An Extensible, Data-Oriented Architecture for High-Performance, Many-World Simulation

BRENNAN SHACKLETT and LUC GUY ROSENZWEIG, Stanford University, USA
ZHIQIANG XIE and BIDIPTA SARKAR, Stanford University, USA
ANDREW SZOT and ERIK WIJMANS, Georgia Institute of Technology, USA
VLADLEN KOLTUN, Jackson, WY, USA
DHRUV BATRA, Georgia Institute of Technology, USA
KAYVON FATAHALIAN, Stanford University, USA

Training AI agents to perform complex tasks in simulated worlds requires millions to billions of steps of experience. To achieve high performance, today's fastest simulators for training AI agents adopt the idea of batch simulation: using a single simulation engine to simultaneously step many environments in parallel. We introduce a framework for productively authoring novel training environments (including custom logic for environment generation, environment time stepping, and generating agent observations and rewards) that execute as high-performance, GPU-accelerated batched simulators. Our key observation is that the entity-component-system (ECS) design pattern, popular for expressing CPU-side game logic today, is also well-suited for providing the structure needed for high-performance batched simulators. We contribute the first fully-GPU accelerated ECS implementation that natively supports batch environment simulation. We demonstrate how ECS abstractions impose structure on a training environment's logic and state that allows the system to efficiently manage state, amortize work, and identify GPU-friendly coherent parallel computations within and across different environments. We implement several learning environments in this framework, and demonstrate GPU speedups of two to three orders of magnitude over open source CPU baselines and 5-33× over strong baselines running on a 32-thread CPU. An implementation of the OpenAI hide and seek 3D environment written in our framework, which performs rigid body physics and ray tracing in each simulator step, achieves over 1.9 million environment steps per second on a single GPU.

CCS Concepts: • **Computing methodologies → Graphics systems and interfaces**.

Additional Key Words and Phrases: game AI, reinforcement learning

ACM Reference Format:
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 Trans. Graph.* 42, 4, Article 1 (August 2023), 13 pages. https://doi.org/10.1145/3592427

Authors' addresses: Brennan Shacklett, bps@cs.stanford.edu; Luc Guy Rosenzweig, lrosenz@cs.stanford.edu, Stanford University, USA; Zhiqiang Xie, xiezhq@cs.stanford.edu; Bidipta Sarkar, bidiptas@cs.stanford.edu, Stanford University, USA; Andrew Szot, aszot3@gatech.edu; Erik Wijmans, etw@gatech.edu, Georgia Institute of Technology, USA; Vladlen Koltun, vkoltun@gmail.com, Jackson, WY, USA; Dhruv Batra, dbatra@gatech.edu, Georgia Institute of Technology, USA; Kayvon Fatahalian, kayvonf@cs.stanford.edu, Stanford University, USA.

0730-0301/2023/8-ART1 $15.00
https://doi.org/10.1145/3592427

1 INTRODUCTION

Training AI agents to learn complex problem solving skills in simulated environments is of high interest to graphics, robotics, and foundational AI. For example, by learning from millions to billions of steps of experience, reinforcement learning (RL) methods can produce agents that demonstrate intelligent team play in video games [OpenAI et al. 2019; Vinyals et al. 2019], perform human-like athletic skills [Ling et al. 2020; Peng et al. 2022; Won et al. 2021], use tools to achieve high-level goals [Baker et al. 2020], and even play test video games [Bergdahl et al. 2020; Sestini et al. 2022]. Unfortunately, even with supercomputing-scale resources, training policies for state-of-the-art learning tasks can take days to complete. As learning tasks become increasingly complex, we expect both the amount of required experience and the complexity of the environments being simulated to grow. This hinders progress in the field and reduces the practicality of training advanced AI agents.

To address the high cost of environment simulation, recent efforts have redesigned simulators from the ground up to achieve higher efficiency when training agents [Dalton et al. 2020; Freeman et al. 2021; Hessel et al. 2021; Makoviychuk et al. 2021; Shacklett et al. 2021]. These efforts share the idea of *batched simulation*, executing many independent environments (training episodes) at the same time within a single simulator engine. Visibility into the state of many concurrent environments allows batched simulators to exploit parallelism and coherence (both instruction and data) across environments to achieve high efficiency on throughput-oriented computing platforms like GPUs and TPUs. As a result, batch simulators can realize orders of magnitude higher training performance than solutions that naively parallelize by running copies of pre-existing simulators on different threads or machines.

The challenge is that writing an efficient batched simulator requires task domain knowledge and parallel programming expertise. For this reason, existing batch simulators provide a limited, fixed set of functionality (e.g., a simulator for 3D dynamics simulation, agent navigation in 3D environments, or Atari games). Creating a new batched simulator for a novel learning task requires a developer to modify an existing batch simulator's internals, or create a new high-performance simulator from scratch. This contrasts with the process for creating novel interactive environments using popular game engines like Unity [Juliani et al. 2018] or Unreal [Epic Games 2022]. These engines are scriptable and extensible, and provide APIs for developers to specify the logic of their game without knowledge of how the engine internally executes those computations efficiently.

In this paper we pursue the best of both worlds. We seek a programmable framework that facilitates productive creation of novel, customized agent training environments, but also realizes the high performance of GPU-accelerated, batched environment simulation designs. We make and operationalize the observation that the entity-component-system (ECS) architecture, a widely used design pattern that facilitates extensible development and parallel execution on multi-core CPUs, is also well-suited for authoring batched simulators that run efficiently on the large-scale, fine-grained parallelism of GPUs. Specifically, our contributions are:

- We identify that ECS programming abstractions are well suited for authoring novel, GPU-accelerated batch environment simulators. These designs permit expression of custom game logic using simple, parallelism-free programming concepts (a function per world, per agent, per object) and provide efficient interfaces between the subsystems required to simulate custom learning environments. ECS abstractions also provide crucial structure over custom logic and state that allows batch simulators to efficiently manage memory, amortize work, and identify coherent parallel computations within and across different environments.
- We provide Madrona, an ECS implementation that efficiently executes batch environment simulation on a GPU. Madrona organizes simulation state for coherent, parallel data access, employs both intra-environment (across agents, objects, events) and cross-environment parallelism, and groups computations to increase instruction stream coherence. To our knowledge, we are the first to present a detailed description of mapping the ECS architecture fully onto the GPU.
- We demonstrate the flexibility and performance of Madrona by using it to implement several learning environments. By scheduling all aspects of environment simulation onto the GPU, we demonstrate performance that is two to three orders of magnitude higher than existing open source CPU baselines and 5-33× faster than very strong parallel CPU baselines. For example, we demonstrate a port of the hide and seek multi-agent learning environment (with rigid body physics and LIDAR ray tracing) [Baker et al. 2020] running at over 1.9M steps per second on a single GPU.

Madrona is available open source at madrona-engine.github.io.

2 REQUIREMENTS AND GOALS

To motivate the requirements of RL environment simulation, consider OpenAI's hide and seek [Baker et al. 2020], a video-game-like environment used for learning teamwork and tool use skills. Hide and seek is a competitive multi-agent game where agents on the "hider" team cooperate to stay out of sight of agents on the "seeker" team. Play occurs within a 3D environment containing rooms where agents can hide and perform actions on objects (like moving, stacking, or locking) in order to impede the other team. Learning from billions of frames of play in randomized hide and seek environments leads to the emergence of creative team play strategies such as using objects as tools to block doors, form fortresses that keep seekers away, or use as ladders to climb over walls (for more detail see [Baker et al. 2020]).

Implementing hide and seek involves providing code for three parts of the end-to-end training system that not only define the environment simulation itself, but also logic that computes metadata necessary for learning (policy observations, rewards) as well as procedural generation of new environment configurations.

Environment simulation. Simulating the hide and seek environment requires basic rigid body dynamics and collision detection between agents, objects, and floor plan walls. Simulating the rules of play also requires maintaining state for which agents are on which team, tracking who is grasping what objects, and whether objects are "locked" so they cannot be used by the opposing team.

Policy input generation. The system must generate agent policy inputs from an environment's state. In hide and seek this involves geometric computations such a ray casting a low-resolution 360-degree depth map around each agent (agents have LIDAR) and computing pairwise visibility for all agents (agents do not receive information about opposing agents they cannot see). In addition to policy inputs, the system must also compute learning rewards for each agent (derived from the pairwise visibility calculations).

Environment instance generation. To train agents in a diverse set of scenarios, hide-and-seek generates a novel environment for each training episode. This involves generating a physically plausible (no interpenetrations) floor plan and the initial starting positions and orientations of agents and manipulable objects. New environments have random numbers of objects and objects of different size.

