Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning

ABSTRACT

The performance of the code a compiler generates depends on the order in which it applies the optimization passes Choosing a good order–often referred to as the *phase-ordering* problem–is an NP-hard problem. As a result, existing solutions rely on a variety of heuristics.

In this paper, we evaluate a new technique to address the phase-ordering problem: deep reinforcement learning. To this end, we implement a framework that takes a program and finds a sequence of passes that optimize the performance of the generated circuit. Without loss of generality, we instantiate this framework in the context of an LLVM compiler and target high-level synthesis programs. We use random forests to quantify the correlation between the effectiveness of a given pass and the program's features. This helps us reduce the search space by avoiding phases that are unlikely to improve the performance of a given program. We compare the performance of deep reinforcement learning to state-of-the-art algorithms that address the phase-ordering problem. In our evaluation, we show that reinforcement learning improves circuit performance by 29% when compared to using the -O3 compiler flag, and it achieves competitive results compared to the state-of-the-art solutions, while requiring fewer samples. More importantly, unlike existing state-of-the-art solutions, our reinforcement learning solution can generalize to more than 12,000 different programs after training on as few as a hundred programs for less than ten minutes.

1 INTRODUCTION

High-Level Synthesis (HLS) automates the process of creating digital hardware circuits from algorithms written in high-level languages. Modern HLS tools [1–3] use the same front-end as the traditional software compilers. They rely on traditional compiler techniques to optimize the input program's intermediate representation (IR) and produce circuits in the form of RTL code. Thus, the quality of compiler front-end optimizations directly impacts the performance of HLS-generated circuit.

Program optimization is a notoriously difficult task. A program must be just in "the right form" for a compiler to recognize the optimization opportunities. This is a task a programmer might be able to perform easily, but is often difficult for a compiler. To add to this complexity, often the optimization is hardware-dependent. Despite a decade of research on developing sophisticated optimization algorithms, an expert designer can still produce RTL that outperforms the results of HLS.

Without loss of generality, in this paper, we assume an LLVM compiler. In this case, the optimization of an HLS program consists of applying a sequence of analysis and optimization phases, where each phase in this sequence consumes the output of the previous phase, and generates a modified version of the program for the next phase. Unfortunately, these phases are not commutative which makes the order in which these phases are applied critical to the performance of the output.

Consider the program in Figure 1, which normalizes a vector. Without any optimizations the `norm` function will take $\Theta(n^2)$ to normalize a vector. However, a smart compiler will implement the *loop invariant code motion (LICM)* [4] optimization, which allows it to move the call to `mag` above the loop, resulting in the code on the left column in Figure 2. This optimization brings the runtime down to $\Theta(n)$—a big speedup improvement. Another optimization the compiler could perform is *(function) inlining* [4]. With inlining, a call to a function is simply replaced with the body of the function, reducing the overhead of the function call. Applying inlining to the code will result in the code in the right column of Figure 2.

```
__attribute__((const))
double mag(const double *A, int n) {
    double sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    return sqrt(sum);
}
void norm(double *restrict out,
          const double *restrict in, int n) {
    for(int i=0; i<n; i++) {
        out[i] = in[i] / mag(in, n);
    }
}
```

Figure 1: A simple program to normalize a vector.

Now, consider applying these optimization passes in the opposite order: first inlining then LICM. After inlining, we get the code on the left of Figure 3. Once again we get a modest speedup, having eliminated n function calls, though our runtime is still $\Theta(n^2)$. If the compiler afterwards attempted to apply LICM, we would find the code on the right of Figure 3. LICM was able to successfully move the allocation of sum outside the loop. However, it was unable to move the instruction setting `sum=0` outside the loop, as doing so would mean that all iterations excluding the first one would end up with a garbage value for sum. Thus, the internal loop will not be moved out.

As this simple example illustrates, the order in which the optimization phases are applied can be the difference between the program running in $\Theta(n^2)$ versus $\Theta(n)$. It is thus crucial to determine the optimal phase ordering to maximize the circuit speeds. Unfortunately, not only is this a difficult task, but the optimal phase ordering may vary from program to program. Furthermore, it turns out that finding the optimal sequence of optimization phases is an NP-hard problem, and exhaustively evaluating all possible sequences is infeasible in practice. In this work, for example, the search space extends to more than 2^{247} phase orderings.

```
void norm(double *restrict out,
          const double *restrict in, int n) {
    double precompute = mag(in, n);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

```
void norm(double *restrict out,
          const double *restrict in, int n) {
    double precompute, sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    precompute = sqrt(sum);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

Figure 2: Progressively applying LICM (left) then inlining (right) to the code in Figure 1.

```
void norm(double *restrict out,
          const double *restrict in, int n) {
    for(int i=0; i<n; i++) {
        double sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

```
void norm(double *restrict out,
          const double *restrict in, int n) {
    double sum;
    for(int i=0; i<n; i++) {
        sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
```

Figure 3: Progressively applying inlining (left) then LICM (right) to the code in Figure 1.

