AutoAnnotate: Reinforcement Learning based Code Annotation for High Level Synthesis

Hafsah Shahzad*‡, Ahmed Sanaullah†, Sanjay Arora†, Uli Drepper†, and Martin Herbordt*§
*ECE Dept., Boston University †Red Hat, Inc. ‡Email: {hshahzad,herbordt}@bu.edu

***Abstract*—High Level Synthesis (HLS) allows custom hardware generation using developer-friendly programming languages. Often, however, the HLS compiler is unable to output high quality results. One approach is to pre-process the source code, e.g., to restructure the computational flow, or to insert compiler hints using annotations or pragmas. But while the latter approach appears to enhance programmability, it also requires developer expertise, both regarding hardware design patterns and even compiler internals: an incorrect annotation strategy can worsen performance or result in compilation deadlocks.**

To address these challenges, this work presents AutoAnnotate, an automatic code annotation framework for HLS. It demonstrates the efficacy, novelty, and benefit of applying ML methods to code annotation. AutoAnnotate replaces the need for developer expertise by using Reinforcement Learning (RL) to determine the best set of annotations for a given input code. To demonstrate the effectiveness of this approach, we ran AutoAnnotate on a number of common FPGA benchmarks derived, e.g., from Rodinia and OpenDwarfs, with state-of-art HLS tools (AMD Vitis and Intel HLS). We obtained a geometric mean of 42× performance improvement for Vitis HLS and 3.42× for Intel HLS. We then hand optimized these codes using standard best practices and again applied AutoAnnotate, this time still achieving 32.3× performance improvement for Vitis HLS and 3.1× for Intel HLS. Interestingly, the best overall performance obtained by AutoAnnotate was generally with unoptimized codes.

I. Introduction

A fundamental challenge in Electronic Design Automation (EDA) is the creation of code that is not only programmable with reasonable effort, but that is also performant. Basic to creating high performance FPGA applications is that they are usually programmed by developers experienced in that domain; moreover, programs for spatial accelerators such as FPGAs often do not follow the optimization principles of a traditional software design. High Level Synthesis (HLS) tools that enable transformation of High-Level Language (HLL) code into an FPGA specific design have the potential to offer a considerable advantage by enabling complex hardware designs using procedural languages. Among the vast number of academic and commercial products in this and related spaces are Electronic System Level (ESL) design tools [1]–[4], runtime libraries [5], autotuners [6], [7], and other program development infrastructure [8].

It is often the case, however, that the HLS compiler is unable to output high quality results. Many studies have tackled this problem using pre-processing of the source-code, e.g., [9]–[11]. For example, FPGA vendors suggest source code reconstruction focusing on optimization for pipelined registers, predictions, and memory coalescing, among others. However, improving source code by using manual code rewriting also requires significant expertise in FPGA architecture, including programming for distributed memory resources, deep pipelines, and data-flow routing. Also, this approach is compiler agnostic; while this seems a positive, it turns out to be the opposite: we find that some practices that yield benefits with Intel HLS can result in the opposite with AMD Vitis.

An alternative, source code annotation with pragmas (or directives), aims to make apparent to the compiler certain opportunities in the code, such as potential for parallelism or reducing memory latency. Most existing HLS tools employ user directives to transform code. However, these require that a naive code be properly annotated with *specific combinations* of pragmas in order to improve performance [12]. There are several limitations: Which pragmas should be used to exploit inter- and intra-module parallelism and memory abstractions? And, Which combinations of these pragmas work well together and guarantee maximum performance? In particular, specifying HLS directives and pragmas presents several challenges:

1. Transforming source code requires not only knowledge of hardware micro-architecture, but also familiarity with the proper use of tool-specific optimization directives and pragmas. It also requires facility with the overall coding style, e.g., applying appropriate memory partitioning, which itself is dauntingly dependent on code complexity. In fact attempting to use pragmas may actually exacerbate the programmability problem as the designer now needs expertise with the HLS tool as well as with hardware design. Moreover, the effect of inserting a pragma depends not only on the HLS tool, but also its version and the target FPGA type.

2. Pragmas may need to be added at tool-dependent locations in the code. For example, the loop unrolling pragma for Vitis HLS must be inserted after the *for* loop; this is in contrast to other compilers such as Intel HLS, GCC, Clang, and OpenMP where loop unrolling must be declared before the loop.

3. Even if the pragmas are inserted correctly, good performance is still not guaranteed. We have observed that incorrect (yet legal) insertion of certain pragmas can result in multiple factors worse performance. Infeasible configurations may increase latency costs, cause deadlock or result in runtime errors. For example, for an array used in a *for* loop: if the array partitioning factor is less than the loop unrolling factor, then the loop may not be unrolled successfully because the visits to the array are limited by the partitioning factor [3]. If the array partitioning factor is greater than the loop unrolling factor, more memory resources are consumed without increasing the parallelism. Compatible directives and factors are needed [13].

These challenges indicate that code rewriting with HLS

directives is a hard problem and that there is a need for an intelligent automated toolflow to productively annotate code in a reasonable amount of time. While the objective is still to leverage parallelism and optimize resource usage, our approach is ***to extract maximum performance benefit without any additional programming effort, while using state-of-the-art tools to guarantee universal availability***.

This paper presents AutoAnnotate, an automatic code annotation framework for HLS. AutoAnnotate reduces the need for developer expertise and effort by using Reinforcement Learning (RL), a ML approach that has proven valuable and superior to human experts in many fields [14]–[18]. We find that an RL agent can efficiently capture code characteristics and the HLS tool annotations that work well together, and thus to enable learning a model that predicts improved code insertions for a given application. AutoAnnotate adds value for both less and more experienced developers: the former in developing performant FPGA codes; the latter for generating design alternatives without having to go through hundreds of pages of reference manuals for each target tool.

