VPR-Gym: A Platform for Exploring AI Techniques in FPGA Placement Optimization

Ruichen Chen[1], Shengyao Lu[1], Mohamed A. Elgammal[2], Peter Chun[1], Vaughn Betz[2], and Di Niu[1]

[1]*Dept. of Electrical & Computer Engineering, University of Alberta, Canada*
[2]*Dept. of Electrical & Computer Engineering, University of Toronto, Canada*

***Abstract*—With the increasing complexity and capacity of modern Field-Programmable Gate Arrays (FPGAs), there is a growing demand for efficient FPGA computer-aided design (CAD) tools, particularly in the placement stage. While some previous works, such as *RLPlace*, have explored the efficacy of single-state Reinforcement Learning (RL) to optimize FPGA placement by framing it as a multi-armed bandit (MAB) problem, numerous AI techniques remain unexplored due to the outstanding engineering challenges to integrate them into the FPGA CAD flow which is based on C++. In this paper, we present VPR-Gym, a Python environment built on OpenAI Gym, that allows seamless integration with various machine learning libraries including PyTorch, TensorFlow, and Nevergrad while enabling the comparison between different AI techniques for FPGA placement. Moreover, we introduce a learning objective that reformulates the FPGA placement task as an optimization problem, thereby expanding the range of AI techniques that can be investigated beyond those for MAB problems. To showcase the capabilities of our platform, we conduct experiments comparing the performance of various MAB algorithms and evolution strategy (ES) algorithms. Our findings demonstrate that the ES approaches exhibit superior performance over the existing MAB approaches, highlighting the effectiveness of VPR-Gym in facilitating AI research to enhance FPGA placement.**

***Index Terms*—Reinforcement Learning, FPGA, Placement, VPR, OpenAI Gym**

I. Introduction

The increasing capacity and architectural complexity of modern Field-Programmable Gate Arrays (FPGAs) have led to a growing demand for efficient FPGA computer-aided design (CAD) tools. Among the various stages in the CAD flow, placement is the most time-consuming, while its outcome significantly impacts the runtime of the subsequent routing stage [1]. Improving the runtime of simulated annealing (SA) is vital for enhancing the overall efficiency of traditional FPGA CAD tools during the placement phase, given that SA accounts for a significant portion of the runtime in this stage [2].

Some researchers reformulate SA into a multi-armed bandit (MAB) problem [3]–[5], allowing the annealer to schedule directed moves [6] or choose block types rather than randomly swapping blocks, which can significantly improve the runtime without sacrificing Quality of Result (QoR). However, a wide variety of other AI techniques remain unexplored due to the difficulty to directly integrate them into the FPGA CAD flow based on C++, creating a significant obstacle for AI researchers who seek to contribute to the field without specialized knowledge in FPGA CAD.

To reduce the entry barrier, we propose and develop VPR-Gym, a Python environment built on top of the OpenAI Gym framework for SA-based FPGA placement. Our platform enables seamless integration with several Python-based machine learning libraries, including PyTorch [7], TensorFlow [8], and Nevergrad [9], which allows researchers to focus on high-level algorithm design and reduces the engineering efforts required for transplanting ML libraries from Python to C++.

Moreover, our research introduces a learning objective that formulates the FPGA placement task into an optimization problem, expanding the range of AI techniques that can be explored beyond algorithms for the MAB problem. To demonstrate the capability of our platform, we conducted experiments on VPR-Gym, comparing the performance of various MAB algorithms as well as different algorithms that belong to evolution strategy (ES). Note that ES is one of the AI techniques that has not been previously utilized for FPGA placement. Our experiments reveal that the ES approaches exhibit superior performance compared to the existing MAB approaches, demonstrating the potential of VPR-Gym to facilitate AI research for FPGA Placement and enhance its performance. The project source is open and available at https://github.com/cccrrrccc/VPR-Gym.