The goal of this paper is to provide a mechanism for automatically determining good phase orderings for HLS programs to optimize for the circuit speed. To this end, we aim to leverage recent advancements in deep reinforcement learning (RL) [5] to address the phase ordering problem. With RL, a software agent continuously interacts with the environment by taking actions. Each action can change the state of the environment and generate a "reward". The goal of RL is to learn a policy—that is, a mapping between the observed states of the environment and a set of actions—to maximize the cumulative reward. An RL algorithm that uses a deep neural network to approximate the policy is referred to as a deep RL algorithm. In our case, the the observation frmo the environment could be the program and/or the optimization passes applied so far. The action is the optimization pass to apply next, and the reward is the improvement in the circuit performance after applying this pass.

The particular framing of the problem as an RL problem has a significant impact on the solution's effectiveness. Significant challenges exist in understanding how to formulate the phase ordering optimization problem in an RL framework. In this paper, we consider three approaches to represent the environment's state. The first one is to directly use salient features from the program. The second approach is to derive the features from the sequence of optimizations we applied while ignoring the program's features. The third approach combines the first two approaches. We evaluate these observation space representations by implementing a framework that takes a group of programs as input and quickly finds a phase ordering that competes with state-of-the-art solutions. In particular, our results show that our best RL algorithm gets a 29% improvement over -O3 for nine real benchmarks. More importantly, unlike all state-of-the-art approaches, RL demonstrates the potential to generalize to thousands of different programs after training on just a hundred programs for less than ten minutes. Our main contributions are:

- Leverage Deep RL to address the phase-ordering problem, and show that it outperforms existing solutions.
- An importance analysis on the feature using random forests to significantly reduce the state (feature) space.
- A framework that integrates the current HLS compiler infrastructure with the deep RL algorithms.

2 BACKGROUND

2.1 Compiler Phase-ordering

Compilers execute optimization passes to transform programs into more efficient forms to run on various hardware targets. Groups of optimizations are often packaged into "optimization levels" , such as -O0 and -O3, for ease. While these optimization levels offer a simple set of choices for developers, they are handpicked by the compiler-designers and often most benefit certain groups of benchmark programs. The compiler community has attempted to address the issue by selecting a particular set of compiler optimizations on a per-program or per-target basis for software [6–9].

Since the search space of phase-ordering is too large for exhaustive search, many heuristics have been proposed to explore the space by using machine learning. Huang *et al.* tried to address this challenge for HLS applications by using modified greedy algorithms [10, 11]. It achieved 16% improvement vs -O3 on the CHstone benchmarks [12], which we used in this paper. In [13] both independent and Markov models were applied to automatically target an optimized search space for iterative methods to improve the search results. In [14], genetic algorithms were used to tune heuristic priority functions for three compiler optimization passes. Milepost GCC[15] used machine learning to determine the set of passes to apply to a given program, based on a static analysis of its features. It achieved 11% execution time improvement over -O3, for the ARC reconfigurable processor on the MiBench program suite1. In [16] the challenge was formulated as a Markov process and supervised learning was used to predict the next optimization, based on the current program state. OpenTuner [9] autotunes a program using an AUC-Bandit-meta-technique-directed ensemble selection of algorithms. Its current mechanism for selecting the compiler optimization passes does not consider the order or support repeated optimizations. Wang *et al.* [17], provided a survey for using machine learning in compiler optimization where they also described using program features might be helpful.

2.2 Reinforcement Learning Algorithms

Reinforcement learning (RL) is a machine learning approach in which an agent continually interacts with the environment [18]. In particular, the agent observes the state of the environment, and based on this observation takes an action. The goal of the RL agent is then to compute a policy–a mapping between the environment states and actions–that maximizes a long term reward.

RL can be viewed as a stochastic optimization solution for solving Markov Decision Processes (MDPs) [19], when the MDP is not known. An MDP is defined by a tuple with four elements: $S, A, P(s,a), r(s,a)$ where S is the set of states of the environment, A describes the set of actions or transitions between states, $s' \sim P(s,a)$ describes the probability distribution of next states given the current state and action and $r(s,a) : S \times A \longrightarrow R$ is the reward of taking action a in state s. Given an MDP, the goal of the agent is to gain the largest possible aggregate reward. The objective of an RL algorithm associated with an MDP is to find a decision policy $\pi^* : s \longrightarrow A$ that achieves this goal for that MDP:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau \sim \rho_{\pi(\tau)}} \left[\sum_t r(s_t, a_t) \right] = \arg\max_{\pi} \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t) \sim \rho_\pi(s_t,a_t)} \left[r(s_t, a_t) \right]. \quad (1)$$

Deep RL leverages a neural network to learn the policy (and sometimes the reward function). Policy Gradient (PG) [20], for example, updates the policy directly by differentiating the aggregate reward $\mathbb{E}$ in Equation 1:

$$\nabla_\theta J = \frac{1}{N} \sum_{i=1}^{N} \left[(\sum_t \nabla_\theta log \pi_\theta(a_{i,t}|s_{i,t}))(\sum_t r(s_{i,t}, a_{i,t})) \right] \quad (2)$$

and updating the network parameters (weights) in the direction of the gradient:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J, \quad (3)$$

Note that PG is an on-policy method in that it uses only the current policy to compute the new policy.

