Reinforcement Learning Strategies for Compiler Optimization in High level Synthesis

Hafsah Shahzad*, Ahmed Sanaullah**, Sanjay Arora**, Robert Munafo*, Xiteng Yao*, Ulrich Drepper**, and Martin Herbordt*

*CAAD Lab, ECE Department, Boston University
**Red Hat Inc.

***Abstract*—High Level Synthesis (HLS) offers a possible programmability solution for FPGAs by automatically compiling CPU codes to custom hardware configurations, but currently delivers far lower hardware quality than circuits written using Hardware Description Languages (HDLs). One reason is because the standard set of code optimizations used by CPU compilers, such as LLVM, are not well suited for a FPGA back end. Code performance is impacted largely by the order in which passes are applied. Similarly, it is also imperative to find a reasonable number of passes to apply and the optimum pass parameter values. In order to bridge the gap between hand tuned and automatically generated hardware, it is thus important to determine the optimal sequence of passes for HLS compilations, which could vary substantially across different workloads.**

Machine learning (ML) offers one popular approach to automate finding optimal compiler passes but requires selecting the right method. Supervised ML is not ideal since it requires labeled data mapping workload to optimal (or close to optimal) sequence of passes, which is computationally prohibitive. Unsupervised ML techniques don't take into account the requirement that a quantity representing performance needs to be maximized. Reinforcement learning, which represents the problem of maximizing long-term rewards without requiring labeled data has been used for such planning problems before. While much work has been done along these lines for compilers in general, that directed towards HLS has been limited and conservative. In this paper, we address these limitations by expanding both the number of learning strategies for HLS compiler tuning and the metrics used to evaluate their impact. Our results show improvements over state-of-art for each standard benchmark evaluated and learning quality metric investigated. Choosing just the right strategy can give an improvement of 23× in learning speed, 4× in performance potential, 3× in speedup over -O3, and has the potential to largely eliminate the fluctuation band from the final results. This work provides basis for an efficient recommender system enabling developers to choose the best possible reinforcement learning training options based on their target goals.

I. Introduction

High Level Synthesis (HLS) is a critical part of the Field Programmable Gate Array (FPGA) tool chain since it can substantially reduce the complexity and turnaround time for building custom hardware. This is especially crucial as FPGAs become critical components in the data center and HPC, e.g., in SmartNICs and disaggregated clusters, and run a variety of complex applications that must be coded by programmers without expertise in traditional logic design. Unlike traditional Hardware Description Languages (HDLs), HLS can generate hardware directly from CPU codes by automatically translating sequential functional descriptions into spatial circuits. This overlap between CPU and FPGA means new compilers need not be written from scratch. Rather, existing backend compilers such as LLVM can be modified to target FPGAs. Such a modification typically involves: i) changing the optimization strategy for the Intermediate Representation (IR) code to better map the CPU-like sequential to FPGA-like spatial programming model, and ii) adding support for hardware generation through mapping of IR code fragments to hardware blocks and the interconnects between them. The former is especially important here since a different back end means simply reusing CPU optimization strategies delivers far lower hardware quality than circuits written using Hardware Description Languages (HDLs) [1], [2].

Since there are dozens of possible optimizations passes, the number of possible sequences of passes undergoes combinatorial explosion and discovering the appropriate optimization strategy for FPGAs is not practical. Moreover, any discovered optimization strategy has limited reuse given the diversity of FPGA workloads. Automating this discovery process is thus essential. One potential approach is to use supervised machine learning algorithms to model the relationship between input IR codes and effective optimization strategies. However, this approach is not practical due to the complexity of

building a required labeled training data set - we hit the manual discovery roadblock again here. Another, more promising approach, is to use Reinforcement Learning (RL). Unlike manual or supervised approaches, RL can traverse the optimization space for input codes and, based on system state evolution and reward feedback, learn the LLVM optimization pass orderings that give good hardware quality.

While existing efforts in RL based HLS tuning have demonstrated improvements over -O3, they have been limited in scope. Specifically, these efforts have: i) only explored a small number of learning strategies, and ii) evaluated the impact of these strategies using a single metric for learning quality i.e. speedup over *-O3*. This is a significant drawback since learning goals can vary substantially across FPGA workloads and developer requirements. For example, developers can prioritize lower turnaround times over largest speed up values. In such cases, a different learning strategy would be needed which is able to trade off achieved speed up for a faster learning rate. Thus, similar to a uniform optimization strategy, a generic learning strategy is also inefficient.

In this work, we address the above limitation by expanding both the number of valid learning strategies for HLS compiler tuning and the metrics used to evaluate their impact. To achieve this, we start by implementing an existing effort as the baseline strategy - this strategy trains a reinforcement learning model to learn the number of times each optimization pass should be applied. Next, we identify and implement a number of additional strategies that govern the agent-environment interaction and can potentially impact learning - we use these strategies to vary what is being learnt and not just *how* it is learnt. Next, we identify speed, fluctuation band and performance potential as metrics for evaluating learning quality, in addition to the existing metric of speedup over *-O3*. Finally, we evaluate the effectiveness of our approach by training the RL model using the Proximal Policy Optimization (PPO) method on all 9 benchmarks evaluated in former work and shortlisted from CHStone suite and Legup examples.