This paper makes the following contributions:

- An end-to-end RL-based source code annotation framework that takes C code as input and outputs a performant, pragma-injected C code. Automating insertion of compiler directives/pragmas improves not only usability and portability but also the quality of customized programs on FPGA platforms.
- An extensible method for design space identification involving source code profiling. This outputs code-specific pragmas and their insertion points. The RL agent then learns to annotate the C code with the minimum number of pragmas from the design space, minimizing the overall execution cycles within a set number of iterations.
- We demonstrate the framework on common FPGA workloads that have been analyzed in prior work [10] and on state-of-the-art HLS tools. Our results show that using AutoAnnotate on these baseline codes gives an average of 42× performance improvement for AMD Vitis HLS and 3.42× for Intel HLS.
- We recognize that performance improvements over baseline HLL code can be difficult to interpret. We have therefore hand-optimized these codes using a procedure based on best practices [10] and again applied AutoAnnotate, this time still achieving a 32.3× performance improvement for AMD Vitis HLS and 3.1× for Intel HLS.
- We obtain the unexpected result that, of all of the combinations, the best performance was generally from running AutoAnnotate on the original baseline, rather than either code that has only been hand-optimized or hand-optimized code further optimized with AutoAnnotate.

The rest of this paper is organized as follows. Section II discusses some of the closely related work. Section III gives the motivation and background for using RL to annotate codes for HLS. Section IV presents the proposed framework, AutoAnnotate. Section V lists the evaluation methods. Section VI evaluates the effectiveness of the approach using mainstream FPGA benchmarks. Section VII gives the conclusion.

II. Relevant Work

Pragmas are inserted into source codes to give appropriate hints to the compiler and guide its process of performance optimization. Source code annotation for performance enhancement has been done several times; however, only a handful of publications have actually looked at the pragma insertion problem for HLS tools. *To the best of our knowledge, no one has yet used Reinforcement Learning to annotate generic C codes with pragmas for HLS workflows on any state-of-art HLS tool.* Our work is also the first in this category to verify that all transformed C codes indeed generate functionally correct codes.

In other work, Amiri et al. [19] explored code annotations with regards to data-flow programming in streaming applications that have multiple kernels within a single code. Similarly, the source-to-source compiler, SpecHLS [20] focuses on HLS kernels that can benefit from speculative pipelining. Design Space Exploration for HLS [13], [21]–[23] focuses on heuristics-based approaches to sample the design space. Other work identifies regions of interest, or predicts optimal factors, but for limited directives such as loop unrolling or loop pipelining. Moreover, most of the work in this domain uses performance estimation instead of actual synthesis with an HLS tool. Since these quality-of-results (QoR) estimators are not the actual HLS tool, their performance values might differ significantly from the correct results. Also, they may not have proper verification of generated codes through regression testing. Some work that does invoke downstream HLS tools has run-times of days. And, to the best of our knowledge, few of these use state-of-art tools for performance estimation. [24] presents an HLS optimizer for design space exploration within Merlin compiler. Others, such as ScaleHLS [25], use their own QoR estimator to provide performance estimates.

III. Background

In this section we look at the programmability model for traditional HLS tools and the available "knobs" for users to pre-process their codes using directives/pragmas. We find that the complexity motivates the RL assisted framework. We then discuss an alternate pre-processing approach, namely, source code rewriting, and show that it can be combined with RL. Our overall framework is presented in Section IV.

A. Programmability model for HLS tools

While HLS tools have raised the abstraction level for programming FPGAs, extracting performance from the input code is still a challenge. HLL codes are traditionally sequential, and typically written by developers with little knowledge of FPGAs' essential characteristics. Most HLS tools improve performance at their back-end by leveraging data-level parallelism using dependency analysis. However, much parallelism is still hard to predict. For this, tools such as AMD Vitis, Intel HLS, and Microchip's SmartHLS [3], [4], [26] allow users to insert annotations to highlight explicit parallelism, pipelining opportunities, or optimal memory interfacing.

The task of pragma insertion is notoriously demanding.

Tables I and II list some available pragmas for HLS tools that have been explored in this work. Since applications investigated are single kernels, the scope of this study does not include task level pipelining and structure packing using pragmas such as pragma HLS dataflow (along the lines of work presented in [19] for multiple kernels).

In HLS the constructs that have the highest impact on the final RTL micro-architecture are *functions*, *loops*, and *arrays*. Typically, an HLS tool first converts each function into a specific hardware component. For a given code, its ports and *functions* can be either be inlined or not. Inlining is especially useful for functions that are small and rarely called. These can be inlined into a larger caller function, allowing operations within the inlined function to be shared and optimized effectively. However, inlining can also worsen performance, specifically when the inlined function needs to be called multiple times within the parent function [3].

Next, *Loops* can be unrolled completely or only partially. Unrolling introduces hardware duplication. The amount of unrolling, however, is constrained by the memory bandwidth. *Loops* can also be pipelined with different initiation intervals. But an inter-iteration loop carried dependency will cause loop pipelining to fail.

Finally, *Arrays* can be mapped to registers or memories of different types. This is critical since interface contention—i.e., a RAM that allows fewer reads/writes than accesses in the same iteration—will cause a bottleneck. Deciding on the optimal number of concurrent ports for memory accesses is important so that if multiple arrays are feeding data to a compute task, these arrays can be mapped to different interface ports so as to access the data in parallel. Also, the data must be stored in different memory banks or the accesses will become serialized. Declaring appropriate interface pragmas allows users to resolve such hardware contentions.

