QuadSwarm: A Modular Multi-Quadrotor Simulator for Deep Reinforcement Learning with Direct Thrust Control

Zhehui Huang, Sumeet Batra, Tao Chen, Rahul Krupani, Tushar Kumar
Artem Molchanov, Aleksei Petrenko, James A. Preiss, Zhaojing Yang, Gaurav S. Sukhatme

***Abstract*— Reinforcement learning (RL) has shown promise in creating robust policies for robotics tasks. However, contemporary RL algorithms are data-hungry, often requiring billions of environment transitions to train successful policies. This necessitates the use of fast and highly-parallelizable simulators. In addition to speed, such simulators need to model the physics of the robots and their interaction with the environment to a level acceptable for transferring policies learned in simulation to reality. We present QuadSwarm, a fast, reliable simulator for research in single and multi-robot RL for quadrotors that addresses both issues. QuadSwarm, with fast forward-dynamics propagation decoupled from rendering, is designed to be highly parallelizable such that throughput scales linearly with additional compute. It provides multiple components tailored toward multi-robot RL, including diverse training scenarios, and provides domain randomization to facilitate the development and sim2real transfer of multi-quadrotor control policies. Initial experiments suggest that QuadSwarm achieves over 48,500 simulation samples per second (SPS) on a single quadrotor and over 62,000 SPS on eight quadrotors on a 16-core CPU. Code: https://github.com/Zhehui-Huang/quad-swarm-rl**

I. INTRODUCTION

Deep reinforcement learning (RL) has shown promise in developing agile control policies for quadrotors [1]. However, RL algorithms require a large number of environment transitions to train successful policies in simulation. This motivates building fast and highly-parallelizable simulators. Additionally, it is important for the simulator to be good enough that policies trained on it transfer to the real world in spite of unmodeled environment dynamics and the simplified physics assumptions it will inevitably entail.

We describe a simulator, QuadSwarm, to facilitate research in single and multi-robot RL for quadrotors that addresses the aforementioned issues. Specifically, QuadSwarm supports five main ingredients required to enable the development of RL control policies for real quadrotors: (i) A reasonably accurate physics model of a popular existing hardware platform, Crazyflie 2.x, and sufficient domain randomization to account for unmodeled effects; (ii) Supports per-rotor thrust control; (iii) Fast single-threaded throughput, highly parallelizable, and scales with additional compute; (iv) A diverse collection of learning scenarios for single and multi-quadrotor teams; (v) 100% written in Python, which simplifies further development and experimentation.

The authors did this work at the Department of Computer Science, University of Southern California, Los Angeles, CA 90089 USA (e-mail: zhehuihu@usc.edu). GSS holds concurrent appointments as a Professor at USC and as an Amazon Scholar. This paper describes work performed at USC and is not associated with Amazon.

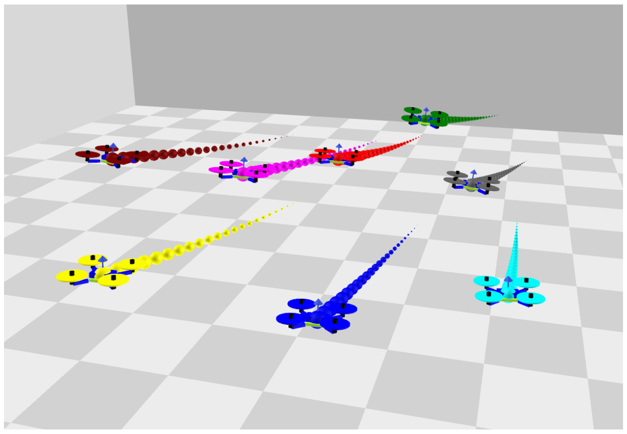

Fig. 1. QuadSwarm visualization: 8 quadrotors heading toward a common goal location

We evaluate the speed of QuadSwarm on a machine with AMD Ryzen 7 2700X CPU (16 CPU cores). QuadSwarm achieves >48,500 simulation samples per second (SPS) in an environment with a single quadrotor and >62,000 SPS in an environment with eight quadrotors, enabling collision simulation. In the environment with eight quadrotors, QuadSwarm receives eight samples per simulation step, which speeds up simulation even though additional computation is required for collision. We have demonstrated zero-shot transferability of RL control policies onto real hardware utilizing QuadSwarm in a single [2] and multi-quadrotor [3] scenarios.