The specific contributions of this paper are:

- Improving RL for HLS by enabling greater flexibility for developers in making trade offs based on their target learning goals.
- Identifying four novel learning strategies for HLS compiler tuning.
- Identifying and utilizing three additional novel metrics for learning quality.
- Demonstrating the effectiveness of our approach by comparing against the state-of-the-art using the CHStone benchmark suite.

The rest of this paper is organized as follows. Section II discusses some of the prior work in this area. Section III discusses the RL framework for HLS compiler tuning used in existing efforts. Section IV presents our proposed learning strategies and learning quality metrics. Section V evaluates the effectiveness of our strategies using the 9 benchmarks suite. Finally, Section VI gives the conclusion.

II. Relevant Work

Compilers execute optimization passes to transform programs into efficient forms by leveraging the hardware design patterns. Optimization can serve several goals: reduce area/resource utilization, reduce latency, increase parallelism, reduce power/energy etc. Modern compilers offer various compilation options to the user [3]. Compilers provide options to select optimization levels for different goals (size, speed, debuggability) and within these goals sometimes allow to adjust for the time allotted to the optimization passes to control turn-around times. However, the work performed for each of the optimization levels is defined statically and on a rather ad-hoc basis which doesn't take the problem nor really the variety of the target hardware into account. Even if the passes type, number and order applied to different applications in the standard -Ox level vary, the strategy is primarily fixed and insufficient [4]. Moreover the strategy is designed primarily for CPU workloads and hence not optimal for other specialised back ends like FPGAs. For simple applications like matrix multiply, we have seen that with appropriate number of passes and their ordering, more than 60% improvement in performance over the standard -O3 is possible.

Compiler optimization problem space for HLS is vast. Hardware quality is affected not only by the high level program but also by several other factors such as pass ordering, the number of passes and their parameter values. Hand tuned heuristic-based methods that been formerly proposed for compilers [5] no longer suffice. Compiler optimization has evolved over the years, from iterative compilation exploring the enumerations of the observation space one by one [4], [6] to machine learning based modeling in which models are trained to make relevant predictions [7], [8]. Recent years have seen a surge in compiler optimization work employing reinforcement learning (RL), a subset of machine learning in which the agent learns continually by trial and error using its interactions with the environment. Prior research work employing RL for compiler optimization typically address the following categories:

1) Those that have looked at phase ordering to tackle the compiler optimization problem such as [9]–[14].

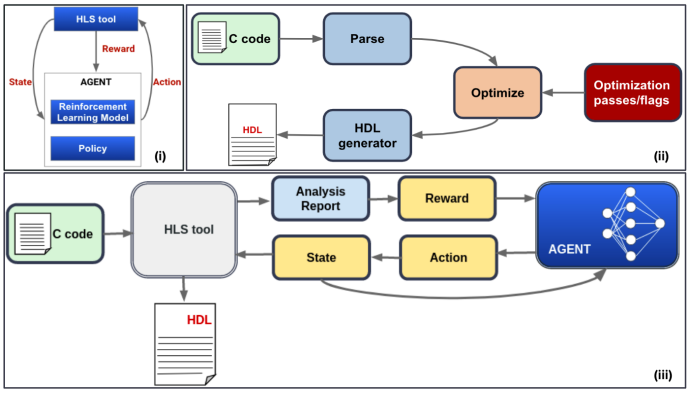

Fig. 1. Overall Block Diagram: (i) - generic Reinforcement Learning (ii) - generic High Level Synthesis flow (iii) - combined, RL based HLS

Amongst these, [12] is the only state-of-art framework, to the best of our knowledge, that addresses the problem particularly using High Level Synthesis for specialised backends such as FPGAs. Our works builds upon it to improve the existing approach and explore further possibilities.

2) Others that analyze specific parameters with respect to compiler passes for example inferring appropriate vectorization and interleaving factors for the loops in the code [15], Improving the register allocation problem in compilers such as in [16]. Polyhedral models that optimize loops in the program and try to find an optimal schedule for the instructions in the defined polyhedral [17]–[19]. MLGO [20] where machine learning is applied with respect to the inlining pass.

3)Proposed frameworks and platforms such as Supersonic [21] to choose and tune a RL architecture for code optimization tasks, CompilerGym [22] that consolidates some compiler optimization frameworks into a platform to motivate research under the hood and generic tuning frameworks such as OpenTuner [23].

III. Background

In this section we delve deeper into reinforcement learning and high-level synthesis and how the two can be combined for compiler tuning. Figure 1 part (i) shows the generic RL flow,1 part (ii) shows the generic high level synthesis flow and 1 part (iii) shows the combined RL based HLS.

A. Reinforcement Learning

Reinforcement learning relies on a trial-and-error mechanism in order to learn, as opposed to labeled data in supervised learning. There are two major components to a RL framework: i) *Environment*: the problem that we are trying to learn to solve, and ii) *Agent*: used to perturb the environment and learn based on feedback. The smallest unit of *Environment-Agent* interaction is typically a *Step*. At each *Step*, the *Agent* predicts an *Action* that the *Environment* should take e.g. the next move in a game of chess. After taking the action, the *Environment* returns a *Reward* indicating the impact of the *Action*, as well as an updated *State* which represents the change in the *Environment* as a result of taking the *Action*. The next *Step* then starts and a new *Action* is predicted.