II. Background on FPGA Placer

Versatile Place and Route (VPR), which is part of the FPGA CAD design flow Verilog-to-routing (VTR) 8 [10], applies Simulated Annealing (SA) to achieve high QoR in placement. SA randomly performs perturbations that move and swap the blocks on an FPGA board to optimize the placement. *RLPlace*, as illustrated in Fig. 1, is an SA-based FPGA Reinforcement Learning (RL) placer on VPR by Elgammal et al. [3]. Directed moves, e.g., Bounded Median Move, Critical Random Move, etc. [6], are more likely to produce useful perturbations compared with random moves. Six different types of directed moves are added to the standard VPR 8 SA placer and can benefit QoR in different ways, e.g., in terms of wire length, critical path delay (CPD), or both. Moreover, their effectiveness depends on the operating netlist, current placement progress, targeted architecture, etc. Since manually scheduling directed moves that can balance all these factors is challenging, a multi-armed bandit (MAB) agent is designed to learn and select the most effective directed move type for optimizing the SA placement process [3]–[5]. Specifically, for

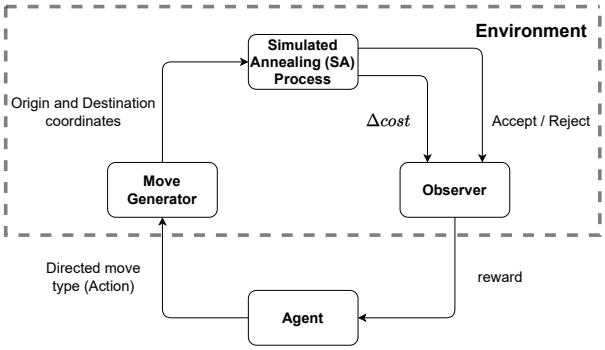

Fig. 1. *RLPlace* mechanism.

every single step:

- The agent chooses a directed move type as an action;
- The move generator decides the origin and destination of the next move;
- SA process swaps the logical blocks at the origin and destination and returns the change in cost and whether to accept the swap;
- The observer sends a reward to the agent based on SA's outcome.

Murray et al. [5] propose another way to enhance the VPR by defining logical block types as the action space and performing RL on it, which also outperforms the standard VPR 8 placer. Similarly, the agent is designed to solve a MAB problem, except that the available actions are logical block types instead of directed move types.

However, there are several challenges to investigating AI techniques for FPGA placement:

- **Applying latest AI methods**: since VPR is written in C++, it is hard to apply popular state-of-the-art AI techniques to FPGA placement, as most of these techniques are developed in Python language.
- **Algorithm-Environment Coupling**: Existing FPGA RL placers [3], [11] are all highly coupled with the rest parts of VPR, serving as one phase of VPR's FPGA CAD workflow. This means employing a new algorithm requires a deep understanding of VPR, and the new agent must be written inside the source code of VPR. Besides, every improvement to the agent learning code requires a recompilation of the entire VPR project, which is time-consuming.
- **Algorithm Limitation**: Existing FPGA RL placers [3], [11] only support a reinforcement learning formulation, specifically the Multi-armed Bandit (MAB) problem, maximizing the long-term rewards during the simulated annealing process, which are assumed to be stationary. However, *RLPlace* has investigated its agent behavior on different FPGA designs and observed that the optimal directed move type is changing over time [3].

III. Design of the VPR-Gym

We design and implement VPR-Gym, a novel approach to interact with VPR through the OpenAI Gym framework in Python. As illustrated in Fig. 2, VPR-Gym abstracts the Simulated Annealing (SA) scheduling in VPR away from the agent such that the agent takes actions and receives rewards in steps. The agent learns to adjust its actions intelligently during the SA process in order to maximize some function of the received rewards.

A. Problem Formulation

In VPR-Gym, we first propose a more general learning objective that can accommodate a wider range of AI optimization techniques. Our learning objective converts the current RL (MAB) formulation in FPGA placement, e.g., RLPlace [3], into an optimization problem, which allows us to solve the problem using Evolution Strategies, MAB algorithms, as well as a potentially broad range of other AI optimization algorithms.

We define a parameter vector $\mathbf{Q}$ to control the probability of choosing each directed move type. The length of $\mathbf{Q}$ is determined by the number of available actions. For example, if the agent needs to choose among 7 directed move types, the length of $\mathbf{Q}$ is 7. Then, the probability P_a of each action a is given by

$$P_a = \frac{S(Q_a)}{\sum_{i=1}^{N} S(Q_i)}, \tag{1}$$

where $S(x) = 1/(1 - e^{-x})$ is the Sigmoid activation function over a domain of real values and N is the number of available actions.