Over the past couple of years, a plethora of new deep RL techniques have been proposed [20–23]. In this paper, we mainly focus on Proximal Policy Optimization (PPO) [24], Asynchronous Advantage Actor-critic (A3C) [21].

PPO is a variant of PG that enables multiple epochs of minibatch updates to improve the sample complexity. It performs updates that maximizes the reward function while ensuring the deviation from the previous policy is small by inserting a penalty to the objective function. The loss function of PPO is defined as:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \quad (4)$$

where $r_t(\theta)$ is defined as a probability ratio $\frac{\pi_\theta(\mathrm{a_t}|\mathrm{s_t})}{\pi_{\theta_{old}}(\mathrm{a_t}|\mathrm{s_t})}$ so $r(\theta_{old}) = 1$. This term penalizes policy update that move $r_t(\theta)$ from $r(\theta_{old})$. $\hat{A}_t$ denotes the estimated advantage that approximates how good $\mathbf{a_t}$ is compared to the average. The second term in the *min* function acts as a disincentive for moving r_t outside of $[1-\varepsilon, 1+\varepsilon]$ where ε is a hyperparameter.

A3C uses an actor (usually a neural network) that interacts with the critic, which is another network that learns the environment and is used by the actor in predicting an estimate of the advantage function. The update performed by the algorithm can be seen as $\nabla_\theta log \pi_\theta(a_{i,t}|s_{i,t})\hat{A}_t$.

2.3 Evolutionary Algorithms

Evolutionary algorithm is another technique that can be used to search for the best compiler pass ordering. It contains a family of population-based meta-heuristic optimization algorithms inspired by natural selection. The main idea of these algorithms is to sample a population of solutions and use the good ones to direct the distribution of future generations. Two commonly used Evolutionary Algorithms are Genetic Algorithms (GA) [25] and Evolution Strategies (ES) [26].

GA generally requires a genetic representation of the search space where the solutions are coded as integer vectors. The algorithm starts by having a pool of candidates, then it iteratively evolves the pool to include solutions with higher fitness by the three following strategies: selection, crossover, and mutation. Selection keeps a subset of solutions with the highest fitness values. These selected solutions act as parents for the next generation. Crossover merges pairs from the parent solutions to produce new offsprings. Mutation perturbs the offspring solutions with a low probability. The process repeats until a solution that reaches the goal fitness is found or after a certain number of generations.

ES works similarly as GA. However, the solutions are coded as real numbers in ES. In addition, ES is self-adapting. The hyperparameters, such as the step size or the mutation probability, are different for different solutions. They are encoded in each solution, so good settings get to the next generation with good solutions. Recent work [27] has used ES to update policy weights for RL and showed it is a good alternative for gradient-based methods.

3 COMPILATION FRAMEWORK

We leverage an existing open-source HLS framework called LegUp [3] that compiles a C program into a hardware RTL design. In [10], an approach is devised to quickly determine the number of hardware execution cycles without requiring time-consuming logic simulation. We develop our reinforcement learning simulator environment based on the existing harness provided by LegUp and validate our final results by going through the time-consuming logic simulation. The framework takes a program (or multiple programs) and intelligently explores the space of possible passes to figure out an optimal pass sequence to apply. Table 1 lists all the passes used in our framework. The framework is illustrated in Figure 4.

3.1 HLS Compiler

Our framework takes a set of programs as input and compiles them to a hardware-independent intermediate representation

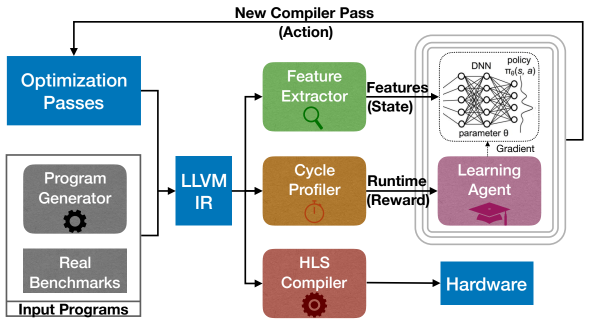

Figure 4: System Block Diagram

(IR) using the Clang front-end of the LLVM compiler. Optimization and analysis passes act as transformations on the IR, taking a program as input and emitting a new IR as output. The HLS tool LegUp is invoked after the compiler optimization as a back-end pass, which transforms LLVM IR into hardware modules.