The above process continues till the *Environment* indicates that a conclusion has been reached e.g. the game of chess has ended. This collection of *Steps*, from the first predicted action *Action* to the *Action* that causes the *Environment* to reach a conclusion, is referred to as an *Episode*. At the end of an *Episode*, the *Environment* is reset and a new *Episode* starts. A collection of such episodes is referred to as an Iteration. The agent is updated once per iteration. The number of *Episodes* per *Iteration* can vary significantly based on a number of factors. At the end of each *Iteration*, the learning portion of a RL framework updates a *Policy* (deterministic/stochastic strategy) about which *Actions* cause *Agents* to maximize their long-term, cumulative rewards. RL assumes that the environment is Markov i.e. that the updated state also depends on the previous state and the action taken. It also assumes that the action taken is only dependent on the current state.

B. High Level Synthesis

High Level Synthesis (HLS) allows developers to implement workloads using High Level Language (HLL) codes, such as C, which can be compiled into circuits. Typical HLS compilation flows involve: 1) using a *Parser* to convert HLL codes into a generic Intermediate Representation (IR), 2) running an *Optimizer* on the IR that use a series of codes transformation passes to optimize the code, and 3) mapping optimized IR code fragments to hardware blocks and adding an appropriate interconnect using an *HDL Generator* by replacing the normal CPU-specific back end of the compiler. The optimization step (2) is critical since it can improve hardware quality by increasing parallelism and reducing dependencies/hazards in the code structure [24].

C. RL based HLS compiler tuning

The overall reinforcement learning framework for a high level synthesis compiler is illustrated in Figure 1. Reinforcement learning in this case requires specifying and tuning several components:

i) an *action space* that specifies appropriate actions that are suggested by the agent. Code optimization in compilers such as LLVM, is implemented as Passes. These operate on the intermediate representation(IR) of

the code and either analyse or transform portions of the program [3]. An action is usually the pass to apply next. Each pass transforms the IR into a valid yet, modified IR.

ii) an *observation space* that characterises the environment and provides its state representation after an action is applied. In this case, it can be (a) a statistical analysis of the IR into a vector of features such as information extracted from the code about the loops, arithmetic operations, memory blocks etc. [25], (b) a runtime profiling of the program source code to dynamically characterise the system [26], (c) a graph based modeling of the feature set using Abstract Syntax Trees (ASTs), Control and Data flow graphs (CDFGs), graph based embeddings of Single Static Assignment (SSA) [27] (d) DNNs, LSTM or other neural nets to characterise the target code into a vector of features [28], [29], (e) A vector of applied passes (action history) or a histogram of applied passes (action histogram) to define the state of the environment (f) a concatenation/tuple of multiple aforementioned states.

iii) a *reward computation mechanism* that provides feedback to the agent in terms of the efficacy of the applied action and configures the frequency of reward computation. For high level synthesis this is usually dependent on the HLS tool used [30]–[32]. Most prior work have used cycle count as a reasonable metric for reward. However, other metrics such as post-routing Frequency, wall clock times, area and resource utilization details etc can also be used for comparison [4]. Cycle count can be estimated a) using software profiling as in Legup [33] b) simulating VHDL/Verilog code using tools such as Modelsim [34] or c) estimating from the Control Flow Graphs (CFGs). For the overall reinforcement learning framework, the reward has to be structured carefully for example, each good intermediate *step* of training returns a less negative reward, while a bad intermediate step returns a more negative reward, and achieving the goal by the agent returns a large positive reward. This ensures the agent learns in an effective manner and does not get trapped in a vicious cycle while trying to maximise its rewards.

v) the *environment* which in this case, embodies the application program and its various configurations. It also encapsulates the High Level Synthesis tool that supports user specified custom optimization passes. HLS is like a compiler however, the difference between CPU-targeting compilers and HLS environments is that in case of the latter the "instruction set" is not fixed and can be created alongside the logic which then uses these instructions. It has a front end where high level language (HLL) code is converted to language independent format such as intermediate representation (IR). At the middle-end, optimizations are applied to modify the IR and generate a new one. At the back-end, the HLS tool also contains some mechanism of returning performance estimates (clock cycle count, resource utilization etc) to execute the application code.

While numerous efforts have used reinforcement learning for compiler space, there is still a need to understand the options available in hyper-parameter exploration of the compiler reinforcement learning framework i.e. optimising *step* of training, *episode* length, and a quantitative analysis of the state, action and reward space for high level synthesis compilers in particular. Our goal in this work, is to highlight this research area and make valuable contributions.

IV. Learning Strategies

In this section, we start off by discussing the base learning strategy. Next we discuss the 4 additional strategies implemented in our work. Finally we talk about learning quality metrics that can be used to compare the impact of these strategies.

A. Base Learning Strategy

Our base learning strategy, based on Autophase [12], is given below.