II. RELATED WORK

A. Open-source Simulators that Support Single-robot RL

1) AirSim and Air Learning: AirSim [4] is a photo-realistic simulator for multiple vehicles, such as cars or quadrotors. However, there are three main limitations of using AirSim in RL research. First, AirSim's physics simulation is coupled with rendering, which limits its simulation speed and parallelization ability. Second, although AirSim supports multiple quadrotors, the physical simulation of collisions is overly simplified [8]. This makes AirSim unsuitable for control tasks. Third, AirSim does not provide OpenAI Gym [9] interface for multiple quadrotors. Air Learning [5], based on AirSim, focuses on system-level design to address the challenges of training RL policies and deploying them to resource-constrained quadrotors. Air Learning makes AirSim a better fit for learning by addressing several limitations of AirSim, such as using an environment generator to increase the generalization ability of trained policies. However, Air

TABLE I

FEATURES COMPARING QUADSWARM WITH OTHER SIMULATORS THAT APPLICABLE FOR DEEP RL RESEARCH

Simulator	Physics Dynamics	Rendering	Supports Crazyflie	Per-rotor Thrust	Multi-agent Gym Wrapper	Multi-agent Unified Reward Func	Multi-agent Zero-shot Transfer
AirSim [4]	FastPhysicsEngine	UE4 / Unity	✗	✗	✗	✗	Wait to Verify
Air Learning [5]	FastPhysicsEngine	UE4	✓	✗	✗	✗	Wait to Verify
GymFC [6]	Gazebo	OGRE	✗	✓	✗	✗	Wait to Verify
Flightmare [7]	Ad hoc	Unity	✗	✓	✗	✗	Wait to Verify
gym-pybullet-drones [8]	PyBullet	OpenGL	✓	✓	✓	✗	Wait to Verify
QuadSwarm	Ad hoc	OpenGL	✓	✓	✓	✓	✓

Learning still inherits the three main limitations of AirSim, mentioned above. Different from QuadSwarm, AirSim and Air Learning do not support direct per-rotor thrust control.

2) GymFC: GymFC [6] focuses on tuning flight controllers and developing neuro-flight controllers via RL and supports per-rotor thrust control. While well-suited for developing and tuning single-robot controllers, there is very little support for multi-robot control policies and a lack of a diverse set of training scenarios for multi-robot teams.

3) Flightmare: Flightmare [7] balances simulation speed, photo-realism, and physical accuracy. It supports a large multi-modal sensor suite and supports two control modes: collective thrust and body rates, and per-rotor thrust. However, Flightmare does not directly support multi-robot RL.

B. Open-source Simulators that Support Multi-robot RL

To the best of our knowledge, gym-pybullet-drones [8] is the only multi-drone simulator besides QuadSwarm that facilitates Deep RL research and development of multi-quadrotor teams. Compared with gym-pybullet-drones, QuadSwarm has three main features that gym-pybullet-drones does not have. First, QuadSwarm implements diverse training scenarios and provides a unified reward function for these scenarios, which increases the generalization ability of trained policies. Second, in multi-robot environments, QuadSwarm uses interaction-related rewards, such as the reward when two quadrotors collide with each other. The interaction-related rewards can provide extra information, besides post-collision dynamics, to quadrotors to learn collision avoidance behaviors. Third, QuadSwarm simulates non-ideal motors and sensor noise to decrease the sim2real gap. Besides, in a multi-robot environment with N quadrotors, QuadSwarm uses the relative position and the relative velocity of a fixed number K of nearest robots to represent the neighbor information, where $K \ll N$ when N is large, such as 128, while gym-pybullet-drones uses a boolean distance adjacency matrix $A \in \mathbb{R}^{N\times N}$. Compared with policies trained in gym-pybullet-drones, policies trained in QuadSwarm are thus more easily scalable to larger teams.