3.2 Clock-cycle Profiler

Once the hardware RTL is generated, one could run a hardware simulation to gather the cycle count results of the synthesized circuit. This process is quite time-consuming, hindering RL and all other optimization approaches. Therefore, we approximate cycle count using the profiler in LegUp [10], which leverages the software traces and runs 20× faster than hardware simulation. In LegUp, the frequency of the generated circuits is set as a compiler constraint that directs the HLS scheduling algorithm. In other words, HLS tool will always try to generate hardware that can run at a certain frequency. In our experiment setting, without loss of generality, we set the target frequency of all generated hardware to 200MHz. We experimented with lower frequencies too; the improvements were similar but the cycle counts the different algorithms achieved were better as more logic could be fitted in a single cycle.

3.3 IR Feature Extractor

Wang *et al.* [17] proposed to convert a program into an observation by extracting all the features from the program. Similarly, in addition to the LegUp backend tools, we developed analysis passes to extract 56 static features from the program, such as the number of basic blocks, branches, and instructions of various types. We use these features as partially observable states for the RL learning and hope the neural network can capture the correlation of certain combinations of these features and certain optimizations. Table 2 lists all the features used.

3.4 Random Program Generator

As a data-driven approach, RL generalizes better if we train the agent on more programs. However, there are a limited number of open-source HLS examples online. Therefore, we expand our training set by automatically generating synthetic HLS benchmarks. We first generate standard C programs using CSmith [28], a random C program generator, which is originally designed to generate test cases for finding compiler bugs. Then, we develop scripts to filter out programs that take more than five minutes to run on CPU or fail the HLS compilation.

3.5 Overall Flow

We integrate the compilation utilities into a simulation environment in Python with APIs similar to an OpenAI gym [29]. The overall flow works as follows:

(1) The input program is compiled into LLVM IR using the Clang/LLVM.
(2) The IR Feature Extractor is run to extract salient program features.
(3) LegUp compiles the LLVM IR into hardware RTL.
(4) The Clock-cycle Profiler estimates a clock-cycle count for the generated circuit.
(5) The RL agent takes the program features or the histogram of previously applied passes and the improvement in clock-cycle count as input data to train on.
(6) The RL agent predicts the next best optimization passes to apply.
(7) New LLVM IR is generated after the new optimization sequence is applied.
(8) The machine learning algorithm iterates through steps (2)–(7) until convergence.

4 CORRELATION OF PASSES AND PROGRAM FEATURES

Similar to the case with many deep learning approaches, explainability is one of the major challenges we face when applying deep RL to the phase-ordering challenge. To analyze and understand the correlation of passes and programs features, we use random forests [30] to learn the importance of different features. Random forest is an ensemble of multiple decision trees. The prediction made by each tree could be explained by tracing the decisions made at each node and calculating the importance of different features on making the decisions at each node. This helps us to identify the effective features and passes to use and show whether our algorithms learn informative patterns on data.

For each pass, we build two random forests to predict whether applying it would improve the circuit performance. The first forest takes the program features as inputs while the second takes a histogram of previously applied passes. To gather the training data for the forests, we run PPO with high exploration parameter on 100 randomly generated programs and nine real benchmarks to generate feature–action–reward tuples. The algorithm assigns higher importance to the input features that affect the final prediction more.

4.1 Importance of Program Features

The heat map in Figure 5 shows the importance of different features on whether a pass should be applied. The random forest is trained with 150,000 samples generated from the random programs. The index mapping of features and passes can be found in Tables 1 and 2. For example, the yellow pixel corresponding to feature index 17 and pass index 23 reflects that *number-of-critical-edges* affects the decision on whether to apply *-loop-rotate* greatly. A critical edge in control flow graph is an edge that is neither the only edge leaving its source block, nor the only edge entering its destination block. The critical edges can be commonly seen in a loop as a back edge

Table 1: LLVM Transform Passes.

0	1	2	3	4	5	6	7	8	9	10	
-correlated-propagation	-scalarrepl	-lowerinvoke	-strip	-strip-nondebug	-sccp	-globalopt	-gvn	-jump-threading	-globaldce	-loop-unswitch	
11	12	13	14	15	16	17	18	19	20	21	
-scalarrepl-ssa	-loop-reduce	-break-crit-edges	-loop-deletion	-reassociate	-lcssa	-codegenprepare	-memcpyopt	-functionattrs	-loop-idiom	-lowerswitch	
22	23	24	25	26	27	28	29	30	31	32	33
-constmerge	-loop-rotate	-partial-inliner	-inline	-early-cse	-indvars	-adce	-loop-simplify	-instcombine	-simplifycfg	-dse	-loop-unroll
34	35	36	37	38	39	40	41	42	43	44	45
-lower-expect	-tailcallelim	-licm	-sink	-mem2reg	-prune-eh	-functionattrs	-ipsccp	-deadargelim	-sroa	-loweratomic	-terminate

Table 2: Program Features.