The optimization problem is to find an input vector $\mathbf{Q}$ to minimize a loss function $Loss(\mathbf{Q})$, which is defined as

$$Loss(\mathbf{Q}) = -\sum_{i=(t-\tau+1)}^{t} r_i. \tag{2}$$

Minimizing $Loss(\mathbf{Q})$ will maximize the sum of reward during the past τ steps. As the agent maximizes the reward dynamically in moving windows, the overall placement performance is maximized. We will show that this formulation can accommodate multiple algorithms by changing the specific form of action sampling probability P_a and minimizing different losses as functions of step-wise rewards. In fact, by adjusting τ, we can dynamically generate a non-stationary policy to sample different actions as the SA process progresses.

B. System Implementation

The agent takes actions and receives rewards from the VPR environment. The code listing 2 shows an example of how to use VPR-Gym. Line 2-3 shows the initialization of VPR-Gym. We provide two options for the environment **VprEnv** and **VprEnv_blk_type**, depending on whether *block type* will be a part of the action, which will be introduced in detail. Line 5-8 shows how the agent learns using VPR-Gym. At each step, the agent takes an action and passes the action to VPR via function **env.step()**. The function will return a reward and other necessary information from the VPR environment to the agent. Researchers are free to define the agent's behavior. For example, to follow the optimization learning objective,

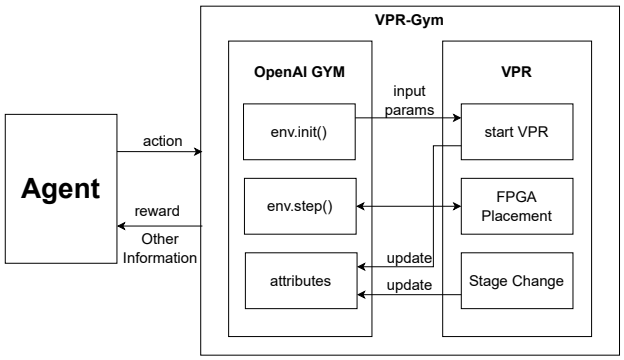

Fig. 2. Architecture for VPR-Gym.

researchers can collect the rewards over any given number of steps and use that to update the agent action probability distribution.

1) Gym Initialization: The initialization of the gym environment starts a VPR kernel under the Gym mode. Under this mode, the VPR program will pause at certain points and communicate with the OpenAI Gym framework via the Zeromq socket.

Users can specify the arguments in the **VprEnv()** constructor to control the setting of VPR. The OpenAI Gym framework uses those arguments to initialize the VPR kernel via the command line interface (CLI). Therefore, any viable simulation settings in the VPR kernel are still viable in the VPR-Gym. Note that our VPR-Gym supports parallel initialization and parallel execution. This, however, requires properly assigning different working directories and port numbers for different tasks to avoid conflicts between multiple environments, which is shown in code listing 1.

```
from src.vprGym import VprEnv, VprEnv_blk_type
env1 = VprEnv(directory='env1', port='5555',
    benchmark='...')
env2 = VprEnv_blk_type(directory='env2', port='6666'
    , benchmark='...')
agent1 = Agent()
agent2 = Agent()
train(env1, agent1)
train(env2, agent2)
```

Listing 1. Example of parallel initialization

2) Action Space: To build our basic environment **VprEnv** in VPR-Gym, we employ the same action space as that in *RLPlace* [3]: there are 7 directed move types, which are Random Move, Bounded Median Move, Bounded Edge-Weighted Median Move, Bounded Centroid Move, Bounded Weighted Centroid Move, Critical Random Move, and Quasi-Bounded Feasible Region Move. Therefore, in VPR-Gym, an action space is defined as a single discrete number from 1 to N, where the size of action space N is also subject to changes because there are two placement stages in VPR, for which the numbers of available directed move types are 4 and 7, respectively.