TABLE I
AMD VITIS HLS PRAGMAS EXPLORED IN THIS WORK [3]

Type	Attribute	Additional options
Function Inlining	pragma HLS inline	off, on
Interface Synthesis	HLS interface	mode= ap_fifo ,s_axilite, m_axi; port; bundle; offset; depth
Pipeline	HLS Pipeline	rewind; enable_flush; II
Loop Unrolling	HLS unroll	factor
Array Optimization	HLS array_partition	variable; dim; factor type=cyclic,complete,block;

TABLE II
INTEL HLS PRAGMAS EXPLORED IN THIS WORK [26]

Type	Attribute	Additional options
Pointer Interface	var_type*	restrict, volatile
Host Interfaces	ihc::mm_host< var >&var	Data Width, Alignment, Address Width, Burst Width, Latency Stable=1,0
Agent Memories	registers,memories	configurations= numbanks writeonly, singlepump, doublepump, bankwidth, volatile
Constant Interfaces	stable_argument	avalon_agent_ register_argument
Loop Unrolling	unroll	factor
Max Concurrency	max_concurrency	1,0
Max Interleaving	max_interleaving	1,0
Other Loop Optimizations	#pragma ivdep loop_coalesce, ii disable_loop_pipelining	factor
Global Optimizations	clang fp contract	fast,on,off
Other Global Optimizations	component_pipelining max_concurrency use_stall_enable_clusters	

For designs with large numbers of loops, arrays, and functions, it is impossible to systematically explore all of the different design-space combinations. To come close to an optimal solution, an expert who can adapt rapidly is required. In this work, we have leveraged the particular capabilities of AI so that the expertise and adaptability come from reinforcement learning (RL).

B. Best Practices Optimizations of Baseline Codes

In prior work using source code transformation [9]–[11], researchers benchmarked common FPGA workflows by exploiting best practices of manual source code reconstruction. These best practices come mainly from (i) vendor-specific best practices HLS tool reference manuals, (ii) universally applied code optimization strategies such as use of loop unrolling and loop fusion; and (iii) FPGA-specific good practices, such as allowing concurrent memory accesses, using memory bandwidth effectively, and efficient resource binding. While applying best practices is compiler-agnostic, it does benefit from programmer understanding of typical HLS compiler behavior and of FPGA architecture.

An observation of note to this study is that, in some cases, *source code reconstruction offers benefits not possible with annotations alone*. This observation illustrates limitations of current HLS compilers: not all performance can be extracted using just compiler hints; in some cases programmer involvement is still required. Some examples include:

- reordering of loops for spatiality and for exploiting input caching;
- loop tiling to remove loop carried dependencies;
- rewriting code to remove overlapping array accesses;
- fusing multiple loops to achieve runtime reduction;
- extracting task level parallelism by implementing independent tasks, such as kernels, and connecting them using channels;
- restructuring code to remove conditional accesses and if-else conditions that can create bottlenecks for parallel execution; and
- since some HLS compilers cannot estimate latency if a while loop is present, converting them into for loops.

Accordingly, evaluations of HLS enhancements, such as those proposed in this study, should have multiple baselines: the original code, but also code to which best practices have been applied. The latter is essential since it represents versions of the code that are in states that are (apparently) not accessible to these HLS compilers. As a result our evaluations are with respect to both baseline code (version 1) and code that has undergone programmer-based reconstruction as in Table III. As we see in Section VI, however, it turns out that AutoAnnotate on *unoptimized* code most often gives the best results.

Figure 1 shows code optimizations performed by AutoAnnotate on matrix multiply. When the baseline code is run through AutoAnnotate, it outputs annotated code that improves the latency by 7.3×. The baseline code is also hand optimized

TABLE III
SUMMARY OF CODE VERSIONS AND OPTIMIZATIONS APPLIED

Version	Optimization
1	CPU-like single kernel C code
2	Implement task parallel computations in separate kernels connected using channels
	Apply optimizations such as loop unrolling, minimizing variable declaration using temps
	Using constants for problem sizes and data values to reduce off-chip memory access
	Coalesce memory operations
3	Implement all optimizations proposed for version 2 within a single kernel instead
4	Reduce array sizes to infer pipeline registers as registers, instead of BRAMs
5	Perform computations in detail,using temporary variables to store intermediate results
6	Use predication instead of conditional branch statements

(as outlined in [10]); these results are summarized in Table III. We find the version (#5) that gives the best performance; it is the same for both AMD HLS and Intel HLS. Results are given in Figures 3; to summarize, there is a latency improvement of 2.7× versus the baseline code. We then run the best hand optimized code through AutoAnnotate and the annotated code improves the hand optimized code by 1.98×. In this case, the combined benefit of hand-optimization plus AutoAnnotate is a speedup of 5.34× over the original baseline.

IV. THE AUTOANNOTATE FRAMEWORK

A. AutoAnnotate Design

The proposed framework for automated source code annotation for HLS is illustrated in Figure 2. The C code is fed into the framework together with HLS tool specific *pool* of pragmas. Tables I and II list the pragmas used in AutoAnnotate and their various configuration options. Most of these pragmas feature placeholders that must be correctly labeled with variable names, port designations, bundle names, and offsets. *Pragma-Generator* addresses this by accurately labeling placeholders based on variables extracted by the *Code-Profiler*. The *labeled-pool-of-pragmas* is passed as *action space* to the RL framework. The RL agent selects pragmas and the *Code-Annotator* places them at designated line numbers. The annotated code undergoes latency evaluation, enabling the RL agent to learn which pragmas yield the best rewards over time. Pragmas resulting in minimal latency after RL training are integrated back into the original C code to output the best-annotated version, subject to validation through co-simulation.

B. Code Profiler

Input code is first fed into a code profiler that outputs code characteristics that are required for HLS annotation. These include information about (i) various functions in the code and their declaration points; (ii) variables in each function and their names and attributes (e.g., pointer, read-only); and (iii) loops in each function, their declaration points and nesting levels.