- Environment: The environment is composed of Legup HLS tool [30] and the specific application. It outputs a count of the number of clock cycles needed to execute the target application.
- Agent: Proximal Policy Optimization(PPO) is used as the reinforcement learning agent [35].
- Action: Each action represents a single optimization pass. This appended to an ordered list of actions for the episode, which in turn is used during compilation.
- State: A histogram of applied passes is used to represent the state - this is referred to as an *action histogram*. To implement this histogram, a list equal to length of passes is initialised and each element of the list is incremented and updated when a pass corresponding to the pass number is applied. The agent learns to map the distribution of number of passes applied, to the next pass that should be applied, in order to maximize the averaged/expected (across episodes) sum of rewards across time-steps in an episode.
- Reward: The reward is defined as the difference in cycle count between the previous step of training and the current step - lower cycle count thus results in a higher reward value. Reward is set to a default value of 0 for each step, except for the last step in

the episode in which the actual clock cycle count is obtained from the HLS tool. The "previous step" reward value for the first step is set to the cycle count for the -O0 flag. *Maximum Reward* is defined as the highest value of reward obtained by any episode during training.

- Episode size: The episode size is fixed to 45 passes. This also places an upper bound on the size of action histogram.

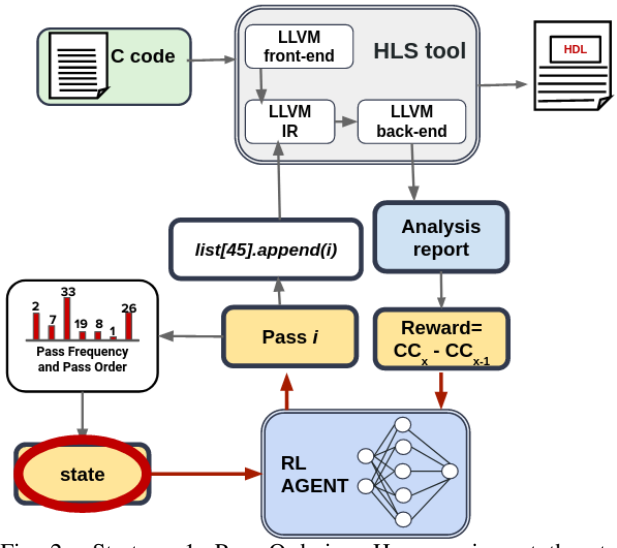

Fig. 2. Strategy 1: Pass Ordering. Here we impact the state by passing information about the pass order to the agent and evaluating the impact on its learning policy.

B. Strategy 1: Pass Ordering

In a typical set of compiler optimization passes, both the repetition and ordering of individual passes impact the outcome of the optimization process. In the base learning strategy above, pass repetition is available as the action histogram while pass ordering is available as a list that is passed to the HLS tool during the compilation process. This means that, while both pass repetition and order are factored in during reward calculation, only the pass repetition is used to represent state. This means that the same state can potentially correspond to substantially different reward values - these collisions can potentially negatively impact learning. To address this, we propose a new learning strategy that explicitly takes pass ordering into account. This can be done using two methods.

In Method 1, instead of using *action histogram* as the observation space, we use *action history* to learn the optimal pass ordering. Here, each value in the observation space is the actual pass that was applied at the corresponding step of training. Just like the action histogram, a fixed number of passes are applied in each episode and thus the observation space has known bounds.

In Method 2, the reward computation frequency is varied while the observation space continues to be the *action histogram*. In the base learning strategy, the actual reward from the HLS tool is calculated once per episode while all other steps use a default reward value. This corresponds to a reward frequency of 1, and only assigns a non-zero reward to the observation space represented by a complete histogram. In order to vary the reward frequency, we evaluate additional frequency values. For a reward frequency of less than the episode size, the HLS tool is invoked to give the cycle count after every fixed number of actions. In this case, instead of ordering between individual passes, we factor in the ordering of multiple sets of passes. For example, for a reward frequency of 2 and maximum episode size of 45, we get the reward from the HLS tool after 23 and 45 passes. For a reward frequency equal to the maximum episode size, reward is computed every step once an action is applied. In this case the complete ordering information for individual passes is factored in (represented by incremental changes to the action histogram).

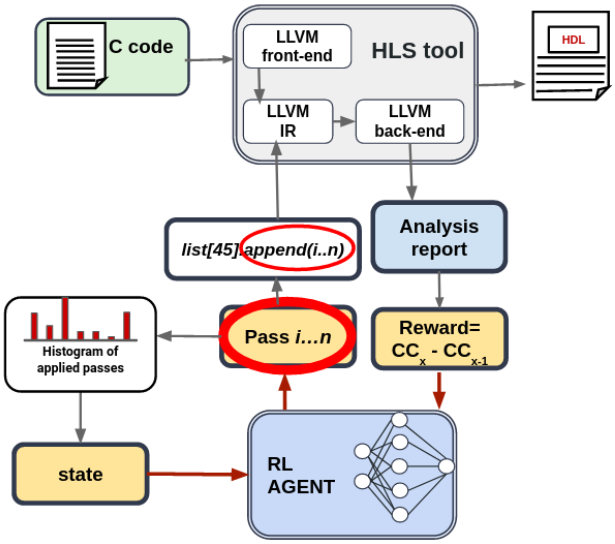

Fig. 3. Strategy 2: Action Tuples. In this case multiple actions are applied as opposed to a single action.