In VPR-Gym, we also introduce a new custom environment called **VprEnv_blk_type** where the logical blocks' types are added to the action space. The action space of **VprEnv_blk_type** is defined as a 2-tuple, where the first element represents directed move types and the second element represents the block types. Note that for the two directed moves that only happen in the second stage (**Bounded Weighted Centroid Move** and **Critical Random Move**), selecting logical block types is not supported because these directed moves only manipulate blocks with highly critical connections. If there are some block types which do not contain highly critical blocks, this action pair will always lead to an aborted result and become a trivial action.

```
from src.vprGym import VprEnv
import Agent
env = VprEnv()
agent = Agent()
done = False
while (done == False):
    action = agent.action()
    state, reward, done, info = env.step(action)
    agent.getReward(reward)
print('Wire Length:', info['WL'])
print('Critical Path Delay:', info['CPD'])
print('Runtime:', info['RT'])
```

Listing 2. Example of VPR-Gym

3) Reward: VPR-Gym provides more freedom for researchers to design the agent's reward. Three variables are passed from VPR to the OpenAI Gym framework by storing them in the **info** dictionary, including the normalized change of the wire length cost ($\Delta cost$), the normalized change of the timing cost (Δt_{cost}), and the change of the bounding box (related to Wire Length) cost (Δbb_{cost}). All existing reward functions can be calculated from these three parameters. For example, the reward function of WLbiased_runtime_aware [3] is

$$r_t = \begin{cases} \frac{-1}{t(i)}((0.5 + R_s) * \Delta bb_{cost} \\ +(1 - R_s) * \Delta t_{cost}), & if\ \Delta cost < 0 \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

where R_s is a constant empirically set to 0.4. Eq. (3) can be computed locally at the Python end with these three variables. Moreover, in VPR-Gym, the value of R_s becomes an adjustable variable, which can be learned by the agent or tuned by a hyperparameter optimization mechanism.

4) Environment Attributes: Other important information is stored in the attributes of the environment, including:

- num_actions: An integer shows the number of available directed moves.
- num_types: An integer shows the number of logical block types in the targeted FPGA design;
- num_blks: This is a list consisting of the number of blocks for each logical block type category;
- horizon: An integer shows the number of steps within one temperature in the Simulated Annealing (SA) Process. Note that some multi-armed bandit algorithms require the horizon information in order to better plan the exploration and exploitation;

- is_stage2: This is a Boolean variable showing the current stage of the SA process. As mentioned above, it would affect the number of available directed move types.

C. Algorithms

By varying the learning objective (2) and the specific form of action sampling probability P_a, we can interpret and implement different algorithms.

1) Multi-armed Bandit: Let $\mu_1, \ldots, \mu_N$ be the mean values associated with the reward distributions for move types $1, \ldots, N$ (or block types). In a MAB algorithm, the agent iteratively chooses one move type per step and observes the associated reward. The objective is to maximize the sum of the collected rewards, i.e., the overall reward in the long run.

Upper Confidence Bound (UCB) selects the action with the highest score with a probability of 1. On the other hand, probabilistic MAB algorithms, such as Boltzmann Exploration (softmax) and Thompson Sampling, calculate the action probability distribution given by P_a and then take actions sampled from this distribution.

However, a limitation of these MAB algorithms is that they are designed to maximize the overall rewards, while concept drift might be present in the SA process of FPGA placer, i.e., the expected reward for a directed move type can change at every time step. In this case, the optimal policy will be a dynamic schedule represented by a non-stationary probability distribution P_a to be learned.

2) Evolution Strategy: Evoluation Strategy is an algorithm inspired by biological evolution. It maintains a population of candidate solutions and creates new candidate solutions by recombining and mutating the existing candidates. By eliminating the worse candidates in the population, all candidates generally converge to the optimal solution. Because ES has emerged as a highly effective solution for continuous optimization problems and its popularity has been further bolstered by the recent surge of research [12], we introduce ES to solve FPGA placement with VPR-Gym, which also flexibly adapts to the non-stationary reward distributions of different action types during SA.

Figure 3 shows the flow of ES, which proposes a candidate vector $\mathbf{Q}$ from its population and transfers it into a probability distribution via a Sigmoid activation function. Action will be sampled from the probability distribution until τ steps have been taken. Then the agent performs an update and proposes a candidate $\mathbf{Q}$ again. The ES agent optimizes the probability P_a periodically and dynamically as moving windows, i.e., every τ steps, the ES agent will propose $\mathbf{Q}$ to minimize the loss term in (2).