III. QUADSWARM

QuadSwarm is a modular quadrotor simulator that supports multiple quadrotors. Figure 2 shows six portable and easy-to-modify modules of the simulator.

A. Quadrotor Dynamics

We use the following quadrotor dynamics [2]:

$$\ddot{x} = g + \frac{\mathbf{R}f}{m} \qquad \dot{\mathbf{R}} = \boldsymbol{\omega}_\times \mathbf{R}$$
$$\dot{\omega} = \mathbf{I}^{-1}(\tau - \omega \times (\mathbf{I}\cdot\omega)) \qquad \tau = \tau_p + \tau_{th}$$

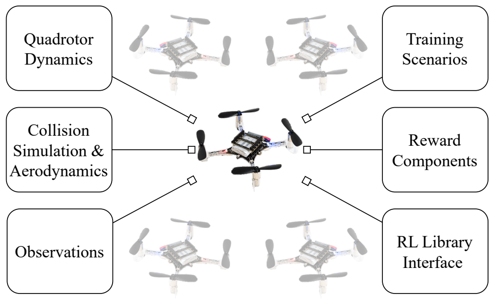

Fig. 2. QuadSwarm Simulator Overview

where $\ddot{x}$ is linear acceleration, g is the gravity vector, $\mathbf{R}$ is the rotation matrix, f is the total thrust force in the body frame, m is the mass, $\boldsymbol{\omega}_\times$ is the skew matrix of the ω, $\mathbf{I}$ is the inertia matrix, τ is the total torque, τ_p is the torque along z-axis, τ_{th} is the torque produced by motor trusts.

The action of quadrotor i is $a_i \in \mathbb{R}^4$, which represents the normalized thrust provided by each motor. Following [2], QuadSwarm models several aspects of real hardware in order to prevent policies from overfitting to the simulator and to facilitate sim2real transfer.

1) Motor Lag: At timestep t, given actions $a^{(t)}$ from a policy sampled from an unconstrained Gaussian distribution, we constrain the actions to be in the range [0, 1] and use this to construct the normalized rotor angular velocity $\hat{u}^{(t)}$.

$$\hat{f}^{(t)} = \frac{1}{2}(\text{CLIP}(a^{(t)}) + 1) \qquad \hat{u}^{(t)} = \sqrt{\hat{f}^{(t)}}$$

We then use a first-order low-pass filter to model motor lag $\hat{u}_f^{(t)} = \alpha_{lag}(\hat{u}^{(t)} - \hat{u}_f^{(t-1)}) + \hat{u}_f^{(t-1)}$, where $\hat{u}_f^{(t)}$ is the filtered rotor angular velocity, and α_{lag} is the motor lag time coefficient, which has been set such that the $\hat{u}_f^{(t)}$ satisfies 2% settling time.

2) Motor Noise: At each timestep, we sample noise from a Gaussian distribution and apply it to the motor noise value produced on the previous timestep such that $\epsilon_f^{(t)} = \alpha_{nd}\epsilon_f^{(t-1)} + \alpha_{ns}\mathcal{N}(0,1)$, where $\epsilon_f^{(t)}$ is the motor noise at timestep t, α_{nd} is the decay ratio of the motor noise, α_{ns} is the scale factor for the motor noise, and $\mathcal{N}(0,1)$ denotes the Gaussian distribution with zero mean and unit variance.

The final thrusts provided by each motor $f^{(t)} \in \mathbb{R}^4$ is constructed by the maximum thrust that each motor can provide f_{max}, the filtered rotor angular velocity $\hat{u}_f^{(t)}$, and the motor noise $\epsilon_f^{(t)}$. Specifically, $f^{(t)} = f_{max} \cdot (\hat{u}_f^{(t)})^2 + \epsilon_f^{(t)}$.

B. Collision Simulation and Aerodynamics

Modeling accurate collisions is important for learning robust collision-avoidance policies but is a non-trivial task. In this section, we outline simple collision models used by default in QuadSwarm that is implemented in a modular way and can easily be swapped with a different collision model. Although these models are simple, in [3], we demonstrated they are good enough to train successful policies.