0	Number of BB where total args for phi nodes >5	28	Number of And insts
1	Number of BB where total args for phi nodes is [1,5]	29	Number of BB's with instructions between [15,500]
2	Number of BB's with 1 predecessor	30	Number of BB's with less than 15 instructions
3	Number of BB's with 1 predecessor and 1 successor	31	Number of BitCast insts
4	Number of BB's with 1 predecessor and 2 successors	32	Number of Br insts
5	Number of BB's with 1 successor	33	Number of Call insts
6	Number of BB's with 2 predecessors	34	Number of GetElementPtr insts
7	Number of BB's with 2 predecessors and 1 successor	35	Number of ICmp insts
8	Number of BB's with 2 predecessors and successors	36	Number of LShr insts
9	Number of BB's with 2 successors	37	Number of Load insts
10	Number of BB's with >2 predecessors	38	Number of Mul insts
11	Number of BB's with Phi node # in range (0,3]	39	Number of Or insts
12	Number of BB's with more than 3 Phi nodes	40	Number of PHI insts
13	Number of BB's with no Phi nodes	41	Number of Ret insts
14	Number of Phi-nodes at beginning of BB	42	Number of SExt insts
15	Number of branches	43	Number of Select insts
16	Number of calls that return an int	44	Number of Shl insts
17	Number of critical edges	45	Number of Store insts
18	Number of edges	46	Number of Sub insts
19	Number of occurrences of 32-bit integer constants	47	Number of Trunc insts
20	Number of occurrences of 64-bit integer constants	48	Number of Xor insts
21	Number of occurrences of constant 0	49	Number of ZExt insts
22	Number of occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (of all types)
24	Number of Binary operations with a constant operand	52	Number of memory instructions
25	Number of AShr insts	53	Number of non-external functions
26	Number of Add insts	54	Total arguments to Phi nodes
27	Number of Alloca insts	55	Number of Unary operations

so the number of critical edges might roughly represent the number of loops in a program. The transform pass *-loop-rotate* detects a loop and transforms a while loop to a do-while loop to eliminate one branch instruction in the loop body. Applying the pass results in better circuit performance as it reduces the total number of FSM states in a loop.

Other expected behaviors are also observed in this figure. For instance, the correlation between *number of branches* and the transform passes *-loop-simplify*, *-tailcallelism* (which transforms calls of the current function *i.e.*, self recursion, followed by a return instruction with a branch to the entry of the function, creating a loop), *-lowerswitch* (which rewrites switch instructions with a sequence of branches). Other interesting behaviors are also captured. For example, in the correlation between *binary operations with a constant operand and -functionattrs, which marks different operands of a function as read-only (constant)*. Some correlations are harder to explain, for example, *number of BitCast instructions* and *-instcombine*, which combines instructions into fewer simpler instructions. This is actually a result of *-instcombine* reducing the loads and stores that call bitcast instructions for casting pointer types. Another example is *number of memory instructions* and *-sink*, where *-sink* basically moves memory instructions into successor blocks and delays the execution of memory until needed. Intuitively, whether to apply *-sink* should be dependent on whether there is any memory instruction in the program. Our last example to show is *number of occurrences of constant 0* and *-deadargelim*, where *-deadargelim* helped eliminate dead/unused constant zero arguments.

Overall, we observe that all the passes are correlated to some features and are able to affect the final circuit performance. We also observe that multiple features are not effective at directing decisions and training with them could increase the variance that would result in lower prediction accuracy of our results. For example, the total number of instructions did not give a direct indication of whether applying a pass would be helpful or not. This is because sometimes more instructions could improve the performance (for example, due to loop unrolling) and eliminating unnecessary code could also improve the performance. In addition, the importance of features varies among different benchmarks depending on the tasks they perform.