Environment simulation, policy input generation, and novel environment generation are key aspects of many RL training systems. Of course, the implementation details of these operations are specific to a given learning task. For example, alternative variants of hide and seek introduce new types of objects and new game rules. The Hanabi card game environment we use in our evaluation does not require rigid body simulation, but must simulate a multi-agent, turn-based game. Tasks involving robot navigation in 3D environments may require RGB-D rendered outputs to be fed to the policy as well as reward computations that involve complex geometric computations such as geodesic distance calculations on a navigation mesh. For these reasons, we believe that it is valuable for batched environment simulation frameworks to be programmable by environment creators as well as performant. Specifically, we seek a system that meets the following requirements:

- **Programmable.** To allow for creation of a wide range of novel learning environments, developers must be able to programmatically specify custom simulation environment state and provide their own implementations of task and environment specific logic. Furthermore, to support interactive, modifiable environments, custom logic must be able to dynamically create and destroy environment state, operate on variably sized collections, and perform complex control flow.
- **High performance.** The system must execute user-defined logic efficiently on a high-throughput parallel processor like a GPU. This requires parallelizing as much of the learning environment workload as possible, and structuring parallel execution to encourage instruction and data access coherence. To ensure

that end-to-end performance is not limited by synchronization or data transforms between subsystems (e.g, physics, rendering, learning) of a complex environment simulator, subsystems must be able to efficiently interface and exchange data with each other. High efficiency should be achieved under varying operating conditions, such as different numbers of batched environments or agents per environment.

- **Productive.** Programmers should not need to be parallel programming experts to create new high-efficiency batch simulators. To make it easy to define new environments, the system must provide abstractions for expressing logic using per-world, per-agent, or per-object functions without concern for how they are parallelized. The resulting logic should also be modular and easily composable via well-defined data layouts and calling conventions to allow construction of complex learning environments from combinations of custom and preexisting implementations.

3 RELATED WORK

AI simulators. Many AI toolkits, in particular those targeting embodied AI and robotics applications, repurpose existing computer game engines (e.g., Carla [Dosovitskiy et al. 2017] on Unreal, AI2-Thor [Kolve et al. 2017] on Unity, iGibson [Shen et al. 2021] on PyBullet [Coumans and Bai 2021], OmniGibson [Li et al. 2022] on Omniverse, Habitat [Savva et al. 2019] on Magnum) or simulation libraries (MuJoCo [Todorov et al. 2012]) to simulate complex training environments. In these cases environment creators relied on mature asset pipelines and scripting APIs provided by these engines to create rich interactive environments appropriate for specific learning tasks. However, since these engines are designed to simulate a single environment at a time, parallel execution of multiple environments requires executing many copies of the engine.

Distributing black-box simulator instances. To accelerate training, much work has focused on efficiently scheduling parallel execution of many black-box simulator instances on large numbers of CPU or GPU resources [Espeholt et al. 2020, 2018; Hessel et al. 2021; Horgan et al. 2018; OpenAI 2018; Petrenko et al. 2020; Weng et al. 2022]. These "scale-out" designs provide high throughput and enable large-scale training experiments, but their efficiency is fundamentally limited because they do not modify the simulator itself.

Batched simulation architectures. Batch simulators [Dalton et al. 2020; Freeman et al. 2021; Makoviychuk et al. 2021; Shacklett et al. 2021] execute many environment instances inside a single simulator at once. These designs amortize data storage and computation costs across environment instances and demonstrate significantly greater efficiency when mapping simulations to the fine-grained, wide parallelism of a GPU. However, since implementing batched simulation requires a simulator rewrite, most embodiments are specialized to a specific task domain (Shacklett et al. [2021] for point-goal navigation in 3D environments, Makoviychuk et al. [2021] for 3D physics simulations). While GPU execution engines exist for training agents, such as Lan et al. [2022], these systems simply map many environments to independent GPU threads and don't leverage batch simulation internally. We are unaware of any batched simulator that meets our programmability requirements.

Array-based programming. An alternative approach to meeting our programmability and performance goals is to author batch environment simulators using array-based programming environments like JAX [Bradbury et al. 2018] or PyTorch [Paszke et al. 2019] that compile to accelerated computing platforms like GPUs and TPUs. While array-based programming in Python may seem like it also addresses our productivity goals, implementing complex training environments using state-of-the-art simulation methods requires complex data structures and non-trivial control flow (traversing acceleration structures, collision solvers, state machines, conditional logic in functions). It is cumbersome (and often inefficient) to express these computations using array-based abstractions. As a result, simulators written in array-based systems often eschew complex data structures [Freeman et al. 2021], limiting the complexity of the environments they can efficiently simulate.

High-level GPU programming languages. Many systems seek to improve the accessibility and productivity of GPU programming by allowing CUDA kernels to be written in high-level languages such as Python (e.g. Warp [Macklin 2022] and Numba [Lam et al. 2015]). We view these projects as complementary to our work: given appropriate engineering, logic written in these high-level languages could interface with our framework to combine the productivity benefits of high-level languages with the runtime services provided by our system for batch simulation (support for dynamic creation of simulation state, efficient interfaces between subsystems, etc).

Entity-component-system architecture. Game engine frameworks provide APIs for developers to implement custom game logic and environment state on top of the built-in services of the engine. While many different game engine architectures exist across a range of successful titles, the Entity-Component-System (ECS) architecture has gained recent popularity through implementations in major game engines such as Unity [Johansson 2018] and Unreal Engine [Palermo 2022], as well as a wide array of open-source implementations of the architecture [Caini 2019; Mertens 2020]. The ECS architecture has been shown to provide code organization and flexibility benefits, for example in Blizzard's *Overwatch* [Ford 2017], as well as performance benefits due to data access efficiency and parallelism [Acton 2019].

4 ECS ABSTRACTIONS

Madrona leverages ECS abstractions to provide a programmable, performant, and productive platform for constructing batch simulators. By design, Madrona's programming interfaces for expressing simulator logic are similar to those of ECS implementations in popular game engines [Johansson 2018]. We consider the extension of a well-established game programming paradigm to the domain of batch simulation as a strength of our approach. In this section we describe key ECS concepts by illustrating how hide and seek is expressed in Madrona.

4.1 ECS Concepts

Defining state: entities and components. In ECS programming, a simulation environment's state is organized into collections of *entities*. In hide and seek two important types of entities are `Agent` (agents on the hider or seeker teams) and `Obstacle` (non-agent scene

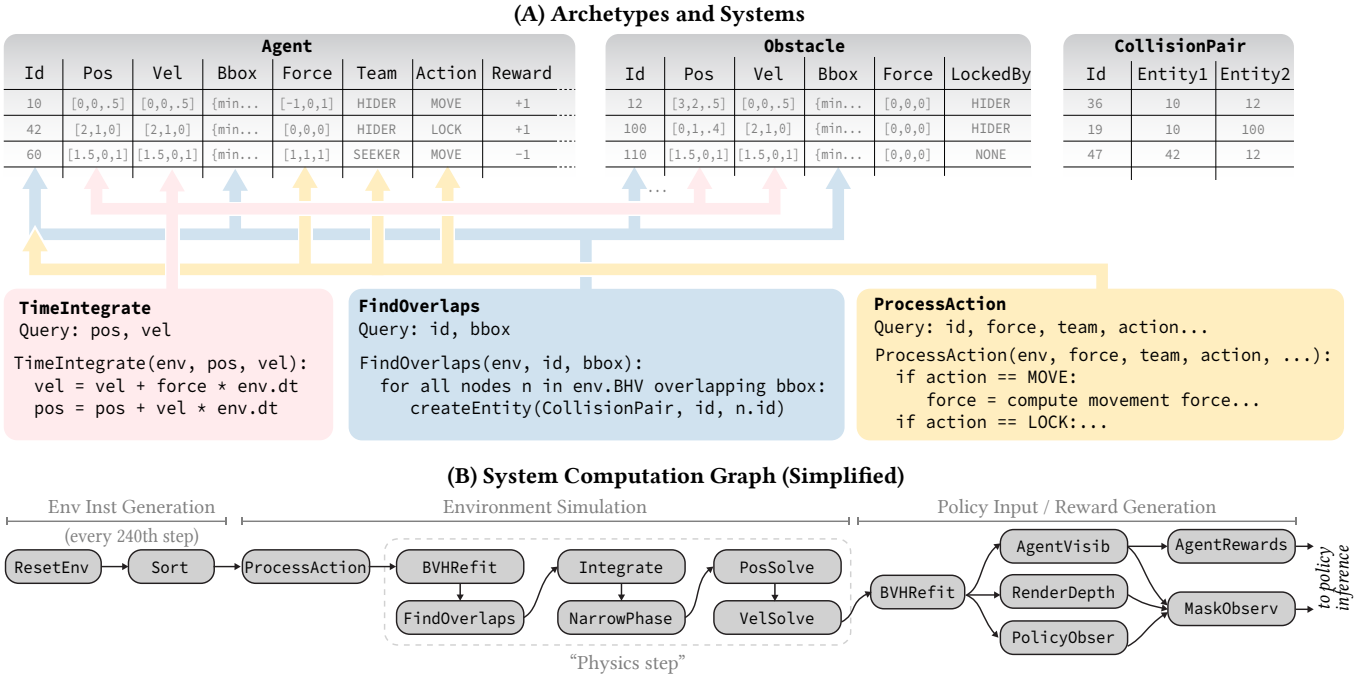

Fig. 1. The HideSeek environment expressed using ECS concepts. (A) Environment simulation logic is organized as a set of systems (`TimeIntegrate`, `FindOverlaps`, `ProcessAction`) that operate on collections of entities. Arrows in the figure indicate the components (data fields) accessed by each system. (B) The system computation graph defines the entire set of computations for a simulation step.

objects). Entities can also represent ephemeral state, such as the interaction details of an agent pushing an obstacle or a potential `CollisionPair` event between two objects (a record of object pairs that require a collision test). Note that entities may have different lifetimes throughout a simulation. For example, `Agent` entities exist for the duration of the simulation, entities representing agent-obstacle interactions may exist for a few simulation time steps, and `CollisionPair` entities are created and destroyed as part of collision detection logic within a single simulation step.