1) Quadrotor to Quadrotor: When two quadrotors collide, instead of modeling complex interactions, such as whether the propellers of two quadrotors touch, we implement a simple collision model based on the linear velocity and the angular velocity.

$$n_{col} = \frac{x_1 - x_2}{\|x_1 - x_2\|_2} \qquad \tilde{v} = (v_2 \cdot n_{col} - v_1 \cdot n_{col}) \cdot n_{col}$$
$$v_1 \leftarrow \alpha_1(v_1 + \tilde{v} + \epsilon_{v1}) \qquad v_2 \leftarrow \alpha_2(v_2 - \tilde{v} + \epsilon_{v2})$$
$$\omega_1 \leftarrow \omega_1 + \epsilon_{\omega 1} \qquad \omega_2 \leftarrow \omega_2 + \epsilon_{\omega 2}$$

Where x_1, x_2 are the positions of two quadrotors, v_1, v_2 are the linear velocity of two quadrotors, α_1, α_2 are the linear velocity decay factor of two quadrotors, $\epsilon_{v1}, \epsilon_{v2}$ are the linear velocity noise of two quadrotors, and $\epsilon_{\omega 1}, \epsilon_{\omega 2}$ are the angular velocity noise of two quadrotors.

2) Quadrotor to Wall or Ceiling: The collision model between a quadrotor and walls or ceiling is the same as the quadrotor-quadrotor collision model, except that the collision updates are only applied to the quadrotor.

3) Quadrotor to Ground: We consider two situations of quadrotor interaction with the ground. When the quadrotor hits the ground we set the linear velocity, angular velocity, and acceleration to zero, regenerate the rotation matrix by setting the normal vector of the quadrotor upward, and reset all momenta. When the quadrotor is on the floor, and the thrust is not enough to allow the quadrotor to take off, we arrest motion on the floor with sufficiently high friction. When the linear velocity of the quadrotor is 0, the friction direction is opposite to the thrust force direction in the xy plane, and the final force function is: $f_{xy} \leftarrow \max(f_{xy} - \mu(mg - f_z), 0)$. When the linear velocity is bigger than 0, the friction direction is opposite to the velocity direction in the xy plane, and the final force function is: $f_{xy} \leftarrow f_{xy} - \mu(mg - f_z)$. In functions above, f_{xy} is the thrust force in the xy plane, f_z is the thrust force in z axis, μ is the friction coefficient, and g is the gravity constant.

4) Downwash: Our downwash model is a simplified version of the model proposed in [10]. We only model downwash effects when two quadrotors overlap in the xy plane and within a pre-defined distance along the z axis.

$$\ddot{x} = k_1(k_2\delta_{pos} + b_1) + \epsilon_d \qquad \dot{\omega} = \epsilon_{\omega d}$$

Where δ_{pos} is the relative distance between quadrotors, $\dot{\omega}$ is the change rate of angular velocity, which is used to simulate the aerodynamic disturbances, and $k1, k2, b1$ are constants, $\epsilon_d, \epsilon_{\omega d}$ are Gaussian noise.

C. Observations

The observations of quadrotor i are:

$$[\delta_{xi}, v_i, R_i, \omega_i, [\tilde{x_{i1}}, \tilde{v_{i1}}, ..., \tilde{x_{iK}}, \tilde{v_{iK}}]]$$

where δ_{xi} represents the relative position between the quadrotor i and its goal, $\tilde{x_{i1}}, \tilde{v_{i1}}$ represent the relative position and relative velocity to the closest quadrotor, $\tilde{x_{iK}}, \tilde{v_{iK}}$ represent the relative position and relative velocity to the Kth closest quadrotor. K is a hyperparameter. In the single quadrotor environment, K is set to 0.

To increase zero-shot sim-to-real transfer ability, we add sensor noise to the observations [2]:

$$\epsilon_x = U(0, 5e^{-3}) \quad \epsilon_v = U(0, 1e^{-2}) \quad \epsilon_\omega = \mathcal{N}(0, 1.75e^{-4})$$

where U represents the uniform distribution, $\mathcal{N}$ represents the Gaussian distribution, ϵ_x is the position noise, ϵ_v is the linear velocity noise, ϵ_ω is the angular velocity noise.