4.2 Importance of Previously Applied Passes

Figure 6 illustrates the impact of previously applied passes on the new pass to apply. From this figure, we learn that for the programs we trained on passes *-scalarrepl, -gvn, -scalarrepl-ssa, -loop-reduce, -loop-deletion, -reassociate, -loop-rotate, -partial-inliner, -early-cse, -adce, -instcombine, -simplifycfg, -dse, -loop-unroll, -mem2reg, and -sroa*, are more impactful on the performance compared to the rest of the passes regardless of their order in the trajectory. Point (23,23) has the highest importance in which implies that pass *-loop-rotate* is very helpful and should be included if not applied before. From examining thousands of the programs, we find that *-loop-rotate* indeed reduces the cycle count significantly. Interestingly, applying this pass twice is not harmful if the passes were given consecutively. However, giving this pass twice with some other passes

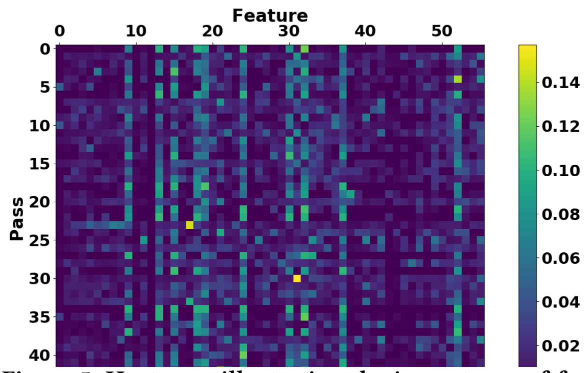

Figure 5: Heat map illustrating the importance of feature and pass indices.

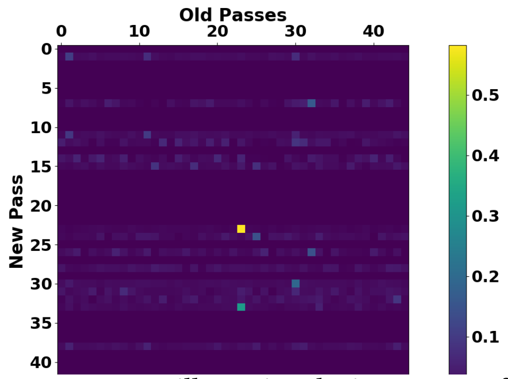

Figure 6: Heat map illustrating the importance of indices of previously applied passes and the new pass to apply.

between them is sometimes very harmful. Another interesting behavior our heat map captured is the fact that applying pass 33 (*-loop-unroll*) after (not necessarily consecutive) pass 23 (*-loop-rotate*) was much more useful compared to applying these two passes in the opposite order.

5 PROBLEM FORMULATION

5.1 Configuration 1: Single-Action

Assume the optimal number of passes to apply is N and there are K transform passes to select from in total, our search space $\mathcal{S}$ for the phase-ordering problem is $[0, K^N)$. Given M program features and the history of already applied passes, the goal of RL is to learn the next best optimization pass a to apply that minimizes the overall cycle count of the generated hardware circuit. Note that the optimization state s is partially observable in this case as the M program features cannot fully capture all the properties of a program.

Action Space – we define our action space $\mathcal{A}$ as $\{a \in \mathbb{Z} : a \in [0, K)\}$ where K is the total number of transform passes.

Observation Space – there are two types of features we can observe: ① program features $\mathbf{o_f} \in \mathbb{Z}^M$ listed in Table 2 and ② a histogram of previously applied passes $\mathbf{o_a} \in \mathbb{Z}^K$. After each RL step where the pass i is applied, we call the feature extractor in our environment to return new $\mathbf{o_f}$, and update the action histogram element o_{a_i} to $o_{a_i} + 1$.

Reward – the cycle count of the generated circuit is reported by the clock-cycle profiler at each RL iteration. Our reward is defined as $R = c_{prev} - c_{cur}$, where c_{prev} and c_{cur} represent the previous and the current cycle count of the generated circuit respectively.

Normalization Techniques In order for the trained RL agent to work on new programs, we need to properly normalize the program features and rewards so they represent a meaningful state among different programs. In this work, we experiment with two techniques: ① taking the logarithm of program features or rewards and, ② normalizing to a parameter from the original input program that roughly depicts the problem size. For technique ①, note that taking the logarithm of the program features not only reduces their magnitude, it also correlates them in a different manner in the neural network. Since, $w_1 \log(o_{f1}) + w_2 \log(o_{f2}) = log(o_{f1}^{w_1} o_{f2}^{w_2})$, the neural network is learning how the product of $\mathbf{o_f}$ affects the prediction instead of a linear combination of $\mathbf{o_f}$. For technique ②, we normalize the program features to the total number of instructions in the input program ($\mathbf{o_{f\_norm}} = \frac{\mathbf{o_f}}{o_{f51}}$), which is feature #51 in Table 2.

5.2 Configuration 2: Multiple-Update

An alternative of the formulating the actions above is to evaluate a complete sequence of passes with length N instead of a single action a at at each RL iteration. Upon the start of training a new episode, the RL agent resets all pass indices $\mathbf{p} \in \mathbb{Z}^N$ to the index value $\frac{K}{2}$. For pass p_i at index i, the next action to take is either to change to a new pass or not. By allowing positive and negative index update for each p, we reduced the total steps required to traverse all possible pass indices. The sub-action space a_i for each pass is thus defined as $[-1, 0, 1]$. The total action space $\mathcal{A}$ is defined as $[-1, 0, 1]^N$. At each step, the RL agent predicts the updates $[a_1, a_2, ..., a_N]$ to N passes, and the current optimization sequence $[p_1, p_2, ..., p_N]$ is updated to $[p_1 + a_1, p_2 + a_2, ..., p_N + a_N]$.