Specifically, we introduce two ES algorithms into VPR-Gym, including Test-Based Population Size Adaptation (TBPSA) [13] and Covariance Matrix Adaptation (CMA) [14]. TBPSA and CMA both have the ability to escape from local optima, increasing the robustness to non-stationary in FPGA placement. TBPSA applies a population size adaptation technique [15] to increase the population size to explore the search space more thoroughly when getting stuck in local optima or when there is a need to handle the noise. Conversely, it can decrease the population size to accelerate convergence. On the other hand, CMA uses a restart strategy [16] to escape from local optima, i.e., when the algorithm stagnates restart from a random initial point. According to [13], [15], TBPSA outperforms CMA in noisy settings, as well as multimodal settings (which means the presence of many local optima).

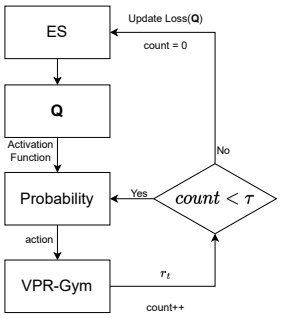

Fig. 3. Evolution Strategy Solution for the Optimization Problem.

IV. Experiments

To demonstrate the capability of our platform in solving FPGA placement with a diversity of AI techniques, we evaluate the QoR-runtime tradeoffs on four MAB algorithms with different action sampling techniques and two ES algorithms. Specifically, the MAB algorithms include Softmax (the original choice in VPR), Upper Confidence Bound (UCB) [17], Discounted Thompson-Sampling (DTS) [18], and Boltzmann–Gumbel Exploration (BGE) [19]. A brief overview of these algorithms is presented below:

- UCB: Upper confidence bound represents a combination of historical observed reward and uncertainty. Action with the highest upper confidence bound is selected.
- DTS: Thompson sampling generates posterior distribution for each action based on Bayesian control rules [20] and uses it as the probability distribution to sample the action. In DTS, a discount factor is implemented to reduce the weight of older records.
- BGE: BGE belongs to the Boltzmann Exploration category like the softmax algorithm. BGE includes a noise term drawn from a Gumbel distribution to guarantee near-optimal performance [19].

The ES algorithms include TBPSA and CMA, which are introduced in Section III-C.

A. Experimental Setup

- Environment: We compare the AI algorithms in two different environments, which include a basic environment **VprEnv** and a custom environment **VprEnv_blk_type** as defined in Section III-B2.
- Benchmark: Experiments on are **VprEnv** conducted on eight circuits from the Titan23 benchmark suite [1], which are stereo_vision, bitonic_mesh, neuron, SLAM_spheric, dart, mes_noc, denoise, and

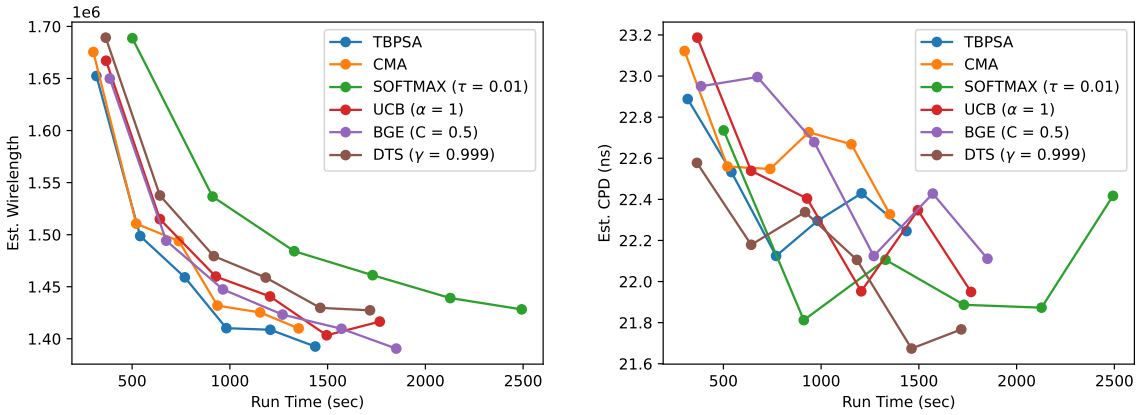

Fig. 4. Comparison of different algorithms in VprEnv at 6 different runtime points (3-seed average). The lower the better. TBPSA and CMA are ES algorithms. SOFTMAX, UCB, BGE, and DTS are MAB algorithms.

cholesky_mc. Experiments on **VprEnv_blk_type** are conducted on five circuits from the Titan23 benchmark suite [1], which are stereo_vision, bitonic_mesh, neuron, SLAM_spheric, and dart.