C. Strategy 2: Action Tuples

As discussed in Strategy 1, pass ordering has an impact on learning and can be factored in by modifying the observation space and reward frequency. Another approach to factoring in pass ordering is through action tuples. Unlike Strategy 1, which focuses on learning a global pass ordering, this strategy is aimed at determining groups of passes that work well together i.e. local pass ordering. To achieve this, we modify the action space to represent a tuple of passes as opposed to individual ones. That is, at each step, the agent predicts multiple

passes to apply. The maximum number of applied passes is kept (approximately) the same, which means that the episode size is reduced by an appropriate factor based on the size of the action tuple. Having a large tuple size for the action space allows the agent to learn which passes work well together and how to apply these tuples, which in turn can improve learning. However, if the tuple sizes become too large, the agent can take substantially longer to learn them. In the extreme case, this would be equivalent to trying to predict the entire set of pass orderings in a single step - the neural net for this can become fairly large.

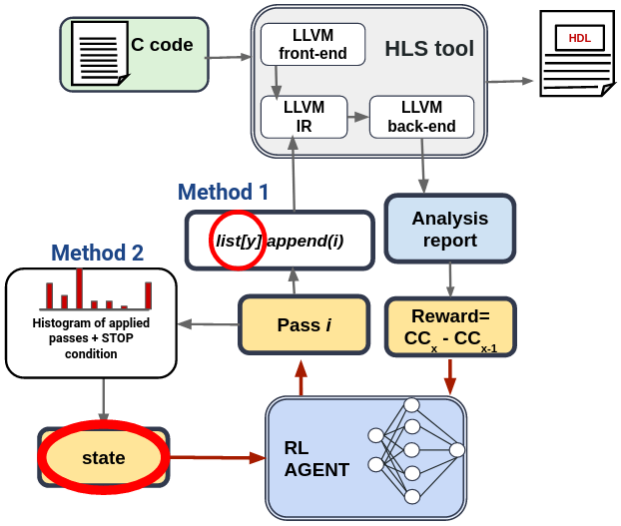

Fig. 4. Strategy 3: Episode Sizing. In this case the number of steps that constitute a single episode of training are varied to evaluate the imapct on learning quality.

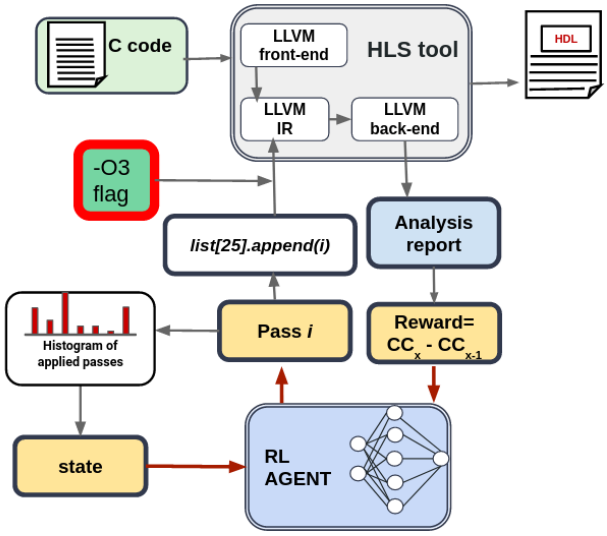

Fig. 5. Strategy 4: -O3 Backend. In this strategy -O3 flag is applied at the end of each episode to train the agent to perform better than the compiler standard of -O3 level.

D. Strategy 3: Episode Sizing

The size of an episode can also impact the learning quality since it determines the dimensions of the action histogram and the maximum number of passes that are applied. Having a smaller episode size means the upper bound on the size of the action histogram is lower, which can improve learning. However, the entire sequence of optimal passes must be specified within the limited number of steps. This not only limits the room for redundancy or sub-optimal passes, but it might not be possible to do so if the episode size is smaller than the size of the optimal pass ordering sequence. On the other hand, larger episode sizes have a greater margin for sub-optimal passes, but can have drawbacks such as greater combinations of action histograms. There are two methods for varying episode size.

In Method 1, we use static episode sizing. Similar to the base learning strategy, we fix the episode size for the duration of learning. In Method 2, we use dynamic episode sizing. Here, we train the agent to learn not only the pass frequency that maximises reward, but also the number of passes to apply within an *episode*. We insert a *stop* condition within the list of possible passes. Whenever, a stop pass is suggested by the agent, the training episode is terminated and the environment is reset after computing the reward. This reward is then mapped to the *action histogram* of the passes applied in that episode before the *stop* pass. The expectation is that the agent will learn the shortest sequence of passes that gives the highest reward.