C. Pragma Generator

The HLS tool-specific configurations are fed separately into the toolflow. Each configuration contains a directory of HLS tool-specific information such as (i) type/version of HLS tool,

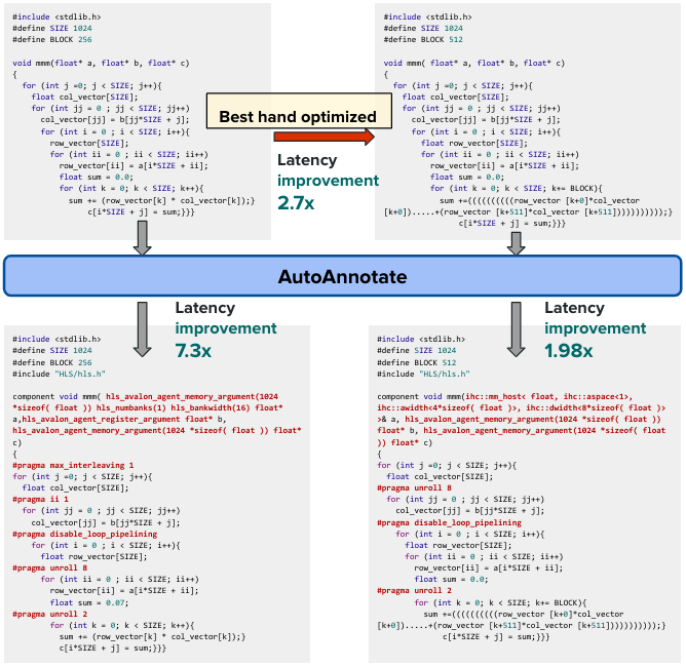

Fig. 1. Example output from AutoAnnotate for Intel HLS: MatrixMultiply baseline code can be hand optimized using version 5 in Table III to increase performance by 2.7x. AutoAnnotate inserts pragmas into each of the baseline and hand optimized versions to increase performance by 7.3x and 5.34x with respect to the baseline code.

(ii) built-ins that are supported after function declaration, and (iii) pragma optimizing loops. Pragmas supported in this work are given in Tables I and II. Each of these pragmas is categorised into a pool of function and loop pragmas. Function pragmas are inserted either after function declaration (AMD HLS) or within the function arguments (Intel HLS). Similarly, loop pragmas are inserted either after for loop (AMD HLS) or before it (Intel HLS). This information, together with the code profiler output, is fed into a Pragma Generator.

The HLS Tool Configuration Database (see Figure 2) is the same regardless of the source code and depends on the HLS tool and its version. This is like a registry of possible pragmas supported by the tool. Most of these pragmas have placeholders that need to be annotated with appropriate variables from the function and other configuration values. The Pragma Generator labels the placeholders within each pragma with all possible variable names and configuration options and outputs a list of labelled pragmas and their insertion points for the specific source code. This list of (pragma, insertion point) is fed as an action space to the RL framework.

D. RL based Environment

Reinforcement learning (RL) is an ML approach where an agent learns by continually interacting with its environment. We propose that an RL agent can learn code characteristics and HLS tool annotations that work well together and so predict the best code insertions for a given application. In contrast to supervised ML, the RL agent can be tuned to optimize multiple objectives with large search spaces and does not need pre-labelled data for training. An RL framework has two major components: i) the *environment*, or the problem to be solved, and ii) the *agent*, which is used to perturb the environment and

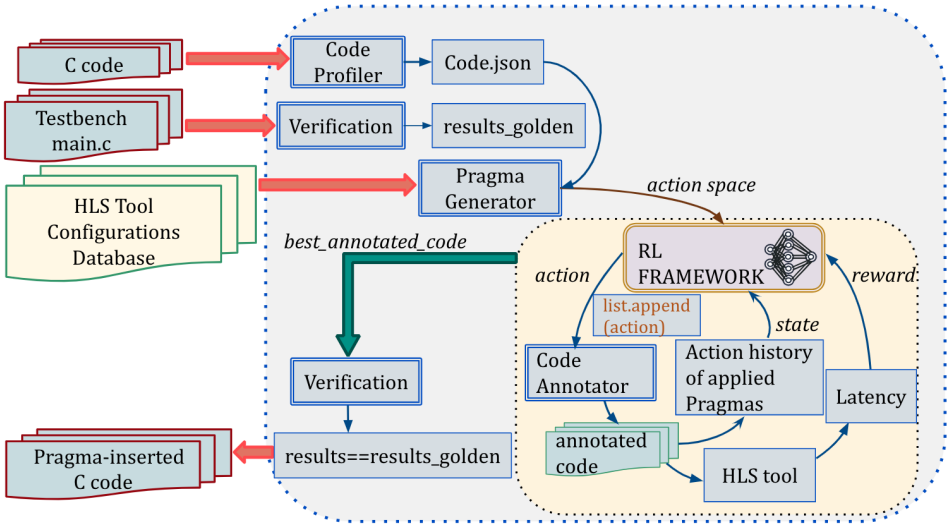

Fig. 2. AutoAnnotate: Proposed framework for automatic HLS annotations using RL

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. After taking the action, the environment returns a *reward*, indicating the impact of the action, and an updated *state*, which represents the change in the environment resulting from taking the action. The next step then begins and a new action is predicted.

The above process continues until the environment indicates that a conclusion has been reached. This collection of steps, from the first predicted 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. At the end of each iteration, the learning portion of the RL framework updates a *policy* (a deterministic or stochastic strategy) about which actions cause agents to maximize their long-term, cumulative rewards. RL assumes that the environment is Markovian, i.e., that the updated state depends only on the previous state and the action taken. It also assumes that the action taken is only dependent on the current state.