D. Training Scenarios

To design diverse training scenarios, we use the quadrotor team's goals to construct several geometric formations, including a circle, grid, sphere, cylinder, and cube. We use this pool of geometric formations to design three groups of training scenarios.

1) Static formations: Uniformly sample a geometric formation from the pool and randomly place it in the room.

2) Dynamic formations: Change the positions and/or the geometric formation of goals after a random period of time within an episode. There are four variants:

- Dynamic goals: regenerate the positions and the geometric formation of goals after a random period of time.
- Swap goals: keep the geometric formation but shuffle the positions of goals after a random period of time.
- Shrink & Expand: keep the geometric formation of goals, but change the formation size over time.
- Swarm-vs-Swarm: split quadrotors into two groups, and fix the formation center of each group. After a random period of time, resample the formation shape and swap the goals of the two groups.

3) Evader Pursuit: Quadrotor(s) pursue one moving goal. We parameterize the trajectories in two ways - using a 3D Lissajous curve, and randomly sampled consecutive points connected by Bezier splines, respectively.

E. Reward Components

We provide diverse reward components in the simulator. There are two groups of reward components. One is based on the quadrotor's state, and the other is based on the interactions with other objects. All α below are constants.
Quadrotor State:

$$r^{(t)}_{pos} = \alpha_{pos}\left\|\delta^{(t)}_{xi}\right\|_2 \qquad r^{(t)}_{vel} = \alpha_{vel}\left\|v^{(t)}\right\|_2$$
$$r^{(t)}_{ori} = \alpha_{ori}R^{(t)}_{22} \qquad r^{(t)}_{spin} = \alpha_{spin}\left\|\omega^{(t)}\right\|_2$$
$$r^{(t)}_{act} = \alpha_{act}\left\|f^{(t)}\right\|_2 \qquad r^{(t)}_{\delta act} = \alpha_{\delta act}\left\|f^{(t)} - f^{(t-1)}\right\|_2$$
$$r^{(t)}_{rot} = \alpha_{rot}\frac{tr(R^{(t)}) - 1}{2} \qquad r^{(t)}_{yaw} = \alpha_{yaw}R^{(t)}_{00}$$

where reward components based on the distance to the goal, linear velocity, the normal vector in the z-axis, angular velocity, actions, change of actions, rotation, and yaw.
Interaction with Other Objects: We use a weighted combination of indicator functions for the conditions when the quadrotor hits the floor, stays on the floor, hits a wall, hits

the ceiling, or hits other quadrotors. We also use a weighted combination of the relative distance between quadrotors for the condition when quadrotors are close to each other.

F. Reinforcement Learning Library Interface

We integrate Sample Factory [11], a fast RL library, with QuadSwarm to decrease the wall-clock training time. Sample Factory supports synchronous and asynchronous modes of policy proximal optimization (PPO) algorithms. For multi-agent RL, it currently supports Independent PPO.

IV. Simulation Speed

To balance speed, readability, and flexibility, we decide to: (i) use Python to implement the minimum requirements of physics simulation and rendering, (ii) use Numba [12], a just-in-time compiler that is able to translate Python and NumPy code into machine code to speed up physics simulations, and (iii) decouple rendering from physics simulations.

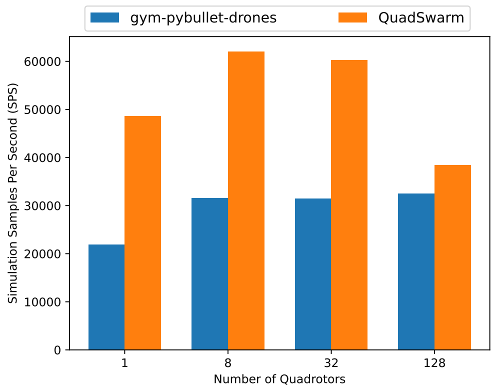

Fig. 3. Simulation Speed: gym-pybullet-drones vs QuadSwarm

We evaluate simulation speed on a machine with AMD Ryzen 7 2700X CPU (16 CPU cores). To fairly compare QuadSwarm with gym-pybullet-drones, we set both simulators with 100 Hz control frequency, 200 Hz simulation frequency, and 15 seconds episode duration time. In an environment with multiple quadrotors, each quadrotor has the same observation space, thus QuadSwarm receives multiple samples per simulation step.