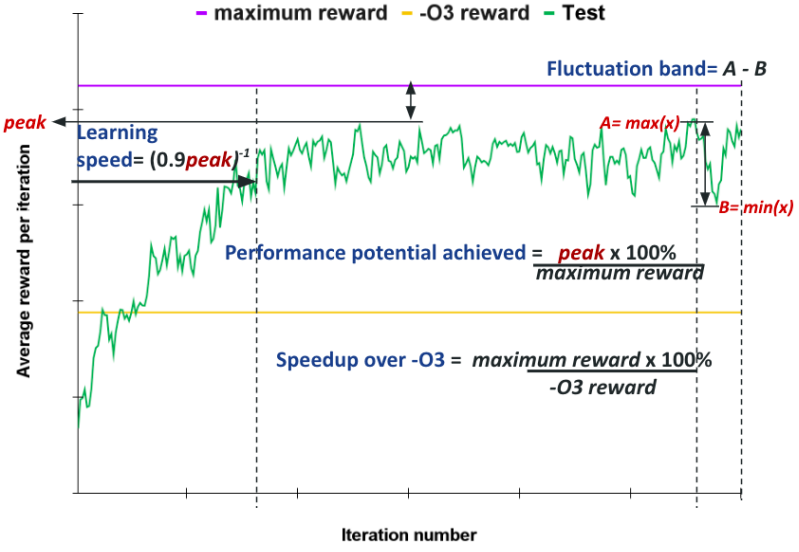

Fig. 6. Learning quality metrics and how they are calculated

E. Strategy 4: -O3 Backend

In the above strategies, our goal has been to learn the optimal pass ordering for the HDL generator back end. That is, what is the optimal code structure that allows the HDL generator to map code fragments and CFGs to high quality hardware. It assumes that the CPU and FPGA back ends are completely orthogonal and an entirely new optimization strategies must be learnt. However, this is not always the case and there is some potential overlap in how CPUs and FPGAs leverage different forms of parallelism.

This strategy aim to reuse the default CPU optimization strategy of a compiler instead of deriving a completely new one from scratch. Specifically, this is done by redefining the back end and, instead of targeting the HDL generator, learning the code transformations that enable the -O3 flag to be effective. Implementing the -O3 back end involves adding the -O3 flag to the action histogram before the HLS tool is invoked for clock cycle calculation. We also reduce the episode size to 25 (instead of the default 25) since we are not learning a complete optimization strategy.

F. Learning Quality Metrics

Figure 6 illustrates the different metrics used to evaluate learning quality for given training run.

i) Speedup over -03: This is defined as the ratio of the maximum reward obtained by any episode during training, and the reward from the -O3 flag.

i) Learning speed: This is defined as the number of iterations taken to reach 90% of the peak reward value. Peak reward value is defined to be the highest average reward value achieved across all training iterations.

ii) Performance potential achieved: This is defined as the ratio of the peak reward to the maximum reward obtained by any episode during training.

iii) Fluctuation band: This is the difference between the maximum and minimum average reward observed in the last 7% iterations of a training run.

V. Experimental Results

A. Experimental Setup

The framework is setup using Open AI Gym (0.21.0) to provide the unified environment interface for the reinforcement learning framework, Ray (1.7.0) to provide the unified API for reinforcement learning and its RLlib library to provide the agent interface. Tensorflow (2.6.0) is used for machine learning tasks together with Keras (2.6.0) to provide the neural net API for Python (3.9). To ensure standardization, each test is run using *2 workers* on a standard multi-core CPU, a total training time of *300 iterations* and the same initial *seed* value for random number generation. The same 9 benchmarks used in Autophase [12] are evaluated in this work. These include 6 CHStone benchmarks [36]: adpcm, aes, blowfish, gsm, motion and sha; and 3 benchmarks from Legup examples: matrixmultiply, dhrystone and qsort.

Similar to Autophase, we use Legup [33] as the HLS tool, which is built by modifying the LLVM 3.5 compiler. While we are constrained in using an older version of the LLVM compiler due to Legup, the methods for learning effective optimization strategies and generating high quality HDL code are applicable to the latest versions as well.

B. Baseline Strategy Validation

To validate our baseline implementation, which is based on Autophase, we run each of the 9 benchmarks evaluated in the paper. The *maximum reward* for each benchmark is computed by letting it run at the baseline configuration for around 10,000 iterations, and then setting the maximum reward as the highest performance achieved. The means of maximum rewards we obtained as a results of the above matches the 26% efficiency over the -O3 flag of the LLVM compiler stated by the authors in [12].

C. Aggregate Results

Figure 7 shows the results for each standard application used and every strategy investigated. Note that results for each quality metric are normalized with respect to each application. The average gives the arithmetic mean for each strategy over all applications. Below we discuss our findings with respect to each learning quality metric.

1) Learning Speed

Figure 7a shows how learning speed is impacted by strategies. Higher learning speed is more preferable since it means the agent can train faster. We note that on average Strategy 4 takes the least number of iterations to reach 90% of the peak reward value achieved. Overall up to 23× improvement in learning speed is possible if the correct learning strategy is selected versus an inefficient strategy.