The RL framework for AutoAnnotate is set up as follows:

- Environment: The environment is composed of the HLS tool and the specific application. It outputs the number of clock cycles needed to execute the target application.
- Agent: Proximal Policy Optimization (PPO) is used as the reinforcement learning agent [27]. It is stable, easy-to-use, and gives good optimization decisions for source code annotations. Based on its good performance, it is also the default policy at OpenAI [28].
- Action: Each action represents a single annotation decision. This could be either one of the annotations generated by the pragma generator or a null action specifying that the agent do nothing. These decisions are appended to an ordered list of actions for the episode.
- Code Annotator: The list of actions suggested by the PPO agent is passed to a code annotator after every episode. This includes the pragma and its insertion point in the C code. The code annotator inserts all the actions recommended by the RL agent into appropriate locations in the source code and outputs an annotated code.
- State: In RL, a state represents the current environment that the agent is in. In this case, a memory-based state that stores information about previous actions taken within an episode is configured as the state. This is important since it allows the agent to plan its next actions based on the current situation and goals. This is referred to as an *action history* and is a list equal to the length of the action space (output by the pragma generator). Each element of the list is incremented and updated when a pragma corresponding to that list element is applied. PPO learns to map the distribution of applied pragmas to the next pragma that should be applied while maximizing the cumulative rewards across time-steps in an episode. Since the possible action can also be a null pragma, the agent is simultaneously forced to learn both the optimal combination and the optimal number of pragmas to insert.
- Reward: The reward is defined as the difference in latency between the unoptimized, unannotated code and the current step; lower latency thus results in a higher reward value. The reward is set to a default value of 0 for each step, except for the last step in the episode in which the actual latency is obtained by running through the HLS tool. *Maximum Reward* is defined as the highest reward value obtained by any episode during training.

E. HLS Tool

In the current study we use both AMD Vitis HLS and Intel HLS. Moreover, adding support for any HLS compiler is straightforward. The annotated code after each episode is fed into the HLS tool and the post-synthesis reports generated are parsed by the toolflow to output the latency values. HLS tools typically try to generate scheduling algorithms based on the operating frequency. The frequency target and FPGA type can also be altered according to user preferences. While the goal in this work is reduction in latency, the system can also be set to optimize for area, or other measure of resource utilization, or for both latency and resource utilization. These are specified by reading the appropriate performance value from the tool

and setting it as the reward function for the RL agent.

V. Evaluation Methods

A. Benchmarks

The seven benchmark codes in [10] are used for evaluation in this work: Needleman Wunsch (NW), Fast Fourier Transform (FFT), Range Limited Molecular Dynamics (RL), Particle Mesh Ewald (PME), Matrix Multiplication (MMM), Sparse Matrix Dense Vector Multiplication (SpMV), and Cyclic Redundancy Check (CRC). These were derived from mainstream FPGA benchmarks including Rhodinia [29] and OpenDwarfs [30], [31], including an optimization in [32]. In the present work, we have first validated their approach by porting their C code to Vitis HLS and Intel HLS. Here, and at all stages of the evaluation, we have verified correct program execution.

To obtain the baseline, the applications are all transformed into CPU-like single kernel C code, the standard starting best practice (Version 1 in Table III). Because of the often orthogonal capabilities of programmer optimization versus code annotations (as described in Section III-B) we also apply AutoAnnotate to codes that were transformed using the methods in Table III.

B. Experimental Setup

The reinforcement learning framework is set up using Open AI Gym (0.21.0)'s environment interface, Ray (1.7.0) unified API, and Ray's RLlib library to provide the agent interface. Keras (2.6.0) is used to provide the neural net API for Python (3.9). Tensorflow (2.6.0) is used for machine learning. Each test is run using *a single worker* on a standard multi-core CPU, the same initial *seed* value for random number generation, and a total training time of *300 iterations*.

The HLS tools used are AMD Vitis 2021.1 and Intel HLS 23.3 [3], [26]. We set the target frequency to 300MHz. Artix-7 is used as the default FPGA for AMD Vitis HLS and Stratix 10 for Intel HLS. We believe that these methods for learning effective code annotations are general and applicable to different versions of these tools as well as to other HLS tools. Moreover, different FPGA types can also be chosen.

C. Applying Code Optimizations

Some details are as follows. First, Version 2 from Table III is omitted. It involves leveraging task-level parallelism using multiple kernels; the focus here is on single kernel optimization. Moreover, our premise is that the HLS compiler is still able to automatically infer task parallel pipelines [10]; also, having a single kernel offers several advantages such as reduced computation overhead and simple algorithmic flow [33].

Figure 3 show the results of individual optimizations proposed by [10] performed on top of the baseline code (Version 1) as evaluated using the AMD Vitis HLS and Intel HLS tools. In [10] the authors observed that each subsequent version of the source code restructuring typically increases performance. We find here that source code rewriting does not give much benefit at all for multiple benchmarks including FFT, RangeLimited, and CRC. For AMD HLS, the benefit is seen largely in the systolic-array-based Needleman Wunsch benchmark,

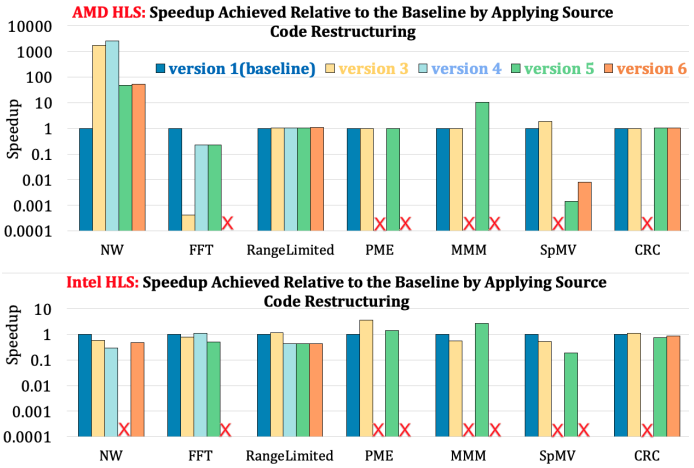

Fig. 3. Systematic optimizations performed in [10] are applied for AMD Vitis HLS and Intel HLS. 'X' indicates that either the version was not created since the corresponding optimizations did not exist for the specific benchmark or the tool run into a deadlock/compilation error.

where each version gives better performance than the baseline. Even here, however, Versions 4 and 5 unexpectedly give poor results when compared to the prior work. For Intel HLS, in the case of PME, Version 3 optimizations give best results. In the case of MMM, for both Intel and AMD, Version 5 gives best performance.