An entity's state is defined by the values of its *components*. For example, an `Agent` will have components such as its position in 3D space, the team it is on, and what action its policy specifies it should perform next. Entities that have the same components are said to share an *archetype*. Figure 1-A illustrates three hide and seek archetypes as well as example component values for three entities of each archetype.

Manipulating state: systems and queries. An environment's state is manipulated by *systems*, data-parallel computation on collections of entities. A system is defined by a *query* that defines what entities are contained in the input collection and a function that is invoked for each entity. Queries select entities that have a specified set of components. In Figure 1-A the `TimeIntegrate` system executes on all entities with `pos` and `vel` components, which is all entities with the `Agent` or `Obstacle` archetypes. The components matched by the query are passed to the system as function arguments (colored arrows indicate component access). The additional argument `env` is a context variable passed to all systems, and serves to provide access to global variables and random access to entities by ID. For example, the `FindOverlaps` system uses `env` to access the environment BVH in order to identify potential collision pairs. Note that executing a system may result in the construction (or deletion) of new entities. For example, `FindOverlaps` creates new `CollisionPair` entities.

The *system computation graph* defines the entire set of computations needed to execute a simulation step. Nodes in the DAG represent systems and edges represent dependencies between systems that must be respected by the ECS runtime. A simplified version of the computation graph for hide and seek is given in 1-B. Observe that the graph involves systems that carry out all three parts of environment processing described in Section 2.

4.2 Putting it All Together: A Madrona Application

Given these concepts, developers can implement a novel batch environment simulator with Madrona as follows.

First, the application defines all required components (name, datatype), as well as all the archetypes that use these components. New ECS systems are then implemented as C++ functions that accept component data for each entity as arguments (the argument types also define the ECS query that matches entities for the system). Since C++ is a general purpose programming language, ECS systems have flexibility in how environment logic is implemented, although a large amount of functionality can be written as straight-line functions that read and write component data on a per-entity basis.

```
HideSeekExperienceCollection:
1   # Initialize Hide and Seek (HS) Batch Simulator
2   components = HSDefineComponents() # pos, vec, reward, ...
3   archetypes = HSDefineArchetypes(components)  # agent, ...
4   graph      = HSBuildComputationGraph(components)
5   envs       = HSInitEnvironments(NUM_ENVS, archetypes)

6   HSBindExternalComponents(envs, Policy.GetActionTensors())
7   observationPtrs = HSGetObservationRewardPtrs(envs)

8   # Experience Collection Loop
9   while !done:
10    Policy.Execute(observationPtrs)
11    MadronaGPUStep(envs, graph)
```

Listing 1. An experience collection loop for hide and seek using a batch simulator built on Madrona. Internally, the *HS** functions use Madrona's ECS interfaces to define simulation state and the computation graph. Within the loop, Madrona advances the batch of environments (*envs*) by one step, assuming control of entity state management and parallel execution of simulator logic. After the step has completed, Madrona returns control to the outer training application that can implement policy inference (red) and policy optimization (not shown) using standard learning frameworks capable of interfacing directly with ECS state through tensor interfaces.

Additionally, Madrona provides C++ APIs for standard ECS operations such as fetching component data based on entity ID, creating and deleting entities, and building the system computation graph. Note that the use of C++ should be considered an implementation detail, bindings could be written to Madrona's ECS APIs and data layouts for any language with a suitable GPU compiler. Throughout the paper, figure and listings use python-esque pseudocode rather than C++ for brevity.

Once the core batch simulator is implemented with the ECS, Madrona enables easy integration with existing training systems as shown in Listing 1. During initialization, the training system first calls into application code that initializes the ECS (lines 2-3), builds the system computation graph (line 4), and instructs the Madrona runtime to instantiate a given number of unique environments to simulate concurrently (line 5). To facilitate efficient communication between training system code and the batch simulator, Madrona provides APIs that allow the application to expose ECS component data as tensors for use in standard learning frameworks. For hide and seek, the training system binds pointers to policy action tensors as input to the simulator (line 6) that become accessible as regular components (e.g. `Action` in Figure 1), and observations and rewards (from agent components, e.g. `Reward`) are exported from the simulator for input to the policy (line 7).

Finally, experience collection proceeds by iteratively executing the application's policy inference code (line 10) and stepping the batch simulation across all environments (line 11). Importantly, Madrona only imposes ECS structure on the operations the application decides to place in the system computation graph. The application retains flexibility to implement its own training loop, define how policies are evaluated to produce new actions, and how learning uses observations and rewards to optimize the policy.

5 MAPPING A BATCH ECS ONTO THE GPU

ECS abstractions impose structure on custom application logic and state that allows the Madrona ECS runtime to efficiently map batch simulation workloads onto the GPU. Through components, archetypes, and system queries, applications provide Madrona with definitions of custom state cleanly decomposed from the logic modifying that state, allowing the runtime to internally manage storage and data flow. Through the system computation graph, the runtime understands that all environments will execute the same systems during simulation step, allowing safe control over parallel execution.

This explicit knowledge about data types and execution makes high performance simulation possible by giving Madrona a large degree of control over how the user's logic is mapped onto the GPU. First, archetypes (and their components) are instantiated as column stores in GPU memory. Next, Madrona gathers together the user-provided functions that implement each ECS system and compiles them, together with wrapper code to manage component access, into a single GPU megakernel [Parker et al. 2010]. Finally, the system computation graph is loaded onto the GPU and executed by the megakernel for each simulation step. The following subsections describe these steps in more detail.

5.1 Managing Component Storage

Centralized table storage. Madrona centrally manages storage for all component data in a batch simulation. Specifically, for every archetype declared by the application, Madrona creates a single in-memory table to store component data for the entities of that archetype *across all environments*. To allow efficient access to consecutive components, these tables are column stores with component data stored contiguously in memory, following standard ECS practice. Since systems may access per-environment state when processing entities (such as the `FindOverlaps` system in Figure 1-A that requires the per-environment BVH data), Madrona adds an implicit `EnvID` component to every table (Figure 2, left table). `EnvID` allows Madrona to quickly look up the per-environment data for every entity and provide it to ECS systems as required.

Madrona's single-table-per-archetype storage scheme has three significant performance benefits. First, when executing systems, neighboring GPU threads processing consecutive entities in a table maintain coherent access to component data even when those entities belong to different environments. If Madrona maintained separate tables for each environment, incoherent data access would occur when only a small number of entities per environment match an ECS query, because neighboring threads in a GPU warp would require data stored across *different tables*. For example, consider the `ProcessAction` system for hide and seek from Figure 1-A. If environments contained only two agents, per-environment tables would result in only two threads per warp accessing data in the same table.