Fig. 3 shows the simulation speed comparison between gym-pybullet-drones and QuadSwarm. In a single quadrotor setting, QuadSwarm approaches 48,589 SPS - $\sim$2.2x faster than gym-pybullet-drones. With multiple quadrotors and collision simulation, QuadSwarm approaches the fastest simulation speed, 62,042 SPS, when the number of quadrotors is eight - $\sim$2.0x faster than gym-pybullet-drones.

V. Examples

We used QuadSwarm as the main simulation platform in two projects that demonstrated the transfer of learned control policies on single and multiple quadrotors. For a single quadrotor [2], we show how to learn a policy to stabilize multiple different quadrotors with domain randomization. For multiple quadrotors [3], we show how to learn a policy to control up to 128 quadrotors to approach their goals while avoiding collisions in diverse scenarios.

VI. Conclusions

We describe QuadSwarm, a simulator for Deep RL research on single and multi-quadrotor control policies and their sim2real transfer to real hardware. We demonstrate how QuadSwarm integrates five key ingredients: (i) a reasonable physics model of Crazyflie 2.x, with domain randomization to account for unmodeled effects; (ii) per-rotor thrust control; (iii) fast, high parallelization, and scaling with additional compute; (iv) a diverse collection of learning scenarios for single and multi-quadrotor teams; (v) 100% written in Python. Our experiments suggest that QuadSwarm can be used to create robust quadrotor policies that successfully deploy to real hardware and that it is a useful and promising tool that will accelerate research in robust single and multi-quadrotor control policies for agile flight. We are working on extending QuadSwarm to support multiple obstacles, providing more accurate aerodynamic effects, and integrating with additional Deep RL libraries, such as PyMARL2 [13].

References

[1] Y. Xie, M. Lu, R. Peng, and P. Lu, "Learning agile flights through narrow gaps with varying angles using onboard sensing," *arXiv preprint arXiv:2302.11233*, 2023.

[2] A. Molchanov, T. Chen, W. Hönig, J. A. Preiss, N. Ayanian, and G. S. Sukhatme, "Sim-to-(multi)-real: Transfer of low-level robust control policies to multiple quadrotors," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 59–66.

[3] S. Batra, Z. Huang, A. Petrenko, T. Kumar, A. Molchanov, and G. S. Sukhatme, "Decentralized control of quadrotor swarms with end-to-end deep reinforcement learning," in *Conference on Robot Learning*. PMLR, 2022, pp. 576–586.

[4] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[5] S. Krishnan, B. Boroujerdian, W. Fu, A. Faust, and V. J. Reddi, "Air learning: a deep reinforcement learning gym for autonomous aerial robot visual navigation," *Machine Learning*, vol. 110, pp. 2501–2540, 2021.

[6] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for uav attitude control," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 2, pp. 1–21, 2019.

[7] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Conference on Robot Learning*. PMLR, 2021, pp. 1147–1157.

[8] J. Panerati, H. Zheng, S. Zhou, J. Xu, A. Prorok, and A. P. Schoellig, "Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 7512–7519.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[10] G. Shi, W. Hönig, Y. Yue, and S.-J. Chung, "Neural-swarm: Decentralized close-proximity multirotor control using learned interactions," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 3241–3247.

[11] A. Petrenko, Z. Huang, T. Kumar, G. Sukhatme, and V. Koltun, "Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7652–7662.

[12] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[13] J. Hu, S. Jiang, S. A. Harding, H. Wu, and S.-w. Liao, "Rethinking the implementation tricks and monotonicity constraint in cooperative multi-agent reinforcement learning," *arXiv preprint arXiv:2102.03479*, 2021.