These findings highlights a crucial observation: *the benefits of programmer-applied optimizations, e.g., based on best practices, are highly compiler dependent*. For example, pipeline stages are inferred differently by both HLS tools. Performing computations in detail by storing intermediate results in pipeline registers has little to no impact for AMD HLS, but results in improvements in some cases for Intel HLS. This result is also heavily dependent on the application: dissolving branch statements into conditional assignments gives performance improvement for MMM; for other applications, however, there is little benefit.

VI. Results

A. AMD HLS

1) Speedup achieved using AutoAnnotate

Figure 4 demonstrates the effectiveness of AutoAnnotate when compared to source code restructuring proposed in [10]. We postulate that modern HLS compilers can do a decent job at optimizing C codes without additional source code restructuring and using HLS tool specific annotations only. The results in Figure 4 support this claim for AMD Vitis HLS. Two versions are compared with the baseline. In the first, the baseline code for each of the seven FPGA benchmarks is optimized using source code rewriting; the code version that gives the maximum speedup is selected. In the second, AutoAnnotate applied to the baseline code. The results show that the RL agent does a good job at annotating code to enhance performance. On average, AutoAnnotate improves the baseline codes by 42× versus source code restructuring that improves it by 4.8×. At worst, AutoAnnotate gives performance comparable to source code rewriting. This also answers one of the big questions in HLS: Is it possible to

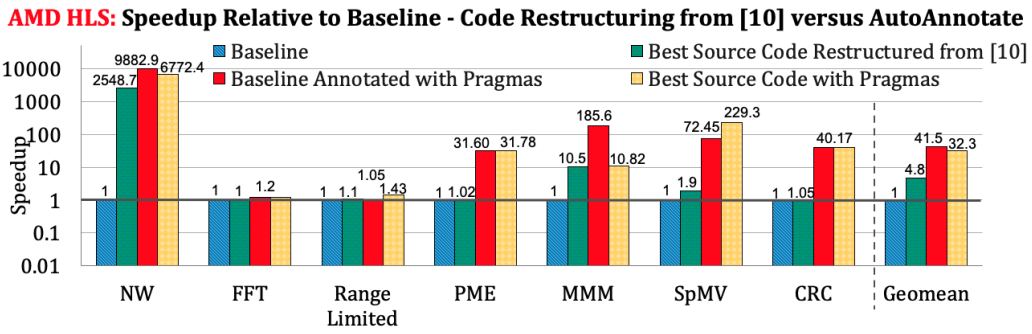

Fig. 4. The performance of proposed annotation framework, AutoAnnotate on baseline source codes compared to source code restructuring proposed in [10]

perform better than source code restructuring using compiler annotations? The answer, as backed by the results is Yes. For some applications such as NW, PME, MMM, SpMV and CRC this is by a large margin. For the balance, FFT and Range Limited, it is still possible to give slightly better (by 20%) or, at least, comparable performance (within 5%).

2) Do random annotations also improve performance?

As discussed in Section IV AutoAnnotate uses Proximal Policy Optimization (PPO) as the RL agent. PPO updates its existing policy at each *step* in order to minimize the cost function without deviating too much from the previous policy. Our goal here is to understand how much of the performance gain (in Figure 4) is because of RL (PPO in this case); to do this we compare with a baseline *random* strategy. For the random strategy, we replaced the PPO agent with simple nested loops that ran for an equivalent number of steps, episodes, and iterations, and each time picked random actions of length equal to the list of pragmas (as set up for PPO-RL toolflow as well).

The results are given in Figure 5. For five out of seven benchmarks, the random strategy could not get even a single random occurrence of annotations without compilation errors. This is because for Vitis HLS, only a small subset of pragmas/annotations can work well together. And since the random strategy does not learn over time, for every iteration it ran into errors. For CRC and PME it was possible to get legal annotations even with random strategy, perhaps because the action space of effective pragmas is so small that it is possible to pick the acceptable ones just by chance. In both cases, however, the resulting performance was indistinguishable from the original baseline code.

3) Combining code restructuring with AutoAnnotate

Figure 6 shows the results of combining source code rewriting (as proposed in [10]) with AutoAnnotate. For the established FPGA benchmarks, we choose different code versions as proposed in Table III as our starting points. Code structures A-D refer to versions 3-6. Our goal is to understand just how much performance potential is achievable when AutoAnnotate is applied to each code structure. The results show:

- Figure 3 gives the best source code version for each application, e.g., for NW, it is version 4. Applying annotations on top of version 4 (code structure B) gives a speedup of 2.67× as can be seen in Figure 6. For SpMV, we can improve the best-case hand optimized code (code structure A-version 2 from fig 3) using AutoAnnotate by 120×. On average, AutoAnnotate improves the best hand optimized source codes by around 32.3× with respect to the unoptimized baseline.
- Results in Figure 6 show that pre-processing using AutoAnnotate gives the maximum performance benefit when applied to baseline code without any source code