The second benefit of Madrona's single-table-per-archetype design is that it reduces memory footprint when executing large batches of environments. Since entities can be dynamically created, over-allocation of table storage (to accommodate efficient addition of new entities) is amortized across all environments. An implementation that maintained separate tables for each environment would suffer from internal memory fragmentation that scales with

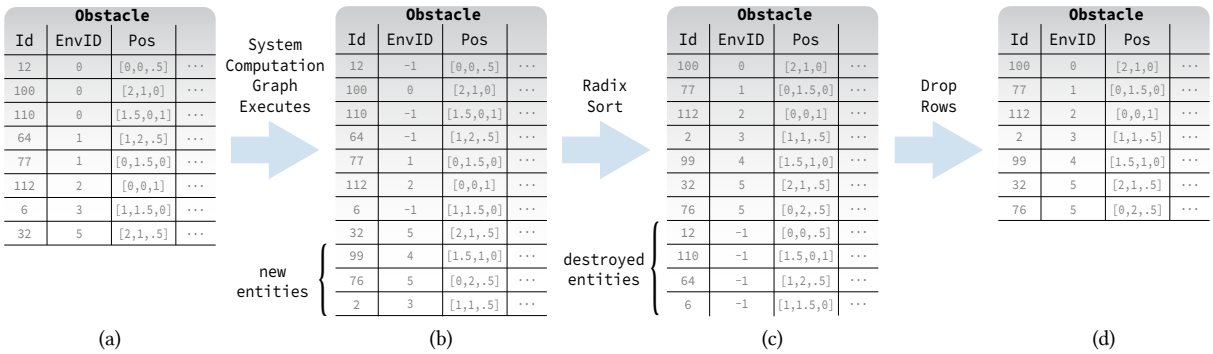

Fig. 2. Madrona uses an implicit `EnvID` component in every archetype to map entities back to the environment they belong to (a). This column is also used as a sentinel value to mark deleted rows (b). Deleted rows can be efficiently reclaimed by sorting the table by `EnvID` (c), then dropping the trailing deleted rows (d).

the number of environments simulated concurrently, reducing the maximum number of environments simultaneously resident in GPU memory and therefore GPU utilization (we observe these costs in Section 6.2). Madrona also leverages the unified table design to amortize the costs of ECS metadata. For example, query objects that store table and column IDs for every matching archetype in the system need only be stored once across all environments.

The third benefit of centralizing storage across all environments is support for zero-copy communication with subsystems external to the ECS, in particular learning frameworks that use batched tensor interfaces. Madrona allows the application to expose ECS component data (for example the `Reward` component of the `Agent` archetype) as tensors that alias the same GPU memory used by the archetype's column store. This aliasing allows learning code and ECS logic to both directly read and write the same component data without requiring copies. Additionally, the resulting tensor is automatically batched across all environments as a consequence of the single-table-per-archetype design.

Dynamic entity creation and destruction. Since entities can be dynamically created and destroyed at runtime (including at fine temporal frequencies), Madrona requires an efficient dynamic memory allocation strategy. For an ECS, creating entities for a given archetype simply requires adding a new row in the corresponding table (Figure 2, b). Madrona grows tables by allocating large regions of virtual memory at startup and paging in physical memory on demand (leveraging support originally added for virtual texturing systems [Obert et al. 2012]). This bypasses the need for a custom GPU allocator, and maintains contiguous storage in virtual address space. Contiguous storage is necessary to export ECS component data as tensors for learning frameworks without performing copies.

Madrona also leverages the implicit `EnvID` component to support efficient parallel entity deletion. When an entity is deleted, the `EnvID` component is set to a sentinel value (-1) that represents a deleted entity (Figure 2, b). This implementation requires no synchronization between deleting threads. Many existing CPU ECS implementations reclaim deleted table rows incrementally using small chunk allocation schemes (return chunks when all rows are free). However since our goal is maximum environment stepping throughput, not predictable per-step latency, Madrona takes a simpler, GPU-friendly, approach to reclamation analogous to garbage collection. Specifically, Madrona periodically compacts sparse tables using a parallel radix sort on the `EnvID` column to sort all table rows with `EnvID` of -1 to the end of the table (Figure 2, c). After sorting, the table can be trivially resized to cut off the -1 entries, reclaiming memory.

Sorting by `EnvID` also provides a performance benefit: improved data coherence when accessing per-environment state. When tables are sorted by `EnvID`, threads working on neighboring entities will access the same shared environment state, leading to coherent and cache friendly accesses. While sorting immediately is not necessary for correctness, empirical testing shows that sorting at the beginning of every step (conditioned on row addition or removal for each table) provides the best performance due to the low cost of radix sort and the data coherence benefits.

5.2 Mapping Systems onto the GPU

As described in Section 4.2, Madrona ECS systems are implemented in C++. At a high level, Madrona compiles these systems onto the GPU by leveraging the NVIDIA CUDA C++ compiler to directly map each system invocation onto a single GPU thread using the CUDA SIMT programming model. To allow Madrona to manage parallelization of entity updates and dataflow throughout the system computation graph, users write ECS systems at a level of abstraction above standard CUDA kernels. Concretely, ECS systems are implemented as functions that receive a single entity's components as arguments, which the user's code modifies to update the entity (see Figure 1-A pseudocode). When the application adds an ECS system to the system computation graph, Madrona uses the provided ECS query to find all matching archetypes and the corresponding column indices for each accessed component. Using this information, Madrona generates code that manages component access and calls the user's ECS system function across multiple GPU threads for each matching entity.

For example in Figure 1-A, the `FindOverlaps` system will match column 0 and column 3 of both the `Agent` and `Obstacles` table. Using this information, Madrona generates a *system entry function* that maps GPU threads to table row indices and passes the fetched component data into the user's ECS system function. Listing 2 shows simplified pseudocode for the system entry function of the `FindOverlaps` system. Given a table with N rows, Madrona will *invoke* the system entry

```
FindOverlapsEntry(invocationID, envs, envIDColumn,
                  entityIDColumn, bboxColumn):
1   envID = envIDColumn[invocationID]
2   if envID == -1:
3       return

4   env = envs[envID]

5   entityID = entityIDColumn[invocationID]
6   bbox     = bboxColumn[invocationID]
7   FindOverlaps(env, entityID, bbox)
```

Listing 2. An example of the entry function for the `FindOverlaps` system. Madrona controls how GPU threads map to rows in the ECS table by varying `invocationID` across threads. The linear column stores for component data trivially allow the entry function to fetch the appropriate row data for each thread in constant time, and the `EnvID` column ensures the correct per-environment data is passed to each system invocation.

function N times, where each invocation results in a single GPU thread executing the user's ECS system function (`invocationID` is incremented so each thread maps to a unique row). In the case of the `FindOverlaps` system, the query matches N rows in the `Agent` table and M rows in the `Obstacle` table, so Madrona will invoke the entry function $N + M$ times. Madrona provides the first N invocations column pointers for the `Agent` table and provides the subsequent M invocations column pointers for the `Obstacle` table.

The system entry function also manages passing per-environment global state into the application's system function using the `EnvID` component that Madrona adds to each archetype. To ensure application code does not run on entities that have been deleted, if the `EnvID` is -1, the row is automatically skipped and the thread simply returns early without executing the user's ECS system function.

For the vast majority of systems in the hide and seek task, Madrona's default parallelization strategy for ECS systems (each GPU thread invokes the system entry function to process one entity) is sufficient. However some systems may be able to achieve increased GPU utilization with advanced parallelization strategies that use warp-level parallelism or on-chip shared memory. Therefore Madrona allows the application to modify the behavior of the system entry function in two key ways: to increase the number of rows processed per invocation and to increase the number of threads dispatched to each invocation to either a warp (32 threads) or a full thread block (256 threads in our implementation). For example, our hide and seek implementation uses this feature to optimize narrow phase collision detection by processing 32 `CollisionPair` entities cooperatively across a full warp of GPU threads. Convex hull data is loaded into on-chip shared memory and then inner loops over convex hull features are cooperatively parallelized across the warp, while the final serial step of contact creation is parallelized across the 32 loaded pairs. Thread-block-level parallelism is also used internally by Madrona to efficiently sort tables (the radix sort stages chunks of the `EnvID` column into shared memory for partial sorting). Exploring extensions to ECS APIs that can expose this kind of fine-grained parallelism and advanced memory hierarchy usage in higher-level interfaces is an interesting area for future work.

5.3 Megakernel-Based Computation Graph Execution

Given how data is stored and how a single system is executed, the final major remaining detail of Madrona's implementation is how the full system computation graph is executed. The naive solution for executing the graph is to launch each system as a separate CUDA kernel; however, this approach has two major issues. First, even moderately complex tasks like hide and seek are composed of a large number of ECS systems (many of which are inexpensive entity updates), so kernel launch overheads would significantly reduce GPU utilization. Second, certain systems process a dynamic number of entities each step, so the number of GPU threads needed for the launch is not known ahead of time. For example, the `NarrowPhase` system will be invoked for every `CollisionPair` created by the `FindOverlaps` system, which can only be known *after* `FindOverlaps` has executed just one system prior in the computation graph. For dynamic systems like `Narrowphase`, a CPU readback step would need to execute before the CUDA kernel launch to determine the number of entities matching the system's ECS query. For computation graphs that contain many low-cost, dynamic systems, the overhead of frequent CPU synchronization would severely limit performance.