6 RESULTS

To run our RL algorithms we use RLlib [31], an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib is built on top of Ray [32], a high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications. We ran the framework on a four-core Intel i7-4765T CPUwith a Tesla K20c GPUfor handling the training and inference.

We use the number of clock cycles reported by the HLS profiler as the circuit performance metric. In [10], results showed a one-to-one correspondence between the clock cycle count and the actual hardware execution time. Therefore, better clock cycle count will lead to better hardware performance.

6.1 RL Performance

To evaluate the effectiveness of various algorithms for tackling the phase-ordering problem, we run them on nine real HLS benchmarks and compare the results based on the final HLS

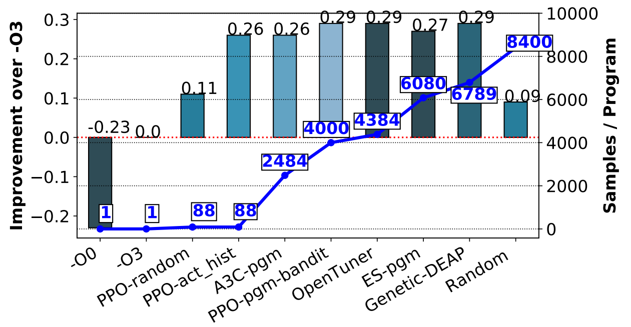

Figure 7: Circuit Speedup and Sample Size Comparison.

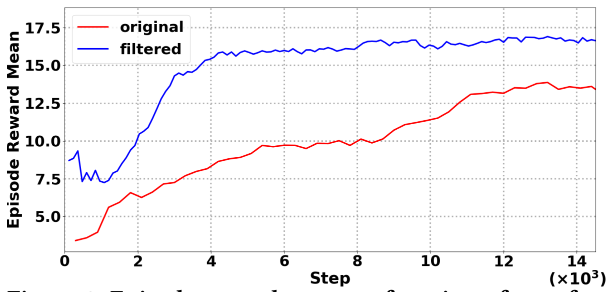

Figure 8: Episode reward mean as function of step for the original approach where we use all the program features and passes and for the filtered approach where we filter the passes and features. Higher values indicate faster circuit speed.

circuit performance and the sample efficiency. These benchmarks are adapted from CHStone [12] and LegUp examples. They are: *adpcm*, *aes*, *blowfish*, *dhrystone*, *gsm*, *matmul*, *mpeg2*, *qsort*, and *sha*. For this evaluation, we set the pass length to 45 and run each algorithm on per program basis. The bar chart in Figure 7 shows the percentage improvement of the circuit performance compared to -O3 results on the nine real benchmarks. The dots on the blue line in Figure 7 show the total number of samples for each algorithm, which is the number of times the algorithm calls the simulator to gather the cycle count.

`-O0` and `-O3` are the default compiler optimization levels. `PPO_random` is the PPO explorer that randomly samples the next pass to apply. `PPO_act_hist` is the PPO agent that learns the next pass based on a histogram of applied passes. `A3C_pgm` is the A3C agent that learns based on the program features. `PPO_pgm_bandit` uses a PPO agent and the program features but with the formulation in Configuration 2 5.2 explained in the above section. `ES_pgm` is similar to A3C agent that learns based on the program features, but updates the policy network using the evolution strategy instead of backpropagation. `DEAP`[33] is a genetic algorithm implementation. `OpenTuner` runs an ensemble of six algorithms which include two family of algorithms, particle swarm optimization [34] and GA, each with three different crossover settings.

`PPO_act_hist` achieves higher performance than `PPO_random`, which shows that the RL captures useful information during training. Using the histogram of applied passes results in better sample efficiency, but using the program features with more samples results in slightly higher speedup. `PPO_act_hist`, for example, at the minor cost of 2.3% lower speedup, achieves 50× more sample efficiency than `OpenTuner`. Using ES to update the policy is supposed to be more sample efficient for problems with sparse rewards like ours, however, our experiments did not benefit from that. Furthermore, using Configuration 2 with multiple action updates achieves slightly better than using Configuration 1 with a single actions.

6.2 RL Generalization

Training on completely different programs and benchmarks and then using the trained network to inference on any existing program is not practical. Yet, we believe programs should still benefit from prior knowledge on other completely different programs. This knowledge should be transferable from one program to another. For example, as discussed in section 4 applying pass *-loop-rotate* is always helpful, and *-loop-unroll* should be applied after *-loop-rotate*. Note that OpenTuner, GA, and greedy algorithms cannot generalize, as they do not learn any patterns from the programs and could be viewed as a smart random search. Therefore, these algorithms will always perform a completely new search for every program tested.