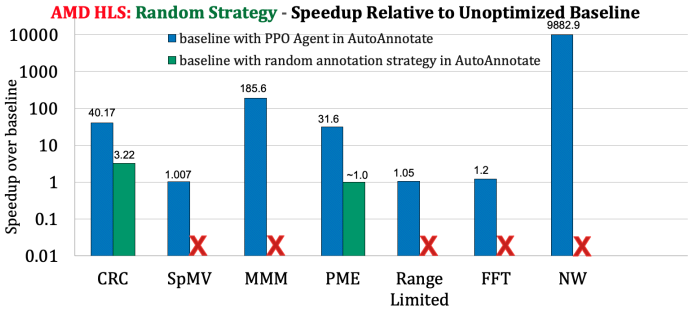

Fig. 5. Relative Speedup achieved by using base configuration (with PPO agent) in AutoAnnotate versus using a random strategy to annotate code for equivalent number of iterations. X is used to denote the case when not even a single episode of random annotations output compilable code; the proposed annotations were outputting compilable errors on Vitis HLS

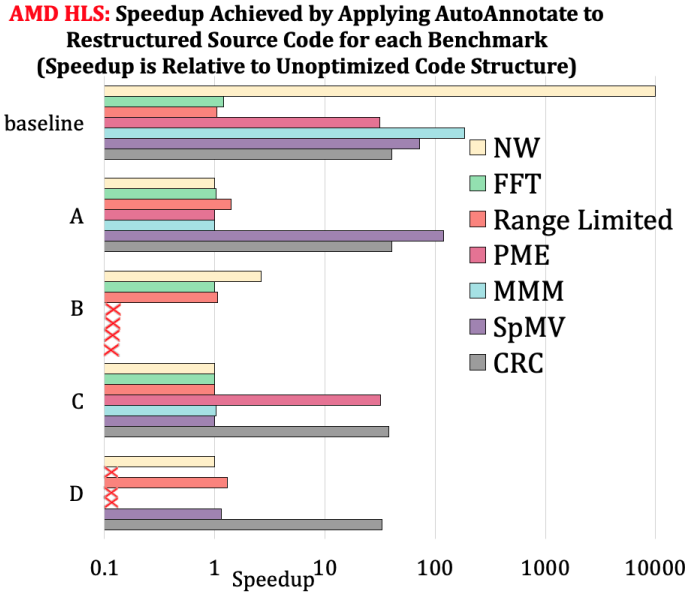

Fig. 6. Impact of applying AutoAnnotate to each starting code version. 'X' indicates that code structure was not created since the corresponding optimizations did not exist for the specific benchmark.The speedup >1 shows performance was enhanced by applying strategy of AutoAnnotate. Even for the worst case, performance is equal to the unoptimized code. Baseline is the same as Version 1. Code structure A refers to version 3, code structure B refers to version 4, code structure C refers to version 5, code structure D refers to version 6 from Table III.

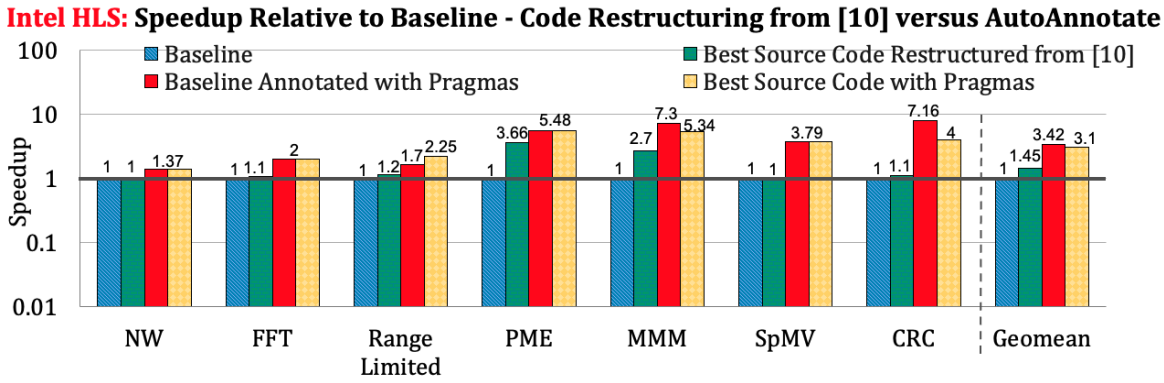

Fig. 7. Speedup achieved by using AutoAnnotate to insert Intel HLS pragmas for both baseline and best source code restructured versions

restructuring. Exceptions include Range Limited where AutoAnnotate applied to code structures A and D gives slightly better performance than when it is applied to baseline codes; and SpMV where AutoAnnotate applied to code structure A gives 3.16× better performance than when it is applied to the baseline code. AutoAnnotate improves baseline codes by 42×, code structure A by 3.54×, code structure B by 1.42×, code structure C by 2.77× and code structure D by 2.65×.

- From the results we can also conclude that the HLS compiler in general, does well at inferring pipeline stages and leveraging parallelism; further benefit can be achieved by applying tool-specific annotations only (something that AutoAnnotate does) without requiring to manually rewrite the code structure. This is especially evident in the case of CRC. AutoAnnotate applied to every code structure results in nearly identical performance improvement. It is also evident in PME where AutoAnnotate applied to the baseline and to code structure C give the same performance improvement.

B. Intel HLS

1) Combining code restructuring with AutoAnnotate

For Intel HLS we combined strategies evaluated in Sections VI-A1 and VI-A3. Hand optimized versions as proposed in Table III were first run on the tool. The results are given in Figure 3. Out of these, the maximum performance improvement possible using hand optimizations (source code restructuring) for each workload is selected and its performance advantage is displayed as green bar in Figure 7. Next, the baseline codes for each workload are annotated using AutoAnnotate and the performance improvement is given by the red bars. Lastly, the best source code restructured codes are run through AutoAnnotate and the performance improvement possible is given by the yellow bars. We note that in most cases, annotating the baseline codes gives performance improvement either better than, or at least comparable to annotating the best source code restructured codes. AutoAnnotate improves the performance of baseline codes by a geomean of 3.42× and best source code restructured by 3.1×.