Therefore rather than separate kernel launches, Madrona opts to use a GPU-driven approach with a *megakernel* design [Parker et al. 2010]. Madrona compiles all systems in the computation graph together into a single CUDA kernel that is launched once by the CPU per batch environment step and *completes the entire computation graph before returning to the CPU*. To manage both the execution of application-defined systems and engine-level operations that need to be performed, such as evaluating ECS queries and sorting tables, Madrona builds a task graph that describes the full execution of each batch simulation step. Madrona uses a straightforward task graph scheduling policy where all GPU threads work on the same node together before advancing to the next node in the task graph. While this strategy could lead to poor GPU utilization if not enough work is available per node, large-batch simulation of many environments typically yields large amounts of work per node. Beyond simplicity, the primary advantage of this approach is that Madrona has full knowledge of the GPU's execution at any point. This means global operations like remapping memory or compacting tables to minimize peak memory usage during a step can be safely performed in between other nodes in the task graph, without concern for race conditions resulting from the application's system code running while Madrona attempts to perform these engine-level operations.

At a low level, the megakernel is implemented using a persistent-threads [Aila and Laine 2009] style design, where warps of threads loop repeatedly, fetching work at warp-level granularity until the node is completed. When a warp finishes the current node (completes the final outstanding invocation for that node), it immediately updates the global task graph state to point to the next node, and all warps across the GPU immediately begin the next task. This allows low-cost operations, such as evaluating ECS queries to find the number of matching rows (a necessary step before each system can begin executing), to be inserted as nodes in the task graph with low overhead. For hide and seek, the end result is a task graph with

around 200 total nodes; refer to Section 6.6 for a performance breakdown of a single evaluation of the task graph, which demonstrates the low overhead of the megakernel scheduling approach.

5.4 Optimizing GPU Utilization via Up-Front Profiling

A consequence of the megakernel design is that all ECS systems share the same per-thread register allocation. When task graphs contain systems with different register footprint and memory latency constraints, using a single register allocation strategy for the entire task graph may result in GPU under-utilization due to stalls from insufficient occupancy or register spilling. Madrona addresses this issue by optionally compiling multiple versions of the megakernel with different register allocations and then splitting task graph execution into multiple kernel launches using a profile guided optimization (PGO) strategy.

To determine optimal register allocations for each ECS system, Madrona employs an up-front profiling phase that executes a small number of environment simulation steps with each register allocation while measuring per-system execution times using low-overhead tracing infrastructure built into the task graph. Once profiling completes, Madrona chooses when to change register allocations during task graph execution by comparing the profiled speedups for each ECS system to the estimated cost of additional kernel launches. Overall, the time spent in the profiling phase is negligible and can lead to significantly higher throughput for the millions of steps later in a training run.

6 EVALUATION

We evaluate Madrona's expressivity by implementing batch simulators for several learning environments. To evaluate performance and efficiency, we compare GPU batch simulation using Madrona against parallel (but non-batch simulation) CPU and GPU baselines, as well as against widely used third-party reference implementations.

6.1 Experimental Setup

6.1.1 Environments.

- HideSeek implements a three hider, two seeker game of the hide and seek environment [Baker et al. 2020] described in Section 2. This 3D environment presents multiple granularities of parallelism, dynamic entity creation and deletion, and systems that perform broad- and narrow-phase collision detection, contact solving, ray casting, game-play logic, and non-trivial policy input generation.
- Overcooked is a fully cooperative game used as a benchmark task for Human-AI coordination [Carroll et al. 2019] where multiple chefs navigate a grid world to serve as many dishes as possible within a time limit. This 2D environment features complex agent interactions, customizable layouts, and parallelism over both agents and cells in the grid world. Our implementation uses the standard "lossless state encoding" designed for use by convolutional neural networks.
- Hanabi is a card game where two-to-five agents [Bard et al. 2020] cooperate to win. Hanabi requires reasoning about the motivations of other agents based on observing their actions, and it is considered a state-of-the-art challenge in modern AI. Being just a card game, Hanabi is cheap to simulate, but the best performing Hanabi policy requires tens of billions of environment steps to train [Yu et al. 2022], making high-performance execution critical to research progress.
- Cartpole is a classic RL environment [Barto et al. 1983] where an agent moves a cart to balance an inverted pendulum. It is a simple "hello world" test of implementing a custom environment in Madrona. It is an "embarrassingly parallel" task, with minimal state and no conditional execution.

6.1.2 Configurations. We evaluate Madrona's performance in two ways. First we compare Madrona's throughput against that of existing open-source reference implementations: OpenAI's MuJoCo & Python implementation [Baker et al. 2020] of HideSeek, Carroll et al. [2019]'s Python implementation of Overcooked, DeepMind's C++ implementation of Hanabi, and OpenAI Gym's [Brockman et al. 2016] NumPy implementation of Cartpole. Since notable differences in programming language and simulator algorithms (e.g, physics) exist between our implementations and these baselines, direct comparison is difficult. Therefore, we also implement three Madrona backends that allow us to more directly compare our GPU-accelerated batch simulation performance with other scheduling strategies while running the same ECS system implementations.

- BATCH-ECS-GPU is the GPU-accelerated, batch environment simulation implementation described in Section 5.
- ECS-CPU is a CPU-implementation of Madrona that concurrently executes N environments but *does not* perform batch simulation. Instead, it simulates environments on each of the CPU's T threads using an independent ECS instance per environment. To complete an environment step for all N environments, worker threads grab environments from a work queue and complete the next step for a single environment before moving on to the next environment. ECS-CPU can be viewed as an implementation of the synchronous mode of fast RL scheduling systems like EnvPool [Weng et al. 2022]. However, rather than treat the simulator as a (potentially slow) black box, in ECS-CPU the simulator is single-threaded C++ that benefits from the data access optimizations of ECS-structured code. We use ECS-CPU as a *very strong* baseline for CPU performance.
- ECS-GPU is a GPU-implementation of Madrona that *does not* perform batch simulation. Similarly to ECS-CPU, this implementation maps each environment (and its ECS runtime) to a single CUDA thread. We use ECS-GPU to highlight the importance of batch environment simulation on the GPU compared to naive "one-environment-per-thread" execution.
- REF-CPU represents the open-source reference implementation for each environment. These simulators are single threaded and only simulate one environment. To scale to multi-core CPUs, we run one simulator process per CPU thread and batch results in a central learning process.

We evaluate all configurations on a system with an Intel Core i9 13900K CPU (32 threads) and a NVIDIA GeForce RTX 4090 GPU.

6.2 Peak Simulation Throughput

Table 1 plots the peak throughput (environment steps per second) of all implementations. We compute throughput as the time spent

Table 1. Throughput (environment steps per second) of each system configuration under peak operating conditions. When allowed to run under large batch configurations (HideSeek: 32K, Overcooked: 64K, Hanabi: 128K, Cartpole: 1024K) BATCH-ECS-GPU significantly outperforms strong CPU baselines (ECS-CPU) on a high-core CPU, achieving 11× and 33× speedups on HideSeek and Overcooked respectively. ECS-GPU's comparatively low performance shows that GPU parallelization schemes that do not restructure code and data for coherent execution can yield poor performance.

	Environment Steps/Sec			
Config	HideSeek	Overcooked	Hanabi	Cartpole
BATCH-ECS-GPU	$\mathbf{1.9 \times 10^6}$	$\mathbf{4.0 \times 10^7}$	$\mathbf{2.1 \times 10^7}$	$\mathbf{3.4 \times 10^9}$
ECS-GPU	4.5×10^4	1.1×10^6	4.1×10^6	1.4×10^8
ECS-CPU	1.6×10^5	1.2×10^6	4.1×10^6	2.8×10^7
REF-CPU	6.9×10^3	9.7×10^3	1.3×10^4	1.4×10^5

to execute all work defined in the computation graph. This includes environment generation and generation of policy inputs/rewards, but does not include policy inference. All implementations provide their results in contiguous tensors in GPU memory for consumption by standard learning frameworks. BATCH-ECS-GPU already stores component data in this manner, while the other backends must perform additional data movement. Measurement is performed over 1920 environment steps (eight episodes for HideSeek, including environment resets). Using BATCH-ECS-GPU, peak throughput is observed when running with large batch sizes (HideSeek: 32K, Overcooked: 64K, Hanabi: 128K, Cartpole: 1024K). We evaluate throughput at smaller batch sizes in Section 6.4.

Public reference simulators are not performant. One notable conclusion from Table 1 is that open-source reference implementations of these popular ML environments are not performant. Although the reference implementation of HideSeek uses MuJoCo [Todorov et al. 2012] for physics and Hanabi is written in C++, they still achieve low throughput. (HideSeek's performance is limited by Python code around MuJoCo that simulates game logic and generates observations for learning.) Overall, these reference implementations are 23× (HideSeek) to 320× (Hanabi) *slower* than our strong baselines implemented using the Madrona CPU runtime (ECS-CPU).