- Metrics: We evaluate the runtime/QoR tradeoffs of different placement techniques by the post-placement estimated wirelength (WL) (based on VPR's fanout-adjusted HPWL metric) and critical path delay (CPD) versus runtime to align with [3], [21].
- Runtime point: For each algorithm, we set 6 different numbers of moves attempted at each temperature.
- Implementation details: The parameters of the ES algorithms are updated every 100 steps. The reward that is used for updating the MAB and ES algorithms is calculated based on *basic* reward function from [21] as suggested in [3].

B. Results in Basic Environment

Fig. 4 presents the post-placement estimated WL and CPD of different algorithms on the Titan23 benchmark suite. We discuss the performance to showcase different algorithms' performance on VPR-GYM.

1) MAB algorithms vs. ES algorithms: Compared to the ES algorithms, the MAB agents require a higher runtime to complete all the runtime points in each SA temperature, as they update on every move, leading to the increased computational cost. In terms of the estimated WL, the ES algorithms exhibit superior performance across all runtime points. In terms of the estimated CPD, DTS shows the best overall performance. However, all six algorithms demonstrate CPD that doesn't monotonically decrease, which is consistent with the results of *RLPlace* method [3].

2) TBPSA vs. CMA: TBPSA demonstrates better performance compared with the CMA algorithm on both estimated WL and CPD, although its runtime is slightly higher under the same runtime setting. Recall that TBPSA outperforms CMA in noisy settings and multimodal settings supported by experimental and theoretical results [13], [15]. Either the robustness to noise or the stronger ability for escaping local optima helps TBPSA achieve superior performance compared with CMA.

3) BGE vs. Softmax: Cesa-Bianchi, Nicolò, et al. [19] suggest that the Boltzmann Exploration strategy with a monotone learning-rate sequence is sub-optimal and they offer a non-monotone schedule strategy which is the BGE algorithm. Although according to the conclusion by Cesa-Bianchi [19] BGE algorithm should be better than the softmax algorithm, the BGE agent doesn't show a dominant result compared with softmax: BGE is better in wire length but worse in CPD compared with softmax. As the optimal directed move type is subject to change over time, we anticipated this result since it violates the assumption underlying the MAB problem.

4) DTS vs. Other MAB algorithms: DTS is a variant of Thompson Sampling designed for non-stationary multi-armed bandit problems. Non-stationary multi-armed bandit means that the expected reward of each arm (action) is subject to change, which is matching the SA-based FPGA RL placement problem's nature, as different directed moves would be preferred at different annealing temperatures. Hartland et al. [22] propose that standard MAB algorithms such as UCB and Boltzmann Exploration, are not appropriate for abruptly changing environments. The results demonstrate that DTS outperforms other MAB algorithms, which matches Hartland et al.'s conclusion. The results indicate that it's fruitful to consider the non-stationary problem in designing the SA-based FPGA RL placement agents.

C. Results in Custom Environment

Fig. 5 shows the comparison of post-placement estimated wirelength and CPD between multiple algorithms on custom environment. We see that the TBPSA outperforms all other algorithms in both wirelength and CPD. This result demonstrates that the TBPSA is able to learn better than the MAB agents in environments with larger action space. In the case of employing more directed move types and block types to the