VII. Conclusion and Future Directions

We have developed an automatic source code annotation framework that replaces developer expertise and effort with Reinforcement Learning. We found that there is a substantial benefit to using AutoAnnotate, not only in automating the challenging task of pragma insertion, but also in obtaining performance that is generally better than that of applying programmer-directed best practices. In fact, this performance benefit (of AutoAnnotate on the baseline code) extended even to the application of AutoAnnotate to the hand optimized code. This last observation needs to be studied further. It could be that these hand optimizations have somehow constrained the degrees of freedom available to AutoAnnotate.

Going into this study we conjectured that a PPO agent could give good performance. We found (see Sections V and VI) that the agent does indeed learn, not only which annotations work well together, but also the minimum set of annotations to give that performance. In the future we will explore replacing it with an even "smarter" agent.

Also in this work we passed the history of applied annotations as a state for the agent. This works since FPGA workloads are generally application-specific. This is also important since only a handful of pragmas work well in combination with each other for each application. Hence the RL agent learns the minimum number of annotations that work well for that specific application. The information about code features is captured during code profiling in order to generate code specific pragmas for the application case. In the future we can incorporate the code profiling block into the RL framework. We can encode the state of the RL framework so as to include code profiling information. This will drive the agent to learn not only the best annotations but also the best insertion points in the code.

Finally, in this work, we ran each training case for 300 iterations so as to limit the time in which we can output a pragma inserted C code. In the future, the agent could be trained to run over an optimal number of iterations until performant pragma inserted code has been achieved.

Acknowledgments

This work was supported, in part, by the NSF through award CCF-1919130; by a grant from Red Hat; and by AMD and Intel both through donated FPGAs, tools, and IP.

References

[1] 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].
[2] LegUp, "High-Level Synthesis Open Source Software," http://legup.eecg.utoronto.ca/ [Last accessed: Feb 28, 2023].
[3] 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].
[4] Microchip, "SmartHLS High-Level Synthesis Software," https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler [Last accessed: Feb 28, 2023].
[5] Xilinx, "Xilinx Runtime Library," https://www.xilinx.com/products/design-tools/vitis/xrt.html [Last accessed: Feb 22, 2023].
[6] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, "Lamda: Learning-assisted multi-stage autotuning for fpga design closure," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 74–77.
[7] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A parallel bandit-based approach for autotuning fpga compilation," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 157–166.
[8] Xilinx, "Xilinx Vitis Unified Software Platform," https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html [Last accessed: Feb 22, 2023].
[9] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs," in *Proc. Int. Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 409–420.
[10] A. Sanaullah, R. Patel, and M. Herbordt, "An empirically guided optimization framework for FPGA OpenCL," in *International Conference on Field-Programmable Technology*, 2018, pp. 46–53.
[11] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE TPDS*, vol. 32, no. 5, pp. 1014–1029, 2020.
[12] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS today: Successes, challenges, and opportunities," *ACM TRETS*, vol. 15, no. 4, pp. 1–42, 2022.
[13] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu, "Correlated multi-objective multi-fidelity optimization for hls directives design," *ACM TODAES*, vol. 27, no. 4, pp. 1–27, 2022.
[14] 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.
[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. 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.
[17] H. Shahzad, A. Sanaullah, S. Arora, R. Munafo, X. Yao, U. Drepper, and M. Herbordt, "Reinforcement learning strategies for compiler optimization in high level synthesis," in *IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC*, 2022, pp. 13–22.
[18] R. Munafo, H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt, "Improved Models for Policy-Agent Learning of Compiler Directives in HLS," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–8.
[19] P. Amiri, A. Pérard-Gayot, R. Membarth, P. Slusallek, R. Leißa, and S. Hack, "FLOWER: a comprehensive dataflow compiler for high-level synthesis," in *Int. Conf. on Field-Programmable Tech.*, 2021, pp. 1–9.
[20] J.-M. Gorius, S. Rokicki, and S. Derrien, "SpecHLS: speculative accelerator design using high-level synthesis," *IEEE Micro*, vol. 42, no. 5, pp. 99–107, 2022.
[21] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE TCAD*, vol. 39, no. 10, pp. 2628–2639, 2020.
[22] N. Wu, Y. Xie, and C. Hao, "IronMan-Pro: Multiobjective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network-Based Modeling," *IEEE TCAD*, vol. 42, no. 3, pp. 900–913, 2023.
[23] L. Ferretti, G. Ansaloni, and L. Pozzi, "Cluster-based heuristic for high level synthesis design space exploration," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 1, pp. 35–43, 2021.
[24] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Automated accelerator optimization aided by graph neural networks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 55–60. [Online]. Available: https://doi.org/10.1145/3489517.3530409
[25] H. Jun, H. Ye, H. Jeong, and D. Chen, "AutoScaleDSE: a scalable design space exploration engine for high-level synthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 3, pp. 1–30, 2023.
[26] Intel, "Intel® High Level Synthesis Compiler," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html [Last accessed: October, 2023], 2023.
[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347*, 2017.
[28] OpenAI, "Proximal Policy Optimization," https://openai.com/research/openai-baselines-ppo [Last accessed: October 31, 2023].
[29] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.
[30] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas, "Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," *Journal of Signal Processing Systems*, vol. 85, pp. 373–392, 2016.
[31] Berkley, "OpenDwarfs Benchmarks," https://github.com/vtsynergy/OpenDwarfs [Last accessed: Mar 27, 2023].
[32] 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 *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–8.
[33] Intel, "Intel® Rendering Framework Using Software-Defined Visualization," https://www.intel.com/content/dam/develop/public/us/en/documents/parallel-universe-issue-35.pdf [Last accessed: February 14, 2024].