HideSeek performance. BATCH-ECS-GPU achieves a peak rate of over 1.9M steps per second when executing HideSeek with a batch of 32K environments, outperforming the strong ECS-CPU baseline by 11× (270× faster than the reference implementation). Preliminary experiments with an end-to-end RL training pipeline indicate that the costs of policy inference and policy optimization during training are approximately balanced with that of BATCH-ECS-GPU environment simulation (54% of runtime is environment generation, simulation, and observation generation). This suggests that the 115 billion environment steps needed to observe emergence of all seven phases of emergent tool use in the HideSeek environment (as described in Baker et al. [2020]) could be executed on a *single RTX 4090 GPU in about 1.5 days*. The first instances of fort building and ramp use would emerge in under four hours. Running OpenAI's open-source environment simulator on the 32-thread 13900K CPU would reach this early learning milestone after 40 days.

Table 2. Throughput and GPU utilization relative to hardware speed of light while simulating HideSeek. Using profiling data to optimize megakernel register allocation (PGO Enabled) provides a 35% overall throughput improvement due to improved compute and memory bandwidth utilization. The naive ECS-GPU backend utilizes the GPU poorly due to instruction divergence and incoherent memory access.

Config	PGO Enabled	Env Steps / Second	Compute Util	Memory Util
BATCH-ECS-GPU	Yes	$\mathbf{1.9 \times 10^6}$	**33%**	**69%**
BATCH-ECS-GPU	No	1.4×10^6	22%	46%
ECS-GPU	No	4.5×10^4	1.9%	1.9%

The poor performance of ECS-GPU (GPU-accelerated but not batched) shows the value of Madrona's batched ECS design. ECS-GPU's peak performance is over 42× slower than BATCH-ECS-GPU (and over 3× slower than ECS-CPU). One reason for this difference is that memory capacity limits ECS-GPU to approximately 8K concurrent environments (utilizing only 8K CUDA threads). ECS-GPU suffers from memory capacity issues due to fragmentation since each environment allocates its own ECS table structures (and reallocates them as entities are created and destroyed). A second reason is that ECS-GPU execution suffers from GPU warp divergence and incoherent memory access because threads execute different ECS systems at the same time. Analysis of GPU performance counters confirms that ECS-GPU realizes only 1.9% of the GPU's peak compute and memory bandwidth (Table 2). By centralizing component storage and graph scheduling across all environments, BATCH-ECS-GPU's design encourages coherent execution and data access, increasing compute and bandwidth utilization to 33% and 69% respectively.

Overcooked, Hanabi and Cartpole performance. BATCH-ECS-GPU can generate experience at 40M (Overcooked, batch size 64K), 21M (Hanabi, batch size 128K) and 3.4B (Cartpole, batch size 1024K) steps per second per GPU. These results demonstrate Madrona's ability to realize extremely high-throughput environment simulation when learning from large batch sizes is possible.

Echoing trends from HideSeek, ECS-GPU executes Overcooked inefficiently due to instruction divergence and its inability to exploit intra-environment parallelism. The smallest gap between BATCH-ECS-GPU and ECS-GPU occurs for Hanabi (5×), where BATCH-ECS-GPU's performance is limited by a lack of intra-environment parallelism and frequent memory accesses to game state, resulting in BATCH-ECS-GPU realizing over 90% of peak memory bandwidth, but only 6% compute utilization. Although Cartpole lacks intra-environment parallelism, dynamic state, or conditional execution, BATCH-ECS-GPU still outperforms ECS-GPU by 24×. This is because ECS-GPU must perform additional data movement to combine the state from each environment's separate ECS data structures into a unified tensor for export outside the simulator. Since environment simulation for Cartpole is cheap, this additional copy adds significant overhead to ECS-GPU execution. BATCH-ECS-GPU maintains state for all environments in a contiguous ECS table by construction, so it does not incur this cost.

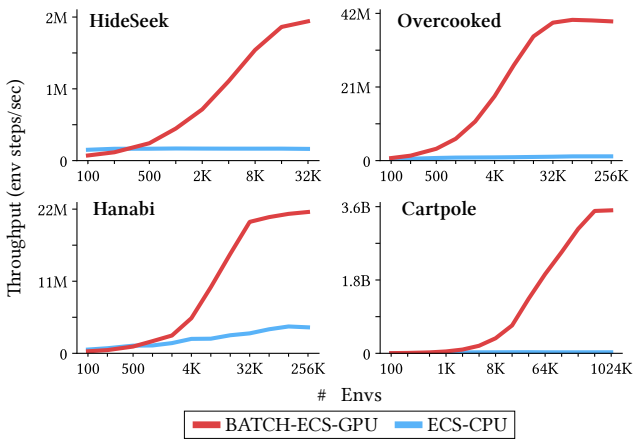

Fig. 3. For HideSeek, BATCH-ECS-GPU maintains >75% of peak throughput down to 8K environments. Hanabi and Overcooked are less computationally intensive, but reach close to peak performance at 32K environments, while Cartpole performance scales logarithmically until 512K environments. The CPU-based runtime (ECS-CPU) can realize near peak throughput when concurrently executing much smaller numbers of environments.

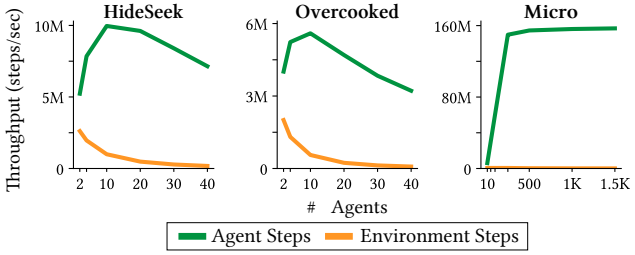

Fig. 4. Madrona exploits intra-environment and cross-environment parallelism, so increasing agents per environment can translate into higher throughput. The HideSeek and Overcooked tasks require work that scales quadratically with the number of agents (computing pairwise observations for learning). As a result, per-agent throughput increases with agent count (as GPU utilization improves) until the quadratic costs dominate. The Micro environment contains no pairwise dependencies, and demonstrates agent update throughput growing linearly with the number of agents until the GPU is fully saturated.

6.3 Impact of Profile Guided Optimization on Utilization

BATCH-ECS-GPU leverages profile guided optimization (PGO) to improve GPU utilization by adjusting register usage and concurrency across the system computation graph (Section 5.4). Table 2 shows that PGO increases BATCH-ECS-GPU's throughput by 500K steps per second, a 35% improvement. With PGO disabled, BATCH-ECS-GPU operates with the maximum register allocation of 255 registers per thread for the entire megakernel. This configuration is the best for a single megakernel due to the register requirements of complex ECS systems such as narrow phase collision detection, but the result is a reduction in the GPU's ability to hide memory access latency for systems with more modest register requirements. Splitting the megakernel into subkernels allows some ECS systems to execute with more active threads and benefit from increased latency hiding. The result of this optimization is shown in Table 2 as an increase in compute utilization (as a percent of GPU speed of light) from 22% to 33% and an increase memory bandwidth utilization (specifically bandwidth to L2 cache) from 46% to 69%.

6.4 Throughput Sensitivity to Batch Size

The BATCH-ECS-GPU peak throughputs reported in Table 1 are obtained from execution with large batch sizes. Large batch sizes expose more parallel work, but incur the cost of higher memory footprint to store environment state and rollouts for more environments. Also, since more experience is collected before updating a policy during training, larger batch sizes may reduce training sample efficiency, diminishing the benefits of fast environment simulation [McCandlish et al. 2018]. To understand the effect of batch size on throughput, Figure 3 plots BATCH-ECS-GPU and ECS-CPU performance at a range of batch sizes.

HideSeek retains >75% of peak throughput down to a batch size of 8K. Performance remains close to peak with only 8K environments because BATCH-ECS-GPU keeps GPU utilization high by exposing coherent parallel work across environments and across entities in each environment. With approximately 30 colliding obstacles in the environment, the number of collision pairs is often a significant multiple of the number of environments. Other expensive computations, such as building agent observations, can be parallelized across the five agent entities, and ray-casting for visibility is parallelized further down to the per-ray level by manually exploiting warp-level parallelism as described in Section 5.2.

Similarly to HideSeek, Overcooked exposes significant intra-environment parallelism and achieves almost 90% of peak throughput running just 16K environments. Hanabi contains no internal parallelism, but reaches a memory bandwidth bottleneck at 32K environments, limiting throughput. Cartpole is so simple that many environments (524K) are needed to saturate the GPU.

BATCH-ECS-GPU begins to outperform ECS-CPU with just 500 environments for HideSeek and 200 environments for Overcooked. For more complex environments with large amounts of internal parallelism, these crossover points may move even lower. These results suggest that GPU-based batch environment simulators can reach high throughput at batch sizes that are modest for modern policy learning algorithms.