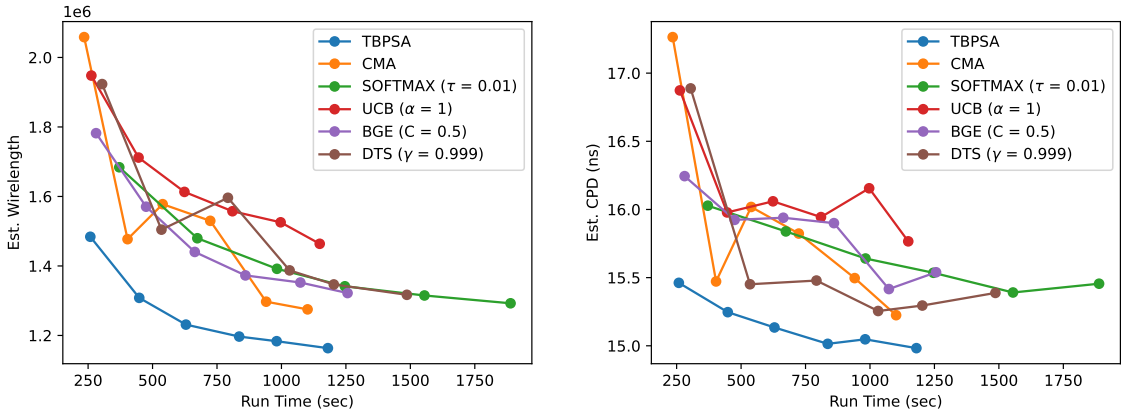

Fig. 5. Comparison of different algorithms on **VprEnv_blk_type** at 6 different runtime points. TBPSA outperforms other algorithms on both post-placement estimated wirelength and CPD.

TABLE I
CMA AGENT'S OVERHEAD AT 6 DIFFERENT RUNTIME POINTS

Inner_num	Overhead (s)	Total Time (s)	Overhead (%)
0.1	48.40	97.92	49.43
0.2	93.11	155.93	59.71
0.3	129.42	204.03	63.43
0.4	185.86	280.81	66.19
0.5	209.44	312.96	66.92
0.6	275.28	386.41	71.24

toolkit, using the TBPSA algorithm can better enhance the CAD tool's performance than other algorithms.

D. Discussion

Through the above experiments, the proposed ES has proven to be superior and more efficient than the traditional RL (MAB) approaches for FPGA placement, especially in environments with a more complex action space. Besides, VPR-Gym enables the evaluation of a diverse range of AI optimization algorithms without implementing them in the VPR source code in C++, leading to insightful discoveries such as introducing discounting factors into MAB agents and adding robustness to noise. Furthermore, Table I shows the VPR-Gym's overhead using the CMA agent on the neuron circuit which belongs to the Titan23 benchmarks, in terms of the time spent between the point the agent receives a reward and the point the agent takes an action in the next step, and its proportion in the total running time. Although the CMA agent can perform calculations fast, the overhead is mainly caused by Inter-process communication between VPR and the OpenAI Gym framework. This means that VPR-Gym can mainly be used as a tool to fast investigate, prototype, select, and improve a wide range of AI techniques, while the optimized algorithms should still be transplanted to the original VPR kernel to further conduct and finalize FPGA placement in practice.

V. RELATED WORK

A variety of Gyms have been built in different application areas, such as compiler [23], auto driving [24], Robotics [25], and networking [26]. For example, Piotr et. [26] have built Ns3-gym in order to apply ML techniques such as RL to the Ns-3 network simulator. Their ns3-gym toolkit brings significant convenience to the usage of machine learning algorithms in the area of networking.

Machine learning shows a growing trend in the FPGA Placement area. Abeer et al. [27] employed a Convolutional Neural Network (CNN) to approximate the routability based on the heatmap of the circuit placement. In addition, they proposed a novel algorithm [11] that forecasts the final placement congestion based on features available early in placement using a convolutional encoder-decoder.

Murray et al. [5] proposed to use the MAB agent to determine the type of block during the SA. As a result, the runtime needed to achieve the same QoR is greatly reduced. In this work we provide a custom environment **VprEnv_blk_type** that combines both search space in their work and *RLPlace*, granting more freedom to the agent.

VI. CONCLUSION

In this paper, we propose the VPR-Gym for the sake of lowering the barrier-of-entry to the development of AI optimization algorithms for FPGA RL placement. VPR-Gym significantly simplifies the agent implementation in the VPR placement research area and enables the interaction between VPR and some advanced Machine Learning and optimization libraries such as Nevergrad [9]. Furthermore, we perform experiments on VPR-Gym and provide exciting findings that can help enhance the FPGA CAD tools' placement performance from an algorithmic perspective.