2) Performance Potential Achieved

From figure 7b we can see that that most strategies exhibit reasonable performance potential however, some strategies notably give poor performance for certain applications. For *matrixmultiply*, setting the state as action history and using dynamic sizing gives drastically poor performance. However, for other applications such as *adpcm, blowfish, sha* these result in high performance

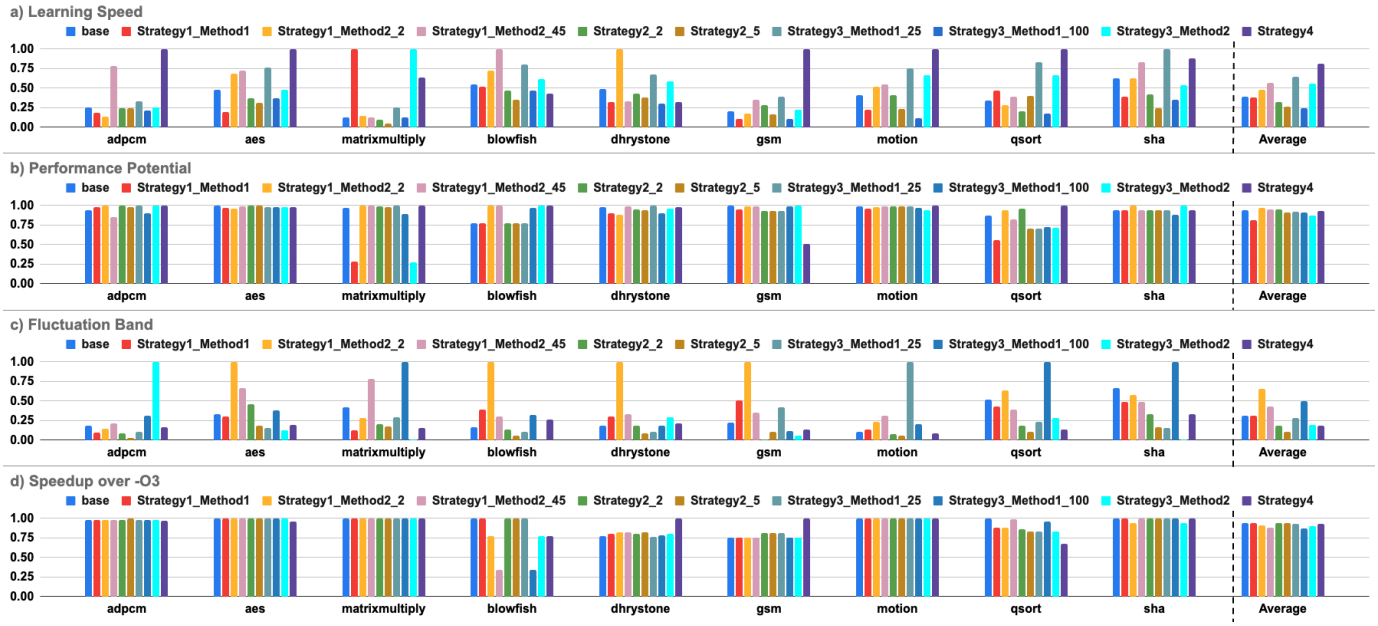

Fig. 7. Normalized Results for each metric with respect to the standard application and strategy used. A higher learning speed, higher performance potential, lower fluctuation band and a higher speedup over -O3 are desirable.

TABLE I
BEST STRATEGY IN TERMS OF EACH BENCHMARK AND METRIC

Benchmark	Learning speed	Performance potential	Fluctuation band	Speedup over -O3
adpcm	Strategy_4	*multiple*	Strategy2_5	Strategy2_5
aes	Strategy_4	*multiple*	Strategy3_Method2	All except Strategy_4
matrixmul	*multiple*	*multiple*	Strategy3_Method2	*all*
blowfish	Strategy1_Method2_45	*multiple*	Strategy2_5	*multiple*
dhrystone	Strategy1_Method2_2	*multiple*	Strategy2_5	Strategy_4
gsm	Strategy_4	*multiple*	Strategy2_2	Strategy_4
motion	Strategy_4	Strategy_4	Strategy3_Method2	*multiple*
qsort	Strategy_4	Strategy_4	Strategy2_5	*multiple*
sha	Strategy3_Method1_25	*multiple*	Strategy3_Method2	*multiple*

and even better than the baseline. Overall up to 4× improvement in performance potential is achievable by selecting the best strategy.

3) Fluctuation Band

A high fluctuation band means that the ripple in final steady state reward is high hence the confidence value is low. From figure 7c we can see that on average Strategy2_5 gives the best results in terms of the fluctuation band. On average Strategy1_Method2_2 results in the highest ripple. For *motion*, it is possible to largely eliminate the ripple by selecting Strategy3_Method2. From actual values, we note that more than 7700× improvement in fluctuation band is possible through the selection of reasonable strategy.

4) Speedup over -O3

As seen in figure 7d, up to 3% improvement in speedup over -O3 is possible by selecting the right strategy. Most strategies give a reasonable speedup over -O3. However, depending on the application, some like Strategy1_Method2_45 and Strategy3_Method1_100 give pretty poor performance for *blowfish* benchmark. Overall up to 3× improvement in speedup over -O3 is possible by choosing the correct strategy.

5) Strategy Selection

Table I gives the best strategy for each benchmark and metric. Where more than one strategies favor the particular test case, *multiple* is seen. For *matrixmultiply*, all strategies give a speedup over -O3 hence showing that for this benchmark, the standard compiler optimization strategy is highly inadequate. For *aes*, Strategy_4 leads to a degradation of the metric, speedup over -O3. An important deduction is that there is no one strategy that fits all. Different strategies favor different design criterion and it is up to the developer to choose according to their requirements.