6.5 Intra-Environment Throughput Scaling

As discussed in the prior section, Madrona's ability to exploit intra-environment parallelism is critical for maintaining high throughput on the GPU across a range of batch sizes. To understand how performance scales as intra-environment parallel work increases, Figure 4 evaluates performance in environments with increasing numbers of agents. Using modified versions of HideSeek (16K environment batch) and Overcooked (1K environment batch) capable of supporting up to 40 agents, we measure throughput both in terms of environment steps per second and agent steps per second (environment steps per second × # agents). An important note is that the task definition for both environments requires pairwise observations between all agents, causing total work (and storage) to scale quadratically as more agents are added.

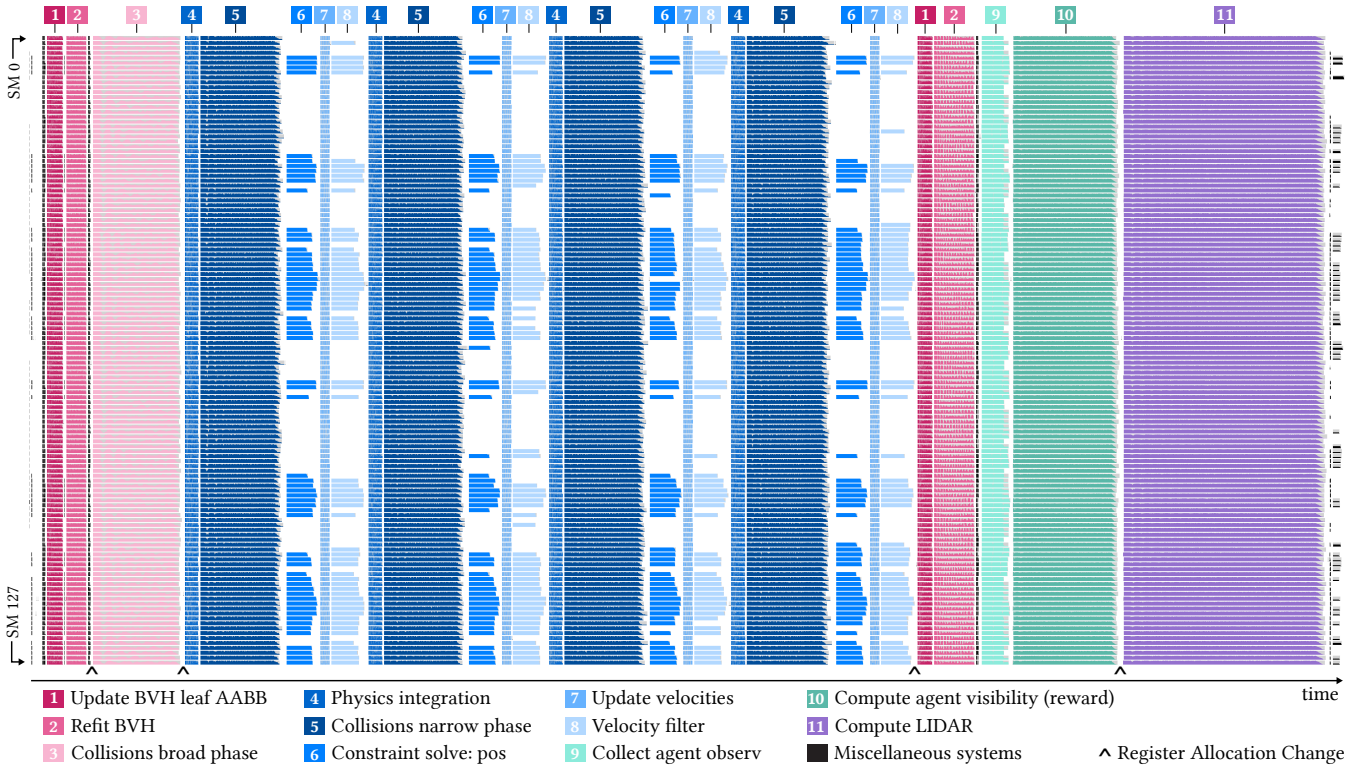

Fig. 5. Profile of one computation graph step of HideSeek (BATCH-ECS-GPU runtime, 16K environment batch). Each horizontal row visualizes the activity of one of the 128 SMs on the GPU over the 7.9 ms time interval required to step the batch of environments. Colors depict the ECS system under execution by each warp. Zooming into each row shows the fraction of warps on the SM assigned work at a given time, represented by the colored portion of the row. For the majority of the computation, most systems process a large number of entities and Madrona successfully achieves good work distribution onto all SMs. Since there is only one invocation of the constraint solver (system 6) and velocity filter (8) per environment (no intra-environment parallelism), some SMs go idle during these periods as there is not enough work to fill the GPU's 30,000+ threads.

Initially, per-agent throughput increases with agent count since GPU utilization increases with additional parallel work. However, when simulating more than 10 agents, per-agent throughput begins to decrease because the cost of generating agent observations is quadratic in the number of agents. Note that for HideSeek, total agent steps per second is still higher with 40 agents than two agents, showing Madrona's ability to accelerate experience collection for multi-agent learning in environments with many agents.

To explore scaling without the impact of quadratic observations, we also include a simple synthetic microbenchmark environment, Micro, that consists of a compute-heavy ECS system that performs work proportional to the number of agents per environment. As expected, Micro underutilizes the GPU when executing 128 environments with low numbers of agents, but per-agent throughput scales linearly as more agents are added until the GPU is fully utilized at 250 agents per environment.

6.6 HideSeek Execution Profile

Figure 5 displays a trace of BATCH-ECS-GPU execution when running one step of HideSeek with batch size 16K. The trace spans 7.9 ms of wall-clock time (measured with the tracing infrastructure used for profile guided optimization, see Section 5.4). Each of the 128 thick rows in the figure corresponds to one streaming multiprocessor (SM) on the RTX 4090. Readers that zoom into the figure will observe that each SM row is further vertically subdivided to show the fraction of total warps per SM executing an ECS system at any given time (half the row will be colored if 50% of the SM's warps are assigned work). The color of each SM row corresponds to the ECS system currently being executed on the GPU. Systems that take only a small fraction of total execution time are accumulated in the "miscellaneous" category and colored black. Note that this visualization provides insight into how well the BATCH-ECS-GPU scheduler utilizes the GPU by assigning ECS system invocations to warps. It does not model SM datapath underutilization due to instruction divergence or memory latency stalls.

Work distribution. Overall, execution exhibits good workload balance onto the GPU. Many of the more costly systems in the computation graph process a large number of entities (intra-environment parallelism), so there is often sufficient parallel work to map to all available GPU threads. Since BATCH-ECS-GPU's scheduler forces the entire GPU to cooperatively process one graph node at a time, having sufficient parallel slackness is also important to reduce effects

of stragglers. Only when executing systems with one invocation per environment, such as the constraint solver (system 6) or velocity filter (system 8), does the computation have insufficient parallelism leading to idle SMs (there are only 16,384 environments and over 30,000 threads across the 128 SMs.)

For systems with large amounts of available parallelism and relatively low register pressure, such as broadphase (system 3), increasing the number of resident GPU threads by reducing the per-thread register allocation can improve performance. Madrona's PGO implementation takes advantage of this observation by splitting the HideSeek megakernel into five sub-sections with register allocations ranging from the default of 255 registers per thread down to 64 registers per thread. In the figure, the black arrows labeled "Register Allocation Change" mark these transition points.

Scheduling efficiency. A key takeaway from Figure 5 is something that cannot be seen in the figure. The full HideSeek ECS task graph is over 200 nodes, yet a vast majority of those nodes fall within the tiny black regions of the figure. Given the low cost of many nodes (e.g, runtime integral nodes that determine the number of invocations to execute or sort tables, application systems like applying agent actions), an alternative implementation that launched a separate kernel for each node would be dominated by launch overheads and CPU-GPU synchronization. The result of Madrona's low-overhead graph scheduling approach is that low-cost nodes essentially disappear from the profile, leaving the timeline dominated by key application tasks such as rigid body physics (systems 4-8) and ray tracing visibility (system 10) and depth (system 11). This contrasts with the HideSeek open source reference implementation which, although built on a high performance physics simulator (MuJoCu), has a performance profile that is dominated by game play and observation generation logic in scripted Python. Similarly, when executing HideSeek with physics and LIDAR disabled on the naive ECS-GPU backend (in other words, only systems 9, 10 and the unlabeled black regions in the figure remain), ECS-GPU still is almost 2× slower than BATCH-ECS-GPU executing the full workload. *It is true that physics and graphics libraries incur the majority of the cost in complex environment simulation, but only if the large body of surrounding logic is implemented efficiently.* Our results suggest ECS abstractions in a performant batched implementation provide an excellent framework for authoring this code and integrating it efficiently with the rest of the engine.

