions On Graphics (TOG)* 41, 4 (2022), 1–17.

Xue Bin Peng, Ze Ma, Pieter Abbeel, Sergey Levine, and Angjoo Kanazawa. 2021. Amp: Adversarial motion priors for stylized physics-based character control. *ACM Transactions on Graphics (ToG)* 40, 4 (2021), 1–20.

Valerio Perrone, Huibin Shen, Aida Zolic, Iaroslav Shcherbatyi, Amr Ahmed, Tanya Bansal, Michele Donini, Fela Winkelmolen, Rodolphe Jenatton, Jean Baptiste Faddoul, et al. 2021. Amazon sagemaker automatic model tuning: Scalable gradient-free optimization. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3463–3471.

Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. 2022. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on robot learning*. PMLR, 91–100.

Brennan Shacklett, Luc Guy Rosenzweig, Zhiqiang Xie, Bidipta Sarkar, Andrew Szot, Erik Wijmans, Vladlen Koltun, Dhruv Batra, and Kayvon Fatahalian. 2023. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Transactions on Graphics (TOG)* 42, 4 (2023), 1–13.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

Russell Smith. 2005. Open Dynamics Engine. http://www.ode.org

Kwang Won Sok, Manmyung Kim, and Jehee Lee. 2007. Simulating biped behaviors from human motion data. In *ACM SIGGRAPH 2007 papers*. 107–es.

Chen Tessler, Yunrong Guo, Ofir Nabati, Gal Chechik, and Xue Bin Peng. 2024. Maskedmimic: Unified physics-based character control through masked motion inpainting. *ACM Transactions On Graphics (TOG)* 43, 6 (2024), 1–21.

Chen Tessler, Yoni Kasten, Yunrong Guo, Shie Mannor, Gal Chechik, and Xue Bin Peng. 2023. Calm: Conditional adversarial latent models for directable virtual characters. In *ACM SIGGRAPH 2023 conference proceedings*. 1–9.

Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 5026–5033.

Mark Towers, Ariel Kwiatkowski, John Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Kallinteris Andreas, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, J Terry, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. 2025. Gymnasium: A Standard Interface for Reinforcement Learning Environments. In *Advances in Neural Information Processing Systems*, D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (Eds.), Vol. 38. Curran Associates, Inc. doi:10.52202/085713-4916

Hao Wang and Dit-Yan Yeung. 2016. Towards Bayesian deep learning: A framework and some existing methods. *IEEE Transactions on Knowledge and Data Engineering* 28, 12 (2016), 3395–3408.

Jungdam Won, Deepak Gopinath, and Jessica Hodgins. 2020. A scalable approach to control diverse behaviors for physically simulated characters. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 33–1.

Jungdam Won and Jehee Lee. 2019. Learning body shape variation in physics-based characters. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–12.

KangKang Yin, Kevin Loken, and Michiel Van de Panne. 2007. Simbicon: Simple biped locomotion control. *ACM Transactions on Graphics (TOG)* 26, 3 (2007), 105–es.

Yanjie Ze, Siheng Zhao, Weizhuo Wang, Angjoo Kanazawa, Rocky Duan, Pieter Abbeel, Guanya Shi, Jiajun Wu, and C Karen Liu. 2025. Twist2: Scalable, portable, and holistic humanoid data collection system. *arXiv preprint arXiv:2511.02832* (2025).

Michael R Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. 2023. Using large language models for hyperparameter optimization. *arXiv preprint arXiv:2312.04528* (2023).

Yi Zhou, Connelly Barnes, Jingwan Lu, Jimei Yang, and Hao Li. 2019. On the continuity of rotation representations in neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5745–5753.