VI. CONCLUSION

In this paper, we have explored a number of learning strategies that enable developers to customise their de-

sign according to their respective learning goals. To do this, we have proposed 4 additional learning strategies. We have also analyzed the results using 3 additional metrics that encapsulate developer goals more effectively when compared with the single speedup over -O3 standard metric. Results show there is no one size fit all solution: different learning strategies give best results in terms of different metrics. Choosing just the right strategy can give an improvement of 23× in learning speed, 4× in performance potential, 3× in speedup over -O3 and has the potential to largely eliminate the fluctuation band from the final results.

REFERENCES

[1] C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. C. Herbordt, "OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–8.

[2] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 409–420.

[3] LLVM, "LLVM's Analysis and Transform Passes," https://llvm.org/docs/Passes.html [Last accessed: Aug 10, 2022].

[4] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, N. Calagar, S. Brown, and J. Anderson, "The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware," *ACM Trans. Reconfigurable Technol. Syst.*, 2015.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[6] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, "Deconstructing Iterative Optimization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, pp. 1–30, 2012.

[7] Z. Wang and M. O'Boyle, "Machine Learning in Compiler Optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[8] H. Leather and C. Cummins, "Machine Learning in Compilers: Past, Present and Future," in *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 2020, pp. 1–8.

[9] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano, "The Phase-Ordering Problem: A Complete Sequence Prediction Approach," in *Automatic Tuning of Compilers Using Machine Learning*. Springer, 2018, pp. 85–113.

[10] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using Machine Learning to focus Iterative Optimization," in *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 2006, pp. 11–pp.

[11] J. Davidson, G. Tyson, D. Whalley, P. Kulkarni, J. Davidson, G. Tyson, D. Whalley, and P. Kulkarni, "Evaluating Heuristic Optimization Phase Order Search Algorithms," in *International Symposium on Code Generation and Optimization (CGO'07)*, 2007, pp. 157–169.

[12] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "Autophase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning," *arXiv preprint arXiv:2003.00671*, 2020.

[13] R. Mammadli, A. Jannesari, and F. Wolf, "Static Neural Compiler Optimization via Deep Reinforcement Learning," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2020, pp. 1–11.

[14] J. Wu, J. Xu, X. Meng, and Y. Lei, "A Highly Reliable Compilation Optimization Passes Sequence Generation Framework," *IEICE Transactions on Information and Systems*, vol. 103, no. 9, pp. 1998–2002, 2020.

[15] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "Neurovectorizer: End-to-end Vectorization with Deep Reinforcement Learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 242–255.

[16] M. Kim, J.-K. Park, and S.-M. Moon, "Solving PBQP-Based Register Allocation using Deep Reinforcement Learning," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 1–12.

[17] A. Brauckmann, A. Goens, and J. Castrillon, "PolyGym: Polyhedral Optimizations as an Environment for Reinforcement Learning," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 17–29.

[18] A. Brauckmann, A. Goens, and J. Castrillon, "A Reinforcement Learning Environment for Polyhedral Optimizations," *arXiv preprint arXiv:2104.13732*, 2021.

[19] J. Koo, P. Balaprakash, M. Kruse, X. Wu, P. Hovland, and M. Hall, "Customized Monte Carlo Tree Search for LLVM/Polly's Composable Loop Optimization Transformations," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2021, pp. 82–93.

[20] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "MLGO: a Machine Learning Guided Compiler Optimizations Framework," *arXiv preprint arXiv:2101.04808*, 2021.

[21] H. Wang, Z. Tang, C. Zhang, J. Zhao, C. Cummins, H. Leather, and Z. Wang, "Automating Reinforcement Learning Architecture Design for Code Optimization," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, 2022, pp. 129–143.

[22] 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.

[23] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An Extensible Framework for Program Autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[24] D. Koch, F. Hannig, and D. Ziener, *FPGAs for Software Programmers*. Springer International Publishing, 2016. [Online]. Available: https://books.google.com/books?id=p-N6DAAAQBAJ

[25] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost GCC: Machine Learning Enabled Self-tuning Compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[26] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly Selecting Good Compiler Optimizations using Performance Counters," in *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 2007, pp. 185–197.

[27] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, and H. Leather, "ProgramL: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations," in *International Conference on Machine Learning*. PMLR, 2021, pp. 2244–2253.

[28] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional Neural Networks over Tree Structures for Programming Language Processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[29] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-End Deep Learning of Optimization Heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 219–232.

[30] Microchip, "LegUp High-Level Synthesis Software," https://www.legupcomputing.com/ [Last accessed: April 18, 2021].

[31] A. Xilinx, " Vitis High Level Synthesis User Guide," https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS [Last accessed: April 18, 2021].

[32] Intel, "Intel® High Level Synthesis Compiler," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html [Last accessed: April 18, 2021].

[33] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski *et al.*, "From Software to Accelerators with LegUp High-level Synthesis," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013, pp. 1–9.

[34] Intel, "Modelsim FPGA Edition," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html/ [Last accessed: August 19, 2022].

[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[36] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2008, pp. 1192–1195.