In the future, we plan to extend the VPR-Gym's features. The current trend of RL is using deep neural networks as the agent policy. We will focus on adding state information to VPR-Gym to facilitate studies of Deep Q-Network (DQN), contextual bandit, and other RL techniques in our future work. Another interesting avenue is shaping the reward function, which can better relate to the post-placement estimated wirelength and CPD metrics.

REFERENCES

[1] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic cad," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–8.

[2] V. Betz and J. Rose, "Vpr: A new packing, placement and routing tool for fpga research," in *FPL*, vol. 97, 1997, pp. 213–222.

[3] M. A. Elgammal, K. E. Murray, and V. Betz, "Rlplace: Using reinforcement learning and smart perturbations to optimize fpga placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2532–2545, 2022.

[4] M. A. Elgamma, K. E. Murray, and V. Betz, "Learn to place: Fpga placement using reinforcement learning and directed moves," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 85–93.

[5] K. E. Murray and V. Betz, "Adaptive fpga placement optimization via reinforcement learning," in *2019 ACM/IEEE 1st Workshop on Machine Learning for CAD (MLCAD)*, 2019, pp. 1–6.

[6] K. Vorwerk, A. Kennings, and J. W. Greene, "Improving simulated annealing-based fpga placement with directed moves," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 179–192, 2009.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[9] J. Rapin and O. Teytaud, "Nevergrad - A gradient-free optimization platform," https://GitHub.com/FacebookResearch/Nevergrad, 2018.

[10] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker *et al.*, "Vtr 8: High-performance cad and customizable fpga architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.

[11] A. Al-Hyari, A. Shamli, T. Martin, S. Areibi, and G. Grewal, "An adaptive analytic fpga placement framework based on deep-learning," in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, 2020, pp. 3–8.

[12] Z.-H. Zhan, L. Shi, K. C. Tan, and J. Zhang, "A survey on evolutionary computation for complex continuous optimization," *Artificial Intelligence Review*, pp. 1–52, 2022.

[13] M.-L. Cauwet and O. Teytaud, "Population control meets doob's martingale theorems: the noise-free multimodal case," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 321–322.

[14] N. Hansen, "The cma evolution strategy: a comparing review," *Towards a new evolutionary computation: Advances in the estimation of distribution algorithms*, pp. 75–102, 2006.

[15] M. Hellwig and H.-G. Beyer, "Evolution under strong noise: A self-adaptive evolution strategy can reach the lower performance bound-the pccmsa-es," in *Parallel Problem Solving from Nature–PPSN XIV: 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings*. Springer, 2016, pp. 26–36.

[16] A. Auger and N. Hansen, "A restart cma evolution strategy with increasing population size," in *2005 IEEE Congress on Evolutionary Computation*, vol. 2, 2005, pp. 1769–1776 Vol. 2.

[17] T. L. Lai, H. Robbins *et al.*, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[18] V. Raj and S. Kalyani, "Taming non-stationary bandits: A bayesian approach," *arXiv preprint arXiv:1707.09727*, 2017.

[19] N. Cesa-Bianchi, C. Gentile, G. Lugosi, and G. Neu, "Boltzmann exploration done right," *Advances in neural information processing systems*, vol. 30, 2017.

[20] P. A. Ortega and D. A. Braun, "A minimum relative entropy principle for learning and acting," *Journal of Artificial Intelligence Research*, vol. 38, pp. 475–511, 2010.

[21] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker *et al.*, "Vtr 8: High-performance cad and customizable fpga architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.

[22] C. Hartland, S. Gelly, N. Baskiotis, O. Teytaud, and M. Sebag, "Multi-armed bandit, dynamic environments and meta-bandits," 2006.

[23] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner *et al.*, "Compilergym: Robust, performant compiler optimization environments for ai research," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 92–105.

[24] L. N. Alegre, "SUMO-RL," https://github.com/LucasAlegre/sumo-rl, 2019.

[25] 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)*, 2021.

[26] P. Gawłowicz and A. Zubow, "Ns-3 meets openai gym: The playground for machine learning in networking research," in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019, pp. 113–120.

[27] A. Alhyari, A. Shamli, Z. Abuwaimer, S. Areibi, and G. Grewal, "A deep learning framework to predict routability for fpga circuit placement," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 334–341.