To evaluate how generalizable RL could be with different programs and whether any prior knowledge could be useful, we train on 100 different randomly generated programs using PPO. We train a network with 256 × 256 fully connected layers and use the histogram of previously applied passes concatenated to the program features as the inputs and passes as actions. We normalized the program features to the total number of instructions. In each pass sequence the intermediate reward was defined as the logarithm of the improvement in cycle count after applying each pass. Two approaches were evaluated: ① using all program features and passes and ② using the data we analyzed from section 4 where we keep only the impactful features and passes. Filtering the features and passes might not be ideal, especially when different programs have different feature characteristics and impactful passes. However, reducing the number of features and passes will help reduce variance among all programs and significantly decrease the search scope that will make the learning process easier and faster.

Figure 8 shows the the episode reward mean as a function of the step for the two approaches. We observe that the filtered approach learns much faster and achieves much higher overall average reward than the old approach with all the features and passes. At roughly 4000 steps (10 minutes of training) the filtered approach already achieves a very high episode reward mean, with minor improvements in later steps. Furthermore, the average reward of the filtered approach was still higher than the old approach even when we allowed the old network to train 20 times more. This indicates that filtering the features and passes significantly improved the learning process.

Both networks learned to always apply pass *-loop-rotate*, and *-loop-unroll* after *-loop-rotate*. Another interesting behavior the networks learned is to apply *-loop-simplify*, which performs several transformations to transform natural loops into a simpler form, which enables subsequent analyses and transformations.This pass will clean up blocks which are split out, but end up being unnecessary, so usage of this pass should not deteriorate generated code.

We later use both networks to inference (rollout) the CHStone benchmarks and LegUp examples as well as the 12874 randomly generated programs. Interestingly, the network with all the features and passes delivered 7% worse results than -O3 while the filtered one delivered 6% better results than -O3 with only 5% of the programs performing slightly worse (less than 1%) than -O3. While in general having more passes and features should give better performance, filtering the features and passes helped reduce the variance and enabled better learning of data that could be transferred to other programs more efficiently.

7 CONCLUSIONS

In this paper, we propose an approach based on deep RL to improve the performance of HLS designs by optimizing the order in which the compiler applies optimization phases. We use random forests to analyze the relationship between program features and optimization passes. We then leverage this relationship to reduce the search space by identifying the most likely optimization phases to improve the performance, given the program features. Our RL based approach achieves 29% better performance than compiling with the -O3 flag after training for just a few minutes, and a 26% improvement after training for less than a minute. Furthermore, we show that unlike prior work, our solution generalizes to a variety of programs, after training for less than ten minutes. While in this paper we have applied deep RL to HLS, we believe that the same approach can be successfully applied to software compilation and optimization. Going forward, we envision using deep RL techniques to optimize a wide range of programs and systems.

REFERENCES

[1] Xilinx, "Vivado Design Suite User Guide - High-Level Synthesis," June 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug902-vivado-high-level-synthesis.pdf

[2] Intel, "Intel FPGA SDK for OpenCL ." [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/developer-zone.html

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[4] S. S. Muchnick, "Advanced compiler design and implementation." Morgan Kaufmann, 1997.

[5] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning.* MIT press Cambridge, 1998, vol. 135.

[6] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization.* IEEE Computer Society, 2003, pp. 204–215.

[7] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 231–239, 2004.

[8] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2006, pp. 319–332.

[9] 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.* ACM, 2014, pp. 303–316.

[10] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis for fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on.* IEEE, 2013, pp. 89–96.

[11] 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 Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 3, p. 14, 2015.

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

[13] 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 *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2006, pp. 295–305.

[14] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03, 2003.

[15] 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.

[16] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, 2012.

[17] Z. Wang and M. OBoyle, "Machine learning in compiler optimization," vol. 106, no. 11, Nov 2018, pp. 1879–1901.

[18] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," vol. 4, 1996, pp. 237–285.

[19] R. Bellman, "A markovian decision process," in *Journal of Mathematics and Mechanics*, 1957, pp. 679–684.

[20] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[22] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 627–635.

[23] C. J. Watkins and P. Dayan, "Q-learning," vol. 8, no. 3-4. Springer, 1992, pp. 279–292.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[25] D. E. Goldberg, *Genetic algorithms.* Pearson Education India, 2006.

[26] E. Conti, V. Madhavan, F. P. Such, J. Lehman, K. Stanley, and J. Clune, "Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents," in *Advances in Neural Information Processing Systems*, 2018, pp. 5032–5043.

[27] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[28] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[29] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[30] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[31] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," *arXiv preprint arXiv:1712.09381*, 2017.

[32] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 561–577.

[33] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[34] J. Kennedy, "Particle swarm optimization," *Encyclopedia of machine learning*, pp. 760–766, 2010.