Ray tracing performance. Systems 10 and 11 perform ray casting via a software implementation written in the ECS. In principle, these systems could be significantly accelerated via use of the ray tracing hardware on the RTX 4090. However, current compute-mode GPU programming APIs (unlike graphics APIs) do not allow CUDA kernels to generate ray queries. Additionally, platform BVH-build APIs are only invokable from the host CPU, not from GPU code. Beyond HideSeek, efficient access to raytracing hardware will become increasingly important to support future tasks that require dense visibility information. In general, as more full applications become based on the GPU (as opposed to using the GPU as a co-processor), platforms should aim to to provide full access to hardware capabilities from both the host and the GPU device.

7 DISCUSSION

In this paper we demonstrated that ECS designs are viable solutions for the performance and productivity challenges of developing new batch simulators for training AI agents. We also demonstrated that exceptionally high performance is possible when mapping ECS structured environment simulators to the GPU. Overall we hope that by providing the community with simulator examples that operate in new high-performance regimes, and by making it easier to create new high-performance batch simulators for novel tasks, our work catalyzes progress in creating intelligent AI agents for interactive systems like computer games and robotics applications. We also hope our efforts help community members that do not have access to immense computing resources to tackle state-of-the-art problems in this growing field.

These results are encouraging, but still at an early stage: we have not yet considered the full range of possible system scheduling optimizations. Additionally, in Section 6 we observed how GPU platform evolutions that provide access to the full set of hardware-accelerated capabilities stand to improve our implementations. Looking forward, the complexity of simulated environments will likely grow over time (more detailed geometry, higher-resolution observations, more agents and unique behaviors), making it possible to better utilize high-end GPUs or multi-GPU configurations. Given these trends, we expect the GPU-CPU performance gap on batch environment simulation workloads to continue to rise.

To our knowledge, Madrona is the first ECS architecture implemented fully on the GPU. Although our motivation was to efficiently schedule batch simulators for policy training, our implementation could serve as a design reference for existing ECS implementations considering GPU-acceleration of traditional (single environment) game logic execution as well.

Finally, the need to simulate many interactive environments efficiently is not unique to agent training. For example, the headless server component of traditional multi-player games runs in the datacenter. For economic efficiency, many game server instances are co-located on the same machine (often isolated in virtual machines). Batch-simulation designs targeting a different balance between high throughput and low latency than Madrona may yield significant efficiency gains for multi-player game servers or game streaming services. Overall, efficient batch simulation is an interesting workload that should trigger reevaluation of graphics system design.

ACKNOWLEDGMENTS

This work was supported by gifts from Meta and Activision and by hardware from NVIDIA. The Georgia Tech effort was supported in part by NSF, ONR YIP, and ARO PECASE awards to DB.

REFERENCES

Mike Acton. 2019. Connecting the DOTS: The Unity Data-Oriented Technology Stack. https://www.gdcvault.com/play/1026171/Connecting-the-DOTS-The-Unity. Game Developers Conference.

Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*. 145–149.

Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. 2020. Emergent Tool Use From Multi-Agent Autocurricula. In *8th International Conference on Learning Representations*.

Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain

Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. 2020. The Hanabi challenge: A new frontier for AI research. *Artificial Intelligence* 280 (2020), 103216.

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13, 5 (1983), 834–846.

Joakim Bergdahl, Camilo Gordillo, Konrad Tollmar, and Linus Gisslén. 2020. Augmenting Automated Game Testing with Deep Reinforcement Learning. In *Proceedings of the 2020 IEEE Conference on Games*. 600–603.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:1606.01540

Michele Caini. 2019. ECS Back and Forth (Part 1). https://skypjack.github.io/2019-02-14-ecs-baf-part-1/.

Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. 2019. On the Utility of Learning about Humans for Human-AI Coordination. In *Advances in Neural Information Processing Systems*, Vol. 32.

Erwin Coumans and Yunfei Bai. 2016–2021. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

Steven Dalton, Iuri Frosio, and Michael Garland. 2020. Accelerating Reinforcement Learning through GPU Atari Emulation. In *Advances in Neural Information Processing Systems*, Vol. 33.

Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning (Proceedings of Machine Learning Research, Vol. 78)*. 1–16.

Epic Games. 2022. *Unreal Engine 5*. https://www.unrealengine.com

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. 2020. SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. In *8th International Conference on Learning Representations*.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*. 1407–1416.

Timothy Ford. 2017. Overwatch Gameplay Architecture and Netcode. https://www.gdcvault.com/play/1024001/-Overwatch-Gameplay-Architecture-and. Game Developers Conference.

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. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1.

Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. 2021. Podracer architectures for scalable Reinforcement Learning. (2021). arXiv:2104.06272

Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. In *6th International Conference on Learning Representations*.

Tim Johansson. 2018. Unity at GDC - Job System & Entity Component System. https://www.youtube.com/watch?v=kwnb9Clh2Is. Game Developers Conference.

Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. 2018. Unity: A General Platform for Intelligent Agents. arXiv:1809.02627

Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, Aniruddha Kembhavi, Abhinav Gupta, and Ali Farhadi. 2017. AI2-THOR: An Interactive 3D Environment for Visual AI. arXiv:1712.05474

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Article 7, 6 pages.

Tian Lan, Sunil Srinivasa, Huan Wang, and Stephan Zheng. 2022. WarpDrive: Fast End-to-End Deep Multi-Agent Reinforcement Learning on a GPU. *Journal of Machine Learning Research* 23, 316 (2022), 1–6.

Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, Mona Anvari, Minjune Hwang, Manasi Sharma, Arman Aydin, Dhruva Bansal, Samuel Hunter, Kyu-Young Kim, Alan Lou, Caleb R Matthews, Ivan Villa-Renteria, Jerry Huayang Tang, Claire Tang, Fei Xia, Silvio Savarese, Hyowon Gweon, Karen Liu, Jiajun Wu, and Li Fei-Fei. 2022. BEHAVIOR-1K: A Benchmark for Embodied AI with 1,000 Everyday Activities and Realistic Simulation. In *Proceedings of the 6th Conference on Robot Learning (Proceedings of Machine Learning Research, Vol. 205)*. 80 – 93.

Hung Yu Ling, Fabio Zinno, George Cheng, and Michiel Van De Panne. 2020. Character Controllers Using Motion VAEs. *ACM Trans. Graph.* 39, 4, Article 40 (Aug 2020), 12 pages.

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

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. 2021. Isaac Gym: High Performance GPU Based Physics Simulation For Robot Learning. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An empirical model of large-batch training. arXiv:1812.06162

Sander Mertens. 2020. Flecs 2.0 Is Out! https://ajmmertens.medium.com/flecs-2-0-is-out-1255a2443fbd.

Juraj Obert, J. M. P. van Waveren, and Graham Sellers. 2012. Virtual Texturing in Software and Hardware. In *ACM SIGGRAPH 2012 Courses (SIGGRAPH '12)*. Article 5, 29 pages.

OpenAI. 2018. OpenAI Five. https://blog.openai.com/openai-five/.

OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. arXiv:1912.06680

Mario Palermo. 2022. Large Numbers of Entities with Mass in UE5. https://www.youtube.com/watch?v=f9q8A-9DvPo.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph.* 29, 4, Article 66 (July 2010), 13 pages.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, Vol. 32.

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 Trans. Graph.* 41, 4, Article 94 (July 2022).

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. 2020. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*. 7652–7662.

Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. 2019. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*.

Alessandro Sestini, Linus Gisslén, Joakim Bergdahl, Konrad Tollmar, and Andrew D. Bagdanov. 2022. Automated Gameplay Testing and Validation with Curiosity-Conditioned Proximal Trajectories. *IEEE Transactions on Games* (2022), 1–14.

Brennan Shacklett, Erik Wijmans, Aleksei Petrenko, Manolis Savva, Dhruv Batra, Vladlen Koltun, and Kayvon Fatahalian. 2021. Large Batch Simulation for Deep Reinforcement Learning. In *9th International Conference on Learning Representations*.

Bokui Shen, Fei Xia, Chengshu Li, Roberto Martín-Martín, Linxi Fan, Guanzhi Wang, Claudia Pérez-D'Arpino, Shyamal Buch, Sanjana Srivastava, Lyne Tchapmi, Micael Tchapmi, Kent Vainio, Josiah Wong, Li Fei-Fei, and Silvio Savarese. 2021. iGibson 1.0: A Simulation Environment for Interactive Tasks in Large Realistic Scenes. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 7520–7527.

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*. 5026–5033.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575, 7782 (2019).

Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng YAN. 2022. EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine. In *Advances in Neural Information Processing Systems*, Vol. 35.

Jungdam Won, Deepak Gopinath, and Jessica Hodgins. 2021. Control strategies for physically simulated characters performing two-player competitive sports. *ACM Trans. Graph.* 40, 4, Article 146 (July 2021), 11 pages.

Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. 2022. The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games. In *Advances in Neural Information Processing Systems*, Vol. 35.